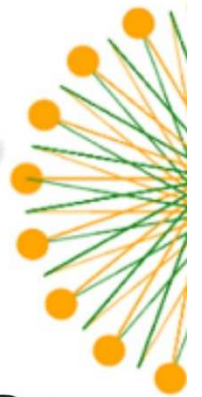
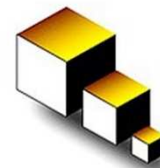
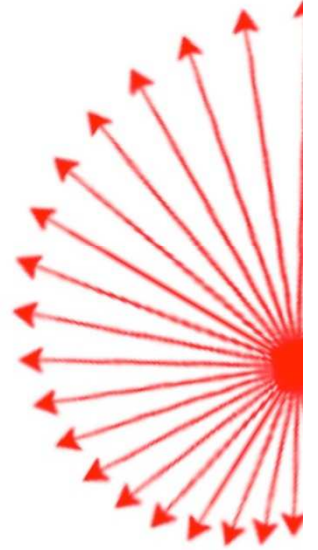
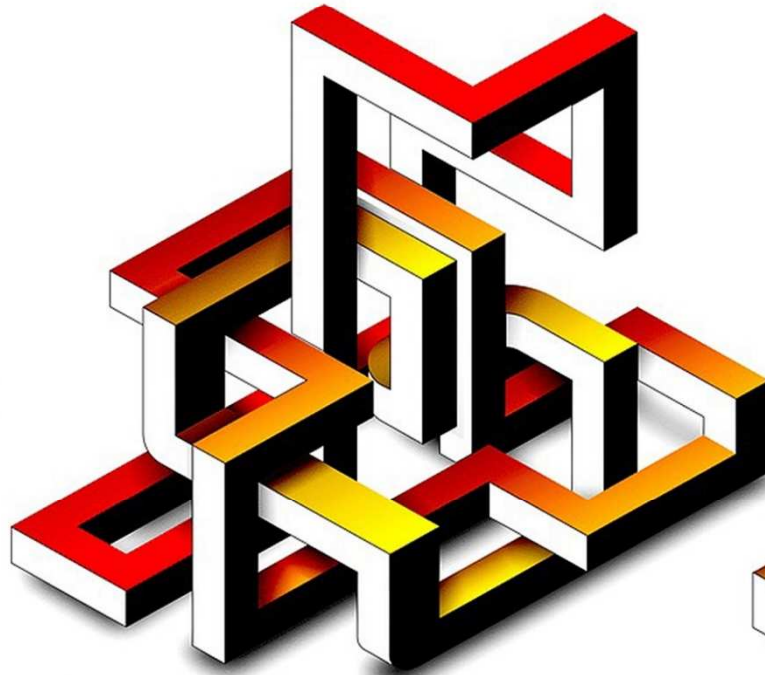
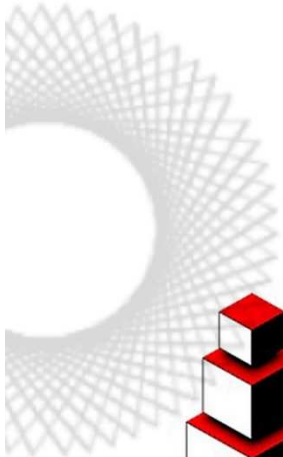




Python Graphics

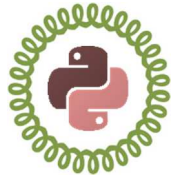


```
while True:
    try:
        sides = int(input("How many sides would you like?"))

    except ValueError:
        print("Please enter an integer.")
        print("")
        continue

    else:
        #the input was recognised, so exit the loop.
        break
```





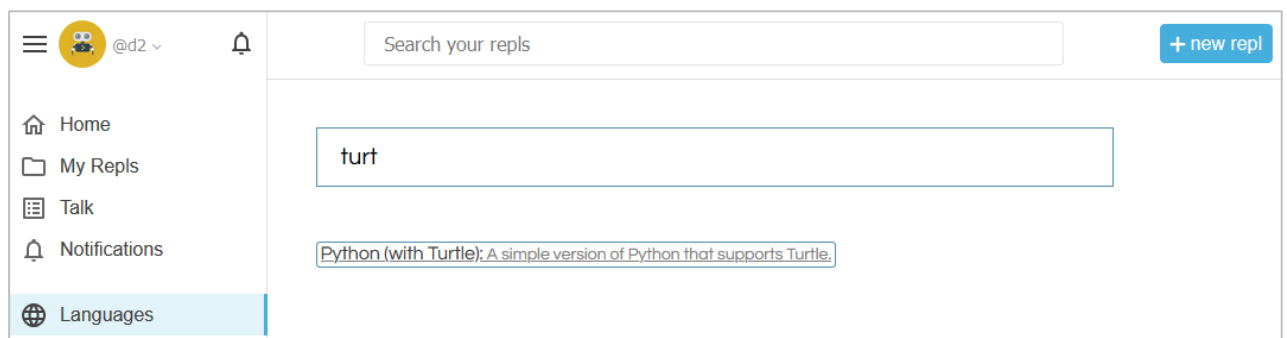
You may have worked through the *Introducing Python with Turtle* activities and learned a little about how programming can be used to create line drawings. We will now create some simple graphics of our own using a straightforward list of instructions. We will also look at some of the other *methods* available when using the *Turtle* programming language.

