PYTHON

STRING MANIPULATION HANDBOOK

RENAN MOURA

String Manipulation in Python

Renan Moura

Table of Contents

Preface

String manipulation is one of those activities in programming that we, as programmers, do all the time.

In many programming languages, you have to do a lot of the heavy lifting by yourself.

In Python, on the other hand, you have several built-in functions in the standard library to help you manipulate strings in the most different ways you can think of.

In this book I will showcase these many features of the language regarding strings specifically along with some nice tricks.

If you come from another programming language, you will notice many things that you can do with the standard library that are only possible to do with the use of Regular Expressions in other languages.

Regulars Expressions are super powerful and useful, but can get really hard to read, so having other alternatives is handy and helps with keeping a more maintainable codebase.

I'm Renan Moura and I write about Software Development on renanmf.com.

You can also find me as @renanmouraf on:

- Twitter: https://twitter.com/renanmouraf
- LinkedIn: https://www.linkedin.com/in/renanmouraf

- Instagram: https://www.instagram.com/renanmouraf

Basics

The text type is one of the most common types out there and is often called *string* or, in Python, just `str`.

```
my_city = "New York"
print(type(my_city))

#Single quotes have exactly
#the same use as double quotes
my_city = 'New York'
print(type(my_city))

#Setting the variable type explicitly
my_city = str("New York")
print(type(my_city))
```

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

Concatenate

You can use the `+` operator to concatenate strings.

Concatenation is when you have two or more strings and you want to join them into one.

```
word1 = 'New '
word2 = 'York'

print(word1 + word2)
```

```
New York
```

Selecting a char

To select a char, use `[]` and specify the position of the char.

Position 0 refers to the first position.

```
>>> word = "Rio de Janeiro"
>>> char=word[0]
>>> print(char)
R
```

Size of a String

The `len()` function returns the length of a string.

```
>>> len('Rio')
3
>>> len('Rio de Janeiro')
14
```

Replacing

The `replace()` method replaces a part of the string with another. As an example, let's replace 'Rio' for 'Mar'.

```
>>> 'Rio de Janeiro'.replace('Rio', 'Mar')
'Mar de Janeiro'
```

Rio means River in Portuguese and Mar means Sea, just so you know I didn't choose this replacement so randomly.

Count

Specify what to count as an argument.

In this case, we are counting how many spaces exist in "Rio de Janeiro", which is 2.

```
>>> word = "Rio de Janeiro"
>>> print(word.count(' '))
```

```
2
```

Repeating a String

You can use the `*` symbol to repeat a string.

Here we are multiplying the word “Tokyo” by 3.

```
>>> words = "Tokyo" * 3
>>> print(words)
TokyoTokyoTokyo
```

How to Split a String

Split a string into smaller parts is a very common task, to do so, we use the `split()` method in Python.

Let's see some examples on how to do that.

Example 1: whitespaces as delimiters

In this example, we split the phrase by whitespaces creating a list named **my_words** with five items corresponding to each word in the phrase.

```
my_phrase = "let's go to the beach"
my_words = my_phrase.split(" ")

for word in my_words:
    print(word)
#output:
#let's
#go
#to
#the
#beach

print(my_words)
#output:
#["let's", 'go', 'to', 'the', 'beach']
```

Notice that, by default, the `split()` method uses any consecutive number of whitespaces as delimiters, we can change the code above to:

```
my_phrase = "let's go to the beach"
my_words = my_phrase.split()

for word in my_words:
    print(word)

#output:
#let's
#go
```

```
#to
#the
#beach
```

The output is the same since we only have 1 whitespace between each word.

Example 2: passing different arguments as delimiters

When working with data, it's very common to read some CSV files to extract information from them.

As such, you might need to store some specific data from a certain column.

CSV files usually have fields separated by a semicolon ";" or a comma ",".

In this example, we are going to use the `split()` method passing as argument a specific delimiter, ";" in this case.

```
my_csv = "mary;32;australia;mary@email.com"
my_data = my_csv.split(";")

for data in my_data:
    print(data)

#output:
#mary
#32
#australia
#mary@email.com

print(my_data[3])
#output:
# mary@email.com
```

How to remove all white spaces in a string

If you want to truly remove any space in a string, leaving only the characters, the best solution is to use a regular expression.

