These tasks start off with some that are easier to solve using for loops, and then migrate to ones that you'll need a while loop to solve effectively, and then a tricky challenge problem.

Task 1:

Write some code that prints out a `thing` the number of `times` specified (one `thing` per line). It doesn't matter what the `thing` is – it can be an `int`, a `string`, or other, more complicated types we haven't learned about yet.

Task 2:

Write some code that takes a list of numbers, which we'll call `nums_list` (very creative, I know), and for each number, prints out:

- The square root of the number,
- The number itself,
- The number, squared,
- The number, cubed

All separated by commas.

For instance if our list contains [1, 2], the your code should print:
1.0, 1, 1, 1
1.41421356237, 2, 4, 8

Hint – you'll want to use the exponentiation operator -- `**`. `2**4` is 2`4`, and you can use **.5 to take a square root.

You can initialise your list with something like the following:

```
nums_list = [1,2,3,4,5]
```

We'll talk more about lists later next week.

Task 3:

Use a `while` loop to prompt the user to enter in a value between 1 and 10, as many times as it takes for them to get it right.

Task 4:

Suppose you've decided to ~~invest~~ gamble your money in Bitcoin. The price of Bitcoin is highly volatile, and changes randomly each day. Each day, the price of Bitcoin can go up by as much as 10%, or down by as much as 20% (so, an initial "investment" of $100 can be worth up to $110, or as little as $80, after one day).

Use a while loop to figure out how long it will take before you've lost at least half of your money.

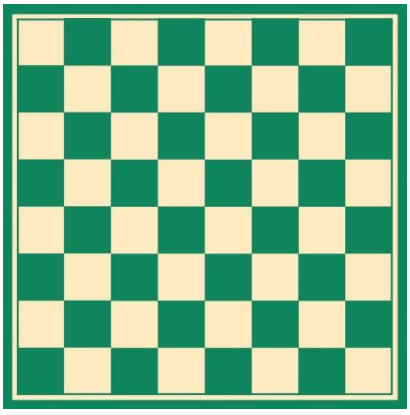
Hint: The following expression can be used to calculate the new value of your investment after each day, relative to its prior value:

```
(randrange(31) + 80)/100
```

(for example, a value of .88 means your investment is worth 88% of what it was worth; a value of 1.06 means it's worth 106% of what it was (6% more than previous)). To make this work, you'll need to run **`from random import randrange`**. This is an example of importing a *function* from a *module* built in to Python. We'll learn more about both of those next week.

Task 5:

The game of chess is played on an eight-by-eight board, where cells alternate white and black, across both rows, and columns (strictly, white-and-black is not required – there are boards that are white-and-green, white-and-blue, etc). For example, here's a white-and-green board:



Here, we'll write some Python code that will generate something very much like a chess board – we'll print out an eight-by-eight grid, and put a `w` or `b` in each cell (or, if you prefer, `w` or `g`). The output might look something like this:

```
w b w b w b w b
b w b w b w b w
w b w b w b w b
b w b w b w b w
w b w b w b w b
b w b w b w b w
w b w b w b w b
b w b w b w b w
```

Note – this task is *tricky!* You'll need to pull together the material we talked about conditionals earlier, and the material on (doubly-nested) loops. To make the board come out right, you'll also (probably) want to tell `print` to not terminate every line automatically. You can do this as follows:

```
print("some text goes here", end="")
```

This will terminate each line of printed text with the empty string, instead of a customary newline.