Herzlichen Glückwunsch!

Sie haben den ersten Programmierkurs des Computerlinguistikstudiums erfolgreich abgeschlossen! Damit verfügen Sie jetzt über eine gute Grundlage für die weiterführenden Kurse in den nächsten Semestern.

Organisatorisches

Falls Sie im Laufe des Semesters genügend Aufgaben eingereicht haben (20 oder mehr), haben Sie sich den BN für diesen Kurs verdient. Wenn Sie in der aktuellen Prüfungsordnung studieren, stellen wir Ihnen den BN in den nächsten Wochen (bis spätestens 31. März) online aus. Wenn Sie nicht Computerlinguistik studieren oder aus irgendwelchen anderen Gründen einen Papier-BN benötigen, schreiben Sie uns bitte eine Mail oder sprechen Sie uns an. Die Papier-BNs werden dann bis Ende März im Sekretariat der allgemeinen Sprachwissenschaft hinterlegt.

Falls Sie nicht die Grenze für den BN erreicht haben, können Sie noch bis einschließlich 23. Februar Aufgaben nachreichen.

Heute werden wir einige Themen besprechen, die wir hier noch nicht behandeln konnten, die Ihnen aber vielleicht später über den Weg laufen werden...

Virtualenv (virtuelle Python-Umgebungen)

In den letzten Wochen haben wir gelegentlich neue **Python-Pakete installiert**, z.B. `NLTK`. Die Module für diese Pakete werden im Python-Installationsordner angelegt, typischerweise in einem Pfad wie `C:\Users\Student\AppData\Local\Programs\Python\Python37-32\Lib\site-packages`.

Um die Python-Umgebung auf einem Computer oder Server zu ändern, braucht man bestimmte Berechtigungen. Das ist ein Problem, wenn man auf einem Gerät arbeitet, auf dem man keinen **Admin-Zugang** hat: Man müsste jedesmal die Admins kontaktieren, um ein Paket nachzuinstallieren...

Ein zweites Problem ist, dass verschiedene Projekte möglicherweise unterschiedliche Versionen von Pythonpaketen brauchen. Von Version zu Version kann so ein Paket sich deutlich verändern. Dadurch kann es Konflikte geben, wenn Ihre verschiedenen Projekte unterschiedliche Versionen der Pakete brauchen.

Aus diesen und weiteren Gründen gibt es die Möglichkeit, **virtuelle Python-Umgebungen** einzurichten. Dabei wird eine minimale Python-Installation in Ihrem Projektordner angelegt, und Sie installieren genau die Pakete, die Sie brauchen, in genau der Version, die für Ihr Projekt geeignet ist. Das Einrichten von virtualenvs benötigt weniger umfangreiche Berechtigungen als das Verändern der systemweiten Python-Installation!

Die Balladen-Webseite, die wir zu Weihnachten gemeinsam programmiert haben, läuft auf einem Server, der bei jeder Anfrage unser Pythonprogramm ausführt. Dabei wird eine minimale, virtuelle Python-Installation verwendet, die nur die benötigten Module enthält. Einzelne Projekte, die alle auf dem gleichen Server laufen, werden gekapselt und können parallel zueinander ausgeführt werden, ohne dass sie sich in die Quere kommen.