You need to import the `re` module that provides regular expression operations.

Notice the `\s` represents not only space `' '`, but also form feed `\f`, line feed `\n`, carriage return `\r`, tab `\t`, and vertical tab `\v`.

In summary, `\s = [ \f\n\r\t\v]`.

The `+` symbol is called a quantifier and is read as 'one or more', meaning that it will consider, in this case, one or more white spaces since it is positioned right after the `\s`.

```
import re

phrase = ' Do   or do    not   there    is  no try   '

phrase_no_space = re.sub(r'\s+', '', phrase)

print(phrase)
# Do   or do    not   there    is  no try

print(phrase_no_space)
#Doordonotthereisnotry
```

The original variable `phrase` remains the same, you have to assign the new cleaned string to a new variable, `phrase_no_space` in this case.

Multiline Strings

Triple Quotes

To handle multiline strings in Python you use triple quotes, either single or double.

This first example uses double quotes.

```
long_text = """This is a multiline,

a long string with lots of text,

I'm wrapping it in triple quotes to make it work."""

print(long_text)
#output:
#This is a multiline,
#
#a long string with lots of text,
#
#I'm wrapping it in triple quotes to make it work.
```

Now the same as before, but with single quotes.

```
long_text = '''This is a multiline,

a long string with lots of text,

I'm wrapping it in triple quotes to make it work.'''

print(long_text)
#output:
#This is a multiline,
#
#a long string with lots of text,
#
#I'm wrapping it in triple quotes to make it work.
```

Notice both outputs are the same.

Parentheses

Let's see an example with parentheses.

```
long_text = ("This is a multiline, "
"a long string with lots of text "
"I'm wrapping it in triple quotes to make it work.")
print(long_text)
#This is a multiline, a long string with lots of text I'm wrapping it
#in triple quotes to make it work.
```

As you can see, the result is not the same, to achieve new lines I have to add `\n`, like this:

```
long_text = ("This is a multiline, \n\n"
"a long string with lots of text \n\n"
"I'm wrapping it in triple quotes to make it work.")
print(long_text)
#This is a multiline,
#
#a long string with lots of text
#
#I'm wrapping it in triple quotes quotes to make it work.
```

Backlashes

Finally, backlashes are also a possibility.

Notice there is no space after the `\` character, it would throw an error otherwise.

```
long_text = "This is a multiline, \n\n" \
"a long string with lots of text \n\n" \
"I'm using backlashes to make it work."
print(long_text)
#This is a multiline,
#
#a long string with lots of text
#
#I'm wrapping it in triple quotes to make it work.
```

lstrip(): removing spaces and chars from the beginning of a string

Use the `lstrip()` method to remove spaces from the beginning of a string.

```
regular_text = "   This is a regular text."

no_space_begin_text = regular_text.lstrip()

print(regular_text)
#'   This is a regular text.'

print(no_space_begin_text)
#'This is a regular text.'
```

Notice that the original `regular_text` variable remains unchanged, thus you need to assign the return of the method to a new variable, `no_space_begin_text` in this case.

Removing Chars

The `lstrip()` method also accepts specific chars for removal as parameters.

```
regular_text = "$@G#This is a regular text."

clean_begin_text = regular_text.lstrip("#$@G")

print(regular_text)
#$@G#This is a regular text.

print(clean_begin_text)
#This is a regular text.
```

rstrip(): removing spaces and chars from the end of a string

Use the `rstrip()` method to remove spaces from the end of a string.

```
regular_text = "This is a regular text.   "

no_space_end_text = regular_text.rstrip()

print(regular_text)
#'This is a regular text.   '

print(no_space_end_text)
#'This is a regular text.'
```

Notice that the original `regular_text` variable remains unchanged, thus you need to assign the return of the method to a new variable, `no_space_end_text` in this case.

The `rstrip()` method also accepts specific chars for removal as parameters.

```
regular_text = "This is a regular text.$@G#"

clean_end_text = regular_text.rstrip("#$@G")

print(regular_text)
#This is a regular text.$@G#

print(clean_end_text)
#This is a regular text.
```

strip(): removing spaces and chars from the beginning and end of a string

Use the `strip()` method to remove spaces from the beginning and the end of a string.

```
regular_text = "  This is a regular text.   "

no_space_text = regular_text.strip()

print(regular_text)
#'  This is a regular text.   '

print(no_space_text)
#'This is a regular text.'
```

Notice that the original `regular_text` variable remains unchanged, thus you need to assign the return of the method to a new variable, `no_space_text` in this case.