Aim: To create some simple graphics using Python with Turtle.

Task 1 – repl.it Accounts

repl.it is a website that allows you to play around with bits of Python code without the need to install applications. You should create an account on *repl.it* in order to save and reuse your programs.

- Navigate to the www.repl.it website and follow the instructions to set up an account.
- Once logged in, have a quick look through the different facilities available.
- To start writing a new program, either visit the languages section or click the blue *new repl* button. Search for 'turtle'. Give your program a name and a brief description if you wish.



Task 2 – A Simple Program

The program on the right creates the graphic shown. We have imported the turtle module, created a turtle object (this time named 'Bob' rather than 't') and typed some instructions. The *penup* instruction is like lifting the pen off the page so that lines are not drawn; *pendown* lowers it again.

- Type up the program and name it '**2.2 Simple**'. Use the keyboard shortcuts to copy (Ctrl/Command + C) and paste (Ctrl/Command + V) lines of code; they make coding much faster and help avoid errors.

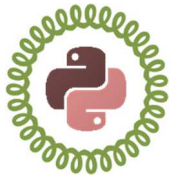
Important: Python is case-sensitive. The name 'Bob' is different to 'bob', the type 'Turtle' is different to 'turtle' and the other instructions in purple must all be lower-case. Check the console for syntax errors if you run into problems.



```

1 import turtle
2
3 Bob = turtle.Turtle()
4
5 Bob.color("red")
6 Bob.forward(50)
7 Bob.left(90)
8 Bob.forward(50)
9 Bob.penup()
10 Bob.forward(50)
11 Bob.pendown()
12 Bob.forward(50)

```



In the previous tasks, we learned about controlling the flow of instructions using count-controlled loops. We now have the knowledge needed to create some fantastic patterns

Aim: Using count-controlled loops to create patterns.

Task 1 – Single Colour Patterns

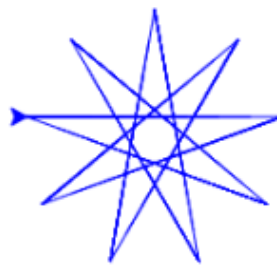
Have a go at creating the graphics below. Our turtle has now been called *Pat*. Save the programs with the names shown.



4.1 Five Point Star

Turns of 144° . We have called the variable 'i' but it can take any name.

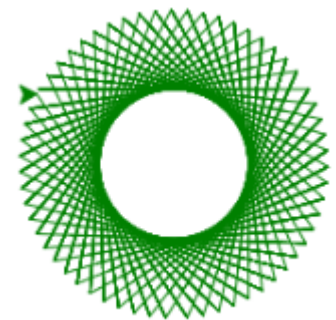
```
10 for i in range(5):
11     Pat.forward(150)
12     Pat.right(144)
```



4.1 Nine Point Star

Turns of 160°
Let's speed things up a little

```
6 Pat.speed(10)
```



4.1 Many Pointed Star

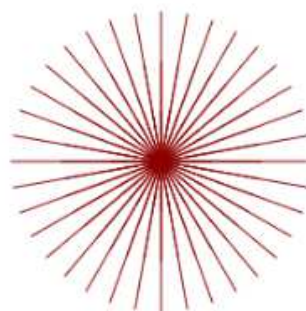
Turns of 122°
You can always click 'Stop' if things are going wrong.

stop



4.1 The Sun

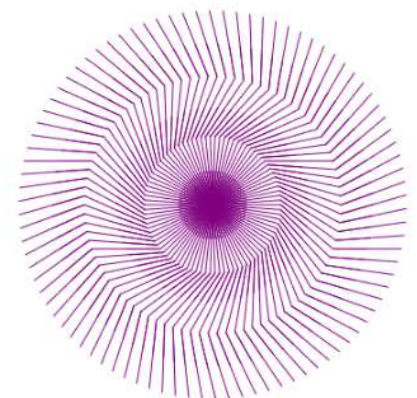
Turns of 170°
Go quickly around a couple of times.



