Strings & Method Chaining

by Woche 7

Ein Thema, das wir relativ lange aufgeschoben bzw. nur stiefmütterlich behandelt haben sind Strings, also Zeichenketten, also Buchstaben/Symbole wie `'Hallo Welt!'` oder `'A1_b'`. Das wollen wir jetzt nachholen.

Strings kann man in Python natürlich auch ohne Pandas verwenden und das haben wir ja auch in vergangenen Kapiteln schon getan. Allerdings werden wir ja nun häufiger "echte" Daten importieren und verarbeiten, sodass wir auch häufiger mit Strings zu tun haben werden - sei es in den Spaltennamen oder in den Zellen der Tabellen. Hier soll also endlich eine Übersicht über die wichtigsten Funktionen und Methoden gegeben werden, die Python für Strings bereitstellt, damit man den Umgang besser einordnen und nutzen kann.

Es sei vorweg genommen, dass (wie immer, aber vor allem hier) nicht erwartet wird all die folgenden Funktionen auswendig zu lernen. Vielmehr soll ein Gefühl dafür entstehen, was möglich ist, sodass man es zum gegebenen Zeitpunkt "nachschlagen" kann.

Außerdem ist dieses Kapitel relativ umfangreich, aber auch das einzige für Woche 7.

Generell

Vorweg sollte sich ins Gedächtnis gerufen werden, dass Strings in gewisser Weise auch Listen sind, nämlich Listen von Buchstaben. Das heißt, dass wir mit `len()` die Länge eines Strings bestimmen können, mit Slicing via `[]` auf einzelne Buchstaben zugreifen können und mit `for` Schleifen über die Buchstaben iterieren können.

```
x = 'Hallo'
```

```
len(x)
```

```
5
```

```
x[2:4]
```

```
'll'
```

```
for buchstabe in x:
    print(buchstabe)
```

```
H
a
l
l
o
```

Als Einstieg wird dann häufig gezeigt, dass der + Operator auch für Strings funktioniert, und zwar um sie zu verketten, anstatt wie bei Zahlen zu addieren.

```
1 + 2
```

```
3
```

```
'Ich bin' + 'Groot'
```

```
'Ich binGroot'
```

Dabei ist es nicht möglich sowohl Strings als auch Zahlen mit dem + Operator zu verketten. Stattdessen müsste in diesem Beispiel also entweder die Zahl direkt mittels Anführungszeichen als String angegeben werden, oder aber mit der Funktion `str()` in einen String umgewandelt werden.

```
alter='2'
'Ich bin '+alter
```

```
'Ich bin 2'
```

```
alter=2
'Ich bin '+alter
```

```
TypeError: can only concatenate str (not "int") to str
```

```
alter=2
'Ich bin '+str(alter)
```

```
'Ich bin 2'
```

Methoden

Es gibt viele Methoden, die auf Strings angewendet werden können. Hier sind einige der wichtigsten:

Einige betreffen die Groß- und Kleinschreibung:

```
x = 'alle Tiere sind gleich'
```

```
x.upper()
```

```
'ALLE TIERE SIND GLEICH'
```

```
x.lower()
```

```
'alle tiere sind gleich'
```

```
x.title()
```

```
'Alle Tiere Sind Gleich'
```

```
x.capitalize()
```

```
'Alle tiere sind gleich'
```

Andere erleichtern das Prüfen von Strings. So gibt `.islower()` beispielsweise `True` zurück, wenn alle Buchstaben im String klein sind, `.endswith()` prüft, ob der String mit einem bestimmten Teil endet und `.find()` gibt die Position des ersten Vorkommens eines Teilstrings zurück.

```
y = '  aber manche sind gleicher'
```

```
y.islower()
```

```
True
```

```
y.endswith('gleicher')
```

```
True
```

```
y.find('manche')
```

```
7
```

Dann gibt es beispielsweise noch `.replace()`, welche einen Teilstring durch einen anderen ersetzt, `.strip()` entfernt Leerzeichen am Anfang und Ende eines Strings und `.split()` teilt einen String anhand eines Trennzeichens in eine Liste von Strings auf.

```
y.replace('manche', 'einige')
```

```
'  aber einige sind gleicher'
```

```
y.strip()
```

```
'aber manche sind gleicher'
```

```
y.split()
```

```
['aber', 'manche', 'sind', 'gleicher']
```

