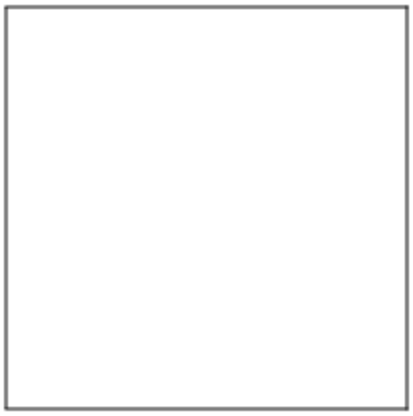
Cubes

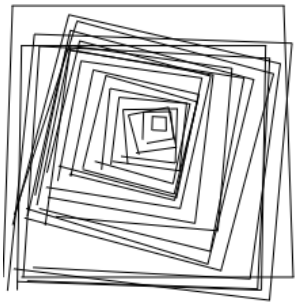
This is a lovely example of how we can start simple, and, by solving one small problem after another, eventually end up with a toolkit we can use for a whole range of nifty visuals.

Drawing a square

This is one of the first exercises anyone playing with the Turtle module will tackle. There are, of course, many ways to do it, but one of the most elegant involves a little **for** loop. Think carefully about what state your turtle will be in at the end of the program, and try to plan for the various possible adaptations that may be made later. Write out a basic program outline that will draw a square of any given size from any starting point, leaving the turtle in the same position as it started, in the same orientation (don't worry about trying to write it in Python yet, pseudo-code is fine for the planning stage):

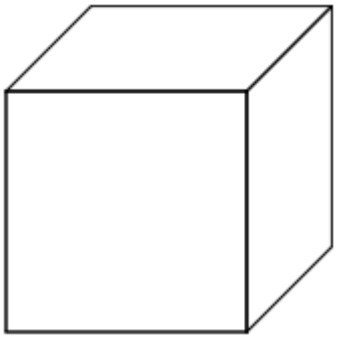


Too easy? Build in some random variation, so the corners aren't quite $90°$, and make a loop so you can draw loads…

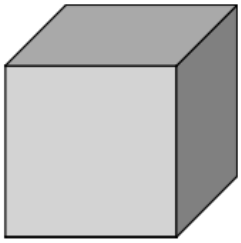


Drawing a cube

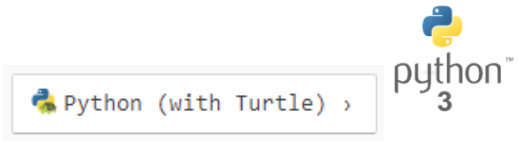
One of the simplest ways to draw a cube is with a square front, as shown below. You might want to experiment with how far back it should go, but I find that half the square width works pretty well. *Write some pseudo-code for drawing one of these.*



Simple? Now think about how you might fill each face with a different colour… Don't forget to finish where you start.



Coding it with Python



A few useful commands to remember:

You will use the *Python (with Turtle)* link in Repl. Don't forget to **import turtle** at the start.

- Use **t = turtle.Turtle()** at the start to create your drawing object.
- Use **t.fd(30)** to make him move forward 30 pixels, and **t.bk(20)** to go back 20.
- Use **t.lt(90)** or **t.rt(135)** to make him turn left or right (anti-clockwise or clockwise).
- You can use **t.speed(0)** to set animation speed to maximum, and **t.tracer(10)** to speed up animation still further (if you don't see anything appear, disable this line, as it may be too fast to animate your drawing).
- Use **t.ht()** to hide the turtle (it can still draw) and **t.pu()** and **t.pd()** to lift the pen up or put it back down.
- Use **t.fillcolor("grey")** or **t.fillcolor("lightblue")** to set the fill colour (the colour the turtle will use if you invoke the next commands..)
- Use **t.begin_fill()** and **t.end_fill()** to start and stop the filling process. The shape produced will be filled with the specified fill colour. *Note: use the US spelling color!*

Debugging tips

If something unexpected happens, first disable **t.tracer(10)**, and reduce the turtle speed so that you can watch him follow the algorithm step by step. This is also a good way to determine if you are using the most efficient method. Make sure each bit of your code works the way you expect it to, and as you develop more pieces, you'll probably find you need to go back and 'refactor' previous functions to make them fit more nicely.

Making your cube

You can figure this bit out on your own. It took me 40 lines of code, but there was nothing very clever in it – just instructions for following the lines. Note that you can't draw a cube without retracing at least some of your lines – to see why, count the number of edges meeting at each corner, and look up the Seven Bridges problem.

