PSPP-Praktikum 12
Python (Teil 1)

1. Python Installation

Für das Praktikum benötigen Sie eine Python-Installation. Auf Unix Plattformen (Linux, Mac) ist diese normalerweise schon vorhanden. Geben Sie zum Test in einer Shell einmal *python* ein. Von Vorteil wäre allerdings eine aktuelle Version von Python 3. Versionen für verschiedene Plattformen sind unter http://www.python.org/download/ zu finden.

Für den Fall, dass Sie nichts installieren möchten, können Sie sich auch via *ssh* auf dem Laborserver *dublin* anmelden und dort *python* ausführen, wenn auch in einer älteren Version von Python 2. Aber es schadet sicher nicht, jederzeit eine leistungsfähige Scripting-Umgebung wie Python, Ruby oder etwas Ähnliches auf dem Notebook bereit zu haben (und diese auch einsetzen zu können...).

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)`
`  (if (null lst)`
`  nil`
`  (cons (* 2 (car lst))`
`  (list-double (cdr lst)))))`
	`def list_double(seq):`
	`  if len(seq) == 0:`
	`  return []`
	`  else:`
	`  return [2 * seq[0]] + \`
	`  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 den Datentypen

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 *(Abgabe)*

Die folgenden Teilaufgaben können Sie einerseits direkt in der Python-Shell durchführen, es empfiehlt sich aber, die Funktionen zunächst in einem Editor zu schreiben, um nicht bei jedem Fehler alles wieder neu eintippen zu müssen.

Probieren Sie auch, die Funktionen in einer Python-Datei zu schreiben. Denken Sie dabei daran, dass ein Modul mit *import* nur beim ersten Aufruf geladen wird.

Zeichencodes

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

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

Statt der *for*-Schleife wäre auch ein Mapping möglich. Das ist dann der eher funktionale Ansatz:

```
>>> list(map(lambda c: ord(c), "Hallo"))
[72, 97, 108, 108, 111]
```

Wozu eigentlich das Lambda? Hier ist es tatsächlich nicht nötig:

```
>>> list(map(ord, "Hallo"))
[72, 97, 108, 108, 111]
```

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

```
>>> strsum("Hallo")
496
```

Dictionaries

Schreiben Sie eine Funktion *printdict*, die ein Dictionary nach Schlüsseln sortiert ausgibt:

```
>>> printdict({"beta": 22, "alfa": 11, "gamma": 33})
alfa    11
beta    22
gamma   33
```

Schreiben Sie ausserdem eine Funktion *addDict(dict1, dict2)*, die zwei Dictionaries vereinigt und dabei ein neues Dictionary erzeugt. Wenn beide für den gleichen Schlüssel unterschiedliche Werte haben, soll dict1 des ersten Parameters Vorrang haben.

```
>>> addDict({1:111, 2:222}, {2:999, 3:333})
{1: 111, 2: 222, 3: 333}
```

Methodenverzeichnis

Geben Sie folgendes ein und versuchen Sie, sich das Ergebnis zu erklären:

```
>>> dir([])
>>> dir({})
>>> {}.values
>>> {}.values.__doc__
```

Mengen

Im Unterricht gab es als Beispiel eine Funktion *intersect*, welche die Schnittmenge von zwei Mengen bildet (Listen als Mengen aufgefasst). Obwohl es in Python bereits ein Modul für die Arbeit mit Mengen gibt, sollen hier zur Übung noch zwei weitere Mengen-Funktionen selbst implementiert werden. Ergänzen Sie die *intersect*-Funktion noch um eine Funktion *union* für die Vereinigungsmenge und eine Funktion *diff* für die Differenzmenge:

```
>>> union([1, 2, 3], [3, 4])
[1, 2, 3, 4]
>>> diff([1, 2, 3], [3, 4])
[1, 2]
```

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. Modul mit Testcode

Schreiben Sie eine Funktion *adder* in einer Python Datei *adder.py*. Die Funktion soll zwei Argumente akzeptieren und die Summe der beiden Argumente zurückgeben. Fügen Sie Test-Code am Ende der Datei hinzu, der die Funktion mit einer Reihe von Datentypen (Zahlen, Strings, Listen) aufruft. Rufen Sie das Script mit *python adder.py* auf.

Hinweis: Um zu testen, ob ein Modul importiert oder als Programm gestartet wurde, kann das Attribut *__name__* überprüft werden.

```
version = 1.0

if __name__ == '__main__':
    print('Module testcode started separately...')
```

Je nachdem, wie das Modul geladen wird, wird dann der Code im if-Block ausgeführt oder nicht:

```
$ python3
...
>>> import testcode
>>> testcode.version
1.0
>>> ^D
$ python3.3 testcode.py
Module testcode started separately...
```

7. Python Standard Library

Die folgende Funktion wurde schnell 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.parse
import http.client
import re

def doload(url):
    o = urllib.parse.urlparse(url)
    conn = http.client.HTTPConnection(o.netloc)
    conn.request("GET", o.path)
    resp = conn.getresponse()
    data = resp.read().decode()
    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("http://dublin.zhaw.ch/~bkrt/hallo.html")
>>> testthis.doload("http://lite.cnn.io/en")
```

Können Sie die einzelnen Anweisungen nachvollziehen? Ein Blick in die Beschreibung der verwendeten Funktionen und Klassen kann nicht schaden...