Chapter 4
Python Strings

4.1 Introduction

In the previous chapter we used strings several times, both as prompts to the user and as output from the `print()` function. We even had the user type in their name and store it in a variable that could be used to access this name at a later point in time. In this chapter we will explore what a string is and how you can work with and manipulate them.

4.2 What Are Strings?

During the description of the *Hello World* program we referred to Python strings several times, but what is a string?

In Python a string is a series, or sequence, of characters in order. In this definition a *character* is anything you can type on the keyboard in one keystroke, such as a letter 'a', 'b', 'c' or a number '1', '2', '3' or a special characters such as '\', '[', '$' etc. a space is also a character ' ', although it does not have a visible representation.

It should also be noted that strings are *immutable*. Immutable means that once a string has been created it cannot be changed. If you try to change a string you will in fact create a new string, containing whatever modifications you made, you will not affect the original string in anyway. For the most part you can ignore this fact but it means that if you try to get a sub string or split a string you must remember to store the result—we will see this later on in this chapter.

To define the start and end of a string we have used the single quote character ', thus all of the following are valid strings:

J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_4

- `'Hello'`
- `'Hello World'`
- `'Hello Andrea2000'`
- `'To be or not to be that is the question!'`

We can also define an empty string which has no characters in it (it is defined as a single quote followed immediately by a second single quote with no gap between them). This is often used to initialise or reset a variable holding a reference to a string, for example

- `some_string = ''`

4.3 Representing Strings

As stated above; we have used single quotes to define the start and end of a string, however in Python single or double quotes can be used to define a string, thus both of the following are valid:

- `'Hello World'`
- `"Hello World"`

In Python these forms are exactly the same, although by convention we default to using single quotes. This approach is often referred to as being more Pythonic (which implies it is more the convention used by experienced Python programmers) but the language does not enforce it.

You should note however, that you cannot mix the two styles of start and end strings, that is you cannot start a string with a single quote and end a string with a double quote, thus the following are both illegal in Python:

- 'Hello World" # This is illegal
- "Hello World' # So is this

The ability to use both " and ' however, comes in useful if your string needs to contain one of the other type of string delimiters. This is because a single quote can be embedded in a string defined using double quotes and vice versa, thus we can write the following:

```
print("It's the day")
print('She said "hello" to everyone')
```

The output of these two lines is:

```
It's the day
She said "hello" to everyone
```

A third alternative is the use of triple quotes, which might at first hand seem a bit unwieldy, but they allow a string to support multi-line strings, for example:

```
z = """
Hello
  World
"""
print(z)
```

Which will print out

```
Hello
  World
```

4.4 What Type Is String

It is often said that Python is untyped; but this is not strictly true—as was stated in an earlier chapter it is a dynamically typed language with all data having an associated type.

The type of an item of data (such as a string) determines what it is legal to do with the data and what the effect of various actions will be. For example, what the effect of using a '+' operator is will depend on the *types* involved; if they are numbers then the plus operator will add them together; if however strings are involved then the strings will be concatenated (combined) together etc.

It is possible to find out what type a variable currently holds using the built-in `type()` function. This function takes a variable name and will return the type of the data held by that variable, for example:

```
my_variable = 'Bob'
print(type(my_variable))
```

The result of executing these two lines of code is the output:

```
<class 'str'>
```

This is a shorthand for saying that what is held in `my_variable` is currently a class (type) of string (actually string is a class and Python supports ideas from Object Oriented Programming such as classes and we will encounter them later in the book).

4.5 What Can You Do with Strings?

In Python terms this means what operations or functions are their available or built-in that you can use to work with strings. The answer is that there are very many. Some of these are described in this section.

4.5.1 String Concatenation

You can concatenate two strings together using the '+' operator (an operator is an operation or behaviour that can be applied to the types involved). That is you can take one string and add it to another string to create a new third string:

```
string_1 = 'Good'
string_2 = " day"
string_3 = string_1 + string_2
print(string_3)
print('Hello ' + 'World')
```

The output from this is

```
Good day
Hello World
```

Notice that the way in which the string is defined does not matter here, `string_1` used single quotes but `string_2` used double quotes; however they are both just strings.

4.5.2 Length of a String

It can sometimes be useful to know how long a string is, for example if you are putting a string into a user interface you might need to know how much of the string will be displayed within a field. To find out the length of a string in Python you use the `len()` function, for example:

```
print(len(string_3))
```

This will print out the length of the string currently being held by the variable `string_3` (in terms of the number of characters contained in the string).

