Chapter 6 - Strings

6 Strings

Strings are text that range from human readable to machine readable. They are the very words used in this course materials to convey information and they are also very widely used in any programming language. In Python, they are characters or words enclosed within a pair of single or double quotation marks or within 3 single or double quotation marks. They are most often used in the `print()` and `input()` functions to convey a message to the user.

There are a several types of Strings in Python and each are denoted differently

- *f* or *F* - strings used for formatting. Nomenclature: *f-strings*
- *r* or *R* - these are *raw* strings, they are used when there is a requirement to treat *Escape Sequences* characters like the newline characters `\n` as literals. Mainly used in Regular Expressions which we will learn later. Nomenclature: *raw strings*
- *fr* or *FR* - these are combination of raw f-strings.
- *u* - these are Unicode strings
- *b* - these strings are of type `bytes`

For this chapter, we will only be using normal ASCII text strings unless specified.

6.1 Basics

The most noticeable part about Strings in Python is that they are enclosed within either 1 or 3 counts of single or double quotation marks and those quotation marks are valid when used in combination. Example

```python
print("-'Hello World'-")
# Output: -'Hello World'-
# Note the single quotation marks are shown
```

Multiline strings are normally enclosed with triple quotation marks but it is also possible to use single quotation marks, the plus (`+`) operator and a backslash (`\`) character. The plus operator is to concatenate a string and the backslash character is to tell the interpreter that there are more strings after it. Note that there **must not** be any whitespace behind the backslash as that would result in a syntax error.

```python
# single quotation, long string
print('This is a ' + \
      'long string with' + \
      'single quotation marks.')
```

As strings are also a Python objects, it has its own set of attributes and functions. Accessing and manipulating Strings can be done via indexing/slicing or through String functions. As Strings are literals in Python, manipulating Strings will always involve the creation of new Strings internally.

Refer back to the sub chapter on *Concept of Indexing and Slicing* in the Lists & Tuples chapter if a refresher is required on. Accessing of String elements via indexing and/or slicing.

```python
var1 = "Hello, Python!"

print("var1[0]:", var1[0])  # output: H
print("var1[1:5]:", var1[1:5])  # output: ello
print("every 3rd character (starting at index 0):", var1[::3])
# output: every 3rd character (starting at index 0): Hl tn
print("every 3rd character (right to left):", var1[::-3])
# output: every 3rd character (right to left): !hPoe
print("Neat trick to reverse a string:", var1[::-1])
# output: Neat trick to reverse a string: !nohtyP ,olleH
```

6.2 Escape Characters

Strings also contains a list of escape characters (characters with the backslash `\` character placed before them). The definition of *Escape Characters* from Wikipedia is as follows

> In computing and telecommunication, an escape character is a character that invokes an alternative interpretation on subsequent characters in a character sequence.

Escape characters are dependent on the context upon which the application is using them and Python has a list of supported escape characters here. A shortened list of escape characters is shown in the table below

Escape Characters	Meaning
\n	ASCII Linefeed
\r	ASCII Carriage Return
\t	ASCII Horizontal Tab
\\	Backslash (\)
\"	Double quotation mark
\'	Single quotation mark

6.3 Special Operators

Strings also have special operators. Assume that the variables `a='hello'` and `b='world'`.

Operator	Description	Example
+	Concatenation - Creates a new string by joining the string on either side of the operator together.	`a+b` will give `helloworld`
*	Repetition - Creates a new string via concatenating multiple copies of the same string.	`a*2` will give `hellohello`
[*index*]	Indexing - Returns the character with a given index. As shown some paragraphs above.	`a[1]` will give `e`
[*start*:*end*]	Range Slicing - Returns the characters within a given index range. As shown some paragraphs above.	`a[1:5]` will give `ello`
in	Membership - Returns `True` if a character exists in the given string.	`e in a` will return `True`
not in	Membership - Returns `True` if a character does not exists in the given string.	`z not in a` will return `True`
%	For legacy string formating.	See section after this

6.4 String Formatting

There are many ways to format strings for printing and we will be looking at some legacy and new methods. Legacy string formatting methods are applicable throughout all Python versions unless stated.