Method Chaining

Eine Besonderheit von Methoden in Python ist, dass sie gechained werden können, also hintereinander geschrieben werden können. Das geht auch für Methoden, die nichts mit Strings zu tun haben, aber hier ist nun ein guter Zeitpunkt, um das sogenannte Method Chaining zu erklären. Im Grunde ist nicht mehr zu sagen außer, dass man die Methoden einfach hintereinander schreibt und das Ergebnis der ersten Methode wird dann an die zweite Methode übergeben und so weiter. Anstatt also Zwischenergebnisse zu speichern, kann die Berechnung direkt hintereinander geschrieben werden.

```
text = '  GeH nIcHT gelASS7EN in die gute Nacht.  '
```

Mit Zwischenergebnissen:

```
neu = text.replace('7', '')
neu = neu.strip()
neu = neu.capitalize()
neu = neu.replace('nacht', 'Nacht')
neu
```

```
'Geh nicht gelassen in die gute Nacht.'
```

Mit Method Chaining:

```
neu = text.replace('7', '').strip().capitalize().replace('nacht', 'Nacht')
neu
```

```
'Geh nicht gelassen in die gute Nacht.'
```

Es ist darüber hinaus sogar möglich die *Chain* in mehrere Zeilen zu schreiben, um den Code leserlicher zu halten. Das geht entweder, indem die gesamte Chain in Klammern geschrieben wird oder am Ende einer Zeile ein `\` geschrieben wird.

```
neu = (
 text
 .replace('7', '')
 .strip()
 .capitalize()
 .replace('nacht', 'Nacht')
)
neu
```

```
'Geh nicht gelassen in die gute Nacht.'
```

```
neu = text \
  .replace('7', '') \
  .strip() \
  .capitalize() \
  .replace('nacht', 'Nacht')

neu
```

```
'Geh nicht gelassen in die gute Nacht.'
```

`.format()` & f-Strings

Manchmal möchte man in Strings auch Variablenwerte einfügen. Nehmen wir an, wir müssten einen monatlichen Report zusammenstellen, der die Anzahl der verkauften Einheiten und den Umsatz enthält. Diese Werte haben wir bereits berechnet und in Variablen gespeichert. Der Bericht soll aber auch mit einem ausformulierten Satz enden. Die Vorlage für diesen Satz könnte schlicht lauten *“In diesem Monat war der Umsatz ??? und es wurden ??? Einheiten verkauft.”*, sodass wir die Platzhalter durch die Variablenwerte ersetzen müssen. Prinzipiell kennen wir jetzt schon mindestens zwei Wege, wie wir das machen könnten:

```
umsatz = 500
einheiten = 42
```

```
satz = 'In diesem Monat war der Umsatz '+str(umsatz)+'€ und es wurden
'+str(einheiten)+' Einheiten verkauft.'
satz
```

```
'In diesem Monat war der Umsatz 500€ und es wurden 42 Einheiten verkauft.'
```

```
satz = 'In diesem Monat war der Umsatz XXX€ und es wurden YYY Einheiten
verkauft.'
satz.replace('XXX', str(umsatz)).replace('YYY', str(einheiten))
```

```
'In diesem Monat war der Umsatz 500€ und es wurden 42 Einheiten verkauft.'
```

Es gibt aber auch eine elegantere Methode, nämlich die Methode `.format()`. Dabei wird der String so formatiert, dass die Platzhalter durch die Argumente der Methode ersetzt werden. Im einfachsten Fall funktioniert das so:

```
satz = 'In diesem Monat war der Umsatz {}€ und es wurden {} Einheiten
verkauft.'.format(umsatz, einheiten)
satz
```

```
'In diesem Monat war der Umsatz 500€ und es wurden 42 Einheiten verkauft.'
```

Man kann das auch noch etwas ausführlicher machen, indem man in die Platzhalter entweder Zahlen schreibt, die angeben, welches Argument an welcher Stelle eingesetzt werden soll oder aber Namen, die dann in der Methode angegeben werden. Sobald man das macht, kann auch die Reinfolge der Argumente vertauscht werden, ohne dass der String angepasst werden muss.

```
satz = 'In diesem Monat war der Umsatz {UMSATZ}€ und es wurden {EINHEITEN}
Einheiten verkauft.'
satz.format(EINHEITEN=einheiten, UMSATZ=umsatz)
```

```
'In diesem Monat war der Umsatz 500€ und es wurden 42 Einheiten verkauft.'
```

Schließlich können wir den Code aber auch einfacher und dennoch flexibel gestalten, indem wir f-Strings einführen. Diese sind seit Python 3.6 verfügbar und bieten eine noch kürzere und übersichtlichere Möglichkeit, Variablen in Strings einzufügen. Dabei wird der String mit einem `f` vorangestellt und die Variablen direkt in den String geschrieben.

```
satz = f'In diesem Monat war der Umsatz {umsatz}€ und es wurden {einheiten}
Einheiten verkauft.'
satz
```

```
'In diesem Monat war der Umsatz 500€ und es wurden 42 Einheiten verkauft.'
```