4.1 Setposition

Use *setposition* to return the turtle to the centre after drawing a line

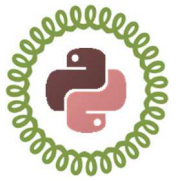
```
11 Pat.penup()
12 Pat.setposition(0, 0)
13 Pat.pendown()
14 Pat.right(10)
```



4.1 Ninja

Draw, return to centre, turn and repeat.





Nested loops are created when one loop sits inside another in a computer program. In Python with Turtle, nested loops can be used to make intricate patterns.

Aim: Using loops within loops to create more complicated patterns.

Task 1 – The Turtle Army

The program below uses a loop to create 3 rows of turtles. It then uses an inner loop so that each of these rows contains 4 turtle stamps.

```

11 for y in range(3):
12     for i in range(4):
13         Pat.stamp()
14         Pat.forward(50)
15     Pat.backward(200)
16     Pat.right(90)
17     Pat.forward(50)
18     Pat.left(90)
19
20 Pat.hideturtle()
21
22 
```

Outer Loop

Set up a loop to create 3 rows of turtles

Nested Loop

This section creates a line of 4 turtle stamps. Notice that the lines of code in this loop are double indented.

Start next line

This section brings the turtle back to the start of the next line.

hideturtle()

Use this method to hide the turtle once the pattern is complete.



- Create the program above. Make sure that you use double tab spaces (or 4 normal spaces) for lines 14 and 15 as this is how Python identifies the inner loop. Save as "5.1 Turtle Army".
- Which line sets up the loop for the 3 rows of turtles? Which line sets up the 4 columns?
- How many pixels does the turtle move forward in a single row before returning to the start of the next row?
- What does the turtle army look like if you delete line 22? Why do you think this happens?
- The program partially shown on the right achieves exactly the same turtle movements without using loops. Roughly how many lines would this program need to complete the pattern?

If you like, set up the full program and save as "5.1 Turtle Army Full". Make sure you use the fork tool and the copy and paste shortcuts for a speedy job.

- Adapt the original program so that it creates 5 rows of 6 turtles, as on the right. You should only need to edit 3 lines of code from the program above.

Note: You will need to work out how far back the turtle must move in line 17.



```

Pat.stamp
Pat.forw

Pat.stamp
Pat.forw

Pat.stamp
Pat.forw

Pat.stamp
Pat.forw

Pat.stamp
Pat.forw

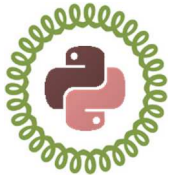
Pat.back
Pat.righ
Pat.forw
Pat.left

Pat.stamp
Pat.forw

Pat.stamp
Pat.forw

Pat.stamp
Pat.back
Pat.righ

```










The problem with the validation in the last activity is that we used code such as the line below to force a choice of either red, green or blue:

```
while line <> "red" and line <> "green" and line <> "blue":
```

Aim: To look at more advanced methods of validating user input.

Python includes a large number of named colours, so by only offering three, you are really limiting the options.

	brown		lemonchiffon		paleturquoise		indigo
	firebrick		khaki		darkslategray		darkorchid
	maroon		palegoldenrod		darkslategray		darkviolet

How do we offer a greater choice of colours to our user? We could add more colours to the conditions set in the while loop, as below.

```
line <> "brown" and line <> "lemonchiffon" and line <> "paleturquoise" and line <> "indigo" and
```

However, this is both untidy and poor programming. We need a better method.

Task 1 – Boolean Data and the Continue Statement

The *Boolean* data type can be one of only two possible values, *True* or *False*. A Boolean variable is useful in validation because we can start off setting it to *False* and then make it *True* if all the conditions are met, therefore exiting the *while* loop.

The *continue* statement sends us back to the start of the loop without executing the remainder of the code.

```

5  is_validated = False
6
7  while is_validated == False:
8
9      sides = int(input("How many sides would you like?"))
10
11     if sides < 3:
12
13         print("Please enter a minimum of 3 sides.")
14         print("")
15         continue
16
17
18     if sides > 100:
19
20         print("Please enter a maximum of 100 sides.")
21         print("")
22         continue
23
24     is_validated = True

```

Path 1

If the data fails the 1st condition then the program *continues* to loop, otherwise it moves on.