The `strip()` method also accepts specific chars for removal as parameters.

```
regular_text = "AbC#This is a regular text.$@G#"

clean_text = regular_text.strip("AbC#$@G")

print(regular_text)
#AbC#This is a regular text.$@G#

print(clean_text)
#This is a regular text.
```

String Lowercase

Use the `lower()` method to transform a whole string into lowercase.

```
regular_text = "This is a Regular TEXT."

lower_case_text = regular_text.lower()

print(regular_text)
#This is a Regular TEXT.

print(lower_case_text)
#this is a regular text.
```

Notice that the original `regular_text` variable remains unchanged, thus you need to assign the return of the method to a new variable, `lower_case_text` in this case.

String Uppercase

Use the `upper()` method to transform a whole string into uppercase.

```
regular_text = "This is a regular text."

upper_case_text = regular_text.upper()

print(regular_text)
#This is a regular text.

print(upper_case_text)
#THIS IS A REGULAR TEXT.
```

Notice that the original `regular_text` variable remains unchanged, thus you need to assign the return of the method to a new variable, `upper_case_text` in this case.

Title Case

Use the `title()` method to transform the first letter in each word into upper case and the rest of characters into lower case.

```
regular_text = "This is a regular text."

title_case_text = regular_text.title()

print(regular_text)
#This is a regular text.

print(title_case_text)
#This Is A Regular Text.
```

Notice that the original `regular_text` variable remains unchanged, thus you need to assign the return of the method to a new variable, `title_case_text` in this case.

Swap Case

Use the `swapcase()` method to transform the upper case characters into a lower case and vice versa.

```
regular_text = "This IS a reguLar text."

swapped_case_text = regular_text.swapcase()

print(regular_text)
#This IS a reguLar text.

print(swapped_case_text)
#tHIS is A REGUlAR TEXT.
```

Notice that the original `regular_text` variable remains unchanged, thus you need to assign the return of the method to a new variable, `swapped_case_text` in this case.

Checking if a string is empty

The pythonic way to check if a `string` is empty is using the `not` operator.

```
my_string = ''
if not my_string:
  print("My string is empty!!!")
```

To check the opposite, if the string is **not** empty:

```
my_string = 'amazon, microsoft'
if my_string:
  print("My string is NOT empty!!!")
```

rjust(): right-justified string

Use the `rjust()` to right-justify a string.

```
word = 'beach'
number_spaces = 32

word_justified = word.rjust(number_spaces)

print(word)
#'beach'

print(word_justified)
#'                           beach'
```

Notice the spaces in the second string. The word ‘beach’ has 5 characters, which gives us 27 spaces to fill with empty space.

The original `word` variable remains unchanged, thus we need to assign the return of the method to a new variable, `word_justified` in this case.

The `rjust()` also accepts a specific char as a parameter to fill the remaining space.

```
word = 'beach'
number_chars = 32
char = '$'

word_justified = word.rjust(number_chars, char)

print(word)
#beach

print(word_justified)
#$$$$$$$$$$$$$$$$$$$$$$$$$$$beach
```

Similar to the first situation, I have 27 `$` signs to make it 32 total when I count the 5 chars contained in the word ‘beach’.

ljust(): left-justified string

Use the `ljust()` to left-justify a string.

```
word = 'beach'
number_spaces = 32

word_justified = word.ljust(number_spaces)

print(word)
#'beach'

print(word_justified)
#'beach                           '
```

Notice the spaces in the second string. The word ‘beach’ has 5 characters, which gives us 27 spaces to fill with empty space.

The original `word` variable remains unchanged, thus we need to assign the return of the method to a new variable, `word_justified` in this case.

The `ljust()` also accepts a specific char as a parameter to fill the remaining space.

```
word = 'beach'
number_chars = 32
char = '$'

word_justified = word.ljust(number_chars, char)

print(word)
#beach

print(word_justified)
#beach$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

Similar to the first situation, I have 27 `$` signs to make it 32 total when I count the 5 chars contained in the word ‘beach’.

isalnum(): checking alphanumeric only in a string

Use the `isalnum()` method to check if a string only contains alphanumeric characters.

