Andrew Shitov

A Tour of Python 3

300+ ready-to-use examples guiding you from the very beginning

Including Python 3.8

DeepText — 2020

A Tour of Python 3

300+ ready-to-use examples guiding you from the very beginning. Including Python 3.8

This book is a collection of small programs that demonstrate different features of the Python programming language. The reader can use this book as a textbook and a collection of exercises for learning Python 3.

The 1st edition published on 13th July 2020

Published by DeepText, Amsterdam
https://andrewshitov.com/a-tour-of-python3

Preface

Dear reader,

This book is an overview of the features of the Python programming language. In the 36 chapters, there are more than 300 small programs, where you will learn many details about all main aspects of programming in this language.

The book is primarily intended to be used with the modern compiler of the third version of the language and highlights the differences between Python 2 and Python 3 where that makes sense. A number of sections explain the new features appeared in Python 3.8.

In general, the difficulty of the examples increases to the end of the book, but you can read the chapters and the sections in any order.

To run the examples, you need a working compiler, which you can download and install from the official site python.org.

The author wishes you a pleasant journey to the world of Python.

Andrew Shitov

July 12, 2020
Amsterdam

Contents

Chapter 4. Numbers

Chapter 5. Strings

Chapter 9. Types

Chapter 10. Lists

Chapter 11. Dictionaries

Chapter 12. Sets

Chapter 13. Ranges

Chapter 14. Regular Expressions

Chapter 15. Date and Time

Part 3

Chapter 16. Control Flow

Chapter 19. Files

Chapter 20. Working with the System

Chapter 24. Multi-Dimensional Lists

Chapter 25. Iterators and Generators

Chapter 31. Maths Problems

Chapter 32. String Problems

Chapter 33. Interview Problems

Part 7

Chapter 34. Internet and Web

Chapter 35. Graphics

Chapter 36. Miscellaneous

Chapter 5

Strings

The length of a string

What is the length of the string?

Save the following file as a UTF-8 file:

```
string = '你好，世界！'

print(len(string)) # Length in characters

print(len(string.encode('utf-8')))
# Length in bytes (if it is UTF-8)
```

In Python 3, you mostly don't have to care about the encoding of text strings if you work with UTF-8.

To get a length of the string in characters, use the built-in `len` function.

To get a length in bytes, you need to explicitly convert the string to UTF-8 and get its length.

```
$ python3 len.py
6
18
```

Multi-line strings

Assign a long string to a variable.

Sometimes you have a long string, but you'd like to keep the lines of your program short. Or you have a string that contains a few logical lines (that is, the string has newline characters in it).

```
str = '''This is a multi-line
string. That you can continue
on another line pressing the
Enter key'''
print(str)
```

Use triple quotes `'''` to create a multi-line string in Python. Just simple as that.

Take care of indentation within the quotes.

```
str2 = '''Notice the spaces
          on the left. Will they
          appear in the output?
'''
print(str2)
```

Run the program:

```
$ python3 multiline.py
This is a multi-line
string. That you can continue
on another line pressing the
Enter key
Notice the spaces
          on the left. Will they
          appear in the output?
```

Indexing a string character

Print the first, the second, and the third character of a string.

Accessing single characters in a string is as easy as providing character's positions.

```
s = 'Hello, World!'
#    01234567...

print(s[0]) # H
print(s[1]) # e
print(s[7]) # W
```

Here is what the program prints:

```
$ python3 sting-index.py
H
e
W
```

Don't forget that Python counts characters from 0, so the index of the first character is 0, the index of the second character is 2 and so on.

```
s = 'Hello, World!'
#    ^^^^^^^^
#    01234567...
```

Taking the last character

Print the last character of the given string.

```
str = 'Hello, World!'
#                  ^

print(str[-1])
```

Of course, you can first determine the length of the string, and then index the corresponding characters, but there is a simpler way.

To count characters from right to left, use negative indices. For instance, to get the last character, index it as `[-1]`.

```
$ python3 last-character.py
!
```

The second-to-last character

Print the second-to-last character of the given string.

