Python Module 4

February 3, 2023

1 Notebook For Learning Python Programming Language

2 Module 4

2.1 Python String Day

In this module, you'll learn all about Python Strings: slicing and striding, manipulating and formatting them with the Formatter class, f-strings, templates and more!

String is a collection of alphabets, words or other characters. It is one of the primitive data structures and are the building blocks for data manipulation. Python has a built-in string class named str. Python strings are "immutable" which means they cannot be changed after they are created. For string manipulation, we create new strings as we go to represent computed values because of their immutable property.

2.1.1 Strings

You can handle textual data in Python using the str object. Strings are immutable sequences of unicode. Unicode is a system designed to represent all characters from languages. In unicode, each letter, character is represented as a 4-byte number. Each number represents a unique character.

To represent a string, you wrap it within quotes. There can be multiple ways of doing this:

- Single quotes, just like in this example: ‘Single quote allow you to embed “double” quotes in your string.’
- Double quotes. For example: “Double quote allow you to embed ‘single’ quotes in your string.”
- Triple quotes, as in this example: “ ““Triple quotes using double quotes”“ “, ’‘’Triple quotes using single quotes.” ’

Triple quoted string let you work with multiple line strings, and all associated whitespace will be included in the string.

[ ]:
```
single_quote = 'Single quote allow you to embed "double" quotes in your string.'
double_quote = "Double quote allow you to embed 'single' quotes in your string."
triple_quote = """Triple quotes allows to embed "double quotes" as well as␣
  ↪'single quotes' in your string.
And can also span across multiple lines."""
```

Strings are immutable which means if you try to change any value in a string, it will throw an error. You must create a new string inorder to incorporate the changes.

[ ]:
```
triple_quote = '''This is triple quoted string using "single" quotes.'''
triple_quote[35] = "'"
```

[ ]:
```
triple_quote_new = triple_quote[0:35] + "'single'" + triple_quote[43:]
print(triple_quote_new)
```

You can find the length of a string using the built-in len() function:

[ ]:
```
len(triple_quote_new)
```

2.1.2 String Slicing in Python

Since strings are a sequence of characters, you can access it through slicing and indexing just like you would with Python lists or tuples. Strings are indexed with respect to each character in the string and the indexing begins at 0:

In the string above, the first index is `C` and it is indexed 0. The last character is a full-stop `.` which is the 16th character in the string. You can also access the characters in the opposite direction starting from -1, which means you can also use -1 as an index value to access `.` in the string. There is also a whitespace between `Chocolate` and `cookie`, this is also a part of the string and has its own index, 9th in this case. You can check this using slicing.

Because each character in a Python string has a corresponding index number, you can access and manipulate strings in the same ways as other sequential data types. Slicing is a technique in Python that allow you to specific element or a sub-set of elements from a container object using their index values. Slicing saves you from having to write loop statements to go through the indexes of your string to find or access certain substrings.

[ ]:
```
snack = "Chocolate cookie."
print(snack[0])
print(snack[9])
print(snack[-1])
```

Let's suppose you wanted to extract the substring 'cookie' from the string below. How would you do this?

In such cases, you use range slicing.

The syntax for range slicing is the following: [Start index (included): Stop index (excluded)]

[ ]:
```
snack = "Chocolate cookie."
print(snack[10:16])
```

You could also do this using negative value for the stop index:

[ ]:
```
print(snack[10:-1]) # -1: since the stop index is excluded in slicing.
```

Slicing without specifying the stop index means that you capture the characters from the start index upto the very last index in the sentence. Similarly, slicing with the start index missing means that you start from the very first index in the string to the stop index:

[ ]:
```
# Stop value not provided
print(snack[0:])

# Start value not provided (Stop value excluded according to syntax)
print(snack[:-1])

# This is also allowed
print(snack[:])
```

String slicing can also accept a third parameter, the `stride`, which refers to how many characters you want to move forward after the first character is retrieved from the string. The value of `stride` is set to `1` by default.

