Working with strings

Introduction

Strings are an example of a data type. There are several different data types in python, so far we have only worked with strings (`str`) and integers (`int`).

String data is always treated as text by the program, even if it contains numeric characters. You cannot do mathematical calculations (addition, subtraction, etc.) with strings. They have to be converted (cast) to a numeric data type (`int` , `float`) first.

Printing strings

As you have already seen, strings can be printed using the `print()` function:

```
print("Hello world!")
print('Hello Python!')
```

Note that strings must be surrounded by quotes - either single (apostrophe key) or double (quotation mark key). Either will work for simple text. You will want to choose judiciously when including apostrophes for quotes in your text, though.

Let's say you want to print the statement `Let's get started!` in your program. If you use single quotes, Python will get confused as to where the string ends:

```
print('Let's get started')
```

Python will consider the string to be defined by the first 2 quotes (`Let`) and it will try to make the rest of the phrase into a variable. It isn't a valid variable name, and you'll get an error.

To print this phrase, use double quotes around the entire string:

```
print("Let's get started")
```

Similarly, if you wanted to include quotation marks in your string, you would use single quotes at the start and end.

```
print('She said, "I like Python!"')
```

What if you want to include both an apostrophe and quotes in your string? You can use something called escape characters - more on that later. (See the section below on escape characters!)

Concatenation

Strings can be stuck together (concatenated) using the `+` operator. For example, `'cat' + 'dog'` will create the string `'catdog'` and `'2' + '3'` will give `'23'` . Note that this is a string and should not be confused with the number `23` .

Storing strings in variables

Strings can be assigned to variables using the assignment operator (`=`):

```
school = 'Athenian'        # assigns the string 'Athenian' to the variable school
```

Printing, revisited

There are multiple ways to print combinations of strings. Consider the following example.

```
>>> pet = input('What kind of pet do you have?')
dog
>>> print('I have a ' + pet + ' too!' )
I have a dog too!
```

The user is prompted to type the kind of pet they have (`dog`), and a response is printed. The parameter in the print function is 3 separate strings that are concatenated. Note the space at the end of the first string and at the start of the last string. Python does not place spaces in between strings when concatenating them, so you need to add them for your sentences to be formatted correctly.

Another way to achieve the same result is to print the strings separately, without concatenation:

```
>>> pet = input('What kind of pet do you have?')
dog
>>> print('I have a', pet, 'too!' )
I have a dog too!
```

Note that the extra spaces are not needed here; Python automatically places spaces between each string here since they are separate strings that are printed sequentially rather than being concatenated.

Finally, there is a relatively new way using "formatted strings" or "f-strings."

```
>>> pet = input('What kind of pet do you have?')
dog
>>> print(f'I have a {pet} too!' )
I have a dog too!
```

In this case, the entire sentence is included as a string, with the variable inserted in {}. Note the `f` in front of the string; the `f` stands for format and tells Python that the string is a formatted string.

F-strings make using multiple variables in one sentence - or paragraph - much easier than the other two versions.

Escape Characters

Escape characters allow you to insert characters into a string that are typically not allowed. For example, you can use them to put quotes inside a string even when you've already used them to define the string.

```
print('Then they said, "Let\'s go to the game."')
```

Escape characters are defined by a backslash (`\`), so to include the extra apostrophe, use the backslash followed by the apostrophe. In the example above, both the single and double quotes are needed in the string. It's recommended to surround the string with the quotes that will allow you to use the fewest escape characters.

Other escape characters are:

Character	Result
`\t`	tab
`\n`	newline
`\"`	double quotes
`\\`	backslash

Operations

For the most part, mathematical operators don't work well with strings, which makes sense. For example, adding `2` to `dog` doesn't make much sense. However, you can multiply `dog` by a number. Just as multiplying a number by two is the same as adding two of that number together, multiplying a string by 2 concatenates the string with itself:

```
>>> 2 * 'dog'
'dogdog'

>>> 4 * 'spam'
'spamspamspamspam'  # 4 spams stuck together
```

You can do some logical comparisons of strings (equal to (`==`) and not equal to (`!=`)). Another operator (`in`) checks to see if a character is *in* a string. These are all very useful and will be revisited in the section on Booleans & Conditionals.