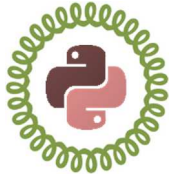
Path 2

If the data fails the 2nd condition then the program *continues* to loop, otherwise it moves on.

Path 3

If both conditions are met, the *is_validated* variable is set to *True* and the flow escapes the loop.

Fork the program named “10.3 Text Validation” and edit it to create the one above. Add comments to each line explaining how it works. Save as “11.1 Is_Validated”.



Many simple games have action that takes place inside an enclosed space, often slightly smaller than the screen. We will look at various ways of setting up boundaries so that our turtle can only move within a confined space.

Aim: To create a confined space for the turtle to move in.

Task 1 – Creating the Border

We will start by creating a rectangular border on the screen then return the turtle invisibly to the centre ready for action. These instructions will be placed in a separate function called *setup*.

The (partly hidden) program on the right sets up the border shown below. Create the program and save as “15.1 Border”.

Notes:

- Decide on a background colour first and set this in place using code such as:

```
screen.bgcolor("orange")
```

We will draw a rectangle over the background for our playing area.

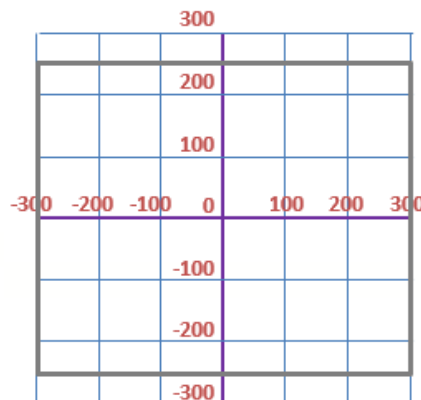
- Create the rectangle with a width of 600 pixels and a height of 500 pixels. The top-right corner will have the coordinates (300, 250). Use the *setposition* method, e.g.:

```
Jill.setposition(300,250)
```

Note: You might find the code in the task “2.7 Position” a useful starting point for the rectangle. Open two browser windows so that you can easily copy and paste code from previous activities.

- Fill the rectangle with white. Once this step is complete, lift the pen up and move the turtle back to (0, 0).
- Increase the speed of the turtle. Perhaps use the special speed of 0 to make things happen instantaneously.

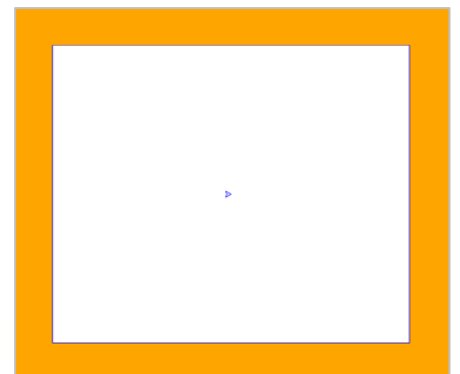
```
Jill.speed(0)
```

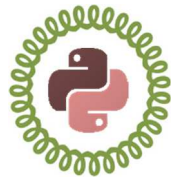


```

1 import turtle
2 Jill = turtle.Tur
3 screen = turtle.S
4
5 #DEFINE FUNCTIONS
6
7 def setup():
8     screen.bgcolor(
9     Jill.color("blu
10    Jill.penup()
11    Jill.speed(0)
12    Jill.setpositio
13    Jill.pendown()
14    Jill.fillcolor(
15    Jill.begin_fill
16    Jill.setpositio
17    Jill.setpositio
18    Jill.setpositio
19    Jill.setpositio
20    Jill.end_fill()
21    Jill.penup()
22    Jill.setpositio
23
24
25 #ACTUAL PROGRAM
26
27 setup()

```



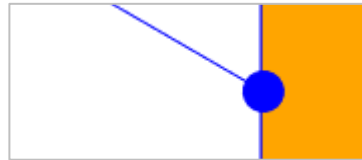


Task 2 – Testing the Border

Add a second function to your program. This one should test the boundaries, sending out lines until you hit the border and then stamping a mark before returning to the centre.

- Copy and paste the code from the task “8.4 Box” as a starting point.
- A stamp placed with an x coordinate of 300 will be centred on the boundary line. Set the limits of movement slightly closer to the centre so that the circle stamp is placed inside the border. We found that an extra 12 pixels worked well.