```
word = 'beach'
print(word.isalnum())
#output: True

word = '32'
print(word.isalnum())
#output: True

word = 'number32' #notice there is no space
print(word.isalnum())
#output: True

word = 'Favorite number is 32' #notice the space between words
print(word.isalnum())
#output: False

word = '@number32$' #notice the special chars '@' and '$'
print(word.isalnum())
#output: False
```

isprintable(): checking printable characters in a string

Use the `isprintable()` method to check if the characters in a string are printable.

```
text = '' # notice this is an empty string, there is no white space here
print(text.isprintable())
#output: True

text = 'This is a regular text'
print(text.isprintable())
#output: True

text = ' ' #one space
print(text.isprintable())
#output: True

text = '                         '  #many spaces
print(text.isprintable())
#output: True

text = '\f\n\r\t\v'
print(text.isprintable())
#output: False
```

Notice that in the first 4 examples, all the character take some space, even if it is an empty space as you could see in the first example.

The last example returns `False`, showing 5 kind of characters that are non-printable: form feed `\f`, line feed `\n`, carriage return `\r`, tab `\t`, and vertical tab `\v`.

Some of these 'invisible' characters may mess up your printing giving you an unxpected output, even when everything 'looks' alright.

isspace(): checking white space only in a string

Use the `isspace()` method to check if the characters in a string are all white spaces.

```
text = ' '
print(text.isspace())
#output: True

text = ' \f\n\r\t\v'
print(text.isspace())
#output: True

text = '                         '
print(text.isspace())
#output: True

text = '' # notice this is an empty string, there is no white space here
print(text.isspace())
#output: False

text = 'This is a regular text'
print(text.isspace())
#output: False
```

Notice in the second example that white space is not only `' '`, but also form feed `\f`, line feed `\n`, carriage return `\r`, tab `\t`, and vertical tab `\v`.

startswith(): checking if a string begins with a certain value

Use the `startswith()` method to check if a string begins with a certain value.

```
phrase = "This is a regular text"

print(phrase.startswith('This is'))
#output: True

print(phrase.startswith('text'))
#output: False
```

You can also set if you want to begin the match in a specific position and end it in another specific position of the string.

```
phrase = "This is a regular text"

#the word regular starts at position 10 of the phrase
print(phrase.startswith('regular', 10))
#output: True

#look for in 'regular text'
print(phrase.startswith('regular', 10, 22))
#output: True

#look for in 'regul'
print(phrase.startswith('regular', 10, 15))
#output: False
```

Finally, you might want to check for multiple strings at once, instead of using some kind of loop, you can use a tuple as an argument with all the strings you want to match against.

```
phrase = "This is a regular text"

print(phrase.startswith(('regular', 'This')))
#output: True

print(phrase.startswith(('regular', 'text')))
#output: False
```

```
print(phrase.startswith(('regular', 'text'), 10, 22)) #look for in 'regular text'
#output: True
```

capitalize(): first character only to upper case in a string

Use the `capitalize()` method to convert to upper case only the first character in a string.

The rest of the string is converted to lower case.

```
text = 'this is a regular text'
print(text.capitalize())
#This is a regular text

text = 'THIS IS A REGULAR TEXT'
print(text.capitalize())
#This is a regular text

text = 'THIS $ 1S @ A R3GULAR TEXT!'
print(text.capitalize())
#This $ 1s @ a r3gular text!

text = '3THIS $ 1S @ A R3GULAR TEXT!'
print(text.capitalize())
#3this $ 1s @ a r3gular text!
```

Notice that any character counts, such as a number or a special character, thus, in the last example, `3` is the first character and suffers no alterations while the rest of the string is converted to lower case.

isupper(): checking upper case only in a string

Use the `isupper()` method to check if the characters in a string are all in upper case.

```
text = 'This is a regular text'
print(text.isupper())
#output: False

text = 'THIS IS A REGULAR TEXT'
print(text.isupper())
#output: True

text = 'THIS $ 1S @ A R3GULAR TEXT!'
print(text.isupper())
#output: True
```

If you notice the last example, the numbers and special characters like `@` and `$` in the string make no difference and `isupper()` still returns `True` because the method only verifies the alphabetical characters.

endswith(): check if a string ends with a certain value

Use the `endswith()` method to check if a string ends with a certain value.

