LINUX VOICE TUTORIAL

PYTHON DRAWING PRETTY PATTERNS

BEN EVERARD

Use the Python turtle to illustrate loops and recursion, and prove that not all art is quite useless.

WHY DO THIS?

- Gain a better understanding common programming techniques
- Draw pretty pictures
- Win a T-shirt!

There are some programming techniques, like loops and recursion, that we use all the time, almost without thinking. However, sometimes it's hard to really see what's going on. Being able to really visualise what's going on behind the code can help you become a better programmer.

Python comes with a turtle module. It's about as simple as a drawing program can be. It enables you to control a turtle with a pen around the screen. You can tell it to go forwards, turn through various angles, put the pen down or lift it up, and change its colour. In a world where almost everything seems to have OpenGL accelerated graphics, sometimes it's nice to take a step back and look at what you can create with very little. In this tutorial, we're going to look at how we can build complex pictures using just this turtle module and few coding techniques.

Let's start really simply, and just draw a square:

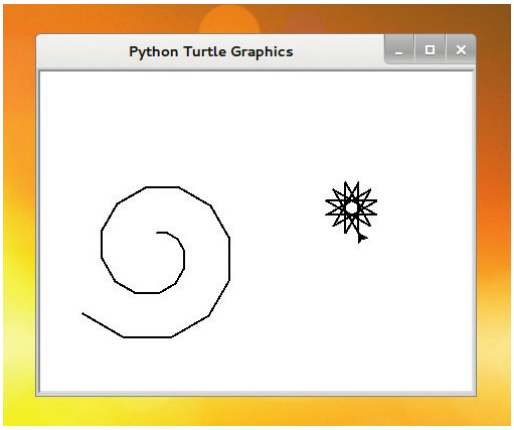
```
import turtle
jonney  = turtle.Turtle()
for i in range(0,4):
  jonney.forward(100)
  jonney.right(90)
turtle.exitonclick()
```

That's all you need. Python will take care of creating the window to display it in. For some reason this author always feels the need to give the turtles anthropomorphised variable names.

You can turn this code into more general polygon drawing code by moving the loop into a function like as follows:

```
import turtle
def draw_polygon(sides, length):
  for i in range(0,sides):
    jonney.forward(length)
    jonney.right(360/sides)
jonney  = turtle.Turtle()
draw_polygon(6,20)
turtle.exitonclick()
```

Because the turtle module only works in whole numbers, this won't work properly for polygons where 360/sides isn't a whole number, but it's good enough for our purposes.



The empty handed painter from your streets is drawing crazy patterns on your screen, with simple Python.

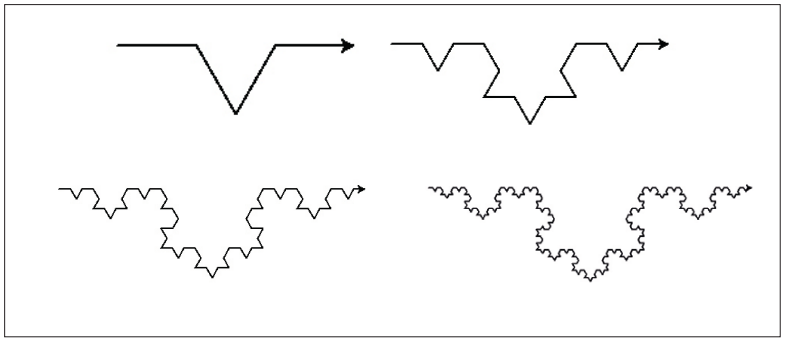


Figure 1. Recursion can quickly build up the number of lines in a fractal as you go to greater depths, so it's best to call **speed(0)** to set it to the fastest speed.

Intensify the artiness

This is an article about creating art from code, and simple polygons aren't very attractive. With a few tweaks, the function can be made a little more artistic:

```
import turtle
def draw_spiral(angle, length_start, length_increase, sides):
  for i in range(0,sides):
    jonney.forward(length_start+(i*length_increase))
    jonney.right(angle)
def draw_petals(length, number):
  for i in range(0, number):
    jonney.forward(length)
    jonney.right(180-(360/number))
jonney  = turtle.Turtle()
draw_spiral(30, 10, 2, 20)
jonney.penup()
jonney.goto(0,200)
jonney.pendown()
draw_petals(50,20)
turtle.exitonclick()
```

You can also vary the colour through the loops. This both helps you see how the images are drawn, and makes the outcomes a little more impressive. We changed the **draw_petals()** function to the following to

fade the lines from blue through purple to red.

```
def draw_petals(length, number):
  red=0.0
  blue=1.0
  for i in range(0, number):
    red=red + (1.0/number)
    blue = blue - (1.0/number)
    jonney.color(red,0.0,blue)
    jonney.forward(length)
    jonney.right(180-(360/number))
turtle.exitonclick()
```

Fractals and recursion

You can get quite artistic using loops to draw shapes, but you can take drawing a stage further using recursion. (Recursion is just when a function calls itself.) You can use this to progressively process all the data in a set, or to draw pretty pictures.