Let's see `stride` in action to understand it better:

[ ]:
```
number_string = "1020304050"
print(number_string[0:-1:2])
```

Tip: Something, very cool that you can do with striding is reverse a string:

[ ]:
```
print(number_string[::-1]) #
```

The value of `-1` for the stride allows you to start from the end character and then move one character at a time.

Alternatively, if you provide `-2` as a value, you start from the end character and move two characters at a time:

[ ]:
```
print(number_string[::-2]) #
```

2.1.3 Common String Operations

Slicing, range slicing are the common operations you would need to perform on strings. There is also string concatenation, which is as simple as addition:

[ ]:
```
string1 = 'Chocolate'
string2 = 'cookie'

snack = string1 + " " + string2
print(snack)
```

However, this will not work if you try to concatenate a string with some other data type.

[ ]:
```
cost = 15
string1 = "The total in Euro is: "

bill = string1 + cost
print(bill)
```

Here, you tried to concatenate a string with an integer value which is not allowed. The interpretor cannot implicitly understand whether you are trying to perform simple integer addition or string concatenation. However, try this now:

[ ]:
```python
bill = string1 + str(cost)
print(bill)
```

This is because you explicitly converted an integer to a string value and then applied concatenation.

To repeat a string, use the `*` operation.

[ ]:
```python
single_word = 'hip '
line1 = single_word * 2 + 'hurray! '
print(line1 * 3)
```

You can also check for membership property in a string using `in` and `not in`:

[ ]:
```python
sub_string1 = 'ice'
sub_string2 = 'glue'
string1 = 'ice cream'
if sub_string in string1:
    print("There is " + sub_string + " in " + string1)
if sub_string2 not in string1:
    print("Phew! No " + sub_string2 + " in " + string1)
```

Python provides many built-in methods or helper functions to manipulate strings. Replacing a substring, capitalizing certain words in a paragraph, finding the position of a string within another string are some of the operations you can do with these built-in methods.

Take a look at some of these in detail:

- 'str.capitalize()': returns a copy of the string with its first character capitalized.

[ ]:
```python
str.capitalize('cookie')
```

- `str.islower()`: returns true if all characters in the string are lowercase, false otherwise.

[ ]:
```python
snack = 'cookie'
snack.islower()
```

- `str.find(substring)`: returns the lowest index in the string where the substring is found. You can also specify the start and end index within the string where you want the substring to be searched for. Returns -1 if the substring is not found.

[ ]:
```python
str1 = 'I got you a cookie'
str2 = 'cook'
str1.find(str2)
```

- `str.count(substring)`: counts how many times a substring occurs in the string. You can also specify the start and the stop index for the string.

[ ]:
```python
str1 = 'I got you a cookie, do you like cookies?'
str2 = 'cookie'
str1.count(str2)
```

- `str.isspace()`: returns True if there are only whitespace characters in the string, false otherwise. Whitespace characters are the characters such as space, tab, next line, etc.

This can be useful when working with real life datasets, that might not always encode proper spacing during conversion from one format to another.

[2]:
```
str_space = '   '
str_space.isspace()
```

[ ]:
```
str_tab = '\t'
str_tab.isspace()
```

[ ]:
```
str_nextline = '''\n'''
str_nextline.isspace()
```

Note: Did you notice the `\t`, `\n` above? These are called escape characters. They start with a (backslash). Internally, they are not interpreted as normal strings, but rather as special characters that represent something else. For example - represents a tab. There are many more escape characters and you can read more about them in python documentation.

- `str.lstrip()`: removes all leading whitespace in string. This is another function that can be handy when you're working with real-life datasets.

[ ]:
```
str1 = " I can't hear you. Are you alright? "
str2 = " Yes, all is good."
str3 = str1.lstrip() + str2.lstrip()
print(str3)
```

- `str.isdigit()`: returns True if string contains only digits and False otherwise.

[ ]:
```
number_string = "1020304050"
number_string.isdigit()
```

- `str.replace(substring, new)`: replaces all occurrences of the substring in string with new. You can also define a third argument `max`, which replaces at most `max` occurrences of substring in the string. Remember that that is not an inplace replacement, which means the immutable property still holds and a new string is actually formed.

