Animating Motion

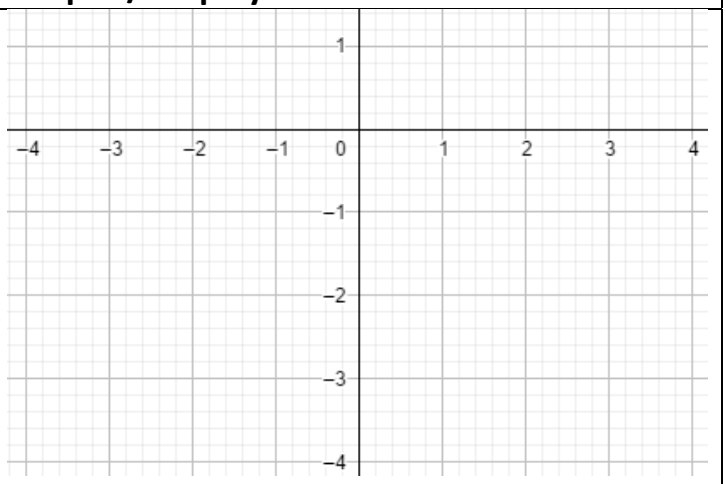
Motion, at least on a screen, is an illusion generated by drawing the same image, but in slightly different positions as time goes by. Our brains (sensibly enough) interpret a stick a metre from us one moment and only half a metre from us the next, not as two unrelated visual phenomena, but a stick moving towards us.

To simulate motion on a computer usually involves:

- ***Storing the current position of an object (eg as x and y coordinates)***
- ***Storing the velocity of an object (eg as vx and vy values)***
- ***Continually updating the values of x and y using vx and vy***

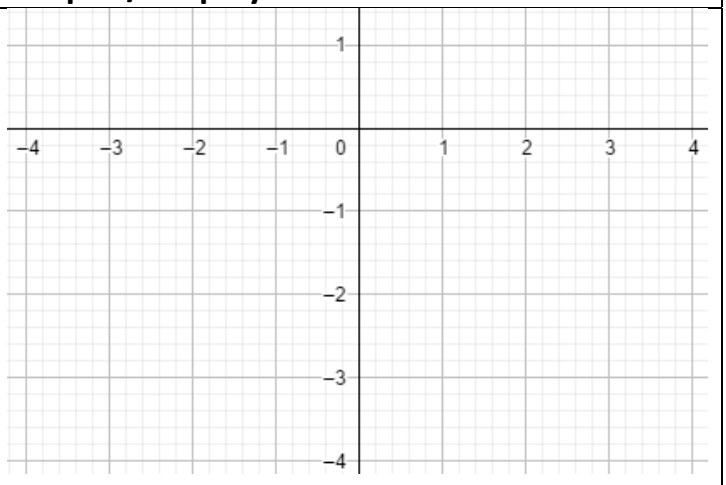
Example 1:

Work through the following example, marking the points on the grid when you do step 7:

Pseudo-code	**Output / Display**
1: Store $x = -3$ 2: Store $y = 1$ 3: Store $vx = 1$ 4: Store $vy = -2$ 5: Add vx to x 6: Add vy to y 7: **Mark a cross at (x, y)** 8: *Repeat steps 5 to 7 forever*	

Example 2:

In this example, we deal with the issue of hitting the bottom of the screen.

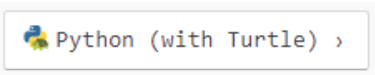
Pseudo-code	**Output / Display**
1: Store $x = -3$ 2: Store $y = 1$ 3: Store $vx = 1$ 4: Store $vy = -2$ 5: Add vx to x 6: Add vy to y 7: If $y < -4$, change the sign of vy 8: **Mark a cross at (x, y)** 9: *Repeat steps 5 to 8 forever*	

Did you notice what happened when line 7 kicked in? Can you think of a way to avoid going off the screen completely? Is there a way to adapt the condition? Can you see how we might factor in acceleration as well? How about $vx += ax$ before $x += vx$?

Turn over to see how to simulate this motion with Python

Animating Motion *with Python*

For this project, if you're using repl.it, you'll need to choose:



By using the Turtle program, we can cheat a little bit. Instead of having to display an image at a certain point, then clear the screen and display the image at a slightly different point, the Python Turtle can simulate motion all on its own when given a destination. We'll make use of this because it is the simplest way to display an object on the screen, but we will still be making use of the fundamental elements of object motion.

Start by setting up a turtle object that will do our bidding:

```
1   import turtle
2
3   t = turtle.Turtle()
4   t.shape("circle")
5   t.speed(0)
6
7   x = 0
8   y = 0
9   vx = 1
10  vy = -2
```

The first line loads the code within Python's '**turtle**' module. Next, I create a **turtle** object called **t**.
To give the impression of, say, a snooker ball, bouncing around the screen, I'll set the **shape** of the turtle to **"circle"**, and to ensure it moves quickly enough, set **speed(0)** (maximum possible speed at normal animation rates).

I've also set up the variables to use for position and velocity.

The next step is to build a loop which updates the x and y coordinates, then moves the turtle to the new position. An added bonus of doing this with turtle is that we get to see the path the turtle follows drawn in automatically. To disable it, use **t.penup()**.

```
11
12  while True:
13      x += vx
14      y += vy
15      t.goto(x, y)
```

This loop increases **x** by adding onto it the value of **vx**, and similarly with **y**. The **t.goto** command tells the turtle to move towards the point with the given coordinates, then stop.

