

Teaching guide: Programming concepts (Iteration)

This resource will help with understanding iteration in programming concepts. It supports Section 3.2.2 of our current GCSE Computer Science specification. We have designed this resource to address the following learning aims:

- Understand why we use iteration in programs.
- Be able to follow the condition-controlled loops of REPEAT-UNTIL and WHILE.
- Know how to rewrite a REPEAT-UNTIL loop using a WHILE loop.
- Be able to follow the count-controlled FOR loop.
- Be able to nest loops within each other.

Repeat-until loops

The following guessing game introduces the use of repeat-until loops:

1. The setter thinks of a number between 1 and 100
2. The guesser has a guess at this number
3. Is this guess the same as the setter's number?
 - a. YES then the game ends
 - b. NO then the setter tells the guesser if it is too high or too low
4. Go back to step 2

Variables can now be introduced at steps 1 and 2:

1. `setters_number` ← `USERINPUT`
2. `guess` ← `USERINPUT`
3. Is this guess the same as the setter's number?
 - a. YES then the game ends
 - b. NO then setter tells the guesser if it is too high or too low
4. Go back to step 2

A Boolean expression can also be introduced to replace line 3 and parts a and b reworded, so we end up with:

1. `setters_number` ← `USERINPUT`
2. `guess` ← `USERINPUT`
3. `setters_number = guess`
 - a. `True` then the game ends
 - b. `False` then the setter tells the guesser if it is too high or too low
4. Go back to step 2

A problem now occurs because at step 4 we have to loop back to step 2. The technique of looping back to a previous place in a program is incredibly common and there are different ways to accomplish it –this section will cover: **REPEAT-UNTIL**.

The use of **REPEAT-UNTIL** is common in everyday English, for example:

- ‘repeat after me until you understand’
- ‘repeat the back flip until you can do it’
- ‘repeat your times tables until you know them off by heart’.

All of these share the properties that you will be doing the action at least once and thereafter for an unknown number of times until some condition is `True`.

The aim in the program is exactly this: enter a guess at least once and repeat this until the guess is correct (which could be 6 more times, 10 more times, no more times etc). So the entire game can now be rewritten as this:

```
setters_number ← USERINPUT
REPEAT
    guess ← USERINPUT
    # tell the guesser if it was too high or too low
UNTIL setters_number = guess
```

The logic for determining whether the guess was higher or lower than the setters number has not been included yet and so there is a slight untidiness with the comment – this is addressed in the ‘Teaching guide – Programming concepts (Selection).

This game could be extended to include a counter that increases after every guess so at the end of the game the program tells the players how many guesses have been made. We now know all of the techniques to do this in our program. We must:

- create an integer variable and assign it a starting value
- make sure this variable goes up by after every guess
- output this variable once the correct guess has been made.

When the program starts the user has had no guesses and so the starting value should be 0. This statement and also the statement for the variable going up (or incrementing) can be added into our program to get:

```
setters_number ← USERINPUT
num_of_guesses ← 0
REPEAT
    guess ← USERINPUT
    num_of_guesses ← num_of_guesses + 1
    # tell the guesser if it was too high or too low
UNTIL setters_number = guess
OUTPUT num_of_guesses
```

The last line in this program obviously outputs the number of guesses made.

While loops

Another kind of iteration is the **WHILE** loop. Again, the use of **WHILE** is common in real life:

- 'while it is raining stay indoors'
- 'while you are asleep you cannot play badminton'
- 'while you are less than 1 metre tall you are not allowed on a ride'

Each of these suggests that although you can or cannot do something at the moment, at some point in the future this will change.

WHILE differs from **REPEAT-UNTIL** because the Boolean condition occurs at the start of the loop. For example, if the first example above was written into a **WHILE** loop it could be written as:

```
is_it_raining ← USERINPUT
WHILE is_it_raining = 'yes'
    OUTPUT 'stay indoors'
    is_it_raining ← USERINPUT
ENDWHILE
OUTPUT 'go outdoors'
```

The condition is evaluated at the beginning of the loop and, if it is true, the code in the loop executes. The program then loops back and the condition is evaluated again. At the point where the Boolean condition evaluates to **False** the code in the loop is skipped and the program executes from after the **ENDWHILE**.