Hier nochmal eine direkte Gegenüberstellung:

```
zahl = 42
'Die Zahl ist {}'.format(zahl)
```

```
'Die Zahl ist 42'
```

```
zahl = 42
f'Die Zahl ist {zahl}'
```

```
'Die Zahl ist 42'
```

Noch mehr Möglichkeiten hat man, wenn man die Platzhalter auch noch mit Formatierungsanweisungen versieht. So könnten wir den Umsatz mit zwei Nachkommastellen ausgeben, indem wir `:.2f` in die geschweiften Klammern schreiben. (Das geht sowohl mit f-Strings, als auch mit `.format()`.)

```
f'In diesem Monat war der Umsatz {umsatz:.2f}€ und es wurden {einheiten}
Einheiten verkauft.'
```

```
'In diesem Monat war der Umsatz 500.00€ und es wurden 42 Einheiten verkauft.'
```

Dabei steht das `f` für float und die `.2` für die zwei Nachkommastellen. Es gibt noch viele weitere Formatierungsanweisungen, die z.B. hier in der Python-Dokumentation aufgelistet sind. Diese schiere Menge an Möglichkeiten mit diesen Anweisungen macht `.format()` zu einer sehr mächtigen Methode, ist aber auch überwältigend. Nicht ohne Grund heißt es in der Dokumentation ja auch “Format Specification Mini-Language”, wird also als eigene kleine Sprache betrachtet. Hier folgen zwei Beispiele und in den weiteren Resources gibt es noch mehr dazu.

`+` führt dazu, dass auch positive Zahlen mit einem Pluszeichen versehen werden. `.0` und `.3` sorgen wieder für die entsprechende Anzahl Nachkommastellen und `f` für float.

```
x, y = 42.2, -3.1
f'{x:+.0f} und {y:+.3f}'
```

```
'+42 und -3.100'
```

`>7` führt zu einem 7 Zeichen breiten String, wobei die Zeichen rechtsbündig sind und ggf. mit Leerzeichen aufgefüllt werden. `.2` führt zu zwei Nachkommastellen und `%` formatiert den Wert als Prozentzahl.

```
p = 0.256
f'{p:>7.2%}'
p = 1
f'{p:>7.2%}'
p = 0.0034
f'{p:>7.2%}'
```

```
' 25.60%'
'100.00%'
'  0.34%'
```

Pandas

Nachdem wir die Grundlagen der String-Manipulation in Python behandelt haben, stellt sich die Frage, wie wir diese Techniken auf Daten anwenden, die in Pandas DataFrames gespeichert sind. Denn ähnlich wie im Abschnitt *“Showcase: Einfacher & Schneller”* des Kapitels *“3.1 Arrays”* können all die String-Methoden oben nicht ohne weiteres auf ganze Arrays/Series/DataFrame-Spalten angewedet werden. Man müsste also wieder eine Schleife schreiben um über alle Elemente zu iterieren. Um das zu vermeiden, bietet Pandas eine Methode `.str`, die es ermöglicht, die Methoden auf ganze Spalten anzuwenden. Als Beispiel wollen wir in der Spalte `C2` des folgenden DataFrames bei sämtlichen Einträgen die Einheit, also genauer gesagt ” in m” mit `.replace()` entfernen.

```
import pandas as pd
df = pd.DataFrame({'C1': [10, 12, 12], 'C2': ['Höhe in m', 'Breite in m',
'Tiefe in m']})
df
```

```
   C1           C2
0  10    Höhe in m
1  12  Breite in m
2  12   Tiefe in m
```

Ohne `.str`

```
for i in range(len(df)):
    df.loc[i, 'C2'] = df.loc[i, 'C2'].replace(' in m', '')

df
```

```
   C1      C2
0  10    Höhe
1  12  Breite
2  12   Tiefe
```

Mit `.str`

```
df['C2'] = df['C2'].str.replace(' in m', '')

df
```

```
   C1      C2
0  10    Höhe
1  12  Breite
2  12   Tiefe
```

Pandas bietet also eine leistungsstarke Erweiterung der Standard-String-Methoden durch die `.str` Accessor-Methode. Dies ermöglicht es uns nahtlos und intuitiv String-Methoden auf die Elemente von Pandas-Serien, also ganzen Spalten, anzuwenden.