4.5.3 Accessing a Character

As a string is a fixed sequence of letters, it is possible to use square brackets and an index (or position) to retrieve a specific character from within a string. For example:

```
my_string = 'Hello World'
print(my_string[4])
```

However, you should note that strings are indexed from *Zero*! This means that the first character is in position 0, the second in position 1 etc. Thus stating `[4]` indicates that we want to obtain the fifth character in the string, which in this case is the letter 'o'. This form of indexing elements is actually quite common in programming languages and is referred to a *zero based indexing*.

4.5.4 Accessing a Subset of Characters

It is also possible to obtain a subset of the original string, often referred to as a substring (of the original string). This can again be done using the square brackets notation but using a ':' to indicate the start and end points of the sub string. If one of the positions is omitted then the start or end of the string is assumed (depending upon the omission), for example:

```
my_string = 'Hello World'
print(my_string[4])   # characters at position 4
print(my_string[1:5]) # from position 1 to 5
print(my_string[:5])  # from start to position 5
print(my_string[2:])  # from position 2 to the end
```

Will generate

```
o
ello
Hello
llo World
```

As such `my_string[1:5]` returns the substring containing the 2nd to 6th letters (that is `'ello'`). In turn `my_string[:5]` returned the substring containing the 1st to 6th letters and `my_string[2:]` the sub string containing the 3rd to the last letters.

4.5.5 Repeating Strings

We can also use the '*' operator with strings. In the case of strings this means repeat the given string a certain number of times. This generates a new string containing the original string repeated *n* number of times. For example:

```
print('*' * 10)
print('Hi' * 10)
```

Will generate

```
**********
HiHiHiHiHiHiHiHiHiHi
```

4.5.6 Splitting Strings

A very common requirement is the need to split a string up into multiple separate strings based on a specific character such as a space or a comma.

This can be done with the `split()` function, that takes a string to used in identifying how to split up the receiving string. For example:

```
title = 'The Good, The Bad, and the Ugly'
print('Source string:', title)
print('Split using a space')
print(title.split(' '))
print('Split using a comma')
print(title.split(','))
```

This produces as output

```
Source string: The Good, The Bad, and the Ugly
Split using a space
['The', 'Good,', 'The', 'Bad,', 'and', 'the', 'Ugly']
Split using a comma
['The Good', ' The Bad', ' and the Ugly']
```

As can be seen from this the result generated is either a list of each word in the string or three strings as defined by the comma.

You may have noticed something odd about the way in which we wrote the call to the split operation. We did not pass the string into `split()` rather we used the format of the variable containing the string followed by '.' and then `split()`.

This is because `split()` is actually what is referred to as a *method*. We will return to this concept more when we explore classes and objects. For the moment merely remember that methods are *applied* to things like string using the *dot* notation.

For example, given the following code

```
title = 'The Good, The Bad, and the Ugly'
print(title.split(' '))
```

This means take the string held by the variable title and split it based on the character space.

4.5.7 Counting Strings

It is possible to find out how many times a string is repeated in another string. This is done using the `count()` operation for example

```
my_string = 'Count, the number     of spaces'
print("my_string.count(' '):", my_string.count(' '))
```

Which has the output

```
my_string.count(' '): 8
```

indicating that there are 8 spaces in the original string.

4.5.8 Replacing Strings

One string can replace a substring in another string. This is done using the `replace()` method on a string. For example:

```
welcome_message = 'Hello World!'
print(welcome_message.replace("Hello", "Goodbye"))
```

The output produced by this is thus

```
Goodbye World!
```

4.5.9 Finding Sub Strings

You can find out if one string is a substring of another string using the `find()` method. This method takes a second string as a parameter and checks to see if that string is in the string receiving the `find()` method, for example:

```
string.find(string_to_find)
```

The method returns −1 if the string is not present. Otherwise it returns an index indicating the start of the substring. For example

```
print('Edward Alun Rawlings'.find('Alun'))
```

This prints out the value 5 (the index of the first letter of the substring 'Alun' note strings are indexed from Zero; thus the first letter is at position Zero, the second at position one etc.