`while xpos <= 288 and...`



```

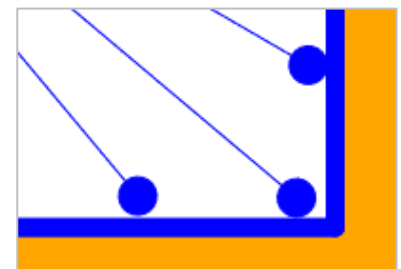
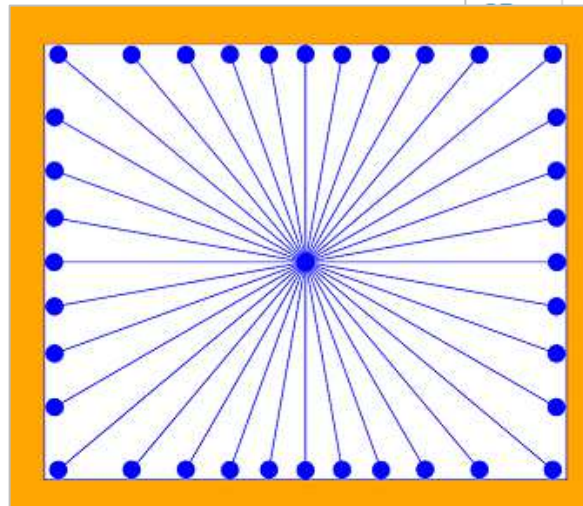
25
26 def testing():
27
28     Jill.pendown()
29     Jill.color("blue")
30     Jill.pensize(1)
31     Jill.shape("circle")
32     Jill.right(50)
33
34     for x in range(360):
35
36         while xpos <= 288 and ypos <= 288:
37
38             Jill.forward(5)
39
40             xpos = Jill.xcor()
41             ypos = Jill.ycor()
42
43             Jill.stamp()
44             Jill.penup()
45             Jill.setposition(0,0)
46             Jill.left(10)
47             Jill.pendown()
48
49
50
51
52 #ACTUAL PROGRAM
53
54 setup()
55 testing()
56
57

```

- Moving the turtle forward 1 pixel at a time will give the neatest results, with the circles being stamped just touching the border. However, it will be a slow process (even if you have set the speed to 0). This is because the program is looping through the code hundreds of times for each line. Setting the movement to 5 pixels is much faster but a little untidy around the edges. We chose 2 pixels when creating the image on the right and waited patiently.
- A better solution might set a thicker line when drawing the border so that the end position is disguised. You may then be able to set the forward value to a much faster 5 pixels without it looking too messy.

Reduce the pen size back to 1 when you have finished drawing the border.

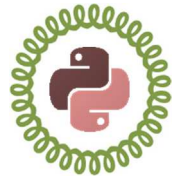
Note: When testing this method, we found we needed to further reduce the limits set in the while loop.



Extension

Start thinking about how you might get the turtle to bounce off the wall. If you would like to have a go at producing a solution, fork your program so that you keep the above code intact.

We will work through the bouncing effect in a later task but it's always better to think about things yourself first.



Released in 1972, Pong was one of the first arcade video games. It consists of two player paddles and a ball that bounces around the court. We will build a basic version of the game.

Aim: To develop a version of the classic Pong game.

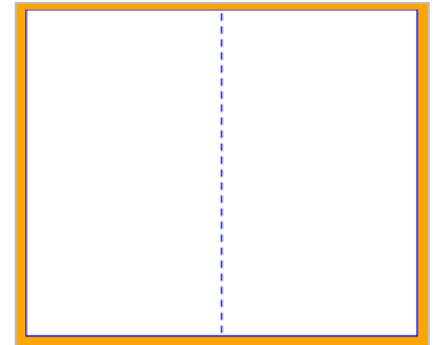
Task 1 – Setting Up the Screen

- a. Create a new repl named “17.1 Pong Setup” and set up the screen as on the right. It is a playing area as used before with a single dotted line down the middle (remember your count-controlled loops).

Note: The code in the program “15.3 Control” will be quite useful for this activity, so perhaps open it in a separate browser window.

Hide the turtle used to set up the screen when finished.

```
Jill.end_fill()
Jill.hideturtle()
```



- b. Add a second turtle for Player 1’s paddle. This can simply be a square shape positioned just inside the left wall (see the picture below). Use the *penup* method so that the turtle doesn’t leave a trace.