Wir erinnern uns, dass wir zum Installieren von Paketen `pip` benutzt haben. `pip` ist ein rekursives Akronym und steht für **"pip installs packages"**. Aber wir können `pip` auch benutzen, um uns anzeigen zu lassen, welche Pakete gerade installiert sind. Im folgenden Screenshot sieht man rechts einen Teil der Liste der installierten Pakete in meiner Systeminstallation von Python; auf der linken Seite sieht man, welche Pakete gelistet werden, wenn man mit `mkdir` einen neuen Ordner anlegt, in diesem Ordner dann mit dem `venv`-Kommando ein neues virtuelles Environment erzeugt, dieses aktiviert und dann die installierten Pakete in dieser isolierten Umgebung auflistet. Nur `pip` und `setuptools` sind vorhanden. Das reicht, um je nach Bedarf für dieses Projekte alles weitere nachzuinstallieren, was ich brauche.

```
  /home/esther  mkdir "new_python_project"
  /home/esther  cd new_python_project
  /home/esther/new_python_project  python -m venv ./env
  /home/esther/new_python_project  ls
env
  /home/esther/new_python_project  source ./env/bin/activate
(env)  /home/esther/new_python_project  pip list
Package    Version
---------- -------
pip        19.0.3
setuptools 40.8.0
You are using pip version 19.0.3, however version 20.0.2 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
(env)  /home/esther/new_python_project
```

```
  /home/esther  pip list
Package             Version
------------------- --------
appdirs             1.4.3
arandr              0.1.10
astroid             2.2.5
attrs               19.3.0
backcall            0.1.0
bleach              3.1.0
boto                2.49.0
boto3               1.9.218
botocore            1.12.218
btrfsutil           1.1.1
CacheControl        0.12.5
chardet             3.0.4
Click               7.0
colorama            0.4.1
cupshelpers         1.0
cycler              0.10.0
decorator           4.4.0
defusedxml          0.6.0
distlib             0.2.9
distro              1.4.0
docopt              0.6.2
docutils            0.15.2
entrypoints         0.3
Flask               1.1.1
gensim              3.8.0
html5lib            1.0.1
idna                2.8
importlib-metadata  0.23
ipykernel           5.1.2
ipython             7.7.0
ipython-genutils    0.2.0
ipywidgets          7.5.1
isort               4.3.21
itsdangerous        1.1.0
jedi                0.15.1
Jinja2              2.10.1
jmespath            0.9.4
joblib              0.13.2
jsonschema          3.0.2
jupyter             1.0.0
jupyter-client      5.3.1
jupyter-console     6.0.0
jupyter-core        4.5.0
keyutils            0.6
kiwisolver          1.1.0
lazy-object-proxy   1.4.2
lockfile            0.12.2
louis               3.11.0
MarkupSafe          1.1.1
matplotlib          3.1.1
mccabe              0.6.1
mistune             0.8.4
more-itertools      7.2.0
```

Zusätzliche Pakete in meiner Systeminstallation sind z.B. `Flask` zum Erstellen von Webseiten, `gensim` für computerlinguistische Aufgaben, `ipython` und `jupyter` zum Verwenden von Jupyter Notebooks, `numpy` und `scipy` für statistische Analysen, `pep8` und `pycodestyle` als Linter für Pythondateien (vgl. Jupyter Notebook vom 17.12.2019), und `requests` und `urllib3` für HTTP-Requests. Einige davon sind immer wiederkehrende Werkzeuge, andere braucht man für spezielle Projekte und danach nie wieder. Es ist grundsätzlich **eine gute Idee**, in virtuellen Environments zu arbeiten. Sie können das ausprobieren, indem Sie eine bereits gelöste Hausaufgabe aus diesem Semester in einem neu angelegten virtuellen Environment ausführen.

Weitere Infos zu virtuellen Environments finden Sie z.B. auf StackOverflow (https://stackoverflow.com/questions/41972261/what-is-a-virtualenv-and-why-should-i-use-one) oder in der Dokumentation des `virtualenv` -Pakets (https://virtualenv.pypa.io/en/latest/).

Objektorientierung

Wir verwenden bisher Variablen, um je einen Wert unter selbstgewählten Namen zu speichern. Dabei können wir die Namen so wählen, dass klar wird, welche Funktion die Variablen jeweils haben.

Mit Objektorientierung haben wir die Möglichkeit, mehrere Variablen und dazugehörige Funktion zu bündeln, und zwar in Form von **Klassendefinitionen**. Eine Klasse ist ein "Rezept" für neue Objekte und kann folgendes enthalten:

- Attribute. Das sind Variablen, die einem Objekt der Klasse zugeordnet werden.
- Methoden. Das sind Funktionen, die auf Objekte der Klasse angewendet werden können.

Eine Einführung in Objektorientierung mit Python finden Sie z.B. auf Youtube (https://youtu.be/sYmuX421wfs). Hier ein kleines Beispiel, das die Klasse "Studierende" definiert und dann beliebig viele Objekte davon instanziieren kann.

