Informatik I: Einführung in die Programmierung

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

Universität Freiburg
Institut für Informatik

Übungsblatt 9 – Lösungen

Abgabe: Dienstag, 07.01.2020, 9:00 Uhr morgens

Hinweis

Kommentieren Sie alle Klassen die Sie auf diesem und den folgenden Übungsblättern implementieren mit *docstrings*. Ein Beispiel finden Sie im Style-Guide im Abschnitt *Doc-Strings* auf der Vorlesungswebsite[1].

Hinweis

Um Ihnen bei der Umsetzung der PEP8-Richtlinien zu helfen, können Sie optional das Tool `pylint`[2] installieren und mit der Konfigurations-Datei `.pylintrc` verwenden, die Sie auf der Vorlesungswebsite finden. Die Konfigurations-Datei muss sich im selben Verzeichnis befinden wie ihre Python-Dateien, dann können Sie aus diesem Verzeichnis heraus mit `pylint *.py` ihre Python-Dateien überprüfen. Überprüft wird, dass

- die Konventionen für Variablen-Namen eingehalten wurden;
- docstrings vorhanden sind;
- keine Zeilen mit mehr als 79 Zeichen vorhanden sind;
- die Einrückung stets ein Vielfaches von 4 entspricht; und
- Leerzeichen statt Tabulatorzeichen zur Einrückung verwendet wurden.

Fehler bezüglich fehlenden Modul-*docstring*s können ignoriert werden.

Aufgabe 9.1 (Wort-Baum; Datei: `wordtree.py`; Punkte: 4+2+2+4+3+3)

In dieser Aufgabe geht es darum, eine Zeichenkette einzulesen und dabei eine Datenstruktur anzulegen, die es später erlaubt für ein gegebenes Wort zu entscheiden, ob und wie oft dieses Wort in der Zeichenkette vorkommt. Unter einem *Wort* verstehen wir im Folgenden jede endliche Folge von Buchstaben des deutschen Alphabets (also den Zeichen a, b, c, ..., z, A, B,..., Z, ä, Ä, ö, Ö, ü, Ü, ß) der Länge ≥ 1. Je zwei Wörter in der Zeichenfolge werden durch eine nicht-leere, endliche Folge von Zeichen, die nicht zu diesen Buchstaben gehören (z.B. Leerzeichen, Satzzeichen, Zeilenumbrüche), getrennt.

Es soll nun ein Suchbaum erzeugt werden, sodass jeder Knoten ein in einem String `s` vorkommendes Wort und dessen Häufigkeit repräsentiert.

[1] `https://proglang.informatik.uni-freiburg.de/teaching/info1/2019/guide/styleguide.html#doc-strings`

[2] `https://www.pylint.org/#install`

(a) Implementieren Sie zuerst die Funktion `next_word(s: str) -> tuple`, die angewendet auf einen String `s`, ein Tupel `(word, rest)` zurückgibt, wobei `word` das erste Wort (im Sinne der Spezifikation) in `s` ist und `rest` die Zeichenfolge ist, die in `s` auf `word` folgt. Beispiele:

```
>>> next_word('asdf asdf xyqr')
('asdf', ' asdf xyqr')
>>> next_word('echatsteinschalosch')
('echatsteinschalosch', '')
>>> next_word('spam&&ham  ham...')
('spam', '&&ham  ham...')
>>> next_word(' &foo @bar')
('foo', ' @bar')
>>> next_word('$$$')
(None, '')
>>> next_word('')
(None, '')
```

(b) Implementieren Sie die Klasse `Node`, welche einen Knoten im Suchbaum repräsentiert und über folgende Attribute verfügt: (1) eine Markierung `mark`, (2) die Worthäufigkeit `frequency`, (3) einen linken Teilbaum `left` und (4) einen rechten Teilbaum `right`.

