HyperionDev

The String Data Type

Task

Visit our website

Introduction

Strings are some of the most important and useful data types in programming. Why? Let’s think about it like this. When you were born, your parents did not immediately teach you to do maths – the first thing they taught you to do was to speak, to say words like “Mum” or “Dad”, to construct full sentences. For the same reason, we are starting your programming journey by learning how to ‘teach’ the computer to be able to communicate with the user. This requires us to begin with a focus on strings.

What are strings?

Strings are probably the most important data type in programming. They are used as a medium of communication between the computer and the user. The user can enter information as a string, and the program can use the data to perform operations and finally display the calculated answer to the user.

A string is a list of letters, numerals, symbols, and special characters that are put together. A simple example of what strings can store is the surname, name, address, etc. of a person, but keep in mind that the range of values that we can store in a string is vast.

In Python, strings must be written within quotation marks (“ ”) for the computer to be able to read them.

The smallest possible string contains zero characters and is called an empty string (i.e. string = “”).

Examples of strings:

```python
# Assigning the name "Linda" to the variable name
name = "Linda"

# Assigning the song title "The Bird Song" to the variable song
song = "The Bird Song"

# Reassigning the name to "John"
name = "John"

# Assigning the joke "Knock, knock, Who's there?" to the variable joke
joke = "Knock, knock, Who's there?"

# Empty string
empty_string = ""
```

You can use any name for your variable, but the actual string you are assigning to the variable must be within “ ” (quotation marks).

String numeric representation

We can even store numbers as strings. When we are storing a number (i.e., 9, 0, 231) as a string, we are essentially storing it as a word. The number will lose all its number-defining characteristics, and cannot be used in any calculations. All you can do with it is read or display it.

In real life, sometimes we don’t need numbers to do calculations, we just need them for information purposes. For example, the house number you live in (let’s say, 45 2nd Street, Hacker Town, 2093) won’t be used for performing any calculations. What benefit would there be for us to find out what the sum is of all the house numbers in an area? When visiting or delivering a package, the only thing we need is to know what number the house is. The number just needs to be visible. The basic concept is the same when storing a number as a string; all we want is to be able to take in a value and display it to the user.

For example:

```
# Assigning the license plate "CTA 456 GP" to a variable
licence_plate = "CTA 456 GP"

# Assigning the telephone number "082 123 4567" to a variable
telephone_number = "082 123 4567"
```

Defining multi-line strings:

Sometimes, it's useful to have long strings that can go over one line. We use triple single quotes (“‘ ’”) to define a multi-line string. Multi-line strings preserve the formatting of the string.

For example:

```
long_string = ''' This is a long string.
Note that using triple quotes preserves everything inside it as a string, even
the different lines and the \n spacing. '''
```

String formatting

Strings can be added to one another. We can do this using an approach called **concatenation**, which looks like this:

```
# Concatenating the strings stored in name and surname variables
name = "Peter"
surname = "Parker"
full_name = name + surname
```

full_name will now store the value "PeterParker".

The + symbol simply joins the strings. If you wanted it to make your code more presentable, you could put spaces between the words.

```
# Adding spaces between the words
full_name = name + " " + surname
```

Above, we have added a blank space in between the two strings, so full_name will now store the value **Peter Parker**. **Note that you cannot concatenate a string and a non-string**; you need to cast the non-string to a string if you want to concatenate it with another string value. If you try to run code that adds a string and a non-string, you will get an error. For instance, if we wanted to add 32 as Peter Parker's age, we would have to cast the number as a string to print it.

```
# Using the str() method to cast a number as a string
print(full_name + str(32))
```

The only exception to this is if the number was stored in a variable as a string, like this:

```
age = "32"
print(full_name + age)
```

Using format()

However, this is a clunky way of formatting strings. This approach is still used in older languages, such as Java, and does have its place, but it is much better practice to use either the **format()** method or an f-string. Some of you may have seen these before, but even if you have, they are worth a recap.