In the game, while the guess is not the same as the number then the guesser has another go and this will continue until the guess is correct. So the Boolean expression in the game can be rephrased to `setters_number ≠ guess` to get:

```
setters_number ← USERINPUT
guess ← USERINPUT
```

```

WHILE setters_number ≠ guess
  # tell the guesser whether it is too high or too low
  guess ← USERINPUT
ENDWHILE

```

Logically this is no different from the game that used `REPEAT-UNTIL` as they will both do the same thing although the `WHILE` loop must have a guess made before the loop starts as well as within the loop.

A `REPEAT-UNTIL` loop can always be rewritten as a `WHILE` loop using this pattern:

```

REPEAT
  # block of code
UNTIL the Boolean condition = True

```

is equivalent to:

```

# block of code
WHILE the Boolean condition = False
  # the same block of code
ENDWHILE

```

Rewriting a `WHILE` loop as a `REPEAT-UNTIL` is sometimes possible using the 'opposite' pattern where the true Boolean condition in the `WHILE` becomes the false Boolean condition in the `REPEAT-UNTIL` although this will not always hold and it may require use of selection as well which is covered in the next section.

While or repeat-until?

It might look from this that a `REPEAT-UNTIL` is always a better choice than a `WHILE` loop because there is less code to write but remember that the code in the `REPEAT-UNTIL` loop must happen at least once and this is not always desirable.

A program to check the temperature of a fridge could use a thermometer to take the temperature and play a warning sound repeatedly until the temperature is below 4°C. As a `WHILE` loop this would be:

```

temp ← THERMOMETER_INPUT
WHILE temp ≥ 4
  OUTPUT warning sound
  temp ← THERMOMETER_INPUT
ENDWHILE

```

However if `REPEAT-UNTIL` had been used then the program would play a warning sound once even if the temperature was acceptable:

```

REPEAT
  OUTPUT warning sound

```

```
temp ← THERMOMETER_INPUT
UNTIL temp < 4
```

It is always possible to write a **REPEAT-UNTIL** loop by using a **WHILE** and if Python is used it will be necessary to do this as the Python language does not include **REPEAT-UNTIL** (or the related **DO-WHILE**) loops.

Count-controlled loops

The two previous examples are loops that are controlled by a Boolean expression and are known as indefinite iteration because they can continue looping an indefinite number of times. In contrast to this, a **FOR** loop that iterates for a specific number of times can also be used.

Most languages have at least one form of **FOR** loop although they vary greatly. Our version of a **FOR** loop will introduce a new integer variable that is given a value to start at and a value to stop at with the assumption that it will increase by one after every loop. For example, this short program will output the string 'hello' ten times:

```
FOR i ← 1 TO 10
  OUTPUT 'hello'
ENDFOR
```

At the first iteration of this loop the value of **i** will be **1**, through the next iteration it will be **2** and so on until it has the value **10** on the last iteration. This program doesn't use the variable **i** within the body of the loop and only uses it as a way to 'count' how many iterations are made but we are free to use the variable within the **FOR** loop just as you would any other variable.

If you wanted to create a program that outputs the 3 times table from 1x3 to 12x3 you could make good use of the variable in the **FOR** loop by studying the pattern in the expressions. Without a loop, our program would look something like this:

```
OUTPUT 1*3
OUTPUT 2*3
OUTPUT 3*3
# etc for 4*3 to 11*3
OUTPUT 12*3
```

Every line of code in this program is the same except for the integer immediately before the ***** symbol. This integer starts at **1** and increases by **1** every time until it reaches **12** and so the program could be rewritten using a **FOR** loop:

```
FOR i ← 1 TO 12
  OUTPUT i*3
ENDFOR
```

This has the obvious advantage that it accomplishes the same as the previous program but using significantly fewer lines of code. It is also easier to spot mistakes and easier to extend (such as changing this to the four times table instead).

Use of loops

Knowing when to use a loop is a skill that improves with practice but you should definitely be thinking about using a loop if any of these are true:

- you see repeating patterns in the code
- you find yourself repeating lines or blocks of code
- you are copying and pasting code underneath or above itself

Sometimes knowing when to use a loop is less clear cut. If you were asked to find out the sum of the positive integers up to and including 100 (i.e. $1 + 2 + 3 + 4 + \dots + 99 + 100$) then you could write out a very long arithmetic expression or you could also use a FOR loop:

```
result ← 0
FOR n ← 1 TO 100
    result ← result + n
ENDFOR
OUTPUT result
```

Nested iteration

Iteration is one of the three main components of the style of programming called structured programming used in these guides, the others are sequence – quite simply the order in which your instructions are executed – and selection which will be covered in the next section. Programmers use these simple building blocks to construct complex programs and one technique that can be used to do this is placing loops within each other.