Formatting using the `%` operator
The `%` operator is from the C/C++ programming language for String formatting where some of the datatypes are mapped as follows

- `%s` - stands for string
- `%d` - stands for integers
- `%f` - stands for floats

This form of mapping is not recommended to be used by the Python docs as it does not format all Python datatypes properly. The in-depth information can be read from the Python docs here.

Formatting using the `str.format()` function
This is a newer version of string formatting available since Python 2.6. This version of of string formatting requires the use of the curly brackets `{}` as a placeholder upon which the `format()` function will replace them with a value. There are several ways from which the `format()` function can be used. Example

```python
name = 'John'
age = 54
person = {'name':'Tom', 'age':85}

# method 1: first come, first served replacement
print('Hello, {}. You are {}.'.format(name, age))

# method 2: replacement via index referencing
# 'name' is index 0 and 'age' is index 1
print('Hello, {0}. You are {1}.'.format(name, age))

# method 3: replacement via variable names
print('Hello, {name_f}. You are {age_f}.'.format(name_f=name, age_f=age))

# method 4: replacement via dictionary (2 ways)
print('Hello, {name_f}. You are {age_f}.'.format(name_f=person['name'],
                                                  age_f=person['age']))
print('Hello, {name}. You are {age}.'.format(**person))
```

With longer strings, this method is can become cumbersome and unreadable.

Formatting using f-Strings
As mentioned at the start of the this topic, the character *f* or *F* at the start of a String means that the string has some formatting information combined into it. *f-strings* are the latest method used for string formatting since the release of Python 3.6. The full documentation is available here. Not only does *f-strings* allows conventional string formatting (similar to method 3 of the `str.format()` function) but it also allows expressions to be evaluated within its curly brackets. Example

```python
name = 'John'
age = 54

# note the 'f' prefix
print(f'Hello, {name}. You are {age}.')
# evaluating the expression 10*2
print(f'{10*5}')
# evaluating the expression str.lower()
print(f'Lower case name {name.lower()}')

# multiline
# Either enclosing the string with rounded brackets or
# using a backslash '\' character at the end of each line
msg = (f'Hi {name}. ' + \
       f'You are {age}')
print(msg)
```

f-strings also allow escape characters such as quotation marks to be printed. The trick is to mix and match the right type of quotation marks and place them **outside** the expression curly brackets. For instance, if a string uses double quotation marks to signify a string literal, you can use single quotation marks within the string for other purposes or use the escape character for quotation marks. Example

```python
print(f"\"{name}\"")
```

More examples of *f-string* formatting via the expression `{<variable>:<width>.<precision>}`.

- floats

```python
float_val = 85.5842

# printing the floating point value up to 2 decimal places where
# 'float_val' is the variable name
# '2f' specifies precision value (in this case 2 decimal places)
print(f'{float_val:.2f}')
```

- string output width

```python
for x in range(1, 11):
    # printing numbers with either a '0' padding or white space padding
    # '02' means a 2 digit character width with 0 padding in front if the
    # digit contains contains less than 2 characters
    # '3' means a 3 digit character width and whitespace padding in front
    # if the digit contains less than 3 characters
    print(f'{x:02} {x*x:3} {x*x*x:4}')
```

- numeric notations

```python
num = 200

# hexadecimal notation
print(f"{num:x}")
# octal notation
print(f"{num:o}")
# scientific notation
print(f"{num:e}")
```

6.5 String Functions

As mentioned in the earlier chapter *Standard Data Types*, Strings are objects and therefore it has attributes and behaviours (functions). The full list of String attributes and functions can be found here but we will be going through the most commonly used functions that you may require. Note that String functions have the notation `str.function_name()`, this means that the function is only callable using a string object.