Let's start with **format()**.

```python
# Assigning the name "Peter Parker" to the variable name
name = "Peter Parker"

# Assigning the age 32 (as an integer) to the variable age
age = 32

# Creating a sentence using string formatting to insert the name and age
sentence = "My name is {} and I'm {} years old.".format(name, age)

# Printing the sentence
print(sentence)
```

In the example above, a set of opening and closing curly braces **{}** serve as a placeholder for variables. The variables that will be put into those placeholders are listed in the brackets after the keyword format. The variables will fill in the placeholders in the order in which they are listed. Therefore, the code above will result in the following output: **My name is Peter Parker and I'm 32 years old**.

Notice that you don't have to cast a variable that contains a number (age) to a string when you use the **format()** method.

Using f-strings

The shorthand for the format function is **f-strings**. Take a look at the example below:

```python
# Using f-strings to format strings in Python
name = "Peter Parker"
age = 32
sentence = f"My name is {name} and I'm {age} years old."
print(sentence)
```

In f-strings, instead of writing **.format()** with the variables at the end, we write an f before the string and put the variable names within the curly brackets. This is a neat and concise way of formatting strings.

String manipulation

A string is essentially a list of characters. For instance, the word “Hello” is made up of the characters H+e+l+l+o. We can use this to our advantage to access the exact character we need using indexing.

```
' H e l l o   w o r l d  ! '
  0 1 2 3 4 5 6 7 8 9 10 11
```

Each character of a string (including blank spaces, which are also characters) is indexed by numbers starting from zero for the first character on the left.

```
# Assigning the string "Hello" to the variable greeting
greeting = "Hello"

# Printing the concatenation of individual characters in the greeting string
print(greeting[0] + greeting[1] + greeting[2] + greeting[3] + greeting[4])
```

Output:

```
Hello
```

Note that, because the indexing of string characters always starts at zero, the value produced by **len()** is always one less than the index of the final character. You will learn why this is useful later!

You can also slice a string. Slicing in Python extracts characters from a string, based on a starting index and ending index. It enables you to extract more than one character or ‘chunk’ of characters from a string. The first print statement below will print out a piece of the string. It will start at **position/index 1** and end at **position/index 4** (which is not included). Note that the original string remains intact, as the second print statement shows.

```
greeting = "Hello"

# Printing from index 1 up to index 4
print(greeting[1:4])
print(greeting) # Note that slicing doesn't affect the original string
```

Output:

```
ell

Hello
```

You can even put negative numbers inside the brackets. As you know, the characters are indexed from left to right starting at zero, but they are also **indexed from right to left using negative numbers,** where -1 is the rightmost index and so on. Using negative indices is an easy way of starting at the end of the string instead of the beginning. This way, -3 means "3rd character from the end".

Look at the example below. The string is printed from the first index, “e”, all the way to the end. Notice that you do not need to specify the end of the index range; if it is left out, the slice will just continue to the end of the string.

```
# Printing from the first(1) index to the end.
greeting = "Hello"
print(greeting[1:])
```

Output:

```
ello
```

Here’s the inverse; in the example below, the slice begins from position 0 (as a starting point is not specified) and goes up to, but not including, position 1 (remember that the character at the ending position index is never included in the output):

```
# Printing from index 0 up to index 1.
greeting = "Hello"
print(greeting[:1])
```

Output:

```
H
```

In the next example, the slice begins from position 1, includes positions 1 and 2, and then continues to the end of the string and skips/steps over every other position. This is known as an **extended slice**. The syntax for an extended slice is **[begin : end : step]**. If the end is left out, the slice continues to the end of the string as we've seen before.

```
greeting = "Hello"

# Printing every second character starting from index 1 in the greeting string
print(greeting[1::2])
```

Output:

```
el
```

In this final example, you can think of the "-1" as a reverse order argument. The slice begins from position 4, continues to position 1 (not included, as it's the ending position), and skips/steps backwards one position at a time:

```
# Assigning the string "Hello" to the variable greeting
greeting = "Hello"

# Printing a slice of the greeting string, starting from index 4 and ending at
index 1 (exclusive), with a step of -1
print(greeting[4:1:-1])
```

Output:

```
oll
```

You can print a string in reverse by using **[::-1]**. Remember that the syntax for an extended slice is **[begin : end : step]**. By not including a beginning or end position, and specifying a step of -1, the slice will cover the entire string, backwards, so the string will be reversed. You can find out more about extended slices **here**.

