Teaching guide: Subroutines

This resource will help with understanding subroutines. It supports Sections 3.2.2 and 3.2.10 of our current GCSE Computer Science specification (8520). The guide is designed to address the following learning aims:

- Understand how using subroutines can benefit developers.
- Understand that a subroutine is a block of code that takes zero or more inputs and possibly returns a value.
- Appreciate that variables declared within subroutines are normally only accessible within those functions.

Defining subroutines

The following code will find the greatest element in an array called `my_array` that contains at least one element:

```
greatest ← my_array[0]
FOR i ← 1 TO LEN(my_array)-1
   IF my_array[i] > greatest THEN
      greatest ← my_array[i]
   ENDIF
ENDFOR
```

Finding the greatest element in an array is a common programming task and, although the code is only 6 lines long, it would be inefficient to type it out in full every time we needed it. We could copy and paste the code but we are relying on the array being called `my_array` every time. This isn't sensible and copying and pasting code is rarely good practice.

It would be much simpler if we could write the code just once and then write something like:

```
get the maximum element in my_array
```

and replace the identifier `my_array` with whatever array we are using. Better still we could reduce this to the following code which we will take to mean exactly the same thing:

```
max(my_array)
```

This expression is made up of an identifier that refers to some code that is already written (`max`) and an input (`my_array`) and the whole expression evaluates to a value. What is the type of the expression max(my_array)? This entirely depends what my_array is an array of. If my_array is an array of integers then the greatest of those will be an integer. If my_array is an array of strings then max(my_array) will have type string and so on. Some languages will not allow this and so you would need to have different versions of max for different types of arrays. For example, given the array:

```
new_array ← [5, 2, 7, 5, 2, 9, 4]
```

The following expression would evaluate to the value 9:

```
max(new_array)
```

We can use the expression max(new_array) anywhere we would be able to use any other integer like 9, as in these examples:

```
ten ← max(new_array) + 1

IF max(new_array) ≠ 9 THEN
   OUTPUT 'this will not happen'
ELSE
   OUTPUT 'but this will'
ENDIF

eighteen ← max(new_array) + max(new_array)

counter ← 1
WHILE counter < max(new_array)
   OUTPUT counter
   counter ← counter + 2
ENDWHILE

big_number ← 10^max(new_array)
```

We are using max(new_array) as a shorthand to refer to code already written. To do this we need to give our original code a label and, as we are calling it max, that would be a sensible name to use:

```
label: max
greatest ← my_array[0]
FOR i ← 1 TO LEN(my_array)-1
   IF my_array[i] > greatest THEN
      greatest ← my_array[i]
   ENDIF
ENDFOR
```

This code uses an array called my_array and we want to be able to say, 'run the code called max but use the array I'm giving you instead of my_array'. That is, whenever the array my_array is used in the code, use the input array instead. We could refine our code to be:

```
label: max
copy the input array to: my_array
greatest ← my_array[0]
FOR i ← 1 TO LEN(my_array)-1
   IF my_array[i] > greatest THEN
      greatest ← my_array[i]
   ENDIF
ENDFOR
```

And lastly we want our code to report what the greatest element in our array is. As this is the value that is returned at the end of our block of code we'll use the word return:

```
label: max
copy the input array to: my_array
greatest ← my_array[0]
FOR i ← 1 TO LEN(my_array)-1
   IF my_array[i] > greatest THEN
      greatest ← my_array[i]
   ENDIF
ENDFOR
return the value: greatest
```

This mix of pseudo-code and wordy instructions doesn't have the same feel as the other code we have written so far. We give the name subroutine to any block of code that is written out once to be used as many times as the programmer wishes in other parts of code. The syntax we will use to create a subroutine and do exactly what has been written above is:

```
SUBROUTINE max(my_array)
   greatest ← my_array[0]
   FOR i ← 1 TO LEN(my_array)-1
      IF my_array[i] > greatest THEN
         greatest ← my_array[i]
      ENDIF
   ENDFOR
   RETURN greatest
ENDSUBROUTINE
```

Which has the meaning that:

- whenever the word max is used in code followed by opening brackets, the name of an array and closing brackets, the program will jump to this subroutine
- the array we actually want to use in our subroutine (the argument) is copied to the array called my_array (the parameter).

