**Strings**

miércoles, 10 de agosto de 2022

10:16 p. m.

Python Tutorial for Beginners 2 - [Python Tutorial: Slicing Lists and Strings](https://www.youtube.com/watch?v=ajrtAuDg3yw)

Slicing within **list** and **strings** is a way to extract elements from it.

It's possible to access an item within a list same way with the strings, with indexes.

And Indexes could go with an ascending order or with a descending order, in which case the first index would be the last ordered item but in **negative** index (-10 for this case)**.**

my\_list =     [0,  1, 2, 3, 4, 5, 6, 7, 8, 9]

# Asc.  Index  0,  1, 2, 3, 4, 5, 6, 7, 8, 9

# Desc. Index -10,-9,-8,-7,-6,-5,-4,-3,-2,-1

To extract a set of items for that list is need to be used this syntax list [ start : **end** : step ]

*Note: here the "****end****" term of the syntax is non-inclusive.* ***This is easy to forget****.*

e.g. with this code, the result would be

**print**(my\_list[0:6]) = [0, 1, 2, 3, 4, 5]

And it work with negative indexes as well, and even with combined positive and negative indexes, while it has sense.

**print**(my\_list[-8:9]) = [2, 3, 4, 5, 6, 7, 8]

But, if counterintuitively a starting index is place after the ending point, let's say start = 5, with end = -9, the result of this is a list with no values in it.

**print**(my\_list[5:-9]) = [ ]

*Same goes if a much higher/lower index is placed, i.e. s = -90; e = 100, a zero-items list would be returned*

list[star:end:**step**]

The step parameter will determine de order in which the resulting data will be delivered. This means either in direction (from left to right or backwards [right to left]) this is if is 1 or -1 respectively.

And also it could be set how many item will the code skip when returning data. If this index is 2, it would skip 1 item when returning.

**print**(my\_list[0:10:2]) = [0, 2, 4, 6, 8]

If the s-point and the e-point are set to be right to left, the **step** should be in consequence positive. If negative this **print**(my\_list[0:10:-1]) = [ ] would happen.

Finally, the starting and ending point could be omitted and just by specifying the **step** it could deliver the entire list either Left to Right or backwards and skipping however items needed.

**print**(my\_list[ : : -1]) = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

Or

**print**(my\_list[ : : 2]) = [0, 2, 4, 6, 8]

The same would be true while treating **strings**

sample\_url = '<http://coreyms.com>'

# Reverse the url

print (sample\_url[::-1]) = moc.smyeroc//:ptth

# Get the top level domain

print (sample\_url[7:]) = coreyms.com

# Print the url without the http:// or the top level domain

print (sample\_url[7:-4]) = coreyms

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

20 Sep 22: **#**Encryption **#**List **#**Slicing During the exercises, in the exercise #25, I learnt something that for me is worthy.

The case requested to create the Caesars Encryption where one should replace letters according to a direction given (either left or right) and a number which will determine how many letters will be between the actual one and its replacement. E.g.: let's say 'Right' and '3', the key to the encryption will be that 'a' will be 'd' and 'b' will be 'e'; if the case is 'Left' and '1', the result will turn 'a' into 'z' and 'b' into 'a' .

Here, the methods .maketrans() and translate() will be necessary. But the main challenge here is to rotate either to the left or the right a string.

I am using base1 = 'abcdefghijklmnopqrstuvwxyz' and if someone say 'to the right' i need the string to be 'bcdefghijklmnopqrstuvwxyza' and if someone say 'to the left' i need the string to be 'zabcdefghijklmnopqrstuvwxy'.

During some trials I found out that this can be done, and quite simple.

By using this syntax to do it base2 = base1[i:]+base1[0:i] and if i is a positive number it will turn to the right, and if negative it would do it to the left.

