Python turtle worksheet[1]

Introduction

A number of different programming lanuages and environments are common in computational neuroscience; MATLAB and Python are common, along with the specialized packages NEURON and NEST, Julia is believed by some to be the next big thing. For those of you who haven't done programming before, this is a Python tutorial using the classic Turtle package, a package that allows you to move a cursor around on the screen. This might seem silly at first, but can be a useful way to learn Python since it gives direct visual feedback.

Anyone who has programmed before should skip ahead to either of the more advanced challenges, trying to draw a neuron with Turtle, an interesting challenge since it makes you think about how Neurons might grow themselves, or writing a short programme in Julia.

There are lots of ways to use Python, using a `repl` where you put in commands and they run straight away, by writing programmes in a file and running the file or using a notebook, which is a mixture of the two. Here, for simplicity we use `repl.it` which allows you to run Python programmes in your browser. All the example programmes are also available at `https://github.com/conorhoughton/teaching_misc/tree/master/python_workshop/code`, the programmes there have extra lines at the end that are needed to keep the Turtle console on the screen and to save figures; a comment in Python is marked with `#` and this bit of the programme is seperated off with a comment.

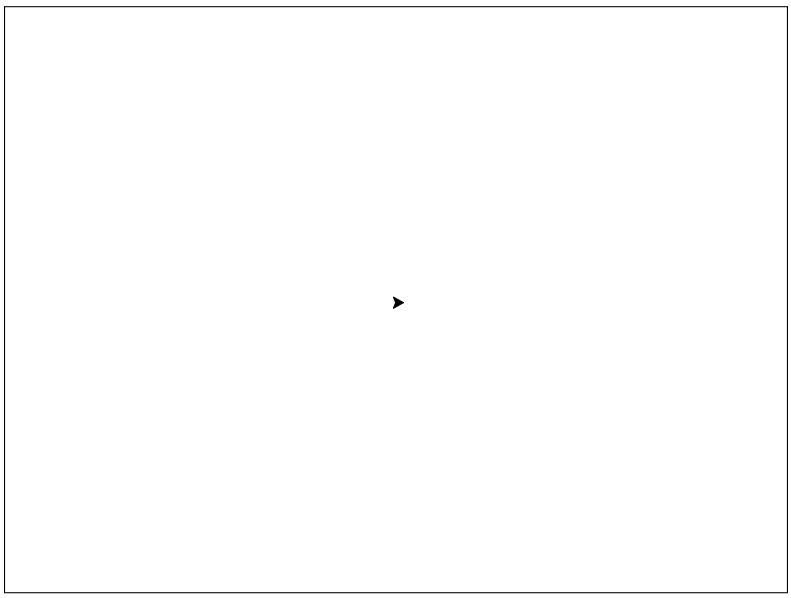
Turtle: drawing a line

Here is a simple turtle programme (`turtle_doing_nothing.py`):

```
1 from turtle import *
2
3 tom=Turtle()
```

Line 1 isn't worth spending much time on here, the first line imports the library of commands related to turtle. **Line 3** is important, it tells the computer to make an object, in this case a `Turtle` and call it `tom`, it knows what a `Turtle` is from the library it imported in **line 1**. In the instructions on what to do when making a `Turtle` the computer is told to open a graphics window and to draw the turtle, a little arrow shape.

[1] `https://github.com/conorhoughton/teaching_misc/tree/master/python_workshop/`

