# PSPP-Praktikum 12
# Python (Teil 1)

## 1. Hinweise zum Python-Praktikum

Die meisten Aufgaben dieses Praktikums sind für den Einstieg in Python gedacht. Wenn Sie Python bereits kennen, können Sie direkt zu Aufgabe 6 springen. Das ist auch die einzige Aufgabe zur Abgabe. Sicher ist es kein Fehler, die Einführung trotzdem schnell durchzusehen.

### Python-Installation

Für das Praktikum benötigen Sie eine Python-Installation. Auf Unix Plattformen (Linux, Mac) ist diese häufig schon vorhanden. Geben Sie zum Test in einer Shell einmal *python* ein. Wichtig ist, eine aktuelle Version von Python 3 zu haben.

Versionen zum Download für verschiedene Plattformen sind hier zu finden:
http://www.python.org/download/

Auf dem *dublin.zhaw.ch* (erreichbar via *ssh* im Campusnetz oder via VPN) ist eine Version von Python 3 bereits installiert, Aufruf: *python3*. Zur Not können die meisten Aufgaben auch damit erledigt werden.

Ganz gut bewährt hat sich zur Arbeit mit Python der *VSCode* mit der Erweiterung *Pylance*. Die Erweiterung Code Runner erlaubt es, Code mit einem Klick auszuführen.

Umfangreiche Dokumentationen, Referenzen und Tutorials befinden sich auf der Python Website unter http://www.python.org/doc/.

# 2. Von Lisp zu Python

In diesem Abschnitt sind zur Einführung die wichtigsten Unterschiede von Lisp und Python zusammengestellt. Python hat einige Elemente von Lisp übernommen, auch wenn die Syntax sich stark unterscheidet.

## Zahlen und Operatoren

Anstatt der Präfix-Notation in Lisp wird in Python die gewohnte Infix-Notation für Ausdrücke verwendet. Testen Sie die Ausdrücke im Python-Interpreter:

| Lisp | Python |
|---|---|
| `(- (* 10 (+ 4 3 3)) 20)` | `(10 * (4 + 3 + 3)) - 20` |
| `(expt 12 32)` | `12 ** 32` |
| `(* 35 1.16)` | `35 * 1.16` |
| `(zerop 25)` | `25 == 0` |

Während in Python die Reihenfolge der Auswertung von der Bindungsstärke der Operatoren abhängt und diese Reihenfolge durch Klammern angepasst werden kann, sind in Lisp die Klammern um jeden (Teil-) Ausdruck erforderlich, so dass die Reihenfolge immer klar ist.

## Symbole, Strings und Listen

Wie Lisp unterstützt auch Python Listen von Elementen. In Python werden Listen in eckigen Klammern geschrieben und die Elemente mit Kommas getrennt. Statt der Symbole in Lisp verwenden wird in den folgenden Beispielen Strings.

| Lisp | Python |
|---|---|
| `(apfel banane zitrone)` | `['apfel', 'banane', 'zitrone']` |
| `(car '(eins zwei drei))` | `['eins', 'zwei', 'drei'][0]` |
| `(cdr '(eins zwei drei))` | `['eins', 'zwei', 'drei'][1:]` |
| `(length '(1 2 3))` | `len([1, 2, 3])` |
| `(append '(1 2 3) '(4 5))` | `[1, 2, 3] + [4, 5]` |

Der Zugriff auf einzelne Elemente einer Liste geschieht in Python über den Index in eckigen Klammern, Teilsequenzen sind über das sogenannte Slicing zugänglich:

`liste[n:m]` Teilsequenz vom n-ten bis zum (m-1)-ten Element
`liste[n:]` Teilsequenz vom n-ten bis zum Ende der Liste
`liste[:m]` Teilsequenz vom Anfang bis zum (m-1)-ten Element

Ein negativer Index wird von hinten gezählt. Listen sind veränderbare Sequenzen. Sie können beliebige Datentypen enthalten, also auch weitere Listen.

Strings werden mit einfachen oder doppelten Anführungszeichen geschrieben. Strings sind unveränderbare Sequenzen.

## Tupel und Dictionaries

Tupel in Python verhalten sich ähnlich wie Listen, sind aber unveränderlich. Sie werden in runden Klammern geschrieben (welche je nach Situation auch weggelassen werden können):

| | |
|---|---|
| `(1, 2, 3)` | Tupel aus drei Zahlen |
| `(1, 2, 3)[:2]` | Teilsequenz vom n-ten bis zum Ende der Liste |
| `a = "er", "sie", "es"` | Zuweisung des Tupels `('er', 'sie', 'es')` |

Dictionaries speichern Schlüssel-Wert-Paare. Sie werden in geschweiften Klammern geschrieben (und ähneln somit Objekten in JavaScript). Die Zuweisung von Werten erfolgt mit Hilfe des Zuweisungsoperators „=“. Werte können beliebige Python-Objekte sein.

```
telefon = { "bernd": "0921/76499", "susi": "0371/233444" }
telefon["susi"]
telefon["hans"] = "0366/873543"
```

## Funktionen

Python-Funktionen werden mit *def* eingeführt. Blöcke von Anweisungen sind dabei durch konsistente Einrückung festgelegt („significant whitespace“). Die Rückgabe des Funktionswerts erfolgt mit Hilfe der *return*-Anweisung.

| Lisp | Python |
|---|---|
| `(defun list-double (lst)`<br>`  (if (null lst)`<br>`  nil`<br>`  (cons (* 2 (car lst))`<br>`  (list-double (cdr lst)))))` | `def list_double(seq):`<br>`  if len(seq) == 0:`<br>`  return []`<br>`  else:`<br>`  return [2 * seq[0]] + \`<br>`  list_double(seq[1:])` |

