Task One

1. Introduction

```
1       import turtle
2       t = turtle.Turtle()
3
4       t.forward(100)          # Go forwards 100 units
5       t.left(45)              # Turn left 45 degrees
6       t.forward(30)           # Go forwards 30 units
```

Click on the "Task 1" button to load in the above program, and then click the "Run" button.

Here you are creating a new turtle, called **t**, then telling it to go forward 100 units, go left 45 degrees, and then go forward again by 30 units.

The text after the **#** is called a **comment**. This is English text that is ignored by Python. This is used to explain how the code works. It has no effect on the behaviour of the turtle.

Modify the above code to make the turtle go forward by 200 units, turn 90 degrees left, then go forward by another 100 units.

Other commands you can use to move the **t** are `t.right` and `t.backward`. These are used in exactly the same way as `t.forward` and `t.left`.

Change the code to make the turtle turn right by 40 degrees, then go backward by 200 units

2. Numbers and variables

You are using a language called Python to tell the turtle what to do.

Python can also do other things, such as basic maths.

```
1       print 3
2       print 8+11
3       print 9-10
4       print 40*5
5       print 10/2
6       print ((87+3)/30*(51-3*9))/6
```

Type in the above code and click "Run". You should see the results of these arithmetic operations in the results box.

+ represents addition, - represents subtraction, * represent multiplication, and / represents divide. These can be combined to make very long arithmetic expressions, just like in maths.

```
1       x = 5
2       print x
3       x = 8
4       print x
5       x = x + 1
6       print x
7       print 2*x
```

Type in the above code and click "Run".

Here, you are using a **variable**. Python is creating some storage space called **x** which is being used to hold a number. Notice how the `print x` statements all result in a different number being output.

Line 5 is

```
5       x = x + 1
```

This can be thought of as **the new value of x** is set to be **the old value of x** plus 1. This has the result of increasing the stored value of **x** by 1.

3. Using numbers and variables

```
1       import turtle
2       t = turtle.Turtle()
3
4       x = 100           # Set a variable 'x' to have value 100
5       t.forward(x)      # Go forward 'x' units
6       t.left(45)
7       t.forward(x)      # Go forward 'x' units again.
```

Click on "Task 1" and modify the code so it looks like the above. (Changes to be made have been highlighted). You do not need to copy the text after the #. Try modifying the value of x on line 4 and see how this affects the shape drawn by the turtle.

Notice how the two lines drawn are the same length. The code can be modified to make the second line twice as long as first by changing lines 7 as follows:

```
5       t.forward(x)      # Go forward 'x' units
6       t.left(45)
7       t.forward(2*x)    # Go forward '2*x' units.
```

Try changing the code to make:

- **The second line three times as long as the first line**
- **The first line 40 units shorter than the second line.**

4. Draw a square

Make the turtle draw a square with edges of size 100. Try to use a variable to control the length of the square's sides, as this will make it easy to resize the square.

Try to draw the biggest square you can without the turtle going out of the canvas.

Task Two

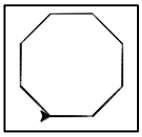
1. A loop

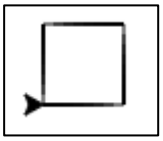
```
1       import turtle
2       t = turtle.Turtle()
3
4       for i in range(6):      # Repeat the following lines 6 times.
5           t.forward(40)
6           t.left(45)
```

Click on the "Task 2" button to load in the above program, and then click the "Run" button.

Here we are using something called a **for loop**. Line 4 is saying "execute the following lines 6 times". This results in the turtle going forward and left 6 times. At the moment it is drawing ¾ of an octagon.

Tweak the above code to make the turtle draw the whole octagon. Then modify the code to draw a square.





2. Blocks and indentation

In more detail, a for loop works by telling Python to run the next **block** of code some number of times. Any lines in this block are said to be **inside the loop**. Indentation is used in python to describe where blocks of code start and end.

```
1       import turtle
2       t = turtle.Turtle()
3
4       for i in range(4):
5           t.forward(40)
6           t.left(45)          # This is 'inside' the for loop
```

Here, lines 5 and 6 are part of the same block; they are both 'inside' the for loop. This means that both lines will be executed in turn as part of the for loop on line 4.

```
1       import turtle
2       t = turtle.Turtle()
3
4       for i in range(4):
5           t.forward(50)
6       t.left(45)              # This is now 'outside' the for loop
```

Here, lines 5 and 6 are not part of the same block; only line 5 in 'inside' the for loop. This means only line 5 is executed as part of the for loop on line 4.

In order for Python to consider a set of lines to be a block, they must have the same number of spaces before the line text starts. Generally, the number of spaces used is in multiples of four.

Try both versions of the code above and see what happens to the turtle in each case. You do not need to copy the text after the #s.