Note: The traditional paddle was actually a rectangle but (at the time of writing) this version of Turtle didn’t recognise the *shapsize* method normally used to change the shape of the turtle. Our paddles will have to be squares.

```
# Player 1
Player1 = turtle
Player1.speed(-1)
Player1.shape("square")
Player1.color("blue")
Player1.penup()
Player1.setposit
```

- c. Write a couple of functions that send Player 1’s paddle up and down the playing area. The code from “15.3 Control” will be useful again here.

```
def P1_go_up():
    ypos = Player1.ycor()
    if ypos <= 230:
```

```
screen.onkey(P1_go_up, "a")
screen.onkey(P1_go_down, "z")

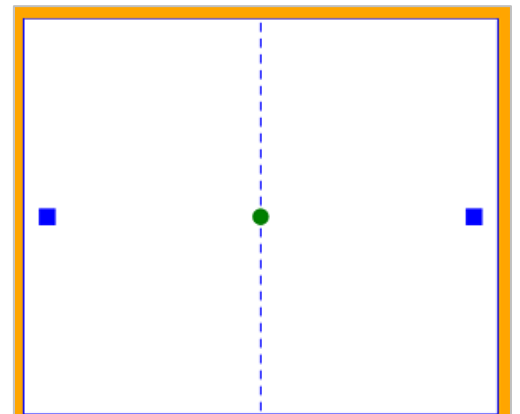
screen.listen()
```

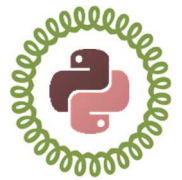
Also, add the function calls and set the keys used for up and down. We have chosen ‘a’ and ‘z’. Finally, make sure the screen is listening for events.

- d. Copy, paste and edit all the code necessary to create Player 2’s paddle. Select some different control keys over to the right of the keyboard for the up and down movement (we chose ‘k’ and ‘m’).

- e. Add another turtle for the ball, positioned in the centre of the screen. Ours is a green circle.

```
# Ball
Ball = turtle.Turtle()
Ball.shape("circle")
Ball.color("green")
```

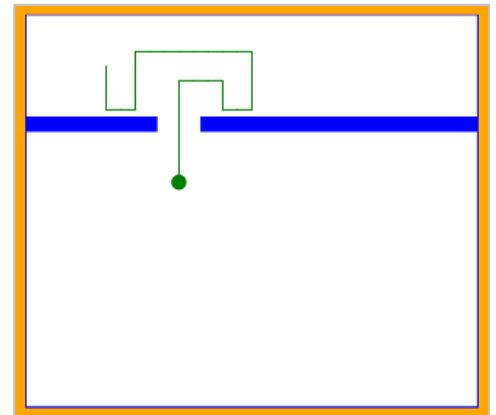




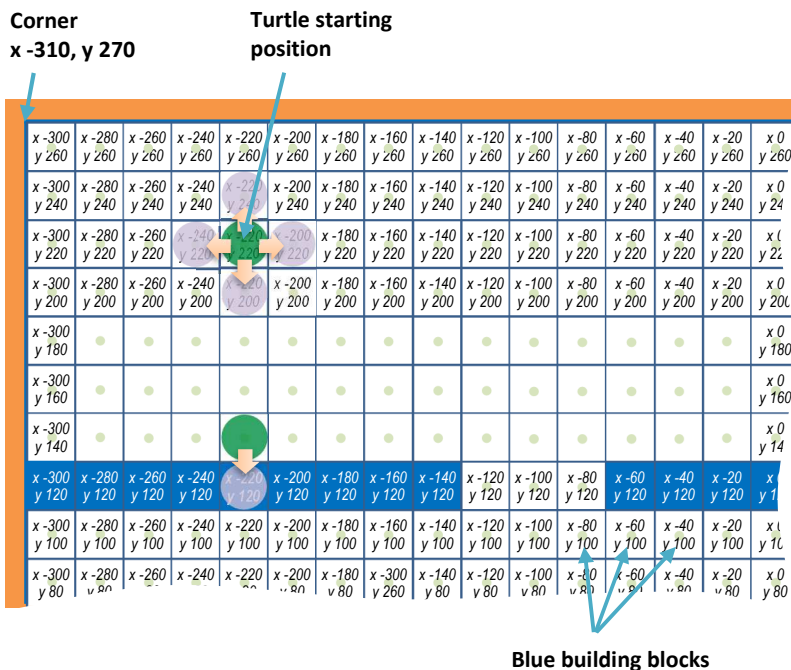
We are going to make a maze for our turtle to navigate. We will start off with something very simple but once you understand the concepts, you will be able to create a much more challenging maze.

Aim: To create a maze for the turtle to navigate.

The picture on the right shows our first maze. It's simply a single hole in a horizontal blue wall. The turtle should not be able to move onto the blue line but it should obviously be able to move through the gap. Unfortunately, you are not able to simply tell the turtle to avoid blue lines. Our solution will be a little more complicated.



We will look at our playing space as a set of small squares, each 20 x 20 pixels. Some of these squares will become building blocks for the walls. The turtle will be able to jump between all the other squares.