Darüber hinaus führt Pandas allerdings sogar noch zusätzliche Methoden ein, die ebenfalls häufig benötigte Funktionen erfüllen, welche aber nicht von den Standard-String-Methoden abgedeckt waren. Diese sind in der Pandas-Dokumentation aufgelistet und hier sind drei Beispiele:

```
# Auffüllen auf 12 Zeichen
df['C2'].str.pad(
  side='right',
  width=12,
  fillchar='*'
  )
```

```
0    Höhe in m***
1    Breite in m*
2    Tiefe in m**
Name: C2, dtype: object
```

```
# Enthält der String 'ei'?
df['C2'].str.contains('ei')
```

```
0    False
1     True
2    False
Name: C2, dtype: bool
```

```
# Verketten zu einem String
df['C2'].str.cat(sep='; ')
```

```
'Höhe in m; Breite in m; Tiefe in m'
```

Regex

Zu guter Letzt soll noch kurz Regex erwähnt werden. Regex steht für *Regular Expressions* und ist eine spezielle Sprache, die es ermöglicht, Muster in Strings zu finden. Regex gibt es nicht nur in Python, sondern in vielen anderen Programmiersprachen und auch in vielen Texteditoren. Regex ist sehr mächtig und umfangreich, aber auch kompliziert bzw. unübersichtlich. Es gibt viele verschiedene Symbole und Anweisungen und es ist nicht einfach, sich diese zu merken. An dieser Stelle sei lediglich zum Einen auf das Video in den weiteren Ressourcen verwiesen, das eine sehr gute Einführung in Regex gibt und zum Anderen festgehalten, dass wir in diesem Kurs Regex nicht weiter erläutern werden. Es ist aber gut zu wissen, dass es existiert, in Python via `import re` genutzt werden kann und, dass es eine sehr mächtige Methode ist, um mit Strings zu arbeiten. Speziell für Regex können LLMs wie ChatGPT extrem hilfreich sein, weil man in natürlicher Sprache beschreiben kann welche Muster man sucht, und dann zumindest einen guten Ansatz rausbekommt.

Weitere Ressourcen

- ALL 47 STRING METHODS IN PYTHON EXPLAINED
- Python Tutorial | F-Strings | (Deutsch, #19)
- F-String Magic | Awesome | Entdecke die ganze Power der F-Strings!
- 5 Useful F-String Tricks In Python
- Übersicht aller String-Methods in pandas-Dokumentation
- Regex Tutorial Deutsch - Regex einfach erklärt! (regex101, regex Tester, Python, Javascript etc.)

Übungen

Erstelle einen String mit dem Inhalt `'Python 3.8'`, ersetze dann “3.8” durch “3.9”, füge am Ende des Strings ” ist toll!” hinzu, verwandle den gesamten String in Großbuchstaben und zähle schließlich mit `.count()` wie oft der Buchstabe “T” (großgeschrieben) im finalen String vorkommt.

`'T'` kommt _ mal im finalen String vor.

In der obigen Übung könnten alle bis auf einen Schritt ohne Weiteres auch mit Method Chaining gelöst werden. Das Hinzufügen von `' ist toll!'` mittels `+` gehört allerdings nicht dazu. In folgenden zwei Beispielen wird das Problem umgangen und zwar streng genommen nicht alles in einer durchgehenden Chain gelöst, aber immerhin in einer einzelnen Zeile Code. Schau dir beide Beispiele an und versuche jeden Aspekt und den Unterschied im Ergbnis zu verstehen. Zerlege dafür ruhig den Code in Teilabschnitte und probiere aus. Es sollte auch klar werden, warum in der ersten Zeile vor `.upper()` und `.count('T')` je ein `.str` steht, in der zweiten aber nicht.

```
pd.Series(['Python 3.8'.replace('3.8', '3.9'), ' ist
toll']).str.upper().str.count('T')
pd.Series(['Python 3.8'.replace('3.8', '3.9'), ' ist
toll']).str.cat().upper().count('T')
```

- (A) Geschafft

Ergänze in folgendem Code die beiden fehlenden Teile `???`, sodass der Output dem zweiten Codeblock entspricht.

```
orangen = 5
namen = pd.Series(['Freja', 'Sofia', 'Magnus', 'Oliver', 'Emma'])

while orangen > 0:
```

```
    print(f"???, sodass noch {orangen-1} übrig sind. ???")
    orangen = orangen - 1
```

```
Von den 5 Orangen hat Emma eine gegessen, sodass noch 4 übrig sind. Das sind
noch 80%.
Von den 4 Orangen hat Oliver eine gegessen, sodass noch 3 übrig sind. Das sind
noch 60%.
Von den 3 Orangen hat Magnus eine gegessen, sodass noch 2 übrig sind. Das sind
noch 40%.
Von den 2 Orangen hat Sofia eine gegessen, sodass noch 1 übrig sind. Das sind
noch 20%.
Von den 1 Orangen hat Freja eine gegessen, sodass noch 0 übrig sind. Das sind
noch 0%.
```

- (A) Geschafft