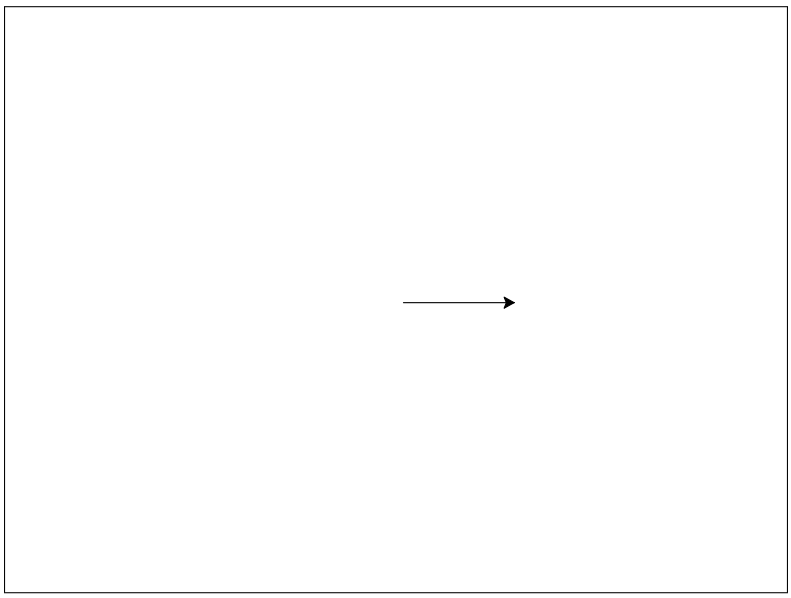


We call `tom` an *instance* of a `Turtle`; `Turtle` is the class of objects, `tom` is an example of it.

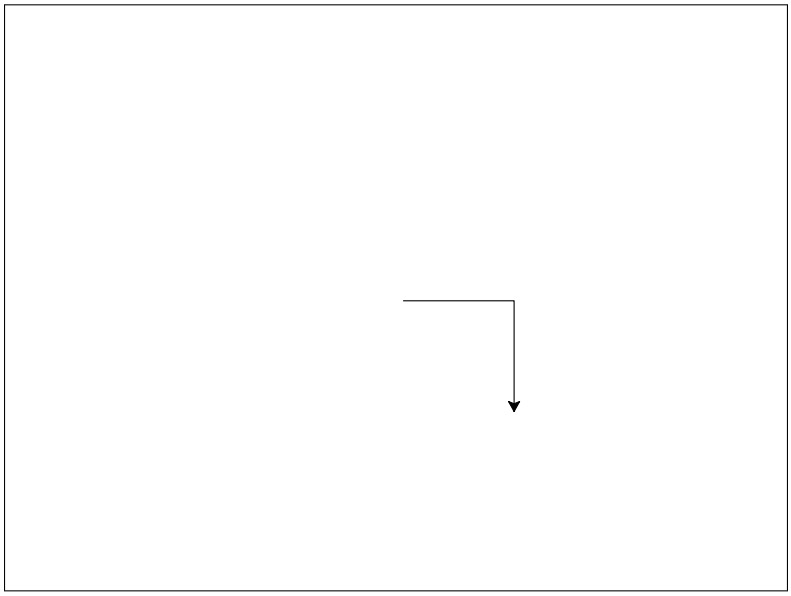
Here the turtle does something (`line.py`):

```
1 from turtle import *
2
3 tom=Turtle()
4
5 tom.forward(100)
```

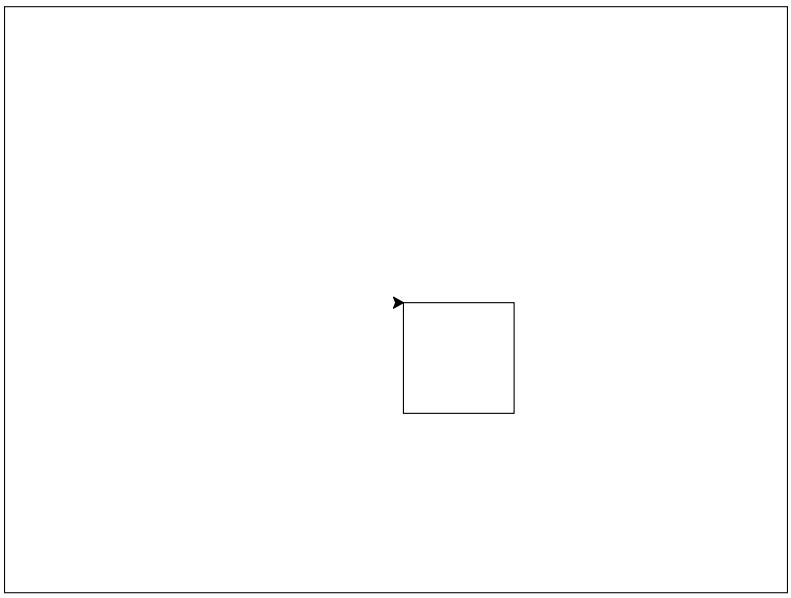
The extra line, **line 5**, tells the turtle to move forward by 100 units, this is an important piece of Python syntax, to tell an object to do something you use a dot followed by the command, here it tells the `Turtle` called `tom` to perform the command `forward`. Of course, the command has to make sense for whatever type of object it is dotted onto, but here it does, `forward` is one of the defined commands for a `Turtle` object; we will see more of them as we go along, or google can provide a list. Generally, commands that are associated with object, the commands that you use with a dot, are called *methods*.



