Hyperiondev

TASK

String Handling

Visit our website

Introduction

WELCOME TO THE STRING HANDLING TASK!

One of the most crucial concepts to grasp when it comes to programming is string handling. You will need to be very comfortable with string handling, so it is important to refresh and consolidate your knowledge. In this task, you will briefly recap some key points and then learn to create more advanced programs with strings which use more functions and programming techniques.

INDEXING STRINGS

You can think of the string 'Hello world!' as a list and each character in the string as an item with a corresponding index.

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

The space and the exclamation point are included in the character count, so 'Hello world!' is 12 characters long, from ‘H’ at index 0 to ’!’ at index 11.

```
String = "Hello"
print(String[0]) # H
print(String[1]) # e
print(String[2]) # l
print(String[3]) # l
print(String[4]) # o
```

Remember that if you specify an index, you’ll get the character at that position in the string. You can also slice strings by specifying a range from one index to another; remember that the character at the starting index is included and that at the ending index is not.

Note that slicing a string does not modify the original string. You can capture a slice from one variable in a separate variable. Try typing the following into the interactive shell:

```
original_string = "Hello world!"
new_string = original_string[0:5]
print(new_string)
```

What will be printed by the code above?

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 METHODS

Once you understand strings and their indexing, the next step is to master using some of the common string methods. These are built-in modules of code that perform certain operations on strings. These methods are useful as they save time since there is no need to write the code over and over again to perform certain operations. The most common string methods are (where *s* is the variable that contains the string we are working with):

- **s.lower()** and **s.upper()** — convert a string to either uppercase or lowercase.
- **s.strip()** — removes any whitespaces from the beginning or end of a string.
- **s.strip(',')** — optional input to s.strip(). The string provided will be removed from the string. In this case, the "," character gets removed from the string.
- **s.find('text')** — searches for a specific text and returns its position in the string you are searching. If the string isn't found, -1 is returned.
- **s.replace('oldText', 'newText')** — replaces any occurrence of 'oldText' with 'newText'.
- **s.split('word')** — breaks down a string into a list of smaller pieces. The string is separated based on what is called a *delimiter*. This is a string or *char* value that is passed to the method. If no value is given it will automatically split the string using whitespace as the delimiter and create a list of the characters.
- **s*.append(item)*** — to add an item to the end of a **list**, you use the *append()* method. For example, *list.append(item)* adds the single item within the brackets to the end of a list.
- **"".join(string_list)** — takes a list of strings or characters and joins them to create one string. You can specify the character, if any, you wish to use to join

the list elements. For example, *"@".join(["apples", "bananas", "carrots"])* would output *"apples@bananas@carrots"*.

Examine **example.py** to see how each of these methods can be used.

ESCAPE CHARACTER

Python uses the backslash (\) as an escape character. The backslash (\) is used 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
- **\t** - Tab

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. What would the code below print out? Try it and see!

```
print("Hello \n\"bob\"")
```

You can also put a backslash in front of another backslash to include a backslash in a string.

```
print("The escape sequence \\n creates a new line in a print statement")
# Output: The escape sequence \n creates a new line in a print statement
```

STRING BUILDING

As you know, the best practice is to create strings through *.format()* and, by extension, f-strings. For example:

```
name = "Peter"
print("Hello, {}!".format(name))
```

And, more succinctly:

```
name = "Peter"
print(f"Hello, {name}!")
```

However, sometimes a form of concatenation is needed in order to *build* a string. One of the main reasons for writing a program is to manipulate information. We turn meaningless data (e.g. 24) into useful information (e.g. Tom is 24 years old). String building allows us to put data in a format that turns data into information. This is important for working with text files, databases, and when you send data from the back end to the front end. Below is an example of string building using a *for loop*.

```
number_builder = ""
i = 0

while i <= 50:
    if i % 2 == 0:
        number_builder += str(i) + " "
    i += 1
print(number_builder)
```

Here, every time **i** is even, it gets cast as a string and added to the **number_builder** string (which starts off empty - (“”)) until **i** is greater than 50.

Another way to do this is with the **.join()** method you've just learned about. As mentioned, this function takes a list and joins the elements together to make a string. We can rewrite the above example as below to incorporate **.join()**:

```
number_builder = [] #note the variable has to be a list rather than a string
i = 0

while i <= 50:
    if i % 2 == 0:
        number_builder.append(str(i))
    i += 1
print(" ".join(number_builder))
```

Here, we have made **number_builder** a list and, for each iteration of the loop, an even number gets appended to the list. Finally, in the print statement, the elements are all joined together with a space in between. You may have noticed that **i** is cast to a string before being appended. This is because you cannot make an integer act like a string without casting it - only strings and characters can be joined together.

For both examples, the output would be:

```
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50
```

A note from our coding mentor

Jared

Hey again, have you ever wondered where a string variable gets its name from? According to an **article on StackOverflow**:

“The 1971 OED (p. 3097) quotes an 1891 Century Dictionary on a source in the Milwaukee Sentinel of 11 Jan. 1898 (section 3, p. 1) to the effect that this is a compositor's term. Printers would paste up the text that they had generated in a long strip of characters.”

Instructions

- First, read and run the **example files** provided. Feel free to write and run your own example code before doing the compulsory task to become more comfortable with the concepts covered in this task.

Compulsory Task 1

Follow these steps:

- Create a program called **alternative.py** that reads in a string and makes each **alternate character** an upper case character and each other alternate character a lower case character.

  e.g. The string “Hello World” would become “HeLlO WoRlD”

- Now, try starting with the same string but making each **alternative word** lower and upper case.

  e.g. The string: “I am learning to code” would become “i AM learning TO code”.

  Tip: Using the split and join functions will help you here.

Rate us

Share your thoughts

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.