[ ]:
```
string1 = 'hip hip hurray! hip hip hurray! hip hip hurray!'
string2 = string1.replace('hip', 'Hip')
print(string1)
print(string2)
```

[ ]:
```
string1.replace('hip', 'Hip', 2)
```

- `str.split(delimiter="")`: splits the string according to the delimiter (space if not provided) and returns a list of substrings. dessert = 'Cake, Cookie, Icecream'

[ ]:
```
dessert = 'Cake, Cookie, Icecream'
list_dessert = string1.split(',')
```

You can find an exhaustive list of string methods in Python here.

2.1.4 String Formatting

Python supports multiple ways to format a string. In this section, you'll learn more about this formatting strings!

% Formatting The modulo `%` is a built-in operation in Python. It is known as the interpolation operator. You will need to provide `%` followed by the datatype that needs to be formatted or converted. The `%` operation then substitutes the '%datatype' phrase with zero or more elements of the specified data type:

```
[ ]: print("I bought %d Euro worth of %s!" %(200, 'cookies'))
```

You have seen `%d` used for integers and `%s` for strings. Some of the other conversion types available are: `o` for octal values, `x` for hexadecimal, `f` for floating point decimal format, `c` for single character (accepts integer or single character string).

The formatter class The `formatter` class is one of the built-in string class. It provides the ability to do complex variable substitutions and value formatting using the `format()` method. It allows you to create and customize your own string formatting behaviors by rewriting the public methods it contains: `format()`, `vformat()`. It has some methods that are intended to be replaced by subclasses: `parse()`, `get_field()`, `get_value()`, `check_unused_args()`, `format_field()` and `convert_field()`. However, for the sake of simplicity, you will check out only the most commonly used `format()` function in action...

```
[ ]: print("I bought {0} Euro worth of {1}!".format(200,'cookies')) #Accessing␣
      ↪values by position
```

```
[ ]: print("I bought {total} Euro worth of {item}!".format(total = 200, item =␣
      ↪'cookies')) #Accessing values by name
```

```
[ ]: '{:#<10}'.format('Cake') #Left aligment for word 'Cake' according to right␣
      ↪alignment, gaps filled with '#'
```

```
[ ]: '{:#^10}'.format('Cake') #Centre aligment for word 'Cake' according to right␣
      ↪alignment, gaps filled with '#'
```

```
[ ]: '{:#>10}'.format('Cake') #Right aligment for word 'Cake' according to right␣
      ↪alignment, gaps filled with '#'
```

```
[ ]: for num in range(1,10):
             print('{0:{width}}'.format(num, width=5), end=' ')
```

The argument `'{0:{width}}'.format(num, width=5)` specifies how the number should be formatted. The `0` inside the curly braces `{}` is a placeholder for the first argument passed to the format method (num in this case). The `width=5` argument specifies the width of the output, meaning that the output should be right-aligned and padded with spaces up to a width of 5 characters.

The end=' ' argument at the end of the print statement specifies that a space should be printed after each number, instead of the default newline character. This results in the numbers being printed on the same line, separated by spaces.

Template Strings Instead of the normal % based substitutions, templates support $ based substitutions. The rationale behind the introduction of template in Python Version 2.4 was that even though % string formatting is powerful and rich, they are error prone and also quite strict in terms of the formats that follow '%', making it complex. A common mistake with % formatting is to forget the trailing format character, for example: the `e` in `%(variabl)e`.

Templates have some methods defined within: `substitute()` and `safe_substitute()`. Here is how you can use them:

[9]:
```
from string import Template #First you will need to import 'Tempalte' class

money = dict(who = 'You', to_whom = 'baker')
Template('$who owe the $to_whom a total of $$100').substitute(money)
```

Explanation: The `dict` method in Python is a built-in function that creates a new dictionary object. It can take several arguments in different forms to define the elements of the dictionary.

