Datentypen

Datentypen sind in Python als Klassen implementiert.

```
>>> type(42)
<class 'int'>
>>> type(3.14)
<class 'float'>
>>> type('Hallo')
<class 'str'>
>>>
```

Deshalb haben sie auch Methoden:

```
>>> dir(int)
['__class__', '__name__', 'from_bytes', 'to_bytes', '__bases__', '__dict__']

>>> dir(str)
['__class__', '__name__', 'count', 'endswith', 'find', 'format', 'index',
 'isalpha', 'isdigit', 'islower', 'isspace', 'isupper', 'join', 'lower',
 'lstrip', 'replace', 'rfind', 'rindex', 'rsplit', 'rstrip', 'split',
 'startswith', 'strip', 'upper', '__bases__', '__dict__', 'center', 'encode',
 'partition', 'rpartition', 'splitlines']

>>> dir(float)
Guru Meditation Error: Core  1 panic'ed (LoadProhibited). Exception was unhandled.
Core 1 register dump:
PC      : 0x400e8bc6  PS      : 0x00060630  A0      : 0x800df1a5  A1      : 0x3ffc25d0
A2      : 0x3f409668  A3      : 0x00000002  A4      : 0x3ffc26b0  A5      : 0x00000000
A6      : 0x000000b3  A7      : 0x3ffc2810  A8      : 0x00000000  A9      : 0x3ffc25d0
A10     : 0x00000000  A11     : 0x3ffc2a58  A12     : 0x00000000  A13     : 0x00000000
A14     : 0x00000001  A15     : 0x3f409668  SAR     : 0x0000001e  EXCCAUSE: 0x0000001c
EXCVADDR: 0x00000004  LBEG    : 0x4000c46c  LEND    : 0x4000c477  LCOUNT  : 0x00000000

ELF file SHA256: 0000000000000000

Backtrace: 0x400e8bc3:0x3ffc25d0 0x400df1a2:0x3ffc2610 0x400df214:0x3ffc2630 0x400eaa5d:0x3ffc26b0 0x400e34be:0x3ffc26e0 0x400df489:0x

Rebooting...


       _  __ _
 _   _(_)/ _| | _____      __
| | | | | |_| |/ _ \ \ /\ / /
| |_| | |  _| | (_) \ V  V /
 \__,_|_|_| |_|\___/ \_/\_/

APIKEY: FFBFFD22
```

None

Auch None ist eine eigene Klasse:

```
>>> type(None)
<class 'NoneType'>
>>>

>>> dir(None)
['__class__']
>>>
```

Diese Klasse hat aber keine Methoden.

> **Tip pythonisch:**
> Wenn man auf None testen will sollte das mit ‘is’, dem Identitätsvergleich, geschehen und nicht mit ‘==’ dem Wertevergleich.

```
# Eien Funktion ohne Rückgabewert liefert None zurück
>>> def test_none():
        pass

>>> a = test_none()
>>> a
>>>                          # a liefert keinen Wert zurück

# Es geht aber auch einfach, wie fast alles in Python:

>>> a = None
>>> a
>>>

>>> a is None
True                        # Der Inhalt von a ist None!
>>>

>>> a == None               # Geht auch, ist aber nicht pythonisch!
True
>>>
```

bool

Bool ist eine eigene Klasse. Es enthält die Eigenschaften `True` und `False`.

Der Versuch Methoden von `bool` zu finden endet im Nirwana bei der Guru Meditation:

```
>>> type(True)
<class 'bool'>
>>>

>>> dir(bool)
Guru Meditation Error: Core  1 panic'ed (LoadProhibited). Exception was unhandled.
Core 1 register dump:
PC      : 0x400e8bc6  PS      : 0x00060730  A0      : 0x800df1a5  A1      : 0x3ffc25d0
A2      : 0x3f40873c  A3      : 0x00000002  A4      : 0x3ffc26b0  A5      : 0x00000000
A6      : 0x000000b3  A7      : 0x3ffc2810  A8      : 0x00000000  A9      : 0x3ffc25d0
A10     : 0x00000000  A11     : 0x3ffc2a58  A12     : 0x00000000  A13     : 0x00000000
A14     : 0x00000001  A15     : 0x3f40873c  SAR     : 0x00000020  EXCCAUSE: 0x0000001c
EXCVADDR: 0x00000004  LBEG    : 0x4000c46c  LEND    : 0x4000c477  LCOUNT  : 0x00000000

ELF file SHA256: 0000000000000000

Backtrace: 0x400e8bc3:0x3ffc25d0 0x400df1a2:0x3ffc2610 0x400df214:0x3ffc2630 0x400eaa5d:0x3ffc26b0 0x400e34be:0x3ffc26e0 0x400df489:0x

Rebooting...


       _  __ _
 _   _(_)/ _| | _____       __
| | | | | |_| |/ _ \ \ /\ / / /
| |_| | |  _| | (_) \ V  V /
 \__,_|_|_| |_|\___/ \_/\_/

APIKEY: FFBFFD22
```