- it will execute the code in the subroutine and will return (give back) the value of the greatest element in that array.

When you “call” or invoke a subroutine you are using it. So, for example, by using the code `subr(A)` we have called the subroutine `subr` with the input `A`.

Examples:

```
SUBROUTINE addTwo(n)
   result ← n + 2
   RETURN result
ENDSUBROUTINE

SUBROUTINE triple(m)
   result ← m * 3
   RETURN result
ENDSUBROUTINE

a ← 4
b ← addTwo(a)              # addTwo(a) evaluates to 6
c ← addTwo(3)              # addTwo(3) evaluates to 5
a ← triple(c)              # triple(5) evaluates to 15
b ← triple(b) + addTwo(c)  # triple(b) evaluates to 18
                           # addTwo(c) evaluates to 7
                           # so b is assigned the value 25
```

Parameters

So far, all of the subroutines looked at only use one input variable, called a parameter, but this is far from true in real programming where subroutines can have any number of parameters (programming languages technically do limit the number but it is normally very high – Python allows over 250). The following are all examples of subroutines that take multiple parameters:

```
# calculate the hypotenuse of a right-angled triangle
SUBROUTINE hypotenuse(a, b)
   c_squared = a^2 + b^2
   c = SQRT(c_squared)
   RETURN c
ENDSUBROUTINE
```

`SQRT` is the name we have already used to return the positive square root of a number.

This subroutine takes two inputs (as they need to be squared and the results added together we can infer that they are numbers – either reals or integers) and returns the square root of the sum of their squares.

```
# check whether an element is in an array
SUBROUTINE is_member(el, arr)
   FOR i ← 0 TO LEN(arr)-1
      IF arr[i] = el THEN
         # return True as soon as a match is made
         RETURN True
      ENDIF
   ENDFOR
   # return False if no match has been found
   RETURN False
ENDSUBROUTINE
```

The `is_member` subroutine iterates over every element in the input array `arr` and returns `True` immediately if any of them are equal to the first parameter `el`. No further execution occurs in the subroutine when a value has been returned. If no match is found after the whole array is iterated over then the subroutine will return `False`.

```
# clip the values in an array to a given range
SUBROUTINE clip(low_limit, high_limit, arr)
   FOR i ← 0 TO LEN(arr)-1
      IF arr[i] < low_limit THEN
         arr[i] ← low_limit
      ELSE IF arr[i] > high_limit THEN
         arr[i] ← high_limit
      ENDIF
   ENDFOR
   RETURN arr
ENDSUBROUTINE
```

The `clip` subroutine takes three parameters, the third of which is an array. This subroutine also iterates over the entire array and checks whether each element is below the value `low_limit` (and if so sets that element to be the `low_limit`) or whether it is above the value `high_limit` (and likewise sets that element to `high_limit`). Once this is complete it returns the potentially amended array.

```
# examples of calling these subroutines
hyp ← hypotenuse(3.0, 4.0)
found ← is_member(hyp, [3.0, 4.0, 5.0])
signal ← [1.1, 3.5, 0.9, 2.5, 1.5, 4.2]
clipped_signal ← clip(1.0, 3.0, signal)
```

Subroutines can also take no inputs. The following subroutine displays a message to the user and then gets user input:

```
SUBROUTINE get_input()
   OUTPUT 'Enter a number between 1 and 100'
   number ← USERINPUT
   RETURN number
ENDSUBROUTINE
```

Subroutines are at their most useful when they can be used in many similar parts of the code. For instance, the get_input example is only useful in its current form for getting a number between 1 and 100. If we wanted a subroutine to get a number between 1 and 50 then we would have to write out a new subroutine. This problem is easily avoided using the lower and upper boundaries as parameters:

```
SUBROUTINE get_input(lower, upper)
   OUTPUT 'Enter a number between'
   OUTPUT lower
   OUTPUT 'and'
   OUTPUT upper
   number ← USERINPUT
   RETURN number
ENDSUBROUTINE
```