This might sound a little strange if you've never thought of trying to draw with code, but actually, recursion is a really versatile tool in the coder-artist's toolbox, and it's all thanks to a mathematical trick called fractals.

The idea is really simple. You take a simple line shape like figure 1 part 1. Then you replace every line in the shape with a copy of the lineshape. The result of doing this once is figure 1 part 2. However, you can keep doing it as many times as you like, each time you replace every line with a copy of the original line shape. Figure 1 part 3 shows it done a third time, but assuming you had a high enough resolution display (or if you kept zooming in), you could just go on repeating this more and more times.

The code we used to create figure 1 is:

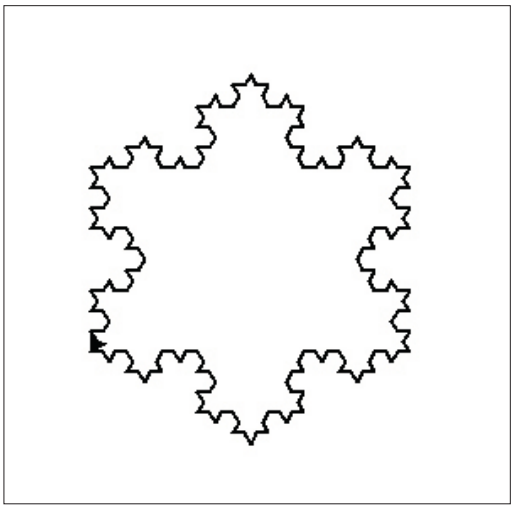
```
import turtle
def draw_fractal(length, depth):
  if depth == 1:
    jonney.forward(length)
  else:
    draw_fractal(length, depth-1)
  jonney.right(60)
  if depth == 1:
    jonney.forward(length)
  else:
    draw_fractal(length, depth-1)
  jonney.left(120)
  if depth == 1:
    jonney.forward(length)
  else:
    draw_fractal(length, depth-1)
  jonney.right(60)
  if depth == 1:
    jonney.forward(length)
  else:
    draw_fractal(length, depth-1)

jonney = turtle.Turtle()
jonney.penup()
jonney.goto(-200,0)
jonney.pendown()
draw_fractal(15,1)
turtle.exitonclick()
```

In this case, we use the **depth** function to limit the number of times the recursion happens, otherwise it could go on indefinitely.

This particular fractal is known as a Kock curve after its creator, Helge von Kock. However, this isn't its best-known form. If you use a triangle for the first iteration, but revert to the line segment for every other iteration, you get a Kock snowflake as shown above.

This is created with the code:

```
import turtle
def draw_snowflake(length, depth):
  draw_fractal(length, depth-1)
  jonney.left(120)
  draw_fractal(length, depth-1)
  jonney.left(120)
  draw_fractal(length, depth-1)
jonney = turtle.Turtle()
jonney.penup()
jonney.goto(-200,0)
jonney.pendown()
draw_snowflake(7,4)
turtle.exitonclick()
```



The Kock snowflake was originally designed by taking a triangle and placing a smaller triangle on every edge, then a smaller triangle on each new edge, etc.

Running Python programs

Python is a very easy programming language to get started in. It's interpreted, which means you don't need to compile the code you've written before you can run it, and it's installed by almost every distribution of Linux by default. What's more, the turtle module is part of the standard Python library, so you don't need to install anything to run the code in this article. Just enter it into a text editor, save the file (a **.py** file extension is usual, but not required), then run it from the command line with:

```
python filename.py
```

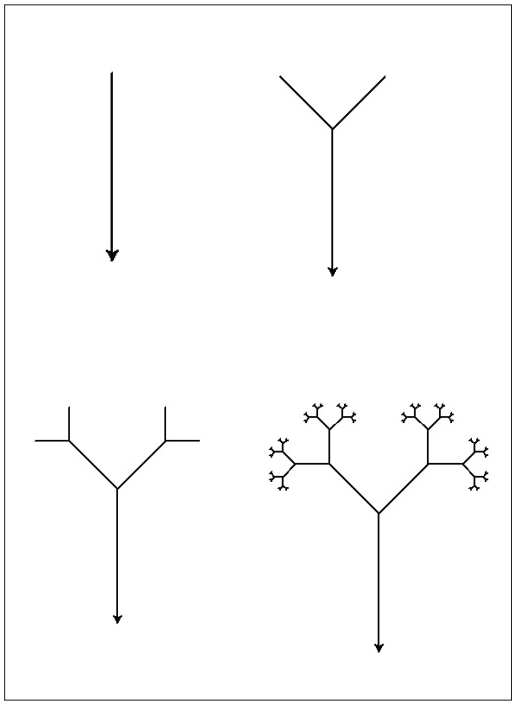
If you don't end your code with the following, then the window will shutdown as soon as it's finished running.

```
turtle.exitonclick()
```

It also includes the **fractal()** function from the previous example.

