Empfohlene Unterrichtsinhalte als Vorbereitung für dieses Thema: 01-02 (Was ist Python?), 02-01 (Variablen)

Einführung in die computerlinguistische Programmierung mit Python

02-02: Strings 💬

Im Kontext von Variablen haben wir besprochen, dass der Inhalt einer Variable immer einen bestimmten **Typ** hat. Wir haben unter anderem Typen für Texte, Zahlen und Listen gesehen. Der Typ einer Variable bestimmt, welche **Operationen** wir auf den Wert anwenden können: Zum Beispiel macht es Sinn, Zahlen zu multiplizieren, aber das Multiplizieren von Texten ist nicht so sinnvoll.

Hier beschäftigen wir uns zunächst mit dem Typ **String**. Dieser Datentyp ist für alle Daten zuständig, die die Form von Texten, Wörtern oder Buchstaben haben. Wir können übrigens jeden Wert in Python zu einem String umwandeln:

In [ ]:

```
print(type(345))
print(type(str(345)))
```

Operationen mit Strings

Python stellt uns für jeden Datentyp eine Reihe von Operationen zur Verfügung, die wir auf Werte dieses Typs anwenden können. Wir sprechen von **Methoden**.

Um eine Methode auf einen einzelnen String anzuwenden, schreiben wir beispielsweise:

In [ ]:

```
print("Hallo Welt".split())
```

Am Anfang steht der Originalstring (oder eine Variable, deren Inhalt ein String ist). Nach dem Punkt folgt der Name der Operation, die wir ausführen wollen. Das ist in diesem Fall `split`. Die öffnenden und schließenden Klammern sind dazu da, die gewünschte Operation noch genauer zu beschreiben. Diese zusätzlichen Bedingungen in den Klammern werden **Parameter** genannt.

Hier also die allgemeine Form für solche Stringoperationen:

```
<string>.<operation>(<parameter1>, <parameter2 usw.>)
```

Jede Operation hat einen **Rückgabetyp**. Zum Beispiel ist die Rückgabe (das Ergebnis) von `"Hallo Welt".split()` eine Liste der Strings, die sich ergeben, wenn man den ursprünglichen String an allen Leerzeichen aufteilt:

In [ ]:

```
# Ausgabe: Typ von "Hallo Welt"
print(type("Hallo Welt"))

# Ausgabe: Typ, den das Ergebnis der Aufteilung von "Hallo Welt" hat
print(type("Hallo Welt".split()))
```

Wenn wir eine Methode auf einen String anwenden und einen Wert als Ergebnis erhalten, kann dieser Wert direkt wieder in einer Variable gespeichert werden. Dadurch vermeiden wir verschachtelte Befehle wie im letzten Code-Beispiel.

Bei Code wie im folgenden Kästchen wertet der Interpreter zunächst aus, was auf der rechten Seite des Zuweisungsoperators steht. Sobald ein Ergebnis bekannt ist, wird dieser Wert in die Variable geschrieben.

Übrigens wird nicht der ursprüngliche String verändert, auf den wir die Methode angewendet haben, sondern ein **neuer Wert** erzeugt. Wir können also weiter auf den ursprünglichen String zugreifen.

In [ ]:

```
gruss = "Hallo Welt"
print(gruss)

gruss_als_liste = gruss.split()
print(gruss_als_liste)

# Und jetzt wieder den ursprünglichen
# String ausgeben...
print(gruss)
```

Die optionalen Parameter für die Methode `split()` (https://docs.python.org/3.6/library/stdtypes.html#str.split) heißen `sep` und `maxsplit`. Damit können wir angeben, an welchem *Substring* der ursprüngliche String aufgeteilt werden soll und wie viele Teilungen erfolgen sollen. Wir könnten zum Beispiel einen String, der ein Datum enthält, an allen Punkten aufteilen:

In [ ]:

```
print("10.11.2020".split(sep="."))
```

Oder wir wollen den folgenden String am ersten `"-"` aufteilen, die anderen Vorkommen dieses Zeichens aber intakt lassen:

In [ ]:

```
print("2020-11-10".split(sep="-", maxsplit=1))
```

Die Parameter einer Methode haben eine vorgegebene Reihenfolge. In diesem Fall wird immer `sep` als erster Parameter erwartet, `maxsplit` als zweiter. Wir können den Namen der Parameter in diesem Fall weglassen, wenn wir die Parameter in der richtigen Reihenfolge einfügen:

In [ ]:

```
print("2020-11-10".split(1, "-"))
```

Das geht allerdings nur, solange kein Parameter ausgelassen wird. Für `split` ist der Parameter `sep` optional, aber eventuell möchten wir trotzdem einen Wert für `maxsplit` angeben. Dann müssen wir den Namen des Parameters angeben.

Übrigens spricht nichts dagegen, längere Substrings als Wert für `sep` zu verwenden. Wir müssen nur daran denken, dass die Substrings in der Liste, die wir als Ergebnis bekommen, nicht mehr enthalten sind:

In [ ]:

```
print("Ich kenne die Weise, ich kenne den Text".split(sep="enn"))
```

Fassen wir zusammen:

In [ ]:

```
# Alle möglichen Teilungen am Standardseparator:
print("Hallo Welt".split())

print("----------")

# Alle möglichen Teilungen am angegebenen Separator:
print("Hallo Welt".split("W"))
print("Hallo Welt".split(sep="W"))

print("----------")

# 2 Teilungen am angegebenen Separator:
print("Hallo Welt".split(sep="l", maxsplit=2))
print("Hallo Welt".split("l", 2))
print("Hallo Welt".split("l", maxsplit=2))

print("----------")

# 1 Teilung am Standardseparator:
print("Hallo Welt".split(maxsplit=1))
```

Nachdem wir gesehen haben, wie man Strings mit Methoden verarbeiten kann, hier einige wichtige Operationen, die wir auf Strings anwenden können. Die Methoden haben unterschiedliche Rückgabetypen.