Subroutines without return values

When a subroutine is called it has the type of its return value (eg hypotenuse has a real number return type and is_member has a Boolean return type). Any subroutine that returns a value can also be called a function. Some subroutines do not return any values (these are often called procedures) and these usually just involve outputting values. This subroutine takes a number as its input, and outputs the first twelve multiples of that number:

```
SUBROUTINE times_tables(n)
   FOR i ← 1 TO 12
      multiple ← i * n
      OUTPUT multiple
   ENDFOR
ENDSUBROUTINE
```

It is a very common mistake to confuse values that are outputted and return values: when this subroutine finishes it does not give anything back or, in other words, this subroutine does not evaluate to a value.

Multiple return values

Subroutines are only allowed to return one value, this might appear limiting. How would you create one subroutine that asks a user for their username and password and returns them both? The answer, use a data structure:

```
SUBROUTINE get_username_and_password()
   OUTPUT 'Enter your username'
   username ← USERINPUT
   OUTPUT 'Enter your password'
   password ← USERINPUT
   RETURN [username, password]
ENDSUBROUTINE
```

Programmers need to know that when this subroutine is called it is going to return an array and to access the username and password separately will need to access the first and second elements of that array respectively. For example:

```
details ← get_username_and_password()
username ← details[0]
password ← details[1]
```

Although this example uses an array, any data structure can normally be used as a return type.

Subroutines calling subroutines

Subroutines are used to:

- make programs more structured
- reduce the risk of introducing errors in a program and make the errors easier to find
- save the programmer time (and reduce the amount of code that needs to be written or amended)
- make it easier to understand what is happening in a program
- make parts of the code easily reusable.

However, all of the above are only possible if the subroutines themselves are relatively short. There isn't a specific guide regarding how long a subroutine should be but if subroutines can be further split up into different, self-contained blocks of code then these blocks should themselves probably be subroutines. Using an example from earlier, if we want to amplify all of the values in an array (ie multiply all of the values in an array by a constant factor), but keep these values within an upper and lower bound (and 'clip' them if they go beyond these boundaries) we can combine subroutines:

```
# multiply all values in an array by a factor
SUBROUTINE multiply_all(arr, factor)
   FOR i ← 0 TO LEN(arr)-1
      arr[i] ← arr[i] * factor
   ENDFOR
   RETURN arr
ENDSUBROUTINE

# clip the values in an array to a given range
SUBROUTINE clip(low_limit, high_limit, arr)
   FOR i ← 0 TO LEN(arr)-1
      IF arr[i] < low_limit THEN
         arr[i] ← low_limit
```

```
        ELSE IF arr[i] > high_limit THEN
            arr[i] ← high_limit
        ENDIF
    ENDFOR
    RETURN arr
ENDSUBROUTINE

# amplify and clip an array
SUBROUTINE amplify(arr, factor, low_limit, high_limit)
    amplified ← multiply_all(arr, factor)
    amplified ← clip(low_limit, high_limit, amplified)
    RETURN amplified
ENDSUBROUTINE

# example subroutine call
signal ← [3.5, -1.1, 0.3, 4.1, 3.2, -2.6]
amplified_signal ← amplify(signal, 2.0, 0.0, 6.0)
```

To understand the flow of this code we need to inspect what happens when the `amplify` subroutine is called on the last line. The subroutine first of all calls `multiply_all`, so we need to jump to the `multiply_all` subroutine to see what that does, after that subroutine has executed and returned a value the `amplify` subroutine calls `clip`, so we likewise have to see what the `clip` function does. The `amplify` subroutine returns the multiplied and clipped array and this is assigned to the array `amplified_signal` on the last line of code.

The values within amplified after this line of code (when called with the example above) are:

```
# amplied has values [7.0, -2.2, 0.6, 8.2, 6.4, -5.2]
```

The values of `amplied` after it has been assigned the value returned from the `clip` subroutine, again following the example above, are:

```
# amplied has values [6.0, 0.0, 0.6, 6.0, 6.0, 0.0]
```

Breaking problems up into manageable modules and implementing the solutions to these modules in subroutines will be covered in more detail in the resource on structured programming.