Making a row

Once you have a self-contained function that can reliably draw you a cube, you can use it to quickly construct more sophisticated shapes:

```
1   import turtle, random
2
3   t = turtle.Turtle()
4   t.speed(0)
5   t.tracer(10)
6   t.ht()
7   t.pu()
8
9   def cube(size):···
51
52  def row(x, size):
53      for i in range(x):
54          cube(size)
55          t.fd(size)
56      t.bk(size * x)
```

The **row** function is very straightforward: you tell it how long a row you want by defining **x** (this is the number of cubes you want to draw), and the size of the cubes. It then initiates a loop, drawing a cube (following your **cube** function instructions) and then moving forwards just far enough to start the next cube right beside the last.

After the loop has terminated, the turtle goes back to the start (not essential, but consistent with the cube code, and it may make life easier when we extend even further.

Making a wall

This is a very similar idea to the row. In fact, if a row is a loop of repeated squares, you can equally well think of a wall as a loop of repeated rows.

```
58  def wall(x, y, size):
59      for i in range(y):
60          row(x, size)
61          t.lt(90)
62          t.fd(size)
63          t.rt(90)
64      t.rt(90)
65      t.fd(size * y)
66      t.lt(90)
```

There's a bit more careful adjustment built in here: at the end of each row, we automatically return to the start of the row, but we also need to move *up* to start the next row directly above.

At the end, move down to the starting point.

Making a block

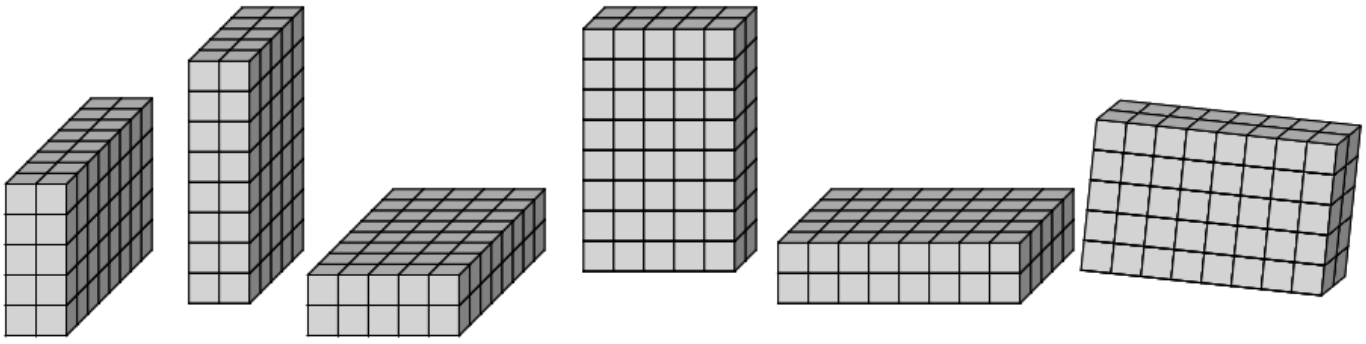
Just as a row was made by repeating squares, and the wall by repeating rows, we can build a complete 3D block by repeating walls.

```
68  def block(x, y, z, size):
69      for i in range(z):
70          wall(x, y, size)
71          t.lt(45)
72          t.bk(size // 2)
73          t.rt(45)
74      t.lt(45)
75      t.fd(size // 2 * z)
76      t.rt(45)
```

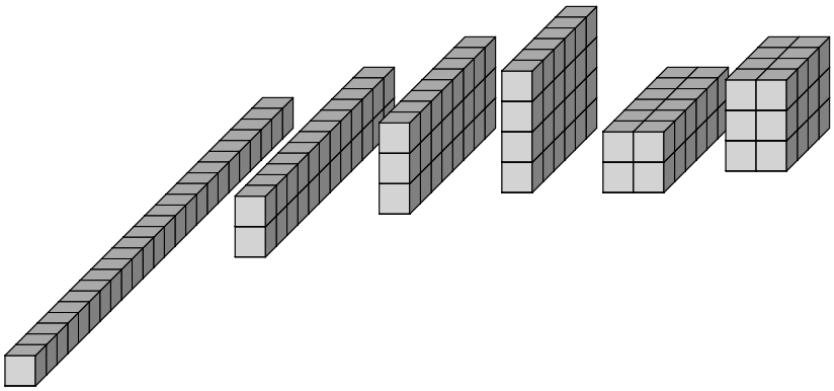
This code preserves the convention of x, y and z that I used for the row and the wall, and this makes it easier to see how it is simply an extension of those functions. Notice how we need to move along the third dimension (or 'diagonally'!) this time to get back to where we started.