Beim Erzeugen eines neuen Objekts geben wir den Namen, die Matrikelnummer und die Emailadresse an. Die Note setzen wir auf den Startwert -1, damit erkennbar ist, dass es zwar eine Note geben soll, wir darüber aber bisher noch keine Informationen haben.

Wenn wir einzelne Objekte ausgeben wollen, wird nur eine kryptische Meta-Information angezeigt, nämlich dass es sich um ein Objekt der Klasse `Student` handelt, das an einer bestimmten Speicheradresse abgelegt wurde.

Um inhaltliche Informationen über die Objekte ausgeben zu lassen, definieren wir die Methode `who_is_this()` , die auf jedes Objekt der Klasse angewendet werden kann. Rückgabe dieser Methode ist eine kurze Beschreibung des aktuellen Objekts.

Die Note können wir mit der Methode `set_final_grade()` einspeichern und mit der Methode `get_final_grade()` auslesen.

Schließlich haben wir auch noch eine Methode, mit der wir das aktuelle Objekt mit einem anderen Objekt der gleichen Klasse vergleichen können und ermitteln, wer von beiden die bessere Note hat.

In [ ]:

```
# Die "Kopfzeile" einer Klassendefinition erinnert an
# Funktionsdefinitionen oder Schleifen.
# Auch Klassendefinitionen werden eingerückt.
class Student:
    def __init__(self, inputname, inputmatrikel, inputemail):
        # Diese Funktion heißt Konstruktor.
        # Die Argumente inputname, inputmatrikel, inputemail
        # werden beim Erzeugen eines neuen Objekts angegeben
        # und dann in den Attributen self.name, self.matrikel
        # und self.email gespeichert.
        self.name = inputname
        self.matrikel = inputmatrikel
        self.email = inputemail

        # Für das Attribut self.finalgrade legen wir
        # den Startwert -1 fest.
        self.finalgrade = -1

        print("New Student:\n\tName: {}\n\tMatrikel: {}\n\tEmail: {}".format(self.name,
self.matrikel, self.email))

    def set_final_grade(self, inputgrade):
        # Diese Methode kann auf ein Objekt
        # der Klasse angewendet werden, um einen
        # Wert für das Attribut self.finalgrade
        # festzulegen.
        self.finalgrade = inputgrade
        print("Set final grade for this student to {}.".format(self.finalgrade))

    def get_final_grade(self):
        # Diese Methode gibt den Wert des Attributs
        # self.finalgrade für das aktuelle Objekt
        # zurück.
        return self.finalgrade

    def who_is_this(self):
        # Diese Methode ergänzen wir, damit wir
        # Objekte der Klasse informativ printen
        # können.
        return "Existing Student:\n\tName: {}\n\tMatrikel: {}\n\tEmail: {}".format(self
.name, self.matrikel, self.email)

    def who_is_better(self, student2):
        # Diese Methode ermittelt, wer von zwei Studierenden
        # die bessere Note hat.
        if self.finalgrade != -1 and student2.finalgrade != -1:
            if self.finalgrade < student2.finalgrade:
                return "Winner: {}, Loser: {}".format(self.name, student2.name)
            elif self.finalgrade > student2.finalgrade:
                return "Winner: {}, Loser: {}".format(student2.name, self.name)
            else:
                return "Winner: {}, Loser: {}".format(student2.name, self.name)
        return "One or both students have no final grade."

# print("Neue Objekte erstellen:")
# # Hier werden neue Objekte der Klasse Student erzeugt.
# # In den Klammern stehen die Eingabeargumente für den
# # Konstruktor.
# lisa = Student("Lisa Meier", "12345", "lisa@meier.de")
```