Die Lisp-Implementierung der Funktion könnte mit *mapcar* vereinfacht werden. In Python ist aber auch noch eine andere, einfachere Variante möglich:

```
def list_double(seq):
    return [2*x for x in seq]
```

## 3. Versuche mit Typen und Literalen

Machen Sie sich mit den Datentypen von Python vertraut. Führen Sie die folgenden Ausdrücke im Python Interpreter aus und versuchen Sie zu erklären was geschieht (sehen Sie bei Bedarf in der Python Doku nach):

```
>>> "spam" + "eggs"
>>> s = "ham"
>>> "eggs " + s
>>> s * 5
>>> s[1]+s[:-1]
>>> "green %s and %s" % ("eggs", s)
>>> ('x', 'y')[1]
>>> z = [1, 2, 3] + [4, 5, 6]
>>> z, z[:], z[:0], z[-2], z[-2:]
>>> z.reverse(); z
>>> {'a':1, 'b':2}['b']
>>> d = {'x':1, 'y':2, 'z':3}
>>> d['w'] = 0
>>> d[(1,2,3)] = 4
>>> d.keys()
>>> list(d.keys()), list(d.values())
```

## 4. Kontrollstruktur

Geben Sie folgende Zeilen (ohne Prompts >>> und ...) in die Python REPL ein und erklären Sie das Ergebnis. Achten Sie darauf, dass die Zeilen unter dem *while* um die gleiche Anzahl Leerschläge eingerückt werden.

```
>>> a, b = 0, 1
>>> while b < 200:
...     print(b, end=' ')
...     a, b = b, a+b
```

# 5. Funktionen

## Zeichencodes

Schreiben Sie eine Funktion *strcodes*, die für einen String eine Liste der Zeichencodes ausgibt. Verwenden Sie die *map*- und die *ord*-Funktion (s. Doku).

```
>>> strCodes("Hallo")
[72, 97, 108, 108, 111]
```

Ändern Sie Ihre Funktion so ab, dass die Summe der Zeichencodes ausgegeben wird:

```
>>> strSum("Hallo")
496
```

## Listen

Schreiben Sie eine Funktion *zipLists*, welche mehrere Listen in eine Liste von Tupeln umwandelt (es gibt bereits eine Funktion *zip*, damit ist es natürlich einfach…):

```
>>> zipLists([1,2,3], [5,6,7], ['a', 'b', 'c'])
[(1, 5, 'a'), (2, 6, 'b'), (3, 7, 'c')]
```

Schreiben Sie eine erweiterte Funktion *zipListsFun* in der die Elemente mit einer Funktion verknüpft werden:

```
>>> def combine(a,b,c): return str(a+b) + c
>>> zipListsFun(combine, [1,2,3], [5,6,7], ['a', 'b', 'c'])
['6a', '8b', '10c']
```

## Dictionaries

Schreiben Sie eine Funktion *dictInvert*, welche ein Dictionary invertiert:

```
>>> dictinvert({'a': 55, 'b': 55, 'c': 88})
{88: ['c'], 55: ['a', 'b']}
```

## Flache Liste

Und zum Schluss noch ein Beispiel, das wir bereits in Lisp umgesetzt haben: Schreiben Sie eine Funktion, die zu einer möglicherweise verschachtelten Listen- oder Tupel-Struktur eine flache Liste zurückgibt. Empfehlung: Wie in Lisp werden solche verschachtelten Listenstrukturen am besten mit einer rekursiven Funktion verarbeitet.

```
>>> flatten([[[1, 2], 3], [[4]], (5, 6), ([7])])
[1, 2, 3, 4, 5, 6, 7]
```

# 6. Wortliste verarbeiten *(Abgabe)*

Die folgende Funktion wurde mit verschiedenen Klassen und Funktionen der Python Standard Library geschrieben. Sie kann auch unter dem Namen *testthis.py* mit den Praktikumsunterlagen heruntergeladen werden.

```
import urllib.request
import html
import re

def doload(url):
    contents = urllib.request.urlopen(url).read()
    contents = contents.decode(encoding="utf-8")
    data = html.unescape(contents)
    withouttags = re.sub(r"<(.|\s)*?>", " ", data)
    return re.sub(r"\s+", " ", withouttags).split()
```

Laden Sie das Modul und testen Sie die Funktion zum Beispiel mit folgendem Aufruf:

```
>>> import testthis
>>> testthis.doload("https://lite.cnn.com")
```

Nun haben wir eine Liste von Wörtern, die auf der geladenen Seite vorkommen (und einige weniger relevante Teile wie CSS-Regeln und Scripts). Schreiben Sie eine Funktion *trending*, welche die zehn häufigsten Wörter als Liste zurückgibt. In der Regel erhält man auf diese Weise noch nicht die wichtigsten Wörter. Sicher müsste man noch Füllwörter wie *the, a, this, with* herausnehmen. Eventuell braucht es ausserdem noch weitere Heuristiken, um eine geeignete Liste relevanter Wörter zu bekommen. Zum Beispiel:

```
>>> trending(doload("https://lite.cnn.com"))
['Gaza', 'Trump', 'hospital', 'Ukraine', 'Israel', 'Elon', 'Biden', 'Al-Shifa', …]
```

Ihre Lösung sollte mindestens die zehn häufigsten Wörter liefern. Nett wäre es natürlich, wenn irrelevante Wörter (*the, a, this, with, says, will, have, …*) ausgefiltert würden.

**Hinweis zur Abgabe:**
Geben Sie die Python-Datei mit Ihrer Lösung bis zum nächsten PSPP ab:
https://radar.zhaw.ch/python/UploadAndCheck.html

Praktikum: `PS12`
Name und Kurznamen angeben und *Upload* wählen.