I found this quite powerful and the explanation would be the following:

If I give a zero '0' number, the first part of the code (base1[i:]) would start on the substring positioned on 0 which will be 'a' and will go until the end 'z'. And the second part of the code (base1[0:i]) would start **and end** in position 0 which be 'a', but the last position term in slices and lists are exclusive, meaning that the second part of the code would return a void, resulting in 'abcdefghijklmnopqrstuvwxyz'

If I give a positive number, let's say '1', the first part of the code (base1[i:]) would start on the substring positioned on 1 which will be 'b' and will go until the end 'z'. And the second part of the code (base1[0:i]) would start in position 0 which be 'a', and exclude de last term on the number given '1', just resulting in 'bcdefghijklmnopqrstuvwxyza'

Now, If I give a negative number, let's say '-1', the first part of the code (base1[i:]) would start on the substring positioned on -1 which will be 'z' and end right there, because that's already the end of the string. And the second part of the code (base1[0:i]) would start in position 0 which be 'a', and exclude the last term on the position of the number given '-1', just resulting in 'zabcdefghijklmnopqrstuvwxy'.

**Summary**

* **Lists** as a concept and how to slice them.
* How to run through list with the syntax list[star:end:step]
* How this is also applicable for **strings**
* Caesar's Encryption - List and Strings moving to the left or to the right

Python Tutorial for Beginners 2 - [Python Tutorial: String Formatting - Advanced Operations for Dicts, Lists, Numbers, and Dates](https://www.youtube.com/watch?v=vTX3IwquFkc)

Formatting **strings** has many ways to be done but there's ones more problematic than others.

e.g.

person = {'name': 'Jenn', 'age': 23} / This is called a Dictionary, btw.

sentence = 'My name is ' + person['name'] + ' and I am ' + str(person['age']) + ' years old.'

print(sentence)

And that way isn't wrong typed, but is just that this way, while it works, is more primitive therefore in the long run is non-sustainable. It has a lot of "+", and additionally it request to create multiple strings over the "sentence" variable, remember **DRY (Don't Repeat Yourself)**, if there's a lot of repetition is a sign of something going the wrong direction. And finally, the integer is needed to be cast into a string in this part str(person['age'])

So, definitely works but it totally could have some polishing.

Another way to do it is as follows

sentence = 'My name is {} and I am {} years old.'.format(person['name'], person['age'])

This way is more readable, and will automatically pass over the variables to the placeholders in respective order, first will be the 'name' item and later de 'age'.

Either way is possible to set which placeholder will go according to what item in the .format() method, it simply requires to number from 0 to 1 the placeholders and the method will assume that 0 will be the first item and respectively the rest.

The resulting code would look like this:

sentence = 'My name is {0} and I am {1} years old.'.format(person['name'], person['age'])

This will come handy when is needed to use the same value multiple times in the same line or code, like here:

tag = 'h1'

text = 'This is a headline'

sentence = '<{0}>{1}</{0}>'.format(tag, text)

The resulting printout would be:

<h1>This is a headline</h1>

Now, until this point, we have been passing within the .format() method directly the items to be accessed, but to be more precise, in the sentence ".format( person['name'], person['age']) ", the dictionary person appears twice, in other words, we are saying to the code that to access once person and retrieves the ['name'], and later, we request to go again to the same place person to now get the ['age'], repetition appears, so we probably would like to go over and try to do better

**Note**: this would make sense if we were talking about two dictionaries, and if that were the case, the use of multiple numbers 0, 1, 2, … n within the placeholders make sense, since we would be indicating to go to different places. But for this case, we only use one number by convention 0 since we will be using only one dictionary for this case

To better do this line, we would rewrite the code as follows

sentence = 'My name is {0[name]} and I am {0[age]} years old.'.format(person)

print(sentence) = My name is Jenn and I am 23 years old.

If is decided to use fStrings instead of the .format() method, the code would look like this

sentence = f'My name is {person["name"]} and I am {person["age"]} years old.'

Similar to this, if the information was not coming from a dictionary but is an attribute from a object, it wouldn't be much difference.

