Editor-Trick des Tages: Go to Symbol in File

Wenn Ihr Programm größer und unübersichtlicher wird, hilft dieser Shortcut Ihnen dabei, zur Definition einer beliebigen Variable in Ihrem Code zu springen. Drücken Sie `Ctrl + Shift + O` und geben Sie den Namen einer Variable in der aktuellen Datei ein. Bestätigen Sie die Auswahl mit `Enter`, springt der Cursor in die Zeile, in der die Variable zum ersten Mal erscheint.

Sie können diese Funktion auch nutzen, um einen Überblick über die Variablen in der aktuellen Datei zu bekommen. Statt einen Namen einzutippen, können Sie mit den Pfeiltasten durch die Liste der bekannten Symbole blättern und einen der Einträge mit `Enter` auswählen.

Strukturierte Formate

Wir haben in den vergangenen Wochen bereits einige Dateien zum Lesen geöffnet oder im Zuge unserer Programme selbst erstellt (z.B. Logdateien, die ein Protokoll über die einzelnen Schritte des Programms speichern).

In der Computerlinguistik verarbeiten wir tagtäglich Daten. Diese Daten sind in den abenteuerlichsten Formaten verfügbar, aber ein paar **Standards** haben sich etabliert. Diese Standards sind Dateiformate und Vorgaben zu Inhalt und Struktur von Dateien des jeweiligen Typs. Für die verbreitetsten Standards gibt es auch **Pythonmodule**, die es uns leichter machen, mit den Daten zu arbeiten.

Heute werden wir uns jeweils einige Beispiele für die unterschiedlichen Formate ansehen.

JSON (JavaScript Object Notation)

JSON-Dateien enthalten Informationen in einem Format, das den Python-Dictionarys ähnelt, mit denen wir schon so viel gearbeitet haben. Ein JSON-Objekt besteht aus folgenden Bestandteilen:

- Geschweifte Klammern am Anfang `{` und Ende `}`.
- Keys, wie wir sie aus Dictionarys kennen: Dies sind immer Strings.
- Values: Dies sind Strings, Zahlen, Listen (in JSON-Terminologie: *Arrays*) oder selbst JSON-Objekte.
- Listen werden durch eckige Klammern am Anfang `[` und Ende `]` begrenzt.
- JSON-Objekte sind, wie Python-Dictionarys, ungeordnet.
- Listen sind, wie Python-Listen, geordnet.
- Strings werden in JSON immer mit **doppelten** Anführungszeichen markiert.

JSON ist ein verbreitetes Format zum Speichern von Konfigurationen, unter anderem für unseren Editor VSCode. Wenn Sie die User Settings öffnen (Command Palette öffnen, `>open user settings`, `Enter`), sehen Sie Ihre aktuellen Einstellungen. Das sieht in etwa so aus:

```
{
  "workbench.colorTheme": "Visual Studio Light",
  "window.zoomLevel": 0,
  "editor.renderWhitespace": "all",

  "python.linting.pylintEnabled": false,
  "python.linting.pep8Enabled": true,
  "python.linting.enabled": true,

  // Configure editor settings to be overridden for [json] language.
  "[json]": {
    "editor.quickSuggestions": {
      "strings": true
    }
  },

  // Patterns used to exclude files or folders from being linted.
  "python.linting.ignorePatterns": [
    ".vscode/*.py",
    "**/site-packages/**/*.py"
  ],
}
```

Die **Einrückungen** hier haben - anders als in Python - **keine hierarchische Bedeutung** und dienen nur der besseren Lesbarkeit. In JSON werden aufeinanderfolgende Items typischerweise untereinander geschrieben - aber da die Klammern, Doppelpunkte usw. die Struktur vorgeben, könnte das JSON-Objekt oben auch stattdessen so geschrieben werden:

```
{ "workbench.colorTheme": "Visual Studio Light", "window.zoomLevel": 0, "editor.
renderWhitespace": "all", "python.linting.pylintEnabled": false, "python.lintin
g.pep8Enabled": true, "python.linting.enabled": true, "[json]": { "editor.quickS
uggestions": { "strings": true } }, "python.linting.ignorePatterns": [ ".vscode/
*.py", "**/site-packages/**/*.py" ], }
```

Was hat das mit Computerlinguistik zu tun?

Abgesehen davon, dass JSON ein verbreiteter Standard ist, ist das Format auch geeignet für die Darstellung sprachlicher Daten. Oft sind die Daten, die wir verarbeiten, hierarchisch strukturiert, was sich in JSON gut codieren lässt. Hier zum Beispiel ein Auszug aus dem Lexikoneintrag für *lawnmower* (http://api.conceptnet.io/c/en/lawnmower) aus der semantischen Datenbank ConceptNet:

```
{
    "@context": [
        "http://api.conceptnet.io/ld/conceptnet5.6/context.ld.json"
    ],
    "@id": "/c/en/lawnmower",
    "edges": [
        {
            "@id": "/a/[/r/AtLocation/,/c/en/lawnmower/,/c/en/garage/]",
            "@type": "Edge",
            "dataset": "/d/conceptnet/4/en",
            "end": {
                "@id": "/c/en/garage",
                "@type": "Node",
                "label": "the garage",
                "language": "en",
                "term": "/c/en/garage"
            },
            "license": "cc:by/4.0",
            "rel": {
                "@id": "/r/AtLocation",
                "@type": "Relation",
                "label": "AtLocation"
            },
            "sources": [
                {
                    "@id": "/and/[/s/activity/omcs/omcs1_possibly_free_text/,/s/
contributor/omcs/guru1/]",
                    "@type": "Source",
                    "activity": "/s/activity/omcs/omcs1_possibly_free_text",
                    "contributor": "/s/contributor/omcs/guru1"
                },
                {
                    "@id": "/and/[/s/activity/omcs/omcs1_possibly_free_text/,/s/
contributor/omcs/jennifer/]",
                    "@type": "Source",
                    "activity": "/s/activity/omcs/omcs1_possibly_free_text",
                    "contributor": "/s/contributor/omcs/jennifer"
                },
                {
                    "@id": "/and/[/s/activity/omcs/vote/,/s/contributor/omcs/wal
tglass/]",
                    "@type": "Source",
                    "activity": "/s/activity/omcs/vote",
                    "contributor": "/s/contributor/omcs/waltglass"
                }
            ],
            "start": {
                "@id": "/c/en/lawnmower",
                "@type": "Node",
                "label": "a lawnmower",
                "language": "en",
                "term": "/c/en/lawnmower"
```

```
            },
            "surfaceText": "*Something you find in [[the garage]] is [[a lawnmow
er]]",
            "weight": 2.82842712474619
        }
    ],
    "view": {
        "@id": "/c/en/lawnmower?offset=0&limit=20",
        "@type": "PartialCollectionView",
        "comment": "There are more results. Follow the 'nextPage' link for mor
e.",
        "firstPage": "/c/en/lawnmower?offset=0&limit=20",
        "nextPage": "/c/en/lawnmower?offset=20&limit=20",
        "paginatedProperty": "edges"
    }
}
```