```
phrase = "This is a regular text"

print(phrase.endswith('regular text'))
#output: True

print(phrase.endswith('This'))
#output: False
```

You can also set if you want to begin the match in a specific position and end it in another specific position of the string.

```
phrase = "This is a regular text"

#look for in 'This is', the rest of the phrase is not included
print(phrase.endswith('This is', 0, 7))
#output: True

#look for in 'This is a regular'
print(phrase.endswith('regular', 0, 17))
#output: True

#look for in 'This is a regul'
print(phrase.endswith('regular', 0, 15))
#output: False
```

Finally, you might want to check for multiple strings at once, instead of using some kind of loop, you can use a tuple as an argument with all the strings you want to match against.

```
phrase = "This is a regular text"

print(phrase.endswith(('regular', 'This', 'text')))
#output: True

print(phrase.endswith(('regular', 'is')))
#output: False
```

```
#look for in 'regular text'
print(phrase.endswith(('regular', 'text'), 10, 22))
#output: True
```

join(): join items of an iterable into one string

Use the `join()` method to join all the items if an iterable into a string.

The basic syntax is: `string.join(iterable)`

As per the syntax above, a string is required as a separator.

The method returns a new string, which means that the original iterator remains unchanged.

Since the `join()` method only accepts strings, if any element in the iterable is of a different type, an error will be thrown.

Let's see some examples with: string, list, tuple, set, and dictionary

join(): Strings

The `join()` method puts the `$` sign as a separator for every character in the string.

```
my_string = 'beach'

print('$'.join(my_string))
#output: b$e$a$c$h
```

join(): Lists

I have a simple list of three items representing car brands.

The `join()` method is gonna use the `$` sign as a separator.

It concatenates all the items on the list and puts the `$` sign between them.

```
my_list = ['bmw', 'ferrari', 'mclaren']

print('$'.join(my_list))
#output: bmw$ferrari$mclaren
```

This another example remembers you that `join()` does not work with non-string items.

When trying to concatenate the `int` items, an error is raised.

```
my_list = [1, 2, 3]

print('$'.join(my_list))
#output:
#Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
#TypeError: sequence item 0: expected str instance, int found
```

join(): Tuples

The tuple follows the same rationale as the list example explained before.

Again, I'm using the `$` sign as separator.

```
my_tuple = ('bmw', 'ferrari', 'mclaren')

print('$'.join(my_tuple))
#output: bmw$ferrari$mclaren
```

join(): Sets

Since the set is also the same as the tuple and the list, I've used a different separator in this example.

```
my_set = {'bmw', 'ferrari', 'mclaren'}
print('|'.join(my_set))
#output: ferrari|bmw|mclaren
```

join(): dictionaries

The dictionary has a catch when you use the `join()` method: it joins the keys, not the values.

This example shows the concatenation of the keys.

```
my_dict = {'bmw': 'BMW I8', 'ferrari': 'Ferrari F8', 'mclaren': 'McLaren 720S'}

print(','.join(my_dict))
#output: bmw,ferrari,mclaren
```

splitlines(): splitting a string at line breaks

Use the `splitlines()` method to split a string at line breaks.

The return of the method is a list of the lines.

```
my_string = 'world \n cup'

print(my_string.splitlines())
#output: ['world ', ' cup']
```

If you want to keep the line break, the `splitlines()` accepts a parameter that can be set to *True*, the default is *False*.

```
my_string = 'world \n cup'

print(my_string.splitlines(True))
#output: ['world \n', ' cup']
```

islower(): checking lower case only in a string

Use the `islower()` method to check if the characters in a string are all in lower case.

```
text = 'This is a regular text'
print(text.islower())
#output: False

text = 'this is a regular text'
print(text.islower())
#output: True

text = 'this $ 1s @ a r3gular text!'
print(text.islower())
#output: True
```

If you notice the last example, the numbers and special characters like `@` and `$` in the string make no difference and `islower()` still returns `True` because the method only verifies the alphabetical characters.

isnumeric(): checking numerics only in a string

Use the `isnumeric()` method to check if a string only contains numeric chars.

Numerics include numbers from 0 to 9 and combinations of them, roman numerals, superscripts, subscripts, fractions, and other variations.