Extension information for reference only (Students do not need to be taught the following at GCSE)

Scope and symbol tables (not included in the specification)

In the following section, assume that the pseudo-code used has strict rules about scoping for the purposes of the examples. The actual rules of scoping differ from language to language but having a grasp of how they work in general will improve understanding across a range of languages.

Students are not required to have an understanding of symbol tables, but it is very useful knowledge that helps demystify much about how subroutines and other blocks of code access variables.

It is important to realise that these subroutines live in a tunnel. They allow values in at the start and return a value at the end but in the middle often nothing else comes in, however there are some exceptions to this (see the resource on user input).

Another trickier thing to realise is that the input parameters and any other variables declared within a subroutine only exist in that subroutine. Take this example:

```
SUBROUTINE addTwo(n)
   result ← n + 2
   RETURN result
ENDSUBROUTINE


x ← addTwo(1)
OUTPUT x       # would output the value 3
OUTPUT n       # would likely cause an error
OUTPUT result  # would also likely cause an error
```

Both the input parameter `n` and the variable result are said to be local variables to the subroutine `addTwo` and to have scope only within this subroutine. This means that any attempt to access them and change them outside of the subroutine will result in an error because that part of the program will not know what those variables are.

A good way to think about this is to return to our definition of a variable as an identifier that points to a memory slot, all of these identifiers and the memory slots they point to can be kept together by the computer running the program in a symbol table. The example shows a small program and the associated detailed symbol table used by the computer to keep track of the example memory slots the program's variables point to:

```
first_var ← 1
second_var ← 2
third_var ← 3
```

identifier	type	example memory slot	scope
first_var	int	100	lines 1 - 3
second_var	int	104	lines 2 - 3
third_var	int	108	line 3

Every time a subroutine is called (not declared) it creates its own symbol table for as long as it is running. We'll use an earlier example to show this (the numbers to the left of the program are line numbers to make referring to the code easier):

```
1   SUBROUTINE addTwo(n)
2      result ← n + 2
3      RETURN result
4   ENDSUBROUTINE
5
6   SUBROUTINE triple(m)
7      result ← m * 3
8      RETURN result
9   ENDSUBROUTINE
10
11  a ← 4
12  b ← addTwo(a)
13  c ← addTwo(3)
14  a ← triple(c)
15  b ← triple(b) + addTwo(c)
```

The first line of code to execute is line 11 in the 'top-level' of the program, remember that subroutines do not execute until they are called. The symbol table for the top-level could look like this (again, the memory slots are made up numbers):

identifier	type	example memory slot	scope
a	int	3240	lines 11 - 15
b	int	3244	lines 12 - 15
c	int	3248	lines 13 - 15

On line 12 the variable `b` is assigned the value of `addTwo(a)`. This involves a call to the subroutine `addTwo` and so a new symbol table is created:

identifier	type	example memory slot	scope
n	int	3252	addTwo (lines 1 – 3)
result	int	3256	addTwo (lines 2 – 3)

The variable `a` is not in this symbol table but as soon as `addTwo(a)` is called the value of `a` is copied to the input parameter `n`. When the value of `result` is returned and addTwo(a) evaluates to 6 this symbol table ceases to exist so any attempted use of the identifiers n and result after this point will cause an error. However the top-level symbol table still exists.

On line 13 another call to addTwo is made using the input value 3 (so the value 3 is copied to the input parameter n) and the following symbol table is created:

identifier	type	example memory slot	scope
n	int	3260	addTwo (lines 1 – 3)
result	int	3264	addTwo (lines 2 – 3)

Again, as soon as the value of result is returned from this function this symbol table will cease to exist although the symbol table with a, b and c will still be present.

On line 14 a call is made to the triple function so a new symbol table will be created just for this function call:

identifier	type	example memory slot	scope
m	int	3268	triple (lines 6 – 8)
result	int	3272	triple (lines 7 – 8)

Just as with the symbol table for addTwo, this symbol table will not exist when the value of result is returned.