You can use this same method of recursion in another way. For example, you can use it to continually add lines in a particular place. For example, you can generate a tree by drawing a 'Y' shape, then continually adding smaller 'Y' shapes to the end of each branch. See right for how this works out. In part 1 to 3, the depth is 1 to 3 respectively. Part 4 has a depth of 7. This was generated with the code:

```
import turtle

def draw_tree(length, depth):
  jonney.forward(length)
  if depth > 1:
    jonney.left(45)
    draw_tree(length/2, depth-1)
    jonney.left(90)
    draw_tree(length/2, depth-1)
    jonney.right(135)
  jonney.right(180)
  jonney.forward(length)

jonney = turtle.Turtle()
jonney.penup()
jonney.goto(0,-100)
jonney.pendown()
jonney.left(90)
jonney.speed(0)
draw_tree(160,7)

turtle.exitonclick()
```

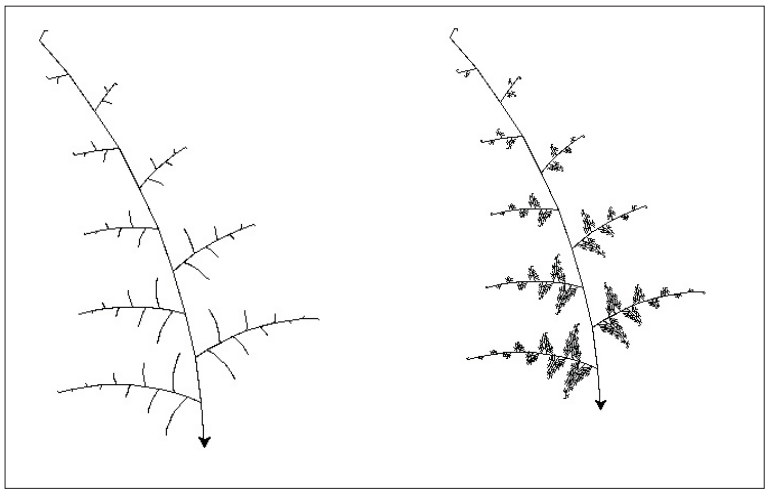


With fractals that get smaller and smaller, you'll quickly reach the limit of the resolution of the screen.

You can extend this. Instead of drawing a Y shape, you can expand a herring bone in the same way (see below). Instead of creating the classic tree shape, this creates a fern-like drawing. The code for this is:

```
import turtle
def floor(x,y):
  if x > y:
    return x
  return y

def draw_fern(length1, angle1, length2, angle2, depth):
  flip = 1
  for i in range (0, length2):
    jonney.left(angle2)
    jonney.forward(length1)
    if depth > 1:
      jonney.left(angle1*flip)

      draw_fern(floor((length1/3)-(i/2), 1), angle1*flip,
floor(length2-i,1), angle2*flip, depth-1)
      jonney.left(180-angle1*flip)
    flip = flip * -1

  jonney.left(180)

  for i in range(0, length2):
      jonney.forward(length1)
      jonney.right(angle2)

jonney = turtle.Turtle()
jonney.penup()
jonney.goto(0,-100)
jonney.pendown()
jonney.left(90)
jonney.speed(0)
draw_fern(40,60,8,4,1)

turtle.exitonclick()
```

These ferns are depths 3 and 5. We've spaced it out to make it easier to see what's going on, but you can change the parameters to create more realistic plants.



This particular code is very sensitive to the parameters you give it. You also can customise the fractal by changing the way the line lengths are passed to the next level of recursion, or by progressively increasing the angle so that the fern starts to curl towards the end.

If you're feeling really adventurous, you could write a program that flips from the tree recursion to the fern recursion at a certain depth. You can develop fractals like this using different shapes. The key is to make sure that, at the end of each run of the function, you return the turtle to the same place it started from.

Fantastic Mr Fractal

There are loads of possible fractals you could draw, and a quick web search will pull some up. One thing to remember when programming fractals like the fern and the tree is to make sure you always finish the fuction at the same physical location the turtle started it. Otherwise it'll end up chaotic.