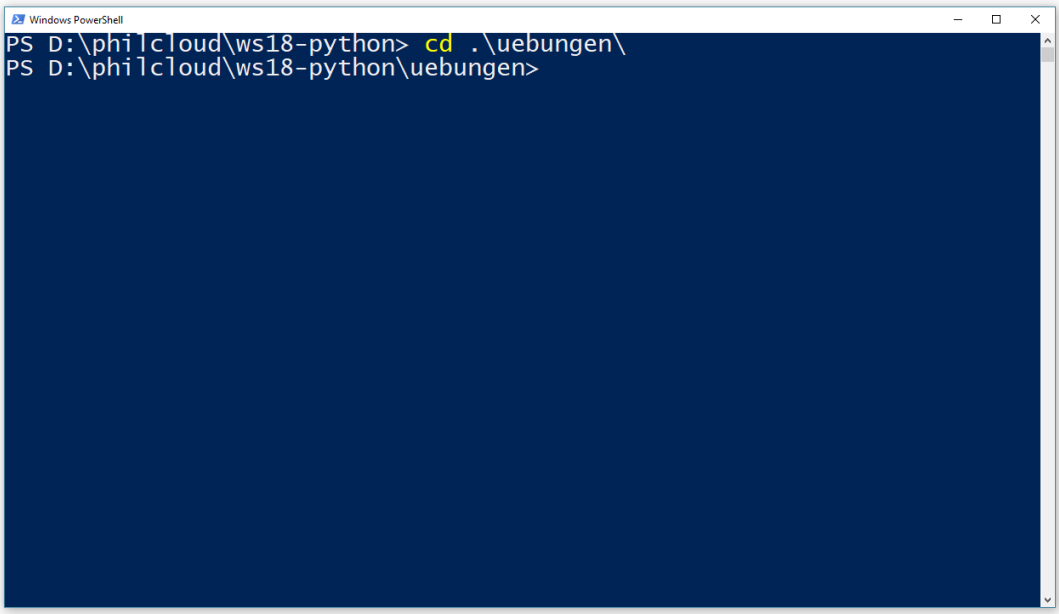
```
# hans = Student("Hans Meier", "54321", "hans@meier.de")

# print("\nBestehende Objekte printen:")
# # Wir können die Variablenwerte printen, sehen aber nur,
# # dass es sich um Objekte der Klasse Student handelt.
# print(lisa)
# print(hans)

# print("\nBestehende Objekte informativ printen:")
# print(lisa.who_is_this())
# print(hans.who_is_this())

# print("\nAttributwerte von Objekten verändern:")
# lisa.set_final_grade(1.0)
# hans.set_final_grade(2.3)

# print("\nZwei Objekte miteinander vergleichen:")
# print(lisa.who_is_better(hans))
# print(hans.who_is_better(lisa))
```

Objektorientierung ist in Python ein möglicher Stil, dem Sie folgen können, aber nicht müssen. Sie werden später im Studium auch noch Java lernen. Das ist eine Programmiersprache, in der alles objektorientiert umgesetzt wird.

Wenn Sie mit Pythoncode von anderen Leuten arbeiten, kann es vorkommen, dass Ihnen objektorientierte Programme begegnen. Sie können mehr über Objektorientierung in Python in diesem Tutorial (https://realpython.com/python3-object-oriented-programming/) oder auf Youtube (https://www.youtube.com/watch?v=ZDa-Z5JzLYM) lernen.

Python in der Kommandozeile

Die Kommandozeile/Terminal/Shell haben wir zu Beginn des Semesters kennengelernt. In der Kommandozeile können wir bestimmte Befehle ausführen, z.B. `cd` (**c**hange **d**irectory) zum Wechseln des aktuellen Verzeichnisses:



Auch Pythonprogramme können von der Kommandozeile aus aufgerufen werden. VSCode macht das für uns automatisch, wenn wir das aktuelle Programm ausführen.