First the Class are created and then the object with the attributes

class Person():

    def \_\_init\_\_(self, name, age):

        self.name = name

        self.age = age

p1 = Person('Jack', '33')

Now, the sentence would look like this

sentence = 'My name is {0.name} and I am {0.age} years old.'.format(p1)

Or fStringed

sentence = f'My name is {p1.name} and I am {p1.age} years old.'

The outcome would be the same

print(sentence) = My name is Jack and I am 33 years old.

And there is still another way to do it by building the dictionary within the method

sentence = 'My name is {name} and I am {age} years old.'.format(name='Jenn', age=23)

*I don't particularly like this way, tho*

And yet, there's another way to do it, and According to Corey Schafer, this is a more readable way to do it. And this include the notion of Unpacking dictionaries that is unknown for me at this point, but it goes like this:

sentence = 'My name is {name} and I am {age} years old.'.format(\*\*person)

Apparently, by just adding " \*\* " the code will understand that the dictionary person contains the items name and age

Printing out numbers

For the first example, there's a loop to print out a set of numbers

for i in range(1, 11):

    sentence = 'The value is {}'.format(i)

   print(sentence)

Result will be

The value is 1

The value is 2

…

The value is 10

To make them two digits print out is enough to add format within the placeholder {}. And this is made by adding a **colon,** and for double digits "02" and the colon {:02}. The 10 stays as it is since is already double digited

for i in range(1, 11):

    sentence = 'The value is {:02}'.format(i)

   print(sentence)

Result will be

The value is 01

The value is 02

…

The value is 10

Now, to add floating point or "decimals" the same is needed to be done but instead of :02, ":.2f" will be the content of the placeholder {}. Then the pi number with 8 decimals 3.14159265 becomes 3.14

pi = 3.14159265

sentence = 'Pi is equal to {:.2f}'.format(pi)

print(sentence) = Pi is equal to 3.14

To print out large numbers with separators, it goes the same as it does with the decimals but after the colon we just add the comma {:,}

sentence = '1 MB is equal to {:,} bytes'.format(1000\*\*2) // \*\* Operator is exponential

print(sentence) = 1 MB is equal to 1,000,000 bytes

On this same example, to add the 2 decimals is enough with just adding the ".2f" after the comma {:,.2f}

sentence = '1 MB is equal to {:,.2f} bytes'.format(1000\*\*2)

print(sentence) = 1 MB is equal to 1,000,000.00 bytes

Printing out dates

Source: <https://docs.python.org/3/library/datetime.html?highlight=datetime#datetime.datetime>

For date printing we now will have to import a module called "datetime". With that, a new variable is created which will be an object from the class datetime. next a quick description of this object and what input it requires

*datetime( year, month, day [, hour[, minute[, second[, microsecond[,tzinfo]]]]] )*

*The year, month and day arguments are required. tzinfo may be None, or an instance of a tzinfo subclass. The remaining arguments may be ints.*

And will be created as follows,

import datetime

my\_date = datetime.datetime(2016, 9, 24, 12, 30, 45)

print(my\_date) = 2016-09-24 12:30:45

Now if we would like to print that date in a different format, for instance "March 01, 2016", we would first have to create a String which will be formatted, but this time we would have to consult the documentation of the language [here](https://docs.python.org/3/library/datetime.html).

So, we first need to take the month, then the day with two digits, then a comma and last the year with four digits

month = %B *(According to the documentation)*

day = %d

*year =* %Y

sentence = '{:%B %d, %Y}'.format(my\_date)

print(sentence) = September 24, 2016

Or fStringed

print(f'{date:%B %d, %Y}') = September 24, 2016

Now we let's try to do the same with the format, for instance "March 01, 2016 fell on a Tuesday and was the 061 day of the year.", we now will have to pull the month, day in three digits and year in four digits again and additionally, we would have to pull the week day name and the day number of the year.

week day name = %A

Day number of the year = %j

To print out correctly the output as we want to, one last thing is necessary here. As we are entering different placeholders to the string created it will be necessary to specify with index 0 to which argument of the .**format**() method we are referring to. So the final string would look like this

sentence = '{0:%B %d, %Y} fell on a {0:%A} and was the {0:%j} day of the year'.format(my\_date)