What we've covered in this tutorial may seem a bit pointless, flippant even, but we've used exactly the same programming techniques that are used in normal software. By learning how to exploit them to draw shapes, you hone your knowledge of how to structure code, and this can only make you a better programmer whatever language you use.

There are also a few cases where fractals themselves are useful to programmers – for example, in creating complex terrain in video games.

Ben Everard is the co-author of *Learning Python with Raspberry Pi*, soon to be published by Wiley. He's also pretty good at turning foraged fruit into alcohol.

Competition time

You've seen how, by using iteration and loops, you can create complex drawings with very little code. In this competition, we're going to put this to the test. The challenge is to create a Python program that uses the turtle module to draw something. I have to be able to output the code, and the winner will be the person that creates what we think is the best piece of art. To put your coding skills to the test, we're going to give you a limit of 100 lines of Python – no more.

You may have noticed that we haven't really tried to keep our code short in this tutorial and quite a few of the functions we've used can be shortened if needed.

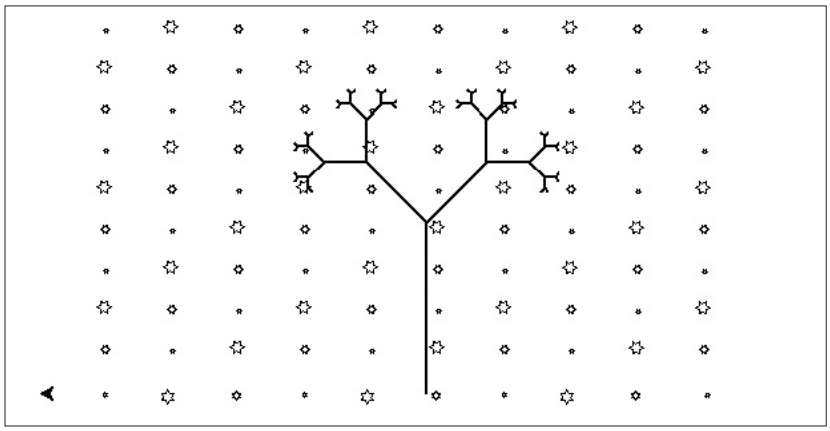
The rules are:

- Using only the turtle module, and no more than 100 lines of Python (2 or 3, your choice), you must draw a picture.
- You may use any of the techniques here, or any others you invent, copy, steal or otherwise come across.
- The only module you can use is the turtle module. Any other import lines will be deleted.
- Pictures will be judged on artistic merit. The judge's decision is final.
- Lines of comments are allowed (and encouraged) and won't be included in the 100 line count. Feel free to add information to your program, and all programs must be licenced under an OSI approved open source licence. Preferably the GPLv3, though you can use a different one should you so choose.
- In order to enter, send your entries to ben@linuxvoice.com by 1 May 2014.
- The winner will receive an exclusive Linux Voice competition winner's T-shirt. These are not, and will not, be available in the shops. The only way to get one is to submit a winning entry to a Linux Voice competition.
- The artwork produced can be in any category. Abstract, still life, impressionism, cubeism. Sculpture is probably out, but otherwise, anything goes as long as it looks good.
- Up to three entries per person will be accepted.
- You don't have to buy a copy of Linux Voice to be eligible. Feel free to pass on the competition details to non LV readers, and details will be posted on **www.linuxvoice.com**.
- Turtles don't have to be called Jonney, and you don't have to limit yourself to a single turtle.

To give you an example of what we're looking for, take a look at figure 5. This was created with the following code (the functions for the **tree()** and **snowflake()** as as they're given earlier in the tutorial, and not repeated here, though if you wish to use them in your example, you WILL have to include them in your 100 lines). Although this picture doesn't make it look like it, the judge does like colour, and garish entries will be looked upon favourably.

```
jonney = turtle.Turtle()
jonney.speed(0)
jonney.left(90)
jonney.width(2)
draw_tree(128,6)
jonney.width(1)
jonney.penup()
jonney.goto(210,0)
for i in range(0, 10):
  for j in range(0,10):
    jonney.pendown()
    draw_snowflake((i+j)%3+1,2)
    jonney.penup()
    jonney.setheading(90)
    jonney.forward(30)
  jonney.setheading(270)
  jonney.forward(300)
  jonney.setheading(180)
  jonney.forward(50)
turtle.exitonclick()
```

Good luck, have fun, and remember that while good artists borrow, great artisits steal!



It may be spring now, but the long, dark winter is still vivid in our memories. Help us forget it with some uplifting artwork, and give yourself the opportunity to win clothing!