In contrast the following call to the `find()` method prints out −1 as 'Alun' is no longer part of the target string:

```
print('Edward John Rawlings'.find('Alun'))
```

4.5.10 Converting Other Types into Strings

If you try to use the '+' concatenation operator with a string and some other type such as a number then you will get an error. For example if you try the following:

```
msg = 'Hello Lloyd you are ' + 21
print(msg)
```

You will get an *error* message indicating that you can only concatenate strings with strings not integers with strings. To concatenate a number such as 21 with a string you must convert it to a string. This can be done using the `str()` function. This contest any type into a string representation of that type. For example:

```
msg = 'Hello Lloyd you are ' + str(21)
print(msg)
```

This code snippet will print out the message:

```
Hello Lloyd you are 21
```

4.5.11 Comparing Strings

To compare one string with another you can use the '==' equality and '!=' not equals operators. These will compare two strings and return either `True` or `False` indicating whether the strings are equal or not.

For example:

```
print('James' == 'James') # prints True
print('James' == 'John')  # prints False
print('James' != 'John')  # prints True
```

You should note that strings in Python are case sensitive thus the string 'James' does not equal the string 'james'. Thus:

```
print('James' == 'james') # prints False
```

4.5.12 Other String Operations

There are in fact very many different operations available for strings, including checking that a string starts or ends with another string, that is it upper or lower case etc. It is also possible to replace part of a string with another string, convert strings to upper, lower or title case etc.

Examples of these are given below (note all of these operations use the dot notation):

```
some_string = 'Hello World'
print('Testing a String')
print('-' * 20)
print('some_string', some_string)
print("some_string.startswith('H')",
some_string.startswith('H'))
print("some_string.startswith('h')",
some_string.startswith('h'))
print("some_string.endswith('d')", some_string.endswith('d'))
print('some_string.istitle()', some_string.istitle())
print('some_string.isupper()', some_string.isupper())
print('some_string.islower()', some_string.islower())
print('some_string.isalpha()', some_string.isalpha())

print('String conversions')
print('-' * 20)
print('some_string.upper()', some_string.upper())
print('some_string.lower()', some_string.lower())
print('some_string.title()', some_string.title())
print('some_string.swapcase()', some_string.swapcase())
print('String leading, trailing spaces', "   xyz   ".strip())
```

The output from this is

```
Testing a String
--------------------
some_string Hello World
some_string.startswith('H') True
some_string.startswith('h') False
some_string.endswith('d') True
some_string.istitle() True
some_string.isupper() False
some_string.islower() False
some_string.isalpha() False
String conversions
--------------------
some_string.upper() HELLO WORLD
some_string.lower() hello world
some_string.title() Hello World
some_string.swapcase() hELLO wORLD
String leading, trailing spaces xyz
```

4.6 Hints on Strings

4.6.1 Python Strings Are Case Sensitive

In Python the string 'l' is not the same as the string 'L'; one contains the lower-case letter 'l' and one the upper-case letter 'L'. If case sensitively does not matter to you then you should convert any strings you want to compare into a common case before doing any testing; for example using `lower()` as in

```
some_string.lower().startswith('h')
```

4.6.2 Function/Method Names

Be very careful with capitalisation of function/method names; in Python `isupper()` is a completely different operation to `isUpper()`. If you use the wrong case Python will not be able to find the required function or method and will generate an error message. Do not worry about the terminology regarding functions and methods at this point—for now they can be treated as the same thing and only differ in the way that they recalled or invoked.

4.6.3 Function/Method Invocations

Also be careful of always including the round brackets when you call a function or method; even if it takes no parameters/arguments. There is a significant difference between `isupper` and `isupper()`. The first one is the *name* of an operation on a string while the second is a call to that operation so that the operation executes. Both formats are legal Python but the result is very different, for example:

```
print(some_string.isupper)
print(some_string.isupper())
```

produces the output:

```
<built-in method isupper of str object at 0x105eb19b0>
False
```

Notice that the first print out tells you that you are referring to the built-in method called `isupper` defined on the type String; while the second actually runs `isupper()` for you and returns either `True` or `False`.

4.7 String Formatting

Python provides a sophisticated formatting system for strings that can be useful for printing information out or logging information from a program.

The string formatting system uses a special string known as the *format* string that acts as a pattern defining how the final string will be laid out. This format string can contain placeholders that will be replaced with actual values when the final string is created. A set of values can be applied to the format string to fill the placeholders using the `format()` method.