Remember: slicing a string does not modify the original string. You can capture a slice from one variable in a separate variable. Try running the following code:

```
# Assigning the string "Hello world!" to the variable my_string
my_string = "Hello world!"

# Assigning a slice of my_string, from index 0 to index 4 (exclusive), to the
variable fizz
fizz = my_string[0:5]

# Printing the value of fizz
print(fizz)

# Printing the original value of my_string
print(my_string)
```

Output:

```
Hello

Hello world!
```

By slicing and storing the resulting substring in another variable, you can have both the whole string and the substring handy for quick, easy access.

String-handling methods

There are various built-in functions in Python that we can use to manipulate strings. Functions are used to save us from having to write monotonous code over and over. Built-in functions are provided as part of Python, meaning that you can just go ahead and use them to do useful things, without having to specify what they need to do in the way you would have to if you were writing your own functions. Special functions called **string methods** are used to manipulate strings.

Here are a few examples of useful string methods:

- The **upper()** and **lower()** methods make a new string with all letters converted to uppercase and lowercase, respectively.

```
a_string = "Hello World"
print(a_string.upper())  # prints out HELLO WORLD
print(a_string.lower())  # prints out hello world
```

- The **replace()** method will replace any occurrence of a string with another string of your choice. In the example below, every **$** character in the string stored in the variable **a_sentence** will be replaced with a space character.

```
a_sentence = "Welcome$to$the$world$of$programming"

# prints out Welcome to the world of programming
print(a_sentence.replace("$" , " "))
```

- The **strip()** method is used to remove a certain character from the start and end of a string value. In the example below, the strip() function will remove all the ***** characters from the start and end of the string value stored in the variable named **str_help**.

```
str_help = "******Please leave me alone******"
print(str_help.strip('*'))  # Prints out Please leave me alone
```

- The **len()** function, while not strictly a string method, plays a crucial role in string manipulation tasks. It's a built-in function compatible with multiple data types, including strings. It allows us to retrieve the number of characters or the length of a string. For example, when applied to a string like "Hello world!", len() accurately counts the number of characters, including spaces and punctuation marks. In the example provided, the print statement outputs 12, indicating the total length of the string.

```
# Printing the length of a string
print(len("Hello World!"))
```

Output:

```
12
```

Most programming languages provide built-in functions to manipulate strings. These usually include functions to concatenate strings, search from a string, extract substrings from a string, etc. There is more to be learned about strings, and depending on your course, you may revisit this in a later lesson.

Escape characters

Python uses the backslash (\) as an escape character, employed as a marker character to tell the compiler/interpreter that the next character has some special meaning. The backslash, together with certain other characters, is known as an 'escape sequence'.

Some useful escape sequences are listed below:

- **\n** – Newline: this will insert the equivalent of pressing enter to take the insertion point for output to the next line.
- **\t** – Tab: inserts a tab.
- **\s** – Space: inserts a space.

The escape character can also be used if you need to include quotation marks within a string. You can put a backslash \ in front of a quotation mark so that it doesn't terminate the string. You can also put a backslash in front of another backslash if you need to include a backslash in a string.

Now it's time to put what you've learned into practice.

Take note

Before you get started, we strongly suggest you use an editor such as VS Code to open all text files (.txt) and Python files (.py).

First, read the accompanying Python example files. These examples should help you understand some simple Python. You may run the examples to see the output. Feel free to also write and run your own example code before doing the practical tasks, to become more comfortable with Python.

Practical task 1

Follow these steps:

- Create a new Python file in this folder called **strings.py**
- Declare a variable called hero that contains the value “$$$Superman$$$”
- Use the string-manipulation method strip() and print hero so that the output is: Superman

Practical task 2

Follow these steps:

- Create a new Python file in this folder called **replace.py**.
- Save the sentence: “The!quick!brown!fox!jumps!over!the!lazy!dog.” as a single string.
    - Reprint this sentence as “The quick brown fox jumps over the lazy dog.” using the replace() function to replace every exclamation mark “!” with a blank space.
    - Reprint that sentence as: “THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.” using the upper() function.
    - Reprint the sentence in reverse.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Take note

Make sure that you have installed and set up all programs correctly. You have set up GitHub correctly if you are reading this, but **Python** or your editor may not be installed correctly.

If you are not using Windows, please ask a code reviewer for alternative instructions.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.