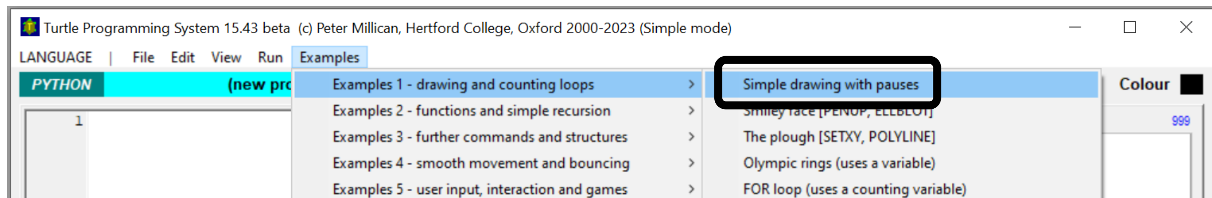
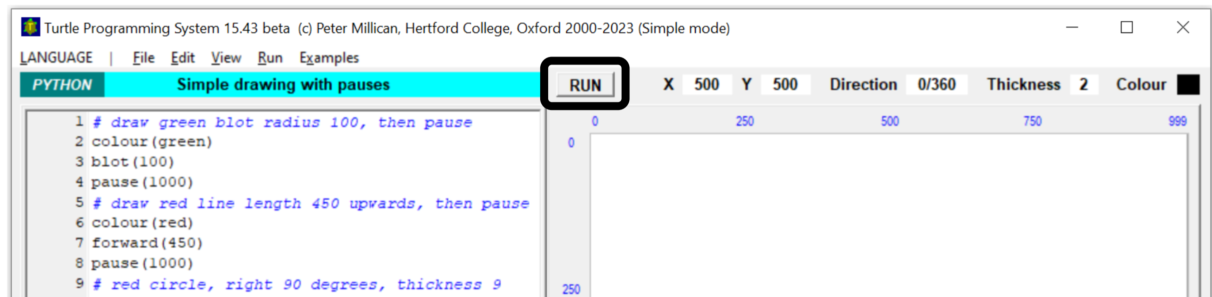
Turtle Python 2 – Spirals and Shapes

1. Introducing Turtle Graphics programming

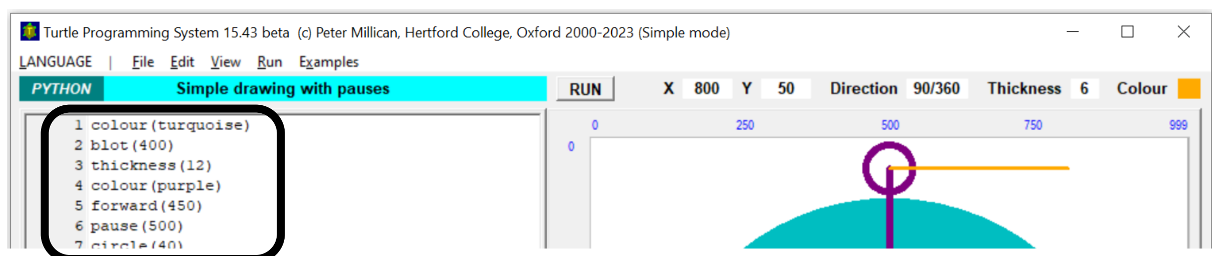
First, click on the “View” menu and select “Simple Mode” to hide the more advanced features of the system. Then click on the “Examples” menu, point to “Examples 1 – drawing and counting loops”, and select the first program on the list (“Simple drawing with pauses”). This contains simple commands for moving around and drawing on the *Canvas*, which is the square area on the right of the screen. This kind of programming is called *Turtle Graphics* because we imagine the moving and drawing being done by an invisible *Turtle*, which starts off in the middle of the Canvas (pointing “north”).



When the program has loaded, click on the **RUN** button and see what happens. Then do it again… Can you work out how the commands are having the effects that you see on the canvas? Notice that the *Turtle* leaves a coloured trail (of some particular colour and thickness) when it moves forward, and that “`pause(1000)`” makes the program pause for 1000 milliseconds (i.e. one second).