Operation	Rückgabe
`s.split([sep, n])`	**Liste** aller Teilstrings von `s`, die durch den Separator voneinander getrennt sind. Wenn weder `sep` noch `n` angegeben werden, wird der String an jedem Leerzeichen geteilt. Mit `sep` kann man spezifizieren, an welchen Teilstrings der String zerteilt werden soll, und mit `n`, wieviele Zerteilungen erfolgen sollen.
`s.lower()`	**String:** Kopie von `s`, in der alle Buchstaben kleingeschrieben sind
`s.upper()`	**String:** Kopie von `s`, in der alle Buchstaben großgeschrieben sind
`s.islower()`	**Bool:** `True`, wenn alle Buchstaben in `s` kleingeschrieben sind; sonst `False`
`s.isupper()`	**Bool:** `True`, wenn alle Buchstaben in `s` großgeschrieben sind; sonst `False`
`sub in s`	**Bool:** `True`, wenn der Teilstring `sub` in `s` enthalten ist; sonst `False`. Groß- und Kleinschreibung wird unterschieden.
`len(s)`	**Integer:** Anzahl der Zeichen, die in `s` enthalten sind.
`s.count(sub)`	**Integer:** Anzahl, wie oft der Teilstring `sub` in `s` enthalten ist
`s.startswith(sub)`	**Bool:** `True`, wenn der Teilstring `sub` am Anfang von `s` steht; sonst `False`
`s.endswith(sub)`	**Bool:** `True`, wenn der Teilstring `sub` am Ende von `s` steht; sonst `False`
`s.find(sub)`	**Integer:** Position des ersten Vorkommens von `sub` in `s`; falls `sub` nicht enthalten ist, wird -1 zurückgegeben
`s.strip([chars])`	**String:** Kopie von `s` ohne alle Vorkommen der angegebenen `chars` an Beginn und Ende des Strings. Wird `chars` nicht angegeben, werden alle führenden und abschließenden Whitespaces von `s` entfernt.
`s.replace(old, new)`	**String:** Kopie von `s`, in der alle Vorkommen des Teilstrings `old` durch den String `new` ersetzt wurden.

Besonderheiten bei Strings

Da wir Anführungszeichen `""` zur Markierung von Strings verwenden, können wir innerhalb von Strings zunächst **keine Anführungszeichen** schreiben. Es gibt einige Möglichkeiten, damit umzugehen:

1. Einfache und doppelte Anführungszeichen mischen:

In [ ]:

```
print('"The time has come," the Walrus said, "to talk of many things"')
```

1. **Mehrzeilige Strings** mit drei Anführungszeichen erzeugen; einzelne Anführungszeichen innerhalb dieses Strings sind dann unproblematisch.

In [ ]:

```
total_langer_string = """
"The time has come," the Walrus said,
"to talk of many things"
"""
print(total_langer_string)
```

1. Anführungszeichen durch einen Backslash (`Alt Gr + ß`) **escapen**:

In [ ]:

```
print("\"The time has come,\" the Walrus said, \"to talk of many things\"")
```

Der Backslash hat noch mehr Funktionen. Wir können ihn verwenden, um innerhalb von Strings Zeilenumbrüche (`'\n'`) und Tabs (`'\t'`) zu markieren:

In [ ]:

```
print("If this be error and upon me proved,\nI never writ, nor no man ever loved.")
```

In [ ]:

```
print("ottos mops\tkotzt")
```

Das Zeichen `'\n'` entspricht übrigens auch solchen Zeilenumbrüchen, die wir ganz normal getippt haben. Der folgende String enthält vier Zeilen. Um ihn an jedem Zeilenumbruch zu zerteilen, rufen wir `split()` mit dem Separator `"\n"` auf:

In [ ]:

```
print("""Am Grunde der Moldau wandern die Steine
Es liegen drei Kaiser begraben in Prag.
Das Große bleibt groß nicht und klein nicht das Kleine.
Die Nacht hat zwölf Stunden, dann kommt schon der Tag.
""".split("\n"))
```

Weil der Backslash diese Sonderfunktion erfüllt, muss er übrigens auch selbst escapet werden, wenn man ihn als bloßes Zeichen in einem String verwenden möchte. Wir schreiben dazu `\\` .

In [ ]:

```
print("D:\\Studium\\01-WiSe2020\\Python")
```

Zusammenfassung

- Mit dem Befehl `str()` kann jeder Wert in Python zum Datentyp String umgewandelt werden.
- Jeder Datentyp bringt eine Reihe von Methoden mit, die für den jeweiligen Typ sinnvoll sind. Methoden haben eine vorgegebene Anzahl von Parametern, deren Wert die genaue Ausführung der Operation beschreibt.
- Werte für Parameter können entweder in der vorgegebenen Reihenfolge angegeben werden, oder durch die Verwendung des jeweiligen Parameternamens einzeln gesetzt werden.
- Jede Methode hat einen Rückgabewert, dessen Typ durch die Definition der Methode bestimmt ist.
- Besondere Zeichen können in Strings mit dem Backslash `\` escapet werden.

Weitere Themen dieser Woche

- 02-01: Variablen
- 02-03: Listen
- 02-04: Integers und Floats (Zahlen)
- 02-05: Booleans (Wahrheitswerte)