To take the second-to-last character you will index it as `[-2]`:

```
str = 'Hello, World!'
#                 ^

print(str[-2]) # d
```

Here is the result:

```
$ python3 last-character-2.py
d
```

Getting a substring

From a given string, take and print its part.

To make it easier for understanding, the index counting of the characters are shown in the comment. In the second comment, the desired part is marked with the `^` characters in the comment.

```
str = 'Hello, World!'
#      0123456789012
#             ^^^^^

# I want to print "World"
```

```
str_world = str[7:12]
print(str_world)
```

With ranges, you specify the index of the first character to copy and the index of the character followed by the last one you need.

```
$ python3 substring.py
World
```

Notice that the second number (12 in the example) is the index of the character after the last character in the substring. The letter `W` has index 7, and the letter `d` has index 11, thus the range is `[7:12]`.

String repetition

*Explaining the * operator when it is applied to strings.*

```
str = 'A'
long_str = str * 15
print(long_str)
```

In Python, you can use * to repeat the string the given number of times.

```
$ python3 string-repetition.py
AAAAAAAAAAAAAAA
```

Reversing a string

Reverse a string and print it.

```
str = 'Hello, World!'
rev_str = str[::-1]

print(rev_str)
```

To reverse a string, walk along it using a range with the step `-1` from right to left. As we want the whole string to be processed, the beginning and the end of the range can be omitted: `[::-1]`. With such a range, index counting goes backwards.

```
$ python3 reverse-string.py
!dlroW ,olleH
```

Splitting a string

How to split a string into characters.

```
str = 'Hello, World!'

# First, print a character
print(str[3])

# Now, make a list out of a string
# Just convert a type:
chars = list(str)
print(type(chars))
print(chars[3])
```

To split a string into characters, convert the string to a list, and you'll get a list of separate characters. Having a list, you can access individual characters directly by indexing a string.

```
$ python3 split-string.py
l
<class 'list'>
l
```

Raw strings

Understanding raw strings in Python.

Here is a regular string:

```
print('Hello, \nWorld!')
```

And here is the so-called raw string. It is prefixed with `r`:

```
print(r'Hello, \nWorld!')
```

In the second example, no escaping happens, and the `\n` sequence appears in the output as is, not as a newline.

```
$ python3 raw-strings.py
Hello,
World!
Hello, \nWorld!
```

More on raw strings

Strings and raw strings.

Here is a regular string with an escaped character:

```
s1 = 'a\tb'
```

And here is a raw string, where `\t` is not escaped and will appear as is in printing:

```
s2 = r'a\tb'
```

```
print(s1)
print(s2)
```

Normally, a character prefixed with a backslash is a special character such as a newline `\n` or a tab `\t`.

In the raw strings, backslash has no special meaning. You prefix raw strings with `'r'`

```
$ python3 raw-string.py
a   b
a\tb
```

Removing accents from a string

Removing accents from characters.

This program removes accents from the accented letters such as é or ü.

```
import unidecode

str = 'Café München'

str = unidecode.unidecode(str)
print(str)
```

One of the simplest ways to do that is to use a module called `unidecode`. It does not come with the default Python distribution but can be easily installed using `pip`.

```
$ python3 remove-accents.py
Cafe Munchen
```

Removing \n

Remove newline characters from the strings in a list.

You have a list of strings which have a newline character at the end and you want to remove the newlines from them.

```
data = [
    'alpha\n',
    'beta\n',
    'gamma\n'
]
print(data)

data = [
    s.strip() for s in data
    # List comprehension
]
print(data)
```

Here, we are using the list comprehension that loops over the strings in `data` and calls the `strip` method on each of the strings. As a result, the data gets cleanrd and you have pure strings without newlines in them.

```
$ python3 strip.py
['alpha\n', 'beta\n', 'gamma\n']
['alpha', 'beta', 'gamma']
```

A more complex stripping

Another way of removing the newline characters from the strings in a list.