The simplest example of a *format* string is one that provides a single placeholder indicated by two curly braces (e.g. `{}`). For example, the following is a *format* string with the pattern 'Hello' followed by a placeholder:

```
format_string = 'Hello {}!'
```

This can be used with the `format()` string method to provide a value (or populate) the placeholder, for example:

```
print(format_string.format('Phoebe'))
```

The output from this is:

```
Hello Phoebe!
```

A *format* string can have any number of placeholders that must be populated, for example the next example has two placeholders that are populated by providing two values to the `format()` method:

```
# Allows multiple values to populate the string
name = "Adam"
age = 20
print("{} is {} years old".format(name, age))
```

In this case the output is:

```
Adam is 20 years old
```

It also illustrates that variables can be used to provide the values for the `format` method as well as literal values. A literal value is a fixed value such as 42 or the string 'John'.

By default the values are bound to the placeholders based on the order that they are provided to the `format()` method; however this can be overridden by providing an *index* to the placeholder to tell it which value should be bound, for example:

```
# Can specify an index for the substitution
format_string = "Hello {1} {0}, you got {2}%"
print(format_string.format('Smith', 'Carol', 75))
```

In this case the second string 'Carol' will be bound the first placeholder; note that the parameters are numbered from *Zero* not one.

The output from the above example is:

```
Hello Carol Smith, you got 75%
```

Of course when ordering the values it is quiet easy to get something wrong either because a developer might think the strings are indexed from 1 or just because they get the order wrong.

An alternative approach is to use *named* values for the placeholders. In this approach the curly brackets surround the name of the value to be substituted, for example `{artist}`. Then in the `format()` method a *key=value* pair is provided where the key is the name in the *format* string; this is shown below:

```
# Can use named substitutions, order is not significant
format_string = "{artist} sang {song} in {year}"
print(format_string.format(artist='Paloma Faith',
song='Guilty', year=2017))
```

In this example the order no longer matters as the name associated with the parameter passed into the `format()` method is used to obtain the value to be substituted. In this case the output is:

```
Paloma Faith sang Guilty in 2017
```

It is also possible to indicate alignment and width within the format string. For example, if you wish to indicate a width to be left for a placeholder whatever the actual value supplied, you can do this using a colon (':') followed by the width to use. For example to specify a gap of 25 characters which can be filled with a substitute value you can use `{:25}` as shown below:

```
print('|{:25}|'.format('25 characters width'))
```

In the above the vertical bars are merely being used to indicate where the string starts and ends for reference, they have no meaning within the format method. This produces the output:

```
|25 characters width      |
```

Within this gap you can also indicate an alignment where:

- < Indicates left alignment (the default),
- > Indicates right alignment,
- ^ Indicates centered.

These follow the colon (':') and come before the size of the gap to use, for example:

```
print('|{:<25}|'.format('left aligned')) # The default
print('|{:>25}|'.format('right aligned'))
print('|{:^25}|'.format('centered'))
```

Which produces:

```
|left aligned              |
|             right aligned|
|         centered          |
```

Another formatting option is to indicate that a number should be formatted with separators (such as a comma) to indicate thousands:

```
# Can format numbers with comma as thousands separator
print('{:,}'.format(1234567890))
print('{:,}'.format(1234567890.0))
```

Which generates the output:

```
1,234,567,890
1,234,567,890.0
```

There are in fact numerous options available to control the layout of a value within the format string and the Python documentation should be referenced for further information.

4.8 String Templates

An alternative to using string formatting is to use string *Templates*. These were introduced into Python 2.4 as a simpler, less error prone, solution to most string formatting requirements.

A string template is a class (type of thing) that is created via the `string.Template()` function. The template contains one or more *named* variables preceded with a `$` symbol. The Template can then be used with a set of values that replace the template variables with actual values.

For example:

```
import string

# Initialise the template with $variables that
# will be substitute with actual values
template = string.Template('$artist sang $song in $year')
```

Note that it is necessary to include an *import* statement at the start of the program as Templates are not provided by default in Python; they must be loaded from a library of additional string features. This library is part of Python but you need to tell Python that you want to access these extra string facilities.

We will return to the `import` statement later in the book; for now just except that it is needed to access the Template functionality.

The Template itself is created via the `string.Template()` function. The string passed into the `string.Template()` function can contain any characters plus the template variables (which are indicated by the `$` character followed by the name of the variable such as `$artist` above).

The above is thus a template for the pattern '*some-artist* sang *some-song* in *some-year*'.

