Teaching guide: String handling

This resource will help with understanding string handling operations in a programming language. It supports Section 3.2.8 of our GCSE Computer Science specification (8525). The guide is designed to address the following outcomes:

- View strings as arrays of characters.
- Use understanding of arrays to access and update characters within strings.
- Explore conversion of other data types to strings.

Characters are the symbols that when put together provide the means for us to communicate and understand each other using written text. Obvious examples are the letters that can be found on the keyboard such as `a`, `b` and `c` which are different from `A`, `B` and `C`. When you read the character `'7'` you will probably mean it as 'the number seven' or 'the quantity seven' but as far as our programs are concerned it is just another symbol.

As well as the alphanumeric characters there are also the punctuation characters such as `;`, `.` and `' '` (the space character).

When these character symbols are put into a sequence they are called strings (as in 'a string of characters'). Programming with strings is such common practice that almost all programming languages have built-in ways to change, manipulate and convert them.

Even though the way a particular programming language implements strings may vary, it helps to think of strings as arrays of characters – this way all of the main subroutines that involve strings are easily understood and applied.

Strings as arrays: length, position and substring

If the string `'lovelace'` is visualised as a string of characters then you have the following sequence of characters with their positions shown above:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
l	o	v	e	l	a	c	e

One string subroutine is immediately obvious: counting the number of characters within the string:

- The length of this string is `8`

At first glance another subroutine looks straightforward too:

- The position of any character is the number above it in the table above, eg `'v'` has position `2`

The position of `'l'` is trickier as `'l'` appears twice in the string – most position subroutines will tell you the first position although some will allow you to specify where the subroutine will start counting.

Furthermore, what is the position of `'b'` (which doesn't appear in the string)? If a character doesn’t exist in a string most position subroutines will give the value of `-1` (which is unequivocally *not* a position) although other languages will give a value that means `'nothing'`. It is always essential to look at a language’s documentation to be sure.

Both length and position are subroutines that take the string as input and return an integer and throughout these Teaching Guides they will be referred to as **`LEN`** and **`POSITION`**.

```
# example subroutine calls of LEN and POSITION
the_string ← 'lovelace'
OUTPUT LEN(the_string) # Should output 8
OUTPUT POSITION(the_string, 'v') # Should output 2
OUTPUT POSITION(the_string, 'l') # Should output 0
OUTPUT POSITION(the_string, 'b') # Should output -1
```

Another subroutine that is commonly used with strings is to extract a sequence of characters found next to each other in that string – this is called a substring. For example, all of the following are substrings of `'lovelace'`:

```
'love'
'ace'
'elac'
'v'
```

Two other valid substrings that need to be considered are the string that contains no characters (the ‘empty’ string, represented as `''`, but not to be confused with `"`, the double quote, or `' '`, the string consisting of a single space) and the string that contains all of the original characters:

`'lovelace'`

To compute a substring we need to know:

- the starting position of the substring
- the ending position of the substring
- the string to be used

Some programming languages (eg Python) require the ending position of the substring to be the index one further along than the last character required, and so with Python the string `'lovelace'`, with a starting position of `0` and ending position of `3` would return `'lov'` (Python does not use a substring subroutine so this would be written `"lovelace"[0:3]`).

If we took the ending position (`3`) to be the last position of the substring within the main string then the subroutine would return `'love'`.

The following subroutine calls show how **`SUBSTRING`** could be used:

```
# example subroutine calls of SUBSTRING
the_string ← 'lovelace'
the_substring ← SUBSTRING(0, 3, the_string) # = 'love'
the_substring ← SUBSTRING(2, 4, the_string) # = 'vel'
the_substring ← SUBSTRING(4, 7, the_string) # = 'lace'
the_substring ← SUBSTRING(4, LEN(the_string) - 1, the_string)
# = 'lace'
the_substring ← SUBSTRING(0, LEN(the_string) - 1, the_string)
# = 'lovelace'
```

The following example uses **LEN**, **POSITION** and **SUBSTRING** and is a subroutine that performs basic validation of an email address. All email addresses contain an @ symbol followed by some text, then at least one full stop character, then more text. So our validation program could do the following:

- find the position of the @ symbol
- find the position of a full stop *after* this @ symbol
- make sure that the position of the full stop is at least two positions further along than the @ symbol (to ensure that there is text between the @ and full stop)
- make sure that the position of this full stop is less than the overall length of the email address (to ensure that some text follows the full stop)

```
SUBROUTINE validate_email(address)
   # check that the @ symbol is in the address
   at ← POSITION(address, '@')
   IF at = -1 THEN
      RETURN False
   ENDIF

   # get the remaining text after @ and find the first
   # position of a . symbol
   rest ← SUBSTRING(at + 1, LEN(address) - 1, address)
   stop ← POSITION(rest, '.')

   # if the position of . is -1 or if the position is
   # immediately after the @ symbol then the email is invalid
   IF stop < 1 THEN
      RETURN False
   ENDIF

   # if there is no more text after . then the email
   # is invalid
   IF stop = LEN(rest) - 1 THEN
      RETURN False
   ENDIF

   # if the email passes all of these checks then it is valid
   RETURN True
ENDSUBROUTINE
```

String concatenation

Concatenate means 'to chain' and string concatenation is chaining two strings together to create a new one. If we took the two strings `'love'` and `'lace'`, the concatenation of these would be the new string `'lovelace'`. This string operation commonly (but not always) uses the + symbol, not to be confused with its more common use as the addition operator.

```
# evaluates to 'lovelace'
concatenated_string ← 'love' + 'lace'
# evaluates to 'lacelove'
concatenated_string ← 'lace' + 'love'
# evaluates to 'lovelacelove'
concatenated_string ← 'love' + 'lace' + 'love'
```

It is easy to forget that the space symbol is also a character, so if we need to concatenate strings to produce an English sentence we need to introduce spaces too:

```
# evaluates to 'computerscience'
concatenated_string ← 'computer' + 'science'

# evaluates to 'computer science' (a space after computer)
concatenated_string ← 'computer ' + 'science'

# also evaluates to 'computer science'
concatenated_string ← 'computer' + ' ' + 'science'
# adds an 's' to an existing string 'developer'
# evaluates to 'developers'
concatenated_string ← 'developer'
concatenated_string ← concatenated_string + 's'
```

String conversion

The following program asks the user to enter a number, adds one to that number and then outputs the answer:

```
OUTPUT 'Enter an integer'
var ← USERINPUT
var ← var + 1
OUTPUT var
```

If you delve deeper into the type of number throughout this short program you might begin to see a problem. **USERINPUT** would typically come from the keyboard and as such would be a sequence of characters, ie a string.

When you output a value it would typically be displayed on a screen and you will also assume that this would need a string to work. However, the third line of the program adds `1` to `var` and this is only possible if `var` is itself a number (either real or integer). It looks as though the program has implicitly converted `var` from a string to a number and then back to a string.

You could create a subroutine called **`STRING_TO_INT`** that takes a string as input and returns the integer represented by this string. Likewise, we could create a subroutine called **`INT_TO_STRING`** that converts an integer to a string. We could use these two subroutines to rewrite the program above to:

```
OUTPUT 'Enter an integer'
string_var ← USERINPUT
int_var ← STRING_TO_INT(string_var)
int_var ← int_var + 1
string_var ← INT_TO_STRING(int_var)
OUTPUT string_var
```

or more tersely:

```
OUTPUT 'Enter an integer'
var ← STRING_TO_INT(USERINPUT)
var ← var + 1
OUTPUT INT_TO_STRING(var)
```

Either way, writing this in pseudo-code would normally not be needed because the intention of the program is clear but you should be aware that operations such as this may be necessary in your chosen language.

There is another issue with this program – what if the user enters something other than a string that can be converted to an integer? Most programming languages would expect the string input to **`STRING_TO_INT`** conversion subroutine to be something that can be unambiguously converted, for example `3`, `5`, `-54` or `0`. If the user enters a string such as `'hello'` then the program will most likely crash. This is covered in the Teaching Guide – Data validation & authentication.

This table lists four subroutines, their purpose and example input and output data:

Subroutine	Purpose	Input	Return value
`STRING_TO_INT`	converts a string to its integer value	`'1'`	`1`
		`'43'`	`43`
		`'-9145'`	`-9145`
		`'one'`	`error`
		`'3.141'`	`error`
`STRING_TO_REAL`	converts a string to its real value	`'1.54'`	`1.54`
		`'-9540.15'`	`-9540.15`
		`'4'`	`4.0`
		`'two thirds'`	`error`
`INT_TO_STRING`	converts an integer value to its string representation	`1`	`'1'`
		`774`	`'774'`
		`4.6`	`error`
`REAL_TO_STRING`	converts a real value to its string representation	`0.44`	`'0.44'`
		`-9949.3`	`'-9949.3'`
		`1`	`'1.0'`

These four string conversion operations are the ones covered in the specification although it is likely that your programming language will contain other string conversions, for example converting dates and times that will enable you to write more complex programs. As always, students should make themselves aware of the necessity and also the functionality of these string conversion operations in their chosen language.