Informatik I: Einführung in die Programmierung

Prof. Dr. Peter Thiemann
Dr. Daniel Büscher, Hannes Saffrich
Wintersemester 2019

Universität Freiburg
Institut für Informatik

Übungsblatt 11 – Lösungen

Abgabe: Montag, 20.01.2020, 9:00 Uhr morgens

Aufgabe 11.1 (Testen; Dateien: `words.py`, `test.txt`; Punkte: 3+4+1)

Unter einem Wort verstehen wir im Folgenden jede endliche Folge von Buchstaben des deutschen Alphabets der Länge ≥ 1. Je zwei Wörter in einer Zeichenfolge werden durch eine nicht-leere, endliche Folge von Zeichen, die nicht zu diesen Buchstaben gehören getrennt. Die Funktion `next_word(s)` soll, angewendet auf einen String `s`, ein Tupel `(word, rest)` zurückgeben, wobei `word` das erste Wort in `s` ist und `rest` die Zeichenfolge ist, die in `s` auf `word` folgt. Falls `s` kein Wort enthält, gibt die Funktion das Tupel `(None, "")` zurück. Die folgenden Definitionen enthalten einen syntaktischen Fehler, einen semantischen Fehler und einen Laufzeit-Fehler, der bei vielen (auch kurzen) Eingaben auftritt[1]. Fehler sind also in genau 3 Zeilen zu finden.

```
LETTERS = ("abcdefghijklmnopqrstuvwxyz"
           "ABCDEFGHIJKLMNOPQRSTUVWXYZÄÖÜäöüß")

def _next_word_helper(s: str) -> tuple:
    """Helper function for next_word."""
    if not s:
        return None, s
    if s[0] not in LETTERS:
        return None, s
    word = s[0]
    word_rest, s_rest = _next_word_helper(s[1:])
    if word_rest:
        word = word_rest
    return word, s_rest

def next_word(s: str) -> tuple:
    """Return the first word of an input string s and the rest of it."""
    while s[0] not in LETTERS
        s = s[1:]
    return _next_word_helper(s)
```

(a) Finden und korrigieren Sie die Fehler! Laden Sie hierzu die Datei `words.py` von der Website oder kopieren Sie den Code in die angegebene Datei. Kommentieren Sie die fehlerhaften Zeilen aus und fügen Sie eine Korrektur der jeweils

[1] Bei Eingaben mit sehr langen Wörtern gibt es einen weiteren Laufzeitfehler, der hier aber nicht behandelt werden soll.

fehlerhaften Zeile nach dem Kommentarblock ein. Ergänzen Sie jeweils auch einen Kommentar, der erklärt warum der Fehler auftritt und um welchen Typ von Fehler es sich handelt (Syntax-Fehler, Laufzeit-Fehler oder semantischer Fehler).

(b) Implementieren Sie 4 Unittests mit Namen `test_next_word_1()` bis `test_next_word_4()`. Die Testfälle sollen insbesondere auch die Fehler aus der vorherigen Aufgabe abdecken, also im nicht-korrigierten Code entsprechend fehlschlagen.

(c) Führen Sie in der Shell bzw. Eingabeaufforderung das Kommando

`pytest -v words.py` (bzw. `python3 -m pytest -v words.py`)

zum Durchführen der Tests aus. Kopieren Sie die Shell-Ausgabe in eine Datei `test.txt` und committen Sie auch diese Datei zum SVN-Repository. Stellen Sie sicher, dass `pytest` auch wirklich installiert ist. Die Ausgabe des Kommandos beginnt dann mit:

```
============ test session starts ==============
```

Lösung:

(a)
```
LETTERS = ("abcdefghijklmnopqrstuvwxyz"
           "ABCDEFGHIJKLMNOPQRSTUVWXYZÄÖÜäöüß")

def _next_word_helper(s: str) -> tuple:
    """Helper function for next_word."""
    if not s:
        return None, s
    if s[0] not in LETTERS:
        return None, s
    word = s[0]
    word_rest, s_rest = _next_word_helper(s[1:])
    if word_rest:

        # Semantischer Fehler:
        # "word" muss mit "word_rest" erweitert (nicht ersetzt) werden.
        #word = word_rest

        word += word_rest
    return word, s_rest

def next_word(s: str) -> tuple:
    """Return the first word of an input string s and the rest of it."""

    # Syntax-Fehler:
    # Nach LETTERS fehlt ein ":".
```

```
    #while s[0] not in LETTERS

    # Laufzeit-Fehler:
    # Strings ohne Buchstaben führen zu "index out of range" exception.
    #while s[0] not in LETTERS:

    for i in range(0, len(s)):
        if s[0] in LETTERS:
            break
        s = s[1:]
    return _next_word_helper(s)
```

(b)
```
def test_next_word_1():
    assert next_word("") == (None, "")

def test_next_word_2():
    assert next_word(" 123 ") == (None, "")

def test_next_word_3():
    assert next_word("foo") == ("foo", "")

def test_next_word_4():
    assert next_word(" foo bar ") == ("foo", " bar ")
```

(c)
```
============================= test session starts ==============================
platform linux -- Python 3.5.2, pytest-5.0.1, py-1.8.0, pluggy-0.12.0 -- /usr/bi
n/python3
cachedir: .pytest_cache
rootdir: /home/...
collected 4 items

pool/errors_nextword19_corrected.py::test_next_word_1 PASSED             [ 25%]
pool/errors_nextword19_corrected.py::test_next_word_2 PASSED             [ 50%]
pool/errors_nextword19_corrected.py::test_next_word_3 PASSED             [ 75%]
pool/errors_nextword19_corrected.py::test_next_word_4 PASSED             [100%]

============================ 4 passed in 0.01 seconds ============================
```

Aufgabe 11.2 (Leap Year Coverage; Datei: `leap_year.py`, Punkte: 4)
Die Datei `leap_year.py` enthält folgende Implementierung der Schaltjahr-Berechnung:

```
def leap_year(year: int) -> bool:
    """Returns whether `year` is a leap year."""
    if year % 4 != 0:
        return False
    elif year % 400 == 0:
        return True
```

```
    elif year % 100 == 0:
        return False
    else:
        return True
```

Laden Sie sich die Datei von der Vorlesungswebseite herunter und fügen Sie 4 Unittests hinzu die volle *code coverage* erreichen, also insgesamt jede Anweisung in der Implementierung von `leap_year` testen.

Schreiben Sie *docstrings* für die Test-Funktionen, in denen Sie kurz erklären welche Anweisungen abgedeckt werden.

Lösung:

```
def test_leap_year_reject_regular():
    '''Not divisible by 4 => reaches first branch.'''
    assert not leap_year(1601)

def test_leap_year_accept_irregular():
    '''Divisible by 4 and 400 => reaches second branch.'''
    assert leap_year(1600)

def test_leap_year_reject_irregular():
    '''Divisible by 4 and 100, but not by 400 => reaches third branch.'''
    assert not leap_year(1700)

def test_leap_year_accept_regular():
    '''Divisible by 4, but not by 100 and 400 => reaches fourth branch.'''
    assert leap_year(1604)
```

Aufgabe 11.3 (Traceback; Datei: `traceback.txt`, Punkte: 2)

Betrachten Sie den folgenden Pythoncode:

```
 1    def f(n: int) -> int:
 2        if n < 2:
 3            return g(n)
 4        elif n % 2 == 0:
 5            return f(3 * n + 1)
 6        else:
 7            return f(n // 2)
 8
 9    def g(n: int) -> int:
10        return 100 // (n * n - 1)
11
12    print(f(5))
```

Die Ziffern auf der linken Seite sind die Zeilennummern, wie sie auch in den meisten Texteditoren angezeigt werden.

