Ninety nine bottles of beer on the wall.

A study in repetition.

Topics covered: Iteration (loops), functions, printing output, problem solving.

Introduction

There is a well-known drinking song which goes:

> *Ninety nine* bottles of beer on the wall,
>
> *ninety nine* bottles of beer.
>
> If you take one down and pass it around,
>
> there'd be *ninety eight* bottles of beer on the wall.

> *Ninety eight* bottles of beer on the wall,
>
> *ninety eight* bottles of beer.
>
> If you take one down and pass it around,
>
> there'd be *ninety seven* bottles of beer on the wall.

It keeps going like that, counting down the bottles of beer on the wall, until only one remains. At that point we reach the end of the song:

> *One bottle* of beer on the wall,
>
> *one bottle* of beer.
>
> If you take it down and pass it around,
>
> there'd be *no more* bottles of beer on the wall!

If you don't like beer you can always use another kind of drink. There are other variations of the ending - one of them suggests that we "go to the store and buy some more", thus taking us back to the start again, producing a song that never ends. There is a Wikipedia page which mentions other variations, and related songs: http://en.wikipedia.org/wiki/99_bottles_of_beer.

Given that the song has a repetitive structure, it is a popular pastime for people to write programs to generate its lyrics. There is even a web site which collects such programs in different programming languages: http://www.99-bottles-of-beer.net/ - you might be surprised by how many different versions there are! We will use Python in this tutorial.

A simple way to implement the program is to write all the lyrics in one big string, and then print it out, but this approach has a couple of shortcomings:

1. It is very tedious and error prone to write out the whole song as one long string, and it doesn't take advantage of the fact that there is a lot of repetition in the lyrics.
2. It is difficult to generate alternative versions of the song, such as versions which start with a different number of bottles on the wall.

It is therefore desirable that we try to avoid redundancy in our code, and at the same time try to keep the program as general as possible. Indeed, good programmers always try hard to achieve these two goals. The benefits are that we tend to produce programs which are easier to maintain, and are usable in a wider variety of situations.

Where to start?

Finding a place to start writing a program can be rather tricky. Like most difficult things, we get better with practice, but that is cold comfort when you are a beginner. It is tempting to dive right in and try to solve the whole thing at once. Invariably this leads to despair - we get tangled up in a mess of details and often give up in frustration. Even if we do manage to get the program to work, it is likely to be badly designed and hard to understand, resulting in "spaghetti code''.

One of the most important skills we need as programmers is the ability to take a complex problem and break it up into manageable pieces. The idea is to start solving smaller problems first, and then work our way towards the final task. As the saying goes ''don't bite off more than you can chew!". There are many benefits to writing programs this way. We are more likely to make progress and not get lost in too many details. Also, solutions to simpler problems often provide key insights into the requirements of the more difficult extensions. From little things, big things grow.

The first thing to do is take a pen and paper (or computer equivalent) and jot down the basic things that the program has to do. Here are some questions you might like to ask yourself:

- Can the whole task be broken up into stages, what are they?
- What kinds of data does the program need to manipulate, and how should it be represented?
- What is the most common task that must be done by the program? If it is repetitive, what are the steps and how does it start and stop?
- What are the special cases that must be handled, such as errors, or unusual circumstances?
- What features of the programming language and computer might be needed? For example, a repetitive task might need a "for" loop. Some data might need to be stored in a file. We might need strings to print to the user. There might be librarys of code that provide ready-made solutions to some of the things we need to do.

Broadly speaking, there are two key things that the bottles-of-beer program has to do:

1. It has to count down the number of bottles.
2. For each bottle count it has to print the corresponding verse.

It makes sense then that we should get the counting sorted out first, and then extend it to print the verses.

Counting down the bottles

Initially there are ninety nine bottles, and we want to count down to one. A reasonable first task is to try to print the numbers out as we count down. We'll need a variable to keep track of the counter. Whenever we use a new variable in our program we have to give it a name. There are some things to consider when we decide on the name:

- We should choose a name which doesn't clash with another use of the same name in the program.
- The name should be meaningful to other programmers, that is, the name should convey the purpose of the variable.
- We must follow the syntax rules of the programming language (only certain names are valid, and programming languages are notoriously picky).

Programmers have all sorts of theories about how to name variables. No doubt you will form your own opinion on the matter in due course. If you end up programming for a company they might even force you to follow their own scheme! For now we will choose something which is meaningful but not too long, so let's make it "count".

Count should start at 99, so we should initialise it to that value:

```
count = 99
```

This line of code declares that the variable called "count" exists, and that its initial value is the number 99. For the sake of testing our program it might be wise to choose a smaller starting value, otherwise we are going to get a lot of output from the program. Fortunately, we can choose any value for count that we like. We could set it to five like so:

```
count = 5
```

Task 1. Write a loop which prints all the numbers from the initial value of count down to one. Each number should be printed on a separate line. You can assume that count is initially greater than zero. For instance, if count starts with the value of 5, the loop should print:

```
5
4
3
2
1
```

Here are some questions you should answer after you have completed this task:

1. Python provides two kinds of looping syntax, "for" loops and "while" loops. Does it make a difference which one you use in this task?

2. What happens in your code if count is initially less than zero?
3. What is the value of count after your loop has finished?
4. What will happen if you execute the loop again without re-setting the value of count?