`bool` hat wohl keine Methoden.

Bytes

Bytes sind Werte die mit 8-Bit dargestellt werden können (0 ... 255). Bytes kommen als Datentyp in Python nicht vor. Es können aber Zahlen und Strings in eine Bytefolge und umgekehrt umgewandelt werden. Das wird benutzt um Daten zur Übertragung zu serialisieren. Dazu gibt es 2 Datentypen:

bytestring

bytearray

Literale

Wenn man etwas über Python oder auch andere Programmiersprachen ließt, trifft man immer wieder auf den Begriff **Literal**. Was ist damit gemeint?

Als Literal bezeichnet man die schriftliche Darstellung einer Zahl. Also z.B. 1, 1.0 oder 1e0. Das sind drei unterschiedliche Literale der Zahl 1.

int

`int` ist die Klasse für ganze Zahlen. Sie enthält alle positiven und negativen ganzen Zahlen. Die Größe eines `int` ist nicht festgelegt. Sie ist praktisch durch die Speichergröße begrenzt.

```
>>> type(42)
<class 'int'>
>>> type(-42)
<class 'int'>
>>> type(0)
<class 'int'>
>>>
```

```
>>> 10 ** 100
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000

>>> 10 ** 1000000
MemoryError: memory allocation failed, allocating 34258 bytes
>>>
```

Methoden von `int`

```
>>> dir(int)
['__class__', '__name__', 'from_bytes', 'to_bytes', '__bases__', '__dict__']
```

from_bytes(bytearray, 'little|big')

Wandelt Bytes in ein `int` um. Es muss die Reihenfolge der Bytes angegeben werden, `little` Endian oder `big` Endian.

https://www.codegrepper.com/code-examples/python/from_bytes+python

https://www.delftstack.com/de/howto/python/how-to-convert-bytes-to-integers/

https://stackoverflow.com/questions/50509017/how-is-int-from-bytes-calculated

to_bytes(int, 'little|big')

Wandelt ein `int` in Bytes um. Es muss die Reihenfolge der Bytes angegeben werden, `little` Endian oder `big` Endian.

Unterstriche zur Strukturierung großer Zahlen

Bei sehr großen Zahlen unterteilen wir die tausender Stellen gerne mit Punkten. Das ist bei der Eingabe von Zahlen auch in Python möglich. Allerdings wird hier nicht der Punkt, sondern der Unterstrich '_' verwendet:

```
>>> a = 100_000_000
>>> a
100000000
>>>
```

Funktionen die auf `int` angewendet werden können.

abs(),

float

Die Wesentliche Eigenschaften von `float` wurden schon bei der Vorstellung von Python erörtert.

Die Funktionen für `int` lassen auch auf `float` anwenden. Im Modul math gibt es eine Reihe weitere Funktionen, die vor allem für den Typ `float` interssant sind.