`Turtle` objects have another command `right(90)` which turns the turtle by 90°. QUESTION: Can you write a programme to draw this (`right_angle.py`):



QUESTION: How about a square?



Flow control

Perhaps you programme to draw a square looked like this

```
1  from turtle import *
2
3  tom=Turtle()
4
5  tom.forward(100)
6  tom.right(90)
7  tom.forward(100)
8  tom.right(90)
9  tom.forward(100)
10 tom.right(90)
11 tom.forward(100)
12 tom.right(90)
```

There are a few problems with this, most obviously it is boring writing in the same two lines again and again; secondly, the programme is inflexible and hard to read, we'll deal with the inflexible bit later, but as for the hard to read, to know that it draws four lines and has four corners you need to count the lines; it would be better if the 'fourness' was more apparent, as it is in this programme (`square_loop`):

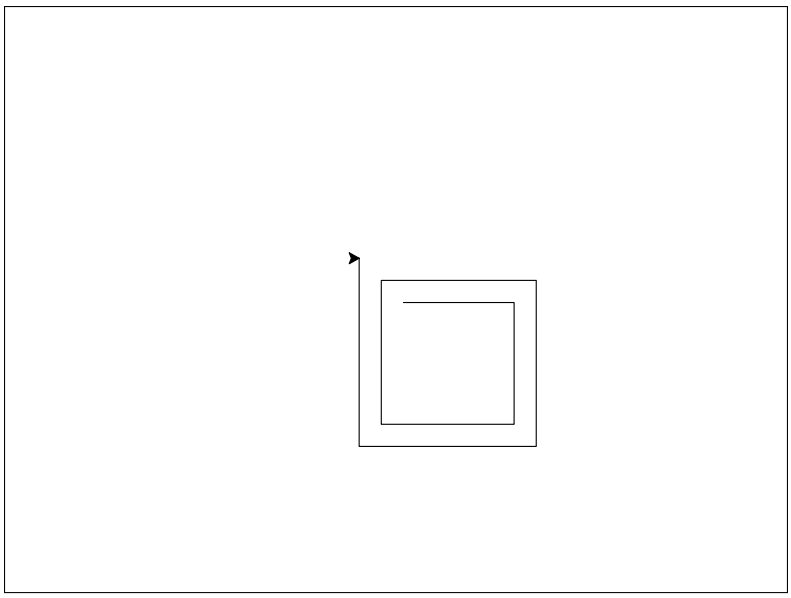
```
1  from turtle import *
2
3  tom=Turtle()
4
5  for i in range(0,4):
6      tom.forward(100)
7      tom.right(90)
```

The business part of this program is **line 5**; `range(0,4)` is the list of numbers `[0,1,2,3]`, so starting at zero and ending one before four; the full command says that `i` takes each value in this list in turn and then does all the stuff belonging to the command. Here the command is a `for`, a command that says 'do everything that belongs to the me once for every value the variable, in this case i, is instructed to take'. In Python stuff belonging to a command is indented, so that means it does **line 6** and **line 7** once for each item in the list: four times. The 'stuff belong to a command' is called the *block*, so in Python blocks are denoted with indents.