The example programs contain some *comments* (after a hash character “#”, and shown in blue italics) – when you’ve read them, they can be removed using “Remove any comments” from the “Edit” menu. Then make some simple changes to the program, like changing the colours, blot sizes, distances, angles, thicknesses, or pause times. Then **RUN** the program again, and repeat…



The full text of this first example program, in its original form, is as follows:

```
# draw green blot radius 100, then pause
colour(green)
blot(100)
pause(1000)
# draw red line length 450 upwards, then pause
colour(red)
forward(450)
pause(1000)
```

```
# red circle, right 90 degrees, thickness 9
circle(20)
right(90)
thickness(9)
# change colour, pause, draw line length 300
colour(blue)
pause(1000)
forward(300)
```

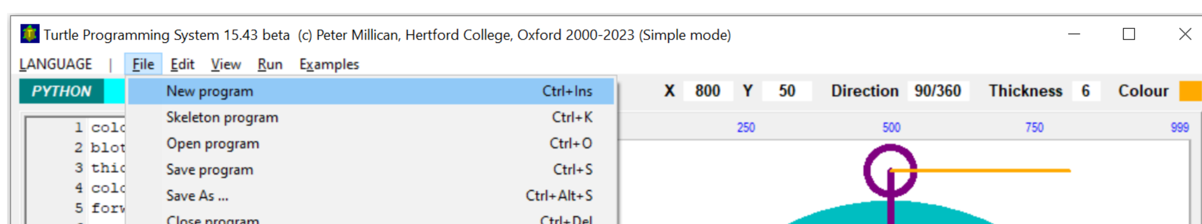
This program includes a number of comments, which begin with a hash (#), and are used to make your program easier to understand (the computer ignores them). It's also common to use a comment in the first line to indicate the name of the program – or the name of the file in which it's stored – but this isn't done with the *Turtle System*'s built-in examples because their name is displayed already.

In this example the first command in the program is `colour(green)`, which specifies the drawing colour as green; then `blot(100)` accordingly draws a green "blot" (i.e. a filled-in circle) of radius 100 around the initial position of the Turtle in the centre of the Canvas. Next is a `pause(1000)` command, which tells the Turtle to pause for 1000 milliseconds (i.e. 1 second). Then another colour command and `forward(450)`, which instructs the Turtle to move forward from its initial position by 450 units. This will leave a red trail, in accordance with the previous `colour(red)` command. After another `pause(1000)`, the Turtle is told to turn right by 90 degrees with `right(90)`, then `thickness(9)` tells it to thicken its pen from the standard 2 unit thickness to 9 units. This means that after `colour(blue)` and another `pause(1000)`, the final command `forward(300)` will draw a thick blue line of 300 units in what is now the direction of the turtle (i.e. horizontal, having turned right by 90 degrees from its original vertical direction).

Having worked through this example, take a look at the next "Examples" program, which draws a smiley face, working out what the commands do and maybe experimenting with variations to confirm your understanding. Having done that, you should be ready to move on to more powerful programs.

2. FOR loops and a spiral pattern

Click on the "File" menu, and then on "New program" (the first entry). Alternatively, you can choose "Skeleton program" (the second entry), which gives you a very short program with a comment header, a variable assignment, and a few example commands already written in.



Type in the following program (or edit the skeleton program accordingly), and then click **RUN** to see what it does.

```
# SpiralA
for count in range(0,130):
    forward(count)
    right(10)
```

The "for" line specifies a *variable* called "count". You can think of this as creating a box labelled "count" which is able to contain just one value. Turtle Python allows a variable to hold either an *integer* (i.e. a whole number) or a *string* (i.e. a sequence of text characters, such as "Turtle!"). In this case, Python can work out for itself that *count* needs to be an *integer* variable from what immediately follows, because the entire line "**for** count **in range**(0,130)**:**" is an instruction to give the *count* variable each value in turn between 0 and 130, and for each value, to perform the two instructions that are indented below it:

```
        forward(count)
        right(10)
```

We think of this as going round and round the "loop" of instructions, with *count*'s value being changed each time. So these two commands – "forward(count)" and "right(10)" – get performed 130 times, first with *count* equal to 0, then 1, then 2, … and finally 129. *Note – importantly – however*, that when the value 130 is reached, the loop terminates immediately – so those two instructions don't actually get carried out with *count* equal to the limit value 130 itself.