Below is a three times table program:

```
FOR i ← 1 TO 12
    OUTPUT i*3
ENDFOR
```

How could this program be extended so that instead of just printing the three times table, it prints all of the times tables from one to ten? First look at the pattern. To output the one times table you need to change it to:

```
FOR i ← 1 TO 12
    OUTPUT i*1
ENDFOR
```

To output the two times table you need to change it to:

```
FOR i ← 1 TO 12
  OUTPUT i*2
ENDFOR
```

And so on up to and including the ten times table:

```
FOR i ← 1 TO 12
  OUTPUT i*10
ENDFOR
```

The only thing that changes in this program is the integer to the right of the * symbol which starts at 1 and ends at 10 and goes up by 1 every time. This is exactly the time when a FOR loop should be used – when the number of iterations is known (in this case 10).

So what about this implementation:

```
FOR i ← 1 TO 10
  FOR i ← 1 TO 12
    OUTPUT i*i
  ENDFOR
ENDFOR
```

There is a big problem here in the use of the variable *i*. In the middle line `OUTPUT i*i` it is not clear which *i* should be used – the *i* from the inner FOR loop or the *i* from the outer FOR loop? In most programming languages this will either be shown as an error, or, possibly, the program will run but both occurrences of *i* will refer to most recent time that program has been assigned (which in this case is the inner FOR loop, that is, the one in which *i* has most recently been declared).

If our program is to run as we want we will need to use another variable name in either one of our FOR loops. A corrected program is:

```
FOR i ← 1 TO 10
  FOR j ← 1 TO 12
    OUTPUT j*i
  ENDFOR
ENDFOR
```

An even better example that achieves the same output but uses clearer variable identifiers would be:

```
FOR table ← 1 TO 10
  FOR multiplier ← 1 TO 12
    OUTPUT multiplier*table
  ENDFOR
ENDFOR
```

There are many conventions used in programming that allow your code to be easily read and understood. One such convention is the use of one-letter variable identifiers when the variable is used as a counter; these typically start at *i*, then *j*, *k* and so on as needed.

The way that the FOR loops are nested is very important. To see why, we can first create a trace table of a shorter version of this program that outputs the 1, 2 and 3 times tables each up to five times:

```
FOR i ← 1 TO 3
  FOR j ← 1 TO 5
    OUTPUT j*i
  ENDFOR
ENDFOR
```

In our trace table for this program we will also include a column that shows the output at each iteration

i	j	OUTPUT
1	1	1
	2	2
	3	3
	4	4
	5	5
2	1	2
	2	4
	3	6
	4	8
	5	10
3	1	3
	2	6
	3	9
	4	12
	5	15

And now compare this with a very similar looking program in which the nesting of the FOR loops has been swapped:

```
FOR j ← 1 TO 5
  FOR i ← 1 TO 3
    OUTPUT j*i
  ENDFOR
ENDFOR
```

The trace table for this algorithm (again including the output) is:

j	i	OUTPUT
1	1	1
	2	2
	3	3
2	1	2
	2	4
	3	6
3	1	3
	2	6
	3	9
4	1	4
	2	8
	3	12
5	1	5
	2	10
	3	15

This algorithm still outputs multiplication, but it computes the first 3x of 5 different tables, instead of the first 5x of 3 different tables.

For another example that uses considerable nesting, consider a park bench with 4 places on it. Each place can only ever be empty or occupied by a person (no sharing, no sitting on laps, no lying over three spaces etc). How many different ways are there for this bench to accommodate people?

We could start by drawing some solutions to this problem:

Seat 4	Seat 3	Seat 2	Seat 1
Empty	Empty	Empty	Empty
Empty	Empty	Empty	Occupied
Empty	Empty	Occupied	Empty
Empty	Empty	Occupied	Occupied
...

And we could continue, but it looks like this is going to be quite a lengthy table so let's try to spot a pattern.

