Aufgabenblatt 4

Allgemeine Informationen zum Aufgabenblatt:

- Die Abgabe erfolgt in TUWEL. Bitte laden Sie Ihre Python-Datei bis spätestens **Freitag, 01.05.2020 20:00 Uhr** in TUWEL hoch.
- Beachten Sie bitte folgende Punkte
  - Ihre Programme müssen ausführbar sein, d.h. einzelne Programme sollten z.B. keine Syntaxfehler oder Typfehler enthalten.
  - Bei kleinen logischen Fehlern (falsche Ergebnisse) werden wir keine Punkte abziehen. Daher sollten Sie immer versuchen, alle Aufgaben vollständig abzugeben.
  - Ihre Programme sollten nur Konstrukte verwenden, die bisher in der Vorlesung vorgekommen sind. Sie sollten daher keine speziellen Aufrufe verwenden, die möglicherweise einzelne Aufgaben abkürzen. Bitte beachten Sie auch die weiteren Einschränkungen bei bestimmten Unteraufgaben.
  - Die Angabedatei muss nicht unbenannt werden.

In diesem Aufgabenblatt werden folgende Themen behandelt:

- Rekursion
- Tupel
- Listen

Aufgabe 1 (1 Punkt)

Gilt für alle zu implementierenden Funktionen: Sie dürfen keine globalen Variablen oder zusätzliche eigene Hilfsfunktionen verwenden. Die vorgegebenen Funktionsköpfe dürfen nicht erweitert oder geändert werden. Für die Implementierung der Aufgabenstellung dürfen keine Schleifen verwendet werden.
Bei jeder Funktion finden Sie eine Beschreibung von Annahmen. Sie können immer davon ausgehen, dass diese Annahmen zutreffen und müssen keine spezielle Behandlung für die übergebenen Argumentwerte implementieren (z.B. überprüfen ob ein Wert > 0 ist). Jede Funktion sollte auch zumindest mit den gegebenen Beispielen getestet werden.
Folgende **rekursive** Funktionen sind zu implementieren:

1. Eine Funktion `power(a, b)`, die die Operation a^b nur mit Multiplikation nachbildet (Potenzieren als wiederholtes Multiplizieren) und das Ergebnis zurückliefert.
   Annahme(n): a und b sind ganze Zahlen, $a >= 0, b > 0$.
   Beispiele:
   `power(1,1)` liefert 1 zurück
   `power(2,2)` liefert 4 zurück
   `power(3,2)` liefert 9 zurück
   `power(4,3)` liefert 64 zurück

2. Eine Funktion `sum_digits(n)`, die die einzelnen Ziffern einer nicht negativen Zahl aufsummiert und diese Summe zurückliefert.
   Annahme(n): n ist eine ganze Zahl, $n >= 0$.
   Beispiele:
   `sum_digits(0)` liefert 0 zurück
   `sum_digits(5)` liefert 5 zurück
   `sum_digits(123)` liefert 6 zurück
   `sum_digits(141683)` liefert 23 zurück

3. Eine Funktion `remove_spaces(text)`, die alle Leerzeichen aus dem String `text` entfernt und den veränderten String zurückliefert.
   Annahme(n): `text` ist ein String.
   Zusätzliche Einschränkung: Sie dürfen für die Bearbeitung des Strings nur Konkatenation und Slicing verwenden.
   Beispiele:
   `remove_spaces("")` liefert den Leerstring zurück
   `remove_spaces("Hello")` liefert `Hello` zurück
   `remove_spaces("This is a test")` liefert `Thisisatest` zurück
   `remove_spaces(" This is a test ")` liefert `Thisisatest` zurück

Aufgabe 2 (1 Punkt)

In dieser Aufgabe geht es darum, dass Sie sich mit Listen und Tupel vertraut machen. In dieser Aufgabe müssen keine Funktionen implementiert werden. Implementieren Sie nacheinander folgende Listenbeispiele:

1. Legen Sie eine Liste `list_1` mit folgendem Inhalt an: `[1, 5, 3, 7, 2, 9]`

   - Geben Sie das erste und letzte Element aus. Ausgabe lautet: 1 9
   - Geben Sie die linke Hälfte als Liste aus. Ausgabe lautet: [1, 5, 3]
   - Geben Sie die rechte Hälfte als Liste aus. Ausgabe lautet: [7, 2, 9]

2. Legen Sie eine Liste `list_2` mit folgendem Inhalt an: `["Amsterdam", "Berlin", "London", "Madrid", "Paris", "Stockholm", "Wien"]`

   - Erzeugen Sie eine neue Liste `list_21`, die alle Städte enthält, deren Namen mehr als 6 Buchstaben haben: Inhalt von `list_21` entspricht daher `['Amsterdam', 'Stockholm']`
   - Erzeugen Sie eine neue Liste `list_22`, die alle Städte enthält, deren Namen auf den Buchstaben `n` enden: Inhalt von `list_22` entspricht daher `['Berlin', 'London', 'Wien']`

   Sie dürfen selbst entscheiden, ob Sie eine Schleife benutzen, oder mit List-Comprehension arbeiten.

3. Legen Sie zwei Listen `list_3` und `list_4` mit folgendem Inhalt an:
   `list_3 = ["Kreuz", "Pik", "Herz", "Karo"]`
   `list_4 = ["Ass", "Koenig", "Dame", "Bube", "10", "9", "8", "7", "6", "5", "4", "3", "2"]`
   Erzeugen Sie eine neue Liste `list_5` die alle Kombinationen (52 Kombinationen) der Elemente der Listen `list_3` und `list_4` jeweils als Tupel enthält. Erzeugen Sie dann eine weitere Liste `poker_hand`, die 5 Karten aus `list_5` enthält. Greifen Sie dabei auf die Positionen 5, 10, 15, 20 und 25 zu. Der Inhalt der Liste `poker_hand` ist dann: `[('Kreuz', '9'), ('Kreuz', '4'), ('Pik', 'Dame'), ('Pik', '7'), ('Pik', '2')]`

Aufgabe 3 (1 Punkt)

Bei jeder Funktion finden Sie eine Beschreibung von Annahmen. Sie können immer davon ausgehen, dass diese Annahmen zutreffen und müssen keine spezielle Behandlung für die übergebenen Argumentwerte implementieren. Jede Funktion sollte auch zumindest mit den gegebenen Beispielen getestet werden.
Folgende Funktionen sind zu implementieren:

1. Eine Funktion `min_max(list)`, die eine Liste zurückgibt, die das Minimum und das Maximum der Liste `list` enthält.
   Annahme(n): `list` ist eine Liste entweder mit ganzen Zahlen oder mit Strings.
   Beispiele:
   `min_max([3, 1, 5, 7, 2, 8])` liefert die Liste `[1, 8]` zurück.
   `min_max(["London", "Berlin", "Paris", "Rom", "Wien"])` liefert die Liste `['Berlin', 'Wien']` zurück.

2. Eine Funktion `split_list(list, threshold)`, die ein Tupel mit zwei Listen zurückgibt. Die erste Liste enthält alle Elemente aus `list`, die kleiner als der Schwellenwert `threshold` sind. Die zweite Liste enthält die restlichen Elemente.
   Annahme(n): `list` ist eine Liste mit ganzen Zahlen, `threshold` ist eine ganze Zahl.
   Beispiele:
   `split_list([3, 8, 5, 1, 9], 5)` liefert das Tupel `([3, 1], [8, 5, 9])` zurück.
   `split_list([3, 8, 5, 1, 9], 1)` liefert das Tupel `([], [3, 8, 5, 1, 9])` zurück.
   `split_list([3, 8, 5, 1, 9], 10)` liefert das Tupel `([3, 8, 5, 1, 9], [])` zurück.
   `split_list([], 5)` liefert das Tupel `([], [])` zurück.