Built-in Functions

Python has lots of string functions already included. This means that they have already been coded and we can use them by **calling** them (typing their names) and setting their parameters like this:

```
string_function_name(parameter)
```

You have already seen an example of a predefined function: `print()` . In this case, the parameter that is typed in the parentheses is whatever you want the function to print.

Some string functions take the string as their only parameter, while others need further parameters. All of these functions **return** a value. We will start by storing these values in variables, but will progress on to skipping that step and using the functions directly as needed. (That will become clear in the examples.)

The advantages of using built-in functions are:

- saving time
- they are pre-tested so we know that they work

There are lots of different string functions. These are some of the more common ones, but you can look others up online if you are curious - or if you need them.

Len - returns the number of characters in a string.

The `len()` function returns the number of characters in a string. It takes the string as its parameter. Note that spaces are counted as characters, as are any punctuation marks.

Try this in the command line:

```
>>> len('aardvark')
8

>>> len('python')
6
```

If we want to use the returned value, we can assign it to a variable. In the command line, it could look like this:

```
>>> word_length = len('parrot')

>>> word_length
6
```

(Remember that the command line dispays the value of the variable without the `print()` function.)

In the editor, the same example would look like this:

```
word_length = len('parrot')
print(word_length)
```

The value of `word_length` (6) would be displayed in the output window.

We can also get a word from the user and then find the length:

```
word = input('Enter a word: ')
word_length = len(word)
print(word_length)
```

An even shorter way of doing this is to use the `len()` function inside the `print()` function:

```
word = input('Enter a word: ')
print(len(word))
```

Whether you use this condensed form is up to you. It involves less typing, but it can be harder to read and understand quickly. If you find it easier to keep track of everything when you write it step by step, do that!

Task: Word-length calculator

- ask the user for a word
- use `len()` to determine the length of the word
- print a response telling the user the length of their word. Use the word and then length in your printed response.

Slices

Slicing returns one or more characters from a string. Each character is given an index number starting with 0.

```
>>> word = 'banana'
>>> word[0]
'b'
>>> word[2]
'n'
>>> word[5]
'a'
```

Note that the six-letter word above has indices 0 through 5. If you reference an invalid index, you will get an error.

Slicing can also return more than 1 letter. If you provide two indices, Python will return the letters between them.
Note that the letter at the first index is returned along with the letters **up to, but not including** the second index.

```
>>> word = 'giraffe'
print(word[0:3])
```

The string `gir` will display when the code above is run.

Task

Write a program that:

- Asks the user to input a word and stores it in a suitably named variable
- Tells the user the length of the word
- Asks the user to input a number between 0 and the largest index and stores it in a suitably named variable.
- Prints a message with the letter at that index.
- Asks the user for a start index and an ending index.
- Prints the characters between those indices

Changing cases

Technically, these are **methods** rather than **functions** but we'll talk more about that later. For now, the main difference is that a parameter doesn't go in the parentheses.

These methods (functions in disguise) allow you to work around issues that arise with different cases (upper, lower). Remember that in Python, 'Yes' and 'yes' are different, even though you may want to treat them as being the same in your program.

Whatever the string is, it will be converted to upper case when string.upper() is used.
Whatever the string is, it will be converted to lower case when string.lower() is used.

```
>>> word = 'yes'
>>> word.upper()    # Converts the string to all uppercase
YES
>>> word.lower()    # Converts the string to all lowercase
```

```
yes
>>> word.title()    # Converts the string to titlecase (1st letter capitalized)
Yes
```

The original string can be in any combination of upper/lower case, it doesn't have to be all caps or all lowercase.
Just like with other functions we've used, this technique can be combined with conditions used in selection and iteration.

Note that the title() method will convert the first letter of every word in the string to uppercase:

```
>>> greeting = 'hello world'
>>> greeting.title()
'Hello World'
```