Eine Übersicht findet man hier: https://docs.micropython.org/en/v1.12/library/math.html

Neben den trigonometrischen und logarithmischen Funktionen gibt es ein paar weitere die ich hier vorstellen möchte:

`math.ceil( x )`

Gibt eine ganze Zahl zurück, wobei x auf positive Unendlichkeit gerundet wird.

```
>>> math.ceil(3.14)
4
>>>
```

`math.copysign( x , y )`

Gibt x mit dem Vorzeichen von y zurück.

```
>>> math.copysign(math.pi, -1)
-3.141593
>>>
```

`math.fabs( x )`

Gibt den absoluten Wert von x zurück.

```
>>> math.copysign(math.pi, -1)
-3.141593

>>> math.fabs(_)
3.141593
>>>
```

`math.floor( x )`

Gibt eine ganze Zahl zurück, wobei x gegen negative Unendlichkeit gerundet wird.

```
>>> math.floor(3.14)
3
>>> math.floor(-3.14)
-4
>>>
```

`math.fmod( x , y )`

Gibt den Rest von x/y zurück.

```
>>> math.fmod(10, 3)
1.0
>>>
```

`math.isfinite( x )`

Gibt `True` zurück, wenn x endlich ist.

Endlich ist eine Zahl dann, wenn sie im gültigen Zahlenbereich von `float` liegt.

```
>>> math.isfinite(1e40)
False
>>> math.isfinite(1e38)
True
>>>
```

`math.isinf( x )`

Gibt True zurück, wenn x unendlich ist.

Unendlich ist eine Zahl dann, wenn sie ausserhalb des gültigen Zahlenbereichs von `float` liegt.

```
>>> math.isinf(1e40)
True
>>> math.isinf(1e38)
False
>>>
```

`math.modf( x )`

Gibt ein Tupel aus zwei Floats zurück, die den gebrochenen und den ganzzahligen Teil von x darstellen. Beide Rückgabewerte haben das gleiche Vorzeichen wie x.

```
>>> math.modf(3.14)
(0.1400001, 3.0)
>>>
```

Hier sieht man einen der gefürchten Rundungsfehler bei floats.

`math.pow( x , y )`

Gibt x als Potenz von y zurück.

```
>>> math.pow(2, 8)
256.0
>>>
```

`math.sqrt( x )`

Gibt die Quadratwurzel von x zurück.

```
>>> math.sqrt(2)
1.414214
>>>
```

`math.trunc( x )`

ergibt eine ganze Zahl, wobei x gegen 0 gerundet wird.

```
>>> math.trunc(3.14)
3
>>> math.trunc(-3.14)
-3
>>>
```

Ausserdem die Konstanten:

`math.e`

Die Basis der natürlichen Logarithmen.

`math.pi`

Die Kreiszahl PI, mit begrenzter Genauigkeit.

```
>>> math.pi
3.141593
>>>
```

Rechnen mit float führt manchmal zu erstaunlichen Ergebnissen.

Da `float` nur eine begrenzte Auflösung bietet, kommt es gelegentlich zu Rundungsfehlern.

```
>>> 10.99999
10.99999
>>> 10.999999
11.0
```

> Wenn insgesamt mehr als 7 Stellen vorhanden sind wird die Fließkommazahl gerundet!

Der Übergang zum Aufrunden liegt aber bei 0.0000006:

```
>>> 9.99999959     # 9 Stellen
9.999999
>>> 9.9999996      # 8 Stellen
10.0
>>>

# Noch präzieser:
>>> 9.99999959999   # 12 Stellen
9.999999
>>> 9.999999599991  # 13 Stellen
10.0
>>>
```

So wie es aussieht wird intern mit 12 Stellen gerechnet und dann auf 7 Stellen gerundet.

Ich habe nur sehr selten typische Rundungsfehler in MP gefunden. Die üblichen Beispiele aus dem Internet funktionierten hier nicht.

Eine Abhandlung dazu für Python 3.3 ist hier zu finden: https://py-tutorial-de.readthedocs.io/de/python-3.3/floatingpoint.html .

Floats vergleichen.

Aufgrund der Rundungsfehler ist ein direkter Vergleich von floats problematisch. Es wird deshalb empfohlen stattdessen die Abweichung der floats zu ermitteln und zu testen, ob diese geringer als ein zugestandener Fehler ist: `abs(x-y) < eps` .