Wie wir `json` mit Python verarbeiten

Für JSON-Objekte können Sie mit den Pythonmodulen `json` oder `simplejson` arbeiten. Die genaue Verwendung dieser Libraries können Sie in der jeweiligen Dokumentation nachlesen. Hier ein

In [ ]:

```
import json

input_data_as_json = """{
    "@context": [
        "http://api.conceptnet.io/ld/conceptnet5.6/context.ld.json"
    ],
    "@id": "/c/en/lawnmower",
    "edges": [
        {
            "@id": "/a/[/r/AtLocation/,/c/en/lawnmower/,/c/en/garage/]",
            "@type": "Edge",
            "dataset": "/d/conceptnet/4/en",
            "end": {
                "@id": "/c/en/garage",
                "@type": "Node",
                "label": "the garage",
                "language": "en",
                "term": "/c/en/garage"
            },
            "license": "cc:by/4.0",
            "rel": {
                "@id": "/r/AtLocation",
                "@type": "Relation",
                "label": "AtLocation"
            },
            "sources": [
                {
                    "@id": "/and/[/s/activity/omcs/omcs1_possibly_free_text/,/s/contrib
utor/omcs/guru1/]",
                    "@type": "Source",
                    "activity": "/s/activity/omcs/omcs1_possibly_free_text",
                    "contributor": "/s/contributor/omcs/guru1"
                },
                {
                    "@id": "/and/[/s/activity/omcs/omcs1_possibly_free_text/,/s/contrib
utor/omcs/jennifer/]",
                    "@type": "Source",
                    "activity": "/s/activity/omcs/omcs1_possibly_free_text",
                    "contributor": "/s/contributor/omcs/jennifer"
                },
                {
                    "@id": "/and/[/s/activity/omcs/vote/,/s/contributor/omcs/waltglas
s/]",
                    "@type": "Source",
                    "activity": "/s/activity/omcs/vote",
                    "contributor": "/s/contributor/omcs/waltglass"
                }
            ],
            "start": {
                "@id": "/c/en/lawnmower",
                "@type": "Node",
                "label": "a lawnmower",
                "language": "en",
                "term": "/c/en/lawnmower"
            },
            "surfaceText": "*Something you find in [[the garage]] is [[a lawnmower]]",
            "weight": 2.82842712474619
        }
    ],
```

```
    "view": {
        "@id": "/c/en/lawnmower?offset=0&limit=20",
        "@type": "PartialCollectionView",
        "comment": "There are more results. Follow the 'nextPage' link for more.",
        "firstPage": "/c/en/lawnmower?offset=0&limit=20",
        "nextPage": "/c/en/lawnmower?offset=20&limit=20",
        "paginatedProperty": "edges"
    }
}"""

#print(type(input_data_as_json))
#print(input_data_as_json[:200] + "\n(...)")  # nur die ersten 200 Zeichen ausgeben

#data_as_a_python_dict = json.loads(input_data_as_json)
#print(type(data_as_a_python_dict))
#print(data_as_a_python_dict)
#print("\n--------------\n")
#print(data_as_a_python_dict["edges"][0]["surfaceText"])
```

Die Pythonpakete `json` und `simplejson` erlauben es auch, JSON-Inhalte aus Dateien zu lesen (ein JSON-Objekt pro Datei). Das ist dann hilfreich, wenn eine ganze Sammlung von JSON-Objekten verarbeitet werden soll, wobei z.B. jedes Objekt einen Satz (ein Wort, ein Konzept, ein Dokument, ...) beschreibt.

Sonstiges über JSON

In VSCode können wir JSON-Code, der schwer lesbar formatiert ist, automatisch neu formatieren. Dazu kopieren wir den Code in den Editor, stellen die Sprache ein (Command Palette öffnen, `>change language`, `Enter`, `json`, `Enter`), und wählen dann in der Command Palette `format document`.
Achtung: Die Formatierung funktioniert nur, wenn die JSON-Datei ein einziges JSON-Objekt enthält.

CSV (Comma-Separated Values) und TSV (Tab-Separated Values)

Wenn die Informationen, die wir repräsentieren wollen, nicht hierarchisch verschachtelt sind (wie es bei JSON möglich ist), können wir stattdessen ein **tabellenähnliches** Format wählen.

Solche Formate bieten sich immer an, wenn wir über jeden Eintrag (jede Zeile der Tabelle) die gleichen Informationen angeben wollen.