# This would be the fStringed version of it

new\_sentence = f'{my\_date:%B %d, %Y} fell on a {my\_date:%A} and was the {my\_date:%j} day of the year'

print(sentence) = September 24, 2016 fell on a Saturday and was the 268 day of the year

**Summary**

* 'Dictionary' Concept
* .**format**() method more extensively, like passing variables and indexing them into strings, calling items from a dictionary argument and unpacking dictionaries
* fStrings within each exercise.
* Printing out numbers with decimals and with separators.
* Printing out dates.

Python Tutorial for Beginners 2 - [Python Quick Tip: F-Strings - How to Use Them and Advanced String Formatting](https://www.youtube.com/watch?v=nghuHvKLhJA&list=PL-osiE80TeTt2d9bfVyTiXJA-UTHn6WwU&index=36)

Everything explained on this video goes over what we already passed over the last one, but here are some tips and insights that could come useful.

With fStrings is possible to do calculations within the placeholder. With the .format() method is possible to do calculation within the parameter of the method as well.

sentence = f'4 times 11 is {4\*11}'

print(sentence) = 4 times 11 is 44

----------------------------------------------------------------------------------------------------------------------------------------------------------

Strings in Python - Advanced Python 05 - Programming Tutorial - [Strings in Python - Advanced Python 05 - Programming Tutorial](https://www.youtube.com/watch?v=e6ivlABOYRI)

.strip( ) method:

The .strip( ) method works removing any leading (spaces at the beginning) and trailing (spaces at the end) **characters [meaning not limited only to 'spaces' or blanks]** (space is the default leading character to remove).

If the method does not receive any argument, it will remove spaces at the beginning and at the end of the string. But any characters could be passed and the method will remove it from the string.

text = "   Maegor Targaryen was the cruel king   sdafsdf"

text = text.strip()

print(text) = Maegor Targaryen was the cruel king   sdafsdf

Now, is important to understand that .strip( ) will remove the characters specified only at the beginning and at the end of the string. It will do nothing if the strings are not at the beginning or at the end. And since python is a case sensitive language, it is needed to be careful with low and upper casing.

text = text.strip("a")

print(text) = Maegor Targaryen was the cruel king   sdafsdf

And it will delete the "sdafsd" subchain but careful on this, because it will only get rid of the whole subchain only if **all** the characters are included, otherwise it will only evaluate one by one, and the first one is the one it will try to find either at the beginning or at the end of the string and if not found the method stops (as seen above with the 'a' argument on the method).

text = text.strip("safd ")

print(text) = Maegor Targaryen was the cruel king

The variations of this methods are the .lstrip( ) & the .rstrip( ). Which does the same but starting from left to right and it will stop until no longer find the character stripped and from right to left, respectively.

.startswith( ) and .endswith( ) method:

this two methods are fairly simple, they only make a validation for either a string ends or starts with some characters and will return True or False

They could take up to 3 arguments: the first will be the assess characters, the second will be the starting position of the evaluation, and the third will be he end position of the evaluation.

.split( ) method:

The .split( ) method works transforming a chain or a string into a list, separating the chain with a given argument, if no argument is passed, it will cut the string with spaces.

Continuing with the same example:

print(text.split()) = ['Maegor', 'Targaryen', 'was', 'the', 'cruel', 'king']

And it could take one more argument besides the "separator". It would be the max items to be split. And this means that the method will split into separate items for the resulting list but if limited, the rest of the string will be delivered as the last item of the list.

print(text.split(' ',3)) = ['Maegor', 'Targaryen', 'was the cruel king']

.join( ) method:

This method joins all the items within an iterable and joins them into one string (kind of the opposite of the split method).

But the syntax is different, it would not operate as the other methods. It will require the separator character specified as a string value and the used the " .join()" and as argument the iterable that will be joined must be passed.