Man kann auch eine entsprechende Funktion schreiben. Hier ein Beispiel in C:

```
bool almostEqual (float a, float b)
{
    return fabs(a - b) <= FLT_EPSILON;
}
```

nan

Nan steht für Not a Number, also keine Zahl und existiert nur für floats. Es Bedeutet, das hier kein Wert zur Verfügung steht.

```
>>> n = float('nan')
>>> n
nan
>>> n = float('Nan')
>>> n
nan
>>> n = float('NaN')
>>> n
nan
>>> n = float('NAN')
>>> n
nan
>>>
```

Mit der folgenden Funktion aus dem Modul math kann darauf getestet werden:

`math.isnan( x )`

Gibt True zurück, wenn x nicht eine Zahl ist

```
>>> a = 42
>>> n = float('nan')
>>> math.isnan(a)
False
>>> math.isnan(n)
True
>>>
```

> ‘==’ funktioniert nicht bei nan!

```
>>> n = float('nan')
>>> m = float('nan')
>>> n == m
False
>>> n is m
False
>>> id(n)
1073493184
>>> id(m)
1073493216
>>>
```

Nan wird verwendet wenn ...

str

Day 6 — String Methods in Python

Exploring different string methods in Python along with examples.

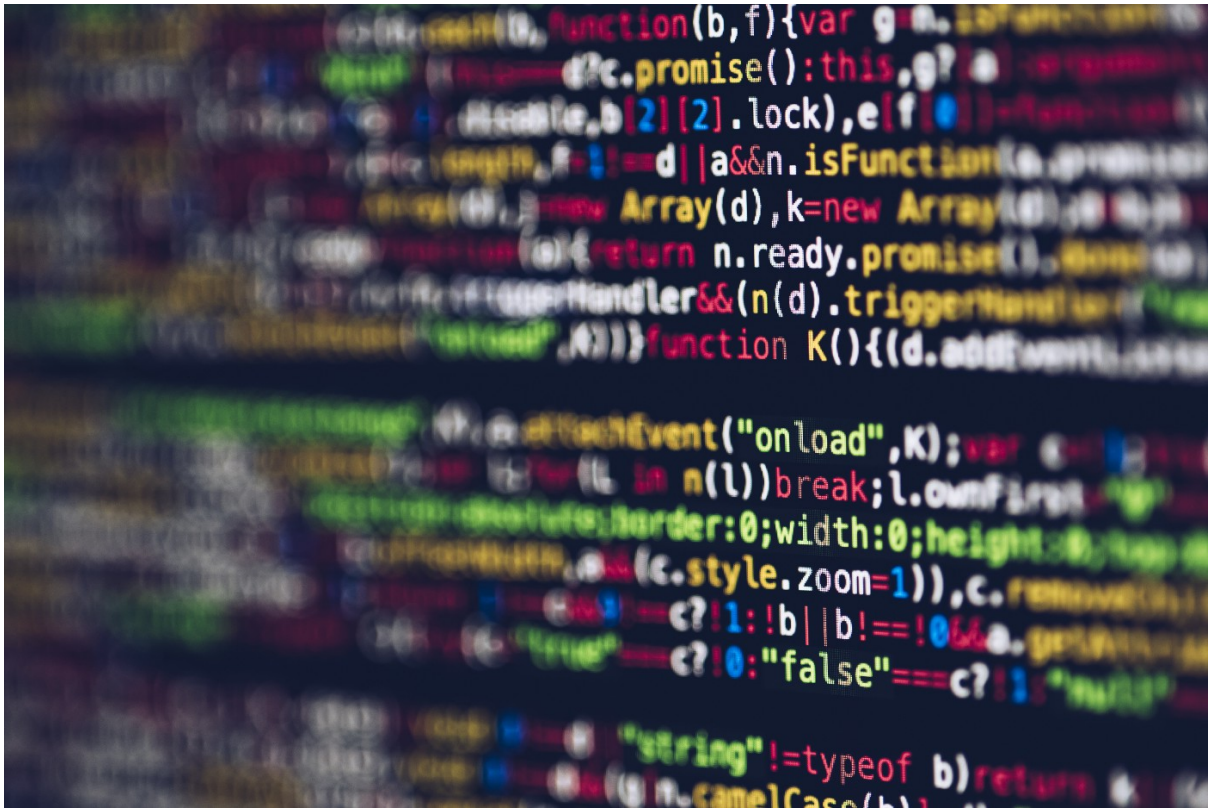

Photo by Markus Spiske on Unsplash

Hello readers!

We explored the string DataType and some string methods (like .string(), .len()) in our previous articles, available here. In this article, we will be exploring several other String Methods with their examples.