The effect of these repeated commands is first to move the *Turtle* forward by *count* units (i.e. by 0 first time, then 1 unit, then 2 units, then 3 units, … and eventually by 129 units), and then (each time) turn the *Turtle* right by 10 degrees. So the *Turtle* moves round a spiral path, getting faster and faster as it goes. If you add a *blot* command to the end of the loop, you'll be able to see more clearly the *Turtle*'s stopping places:

```
for count in range(0,130):
    forward(count)
    right(10)
    blot(5)
```

If you want to stop the *Turtle* from automatically drawing a line as it moves forward from one stopping place to another, insert a line with the command "penup()" before the **for** instruction. This lifts the *Turtle*'s *pen* off the Canvas so that it can move without drawing. To put the *pen* back down on the Canvas at any point, use "pendown()".

Having identified the stopping places, you can now create a simple pattern at each of them, for example a red blot (i.e. a *filled circle*) with a black (*unfilled*) circle around it.

```
# SpiralB
for count in range(0,130):
    forward(count)
    right(10)
    colour(red)        These four lines draw a red blot surrounded
    blot(200)          by a black circle (without moving the Turtle).
    colour(black)      You might like to play with different patterns
    circle(200)        by changing the colours and blot/circle sizes
```

3. Functions and parameters (and the 50 built-in colours)

The *Turtle System* comes with several built-in commands for drawing shapes, like blot, circle, and ellipse, but you can also specify your own commands to make more interesting or complex shapes. To do this, you write a mini-program called a *function* (also known as a *subroutine* or *procedure*), and give it the name of the command you want to define. The following program contains a function called "filledcircle", which draws a red blot of size 200 surrounded by a black circle of size 200:

```
# SpiralC
def filledcircle():
    colour(red)          here is the function, defining
    blot(200)            a new command "filledcircle"
    colour(black)
    circle(200)

for count in range(130):   <----   this is short for "in range(0,130)"
    forward(count)
    right(10)
    filledcircle()   <----   this line calls the function
```

In fact, this program (SpiralC) draws exactly the same pattern as the previous program (SpiralB) – the four lines drawing the red blot surrounded by a black circle have simply been taken out and put inside the `filledcircle` function. This may seem a bit pointless so far, but functions become much more valuable when they are called multiple times in a program, and especially when they are defined with *parameters*, which enable them to be flexible (in the same way as most of the built-in commands). For example, you can change the `filledcircle` function in the last program (SpiralC) as follows:

```
def filledcircle(size):
    colour(red)
    blot(size)
    colour(black)
    circle(size)
```

Now the function is able to draw filled circles of *varying* sizes, so when you call it, you must specify the size you want. To get the previous effect, you need to change the calling line to:

```
filledcircle(200)
```

When this calls the function, it "posts in" the value 200, making *size* equal to 200 – so yet again, the program does exactly the same thing as before. Here is the new version:

```
# SpiralD
def filledcircle(size):
    colour(red)
    blot(size)
    colour(black)
    circle(size)

for count in range(130):
    forward(count)
    right(10)
    filledcircle(200)
```

But now, if you change the value in the last line (e.g. to 100, or 240, or whatever), you'll see that the size of the filled circles changes accordingly. You can also – for example – set the size of the circles to whatever is the current value of *count*, so they get bigger as the spiral grows:

```
filledcircle(count)
```

or else to *half* the current value of count, which gives a different effect by exposing lines drawn as the *Turtle* moves forward (unless you have added the command "`penup()`" as explained earlier):

```
filledcircle(count/2)
```

In what follows, we'll mainly be using this function with just a single parameter (i.e. *size*), but it's worth noting that you can also define a function to have two or more parameters, for example:

```
def filledcircle(size,fillcolour):
    colour(fillcolour)
    blot(size)
    colour(black)
    circle(size)
```