- `len()` - this function is part of the built-in set of functions provided by Python but it is also very useful for many different datatypes including Strings.

```python
str_var = 'hello foo bar'
print(len(str_var))
# output: 13
```

- `split()` - this function splits a string based on a delimiter. The default delimiter is a single whitespace character. This function returns a `List` of strings.

```python
sentence = "The, quick, brown, fox, jumps, over, the, fence"
# delimiter is a whitespace character and comma
print(sentence.split(", "))
# Outputs: ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the',
'fence']
```

- `join()` - this function concatenates a `List` of strings with the given separator. The separator must also be of type `String`. This function returns a `String`

```python
str_list = ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the',
'fence']
# separator is a whitespace character
print(" ".join(str_list))
# Outputs: The quick brown fox jumps over the fence
```

- `replace()` - this function replaces all occurrences of an old string with a new string. If a max number is given, the functions will replace the occurrences up to the maximum number. This function is useful if you know the exact sub string to replace as it works exactly like the "Find and Replace" function in any Word Processing software.

```python
sentence = "The quick brown fox jumps over the fence"

sentence.replace("brown", "red")
# output: 'The quick red fox jumps over the fence'
sentence.replace(" ", "-", 3)
# output: 'The-quick-brown-fox jumps over the fence'
```

- `strip()` - this function strips the unwanted characters from both ends of a string. It is most often used to strip the "padding" whitespaces or other characters from a string read from a file. There are 2 related strip functions called `rstrip()` and `lstrip()` which strips the unwanted characters from either the right or left of side of the string respectively.

```python
str_val = '!!!!!!boo!!!!!!!'
print(str_val.strip('!'))  # output: boo
print(str_val.rstrip('!'))  # output: !!!!!!boo
print(str_val.lstrip('!'))  # output: boo!!!!!!!
```

- `startswith()` or `endswith()` - these are functions which are used to check if a string starts with or ends with a certain string.

```python
quote = "Now cracks a noble heart. Good-night, sweet prince;" + \
        "And flights of angels sing thee to thy rest."

print(quote.startswith("Now"))  # output: True
print(quote.endswith("rest."))  # output: True
```

- `find()` or `index()` - these 2 functions have the same functionality but returns different results when the given search string is not found. The functionality of these function is to find the first occurrence of a given search string (from the direction left to right) and return its index. The `find()` function will return a `-1` if the string is not found whereas the `index()` function will raise an exception. Both `find()` and `index()` functions have related functions to search a string from the direction right to left called `rfind()` and `rindex()`.

```python
quote = "Now cracks a noble heart. Good-night, sweet prince;" + \
        "And flights of angels sing thee to thy rest."

print(quote.find("noble"))  # output: 13
print(quote.rindex("angels"))  # output: 66
```

- `lower()` and `upper()` - these functions are used to either change a string to fully lower case or upper case characters. These functions are especially useful for string comparisons between externally sourced string data and internally string data used for processing.

```python
str_val = 'BanKing InDustrY'

print(str_val.lower())  # output: banking industry
print(str_val.upper())  # output: BANKING INDUSTRY
```

- `center()`, `ljust()`, `rjust()` - these functions deal with the alignment of a string with given the length of characters and a character to fill the spaces with. The default fill character is a whitespace.

```python
str_val = 'bar'
fill_char = '-'
new_str_len = 10

print(str_val.center(new_str_len, fill_char))
# output: ---bar----
print(str_val.ljust(new_str_len, fill_char))
# output: bar-------
print(str_val.rjust(new_str_len, fill_char))
# output: -------bar
```

6.6 References

1. Python 3 - Strings, https://www.tutorialspoint.com/python3/python_strings.htm
2. Text Sequence Type — str, https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str
3. Lubanovic, 2019, 'Chapter 5. Text Strings' in Introducing Python, 2nd Edition, O`Reilly Media, Inc.