Exploring String Methods

All
the built-in string methods of Python work around the string datatype.

They don’t modify the original strings but instead return modified string objects/values. Note that, we will often be using the word substring in this article, which is basically a string, but derived/part of a main/parent string. For instance: - String: “Hello, I am a String.”- SubStrings: “ing”, “Hello”, “I am”, “am a S”, etc.

a) capitalize()In this method, the first character of the string is converted to the upper case (A-Z). In other words, if the first character of the string is lower case(a-z), it will be converted to upper case, else remain as it is.

```
string = 'this is a sample string.'
newString = string.capitalize()
print(newString)
# This is a sample string.
```

b) casefold()This method converts all the characters of the string to lower case.

```
string = 'THIs IS A SAMPLE string.'
newString = string.casefold()
print(newString)
# this is a sample string.
```

c) count()This method takes a substring as an input parameter and returns the number of occurrences of the substring in the main string.

```
subString = 'is'
string = 'This is a sample string.'
print(string.count(subString))
# 2
```

d) encode()This method returns the encoded version of the string, wherein the string is stored in byte format.

```
string = 'This is a sample string.'
print(string.encode())
# b'This is a sample string.'
```

e) endswith()This is an interesting one. If we want to check if a particular string has some specific characters (substring) at the end, this method can be useful. Note that, we get a binary response in this case, either — True (main string ends with the substring) or False.

```
subString = 'is'
string = 'This is a sample string.'
print(string.endswith(subString))
# FalsesubString = 'ng.'
string = 'This is a sample string.'
print(string.endswith(subString))
# True
```

f) find()This is a very useful method for those who normally rely on in-built string functions. In this method, a substring is searched in the main string, and its location in the string (where it is found) is returned.

```
subString = 'is'
string = 'Hello, this is a sample string.'
print(string.find(subString))
# 9
```

Note
that, if multiple instances of a substring are present in the string, the location of the very first instance is returned like in the above example — “is” occurs at the (9,10) as well as (12,13) positions, but we get the start position of the first occurrence.

g) index()This method is very similar to the Find Method as it finds the first occurrence of the specified value — substring. But there is one major difference. In *find()*, the compiler returns *-1* as the output, if not found. On the other hand, *index() raises an exception*. The following example illustrates the same.

```
subString = 'and'
string = 'Hello, this is a sample string.'
print(string.index(subString))
# ValueError: substring not foundsubString = 'and'
string = 'Hello, this is a sample string.'
print(string.find(subString))
# -1
```

h) isalnum()This method returns a T/F depending upon if all the characters of the string are alphanumeric or not. In the below example, string1 has only alphanumeric characters, hence the method returns True, while string2 has whitespaces between words leading to False.

```
string1 = 'Thisisasamplestring1'
print(string1.isalnum())
# Truestring2 = 'This is a sample string 2'
print(string2.isalnum())
# False
```

i) isalpha()This method returns a T/F based on if all the characters of the string are alphabets or not. String1 below comprises all the characters whereas string2 comprises numerical characters as well, hence we get the outputs as True and False respectively.

```
string1 = 'Thisisasamplestring'
print(string1.isalpha())
# Truestring2 = 'Thisisasamplestring2'
print(string2.isalpha())
# False
```

j) isdigit()Like *isalpha()*, isdigit() returns a T/F based on if all the characters of the string are digits or not.

```
string1 = '123456789'
print(string1.isdigit())
# Truestring2 = 'This2'
print(string2.isdigit())
# False
```

k) isidentifier()This method only returns True when the strings are identifiers as discussed here. It accepts an underscore, alphabets as well as digits.

```
string1 = '_sample_string_'
print(string1.isidentifier())
# Truestring2 = '!@sample*^&string'
print(string2.isidentifier())
# False
```

l) islower()This method returns True when all the string characters are in lowercase, else False.

```
string1 = '_sample_string_'
print(string1.islower())
# Truestring2 = 'This is a sample string.'
print(string2.islower())
# False
```

m) isupper()As opposed to *islower()*, this method returns True when all the string characters are in uppercase, else False.

```
string1 = 'THIS IS A SAMPLESTRING'
print(string1.isupper())
# Truestring2 = 'This is a sample string.'
print(string2.isupper())
# False
```