(c) Implementieren Sie die Funktion `tree_str(n: Node) -> str`, die aus dem übergebenen Baum `n` einen String erzeugt und zurückgibt. Hinweis: Nutzen Sie das Attribut `__str__` der `Node` Klasse zum bequemeren Umwandeln, wie in der Vorlesung gezeigt. Beispiel:

```
>>> t = Node('spam', 3, Node('eggs', 2, None, None), None)
>>> str(t) == "Node('spam', 3, Node('eggs', 2, None, None), None)"
True
```

(d) Implementieren Sie die Funktion `word_tree(s: str) -> Node`, die aus dem übergebenen String `s` diesen Suchbaum erzeugt und zurückgibt. Als Ordnungsrelation verwenden wir Python's lexikographische Ordnung von Strings. Das heißt, ein Wort `w` wird im linken Teilbaum eines Knotens eingefügt falls der Vergleich `w < mark` den Wert `True` zurückgibt, etc. Natürlich kann Ihre Funktion selbst-definierte Hilfsfunktionen verwenden. Beispiel:

```
>>> s = "spam eggs spam eggs ham spam hamham Spam"
>>> t = Node('spam', 3, Node('eggs', 2, Node('Spam', 1, None,
             None), Node('ham', 1, None, Node('hamham', 1,
             None, None))), None)
>>> str(word_tree(s)) == str(t)
True
```

(e) Implementieren Sie die Funktion `word_freq(tree: Node, word: str) -> int`, die für einen solchen Suchbaum `tree` und ein Wort `word`, die in `tree` hinterlegte Anzahl der Wortvorkommnisse von `word` zurückgibt. Falls das Wort in dem

Baum nicht vorkommt, soll die Funktion den Wert `0` zurückgeben.

```
>>> t = word_tree("spam eggs spam")
>>> word_freq(t, 'spam'), word_freq(t, 'ham')
(2, 0)
```

(f) Definieren Sie eine Funktion `print_tree(tree: Node)`, die alle in `tree` abgelegten Wörter und die in `tree` jeweils hinterlegte Anzahl der jeweiligen Wortvorkommnisse zeilenweise (pro Zeile ein Wort und dessen Anzahl) ausgibt. Dabei soll der Baum in symmetrischer Reihenfolge (*In-Order*) traversiert werden. Beispiel:

```
>>> print_tree(word_tree("spam eggs spam alma"))
alma: 1
eggs: 1
spam: 2
```

Lösung:

(a)
```
LETTERS = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZÄÜÖäüöß"

def next_word(s: str) -> tuple:
    """Return the first word (or None) of an input string s and the rest of it.

    Args:
        s: The input string.

    Returns:
        Tuple containing the first word (str or None) and the rest of s (str).

    """
    skip = 0
    for c in s:
        if c in LETTERS:
            break
        else:
            skip += 1
    if skip == len(s):
        return None, ''

    take = 0
    for i in range(skip, len(s)):
        if s[i] in LETTERS:
            take += 1
        else:
            break
```

```
        return s[skip:skip + take], s[skip + take:]
```

(b)
```
    class Node:
        """A node of a search tree.

        Attributes:
            mark (str): A word.
            frequency (int): The frequency with which the word (mark) occurs.
            left (tree): The left subtree.
            right (tree): The right subtree.
        """

        def __init__(self, mark, frequency, left, right):
            """Set attributes to their initial values.

            Args:
                mark (str): A word.
                frequency (int): The frequency with which the word (mark) occurs.
                left (tree): The left subtree.
                right (tree): The right subtree.
            """
            self.mark = mark
            self.frequency = frequency
            self.left = left
            self.right = right
```

(c)
```
    def node_str(n: Node) -> str:
        """Return a string representation of a search node.

        Args:
            n: A search tree node.

        Returns:
            A string representing the search node.
        """
        if n is None:
            return repr(None)
        return "Node(" + repr(n.mark) + ", " + repr(n.frequency) + ", " + node_str(
            n.left) + ", " + node_str(n.right) + ")"

    Node.__str__ = node_str
```

