Basic Programming

Lesson 4. Strings. Working with strings.

Alexandra Ciobotaru

29 oct 2021

Syllabus

- **Lesson 1. Computers, Programming and Cognitive Science. From pseudocode to programming languages.**
- **Lesson 2. Variables in Python. Basic calculus. Using Math library, type() and help() functions.**
- **Lesson 3. Collections: Lists, Tuples.**
- **Lesson 4. Strings. Working with strings.**
- Lesson 5. Branching and decisions: Logical operators, If-Statements, Nested conditions. Loops: For and While
- Lesson 6. Working with matrices in Python. Python numpy library. Quiz 1 (25%).
- Lesson 7. Collections: Dictionaries. JSON construction.
- Lesson 8. Working with files. Reading and Writing.
- Lesson 9. Analyzing dataframes. Pandas and Matplotlib Python libraries.
- Lesson 10. Creating functions. Recursive functions. Quiz 2 (25%).
- Lesson 11. Object-Oriented Programming: Encapsulation, Inheritance and Polymorphism.
- Lesson 12. Object-Oriented Programming (cont.). Error handling. Best practices when programming.
- Lab (20%) + final exam (30%).

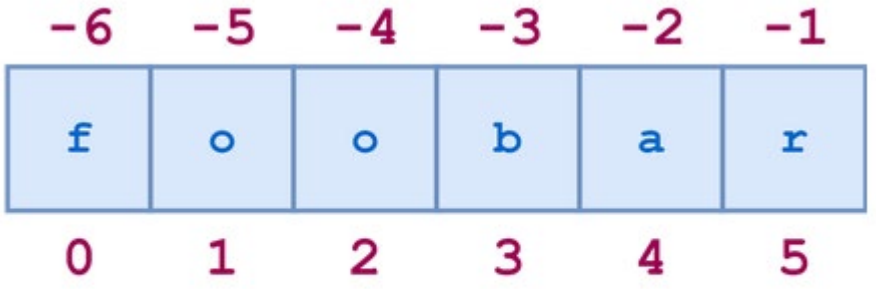
What is a string?

- Strings are sequences of characters. Your name can be considered a string. Or, say you live in Ecuador, then your country name is „Ecuador", which is a string.
- You can define a string variable either using single quotes or double quotes:

```
my_single_quotes_string = 'hello'
my_double_quotes_string = "hello"
print(my_single_quotes_string is my_double_quotes_string)
>>>True
```

Indexing

- Strings are made of substrings - individual items in an ordered set of data can be accessed directly using a numeric index or key value;
- In Python, strings are ordered sequences of character data, and thus can be indexed in this way. Individual characters in a string can be accessed by specifying the string name followed by a number in square brackets ([]).
- Just like lists, string indexing is also zero-based: the first character in the string has index 0, the next has index 1, and so on. The index of the last character will be the length of the string minus one.



Positive and Negative String Indices

Image source: https://realpython.com/python-strings/

```
>>> s = 'foobar'
>>> s[0]
'f'
>>> s[3]
'b'
>>> len(s)
6
>>> s[len(s)-1]
'r‘
>>> s[-1]
'r'
>>> s[-2]
'a'
```

Slicing

- Just like lists, an expression of the form s[m:n] returns the portion of s starting with position m, and **up to but not including** position n:

```
>>> s = 'foobar'
>>> s[2:5]
'oba‘
>>> s[:4]
'foob'
>>> s[0:4]
'foob‘
```

- If you omit the first index, the slice starts at the beginning of the string. Thus, s[:m] and s[0:m] are equivalent. Similarily when you omit the last index.

```
>>> s = 'foobar'
>>> s[:4]
'foob'
>>> s[0:4]
'foob'
>>> s[2:]
'obar'
>>> s[2:len(s)]
'obar'
```

Remember: String indices are zero-based. The first character in a string has index 0. This applies to both standard indexing and slicing.

Strings are immutable

- Strings are considered immutable objects, meaning that you cannot change its elements, at some specific index:

```
>>> s = 'foobar'
>>> s[3] = 'x'
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    s[3] = 'x'
TypeError: 'str' object does not support item assignment
```

- But there is a solution:

```
>>> s = s.replace('b', 'x')
>>> s
'fooxar'
```

String common methods

Remember: Simply calling a method does not change the original variable.

- Get the index of a substring in a string.

```
>>> a = 'hello there'
>>> a.index('h')
0
>>> a.index('l')
2
```

- Join a list of strings using the join method.

```
>>>  combined_string = " ".join(["1", "2", "3"])
'1 2 3'
```

- Break a string based on some rule

```
>>>  "1 2 3".split() # splitting
   ['1', '2', '3']
```

- Or by a delimiter

```
>>> “1:2:3”.split(“:”)
    [‘1’, ‘2’, ‘3’]
```

- Strip spaces at the beginning and at the end of a string:

```
>>> "   something   ".strip()
'something‘
```

- You can also strip some other characters as well:

```
>>> txt = ",,,,,rrttgg.....banana....rrr"
>>> txt.strip(",.grt")
'banana'
```

String common methods

- .**upper**() – uppercase

```
>>> my_string = 'This is my string.'
>>> my_upper_string = my_string.upper()
>>> my_upper_string
'THIS IS MY STRING.‘
```