Note:
String methods starting with is — isalpha, isdigit, etc. are essentially testing methods and are bound to return a binary value, either a True or a False.

n) join()A yet essential and super handy string method, wherein all the elements of iterable — tuples, arrays, lists, etc. can be converted into strings, by adding spaces/special characters in between.

```
myList = ['This','is','a','sample','string']
newStr = ' '.join(myList)
print(newStr)
# This is a sample stringmyList = ['This','is','a','sample','string']
newStr = '#'.join(myList)
print(newStr)
# This#is#a#sample#string
```

o) lower()This method converts all the characters into lowercase.

```
string = 'This Is A Sample String.'
print(string.lower())
# this is a sample string.
```

p) lstrip()This method removes all the leading characters, wherein whitespaces are removed by default.

```
string = 'his Is A Sample String.'
print(string.lstrip('his'))
 # Is A Sample String.string = '      This Is A Sample String.'
print(string.lstrip())
# This Is A Sample String.
```

q) replace()This method returns a new string by replacing specific characters of the string with a new set of specified characters.

```
substring = '--'
string = 'This is a sample string.'
print(string.replace('is',substring))
# Th-- -- a sample string
```

r) rstrip()As opposed to *lstrip()*, *rstrip()* removes all the trailing characters. Again here, whitespaces are removed by default.

```
string = 'This Is A Sample String...........'
print(string.rstrip('..'))
# This Is A Sample Stringstring = 'This Is A Sample String          '
print(string.rstrip())
# This Is A Sample String
```

r) split()This method is the opposite of the *join()*, wherein a given string is split by a character and returns an iterable as a result.

```
string = 'This is a sample string.'
print(string.split(' '))
# ['This', 'is', 'a', 'sample', 'string.']
```

s) splitlines()This method splits the string at *line breaks ('\n')* by default and returns the lines in the form of a list. Note that, it is optional to include the '\n' character in the split words, which can be modified using the True or False parameter passed on to the splitlines() method. Clearly, False is the default wherein we don't require the line break character.

```
string = 'This is a sample string.\nThis is string2.\nThis is string3.'
print(string.splitlines())
# ['This is a sample string.', 'This is string2.', 'This is string3.']string = 'This is a sample string.\nThis is string2.\nThis is
string3.'
print(string.splitlines(True))
# ['This is a sample string.\n', 'This is string2.\n', 'This is string3.']
```

t) startswith()Contrary to endswith(), if we want to check if a particular string has some specific characters (substring) at the start, this method can be useful. Note that here as well, we get a binary response in this case, either — True (main string starts with the substring) or False.

```
subString = 'Th'
string = 'This is a sample string.'
print(string.startswith(subString))
# TruesubString = 'ng.'
string = 'This is a sample string.'
print(string.startswith(subString))
# False
```

u) strip()This method is a union/combination of *lstrip()* and *rstrip()* methods, wherein any leading, as well as trailing characters, are eliminated. Note that whitespaces are removed by default.

```
string = '..........This Is A Sample String............'
print(string.strip('......'))
# This Is A Sample Stringstring = '       This Is A Sample String        '
print(string.strip())
# This Is A Sample String
```

v) swapcase()This method swaps the case of all the characters of the string, i.e. lower ones are converted to uppers and uppers to lowers.

```
string = 'This Is A Sample String.'
print(string.swapcase())
# tHIS iS a sAMPLE sTRING.string = 'tHIS iS a sAMPLE sTRING.'
print(string.swapcase())
# This Is A Sample String.
```

w) upper()This method converts all the characters into lowercase.

```
string = 'This Is A Sample String.'
print(string.upper())
# THIS IS A SAMPLE STRING.
```

In this article, we have covered 80+% of the in-built string methods ensuring knowing and building an understanding of the essential ones.

Thanks for reading! Do subscribe and share with everyone in your community and

don’t forget to like and comment. Follow my Medium Page Rishika Gupta. You can find the entire Python programming Series or Python List of my articles in my stories. Please find the links below.

Casting

Slicing (gehört zu Datenstrukturen)

Links:

https://medium.com/@tuenguyends/python-typing-systems-444b54371f9c