```
word = '32'
print(word.isnumeric())
#output: True

print("\u2083".isnumeric()) #unicode for subscript 3
#output: True

print("\u2169".isnumeric()) #unicode for roman numeral X
#output: True

word = 'beach'
print(word.isnumeric())
#output: False

word = 'number32'
print(word.isnumeric())
#output: False

word = '1 2 3' #notice the space between chars
print(word.isnumeric())
#output: False

word = '@32$' #notice the special chars '@' and '$'
print(word.isnumeric())
#output: False
```

`isdecimal()` is more strict than `isdigit()`, which in its turn is more strict than `isnumeric()`.

isdigit(): checking digits only in a string

Use the `isdigit()` method to check if a string only contains digits.

Digits include numbers from 0 to 9 and also superscripts and subscripts.

```
word = '32'
print(word.isdigit())
#output: True

print("\u2083".isdigit()) #unicode for subscript 3
#output: True

word = 'beach'
print(word.isdigit())
#output: False

word = 'number32'
print(word.isdigit())
#output: False

word = '1 2 3' #notice the space between chars
print(word.isdigit())
#output: False

word = '@32$' #notice the special chars '@' and '$'
print(word.isdigit())
#output: False
```

`isdecimal()` is more strict than `isdigit()`, which in its turn is more strict than `isnumeric()`.

isdecimal(): checking decimals only in a string

Use the `isdecimal()` method to check if a string only contains decimals, that is, only numbers from 0 to 9 and combinations of these numbers.

Subscript, superscript, roman numerals, and other variations will be returned as `False`.

```
word = '32'
print(word.isdecimal())
#output: True

word = '954'
print(word.isdecimal())
#output: True

print("\u2083".isdecimal()) #unicode for subscript 3
#output: False

word = 'beach'
print(word.isdecimal())
#output: False

word = 'number32'
print(word.isdecimal())
#output: False

word = '1 2 3' #notice the space between chars
print(word.isdecimal())
#output: False

word = '@32$' #notice the special chars '@' and '$'
print(word.isdecimal())
#output: False
```

`isdecimal()` is more strict than `isdigit()`, which in its turn is more strict than `isnumeric()`.

isalpha(): checking letters only in a string

Use the `isalpha()` method to check if a string only contains letters.

```
word = 'beach'
print(word.isalpha())
#output: True

word = '32'
print(word.isalpha())
#output: False

word = 'number32'
print(word.isalpha())
#output: False

word = 'Favorite number is blue' #notice the space between words
print(word.isalpha())
#output: False

word = '@beach$' #notice the special chars '@' and '$'
print(word.isalpha())
#output: False
```

istitle(): checking if every word begins with an upper case char in a string

Use the `istitle()` method to check if the first character in every word in a string is upper case and the other characters are lower case.

```
text = 'This is a regular text'
print(text.istitle())
#output: False

text = 'This Is A Regular Text'
print(text.istitle())
#output: True

text = 'This $ Is @ A Regular 3 Text!'
print(text.istitle())
#output: True
```

If you notice the last example, the numbers and special characters like `@` and `$` in the string make no difference and `istitle()` still returns `True` because the method only verifies the alphabetical characters.

expandtabs(): set the number of spaces for a tab in a string

Use the `expandtabs()` method to set the number of spaces for a tab.

You can set any number of spaces, but when no argument is given, the default is 8.

Basic Usage

```
my_string = 'B\tR'

print(my_string.expandtabs())
#output: B       R
```

Notice the 7 spaces between the letters B and R.

The `\t` is at position two after one character, so it will be replaced with 7 spaces.

Let's look at another example.

```
my_string = 'WORL\tD'

print(my_string.expandtabs())
#output: WORL    D
```

Since `WORL` has four characters, the `\t` is replaced with 4 spaces to make it a total of 8, the default tabsize.

The code below gives us *4* spaces for the first tab after four characters 'WORL' and *7* spaces for the second tab after one character 'D'.

```
my_string = 'WORL\tD\tCUP'

print(my_string.expandtabs())
#output: WORL    D       CUP
```

Custom Tabsize

It is possible to set the tabsize as needed.

In this example the tabsize is *4*, which gives us 3 spaces after the char ‘B’.