Let's say we will use .split to create a new string same as the example before

text = text.split() = ['Maegor', 'Targaryen', 'was', 'the', 'cruel', 'king']

Now, in order to turn back the resulting list with .join() we will have to do this:

text = ' '.join(text) = Maegor Targaryen was the cruel king

**Note:** the join( ) method is a good answer to joining characters and business-wise is a much cheaper solution. Next an example for this

For this case we will be running the same process of joining characters from a list but we will compare the time (which at the end is computational capacity, and $$$) doing it through for loop statement compared with .join().

*We will be needing of importing* timeit *library to use timers*

from timeit import default\_timer as timer

my\_list = ['a'] \* 10000000

# For in statment

start = timer()

my\_string = ''

for i in my\_list:

    my\_string += i

stop = timer()

print(stop - start) = 4.772639900009381

# join() method

start = timer()

my\_list = ''.join(my\_list)

stop = timer()

print(stop - start) = 0.06891870000981726

The conclusion is that with the "For - in" statement is needed 4.77s and with the join method 0.069s comparatively is almost 68 times the resource needed

----------------------------------------------------------------------------------------------------------------------------------------------------------

Programming Tutorial - [Python - (6) - String built-in methods - 1](https://www.youtube.com/watch?v=PNoP79KxI98)

.capitalize( ) method:

The .capitalize( ) method turns the first letter of a string into a capital letter.

.title( ) method:

The .title( ) method turns the first letter of each word of a string into a capital letter.

Programming Tutorial - [Python - (7) - String built-in methods - 2](https://www.youtube.com/watch?v=f28jEsTEdII&list=PLAYCSrgc6HVyMgmSQDjx-EcmjrgCHoXxQ&index=7)

.isalpha( ) method:

The .isalpha( ) method is just a validation whether the input is exclusively alphabetic info. It will return True or False. This will exclude special characters (\|@#!"·=) and spaces too.

.islower( ) & .isupper( ) methods:

both methods will validate whether the info is exclusively in lower or upper case. This validation does not consider whitespaces or any special characters as character so will return True if the case match with the validation besides the whitespaces, or any other special character.

.endswith( ) & .startswith( ) methods:

both methods will validate whether the info is starts or ends with a given parameter.

.center( ) method:

This method adds padding (filling characters) up to the input given. So it will sum up the number of characters in the original string and creates a new output where with the given length of padding it sums up and divide in half to center the resulting string.

* There's an alternative with the formatting method (either fStrings or .format()) and is with the syntax { :y^x} being ^ the operator that indicates that the output must be centered and the x is the number of characters in which the output will be centered, and y is the actual characters that will pad the output.

my\_str = 'Hello'

print(f'{my\_str:#^11}') = ###Hello###

----------------------------------------------------------------------------------------------------------------------------------------------------------

Programming Tutorial - [31 ESSENTIAL Python String Methods](https://www.youtube.com/watch?v=HJpiAZDJrRY)

.removeprefix( ) & .removesuffix( ) methods:

If I wanted to remove characters from a string the initial impulse is to go with the strip method, but the downside is that the substring I want to preserve contains any of the character on the suffix or prefix, it will consequently alter the substring I need.

Here, the two methods works similarly, they will remove any leading or trailing specified substrings. Also similarly to the string slicing, but with the advantage of no regarding on the length of the string to cut properly what's needed to be removed.

.re.sub 'regular expression' method:

This is used to work with more complex strings. Say you want to replace the space in "string methods in python" and you use the replace method like replace( ' ' , '-' ) and you would get "string---methods-in-python" but you only need one single - after the word 'string', then this regex (regular expression) method would come handy

To do this it would be necessary to import the re library and rewrite the expression with the pattern ' \s+ ' which will replace multiple whitespaces and replace them with a given argument.

The code the goes like this:

s = 'string    methods in python'

s = re.sub('\s+','-',s)

print(s) = string-methods-in-python

.partition( ) method:

Works similar to the split method, but this one with a given argument would return a tuple with three elements: ('the leading part of the string just before the argument', 'the argument', 'the trailing part of the string after de argument').