Eine TSV-Datei enthält dann die Informationen für die einzelnen Felder sequentiell aneinandergereiht, mit einem Tab zwischen je zwei Feldern. In Textform sieht das so ähnlich aus wie eine Tabelle ohne Ränder. Hier ein Auszug aus dem NRC Emotion Lexicon (http://saifmohammad.com/WebPages/AccessResource.htm), in dem jedem Wort ein Intensitätswert für bestimmte Emotionen zugeordnet wird:

```
tsv
term    score   AffectDimension
outraged    0.964   anger
brutality   0.959   anger
hatred  0.953   anger
forgotten   0.266   fear
crouch  0.266   fear
flurries    0.266   fear
shanghai    0.266   fear
insomnia    0.266   fear
sneak   0.266   fear
wonderful   0.863   joy
ilovechristmas  0.859   joy
hooray  0.859   joy
brilliant   0.859   joy
cheering    0.859   joy
glory   0.859   joy
```

Für Menschen ist dieses Format nicht immer gut lesbar. Im Beispiel sind die Zeilen eher kurz, aber oft werden viele Informationen in jede Zeile geschrieben, sodass es schwer ist, den Überblick zu behalten.

Noch schwerer ist es, wenn die "Spalten" in der Datei durch Kommas oder Semikolons voneinander getrennt sind (CSV-Format). Hier ein Auszug aus einem Datensatz über beliebte Babynamen in Berlin im Jahr 2017 (http://www.berlin.de/daten/liste-der-vornamen-2017/tempelhof-schoeneberg.csv):

```
csv
vorname;anzahl;geschlecht;position
Sophie;62;w;2
Marie;61;w;2
Emilia;52;w;1
Maria;45;w;2
Charlotte;37;w;2
Noah;37;m;1
Ella;37;w;1
Alexander;35;m;2
Jonas;34;m;1
Hannah;34;w;1
Emma;31;w;1
Maximilian;30;m;2
```

Was hat das mit Computerlinguistik zu tun?

Nicht immer sind sprachliche Daten hierarchisch strukturiert, wie im JSON-Beispiel oben. Datensätze wie z.B. Wörterbücher enthalten einheitliche Informationen über einzelne Elemente: Pluralform, verschiedene Kasus usw. Für solche Informationen ist ein **tabellenartiges Format** gut geeignet.

Oft werden computerlinguistische Klassifizierungssysteme für einzelne Wörter oder Konzepte ausgeführt, wie das *Emotion Lexicon* im TSV-Beispiel oben: Für jeden Eingabewert (jedes der Wörter in der ersten Spalte) wird ein dazugehöriger Ausgabewert ermittelt, der in der zweiten und dritten Spalte abgelegt wird. So können die Ergebnisse des Systems im TSV/CSV-Format gespeichert werden.

Wie wir `csv` mit Python verarbeiten

In [ ]:

```
import csv

input_string = """vorname;anzahl;geschlecht;position
Sophie;62;w;2
Marie;61;w;2
Emilia;52;w;1
Maria;45;w;2
Charlotte;37;w;2
Noah;37;m;1
Ella;37;w;1
Alexander;35;m;2
Jonas;34;m;1
Hannah;34;w;1
Emma;31;w;1
Maximilian;30;m;2"""

#input_string_as_list = input_string.split("\n")
#print(input_string_as_list)

#print("\n--------------\n")

# Hier werden die CSV-Inhalte in ein Format gebracht, durch das wir
# danach iterieren können. Der delimiter ist das Trennzeichen
# zwischen den Spalten (hier ein Semikolon, oft ein Komma).
#namereader = csv.DictReader(input_string_as_list, delimiter=';')

#for row in namereader:
#    print("Vorname: {:15}Geschlecht: {}".format(row["vorname"], row["geschlecht"]))
```

Wenn wie in unserem Beispiel **Überschriften** enthalten sind, können wir die Zeilen nacheinander als Dictionary einlesen: Die Keys sind die Bezeichnungen der Spalten, und die dazugehörigen Values sind die Werte in jeder Spalte für die gerade aktuelle Zeile. Deshalb verwenden wir im Code oben die Spaltennamen "vorname" und "geschlecht", um auf die Werte der jeweils aktuellen Zeile zuzugreifen.

Wir können auch stattdessen mit den Zeilen als Listen arbeiten und auf die **Positionsindizes** zugreifen:

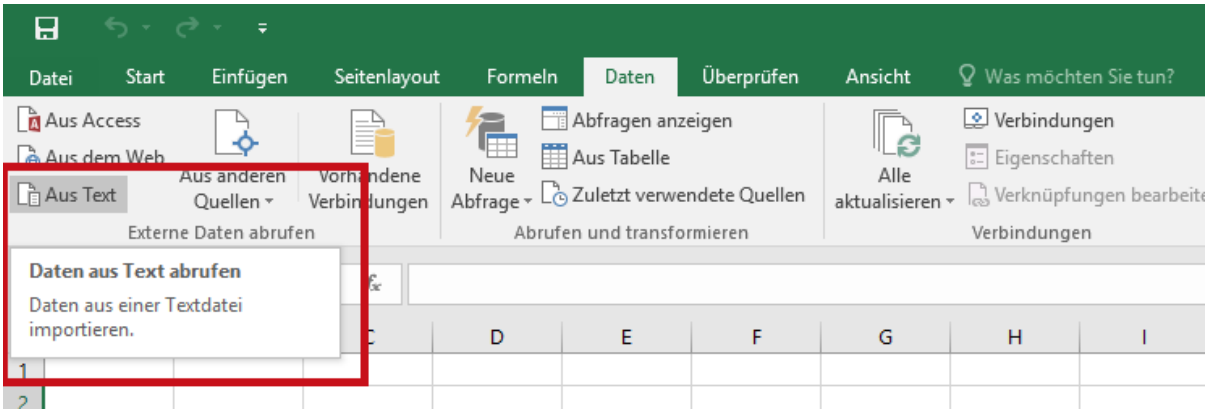
In [ ]:

```
import csv

input_string = """vorname;anzahl;geschlecht;position
Sophie;62;w;2
Marie;61;w;2
Emilia;52;w;1
Maria;45;w;2
Charlotte;37;w;2
Noah;37;m;1
Ella;37;w;1
Alexander;35;m;2
Jonas;34;m;1
Hannah;34;w;1
Emma;31;w;1
Maximilian;30;m;2"""

#input_string_as_list = input_string.split("\n")
#print(input_string_as_list)

#print("\n--------------\n")

# csv.reader() liest die Zeilen als geordnete Sequenzen, ohne
# die Werte zu den Überschriften zuzuordnen. Gut für explorative
# Analyse (wenn man noch nicht so genau weiß, welche Informationen
# enthalten sind).
#namereader = csv.reader(input_string_as_list, delimiter=';')
#for row in namereader:
#    print("Vorname: {:15}Geschlecht: {}".format(row[0], row[2]))
```

