Strings

Strings

Bekannt:

- Notation für einzeilige Strings
- Zeilenumbrüche mit dreifachen Anführungszeichen

```
1 text = """ Zeile 1
2 Zeile 2
3 Zeile 3"""


>>> text
' Zeile 1\nZeile 2\nZeile 3'
```

Strings

Automatisches Verschmelzen

von Literalen, die nur von Leerzeichen getrennt

```
>>> string = "Erster Teil" "Zweiter Teil"
>>> string
'Erster TeilZweiter Teil'

>>> a = ("Stellen Sie sich einen schrecklich "
...  "komplizierten String vor, den man "
...  "auf keinen Fall in eine Zeile schreiben "
...  "kann.")
>>> a
'Stellen Sie sich einen schrecklich komplizierten String vor,
den man auf keinen
Fall in eine Zeile schreiben kann.'
```

Strings

Was ist der Defaultwert von end in print()?

```
1
2 def print(*strings, sep=",", end="??")
```

Strings

Was ist der Defaultwert von end in print()?

```
1
2 def print(*strings, sep=",", end="??")
```

```
1
2 def print(*strings, sep=",", end="\n")
```

sog. *Steuerzeichen* sind unsichtbar (bei print()),
steuern die Darstellung

Strings

Steuerzeichen

Escape-Sequenz	Wirkung
\n	neue Zeile
\t	Tabulator
\a	Piepton
\"	Anführunszeichen
\\	Backslash
\r	Wagenrücklauf, geht an den Anfang der Zeile zurück

weitere s. Tabelle 12.9

Strings

Schreiben Sie eine Programm,

- das die Zahlen von 1 bis 20_000 ausgibt und dabei die jeweils vorherige überschreibt.

  Geben Sie nach Ausgabe aller Zahlen noch einen Piepton aus.

Strings

Wollen wir Steuerzeichen ausgeben,

können wir *raw Strings* verwenden, Präfix "r" oder "R"

```
>>> "String mit \t Steuerzeichen wie \\n .\n"
'String mit \t Steuerzeichen wie \\n .\n'

>>> print("String mit \t Steuerzeichen wie \\n .\n")
String mit   Steuerzeichen wie \n .
```

Strings

Wollen wir Steuerzeichen ausgeben,

können wir *raw Strings* verwenden, Präfix "r" oder "R"

```
>>> "String mit \t Steuerzeichen wie \\n .\n"
'String mit \t Steuerzeichen wie \\n .\n'

>>> print("String mit \t Steuerzeichen wie \\n .\n")
String mit   Steuerzeichen wie \n .

>>> r"String mit \t Steuerzeichen wie \\n .\n"
'String mit \\t Steuerzeichen wie \\\\n .\\n'

>>> print(r"String mit \t Steuerzeichen wie \\n .\n")
String mit \t Steuerzeichen wie \\n .\n
```

Strings

String-Methoden

- sehr viele
- Kapitel 12.5.2
- nach und nach in Übungsaufgaben

Strings

Formatierungen über Steuerzeichen hinaus

- Kommazahlen
- Eingaben variabler Breite in Tabelle
- …

Methode format()

Strings

Platzhalter

- in geschweiften Klammern
- fortlaufend nummeriert, Start Null

```
>>> "Es ist {0}.{1} Uhr".format(11, 20)
'Es ist 11.20 Uhr'
```

Strings

Platzhalter

- in geschweiften Klammern
- fortlaufend nummeriert, Start Null

```
>>> "Es ist {0}.{1} Uhr".format(11, 20)
'Es ist 11.20 Uhr'

>>> "Es ist {}.{} Uhr".format(11, 20)
'Es ist 11.20 Uhr'
```

- bei leeren Klammern Nummerierung implizit

Strings

Platzhalter

- Verwendung unabhängig von Nummerierungsreihenfolge möglich

```
>>> "Es ist {1}.{0} Uhr".format(11, 20)
'Es ist 20.11 Uhr'

>>> "Es ist {1}.{1} Uhr".format(11, 20)
'Es ist 20.20 Uhr'
```

Strings

Platzhalter

- können benamt werden

```
>>> "Es ist {stunden}.{minuten} Uhr".format(stunden=11, minuten=20)
'Es ist 11.20 Uhr'
```

Strings

Platzhalter

- geschweifte Klammern im String darstellen
- doppelt

```
>>> "Menge {{ {0} }} mit Element {0}".format("A")
'Menge { A } mit Element A'
```

Strings

Platzhalter

- geschweifte Klammern im String
- doppelt

```
>>> "Menge {{ {0} }} mit Element {0}".format("A")
'Menge { A } mit Element A'

>>> d="A"
>>> "Menge {{ {0} }} mit Element {0}".format(d)
'Menge { A } mit Element A'
```

alles auch mit Variablen statt Literalen

Strings

f-Strings

- geschweifte Klammern im String
- Referenzierung von Variablen aus Umgebung

```
>>> preis = 12.21

>>> f"Das kostet {preis} €."
'Das kostet 12.21 €.'
```

Strings

f-Strings

- Ausdruck in geschweifte Klammern wird ausgewertet

```
>>> preis = 12.21

>>> f"Das kostet {preis} €."
'Das kostet 12.21 €.'

>>> f"Das kostet {preis*1.19} €."
'Das kostet 14.5299 €.'
```

Strings

f-Strings

- Syntax

```
f’..<text>..{ <ausdruck> <!konvertierung> : <formatierung> }..<text>..’
```