Examples:

d = dict(a=1, b=2, c=3) print(d) {'a': 1, 'b': 2, 'c': 3} d = dict([("a", 1), ("b", 2), ("c", 3)]) print(d) {'a': 1, 'b': 2, 'c': 3} d = dict(zip(["a", "b", "c"], [1, 2, 3])) print(d) {'a': 1, 'b': 2, 'c': 3}

Notice the `$$` in the above example?

This is because with template `$$` is an escape character which is replaced with a single `$`. It is one of the rules for template. Another rule for template is how you define the identifier within your template, there is the usual syntax: `$identifier`, but you also have `${identifier}`.

That means `${identifier}`is the same as `$identifier`, but can be used for scenarios like:

[ ]:
```
word = dict(noun = 'feed')
Template('Please don\'t stop ${noun}ing me').substitute(word)
```

Let's see how you can use `substitute()` and `safe_substitute()`.

[ ]:
```
fact = Template('$alter_ego is weak but wait till he transforms to $superhero!')
fact.substitute(alter_ego='Bruce Banner', superhero='Hulk')
```

[ ]:
```
hero = dict(alter_ego='Peter Parker')
fact.substitute(hero)
```

The above example throws an error, because the superhero is not defined. However, try this...

[ ]:
```
fact.safe_substitute(hero)
```

The `safe_substitute()` is one of the advantage of using template.

Why so many string formatting options, you ask?

Well, it is primarily a matter of syntax preferences. This usually comes down to a simplicity vs verbosity trade-off and also depends on your familiarity with existing syntaxes. For example, using the `%` for string formatting will seem natural to people from C programming background. Templates in Python are simple to write, whereas using `format()` can be more verbose but has more features.

Formatted String Literal (f-string) This is another string formatting method added in Python Version 3.6. A formatted string literal or f-string is a string literal that is prefixed with 'f' or 'F'. You can define identifiers to be used in your string within curly braces { }.

Why yet another string formatting option? Practicality and simplicity is beautiful and that's why!

Check out the examples below to see why f-strings are really the most simple and practical way for formatting strings in Python.

`alter_ego = 'Peter Parker' superhero = 'spiderman' f'{alter_ego} is weak but wait till he transforms to {superhero}!'`

Note: The above code will only work with Python Version 3.6 and above. To check the version of Python you have installed, type: `python -V` on the terminal. Else you can also use the sys module within Python. To use this, simply do the following:

```
[ ]: import sys
     sys.version
```

Formatted strings are really expressions evaluated at run time, which means you can use any Python expressions inside the curly braces of f-strings, which is another great advantage of using f-string.

`f'{alter_ego} is weak but wait till he transforms to {superhero.capitalize()}!'`

Isn't f-string great already! But wait, it gets even better.. f-strings are also faster compared to the other three methods you have seen earlier. This is because they are pre-parsed and stored in efficient bytecode for faster execution.

2.2 Pull the right strings!

You have made it to the end of the module, congratulations! You have seen what strings are, learned about string slicing and seen some operations that can be performed on strings. You have also seen a number of ways to format strings. **But remember, the key to mastering any technique is practice!**

2.3 Exercises

Here are some exercises for Python string manipulation, formatting, and methods:

1. Manipulating strings:
   - Given the string "Hello World!", replace the first word "Hello" with the word "Hi".
   - Given the string "Hi, my name is John", extract the word "John" from the string.
   - Given the string "Hello, world!", capitalize the first letter of each word.
2. Formatting strings:
   - Given the variables name = "John" and age = 30, create a formatted string that says "My name is John and I am 30 years old."

- Given a string “Today is January 1, 2021”, convert the date format to “01-01-2021”.
- Given a string “Python is a powerful language”, split the string into a list of words.

3. String methods:

   - Given the string “Hello, world!”, count the number of times the letter “o” appears in the string.
   - Given the string “Hello, world!”, replace all occurrences of the letter “o” with the number “0”.
   - Given the string “Hello, world!”, remove all whitespaces from the string.

[ ]:

```
```