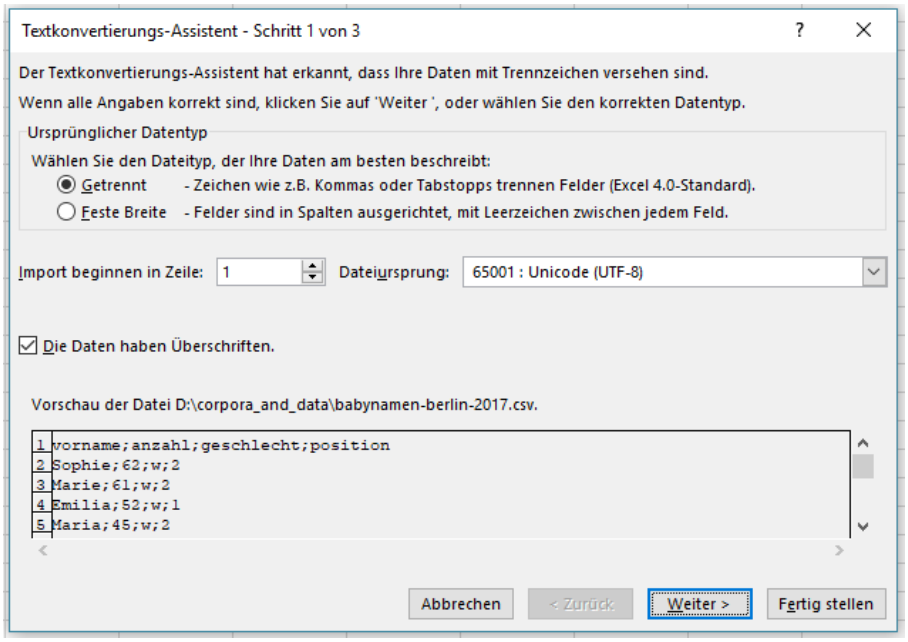
Beachten Sie, dass in diesem Fall in Zeile 22 nicht `DictReader`, sondern `reader` aufgerufen wird.

Sonstiges über CSV

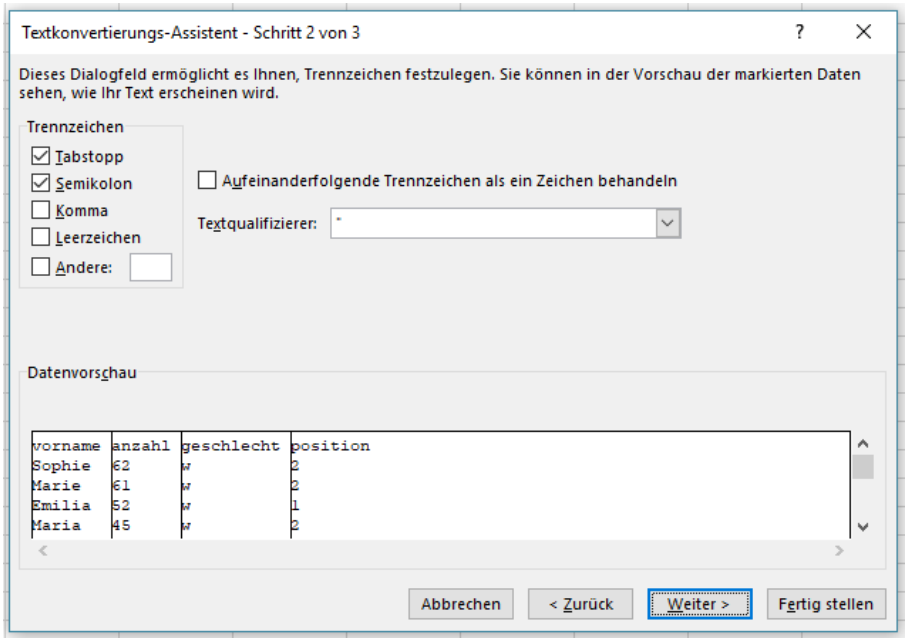
Da CSV der Form einer Tabelle ähnelt, können wir solche Dateien auch in Excel betrachten. Um das zu tun, öffnen wir in Excel zuerst eine leere Mappe und importieren dann die Daten aus der CSV-Datei:

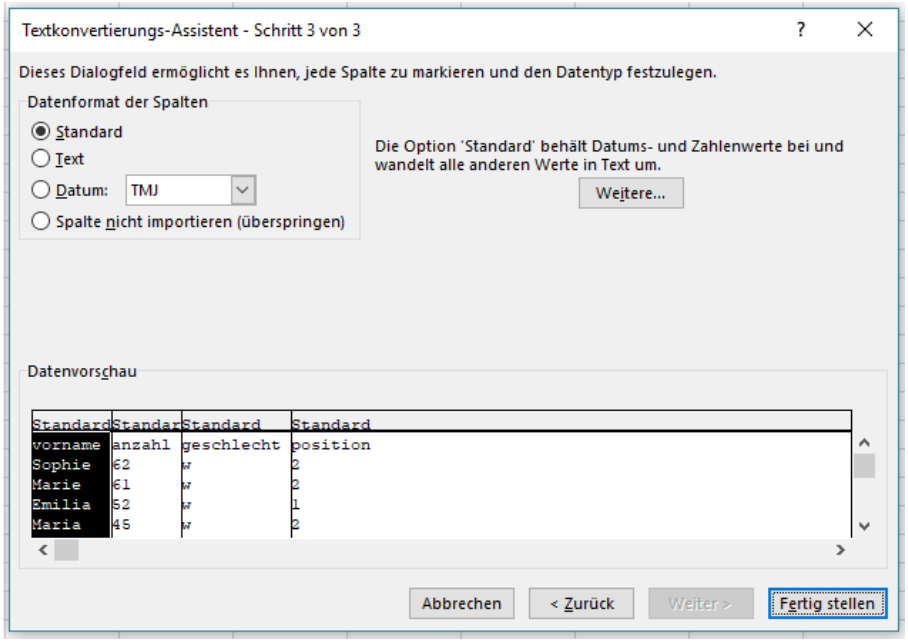


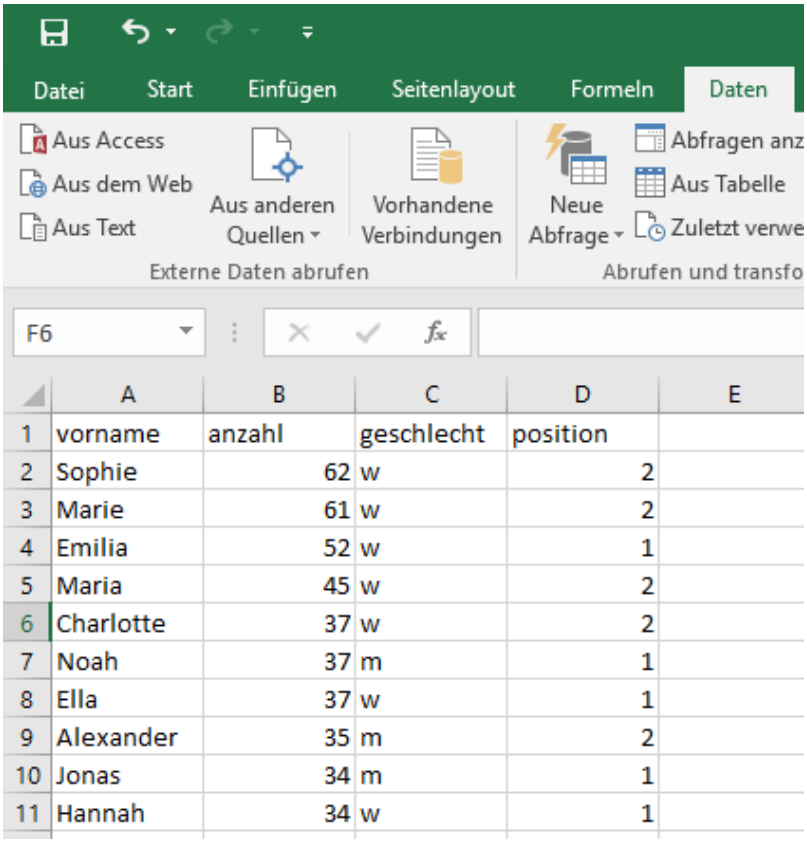
Typischerweise sollten Sie beim Import das Encoding **Unicode** auswählen, um sicherzustellen, dass Umlaute und Sonderzeichen korrekt angezeigt werden.



Als nächstes können Sie festlegen, welches Trennzeichen (im Pythoncode oben *Delimiter*) die Daten haben. Verbreitet sind Komma, Tab, Semikolon und Leerzeichen (Achtung: bei Sprachdaten oft ungeeignet, da Leerzeichen innerhalb eines Felds erlaubt sein müssen).







	A	B	C	D	E
1	vorname	anzahl	geschlecht	position	
2	Sophie	62	w	2	
3	Marie	61	w	2	
4	Emilia	52	w	1	
5	Maria	45	w	2	
6	Charlotte	37	w	2	
7	Noah	37	m	1	
8	Ella	37	w	1	
9	Alexander	35	m	2	
10	Jonas	34	m	1	
11	Hannah	34	w	1	

XML, HTML

Sowohl XML als auch HTML sind sogenannte Markup-Formate (auf deutsch: *Auszeichnungssprachen* (https://de.wikipedia.org/wiki/Auszeichnungssprache). Das bedeutet vor allem, dass wir den enthaltenen Daten **beliebig viele Annotationen auf beliebig verschachtelten Metaebenen** hinzufügen können.

Die Meta-Informationen in HTML beziehen sich auf Format, Struktur und Inhalt der Daten:

```
<span class="example">Dieser Text ist Teil eines Bespiels.</span>
<span class="example-italic">Dieser Text ist Teil eines Bespiels mit <i>kursivem
Text</i>.</span>
```

Die Meta-Informationen werden in sogenannten *Tags* angegeben. Tags bestehen aus spitzen Klammern `<>` mit bestimmten Keywords. Tags werden geschlossen, sobald das dazugehörige schließende Tag - mit einem `/` am Anfang - auftaucht.

Im Beispiel werden zwei `<span>`-Tags geöffnet und jeweils mit `</span>` wieder geschlossen. Die Zusatzinfo `class="example"` ist ein **Attribut** des Tags und muss beim schließenden Tag nicht wiederholt werden.