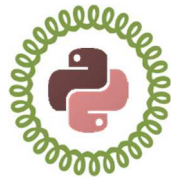
Notes

- We have made our playing space a little larger than in previous activities. The x values now range from -310 to 310 and the y values from -270 to 270. This makes the space easier to work with.
- The coordinates shown are for locations in the centre of each 20 x 20 square.
- The building blocks that make up the wall are square blue stamps. Turtle stamps are 20 pixels by 20 pixels.
- The green turtle will be allowed to jump up, down, left or right distances of exactly 20 pixels.

Task 1 – Observations

Use the diagram above to answer the following questions.

- f. What will be the coordinates of the 4 corners of the new playing space? These are used during the initial setup.
- g. If working from left to right, what are the coordinates of the first blue stamp?
- h. How many blue building blocks are created in a row? What gap is left before we begin stamping again?
- i. The higher of the two green circles is the starting position of the turtle. What are its coordinates?
- j. What are the coordinates of the 4 squares the turtle could jump to from its starting position?



Task 2 – Creating the Maze

- In a new repl named “19.2 Wall”, start by setting up your playing space as in earlier tasks. Remember that the x limits are now -310 and 310 pixels and the y limits -270 and 270 pixels.
- Move your turtle to the location of the first blue block. This will be our *wall*. Set the shape to a blue square.
- Use a loop to stamp a row of building blocks. Jump across the gap (remembering that you have already moved 20 pixels forward at the end of the last loop) and then use a second loop to complete the row.
- We are going to prevent the turtle from moving into any space occupied by a blue building block. For this to work, we will need to remember the location of each building block we add to the maze.

The locations of the blocks will be stored in a list named *Walls*, so put the empty list in place first. We will then add the coordinates to the list as the blocks are created.

#ACTUAL PROGRAM

```
Walls = []
setup()
```

```
Jill.setposition(-300,120)
Jill.shape("square")
Jill.fillcolor("blue")
```

```
for z in range (9):
```

```
    Jill.stamp()
    Jill.forward(20)
```

```
Jill.forward(60)
```

```
for z in range (19):
```

- Each time you stamp a building block, add the coordinates to the *Walls* list. This can be done with the code on the right. You will need to do the same for the second row of blocks.
- Once the row of blocks is complete, use the code below to print the list of stored coordinates to the console. This is a great way to check that the program is working as intended.

```
for z in range (9):
```

```
    Jill.stamp()
```

```
    xpos = Jill.xcor()
```

```
    ypos = Jill.ycor()
```

```
    Walls.append((xpos, ypos))
```

```
print(Walls)
```

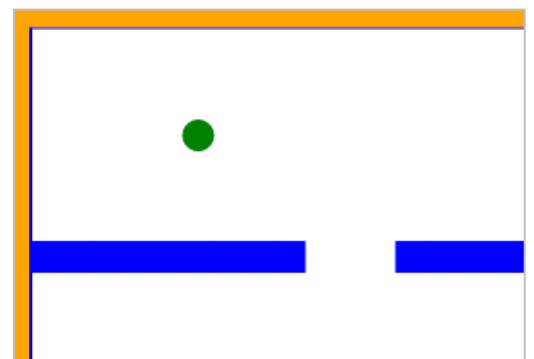
Note: This line of code can be deleted or commented out later.

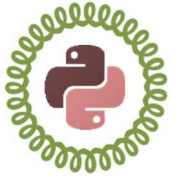
```
[(-300.0, 120.0), (-280.0, 120.0), (-260.0, 120.0), (-240.0, 120.0), (-220.0, 120.0), (-200.0, 120.0), (-180.0, 120.0), (-160.0, 120.0), (-140.0, 120.0), (-60.0, 120.0), (-40.0, 120.0), (-20.0, 120.0), (0.0, 120.0), (20.0, 120.0), (40.0, 120.0), (60.0, 120.0), (80.0, 120.0), (100.0, 120.0), (120.0, 120.0), (140.0, 120.0), (160.0, 120.0), (180.0, 120.0), (200.0, 120.0), (220.0, 120.0), (240.0, 120.0), (260.0, 120.0), (280.0, 120.0), (300.0, 120.0)]
```

- Create a new turtle to act as your escaping player and move it to the starting position. Ours is a green circle called *Mazey*. Yours may be different.
- Add the *onkey* method calls as used in previous activities and make sure the screen is listening for events.

```
screen.onkey(go_up,"up")
screen.onkey(go_down,"down")
screen.onkey(go_left,"left")
screen.onkey(go_right,"right")

screen.listen()
```





Answers

Task 1 – Controlling the Flow

Scenario
Keep rolling the die until you get a 6.
If you are a member then join the queue on the left, else join the queue on the right.
Go around the whole course three times.
First activities: Group 1 is climbing; Group 2 is kayaking; Group 3 is hiking.