Now the procedure is able to draw filled circles of different sizes *and* different fill colours, so when you call it, you must specify both the size and the colour, for example:

```
filledcircle(200,red)
```

Taking this even further, the version below has three parameters, specifying the size, fill colour, and also border colour (so with this, "`filledcircle(200,red,black)`" gives the original pattern):

```
def filledcircle(size,fcolour,bcolour):
    colour(fcolour)
    blot(size)
    colour(bcolour)
    circle(size)
```

We can now make the pattern more colourful, by taking advantage of the *Turtle System*'s built-in range of 50 colours (green, red, blue, yellow, violet, lime, orange, skyblue, brown, pink etc.).[1] *The easiest way of doing this is described in the next section*, but a more controlled method is to choose colours in sequence using our "count" variable and the "%" (called *modulus*) operator which calculates *remainders*. The expression "count % 10", for example, gives the remainder when count is divided by 10. This will be between 0 and 9, so "count % 10 + 1" will give a value between 1 and 10. The function "rgb" converts each number between 1 and 50 into the corresponding "RGB" (red, green, blue) colour code. So we can now choose one of the first 10 standard colours by using the command:

```
colour(rgb(count % 10 +1))
```

To use this to cycle through the 10 colours with the two-parameter version of the filledcircle function (and blots/circles of size 200), we would call that function with the command:

```
filledcircle(200,rgb(count % 10 +1))
```

4. Random colours, positions, and sizes

The easiest way of making the program more colourful, however, is to use the function randcol(n), which randomly sets the *Turtle*'s colour to one of the first *n* colours defined in the system. So to make the filledcircle function multi-coloured, simply replace "colour(red)" with "randcol(30)" (say), so that it chooses one of 30 colours for the blot instead of printing a red blot every time.

So far, the program has just been drawing filled circles in a spiral pattern, but now you have the filledcircle function, you might want to try using it differently. Suppose, for example, that you want to put filled circles at random points on the Canvas. To do this, you can use the command setxy(x,y) to position the Turtle on the Canvas at coordinates (*x*, *y*), together with the function randrange(x) which generates a random number between *0* and *x–1* inclusive. Since the standard Canvas dimensions are 1000x1000, with coordinates 0 to 999 inclusive, the following command will position the Turtle randomly on the Canvas:

```
setxy(randrange(1000),randrange(1000))
```

You can also achieve the same effect using an alternative random function randint(x,y) – which generates a random number between *x* and *y* inclusive – by replacing "randrange(1000)" with the exactly equivalent "randint(0,999)". Having set your random position, you can then draw a filled circle there, with a random size between 20 and 50 (inclusive), using:

```
filledcircle(randint(20,50))
```

If you replace the body of your earlier FOR loop with these instructions, and count up to 2000 instead of only 130, you get:

```
# ScatterA
def filledcircle(size):
    randcol(30)
    blot(size)
    colour(black)
```

[1] The 50 defined colours can be seen by clicking on the "QuickHelp 1" tab below the Canvas, and choosing "Operators/Colours" at the right hand side. The "QuickHelp 2" tab lists the built-in commands.

```
    circle(size)

for count in range(2000):
    setxy(randrange(1000),randrange(1000))
    filledcircle(randint(20,50))
```