Das Ausführen des Pythoncodes führt zu einem Laufzeitfehler, der durch folgende *traceback* ausgegeben wird:

```
Traceback (most recent call last):
  File "stack_trace.py", line 12, in <module>
    print(f(5))
  File "stack_trace.py", line 7, in f
    return f(n // 2)
  File "stack_trace.py", line 5, in f
    return f(3 * n + 1)
  File "stack_trace.py", line 7, in f
    return f(n // 2)
  File "stack_trace.py", line 7, in f
    return f(n // 2)
  File "stack_trace.py", line 3, in f
    return g(n)
  File "stack_trace.py", line 10, in g
    return 100 // (n * n - 1)
ZeroDivisionError: integer division or modulo by zero
```

In vielen Fällen reicht ein genauer Blick auf die *traceback* aus um den Fehler zu finden und nachvollziehen zu können wieso er aufgetreten ist.

Erklären Sie in der Datei `traceback.txt` den Zusammenhang zwischen der *traceback* und der Ausführung des Pythoncodes und wie die *traceback* benutzt werden kann um den Fehler aufzuspüren.

Lösung:

Aufgabe 11.4 (Liste glätten; Datei: `flatten.py`; Punkte: 2+2)

Die geglättete Version einer Liste `xs` ergibt sich wie folgt: jedes Element `x` von `xs`, das eine Liste ist, wird durch die Elemente der geglätteten Version von `x`; jedes andere Element wird direkt übernommen. Die Reihenfolge der Elemente muss dabei erhalten bleiben. *Hinweis:* Bei der Bearbeitung der folgenden Aufgaben sollen nur Hilfsmittel verwendet werden, die bisher in der Vorlesung eingeführt wurden. Lösungen, die zusätzliche Module importieren, werden nicht bewertet.

(a) Definieren Sie eine rekursive Funktion `flatten(xs: list) -> list`, die die geglättete Version der übergebenen Liste `xs` zurückgibt. Ihre Funktion soll dabei eine komplett neue Liste zurückgeben und darf die übergebene Liste bzw. alle enthaltenen Teillisten nicht verändern.

```
>>> a = [3, 4, [[5]]]
>>> b = [[[1, 2, a], (6, [7]), 8], 9, False]
>>> c = flatten(b)
>>> a == [3, 4, [[5]]]  # Teillisten bleiben unverändert
True
```

```
>>> b == [[[1, 2, [3, 4, [[5]]]], (6, [7]), 8], 9, False]
True
>>> c == [1, 2, 3, 4, 5, (6, [7]), 8, 9, False]
True
```

(b) Implementieren Sie 4 sinnvolle und voneinander verschiedene Unittests mit Namen `test_flatten_1()` bis `test_flatten_4`.

Lösung:

(a)
```
def flatten(l: list) -> list:
  if type(l) is list:
    result = []
    for e in l:
      result += flatten(e)
    return result
  else:
    return [l]
```

(b)
```
def test_flatten_1():
    """Corner case of an empty list is treated correctly"""
    assert flatten([]) == []

def test_flatten_2():
    """Corner case of nested empty lists are treated correctly."""
    assert flatten([[[]]]) == []

def test_flatten_3():
    """Flat lists remain flat."""
    assert flatten([1,2,3]) == [1,2,3]

def test_flatten_4():
    """Non-flat lists are flattened correctly."""
    assert flatten([1,[2,3],4,[5,[6,7]]]) == [1,2,3,4,5,6,7]
```

Aufgabe 11.5 (Erfahrungen; Datei: `erfahrungen.txt`; Punkte: 2)

Legen Sie im Unterverzeichnis `sheet11` eine Textdatei `erfahrungen.txt` an. Notieren Sie in dieser Datei kurz Ihre Erfahrungen beim Bearbeiten der Übungsaufgaben (Probleme, Bezug zur Vorlesung, Interessantes, benötigter Zeitaufwand, etc.).