Now what?

Test out your program by asking for various different sized cubes, rows, walls and blocks. In fact, now you've created the block function, you can ignore all the others (want a cube? It's a 1 by 1 by 1 block. Need a wall? Try a 1 by 4 by 5 block. In fact, by messing with the order of the three dimension values, you can make your walls have any orientation:



(use **t.rt(6)** before drawing to make it slightly off: our functions preserve orientation).

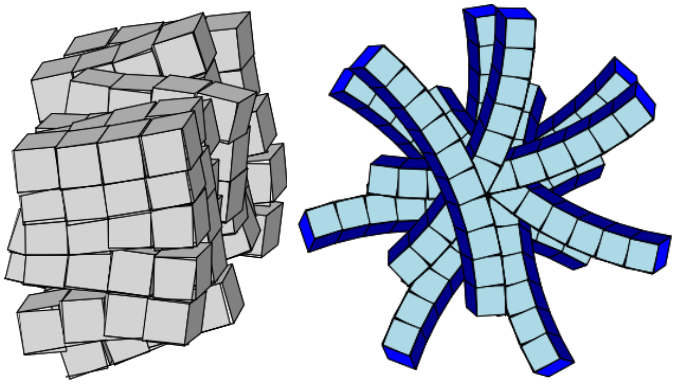


<- All the cuboids with integer side lengths and volume 24.

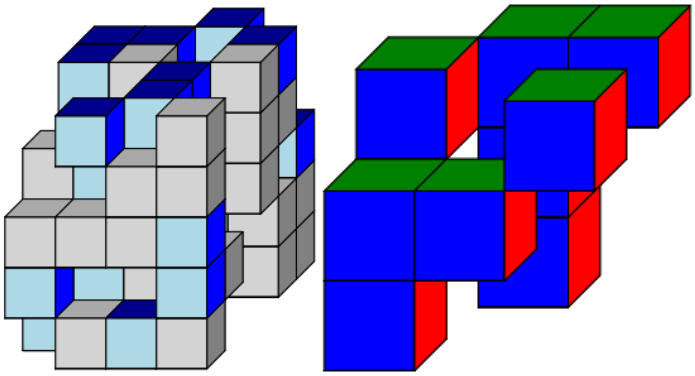
Can you make a program to find the factor triplets of a number? Brute force will work just fine for small numbers.

A few more extension ideas

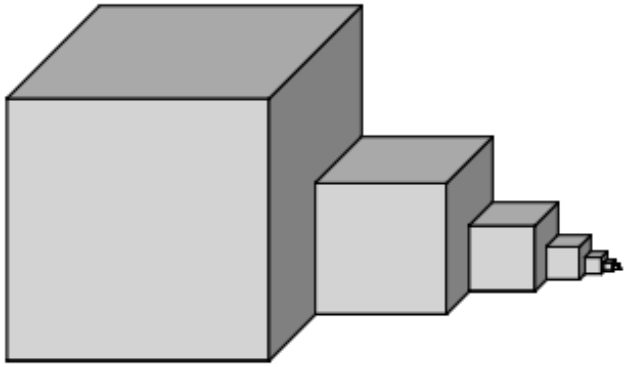
Just like we messed with the square's defining angles, can you muck about with some of the key features of your cube and see what interesting patterns you can create?

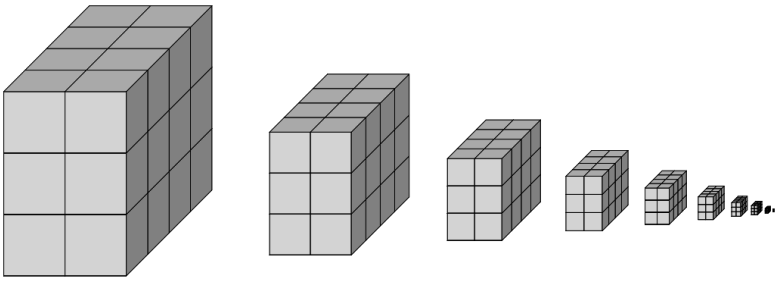


How about using the **random** module to selectively colour (or even missing out) random cubes?



Can you build in some kind of fractal pattern, by repeatedly making the size of your cubes smaller?





What about a representation of integers based on how well they factorise?