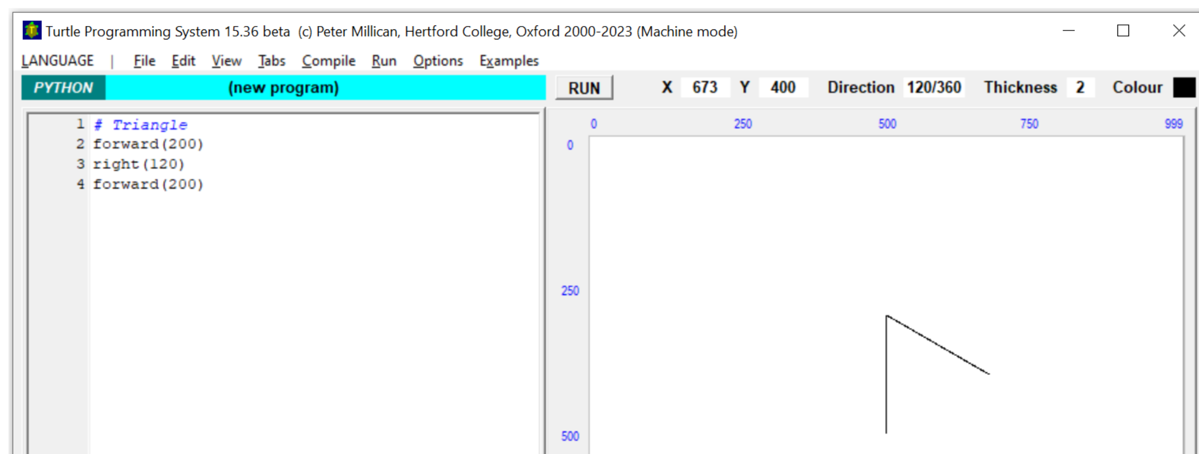
This will display 2000 filled circles, randomly placed, randomly sized (between 20 and 50) and randomly coloured, which should cover nearly all of the Canvas.[2]

5. Drawing a triangle, square, pentagon and hexagon

Now create another new empty program (click on the "File" menu, and then select "New program"), and copy the following code:

```
# Triangle
forward(200)
right(120)
forward(200)
```

Run the program to see what it does. On the Canvas on the right, you should see two sides of an equilateral triangle. The current position of the Turtle is shown by the **X** and **Y** boxes: **X** gives the distance from the left of the canvas, and **Y** gives the distance from the top. The Canvas is 1000 units square, so **X**=500 and **Y**=500 is in the centre. **Direction** gives the direction (in angles) that the Turtle is facing, with 0 meaning North, 90 meaning East, 180 meaning South, and 270 meaning West.



Now try adding two command lines to the end of this program so that it draws a complete equilateral triangle. Can you see how to do this? Can you also add a third new command line (making six in all) so that the *Turtle* ends up pointing in its original direction and in its original place?

Now start another new program, and write a similar program to draw a square, starting like this:

```
# Square
forward(200)
right(90)
...
```

Again make sure that it brings the *Turtle* back to its original position and direction. *(To enable our shape-drawing routines to be used within programs like the original spiral, without messing them up, we want all such routines to leave the Turtle's position and direction unchanged.)*

[2] If you want the circles to go on accumulating forever (or at least, until you HALT the program), then you could replace the FOR loop with an unending WHILE loop. To do this, replace "`for count in range(2000):`" with "`while 1>0:`" (or choose some other condition which is *always* true, such as "`while 1=1:`").

Can you create a similar program to draw a regular pentagon (with 5 sides), and one to draw a regular hexagon (with 6 sides)? You might find it helpful to notice that for the regular triangle you had to turn 120° at each corner, and for the square 90°. So to draw a complete shape, the *Turtle* has to turn enough corners to get back to its original direction. So it has to turn through a total of 360° (a complete circle), which is equal to 3×120° or 4×90°.

As before, if you want your shape outline to take one of the first 10 standard Turtle colours at random, put the command "`randcol(10)`" at the beginning of the function. If you want your shapes to be filled with colour (rather than just outlines), you can use the built-in "`polygon(n)`" command, which fills in the shape made by the last *n* points the *Turtle* has visited, using the current *Turtle* colour.

6. Drawing shapes with a FOR loop

Your program to draw a triangle used the same pair of commands (`forward` and `right`) three times in a row, and your program to draw a hexagon used the same pair of commands six times in a row. Can you see how these programs could be made more efficient by using a FOR loop?

Open a new program again ("File", "New program"). This program is supposed to draw a triangle, as before, but this time using a separate "triangle" function that uses a FOR loop and also takes a "size" parameter. In outline, your program should look like this:

```
# Shapes
def triangle(size):
    for count in range(3):
        commands to draw one side of the triangle
        and turn to be ready to draw the next side

triangle(200)
```

Now rename your "triangle" function "square". Can you edit the commands so that it draws a square instead? Think about how many sides a square has, and the angle needed at each point.