3. Loops within loops

There is nothing stopping you from putting a loop within a loop.

First write a loop to draw a square, if you have not done it already.

You can do this either by modifying the code above or writing it yourself. Try writing it yourself first, using the octagon-drawing code above as guidance, as this will help you learn it better.

Indent these lines of code to form a block, then add the line '`for j in range(4):`' above the block and add the line '`t.forward(80)`' below the block.

```
1       import turtle
2       t = turtle.Turtle()
3
4       for j in range(5):
5           # Your code to draw a square goes here
6       t.forward(80)
```

Make sure your square-drawing code is correctly indented to form a block!

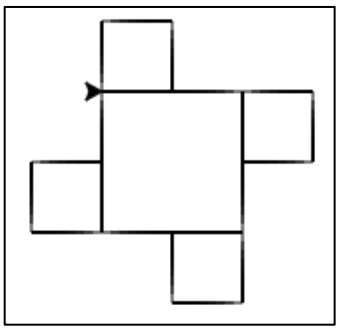
Run this code and see what happens. Try changing the numbers in the code to draw more/fewer squares, larger or smaller squares, and squares that are further or closer spaced.

Lines 5-7 are now 'inside' the for loop on line 4. Lines 6-7 are inside two loops; the loop on line 4 and the loop on line 5. The loop on line 5 is executed 5 times, so its body is executed 4*5 = 20 times.

In order to make it more clear which line the turtle is drawing, you can add in `pencolor` commands to change the colour.

```
        for j in range(4):
            for i in range(4):
                t.pencolor('blue')
                t.forward(40)
                t.left(90)
            t.pencolor('red')
            t.forward(80)
```

Using two loops, draw a square with a square at each corner.



Task Three

1. Procedures

At the moment, Python runs code line-by-line.

```
1       import turtle
2       t = turtle.Turtle()
3
4       x = 60
5       t.pencolor('red')
6       t.forward(x)
7       t.pencolor('blue')
8       t.forward(x)
9       t.pencolor('yellow')
10      t.forward(x)
```

This code will draw a red, blue, and then yellow line.

Sometimes we want to modify the order in which Python runs the lines.

```
1       import turtle
2       t = turtle.Turtle()
3
4       def drawSquare():       # Define a new procedure, drawSquare
5           for i in range(4):
6               t.forward(40)
7               t.left(90)
8
9       drawSquare()            # Call drawSquare
```

Click on the "Task 3" button to load in the above program, and then click the "Run" button.

Here, the code is **defining** a **procedure** called `drawSquare` on line 4. The following block of code (lines 5-7) then 'belong' to that procedure. It is then **calling** that procedure on line 9. This means that the block of code 'belonging' to `drawSquare` is executed.

If you did not do task two, line 5 is saying 'execute the following block of code 4 times'.

Defining a procedure tells Python **how** to do something (in this case, we are telling it how to draw a square). When a procedure is called it is telling Python **when** to do something. A procedure must be defined before it is used.

When defining a procedure, we use the keyword **def**, then give it a name (**drawSquare** in this case). To call the procedure, you need to type the produce's name exactly as it appears in the definition (including any capitals), followed by parentheses.

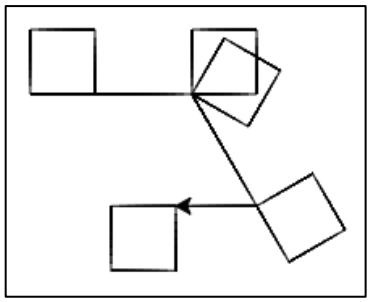
Change line 9 to include a spelling mistake (e.g. '`drowSquar()`') and see what happens when you click "Run". Remember to change it back to `drawSquare()` afterwards.

The main benefits of using procedures are:

- Abstraction – You can now tell Python to 'draw a square' rather than 'go forward and left 4 times'. You are telling Python **what** to do rather than **how** to do it.
- Code re-use – At any point in the program you can just write '`drawSquare()`' and Python will draw a square. There is no need to type out the loop again.

Using `drawSquare()` and the `t.forward`, `t.left`, and `t.right` commands, draw 5 squares at various places.

The results of your drawing may look something like this:



2. Using procedures

On your sheet 'What can the turtle do?' there is a list of instructions we can give to the turtle. These are called **procedures** and they tell the turtle to either **do something** or **tell you something**.

Writing `t.circle(30)` is instructing the turtle to draw a circle of radius 30. You have already used `t.forward`, `t.left`, `t.right` and `t.backward` in a similar way. These are all instructions that tell the turtle to **do something**.

All the procedures under "Getting information from the turtle" will **tell you something**.