Control in Place
Repeat something until a certain condition has been met.
Do one thing if one condition is true, otherwise do another thing.
Repeat something a set number of times.
Select from a range of possibilities depending on a condition.

Task 2 – Looping through a Range

```
import turtle

Bethyl = turtle.Turtle()

Bethyl.color("blue")
Bethyl.pensize(5)
Bethyl.fillcolor("silver")

Bethyl.begin_fill()

for x in range(6):
    Bethyl.forward(50)
    Bethyl.left(60)

Bethyl.end_fill()
```



<https://repl.it/@d2/32-Hexagon>

```
import turtle

Bethyl = turtle.Turtle()

Bethyl.color("orange")
Bethyl.pensize(10)
Bethyl.fillcolor("yellow")

Bethyl.begin_fill()

for x in range(8):
    Bethyl.forward(40)
    Bethyl.left(45)

Bethyl.end_fill()
```



<https://repl.it/@d2/32-Octagon>

Count-Controlled Loops Answers (page 2)

Task 2 – Looping through a Range (Cont.)

```
import turtle

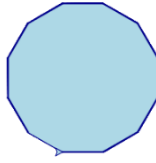
Bethyl = turtle.Turtle()

Bethyl.color("blue")
Bethyl.pensize(1)
Bethyl.fillcolor("light blue")

Bethyl.begin_fill()

for x in range(12):
    Bethyl.forward(30)
    Bethyl.left(30)

Bethyl.end_fill()
```



<https://repl.it/@d2/32-Dodecagon>

```
import turtle

Bethyl = turtle.Turtle()

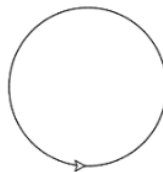
Bethyl.color("black")
Bethyl.pensize(1)
Bethyl.fillcolor("white")

Bethyl.begin_fill()

for x in range(90):
    Bethyl.forward(5)
    Bethyl.left(4)

Bethyl.end_fill()
```

<https://repl.it/@d2/32-Circle>



Task 3 – Looping through a List

<https://repl.it/@d2/33-List-Controlled>

```
import turtle

Bethyl = turtle.Turtle()

for distance in [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110,
120, 130, 140, 150, 160, 170, 180, 190, 200]:

    Bethyl.forward(distance)
    Bethyl.left(90)
```

Count-Controlled Loops Answers (page 3)

Task 4 – How the Sample Program Works

```
1 import turtle
2
3 t = turtle.Turtle()
4
5 for c in ['red', 'green', 'yellow', 'blue']:
6     t.color(c)
7     t.forward(75)
8     t.left(90)
```

Line 1 Imports the *turtle* module into the program.

This enables the program to understand the **turtle** language; it's like the translation guide. This line must stay in place whenever you write a turtle **program**.

Line 3 Says create a turtle and name it 't'.

With our turtle ready to use, we can refer to it by the name 't' whenever we need to. You may change the **name** of your turtle if you prefer, providing you use the new name in the remainder of the code.

Line 5 Places four colours in a list and sets up a loop.

There is a list of 4 **colours** to work through. The 'for c in' part of line 5 creates the loop. The program will run through the loop repeatedly, working through the list of colours until there are no more left.

The colour being used at any point is held in a **variable** called 'c'. A variable is like a **box** that holds a piece of data for later use (in this case, it is used again in line 6).

Line 6 Gives the turtle (named 't') whatever colour is presently held in the variable 'c'.

The first time the program runs through the loop, the turtle is given the colour **red**. The second time, it is given the colour **green** etc.

Line 7 Instructs the turtle to move forward 75 **pixels**.

Line 8 Instructs the turtle to rotate left (or anticlockwise) through 90 **degrees**.

Questions

1. In which line is the *turtle* object created (or defined)?

Line 3

2. What happens if you delete line 1 and run the program? Look at the error description in the console. Why do you think this happens?

NameError: name 'turtle' is not defined on line 3. We haven't imported the turtle module so the program doesn't understand what the word 'turtle' means.

3. How many times will line 6 be read by the computer when the program is run?

4 times

4. What happens if you misspell one of the colours?

The last known colour is used again. If you misspell the first colour in the list, then black is used for that line instead.

5. What do you think would happen if you simply add more colours to the list without changing anything else? Try it.

The lines are redrawn with the new colours.