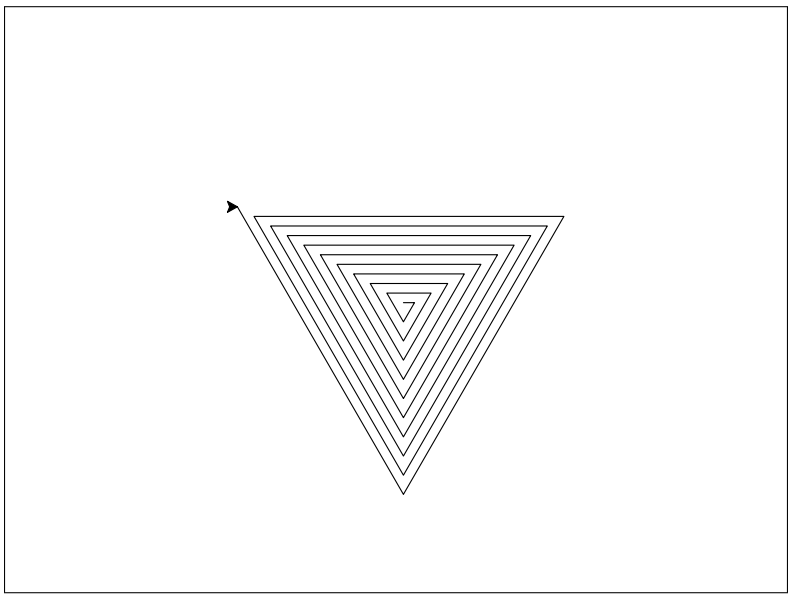
Of course, in this programme `i` isn't used for anything except counting but it could be (`spiral1.py`):

```
1 from turtle import *
2
3 tom=Turtle()
4
5 for i in range(0,8):
6     tom.forward(100+10*i)
7     tom.right(90)
```

giving



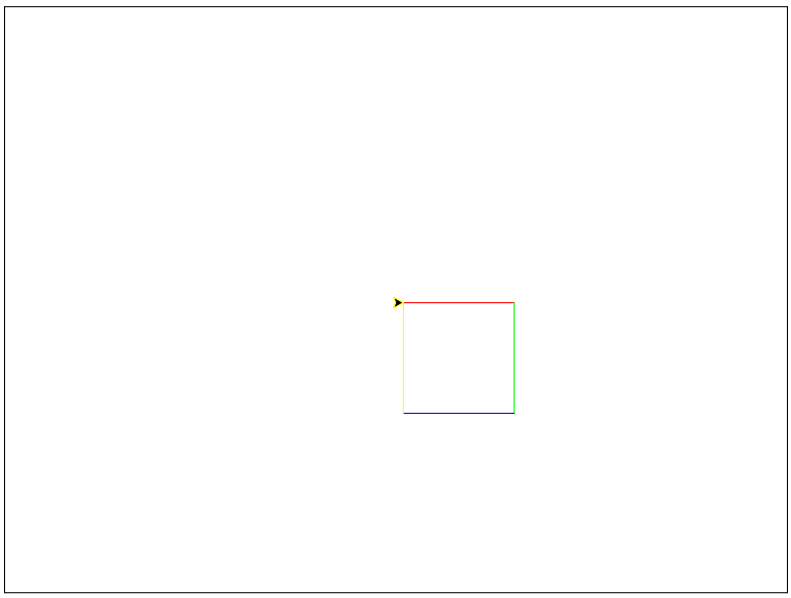
`i` is *variable*, it stores some data, in this case a number which changes each time the programme goes around the `for` loop. One slightly confusing thing is 'scoping', which is where the variable is defined; the `i` is only defined inside the `for` loop, that is, it only exists while the programme is executing **line 5** to **line 7**, but that's fine, that's where we use it. QUESTION: Can you draw a triangular spiral like this:



The list in the `for` loop doesn't have to be a `range`, in this example (`square_color.py`)

```
1  from turtle import *
2
3  tom=Turtle()
4
5  colors=['red','green','blue','yellow']
6
7  for color in colors:
8      tom.pencolor(color)
9      tom.forward(100)
10     tom.right(90)
```

`colors` defined in **line 5** is a list of colours and the variable in the `for` command takes each of these values in turn. `pencolor` is another `Turtle` method, it changes the pen colour so we get