Finally, on line 15 two function calls are made to triple and to addTwo. If we assume that our program will evaluate these left to right (i.e. evaluate triple(b) and then evaluate addTwo(c)) then firstly a new symbol table for triple will be created and then cease to exist and after that a new symbol table for addTwo will be created and likewise not exist once it has returned its value.

We can use this knowledge of symbol tables to work out the potential confusion in this program:

```
1   SUBROUTINE double(n)
2      n ← 2*n
3      RETURN n
4   ENDSUBROUTINE
5
6   n ← 1
7   n ← double(n)
```

It might look like there is only one variable in this whole program (`n`) but because the call to `double` creates a new symbol table it creates its own `n` which is completely separate to the `n` on lines 6 and 7. The `n` used on lines 2 and 3 exists only in the 'tunnel' of that call to `double`.

Call by value and call by reference (not in the specification)

There is a final bit of trickery involved in using input variables in programming languages that can cause a lot of confusion. It concerns the way that some languages pass the values that are used when subroutines are called (often called arguments) to the input variables.

Our explanation of input variables so far has been that when a subroutine is called, the values of the inputs are copied to the input variables declared in the subroutine itself.

The following code will copy the value of the variable `x` (i.e. `1`) to the input variable `n` when the subroutine `double` is called:

```
SUBROUTINE double(n)
   n ← 2*n
   RETURN n
ENDSUBROUTINE

x ← 1
result ← double(x)
```

This process has the name 'call by value' because the subroutine is called with the value of its input.

However, many languages also use 'call by reference' where the memory location of the input, not the value, is copied to the input variable declared in the subroutine.

Often this is the case for data structures such as arrays and records and also objects (which aren't covered in the GCSE specification). We'll use a similar example to explain this:

```
SUBROUTINE double(arr)
   FOR i ← 1 TO LEN(arr)
      arr[i] ← arr[i]*2
   ENDFOR
ENDSUBROUTINE

xs ← [1, 2, 3, 4, 5]
double(xs)
```

Using our previous assumptions, the array `xs` will be copied to the input variable `arr` and inside the subroutine `double` all of the elements of `arr` will be multiplied by two. Because the subroutine double does not return a value, this will not affect any of the elements inside the array xs.

However, when using data structures as inputs this assumption isn't true for many languages (including Python) because it is the memory location of the array that is passed to double not the values within the array (or list in Python). The symbol tables could look like this:

identifier	type	example memory slot	scope
xs	array	1002	top-level (lines 7 – 8)

identifier	type	example memory slot	scope
arr	array	1002	double (lines 1 – 4)
i	int	2432	double (lines 2 – 4)

The highlighted memory slots of both xs and arr are the same, so when the subroutine double finishes it will have changed the values in xs, that is, xs will have the final value of [2, 4, 6, 8, 10].

Recursion (not in the specification)

Recursion occurs where subroutines call themselves. This can produce very elegant solutions to some problems but it is not required until A-Level. The following brief example of a recursive subroutine calculates n! (factorial) which is every positive integer multiplied by each other up to n. For example, 4! is 1x2x3x4.

```
SUBROUTINE factorial(n)
   IF n = 1 THEN
      RETURN 1
   ELSE
      RETURN n * factorial(n-1)
   ENDIF
ENDSUBROUTINE
```

Computing the nth Fibonacci number could be implemented like so:

```
SUBROUTINE fibo(n)
   IF n ≤ 2 THEN
      RETURN 1
   ELSE
      RETURN fibo(n-1) + fibo(n-2)
   ENDIF
ENDSUBROUTINE
```

Both of these subroutines share two common features that are always found in recursive subroutines:

- the subroutines return a final value once the input has reached a certain point, eg when n is 1 in the factorial subroutine and when n is less than or equal to 2 in the fibo subroutine
- if this value is not yet met then a recursive subroutine will call itself, the value which is input to this recursive call will be different to the input to the calling subroutine, eg factorial(n) has the recursive call of factorial(n-1).