Wir können noch eine Reihe weiterer Argumente ergänzen, wenn wir das möchten. Wenn wir den Befehl `python -h` (**h**elp) ausführen, erhalten wir eine Übersicht über mögliche Eingaben:

```
 C:\Users\eseyffarth> python -h
usage: C:\ProgramData\Anaconda3\python.exe [option] ... [-c cmd | -m mod | file
 | -] [arg] ...
Options and arguments (and corresponding environment variables):
-b     : issue warnings about str(bytes_instance), str(bytearray_instance)
         and comparing bytes/bytearray with str. (-bb: issue errors)
-B     : don't write .py[co] files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser; also PYTHONDEBUG=x
-E     : ignore PYTHON* environment variables (such as PYTHONPATH)
-h     : print this help message and exit (also --help)
-i     : inspect interactively after running script; forces a prompt even
         if stdin does not appear to be a terminal; also PYTHONINSPECT=x
-I     : isolate Python from the user's environment (implies -E and -s)
-m mod : run library module as a script (terminates option list)
-O     : optimize generated bytecode slightly; also PYTHONOPTIMIZE=x
-OO    : remove doc-strings in addition to the -O optimizations
-q     : don't print version and copyright messages on interactive startup
-s     : don't add user site directory to sys.path; also PYTHONNOUSERSITE
-S     : don't imply 'import site' on initialization
-u     : unbuffered binary stdout and stderr, stdin always buffered;
         also PYTHONUNBUFFERED=x
         see man page for details on internal buffering relating to '-u'
-v     : verbose (trace import statements); also PYTHONVERBOSE=x
         can be supplied multiple times to increase verbosity
-V     : print the Python version number and exit (also --version)
         when given twice, print more information about the build
-W arg : warning control; arg is action:message:category:module:lineno
         also PYTHONWARNINGS=arg
-x     : skip first line of source, allowing use of non-Unix forms of #!cmd
-X opt : set implementation-specific option
file   : program read from script file
-      : program read from stdin (default; interactive mode if a tty)
arg ...: arguments passed to program in sys.argv[1:]

Other environment variables:
PYTHONSTARTUP: file executed on interactive startup (no default)
PYTHONPATH   : ';'-separated list of directories prefixed to the
               default module search path.  The result is sys.path.
PYTHONHOME   : alternate <prefix> directory (or <prefix>;<exec_prefix>).
               The default module search path uses <prefix>\lib.
PYTHONCASEOK : ignore case in 'import' statements (Windows).
PYTHONIOENCODING: Encoding[:errors] used for stdin/stdout/stderr.
PYTHONFAULTHANDLER: dump the Python traceback on fatal errors.
PYTHONHASHSEED: if this variable is set to 'random', a random value is used
   to seed the hashes of str, bytes and datetime objects.  It can also be
   set to an integer in the range [0,4294967295] to get hash values with a
   predictable seed.
PYTHONMALLOC: set the Python memory allocators and/or install debug hooks
```

In [ ]:

```
import sys      # Wir importieren das Paket sys,
                # das uns Zugriff auf die Argumente gibt.

print(sys.argv)  # In der Liste sys.argv sind alle Eingabeargumente gespeichert.

name = sys.argv[-1]  # Das letzte Argument entspricht dem letzten Argument,
                     # das in der Kommandozeile eingegeben wurde.

print("Hallo {}!".format(name))
```

Bitte nicht wundern - dieser Code macht in Jupyter nicht das, was wir wollen. Das liegt daran, dass Jupyter keine Kommandozeilen-Umgebung ist. Speichern Sie den Code aus dem Kästchen auf Ihrem Computer als Datei und rufen Sie diese Datei dann von der Kommandozeile aus auf:

```
[1]   C:\Users\eseyffarth> cd D:\philcloud\ws18-python\
[2]   D:\philcloud\ws18-python> python inputtest.py Esther
[3]   ['inputtest.py', 'Esther']
[4]   Hallo Esther!
[5]   D:\philcloud\ws18-python>
```

In Zeile `[1]` wechseln wir in das Verzeichnis, in dem die Pythondatei liegt.

Zeile `[2]` enthält den Aufruf unserer Datei mit einem zusätzlichen Argument, z.B. `Esther` .

In Zeile `[3]` werden alle Argumente des Pythonaufrufs als Liste ausgegeben. Der Dateiname ist dabei selbst auch ein Argument.

In Zeile `[4]` wird der print-Aufruf aus dem Programm ausgeführt, und zwar mit dem angegebenen Namen anstelle des Platzhalters im Formatstring.