(d)
```
    def insert(tree, item):
        """Insert an element into a search tree.

        Args:
            tree (Node or None): A search tree.
```

```
            item (str): A word to insert into the tree.

        Returns:
            A new tree (Node) containing the added item.

        """
        if tree is None:
            return Node(item, 1, None, None)
        if item == tree.mark:
            return Node(item, tree.frequency + 1, tree.left, tree.right)
        elif item < tree.mark:
            return Node(tree.mark, tree.frequency, insert(tree.left, item), tree.right)
        elif item > tree.mark:
            return Node(tree.mark, tree.frequency, tree.left, insert(tree.right, item))
        return tree

    def word_tree(s: str) -> Node:
        """Return a word tree for an input string s.

        Args:
            s (str): The input string.

        Returns:
            None or Node: The corresponding search tree.

        """
        word, rest = next_word(s)
        tree = None
        while word:
            tree = insert(tree, word)
            word, rest = next_word(rest)
        return tree
```

(e)
```
    def word_freq(tree: Node, word: str) -> int:
        """Return the occurence count of a word in a word tree.

        Args:
            tree (tree): The word tree.
            word (str): The word to search for.

        Returns:
            int: Number of word occurences.

        """
        if tree is None:
            return 0
```

```
        if word == tree.mark:
            return tree.frequency
        elif word < tree.mark:
            return word_freq(tree.left, word)
        elif word > tree.mark:
            return word_freq(tree.right, word)
```

(f)
```
def print_tree(tree: Node):
    """Print word tree in in-order.

    Args:
        tree (tree): The word tree.

    Returns:
        None

    """
    if tree is None:
        return
    print_tree(tree.left)
    print(tree.mark + ':', tree.frequency)
    print_tree(tree.right)
```

Aufgabe 9.2 (Feuerwerk; Datei: `fireworks.py`; Bonus-Punkte: 3+3+3+1)

2019 war ein Jahr mit Höhen und Tiefen. Zum Abschied wollen wir es noch einmal richtig krachen lassen. Um die Umwelt zu schonen (und nebenbei noch ein wenig Python zu lernen) natürlich rein virtuell. Angedacht ist ein Feuerwerk, dass jeden `tkinter.Canvas` zum Leuchten bringt. Um Ihnen ein wenig Arbeit abzunehmen, haben wir bereits einen Leuchtvulkan implementiert. Diesen finden Sie in der Datei `fireworks.py` auf der Vorlesungs-Website. Ihre Aufgabe besteht im Folgenden darin, das Modul um neue Klassen zu erweitern, damit weitere Typen von Feuerwerkskörpern erstellt und in die virtuelle Szenerie eingefügt werden können. Unter allen Lösungen wird eine Spezialjury das schönste und originellste Feuerwerk auswählen. Der Erschaffer darf fortan den Titel *Pythonista spectaculi pyrotechnici* tragen.

Hinweis: Die Konzepte zur Lösung dieser Aufgabe wurden noch nicht vollständig in der Vorlesung diskutiert. Daher handelt es sich um eine Bonusaufgabe, mit der Sie fehlende Punkte auf anderen Übungsblättern ausgleichen können.

Hinweis: Sie benötigen zum Ausführen des Beispielcodes das `python3-tk` Paket. Auf Ubuntu kann dieses installiert werden mit: `sudo apt install python3-tk`

(a) Leuchtvulkane haben zweifelsohne ihren eigenen Charme. Das Erreichen großer Höhen und das explosionsartige Hinterlassen leuchtender Partikel am Himmel gehören jedoch nicht dazu. Um Ihr Feuerwerk vielschichtiger zu gestalten, implementieren Sie nun eine Klasse `Rocket`. `Rocket`-Objekte sollen an einer beliebigen Stelle auf dem Canvas erzeugt werden können, mit einer bestimmten