The actual values can be substituted into the template using the `substitute()` function. The substitute function takes a set of *key*=*value* pairs, in which the *key* is the name of the template variable (minus the leading `$` character) and the *value* is the value to use in the string.

```
print(template.substitute(artist='Freddie Mercury', song='The
Great Pretender', year=1987))
```

In this example `$artist` will be replaced by 'Freddie Mercury', `$song` by 'The Great Pretender' and `$year` by 1987. The substitute function will then return a new string containing

```
'Freddie Mercury sang The Great Pretender in 1987'
```

This is illustrated in the following code:

```
import string

# Initialise the template with $variables that
# will be substitute with actual values
template = string.Template('$artist sang $song in $year')

# Replace / substitute template variables with actual values
# Can use a key = value pairs where the key is the name of
# the template Variable and the value is the value to use
# in the string
print(template.substitute(artist='Freddie Mercury', song='The
Great Pretender', year=1987))
```

This produces:

```
Freddie Mercury sang The Great Pretender in 1987
```

We can of course reuse the template by substituting other values for the template variables, each time we call the `substitute()` method it will generate a new string with the template variables replaced with the appropriate values:

```
print(template.substitute(artist='Ed Sheeran', song='Galway
Girl', year=2017))
print(template.substitute(artist='Camila Cabello',
song='Havana', year=2018))
```

With the above producing:

```
Ed Sheeran sang Galway Girl in 2017
Camila Cabello sang Havana in 2018
```

Alternatively you can create what is known as a dictionary. A dictionary is a structure comprised of *key:value* pairs in which the key is unique. This allows a *data structure* to be created containing the values to use and then applied to the substitute function:

```
d = dict(artist = 'Billy Idol', song='Eyes Without a Face',
year = 1984)
print(template.substitute(d))
```

This produces a new string:

```
Billy Idol sang Eyes Without a Face in 1984
```

We will discuss dictionaries in greater detail later in the book.

Template strings can contain template variables using the format `$name-of-variable`; however there are a few variations that are worth noting:

- `$$` allows you to include a '$' character in the string without Python interpreting it as the start of a template variable this the double '$$' is replaced with a single $. This is known as *escaping a control* character.
- `${template_variable}` is equivalent to `$template_variable`. It is required when valid identifier characters follow the placeholder but are not part of the placeholder, such as "`${noun}ification`".

Another point to note about the `template.substitute()` function is that if you fail to provide all the template variables with a value then an *error* will be generated. For example:

```
print(template.substitute(artist='David Bowie', song='Rebel
Rebel'))
```

Will result in the program failing to execute and an error message being generated:

```
Traceback (most recent call last):
  File "/emplate_examples.py", line 16, in <module>
    print(template.substitute(artist='David Bowie', song='Rebel
Rebel'))
  File "/Library/Frameworks/Python.framework/Versions/3.7/lib/
python3.7/string.py", line 132, in substitute
    return self.pattern.sub(convert, self.template)
  File "/Library/Frameworks/Python.framework/Versions/3.7/lib/
python3.7/string.py", line 125, in convert
    return str(mapping[named])
KeyError: 'year'
```

This is because the template variable `$year` has not been provided with a value.

If you do not want to have to worry about providing all the variables in a template with a value then you should use the `safe_substitute()` function:

```
print(template.safe_substitute(artist='David Bowie',
song='Rebel Rebel'))
```

This will populate the template variables provided and leave any other template variables to be incorporated into the string as they are, for example:

```
David Bowie sang Rebel Rebel in $year
```

4.9 Online Resources

There is a great deal of online documentation available on strings in Python including:

- https://docs.python.org/3/library/string.html which presents common string operations.
- https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str this provides information on strings and the str class in Python.
- https://pyformat.info has a simple introduction to Python string formatting.
- https://docs.python.org/3/library/string.html#format-string-syntax which presents detailed documentation on Python string formatting.
- https://docs.python.org/3/library/string.html#template-strings for documentation on string templates.

4.10 Exercises

We are going to try out some of the string related operations.

1. Explore replacing a string
   Create a string with words separated by ',' and replace the commas with spaces; for example replace all the commas in 'Denyse,Marie,Smith,21,London,UK' with spaces. Now print out the resulting string.
2. Handle user input
   The aim of this exercise is to write a program to ask the user for two strings and concatenate them together, with a space between them and store them into a new variable called `new_string`.

   Next:

- Print out the value of `new_string`.
- Print out how long the contents of `new_string` is.
- Now convert the contents of `new_string` to all upper case.
- Now check to see if `new_string` contains the string 'Albus' as a substring.