Every seat can be empty or occupied – that is two different states. To use a FOR loop to count how many states that would be for one seat we could write the following program where 0 could represent empty and 1 could represent occupied:

```
count ← 0
FOR seat1 ← 0 TO 1
  count ← count + 1
ENDFOR
```

The next seat also has exactly two states, but for each one of its states, the other seat can also be empty or occupied like so (ignoring the other two seats):

Seat 4	Seat 3	Seat 2	Seat 1
		Empty	Empty
		Empty	Occupied
		Occupied	Empty
		Occupied	Occupied

We could extend our program to:

```
count ← 0
FOR seat2 ← 0 TO 1
  FOR seat1 ← 0 TO 1
    count ← count + 1
  ENDFOR
ENDFOR
```

Seat 3 also has exactly two states as does seat 4, and so our finished program looks like this:

```
count ← 0
FOR seat4 ← 0 TO 1
  FOR seat3 ← 0 TO 1
    FOR seat2 ← 0 TO 1
      FOR seat1 ← 0 TO 1
        count ← count + 1
      ENDFOR
    ENDFOR
  ENDFOR
ENDFOR
```

When this program is run the final value of count tells us how many different permutations there are (the answer is 16). We could extend the program slightly so that within the innermost nested FOR loop we output the values of seat4, seat3, seat2 and seat1 on one line which would give us the complete set of permutations: More on this series of numbers is covered in the Teaching guide - data representation

Seat4	Seat3	Seat2	Seat1
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

All loops can be nested, not just FOR loops. If we look again at the guessing game using the REPEAT-UNTIL loop it is obvious that this game will run once and then stop, but what if we amend this so that it gives the user the option to play again or stop?

The program itself needs to be repeated an unknown number of times (the players could decide to only play it once or they may choose to play it 10 times) but it will be played at least once – because it repeats an unknown number of times but happens at least once we could nest this program inside a REPEAT-UNTIL loop:

```

REPEAT
  setters_number ← USERINPUT
  num_of_guesses ← 0
  REPEAT
    guess ← USERINPUT
    num_of_guesses ← num_of_guesses + 1
    # tell the guesser if it was too high or too low
  UNTIL setters_number = guess
  OUTPUT num_of_guesses

```

```
    play_again ← USERINPUT
UNTIL play_again = 'no'
```

The program prompts the user to enter whether they would like to play again at the end of the game and will play another game if the user answers anything except the string 'no'.

So far **FOR** loops have been nested inside other **FOR** loops and **REPEAT-UNTIL** loops inside other **REPEAT-UNTIL** loops but any loop can be nested inside any other. The game could just as well have been nested within a **WHILE** loop with some small adjustments to ensure it happens at least once:

```
play_again ← 'yes'
WHILE play_again ≠ 'no'
    setters_number ← USERINPUT
    num_of_guesses ← 0
    REPEAT
        guess ← USERINPUT
        num_of_guesses ← num_of_guesses + 1
        # tell the guesser if it was too high or too low
    UNTIL setters_number = guess
    OUTPUT num_of_guesses
    play_again ← USERINPUT
ENDWHILE
```

Infinite loops

We have looked at two main types of iteration in this section: definite iteration (**FOR** loops) where the number of iterations is fixed, and indefinite iteration (**WHILE** and **REPEAT-UNTIL** loops) where it is unknown how many times the loop will repeat.

When you create the condition for indefinite loops you need to be careful that the truthfulness of this condition will eventually change. For example, if you have a **WHILE** loop that loops if $a < 5$, you should ensure that at some point in your program $a < 5$ becomes **False**. If this never happens then you will end up with an infinite loop that is almost always undesirable.

You can illustrate this problem with a small program that adds ten to a number until it reaches a certain point:

```
n ← 1
WHILE n ≠ 30
    n ← n + 10
ENDWHILE
```

The intention is for this loop to stop (or terminate) when the number gets to 30. If you create a trace table to follow the execution we see that there is an error:

n
1
11
21
31
41
51
...

The trace table contains the first six values of n but you could carry on forever – you have inadvertently made an infinite loop. If the intention of the programmer was to stop the loop before it got to a value greater than or equal to 30 then a corrected version could be:

```
n ← 1
WHILE n < 30
  n ← n + 10
ENDWHILE
```

For which the trace table is:

n
1
11
21
31

(Note that the final value of n is 31 not 21 – the final value of n is the value that makes $n < 30$ false.)