Strings

Strings are a feature of every programming language. A string can be defined as an ordered sequence of characters or symbols encased in matching single or double quotation marks. Most languages will support primary ASCII characters as well as Unicode representations. A character will occupy one byte of memory. However, there are many additional ways of representing strings.

In this reading, you will learn about the string data structure, how strings are represented across different languages, what they are commonly used for and why programming languages make strings immutable.

String representation

There are critical differences in how each language represents and supports strings. All languages support the basic operations of creating, modifying, copying, and assigning strings to variables. Additionally, everyday actions include concatenating, appending strings, finding the substring and dealing with collections of strings.

When processing strings, many languages will allow you to perform algebraic actions on them. For instance, `String_A == String_B`, which will return a `true` or a `false` or `String_A < String_B` which determines which is alphabetically first. Some languages allow strings to represent variables in a string but require a special symbol. Often this is a dollar sign `$` before the variable name but might also, in some instances, include being encased in curly brackets `{}`. Escape flags allow the inclusion of symbols in a string. Imagine that you want to have quotations within a string.

```
String = "the man said \"two more pints please\" to the barman"
```

Here the use of the backslash `\` ensured that the string would retain the quotation marks in the sentence. Other escape symbols could be `#, %,'` or a double quote inside a string denoted by single quotes. While the actual implementation may differ, the general approach to dealing with strings is the same.

Common usage

One area that deals heavily with strings is Natural Language Processing (NLP). How strings are encoded when reading them from various locations such as Twitter, PDF files, text documents, or Reddit can raise unexpected issues. These can create complex debugging problems if the application processes an unusual symbol representation that affects the results.

When writing strings into a program, it is common to apply tokenization. This is converting a string into an array of smaller strings. Here you would identify a delimiter and break the string into segments separated by the said delimiter. So, a paragraph might be broken into sentences through the use of the period(`.`) or a sentence into individual words by way of segmenting by space (`" "`). However, care should be taken when tokenizing text. Consider the text below that has three period symbols (`.`).

At 3.30 I went to the shop and spent $40.40 dollars.

Conventionally, you'd read it as a single sentence, but the periods used to indicate the time and amount of money are delimiters so the sentence is broken up into three parts.

Tokenization on processed, formatted data offers less of a challenge. A common way to store formatted string representations is through Comma Separated Values (CSV) or Tab Separated Values (TSV).

Processing such strings is known as parsing, and the returned array will equal the size of the number of delimiters found, with the delimiter removed. Here is an example of a CSV file that holds the columns' age, name, and hair color. Notice how there is no comma at the end. Unless otherwise stated, the new line symbol also functions as a delimiter.

```
age, name, hair color
24, John,   black
25,  Tony, brown
34, Brian, blue
29, Mary, red
43, Martin, yellow
```

Immutability

An important concept to consider when dealing with strings is whether the string is mutable or immutable. Mutability refers to your ability to change a string after it has been created. Some languages like Ruby and PHP allow strings to be changed after creation. It is more common, however, to use immutability, such as in Java, C#, JavaScript, Python, and Go.

There are several reasons why programming languages make strings immutable. For one, it can reduce memory consumption. Instead of creating a variable that contains a string, a string pool is designed to represent all strings used. The immutable approach reuses memory allocation by having all instances of `string` point to one location. So, when a change happens with a string, say it becomes `strings`, instead of changing the value of `string`, the variable is pointed to another instance of `strings`. Or it's pointed to another immutable example representing `strings`, which is then added to the string pool.

This is a vast memory-saving device. Consider reading existing words instead of creating a memory location for each word, including repeating ones. Using a unique set reduces the required space. The drawback is that if your application constantly changes texts, a memory penalty is incurred for every alteration made.

Conclusion

In this reading, you learned about the string data structure, how strings are represented across different languages, what they are commonly used for and why programming languages make strings immutable.