Improved version:

```
1   import turtle
2
3   t = turtle.Turtle()
4   t.shape("circle")
5   t.speed(0)
6   t.tracer(10)
7   x = 0
8   y = 0
9   vx = 1
10  vy = -2
11
12  while True:
13      if x > 300 or x < -300:
14          vx *= -1
15      if y > 300 or y < -300:
16          vy *= -1
17      x += vx
18      y += vy
19      t.goto(x, y)
```

This modification to the loop code checks at each iteration to see if the turtle has moved beyond the bounds of the screen. If it has, the velocity in the relevant direction is multiplied by -1, effectively reversing it (this essentially makes the turtle bounce if it hits a wall).

I've also added in **t.tracer(10)** to speed up the overall animation. The **10** corresponds to the number of milliseconds between the screen being refreshed, so making it a larger number speeds things up because the computer isn't required to update the display so often.

Extending the code

Quick Customisations:
By using turtle functions like **t.penup()** you can hide the track being drawn.
There are also functions for setting the colour of either the turtle itself or the path it draws, as well as the width of the path drawn. You can use the **random** module to generate the colour you use as well:

```
r, g, b = random.randint(0, 255), random.randint(0, 255), random.randint(0, 255)
t.color(r, g, b)
t.pencolor(r, g, b)
t.width(2)
```

Randomization:
The original code set the initial velocity at the start, so the ball will always begin at the point $(0,0)$ and move with velocity $\begin{pmatrix} 1 \\ -2 \end{pmatrix}$. If you import the **random** module, you could use something like **random.randint(-200, 200)** to determine the initial position of the turtle (maybe lift the pen and move it there before you start the motion loop), and you could use **random.uniform(-3, 3)** to define the size and direction of **vx** and **vy**.

More balls:
If you set up another turtle at the beginning (call him **s**, let's say), you could apply the same code to both, and have both moving around the screen simultaneously. To do more than two or three, however, you'll find it helpful to learn about **classes** in Python (the sophisticated way of not writing the same code over and over again to create similar objects).

Different boundary conditions:
In its current form, the ball will bounce when it 'hits a wall' (in other words, when its **x** or **y** coordinate exceeds certain limits. You could define the limits differently, and be more creative with the effect of hitting a boundary. For instance, what if the turtle changed direction if **x**2 + y**2 > 200**2** (that is, when it reached the boundary of a 200px circle)?

Friction:
A good way to make the situation more realistic is to factor in *damping* or *friction*. This can be modelled in a variety of ways, but the simplest would be to multiply the velocity by some scale factor slightly less than one with every iteration. See example to the right.
Another alternative is to consider energy lost with each collision, and on lines 14 and 16 multiply by -0.95 instead of -1.

```
12  while True:
13      if x > 300 or x < -300:
14          vx *= -1
15      if y > 300 or y < -300:
16          vy *= -1
17      vx *= 0.999
18      vy *= 0.999
19      x += vx
20      y += vy
21      t.goto(x, y)
```

Turn over to see how we can interact with the program using mouse clicks…

Processing Mouse Input

Let's say we want to use the position of the mouse pointer when it is clicked to redirect the ball. That means its velocity vector needs to be pointing in the direction of the mouse, relative to its current position.

```
1   import turtle
2   screen = turtle.Screen()
3   t = turtle.Turtle()
4   t.shape("circle")
5   t.speed(0)
6   t.tracer(10)
7   x = 0
8   y = 0
9   vx = 1
10  vy = -2
11
12  def change(mouseX, mouseY):
13      global vx, vy
14      top_speed = 2
15      vx = mouseX - x
16      vy = mouseY - y
17      speed = (vx ** 2 + vy ** 2) ** 0.5
18      sf = top_speed / speed
19      vx *= sf
20      vy *= sf
21
22  while True:
23      if x > 300 or x < -300:
24          vx *= -1
25      if y > 300 or y < -300:
26          vy *= -1
27      x += vx
28      y += vy
29      t.goto(x, y)
30      screen.onclick(change)
```

In order for the turtle module to make use of screen clicks, we need to explicitly create a **screen** object, using **turtle.Screen()** at the beginning of the code (see line 2).

At the end of the **while** loop (line 30), we use the method **screen.onclick** to register the mouse click and run a fuction, **change**.

This function is where the clever stuff happens: it is defined specifically to deal with mouse click input, so it must have two arguments (I've called them **mouseX** and **mouseY**). In order to make changes to the variables **vx** and **vy** (as opposed to Python just creating new 'local' variables limited to the scope of the function), we need to specify them as 'global' (or 'whole program' variables).

mouseX - x gives the horizontal displacement from my current position to the position of the mouse, and similarly for **mouseY - y**. These give the right directions for our redefined velocity values **vx** and **vy**, but they will likely be much too large, so the next part calculates the speed (eg if **vx** and **vy** were 30 and -40 respectively, the speed would be 50. Using **top_speed**, we can scale both **vx** and **vy** so that we preserve the *direction* of the newly calculated velocity values, but fix the *speed*.

How about gravity?

If we consider gravity as a fixed increase in downwards velocity, we could relatively easily encode that to our program. Modifying very slightly the code from above gives:

```
22  while True:
23      if x > 300 or x < -300:
24          vx *= -1
25      if y > 300 or y < -300:
26          vy *= -1
27      vy -= 0.01
28      x += vx
29      y += vy
30      t.goto(x, y)
31      screen.onclick(change)
```

On line 27, we introduce a constant downward acceleration. Regardless of the current velocity in the vertical direction, it will be decreased by 0.01 each cycle.

The beauty of this is that we don't need to know anything about the behaviour of a particle under a constant downward acceleration in order to code it – we get parabolic motion for free when we run the simulation.

If you start to incorporate damping as well, you may find you get strange behaviour when the particle gets close to the boundary. Can you make sense of why this happens? Can you work out how to fix it? What do we know about the forces when an object hits a wall?