```
my_string = 'B\tR'

print(my_string.expandtabs(4))
#output: B   R
```

This code has tabsize set to *6*, which gives us 5 spaces after the char ‘B’.

```
my_string = 'B\tR'

print(my_string.expandtabs(6))
#output: B     R
```

center(): centered string

Use the `center()` method to center a string.

```
word = 'beach'
number_spaces = 32

word_centered = word.center(number_spaces)

print(word)
#'beach'

print(word_centered)
##output: '              beach              '
```

Notice the spaces in the second string. The word ‘beach’ has 5 characters, which gives us 28 spaces to fill with empty space, 14 spaces before and 14 after to center the word.

The original `word` variable remains unchanged, thus we need to assign the return of the method to a new variable, `word_centered` in this case.

The `center()` also accepts a specific character as a parameter to fill the remaining space.

```
word = 'beach'
number_chars = 33
char = '$'

word_centered = word.center(number_chars, char)

print(word)
#beach

print(word_centered)
#output: $$$$$$$$$$$$$beach$$$$$$$$$$$$$
```

Similar to the first situation, I have 14 `$` in each side to make it 33 total when I count the 5 chars contained in the word ‘beach’.

zfill(): add zeros to a string

Use the `zfill()` to insert zeros `0` at the beginning of a string.

The amount of zeros is given by the number passed as argument minus the number of chars in the string.

The word 'beach' has 5 characters, which gives us 27 spaces to fill with zeros to make it 32 total as specified in the variable `size_string`

```
word = 'beach'
size_string = 32

word_zeros = word.zfill(size_string)

print(word)
#beach

print(word_zeros)
#00000000000000000000000000beach
```

The original `word` variable remains unchanged, thus we need to assign the return of the method to a new variable, `word_zeros` in this case.

Also notice that if the argument is less than the number of chars in the string, nothing changes.

In the example below, 'beach' has 5 chars and we want to add zeros until it reaches the `size_string` of 4, which means there is nothing to be done.

```
word = 'beach'
size_string = 4

word_zeros = word.zfill(size_string)

print(word)
#beach
```

```
print(word_zeros)
#'beach'
```

find(): check if a string has a certain substring

Use the `find()` method to check if a string has certain substring.

The method returns the index of the first occurrence of the given value.

Remember the index count starts at 0.

```
phrase = "This is a regular text"

print(phrase.find('This'))

print(phrase.find('regular'))

print(phrase.find('text'))
```

```
0
10
18
```

If the value is not found, the return will be `-1`.

```
phrase = "This is a regular text"

print(phrase.find('train'))
```

```
-1
```

You can also choose to begin the search in a specific position and end it in another specific position of the string.

```
phrase = "This is a regular text"

#look for in 'This is', the rest of the phrase is not included
print(phrase.find('This', 0, 7))
```

```
#look for in 'This is a regular'
print(phrase.find('regular', 0, 17))

#look for in 'This is a regul'
print(phrase.find('a', 0, 15))
```

```
0
10
8
```

Removing a Prefix or a Suffix in a String

As of Python 3.9, the String type will have two new methods.

You can specifically remove a prefix from a string using the `removeprefix()` method:

```
>>> 'Rio de Janeiro'.removeprefix("Rio")
' de Janeiro'
```

Or remove a suffix using the `removesuffix()` method:

```
>>> 'Rio de Janeiro'.removesuffix("eiro")
'Rio de Jan'
```

Simply pass as argument the text to be considered as prefix or suffix to be removed and the method will return a new string as a result.

I recommend the reading of the PEP 616 in the official documentation if you are curious about how these features are added to the language.

This one is a pretty simple change and very friendly for beginners to get used to reading the official documentation.

lstrip() vs removeprefix() and rstrip() vs removesuffix()

This is a confusion many people make.

It is easy to look at `lstrip()` and `removeprefix()` and wonder what is the real difference between the two.

When using `lstrip()`, the argument is a set of leading characters that will be removed as many times as they occur:

```
>>> word = 'hubbubbubboo'
>>> word.lstrip('hub')
'oo'
```

While `removeprefix()` will remove only the exact match:

```
>>> word = 'hubbubbubboo'
>>> word.removeprefix('hub')
'bubbubboo'
```

You can use the same rationale to distinguish between `rstrip()` and `removesuffix()`.

