Übung 01

generate_walk

Lösungsidee

Für die Bestimmung einer zufälligen Richtung wurde die random Bibliothek benutzt.

Code

```
directions = ('N', 'E', 'S', 'W')

def generate_walk(blocks = 1):
    """
        Generates a list of directions
        blocks: the length of the list
    """
    walk = list()
    for x in range(blocks):
        walk.append(rand.choice(directions))

    return walk
```

Tests

```
print(generate_walk())
print(generate_walk(5))
print(generate_walk(10))
print(generate_walk(15))
print(generate_walk(0))
print(generate_walk(-1))
print(generate_walk(-15))

['N']
['W', 'S', 'W', 'N', 'N']
['S', 'N', 'W', 'N', 'N', 'N', 'E', 'S', 'E', 'S']
['S', 'S', 'W', 'N', 'E', 'W', 'W', 'S', 'S', 'W', 'W', 'N', 'S', 'S', 'E']
[]
[]
[]
```

Werden negative Zahlen an die Funktion übergeben, wird ein leerer Walk zurückgegeben.

decode_walk

Lösungsidee

Die Funktion geht davon aus, dass immer bei Punkt (0, 0) gestartet wird. Je nach Richtung werden zwei Integer für die zwei Achsen erhöht oder verringert.

Code

```
def decode_walk(walk):
    """
        Decodes a walk and returns a tuple of int describing the position on a
        grid. Always starts at (0, 0)
        walk: a list of directions (see generate_walk function)
    """
    x = 0
    y = 0
    for d in walk:
        if (d == 'N'):
            y += 1
        elif (d == 'E'):
            x += 1
        elif (d == 'S'):
            y -= 1
        elif (d == 'W'):
            x -= 1

    return (x, y)
```

Tests

Hierfür wurden die Walks der oberen Testfälle verwendet.

```
print(decode_walk(['N']))
print(decode_walk(['W', 'S', 'W', 'N', 'N']))
print(decode_walk(['S', 'N', 'W', 'N', 'N', 'N', 'E', 'S', 'E', 'S']))
print(decode_walk(['S', 'S', 'W', 'N', 'E', 'W', 'W', 'S', 'S', 'W', 'W', 'N',
'S', 'S', 'E']))
print(decode_walk([]))

(0, 1)
(-2, 1)
(1, 1)
(-3, -4)
(0, 0)
```

distance_manhattan

Lösungsidee

Diese Funktion implementiert die Formel beschrieben auf Wikipedia

Code

```
def distance_manhattan(start, end):
    """
        Calculates the manhattan distance between two points on a grid
        start: a tuple of integers e.g. (0, 0)
        end: a tuple of integers e.g. (2, 4)
    """
    return sum(abs(x - y) for x, y in zip(start, end))
```

Tests

Hier wurden wieder die Ergebnisse der vorherigen Tests verwendet.

```
print(distance_manhattan((0, 0), (0, 1)))
print(distance_manhattan((0, 0), (-2, 1)))
print(distance_manhattan((0, 0), (1, 1)))
print(distance_manhattan((0, 0), (-3, -4)))
print(distance_manhattan((0, 0), (0, 0)))

1
3
2
7
0
```

do_walk

Lösungsidee

Eine Helferfunktion, um den Code der folgenden Funktion einfacher lesbar zu machen. Hier wird ein walk generiert und mit der Manhattan-Distanz in einem Tupel zurückgegeben.

monte_carle_walk_analysis

Lösungsidee

Für jede Blocklänge von 1 bis *max_blocks* werden *repetition* viele Walks generiert und mit der Manhattan-Distanz in ein Dictionary gespeichert.

Code

```
def monte_carlo_walk_analysis(max_blocks, repetitions = 10000):
    """
        Generates repetitions amounts of walks for each stepsize from 1 to
max_blocks and saves them in a dictionary
    """
    all_walks = dict()
    for length in range(1, max_blocks + 1):
        walks = list()
        for rep in range(repetitions):
            walks.append(do_walk(length))
        all_walks[length] = walks

    return all_walks
```

Tests

```
print(monte_carlo_walk_analysis(1, 1))
print(monte_carlo_walk_analysis(3, 1))
print(monte_carlo_walk_analysis(1, 3))
print(monte_carlo_walk_analysis(3, 5))
```

```
{1: [(['N'], 1)]}

{1: [(['S'], 1)],
 2: [(['E', 'N'], 2)],
 3: [(['W', 'N', 'E'], 1)]}

{1: [(['W'], 1), (['W'], 1), (['N'], 1)]}

{1: [(['S'], 1), (['E'], 1), (['E'], 1), (['S'], 1), (['E'], 1)],
 2: [(['E', 'S'], 2), (['S', 'N'], 0), (['W', 'W'], 2), (['W', 'E'], 0),
     (['N', 'E'], 2)],
 3: [(['W', 'S', 'S'], 3), (['S', 'N', 'W'], 1), (['N', 'W', 'S'], 1),
     (['S', 'E', 'S'], 3), (['N', 'S', 'W'], 1)]}
```