Überlappungen von Tags sind ausgeschlossen, aber Verschachtelungen sind möglich. Im zweiten Beispiel oben befindet sich ein `<i>`-Tag innerhalb des Gültigkeitsbereiches eines `<span>`-Tags.

Was hat das mit Computerlinguistik zu tun?

Genau wie JSON wird XML oft verwendet, wenn Daten ineinander verschachtelt sein sollen. Dabei ermöglicht XML die relativ komfortable Angabe von beschreibenden Attributen. Hier ein Beispieldatensatz aus *PropBank*, einer semantischen Datenbank für Verben:

```
<!DOCTYPE frameset SYSTEM "frameset.dtd">
<frameset>
  <predicate lemma="giggle">
    <note>
  Frames file for 'giggle' based on sentences in brown.
    </note>

    <roleset id="giggle.01" vncls="40.2" name="laugh nervously">
      <roles>
        <role n="0" descr="giggler">
          <vnrole vncls="40.2" vntheta="Agent"/>
        </role>
      </roles>

      <example name="intransitive">
        <inflection person="ns" tense="ns" aspect="ns" voice="ns" form="ns"/>
        <text>
 Mose giggled .
        </text>
        <arg n="0">Mose</arg>
```

In [ ]:

```
# import ... as ... erzeugt ein Kürzel, um
# Tipparbeit zu sparen und den Code übersichtlicher
# zu machen:
import xml.etree.ElementTree as ET

input_as_string = """<!DOCTYPE frameset SYSTEM "frameset.dtd">
<frameset>
  <predicate lemma="giggle">
    <note>
  Frames file for 'giggle' based on sentences in brown.
    </note>

    <roleset id="giggle.01" vncls="40.2" name="laugh nervously">
      <roles>
        <role n="0" descr="giggler">
          <vnrole vncls="40.2" vntheta="Agent"/>
        </role>
      </roles>

      <example name="intransitive">
        <inflection person="ns" tense="ns" aspect="ns" voice="ns" form="ns"/>
        <text>
 Mose giggled .
        </text>
        <arg n="0">Mose</arg>
        <rel>giggled</rel>
      </example>

    </roleset>
  </predicate>
</frameset>"""

#root = ET.fromstring(input_as_string)

# Die Variable root enthält jetzt einen Zeiger auf das
# Tag auf oberster Ebene (<frameset>):
#print(root)

# Das <frameset>-Tag hat keine Attribute und keinen Text
#print("Tag auf oberster Ebene: <{}>".format(root.tag))
#print("Attribute: {}".format(root.attrib))
#print("Text: {}".format(root.text))

#print("-----------\n")

#for child in root:
#    print("Tag auf dieser Ebene: <{}>".format(child.tag))
#    print("Attribute: {}".format(child.attrib))


#print("Text im <note>-Tag: {}".format(root[0][0].text))
```

Sonstiges zu HTML und XML

Obwohl es strukturelle Gemeinsamkeiten zwischen HTML und XML gibt, unterscheiden die beiden Formate sich. HTML enthält eine vorgegebene Auswahl von Tags, die verwendet werden können; für XML-Dateien kann man selbst definieren, welche Tags vorkommen und welche Inhalte sie haben dürfen.

HTML verarbeiten wir, wenn wir Webseiten *scrapen*, also die Inhalte entweder komplett extrahieren oder nach bestimmten Mustern durchsuchen. Dabei haben wir keine Garantie, dass die Daten *wohlgeformt* sind: Möglicherweise werden Tags nicht ordentlich geschlossen oder es werden Tags verwendet, die nicht vorgesehen sind.

Das XML-Paket, das wir im Beispiel oben verwendet haben, kommt mit solchen fehlerhaften Daten nicht zurecht. Stattdessen können wir für solche Inhalte das Paket `beautifulsoup` verwenden. Verwendungsbeispiele und Anleitungen dafür finden Sie in der Dokumentation (https://media.readthedocs.org/pdf/beautiful-soup-4/latest/beautiful-soup-4.pdf).

Übrigens ermöglichen sowohl XML als auch JSON hierarchisch geordnete Repräsentationen, es ist also möglich, Daten vom einen Format ins andere umzuwandeln. Dabei ist JSON etwas weniger "aufgebläht" (meistens brauchen JSON-Dateien weniger Speicherplatz als ihre XML-Variante). XML wird von einigen als lesbarer empfunden. Mehr zu den **Vor- und Nachteilen der beiden Formate im Vergleich** finden Sie z.B. auf StackOverflow (https://stackoverflow.com/questions/5615352/xml-and-json-advantages-and-disadvantages). In Python empfehlen wir Ihnen, lieber mit JSON zu arbeiten: Das Format erinnert so sehr an Python-Dictionarys und die Umwandlung von JSON zu Dictionarys geht so einfach, dass Sie kaum Aufwand haben, um Daten aus JSON-Quellen mit Python zu verarbeiten.

SQL-Datenbanken, `sqlite3`

Datenbanken sind Ansammlungen von strukturierten Informationen, die in Tabellenform abgelegt werden. Dabei sind die Tabellen untereinander durch vordefinierte Bedingungen verknüpft.

Tabelle: Wort

	WortID	Wort	POS	Morpheme
	Filtern	Filtern	Filtern	Filtern
299988	468225	Oberlawine	NN	Ober#lawine
299989	468226	Oberleder	NN	Ober#leder
299990	468227	Oberlederschere	NN	Ober#leder#s...
299991	468228	Oberleger	NN	Ober#leg#er
299992	468229	Oberlehnsherr	NN	Ober#lehnsherr
299993	468231	Oberlehrer	NN	Ober#lehrer
299994	468232	Oberlehrerexamen	NN	Ober#lehrer#...
299995	468233	Oberlehrerhabitus	NN	Ober#lehrer#...