Zeile `[5]` zeigt uns, dass das Pythonprogramm beendet wurde und die Kommandozeile bereit ist für die nächste Eingabe.

Warum Kommandozeilenargumente verwenden?

Bisher haben wir in den Übungen Programme geschrieben, die genau eine Aufgabe erledigt haben. Ab und zu kamen dabei Nutzereingaben vor, z.B. wenn beliebige lateinische Wörter dekliniert werden sollten oder wenn Dateipfade angegeben werden mussten.

Im Kontext dieses Kurses war es kein Problem, die Programme interaktiv auszuführen: Wir haben das Programm gestartet, an der passenden Stelle die Eingabe getippt und das Programm lief weiter.

Mithilfe von Kommandozeilenargumenten können wir die gleiche Funktionalität erreichen, trennen dabei aber die Nutzereingaben vom eigentlichen Programmablauf. So kann z.B. ein Programm auf dem Server gestartet werden, von einem anderen Skript aufgerufen werden oder für nachts um 3 geplant werden, ohne dass Sie danebensitzen und Texte eingeben müssen.

Wenn Sie sich dafür interessieren, können Sie z.B. die Aufgabe 08-01 umschreiben. Dort haben wir lateinische Wörter dekliniert, die während der Laufzeit eingegeben wurden.

Ändern Sie das Programm so, dass das Eingabewort aus der Kommandozeile gelesen wird. Der Aufruf des Programms sieht dann so aus:

```
D:\python> python 08-01_String-Formatierung.py amica
```

Graphische Programminterfaces (GUIs)

Wir haben bisher immer nur im Editor und der Kommandozeile programmiert. Wenn wir eine Anwendung schreiben, die auch für End-User gedacht ist, ist es eine gute Idee, eine graphische Darstellung des Programms zu bauen.

Wir unterscheiden zwischen **Frontend** ("das, was man sieht") und **Backend** ("das, was im Hintergrund passiert"). Als Grundlage für Ihr Backend können Sie jedes Pythonmodul verwenden, dessen Funktionalität in Funktionen verpackt ist.

Das Frontend können Sie mit verschiedenen Frameworks umsetzen. Ein Paket, wir dafür benutzen können, ist `tkinter`.

Der folgende Code erzeugt beim Ausführen ein Fenster mit einem Titel, einer definierten Höhe und Breite, sowie einem Knopf mit der Beschriftung "Fenster schließen". Wenn der Knopf gedrückt wird, wird das Fenster geschlossen und das Programm beendet.

Der Code ist diesem Online-Tutorial (https://pythonprogramming.net/python-3-tkinter-basics-tutorial/) entnommen, das Schritt für Schritt die Elemente herleitet, die wir in `tkinter` verwenden können.

Die Zeile, die den Button erstellt, enthält auch die Information, was beim Klicken des Buttons passieren soll: `command=root.destroy`. An dieser Stelle können Sie auch selbstdefinierte Funktionen angeben, die dann beim Klick ausgeführt werden. Natürlich müssen Sie sich dann Gedanken darüber machen, wie die Ergebnisse der Funktion auf geeignete Weise dargestellt werden können.

Hier das grundlegende Programm:

In [7]:

```
# Quelle: https://pythonprogramming.net/python-3-tkinter-basics-tutorial/
# ACHTUNG: Beim Klick auf "Fenster schließen" wird der
# Interpreter beendet, der hier im Jupyter Notebook
# enthalten ist. Kopieren Sie den Code in eine Datei
# und führen Sie ihn in VSCode oder von der Kommandozeile
# aus.
from tkinter import *


# Here, we are creating our class, Window, and inheriting from the Frame
# class. Frame is a class from the tkinter module. (see Lib/tkinter/__init__)
class Window(Frame):

    # Define settings upon initialization. Here you can specify
    def __init__(self, master=None):

        # parameters that you want to send through the Frame class.
        Frame.__init__(self, master)

        #reference to the master widget, which is the tk window
        self.master = master

        #with that, we want to then run init_window, which doesn't yet exist
        self.init_window()

    #Creation of init_window
    def init_window(self):

        # changing the title of our master widget
        self.master.title("Unser erstes GUI-Programm")

        # allowing the widget to take the full space of the root window
        self.pack(fill=BOTH, expand=1)

        # creating a button instance
        quitButton = Button(self, text="Fenster schließen", command=root.destroy)

        # placing the button on my window
        quitButton.place(x=0, y=0)



    def client_exit(self):
        exit()

# root window created. Here, that would be the only window, but
# you can later have windows within windows.
root = Tk()

root.geometry("400x300")

#creation of an instance
app = Window(root)

#mainloop
root.mainloop()
```