Adding in the two lines:

```
x = t.xcor()      # Get the turtle's x coord and store in 'x'
print x           # Display the value of 'x'
```

When this line is reached, the turtle's current x coordinate will be printed (it will appear in the top right panel).

Put the two above lines in various places in your code and see how the turtle's x coordinate changes throughout the program. You do not need to copy the text after the #.

After we have done `x = t.xcor()`, the variable **x** will not change value. We can use the value of **x** in other commands, e.g. `t.left(x)`.

3. Letters

In order to do more complex drawings, we may need the turtle to "remember" where it was at some point in time. We can do this using a procedure to get the information, then storing it in a variable for later use.

Write a procedure to draw a letter of your choice.

Here is an example to draw an A. Remember to call your procedure when running your code!

```
def drawA():
    t.left(70)          # Draw left leg
    t.forward(60)
    x = t.xcor()        # Remember position of this partway
    y = t.ycor()        # up the left leg.
    t.forward(90)

    t.right(140)        # Draw right leg
    t.forward(150)

    t.backward(60)
    t.setpos(x,y)       # Draw line to "remembered" spot.
```

Task Four

1. Random numbers

```
1       import turtle
2       from random import randint
3       t = turtle.Turtle()
4
5       length = randint(10,100)      # 'length' has a random value…
6
7       for i in range(4):
8           t.forward(length)         # Go forward 'length' units
9           t.left(90)
```

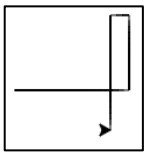
Click on the "Task 4a" button to load in the above program, and then click the "Run" button.

Try clicking the button multiple times. What happens to the size of the square?

Every time we run the program, the **length** variable is assigned a random number between 10 and 100.

Move line 5 to be inside the for loop on line 7 (Put the line between lines 7 and 8, and put 4 spaces before the line if necessary). See how the turtle acts now.

```
1       import turtle
2       from random import randint
3       t = turtle.Turtle()
4
5       for i in range(4):
6           length = randint(10,100)
7           t.forward(length)
8           t.left(90)
```



Originally, the **length** was randomly rolled before entering the loop. Each time the loop was executed the line **t.forward(length)** would move the turtle forward by the same amount. By moving line 5 "inside the loop" the length gets a new value for each line drawn, so the lines now have different lengths.

2. Infinite loops

When the length was rolled randomly just once, turtle was stopped after drawing 4 lines. After this point, the turtle was just retracing its own steps, so there was no point continuing. However, now that you are randomly choosing the length for each line, there is no reason to stop after 4 iterations.

Change line 5 to "`while True:`". How many times does the turtle draw lines?

When we write '`while True:`' what we are saying is 'endlessly repeat the next block of lines'.

WARNING: Never use '`write True:`' without having at least one turtle movement command inside the loop. If you do this your browser will freeze.

3. If statements

Unfortunately, at some point the turtle will go off the canvas. There are no guarantees that it will ever come back.

In order to demonstrate this a little easier, we will look at a wiggly-line turtle.

```
1       import turtle
2       from random import randint
3       t = turtle.Turtle()
4
5       while True:                    # Infinite loop
6           t.forward(20)
7           angle = randint(-45,45)    # 'angle' has a random value…
8           t.left(angle)
```

Click on the "Task 4b" button to load in the above program, and then click the "Run" button.

This turtle moves forward by a fixed amount each time but turns by a random amount left or right, up to 45 degrees.

This turtle will quickly go off screen. What we want to do is say:

"If the turtle is off screen, turn it around 180 degrees. Else draw wiggly lines."

We need to do this with an if statement. For this example, it will be structured as follows

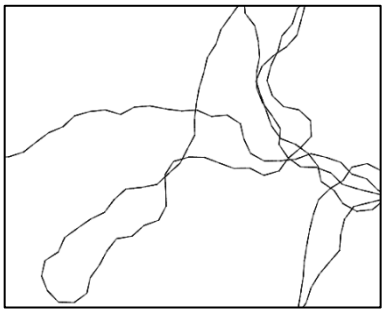
```
        while True:
            if t.offscreen():
                # turn around 180 degrees
                # go forward 30 units
            else:
                # draw wiggly lines, as in lines 6-8 above
```

Replace the loop in lines 5-8 of the original code with the loop above. Replace the lines starting with # with the correct code.

`t.offscreen()` is an **expression** that evaluates to yes or no, depending on whether the turtle is off the canvas or not. The if statement runs lines conditionally:

```
        if t.offscreen():
            # This block is run if t.offscreen() evaluates to yes
        else:
            # This block is run if t.offscreen() evaluates to no
```

If you have got the code correct, it might draw something like this.



You may find the turtle sometimes gets stuck on the edge of the world, constantly turning. There are a number of tweaks that can made to the turtle's behaviour to prevent that.