In the colour example we use a variable `colors` to store a list; here is an example where we use a variable to store an integer `n` which gives a number of sides.

```
1 from turtle import *
2
3 tom=Turtle()
4
5 n=8
6
7 for i in range(0,n):
8     tom.forward(100)
9     tom.right(360.0/n)
```

This is useful because we need the number of sides twice, once in **line 7** where it gives the number of sides, and the second time in **line 9** where it is used to calculate the turning angle. Obviously it would be possible to write the number in those two places, but that would be a very bad programming style because it would mean changing it in those two places if you wanted to change the number of sides to the polygon. It would be easy to get this wrong, maybe not for only two places, but in a more complex programme there may be many places a value needs to be changed, leading to errors. Furthermore, keeping `n` seperate makes the programme easier to understand.

Here is a program for drawing a circle (`circle.py`); `Turtle` has a function that draws a circle directly, but that seems like cheating:

```
1 from turtle import *
2
3 tom=Turtle()
4
5 radius=100
6 step_size=2*3.141592*radius/360
7
8 for i in range(0,360):
```

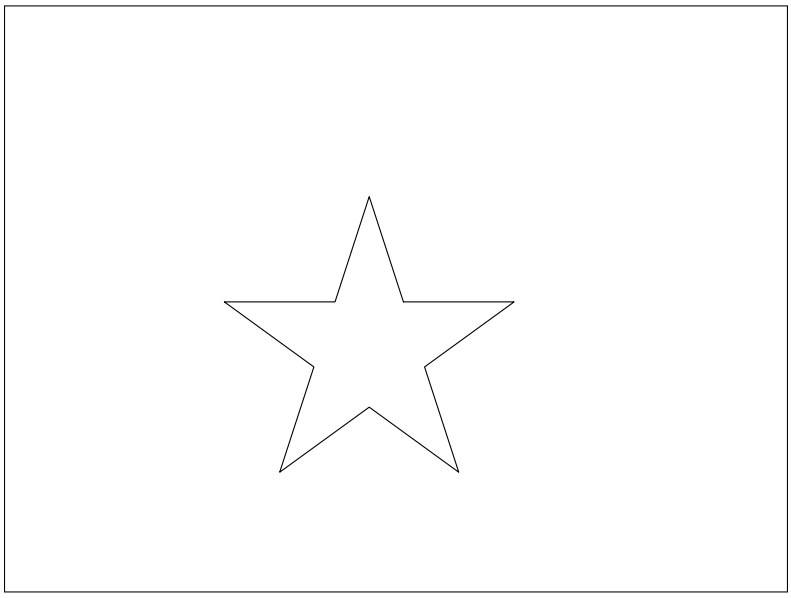
```
9       tom.forward(step_size)
10      tom.right(1)
```

Notice the way the calculation of the step size from the radius is done seperately in **line 5** and **line 6**; these calculations could've put into the loop, directly in **line 9**, as in

```
1  from turtle import *
2
3  tom=Turtle()
4
5  for i in range(0,360):
6      tom.forward(2*3.141592*100/360)
7      tom.right(1)
```

but this is more readible and easier to change.

Along with loops, the other important control statement is the `if`. Consider a programme for drawing a star:

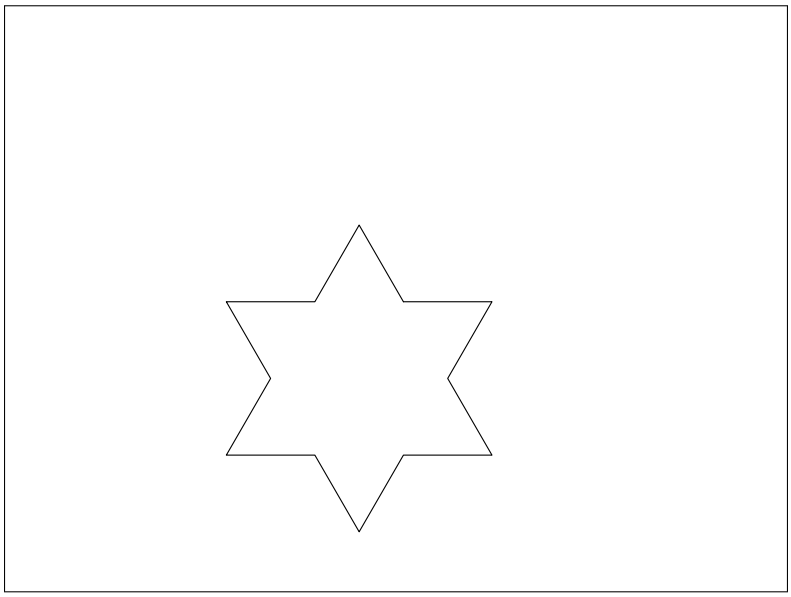


The programme is

```
1  from turtle import *
2
3  tom=Turtle()
4
5  for i in range(0,10):
6      tom.forward(100)
7      if i%2==0:
8          tom.right(144)
9      else:
10         tom.left(72)
```

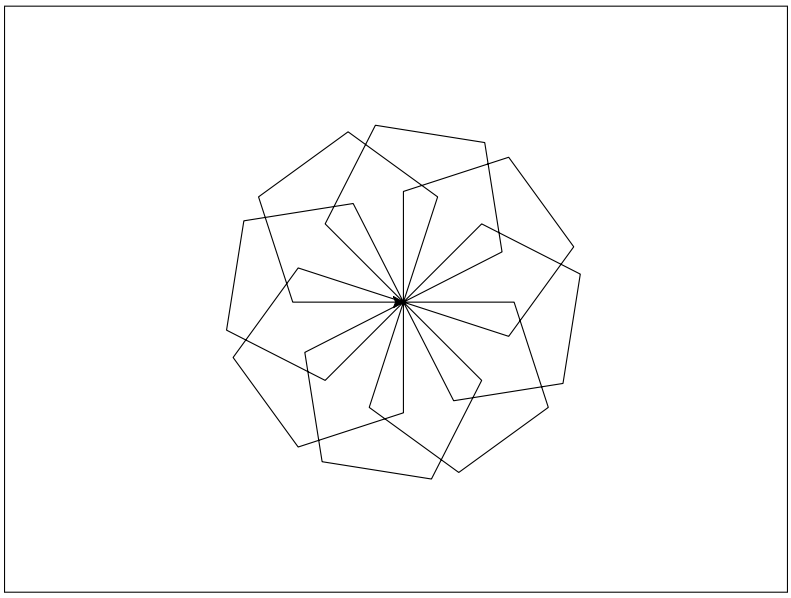
The star has two different angles, every second line either has a right turn of 144° or a left turn of 72°. The `if` on **line 7** decides which to do, basically `i%2` is the remainder you get if you

divide `i` by two, so it is zero if `i` is even and one if it is odd. Now `i%2==0` tests is `i%2` is equal zero, the `==` is 'is equal', the `i%2==0` returns true of false. In the if command, the `if` is followed by a statement, if the statement is true, the programme runs the code belonging to the if, if the statement is false, it skips it. As ever, in Python, the indent is used to show belonging. An if command needn't have an else, but this one does, the code belonging to `else` runs if the statement in the `if` is false. All this means `tom` does one turn for even `i` and another for odd. QUESTION: can you modify the code to give a six-pointed star, the two angles you need are 120° and 60°



Functions

Now imagine you wanted to draw this:



This is made of eight pentagons with an eighth of a full turn between each one; this can be made using the programme:

```
1  from turtle import *
2
3  tom=Turtle()
4
5  repeats=8
6  polygon_sides=5
7
8  for i in range(0,repeats):
9      for j in range(0,polygon_sides):
10         tom.forward(100)
11         tom.right(360.0/polygon_sides)
12     tom.right(360/repeats)
```