Was hat das mit Computerlinguistik zu tun?

Linguistische Datenbanken haben in zwei Hinsichten etwas mit computerlinguistischen Anwendungen zu tun: Sie können als Input für computerlinguistische Anwendungen dienen, zum Beispiel, wenn Texte generiert werden sollen. Oder sie können das Endergebnis von Systemen zur Analyse oder Klassifizierung von Daten sein.

Wie wir `sqlite` mit Python verarbeiten

Das Pythonpaket `sqlite3` stellt Methoden zur Verfügung, mit denen wir Datenbanken lesen, aber auch schreiben können. Das sieht etwa so aus:

```
1   import sqlite3
2
3   conn = sqlite3.connect('D:/corpora_and_data/germanwordsandmorph.sqlite')
4   c = conn.cursor()
5
6   c.execute('select distinct * from Wort join Morph where Wort like "Python%"')
7   for s in c.fetchmany(20):
8       print(s)
9
10
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS D:\PhD\00-organize\philfak_website> & python d:/philcloud/ws18-python/ipynb_notebooks/using_sqlite.py
(506621, 'Python', 'NN', 'Python', 3, 'Pos_Invar')
(506621, 'Python', 'NN', 'Python', 3, 'Masc_Nom_Sg')
(506621, 'Python', 'NN', 'Python', 3, 'Masc_Nom_Pl')
(506621, 'Python', 'NN', 'Python', 3, 'Masc_Gen_Pl')
(506621, 'Python', 'NN', 'Python', 3, 'Masc_Dat_Sg')
(506621, 'Python', 'NN', 'Python', 3, 'Masc_Acc_Sg')
(506621, 'Python', 'NN', 'Python', 3, 'Masc_Acc_Pl')
(506621, 'Python', 'NN', 'Python', 4, 'Fem_Nom_Pl')
(506621, 'Python', 'NN', 'Python', 4, 'Fem_Gen_Pl')
(506621, 'Python', 'NN', 'Python', 4, 'Fem_Dat_Pl')
(506621, 'Python', 'NN', 'Python', 4, 'Fem_Acc_Pl')
(506621, 'Python', 'NN', 'Python', 5, 'Masc_Dat_Pl')
(506621, 'Python', 'NN', 'Python', 6, 'Masc_Gen_Sg')
(506621, 'Python', 'NN', 'Python', 8, 'Masc_Nom_Sg')
(506621, 'Python', 'NN', 'Python', 8, 'Masc_Dat_Sg')
(506621, 'Python', 'NN', 'Python', 8, 'Masc_Acc_Sg')
(506621, 'Python', 'NN', 'Python', 10, 'Fem_Nom_Sg')
(506621, 'Python', 'NN', 'Python', 10, 'Fem_Gen_Sg')
(506621, 'Python', 'NN', 'Python', 10, 'Fem_Dat_Sg')
(506621, 'Python', 'NN', 'Python', 10, 'Fem_Acc_Sg')
```

Da zum Lesen von Datenbanken nicht nur Python beherrscht werden muss, sondern auch die Syntax von SQL-Anfragen, werden wir uns hier nicht näher damit beschäftigen. Wenn Ihnen später im Studium SQL-Datenbanken begegnen, können Sie diese in Ihre Pythonprogramme integrieren.

Weitere Formate in der Computerlinguistik

Außer den eben besprochenen Formaten gibt es noch eine Vielzahl anderer Arten, wie man Daten strukturiert ablegen und einlesen kann. Unter anderem kann jedes Programm, das geschrieben wird, eigene Regeln definieren und bei den Dateien, mit denen gearbeitet wird, anwenden.

In solchen Fällen können Sie auf bewährte Methoden zurückgreifen, die Sie bereits in den vergangenen Wochen trainiert haben: Dateien einlesen, mit Stringmethoden oder regulären Ausdrücken in kleinere Einheiten zerlegen, und mit den entstehenden Datenstrukturen weiterarbeiten.

Eine typische Anwendungssituation in der Computerlinguistik sieht etwa so aus: Ein Text, der verarbeitet werden soll, wird zuerst mit **Preprocessing**-Tools verarbeitet. Beispielsweise werden **Part-of-speech-Tags** vergeben, um die Wortart jedes einzelnen Worts zu markieren, oder ein **Parser** analysiert die syntaktische Struktur des Satzes.

Probieren Sie es aus: Öffnen Sie den Parser unter http://nlp.stanford.edu:8080/parser/index.jsp (http://nlp.stanford.edu:8080/parser/index.jsp) und geben Sie einen englischen Satz ein. Unterschiedliche Ausgabeformate erscheinen unterhalb der Eingabe. Je nachdem, wie gewöhnlich oder ungewöhnlich Ihr eingegebener Satz war, ist die Analyse vollständig korrekt oder nur teilweise.

Wenn wir die folgende Passage eingeben und parsen...

> He's passed on! This parrot is no more! He has ceased to be! He's expired and gone to meet his maker!

... erhalten wir folgenden Output:

Your query

He's passed on! This parrot is no more! He has ceased to be! He's expired and go
ne to meet his maker!

Tagging

He/PRP 's/VBZ passed/VBN on/RP !/.
This/DT parrot/NN is/VBZ no/RB more/JJR !/.
He/PRP has/VBZ ceased/VBN to/TO be/VB !/.
He/PRP 's/VBZ expired/VBN and/CC gone/VBN to/TO meet/VB his/PRP$ maker/NN !/.