It is a bit tedious to have to re-set the value of count each time we run the loop. A solution to this problem is to put the loop inside a function, and make `count` a parameter of the function. This will allow us to pass the initial value of count as an argument when we call the function. For instance, if the function is called `count_down`, we can pass it an initial value of 5 like so:

```
count_down(5)
```

Task 2. Write a function called `count_down` which takes a single parameter called `count`, and executes the loop from task 1. For instance, `count_down(3)` should produce the following output:

```
3
2
1
```

Whereas, `count_down(6)` should produce this output:

```
6
5
4
3
2
1
```

Here are some questions you should answer after you have completed this task:

1. What happens if your pass a negative argument to count_down?
2. If necessary, can you modify your solution so that it does not produce any output if the argument is negative?

The important part of `count_down` is the loop. Note that we solved how to write the loop first, before we wrote the function. This highlights a useful aspect of the Python interpreter, which is that we can experiment with fragments of programs before we try to put all the pieces together. It's a lot like playing with Lego; don't be afraid to tinker.

Printing a single verse

Now that we've managed to count down, it is time to try printing the verses. You would have noticed that all the verses, except for the "one bottle" case, are instances of this template:

X bottles of beer on the wall,
X bottles of beer.
If you take one down and pass it around,
there'd be **X-1** bottles of beer on the wall.

The only thing that changes is the value of **X**.

A complication is that, ideally, the value of **X** should be printed in words, not numerals. For instance, 99 should appear in the verse as "ninety nine", and so on. Also there is the nuisance of having to deal with the capitalisation of the first letter in the word when it appears at the start of a sentence.

Printing numbers as words is rather tricky. Taking the advice from earlier, it would be sensible at this point to simplify the problem. As a first approximation, we should just use numerals when the numbers are printed in the verse. Later, once this is working correctly, we can come back and turn the numerals into words.

The last verse is the song is special, since it is a variation of the above template. Let us handle the common case first, where the number of bottles is greater than one.

Task 3. Write a function called `common_verse`, which takes a single numerical parameter and prints a string representing a "common" verse from the song. For instance, `common_verse(12)` should print:

```
12 bottles of beer on the wall,
12 bottles of beer.
If you take one down and pass it around,
there'd be 11 bottles of beer on the wall.
```

You might have noticed another slightly awkward aspect of the song due to the pluralisation of the word "bottle". Whenever we have more than one bottle, English grammar dictates that "bottle" should have a letter 's' on the end, making it "bottles of beer". But when there is just one bottle, we should not pluralise. This causes particular problems for the second last verse of the song. Most likely your version of common_verse would generate the following as the last line of the second last verse:

there'd be 1 <u>bottles</u> of beer on the wall.

when it ought to be:

there'd be 1 <u>bottle</u> of beer on the wall.

Task 4. If necessary, fix your version of `common_verse` so that it handles the pluralisation of "bottle" correctly.

One skill you will need as a programmer is to pay close attention to detail. Little things like pluralisation are important to users of programs, so it is important to get it right in the final product.

The last verse in the song is slightly different than the common verses; it only deals with one bottle, and it is only sung once - therefore it is constant. We can write a function to generate this verse as a special case, but, unlike the common case, it does not need a parameter because it never changes.

Task 5. Write a function called `last_verse`, which has no parameters, and prints the last verse of the song. For instance, `last_verse()` should print:

```
One bottle of beer on the wall,
one bottle of beer.
If you take it down and pass it around,
there'd be no more bottles of beer on the wall!
```

Now we can write a more general function which can handle the printing of any verse in the song, by choosing between `common_verse` and `last_verse` depending on the number of bottles that we have.

Task 6. Write a function called `verse`, which takes a single numerical parameter representing the number of bottles on the wall, and prints out the appropriate verse from the song. The function should call either `common_verse` or `last_verse` depending on the value of its argument. For instance, `verse(44)` should call `common_verse(44)`, but `verse(1)` should call `last_verse()`.

Actually, the last verse and the common verses do share a fair amount of structure, and it seems that we haven't quite managed to exploit that in the program. If you are keen you might like to consider alternative ways of writing the verse function that takes more advantage of the shared structure of *all* the verses.

Printing the whole song

Now it is time to combine the counting with the verse printing, so that we can print the whole song from start to end. Now that we've done most of the hard work, this should not be too difficult.

Task 7. Modify the `count_down` function from task 2 so that it prints all the verses of the song. For instance, `count_down(99)` should produce the whole song starting with 99 bottles of beer, all the way down to 1 bottle of beer. You should use the verse function for printing out each verse. Make sure you print an empty line in between each of the verses.

How did you print the blank line in between the verses? One minor aesthetic detail to consider concerns last verse; unlike all the previous verses, it should not be followed by a blank line. If necessary can you modify your code to avoid the extra blank line?

One of the nice aspects of `count_down` is that it takes a numerical parameter indicating the initial number of bottles. Therefore we can easily generate customised versions of the song simply by calling the function with different numerical arguments. Make sure you test your function on a variety of argument values, and check the output for correctness.

Extension: the icing on the cake

If you've finished task 7 successfully you might want to try your hand at something more challenging. As noted earlier, the program is not be truly complete unless the numbers of bottles are printed as words instead of numerals. To achieve this, you'll probably want to write a function which turns numbers into words, but be warned, there are lots of special cases to consider (such as the teens). Another question is how high do you want to go? Can you support numbers in the hundreds? Thousands? Obviously whatever you choose as the upper limit must be greater than or equal to 99.

Task 8. Modify the count_down function from task 7 so that the numbers of bottles are printed in words. Also make sure that proper capitalisation is used for the start of sentences. You may need to define other functions as part of your solution.

Conclusion

In this tutorial we have observed the following things:

1. Loops can be used for counting.
2. Functions can be used to make code reusable and more general.
3. Difficult problems can be solved by splitting them into simpler sub-problems.
4. Some problems share common structure, and we can exploit that in programs using abstraction.