3. Eine Funktion `combine_lists(list_1, list_2)`, die eine neue Liste erzeugt und zurückgibt, bei der die Elemente der beiden übergebenen Listen abwechselnd vorkommen. Ist eine Liste länger, dann werden die restlichen Elemente dieser Liste an das Ende der zurückgegebenen Liste angehängt.
   Annahme(n): `list_1` und `list_2` sind Listen.
   Beispiele:
   `combine_lists([1, 3], [2, 4])` liefert die Liste `[1, 2, 3, 4]` zurück.
   `combine_lists([2, 4], [10, 20, 30, 40])` liefert die Liste `[2, 10, 4, 20, 30, 40]` zurück.
   `combine_lists([10, 20, 30], [1, 3])` liefert die Liste `[10, 1, 20, 3, 30])` zurück.
   `combine_lists([], [1, 3])` liefert die Liste `[1, 3]` zurück.
   `combine_lists([], [])` liefert die Liste `[]` zurück.

Aufgabe 4 (1 Punkt)

Implementieren Sie folgende drei Funktionen:

1. Eine Funktion `generate_landscape(n, p)`, die eine Fläche mit $n \times n$ Feldern generiert und zurückgibt. Die Fläche wird mit einer Liste von Listen repräsentiert. Mit einer Wahrscheinlichkeit $1 - p$ wird ein einzelnes Feld eine Grassteppe (dargestellt durch `"."`). Mit einer Wahrscheinlichkeit p wird ein Feld ein Hindernis. Ein Hindernis kann entweder mit einer Wahrscheinlichkeit von 0.5 ein Busch (dargestellt durch `"#"`) oder mit einer Wahrscheinlichkeit von 0.5 ein Felsen (dargestellt durch `"@"`) sein.
   Annahme(n): $n > 0$ und ganzzahlig, $0 <= p < 1$.
   Hinweis: Zufallszahlen im Bereich [0.0, 1.0) können mit `random.random()` erzeugt werden.

2. Eine Funktion `print_landscape(landscape)`, die eine Fläche mit $n \times n$ Feldern zeilenweise auf der Konsole ausgibt. Es werden also n Zeilen mit n Einträgen ausgegeben, wobei in einer Zeile die Einträge durch ein Leerzeichen getrennt werden.
   Annahme(n): `landscape` ist eine Liste von Listen, die eine Fläche mit $n \times n$ Feldern repräsentiert.

3. Eine Funktion `mark_paths(landscape)`, die gerade Pfade in `landscape` einfügt. Ein Pfad wird eingefügt, wenn in einer Zeile nur Einträge vom Typ Grassteppe (`"."`) vorhanden sind. Dann wird jeder Eintrag in der Zeile jeweils durch das Zeichen `"*"` ersetzt.
   Annahme(n): `landscape` ist eine Liste von Listen, die eine Fläche mit $n \times n$ Feldern repräsentiert.

Ein mögliches Programm zum Testen:

```
l = generate_landscape(5, 0.2)
print_landscape(l)
mark_paths(l)
print()
print_landscape(l)
```

Mögliche Ausgabe auf der Konsole (der erste Teil ist die generierte Landschaft, der zweite Teil zeigt die Landschaft mit den eingezeichneten zwei möglichen Pfaden)

```
. . . . .
. . . @ .
. . . . .
# . # . .
# . . . .

* * * * *
. . . @ .
* * * * *
# . # . .
# . . . .
```

Eine weitere mögliche Ausgabe (mit einem Pfad):

```
. # . @ .
. . . # .
@ . . . @
. @ . . .
. . . . .

. # . @ .
. . . # .
@ . . . @
. @ . . .
* * * * *
```

Eine weitere mögliche Ausgabe (hier wurde `generate_landscape(10, 0.2)` verwendet und kein Pfad gefunden):

```
. # . . @ . . . . .
. . . . . . . . . #
# . . . . @ @ . . #
. @ . . . . . . . .
. . . . . # . . . #
# . . . . . . @ . #
. . . . . . . # . .
@ . . . . . . . . .
. @ @ . . # # . . .
. . . @ . . # . @ #

. # . . @ . . . . .
. . . . . . . . . #
# . . . . @ @ . . #
. @ . . . . . . . .
. . . . . # . . . #
# . . . . . . @ . #
. . . . . . . # . .
@ . . . . . . . . .
. @ @ . . # # . . .
. . . @ . . # . @ #
```