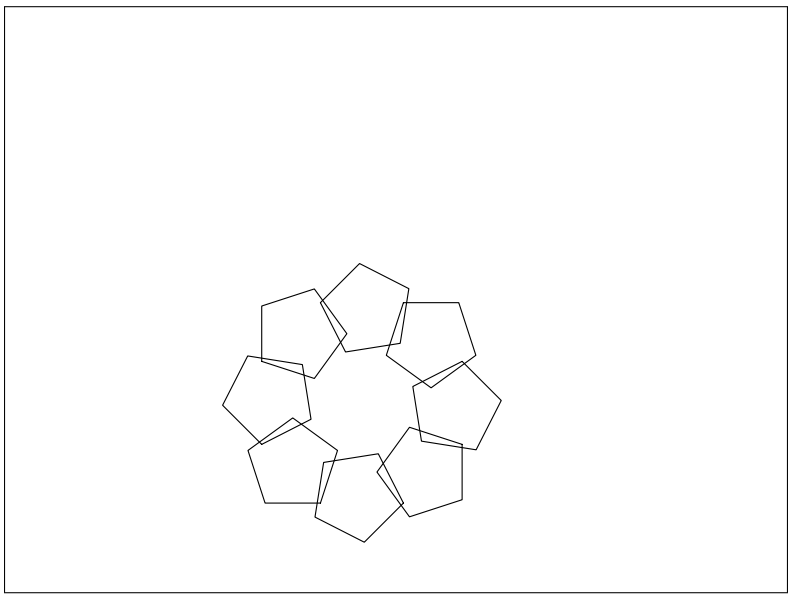
So this programme has two `for` loops, the `j` loop from **line 8** to **line 10** draws the pentagons, the `i` loop repeats that eight times and rotates between pentagon. Notice the two loops need different variables, `i` and `j`, and you can tell what belongs to which loop by the indent, **line 9** and **line 10** belong to the `j` loop so these command are run $40 = 8 \times 5$ times whereas **line 9** and **line 11** are only run eight times.

This programme works, but it fails our readibility and adaptability test; to work out what it does you need to figure your way through the double loop and if you wanted to draw another pentagon later you would have to cut and paste the lines you already had. A much better programme would use a function (`many_polygons.py`):

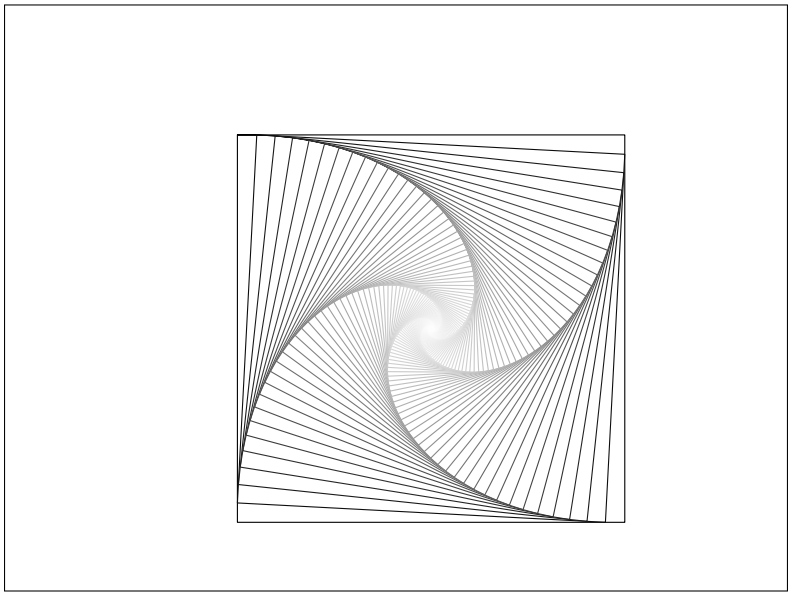
```
1  from turtle import *
2
3  def polygon(n):
4
5      for i in range(0,n):
6          tom.forward(100)
7          tom.right(360.0/n)
8
9  tom=Turtle()
10
11 repeats=8
12 polygon_sides=5
13
14 for i in range(0,repeats):
15     polygon(polygon_sides)
16     tom.right(360.0/repeats)
```

Now in **line 3** to **line 7** we have defined a `function`, the command `def` announces in **line 3** that a function definition is on the way, all the indented lines are the definition. The function is named `polygon`; the brackets after the function name are to give the arguements, values that are sent to the function, in this case the function has one arguement which will give the number of sides of the polygon. Now we have a new command, `polygon(n)`, whenever the command is given, the programme goes up to the function and runs the code belonging to the function, this happens in **line 17**.