```
>>> word = 'peekeeneenee'
>>> word.rstrip('nee')
'peek'
```

```
>>> word = 'peekeeneenee'
>>> word.removesuffix('nee')
'peekeenee'
```

And as a bonus, just in case you have never worked with regular expressions before, be grateful that you have `strip()` to trim

character sets from a string instead of a regular expression:

```
>>> import re
>>> word = 'amazonia'
>>> word.strip('ami')
'zon'
>>> re.search('^[ami]*(.*?)[ami]*$', word).group(1)
'zon'
```

Slicing

Slicing is one of the most useful tools in the Python language.

As such, it is important to have a good grasp of how it works.

Basic Notation

Let's say we have an array called 'list'.

```
list[start:stop:step]
```

- start: where you want the slicing to begin
- stop: until where you want the slicing to go, but remember the value of *stop* is not included
- step: if you want to skip an item, the default being 1, so you go through all items in the array

Indexes

When slicing, The indices are points in *between* the characters, not on the characters.

For the word 'movie':

```
 +---+---+---+---+---+
 | m | o | v | i | e |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

If slice from 0 until 2, I get 'mo' in the example above and *not* 'mov'.

Since a string is just a list of characters, the same applies with to list:

```
my_list = [1, 2 , 3, 4, 5]
```

Becomes:

```
 +---+---+---+---+---+
 | 1 | 2 | 3 | 4 | 5 |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

Examples

We have a variable containing the string ‘movie’ like so:

```
word = 'movie'
```

All the examples below will be applied to this word.

Example 1

To get the first two characters:

```
sliced = word[:2]
print(sliced)
mo
```

Notice that we could have used 0 to denote the beginning, but that is not necessary.

Example 2

The last item:

```
sliced = word[-1]
```

```
print(sliced)
e
```

Example 3

Skipping letters with a step of 2:

```
sliced = word[::2]
print(sliced)
mve
```

Example 4

A nice trick is to easily revert an array:

```
sliced = word[::-1]
print(sliced)
eivom
```

The default step is `1`, that is, go forward 1 character of the string at a time.

If you set the step to `-1` you have the opposite, go back 1 character at a time beginning at the end of the string.

How to reverse a string

To reverse a string use the slice syntax:

```
my_string = "ferrari"

my_string_reversed = my_string[::-1]

print(my_string)

print(my_string_reversed)
```

```
ferrari

irarref
```

The slice syntax allows you to set a step, which is `-1` in the example.

The default step is `1`, that is, go forward 1 character of the string at a time.

If you set the step to `-1` you have the opposite, go back 1 character at a time.

So you start at the position of the last character and move backwards to the first character at position 0.

String Interpolation with f-strings

If you need to concatenate a string and another type, you have to do typecasting when using the print function as explained in Type casting in Python.

So to convert `age` to a string you make `str(age)` in order to print a phrase using the `+` sign.

```
name = 'Bob'
weight = 80

print('My name is ' + name + ' and I weight ' + str(weight) + ' kg')
```

```
My name is Bob and I weight 80 kg
```

But that is not the best way to handle situations like this.

The best solution is to use String Interpolation, also called **f strings**.

Let's first see how our example looks like using string interpolation.

```
name = 'Bob'
weight = 80

print(f'My name is {name} and I weight {weight} kg')
```

```
My name is Bob and I weight 80 kg
```

Notice the `f` at the beginning signaling to the interpreter that we are going to use interpolation, the presence of this `f` is the reason why this is also called **f strings**.

After the `f` you start your string, as usual, using quotes.

The key difference is that when you want to evaluate an expression like using the value of a variable, you just put them inside curly braces.

This is a simpler and more comfortable way to write very complex strings and you don’t have to worry about type conversion using type casting.

Conclusion

That's it!

Congratulations on reaching the end.

I want to thank you for reading this book.

If you want to learn more, check out my blog renanmf.com.

Let me know if you have any suggestions by reaching out to me at renan@renanmf.com.

You can also find me as @renanmouraf on:

- Twitter: https://twitter.com/renanmouraf
- Linkedin: https://www.linkedin.com/in/renanmouraf
- Instagram: https://www.instagram.com/renanmouraf