Parse

```
(ROOT
  (S
    (NP (PRP He))
    (VP (VBZ 's)
      (VP (VBN passed)
        (PRT (RP on))))
    (. !)))

(ROOT
  (S
    (NP (DT This) (NN parrot))
    (VP (VBZ is)
      (ADJP (RB no) (JJR more)))
    (. !)))

(ROOT
  (S
    (NP (PRP He))
    (VP (VBZ has)
      (VP (VBN ceased)
        (S
          (VP (TO to)
            (VP (VB be))))))
    (. !)))

(ROOT
  (S
    (NP (PRP He))
    (VP (VBZ 's)
      (VP (VBN expired)
        (CC and)
        (VBN gone)
        (S
          (VP (TO to)
```

```
            (VP (VB meet)
              (NP (PRP$ his) (NN maker)))))))
    (. !)))

-------------------------

Universal dependencies

nsubjpass(passed-3, He-1)
auxpass(passed-3, 's-2)
root(ROOT-0, passed-3)
compound:prt(passed-3, on-4)

det(parrot-2, This-1)
nsubj(more-5, parrot-2)
cop(more-5, is-3)
neg(more-5, no-4)
root(ROOT-0, more-5)

nsubj(ceased-3, He-1)
aux(ceased-3, has-2)
root(ROOT-0, ceased-3)
mark(be-5, to-4)
xcomp(ceased-3, be-5)

nsubjpass(expired-3, He-1)
auxpass(expired-3, 's-2)
root(ROOT-0, expired-3)
cc(expired-3, and-4)
conj(expired-3, gone-5)
mark(meet-7, to-6)
xcomp(expired-3, meet-7)
nmod:poss(maker-9, his-8)
dobj(meet-7, maker-9)
Universal dependencies, enhanced
nsubjpass(passed-3, He-1)
auxpass(passed-3, 's-2)
root(ROOT-0, passed-3)
compound:prt(passed-3, on-4)

det(parrot-2, This-1)
nsubj(more-5, parrot-2)
cop(more-5, is-3)
neg(more-5, no-4)
root(ROOT-0, more-5)

nsubj(ceased-3, He-1)
nsubj:xsubj(be-5, He-1)
aux(ceased-3, has-2)
root(ROOT-0, ceased-3)
mark(be-5, to-4)
xcomp(ceased-3, be-5)
```

```
nsubjpass(expired-3, He-1)
nsubjpass(gone-5, He-1)
nsubj:xsubj(meet-7, He-1)
auxpass(expired-3, 's-2)
root(ROOT-0, expired-3)
cc(expired-3, and-4)
conj:and(expired-3, gone-5)
mark(meet-7, to-6)
xcomp(expired-3, meet-7)
nmod:poss(maker-9, his-8)
dobj(meet-7, maker-9)

-------------------------

Statistics

Tokens: 27
Time: 0.054 s
Parser: englishPCFG.ser.gz
```

Die Informationen im Abschnitt `Tagging` können wir mit Stringmethoden lesen: Der Text kann zuerst mit `split()` in einzelne Wörter zerlegt werden, dann kann jedes Element mit `split("/")` in ein Paar aus Wort und POS-Tag (Wortart) zerlegt werden.

Die Informationen im Abschnitt `Parse` sind verschachtelt, mit runden Klammern zur Markierung der verschiedenen Ebenen. Zur Verarbeitung dieser Informationen können wir entweder eine eigene Funktion schreiben oder ein passendes Pythonpaket verwenden. Wegen der Verschachtelung der Daten bietet es sich an, ein Dictionary zu definieren, das mit den Informationen aus der Klammerstruktur gefüllt wird.

Die Informationen im Abschnitt `Universal dependencies` können hervorragend mit regulären Ausdrücken gelesen werden.

Aufgabe

1. Schreiben Sie den regulären Ausdruck, der die Informationen aus jeder Zeile extrahiert.

In [ ]:

```
import re

def collect_relations(relation_as_string):
    # Vor der Klammer steht der Name der Relation
    # (z.B. "nsubjpass")
    # Das erste Element enthält das Wort und den Index
    #         des ersten Bestandteils der Relation
    #         (z.B. passed-3)
    # Das zweite Element enthält das Wort und den Index
    #         des zweiten Bestandteils der Relation
    #         (z.B. He-1)

    # FIX THIS LINE
    relation_pattern = re.compile("()()()()()")

    m = re.search(relation_pattern, relation_as_string)
    relname = m.group(1)
    rel_element1 = {m.group(3):m.group(2)}
    rel_element2 = {m.group(5):m.group(4)}

    output = {"relname": relname, "element1": rel_element1, "element2": rel_element2}
    # z.B.: {"relname": "nsubjpass", "element1": {"3": "passed"}, "element2": {"1": "H
e"}}

    return output

print(collect_relations("nsubjpass(passed-3, He-1)"))
print(collect_relations("auxpass(passed-3, 's-2)"))
print(collect_relations("root(ROOT-0, passed-3)"))
print(collect_relations("compound:prt(passed-3, on-4)"))
```

Zusammenfassung

Textdateien sind ein wichtiges Werkzeug in der Computerlinguistik, weil die verschiedensten Informationen und Strukturen in Textdateien abgebildet werden können.

Wenn Ihnen eine der hier besprochenen Arten von Dateien begegnet, wissen Sie jetzt, wie diese Dateien mit Python verarbeitet werden können. Bei anderen Dateien können Sie verfahren wie im letzten Beispiel: Die Struktur der Datei zunächst selbst analysieren und dann eigene Pythonprogramme schreiben, die mit der Struktur der Datei arbeiten können, um die Informationen zu extrahieren, für die Sie sich interessieren.

Und morgen...

In der Übungssitzung morgen werden Sie üben, mit einigen der Dateiarten umzugehen, die wir hier behandelt haben. Denken Sie daran, dass Sie jederzeit in der Dokumentation nachlesen können, wie man bestimmte Dinge mit diesen Dateien macht!