- .**lower**() – lowercase

```
>>> my_lower_string = my_upper_string.lower()
>>> my_lower_string
'this is my string.'
```

- .**capitalize**() – capitalize first letter in string

```
>>> my_lower_string.capitalize()
'This is my string.'
```

- .**title**() – capitalize each word

```
>>> my_lower_string.title()
'This Is My String.'
```

- .**startswith**(<suffix>, <index>) – checks if a string starts with some characters

```
>>> my_string.startswith('x')
False
```

- .**endswith**(<suffix>, <index>) – checks if a string ends with some characters

```
>>> my_string.endswith('.')
True
```

- .**isdigit**() – checks if the string s in made only of digits

```
>>> my_string.isdigit()
False
```

- .**isalpha**() – checks if the string s in made only of letters

```
>>> my_string.isalpha()
False
>>> my_string = 'Thisismystring‘
>>> my_string.isalpha()
True
```

Escaping characters

- To insert characters that are illegal in a string, use an escape character.
- An escape character is a backslash \ followed by the character you want to insert.
- An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

```
>>> txt = "We are the so-called "Vikings" from the north.“
SyntaxError: invalid syntax
>>> txt = "We are the so-called \"Vikings\" from the north."
```

\\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab

Other common escape characters used in Python

Regular expressions

- A *regular expression* is a sequence of characters that specifies a search pattern.
- You will use regular expressions to clean the text in your dataset before feeding it to the machine learning model.

- Python module used to work with regular expressions -> **re**
- Finding all occurrences of a specific string in text:

```
import re
text = 'I have called the service desk 100 times and nobody replies to me. I need a conversation ASAP!! My number is 111-1234567! I repeat, my number is 111-1234567 .'
result = re.findall('111-1234567', text)
print(result)
['111-1234567', '111-1234567']
```

- What if I want to find digits in my text?

```
>>> result = re.findall(r'\d', text)
>>> print(result)
['1', '1', '1', '1', '2', '3', '4', '5', '6', '7', '1', '0', '0', '1', '1', '1', '1', '2', '3', '4', '5', '6', '7']
```

- r - means ‘raw string’ – used here makes python not treat ‘\’ as an escape character:

```
>>> print('hello there \n beautiful \n people')
hello there
 beautiful
 people
>>> print(r'hello there \n beautiful \n people')
hello there \n beautiful \n people
```

- To get ALL the digits – use \d+ pattern:

```
result = re.findall(r'\d+', text)
print(result)
['111', '1234567', '100', '111', '1234567', '111', '1234567']
```

the ‘+’ after ‘\d’ means ONE or MORE digit.

- To match a pattern for telephone numbers:

```
>>> result = re.findall(r'\d{3}-\d{7}', text)
>>> print(result)
['111-1234567', '111-1234567', '111-1234567']
```

{3} — 3 digits, {7} — digits

- Using the OR operator (‘|’) to find either telephone numbers with prefix, or without prefix:

```
>>> text = 'I have called the service desk 100 times and nobody replies to me. I need a conversation ASAP!! My number is 111-1234567! My other number is 7654321!'
>>> result = re.findall(r'\d{3}-\d{7}|\d{7}', text)
>>> print(result)
['111-1234567', '7654321']
```

- Similar for digits, to find characters, use \w:

```
>>> text = 'abcabc aa cc dd e 123123 abcabab'
>>> result = re.findall(r'\w{3}', text)
>>> print(result)
['abc', 'abc', '123', '123', 'abc', 'aba']
```

- Repetition:
  - >>> result = re.findall(r'(\w{3})(\1)', text)
  - >>> print(result)
  - [('abc', 'abc'), ('123', '123')]

Use brackets if you have more that one raw rule

- Here (\w{3}) means 3 characters. (\1) means the following 3 characters need to be the same as (\w{3}). Here ‘1’ means the first bracket.

Substitution in text using re

- This is an example of text cleaning before training:

```
import re
text = "I am learning regular expressions from: https://github.com/dlab-berkeley/regular-expressions-in-python/blob/master/.ipynb_checkpoints/tutorial-checkpoint.ipynb"
new_text  = re.sub(r'(?P<url>https?://[^\s]+)', " ", text)
print(new_text)
>>> I am learning regular expressions from:
```

- A handy website to test regular expressions: https://regex101.com/

Truth value testing of a String

- A string is considered to be true in Python if it is not an empty string. So, we get the following:
- # Test truth value of empty string
- >>> print(bool(""))
- False

- # Test truth value of non-empty string "x"
- >>> print(bool("x"))
- True

- Test if a substring is a member of a larger string.

  >>> "i" in "pythonic"
  True
  "x" in "pythonic" # "x" is not present in "pythonic"
  >>> False

String formatting (remember)

- You can use %s as a formatter which will enable you to insert different values into a string at runtime and thus format the string. The %s symbol is replaced by whatever is passed to the string.

```
>>>  print("I love %s in %s" % ("programming", "Python"))
'I love programming in Python‘
```

- You can also use the keyword format. This will enable you to set your own formatters instead of %s.

```
print("I love {programming} in {python}".format(programming="programming", python="Python"))
'I love programming in Python'
```

Operations on strings (remember)

- Remember strings can be added and multiplied with an integer, but not substracted

```
first_name = "Johan“
last_name = "Gambolputty"
full_name = first_name + " " + last_name
print(full_name)
```

Thank you!