If the argument does not appear in the string, it would return ('the whole string', ' ', ' ').

.swapcase( ) method:

Literally does what's the name says. This is true for the whole string.

.zfill( ) method:

It returns a new string with the length of the argument and it fills the rest of the string with ' 0 '.

s = 'Hello'.zfill(10)

print(s) = 00000Hello

s = '+Hello'.zfill(10)

print(s) = +0000Hello

.casefold( ) method:

This method does not take any argument and is similar to .**lower**(), but the main difference is that lower ignore cases that are in other languages, but casefold is more aggressive and will turn something like the German letter 'ß' into its equivalent 'ss'.

.format\_map( ) method:

This method works similar to the .format(\*\*dict) but the difference is that the format\_map method is slightly faster and more flexible than its counterpart. And this is due to the format\_map creates a dictionary during the method call while the .format(\*\*dict) needs to count with a dictionary and unpacking it to work correctly. This comes handy when missing keys of the dictionary are requested to be print or passed, and with managing dictionaries from a class.

We have three examples on how it works:

Example #1 - Simple dictionary printing with only one passing placeholder:

s = 'Hey, there! my name is {name}'

d = {'name' : 'Gerardo'}

print(s.format\_map(d)) = Hey, there! my name is Gerardo

Example #2 - Simple dictionary printing with two passing placeholders:

s = 'Coordinates are: x: {x}; y: {y} '

d = {'x' : 14, 'y' : 3 }

print(s.format\_map(d)) = Coordinates are: x: 14; y: 3

And this would work exactly the same as to the .format(\*\*dict):

s = 'Coordinates are: x: {x}; y: {y} '

d = {'x' : 14, 'y' : 3 }

print(s.format(\*\*d)) = Coordinates are: x: 14; y: 3

Example #3 - with Class inherit dictionary:

class Foo (dict):

    def \_\_missing\_\_(self, key):

        return 'Not Found!'

print('( x: {x} ; y: {y} )'.format\_map(Foo({'x': 3}))) =

( x: 3 ; y: Not Found! )

What's valuable here is that the format\_map can handle not have the 'y' key defined and will just inherit the 'Not Found!' from the Class definition.

On the other hand, if working with the .format(\*\*dict) unpacking the 'y' key must be defined, otherwise it would return an error.

class Foo (dict):

    def \_\_missing\_\_(self, key):

        return 'Not Found!'

print('( x: {x} ; y: {y} )'.format(\*\*Foo({'x': 3}))) = **Se produjo una excepción: KeyError** 'y'

But this could be fixed just defining the 'y' key:

class Foo (dict):

    def \_\_missing\_\_(self, key):

        return 'Not Found!'

print('( x: {x} ; y: {y} )'.format(\*\*Foo({'x': 3, 'y': 7}))) = ( x: 3 ; y: 7 )

.index( ) method:

This string method works on list containing strings. It simply returns the index within the list where the argument first is located. It could also receive another two arguments, a starting point and an ending point to execute the search. If not found it will raise an error.

.ljust( ) & rjust( ) methods:

This two methods works very similar to the center string method, in which the method will take two arguments, one: the length of the resulting string, and second: the filling character to pad the resulting string, and depending on which one the result will be a string padded either to the left or to the right respectively.

.maketrans( ) & .translate( ) methods:

This two methods were created to collaborate. The translate method would require a table with the corresponding dictionary to translate, and maketrans method would do it so. To further get into this, the following link will help as resource: <https://medium.com/analytics-vidhya/when-and-where-to-use-maketrans-and-translate-in-python-764670ff91fe>

.rfind( ) methods:

This method will return the higher (meaning the right-most or starting from the right to the left) index in which the substring is being searched. It could take three arguments: the substring searched, the starting and ending indexes. It will return -1 if not found.

.rindex( ) methods:

Works similar to rfind but this one does not returns -1 if not found, it would raise an exception. This method will to return the higher value in which the substring are within the string where the search happens.