Let's solve the previous task or removing `\n` from a list of stings using the built-in `map` function.

```
data = [
    'alpha\n',
    'beta\n',
    'gamma\n'
]
print(data)

def strip(s):
    return s.strip()

# Let's use "map" this time
data = list(map(strip, data))
# map returns an object of the "map" type,
# so convert it to a list
print(data)
```

`map` requires a function that takes one argument and returns one argument. Here, we define our own function called `strip` that simply calls the `strip` method on the string. When `map` loops over the data items, the function is called for each of them, and the result is collected in the resulting object.

```
$ python3 strip2.py
['alpha\n', 'beta\n', 'gamma\n']
['alpha', 'beta', 'gamma']
```

Using lambdas to strip a line

Even more complex way to remove the newline characters from the strings in a list.

Here is another solution of the same task of removing newline characters from the strings in a list.

```
data = [
    'alpha\n',
    'beta\n',
    'gamma\n'
]
print(data)
```

Let's replace a separate function with a lambda.

```
data = list(map(
    lambda s: s.strip(), data))

print(data)
```

This code is based on the previous program, but instead of stripping a string in a separate function, it uses an inline lambda.

```
$ python3 strip3.py
['alpha\n', 'beta\n', 'gamma\n']
['alpha', 'beta', 'gamma']
```

ASCII values of the characters

Print ASCII values of all characters in a given string.

```
str = 'Hello'

ascii = [ord(s) for s in str]

print(ascii)
```

This program gets a string and prints a list containing the ASCII numbers of each character in that string.

Here, the built-in `ord` function is used to convert a character to its ASCII value (or, in other words, ASCII position number).

To make the code simple, list comprehension is used.

```
$ python3 str-char-nums.py
[72, 101, 108, 108, 111]
```

Character Unicode names

Printing the Unicode names of the characters.

```
import unicodedata

str = 'Héḷḷö'

for s in str:
    print(unicodedata.name(s))
```

Each character has its name in the Unicode space. To get the names, use the `name` function from the `unicodedata` module (it comes with the Python distribution).

```
$ python3 unicode-name.py
LATIN CAPITAL LETTER H
LATIN SMALL LETTER E WITH ACUTE
LATIN SMALL LETTER L WITH CEDILLA
LATIN SMALL LETTER L WITH CEDILLA
LATIN SMALL LETTER O WITH DIAERESIS
```

Using casefolding

Handling the German letter ß (which is equivalent to 'ss').

```
str1 = 'strasse'
str2 = 'straße' # street in English

print(str1 == str2) # False

print(
    str1.casefold() ==
    str2.casefold()
) # True
```

Before talking about actual normalisation in Unicode, let's see at how the German letter *ß* can be compared with its equivalent sequence *ss* (let's omit the tiny grammatical differences that may occur). Use the `casefold` string method, which not only converts the string to lowercase but also take care of converting ß to ss. For example, `'Straße'.casefold()` gives the string `'strasse'`.

```
$ python3 unicode-normalize.py
False
True
```

Unicode normalization

Canonical normalization in Unicode.

Unicode normalization is required to get different texts to the common representation. For example, the letter `á` can be represented in two ways: by a single character and by a pair of codepoints, a Latin letter `a`

and a combining character for the acute. You can see them both in the following program.

```
import unicodedata

s1 = 'á'
s2 = 'a\u0301' # combining acute

print(s1 == s2) # False

s3 = unicodedata.normalize('NFC', s2)
print(s1 == s3) # True now
```

In the Unicode, 'complex' characters can be represented either by their direct code points (if they exist) or by a sequence of a character modified by combining characters (such as acute). To compare strings encoded differently, you need to normalize them first. Use the `normalize` function from the `unicodedata` module.

```
$ python3 unicode-normalize2.py
False
True
```

Split words

Two methods to split a word into separate letters.

In this program, two methods of splitting a word into letters are demonstrated.

```
word = 'Hello'

# Method 1: converting to a list
letters1 = list(word)
```

```
# Method 2: using list comprehension
letters2 = [ch for ch in word]

print(letters1)
print(letters2)
```

In the first method, you just call the `list` function and convert a string to a list, which ends up with a list of characters.

In the second method, you use list comprehension to loop over the characters in a string (`ch in word`) and collect them to a list.

```
$ python3 split-word.py
['H', 'e', 'l', 'l', 'o']
['H', 'e', 'l', 'l', 'o']
```