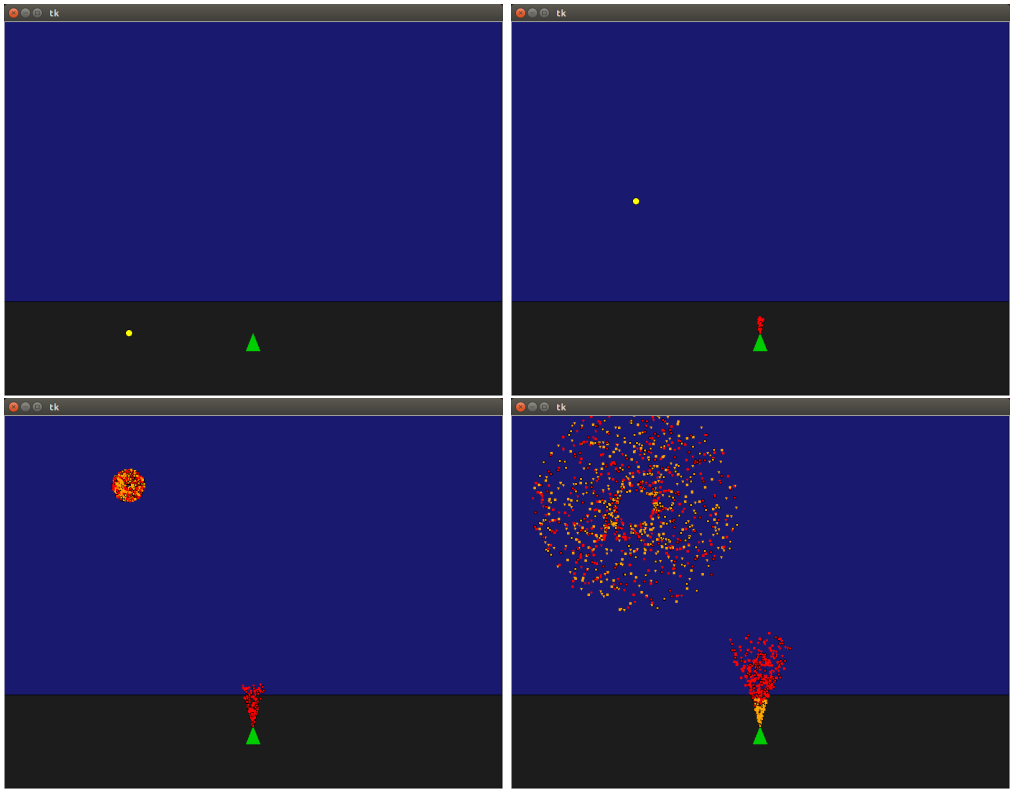

Abbildung 1:

Geschwindigkeit aufsteigen, und nach einer bestimmten Flugzeit explodieren. Dabei soll ein beliebiges Muster aus Partikeln entstehen, welches sich nach einer vorgegebenen Lebensspanne wieder auflöst. Siehe Abbildung **??**.

(b) Implementieren Sie eine Klasse `RocketLauncher`. Instanzen dieser Klasse sollen in variablen Abständen `Rocket`-Objekte erzeugen und in den Himmel schießen. Variieren Sie dabei Abschusswinkel, Geschwindigkeit, und Explosion der `Rocket`-Objekte um den Effekt zu verstärken.

(c) Überlegen Sie sich wenigstens einen weiteren Typ an Feuerwerkskörpern, und implementieren Sie diesen. Vergessen Sie nicht im Docstring die beabsichtigte Wirkung zu beschreiben.

(d) Nutzen Sie Ihre neu erworbenen Pyro-Fähigkeiten zum Inszenieren des ultimativen Feuerwerks zum Jahreswechsel 2019/2020. Erstellen Sie dazu Instanzen der zuvor implementierten Feuerwerks-Klassen und zünden diese auf dem `tkinter.Canvas`.

Aufgabe 9.3 (Erfahrungen; Datei: `erfahrungen.txt`; Punkte: 2)

Legen Sie im Unterverzeichnis `sheet09` 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.).