.splitlines( ) methods:

This method works similar to the split method, but it already take the escaped characters, mainly the 'new line' '\n' but also 'Returns, tabs or whatever'. But essentially would be the same to apply a .split('\n') .

String Validators

|  |  |
| --- | --- |
| isalnum() | Returns True if all characters in the string are alphanumeric |
| isalpha() | Returns True if all characters in the string are in the alphabet |
| isascii() | Returns True if all characters in the string are ascii characters |
| isdecimal() | Returns True if all characters in the string are decimals |
| isdigit() | Returns True if all characters in the string are digits |
| isidentifier() | Returns True if the string is an identifier |
| islower() | Returns True if all characters in the string are lower case |
| isnumeric() | Returns True if all characters in the string are numeric |
| isprintable() | Returns True if all characters in the string are printable |
| isspace() | Returns True if all characters in the string are whitespaces |
| istitle() | Returns True if the string follows the rules of a title |
| isupper() | Returns True if all characters in the string are upper case |

Python Tutorial for Beginners 2

* By convention, variables are all-lowercased, and if contains multiple word, then should be separated by a underscore.

message = 'Hello World'

Or

my\_message = 'Hello World'

Try to name variables as explicit as possible.

Is equally valid to use for strings either single quotes ( ' ) or double quotes ( " ) or even triple quotes ( ''' ), and all will work. But the difference is more on what's the string content. If the content includes a single quote as a part of it, the Editor will understand that there is the end of the string. For this are multiple solution:

1. To "Escape" (\) [on my keyboard is "cmd" + "º key"] the single qoute.

1. To encapsulate the string between double quotes (")

1. In the case the string contains both single and doubles qoute the solution y to triple quote the whole string (''')

Triple quotes are used for multiline strings:

message = ''' Hi

there's something I don't like

and it's weird' '''

print (message) =

Hi

there's something I don't like

and it's "weird"

The **len( )** function is a built-in function that returns the number of items in an object. When the object is a string, **len( )** returns the number of characters in the string.

Within the String it's possible to access to each character individually but understanding the string as a chain of characters ordered in indexes from 0 to n. That index is called by the position number within brackets **[ ]**

my\_message = 'Hello World'

my\_message[0] = H

Is also possible to call a sequence of characters of the same String by using the same method but the range would be separated by a colon ( : ).

*Note: the first term of the range is inclusive but the latest is not.* ***This is easy to forget****.*

my\_message[0:9] = Hello Wor

*The ninth term ( l ) is not included, only up to the eighth ( r )*

And is also possible to place a finishing point or a starting point without having a certain beginning or ending

my\_message[:10] = Hello Worl

my\_message[7:] = orld

*This is called 'Slicing'*

# to comment out something by shortcut (in my keyboard: ctrl + \key )

**Methods** and **Functions** essentially are equal, but, the different is that a **Method** belongs to a specific class (like the ones you create).

Python Method

"Methods are packaged set of instructions which could be directly used on objects."

1. Method is called by its name, but it is associated to an object (dependent).
2. A method definition always includes ‘self’ as its first parameter.
3. A method is implicitly passed the object on which it is invoked.
4. It may or may not return any data.
5. A method can operate on the data (instance variables) that is contained by the corresponding class

Functions

1. Function is block of code that is also called by its name. (independent)
2. The function can have different parameters or may not have any at all. If any data (parameters) are passed, they are passed explicitly.
3. It may or may not return any data.
4. Function does not deal with Class and its instance concept.

There are methods for the class **String**, e.g. to lowercase a String we could use the method " .**lower**() ":

**print**(my\_message.**lower**()) = hello world

Same goes for " .**upper**() "

**print**(my\_message.**upper**()) = HELLO WORLD

Now, there are methods that require **arguments**, like the string method " .**count**() ", it'll count how many time the argument appears on a certain string.