Now rename your "square" function "shape" instead, and give it also a "corners" parameter:

```
def shape(corners,size):
```

Edit the function so that it can draw a shape of any given number of corners (as well as any given size). Test it for "corners" between 3 and 6. Does it work as well when the "corners" value is 7?

We saw above how shapes can be filled with colour using the `polygon` command. But can you find a way of colouring the shape in, and then drawing a black line around it (just as was done with the filled circles)? *[Hint: go to QuickHelp 2, and find the "Intermediate" level shape-drawing command `polyline`.]* You might well prefer the effect this gives when shapes and circles are combined within a single program (as in the next section). Another thing you might like to try is turning the Turtle right (or left) by a random angle before you start drawing the shape, so that not all of them have their first side parallel with each other.

7. Putting it all together

You have now learned how to draw filled circles and various shapes – all in different sizes – and how to arrange patterns in spirals or randomly across the Canvas. So now try using these techniques together, to create an interesting pattern. If you want your program to make a random choice of shapes, you could do something like the following using a *conditional* command (an "if" command which does different things depending on whether some condition is true or false). Here "testnum" is first set randomly between 2 and 6, and serves as the number of sides of the shape, unless it's equal to 2, in which case a circle is drawn instead.

```
testnum:=randint(2,6);
if testnum==2:
    filledcircle(...)      ← if testnum is equal to 2, draw a circle;
else:                        otherwise,
    shape(testnum, ...)    ← draw a shape with testnum sides
```

You could also experiment with drawing *irregular* shapes (e.g. with random vertices, side lengths, or angles), perhaps using variables to record coordinates where necessary. Other potentially useful techniques are illustrated in various *Turtle System* example programs, as listed in the next section.

8. Taking inspiration from *Turtle* example programs

The *Turtle System* has lots of built-in example programs, and a fair number of these exhibit techniques that you could use to enhance your own pattern-creating programs. Here are some of them:

Examples 1 menu – drawing and counting loops

Smiley face	Uses `ellblot` to draw elliptical blots (for the face's eyes).
The plough	Uses `setxy` to place the *Turtle* at precise coordinates on the Canvas.
Spinning triangle	Uses `movexy` to move the *Turtle* by a particular displacement.
Circling circles	Moves out from a central point and back to generate a circular pattern.
Nested FOR loops	Has one FOR loop within another, for more intricate patterns.
Random lines	Draws random lines and then uses `recolour` to fill in the white areas.
Random ellipses	Draws random ellipses and then uses `recolour` to fill in the white areas.

Examples 2 menu – procedures and simple recursion

Polygons	Uses function to draw polygons with a given number of sides and colour.
Procedure with parameter	Uses centring (as with "Circling circles") and turning to give spiral effect.
Resizable face	Uses hierarchical functions for proportional resizing of face and eyes.
Polygons	Draws filled polygons with borders – much like this handout.
Stars	Uses centring and "`polygon`" command (with "`forget`") to draw stars.
Polygon rings	Uses both centring and cycling colours with "%"(*modulus*) operator.
Recursive triangles	Shows how to create triangles within triangles, within triangles etc.

Examples 3 menu – further commands and structures

Cycling colours	Uses the "%" (*modulus*) operator to cycle through 20 colours in order.
Flashlights	Generates a square array of flashing coloured lights.
3D colour effects	Illustrates fine colour gradations with decreasing circles to give 3D effect.

Examples 4 menu – smooth movement and bouncing

Moving ball (both)	Shows how to create movement.
Bouncing ball (both)	Shows how to implement "bouncing" off the edges.
Multiple bouncing balls	Uses *lists* of coordinates and velocities to handle multiple moving balls.
Bouncing triangle	Rotating triangle bounces off edges, changing direction of rotation.
Multiple bouncing shapes	Uses *lists* of coordinates and velocities to handle multiple moving shapes.

Examples 9 menu – self-similarity and chaos

Recursion factory	Provides numerous options for recursive shape patterns.
Recursive tree	Introduces another type of recursive pattern.