QUESTION: can you guess how this was drawn:



The `Turtle` commands `penup()` and `pendown()` lift and drop the turtle's pen, when the pen is up the turtle moves without leaving a line; `hideturtle()` hides the turtle (`many_polygons_extra.py`) A more complicated example (`vanishing_square.py`)



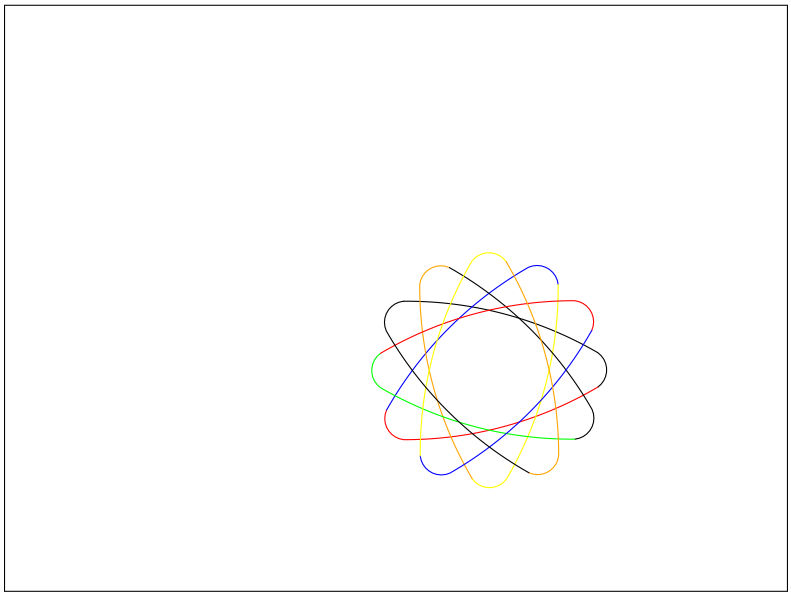
Classes

Throughout this worksheet we've been using the `Turtle` object but haven't considered what we mean by an object; in fact it is a *class*. In *Object Oriented Programming Languages* it is possible to define objects, an object is a some data and some methods that can act on that data. There isn't room here to go into this in detail but here is an example:

```
1 from turtle import *
2
3 class Colors:
```

```
4
5       def __init__(self,color_v):
6           self.color_v=color_v
7           self.current=0
8
9       def change_colour(self,jacob):
10          self.current+=1
11          if self.current==len(self.color_v):
12              self.current=0
13          jacob.color(self.color_v[self.current])
14
15  def curve(arc_length, curvature,steps,jacob):
16      small_arc=arc_length/steps
17      small_curve=curvature/steps
18      for i in range(steps):
19          jacob.forward(small_arc)
20          jacob.right(small_curve)
21
22
23  jacob=Turtle()
24  colors=Colors(['green','red','blue','yellow','orange','black'])
25
26  for i in range(0,12):
27      curve(200,40,30,jacob)
28      curve(40,120,10,jacob)
29      colors.change_colour(jacob)
```

which draws this



The class is defined at the top, starting at **line 3**. This object is for choosing the next colour in a drawing, it remembers the list of colours and the current colour. When a Class is being

defined, the data is called `self`, so `self.color` and `self.current` are variables belonging to the Class. Now an instance of a class has a name, here we make an example of `Colors` at **line 27**, the instance is called `colors` with a small c, it could've been named anything. Now to get the value of `current` for this instance, we would write `colors.current`; the `self` stands in for the name of the object in the general definition. Either way, without going into details, the `__init__` function is a method explaining how to make an instance of `Colours` and `change_colour` is a method for changing the colour of the turtle so that the turtle gets the next colour on the list, if the index gets too big it gets reset so the colours go around and around.