**print**(my\_message.**count**( 'Hello' )) = 1

*Because 'Hello' appears once in the string*

**print**(my\_message.**count**( ' l ' )) = 3

If the method take a '' as argument, it will count the total amount of characters in the string.

a = 'test string'

print(a.count("")) = 12

Another string method that requires **arguments is** " .**find**() ", it'll return the initial position of the argument.

**print**(my\_message.**find**( 'World' )) = 6

*Because 'Word' starts with the W in the 6 position*

**print**(my\_message.**find**( ' l ' )) = 2

*Because it'll return the first position it finds the argument, which is 'l' and the first l is in the 2 position*

**print**(my\_message.**find**( ' Boom ' )) = -1

*Because when the method doesn't find the argument it just return -1*

One could replace information within the string with the " .**replace**() " method, it'll require two arguments separated by a comma, one the value we would like to replace and the second, its replacement

my\_message.**replace**( 'World' , 'Universe' )

**print**(my\_message) = Hello World

*And it's not replaced because the method returns a new string but doesn't affect the original*

**print**(my\_message.**replace**( 'World' , 'Universe' )) = Hello Universe

*Instead:*

new\_message = my\_message.**replace**( 'World' , 'Universe' )

**print**(new\_message) = Hello Universe

It's also possible to concatenate information to build a string. Simply is needed the " **+** " operator and it'll forcibly join strings into a string, but the more common way to do this is via placeholding and the " .**format**() " method.

The method would require to put placeholders identified with **{ }** brackets, and pass in order the arguments that will fill in those places. It could be either variables or straight-up data (strings, numeric.

*Let's rewrite the variable my\_message…*

greeting = 'Hello'

name = 'Michael'

my\_message = ' **{ }**, **{ }**. Welcome!'.**format**( **greeting**, **name** )

**print**(my\_message) = Hello, Michael. Welcome!

*f-Strings*

From Python 3.6 upwards, fStings are functioning to simplify even more the work with strings. It works basically the same as the .format() method but instead passing the arguments at the end, it's just naming the variables directly into the placeholders, the only thing needed is to put an **f** outside and before the actual string

my\_message = **f**' **{greeting}**, **{name}**. Welcome!'

**print**(my\_message) = Hello, Michael. Welcome!

And also understand code within the placeholder

my\_message = **f**' **{greeting}**, **{name**.**upper**( )**}**. Welcome!'

**print**(my\_message) = Hello, MICHAEL. Welcome!

Now, besides strings, to understand what methods are available from a class, the **dir**( ) Function when printed it return all the available methods from the object within the argument.

**print**(**dir**(**name**)) = '\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format\_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex',

'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill'

*Which would the built-in methods for String class*

…………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………

**Summary**

* **len( )** : Length of an object
* String indexing: my\_message[0] = H / my\_message[:10] = Hello Worl / my\_message[7:] = orld
* Differences between Methods and Functions
* .**lower**( ) : to lowercase a string
* .**upper**( ) : to uppercase a string
* .**count**( ) : to count how many time an argument appears on a string
* .**find**( ) : to find the first time an argument appears on a string
* .**replace**( ) : to replace an argument within the string for another
* .**format**( ) / fStringing
* **dir**( ) : Function, to know the available methods from an object.

During the exercise 25 - Caesar's Encryption, the modulo (%) became important

The modulo operator became important because if a shift given is a number higher than 26 (for the English alphabet) the algorithm itself would stop working but by applying Modulo to reassign the reminder it would mean that, for example if 80 is the shift, by %= 26, it would divide in 26 and the integer part of the result would be the laps to the whole alphabet, but the remainder of the division drawn by the modulo operation would say the usable shift for the actual encryption.

E.g.: shift = 80 / shift %= 26 / shift = 2.

2 the would be the shifts to made.

Something additional was that if one would like to decrypt a Caesar's encryption, first the shift is needed to be known and it'd just be matter of placing a new shift that'd be the remainder of the actual encrypting shift minus the total alphabet (26), that new shift would decrypt the code

Say,

word to be encrypted = hello

shift = 2

Encrypted word = 'fcjjm'

Decryption shift = 26 - 2 = 24