Threading (mehrere parallele Prozesse ausführen)

Unsere bisherigen Pythonprogramme liefen linear ab: Der Interpreter führte eine Zeile nach der anderen aus, manchmal als Schleifen, bis das Programm beendet wurde.

Mithilfe von Threading können wir den Interpreter mehrere Dinge parallel ausführen lassen. Ein Tutorial dazu finden Sie hier (https://pythonprogramming.net/threading-tutorial-python/?completed=/tkinter-adding-text-images/).

Threading ist vor allem dann nützlich, wenn große Datenmengen verarbeitet werden, also beispielsweise im Kontext von Machine Learning.

Weitere nützliche Pakete

- `spaCy` (https://spacy.io/) und `textblob` (https://textblob.readthedocs.io/en/dev/) erfüllen ähnliche Aufgaben wie `NLTK`. Dabei ist `spaCy` für einige Dinge besser geeignet als `NLTK` (https://blog.thedataincubator.com/2016/04/nltk-vs-spacy-natural-language-processing-in-python/), dafür aber auch für Anfänger_innen möglicherweise schwieriger zu installieren und zu nutzen. `textblob` ist weniger umfangreich als die beiden anderen Pakete.
- `requests` (http://docs.python-requests.org/en/master/) kann verwendet werden, um URLs zu laden und ähnlich wie ein Browser zu agieren.
- `pronouncing` ist ein Paket (https://pronouncing.readthedocs.io/en/latest/), das eine Schnittstelle zum Aussprachewörterbuch CMU Dictionary bereitstellt. Damit können Sie z.B. ermitteln, ob zwei englische Wörter sich reimen oder wieviele Silben eine englische Phrase hat.
- `numpy` bietet eine Reihe mathematischer Funktionen (http://www.numpy.org/), mit denen Sie wissenschaftlich arbeiten können, um z.B. Messdaten zu verarbeiten.

Zusammenfassung und Ausblick

Falls Sie Computerlinguistik studieren, wird Python Sie in den nächsten Semestern weiterhin begleiten. Auch einige Strategien und Tools, die wir behandelt haben, werden weiterhin relevant sein. Hier eine Auswahl an Dingen, die wir hier behandelt haben und die Ihnen demnächst wieder begegnen werden:

- **Rekursion** ist hilfreich, wenn wir verschachtelte Strukturen verarbeiten. Syntaxbäume sind Beispiele für solche Strukturen und werden im Kurs "Einführung in die Syntax" (Modul L1, 2. Semester) behandelt.
- **Reguläre Ausdrücke** sind ein Universalwerkzeug, das jede_r Computerlinguist_in kennen sollte. Sie werden im Kurs "Einführung in die Computerlinguistik" (Modul CL1, 2. Semester) ausführlicher behandelt.
- **Schleifen, bedingte Ausführung und Funktionen** gibt es in vielen Programmiersprachen. Ihre Fähigkeiten, Programme zu strukturiert zu planen und umzusetzen, wird in jedem weiteren Programmierkurs hilfreich sein, auch wenn Sie in anderen Sprachen diese Konstrukte mit einer anderen Syntax umsetzen.
- **N-Gramme** auf Zeichenbasis haben wir in der 14. Woche oberflächlich besprochen. Was man mit N-Grammen alles machen kann und wie die statistischen Hintergründe davon genau aussehen, wird im Kurs "Einführung in die Computerlinguistik" behandelt.

Wir freuen uns, dass Sie so erfolgreich an diesem Kurs teilgenommen haben, und wünschen Ihnen alles Gute für Ihr weiteres Studium!