Strings

f-Strings

- Syntax

```
f’..<text>..{ <ausdruck> <!konvertierung> : <formatierung> }..<text>..’
```

Ausdruck: wie sonst im Programm

Strings

f-Strings

- Syntax

```
f’..<text>..{ <ausdruck> <!konvertierung> : <formatierung> }..<text>..’
```

Konvertierung: s – str(), r – repr(), a - ascii()

Strings

f-Strings

- Konvertierung

```
1  expr = " Dies ist \u26f5"
2  print(f"{expr!s}")
3  print(f"{expr!r}")
4  print(f"{expr!a}")
```

```
 Dies ist ⛵
' Dies ist ⛵'
' Dies ist \u26f5'
```

Strings

f-Strings

- Syntax

```
f’..<text>..{ <ausdruck> <!konvertierung> : <formatierung> }..<text>..’
```

Formatierung: für Zahlen

Strings

f-Strings

- Formatierung
- .3f: 3 Stellen nach Komma

```
1 print(f"{22.2070 : .3f}")
2 print(f"{22.2070 : .2f}")
3 print(f"{22.2070 : .5f}")

 22.207
 22.21
 22.20700
```

Strings

f-Strings

- Formatierung
- 03d: führende Nullen bis Länge 3

```python
print(f"{1:03d}")
print(f"{11:03d}")
print(f"{111:03d}")
```

```
001
011
111
```

Strings

f-Strings

- Formatierung
- ^15: zentriert

  auf Breite 15

```
1 print(f"{'abc':^15}")
2 print(f"{'a':^15}")
3 print(f"{'abcdefg':^15}")
```

```
    abc
     a
  abcdefg
```

Strings

f-Strings

- mehr Details:

  https://www.pythonmorsels.com/string-formatting/

Strings

Schreiben Sie eine Funktion preci(), die

- eine Ganzzahl n und eine Gleitkommazahl x als Argumente nimmt und
- den String "Mit n Stellen: x" ausgibt, wobei
  - n durch seinen Wert ersetzt ist
  - der Wert von x mit n Stellen Präzision ausgegeben wird

```
Mit 3 Stellen hinter dem Komma: 34.555

Mit 10 Stellen hinter dem Komma: 34.5554444000
```

Strings

Operatoren für Strings:

- (Kon-)Katenation

```
>>> "34" + "35"
'3435'

>>> "34" + 35
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Strings

Operatoren für Strings:

- (Kon-)Katenation

```
>>> "34" + "35"
'3435'

>>> "34" + 35
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str

>>> 34 + 35.0
69.0
>>> 34.0 + 35
69.0
```

Strings

Operatoren für Strings:

- Multiplikation

```
>>> "T" * 7
'TTTTTTT'
```

Strings

Operatoren für Strings:

- Multiplikation

```
>>> "T" * 7
'TTTTTTT'

>>> "T" * 7.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'float'
```

Strings

Operatoren für Strings:

- Multiplikation

```
>>> "T" * 7
'TTTTTTT'

>>> "T" * 7.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'float'

>>> "T" * False
''
```

Strings

split() teilt einen String an allen Leerzeichen

```
>>> "Dieser Satz hat fünf Wörter.".split()

['Dieser', 'Satz', 'hat', 'fünf', 'Wörter.']
```

Ergebnis: Liste

Strings

split() optional Trenner als Argument

```
>>> "Dieser Satz hat fünf Wörter.".split()
['Dieser', 'Satz', 'hat', 'fünf', 'Wörter.']
>>> "Dieser Satz hat fünf Wörter.".split('e')
['Di', 's', 'r Satz hat fünf Wört', 'r.']
>>> "Dieser Satz hat fünf Wörter.".split('er')
['Dies', ' Satz hat fünf Wört', '.']
```

Strings

split() bei direkt aufeinanderfolgenden Trennern

Strings

split() bei direkt aufeinanderfolgenden Trennern

```
>>> "aaa   b   c".split()
['aaa', 'b', 'c']

>>> "aaa...b...c".split(".")
['aaa', '', '', 'b', '', '', 'c']
```

`str.`**`split`**`(`*`sep=None, maxsplit=-1`*`)`

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters as a single delimiter (to split with multiple delimiters, use `re.split()`). Splitting an empty string with a specified separator returns `['']`.

For example:

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
>>> '1<>2<>3<4'.split('<>')
['1', '2', '3<4']
```

If *sep* is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns `[]`.

For example:

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> '   1   2   3   '.split()
['1', '2', '3']
```

Strings

Die Funktion rsplit() tut dasselbe wie split() mit dem Unterschied, dass sie von rechts nach links arbeitet. Dieselben Argumente sind möglich.

- Gibt es Strings, die mit denselben Argumenten unterschiedlich gesplittet werden?
- Gibt es solche Strings bei leerer Argumentliste?

Strings

Vergleiche für Strings:

```
>>> "a" < "b"
True

>>> "b" < "a"
False

>>> "b" < "A"
False

>>> "B" < "a"
True
```

Strings

Vergleiche für Strings:

```
>>> "AA" < "A"
False

>>> "AA" < "AAA"
True

>>> "777" < "A"
True

>>> "AAAA" < "777"
False
```

Strings

Vergleiche für Strings:

Lexikographische Ordnung:

- aufgrund ersten Buchstabens
- wenn gleich, aufgrund des nächsten…
- wenn einer Präfix des anderen, dann Kürzerer kleiner

Strings

Vergleiche für Strings:

Ordnung auf Zeichen:

- Nummer im Unicode