General

Sets and dictionaries in doctests

The elements of a set and the keys of a dictionary do not have a particular order. This means that two sets or two dictionaries are equal, irrespective of the order in which the elements/keys have been added to the set/dictionary.

```
>>> {1, 3, 2, 4} == {4, 3, 1, 2}
True
>>> {'A': 1, 'B': 2, 'C': 3} == {'B': 2, 'A': 1, 'C': 3}
True
```

When working with doctests, however, sets and dictionaries might cause you trouble. The reason for this problem is that in comparing expected and generated results, doctests proceed as follows: the expected result is extracted from the doctest as a string, and the value returned by a function or resulting from the evaluation of an expression is converted to a string. These two strings are then compared with each other (as a string, not as a set or a dictionary). In comparing strings, the order of the characters is important, and that's what's causing the trouble.

```
>>> d1 = {'A': 1, 'B': 2, 'C': 3}
>>> s1 = str(d1)   # executed on computer 1
>>> d1
"{'A': 1, 'C': 3, 'B': 2}"
>>> d2 = {'A': 1, 'B': 2, 'C': 3}
>>> s2 = str(d2)   # executed o computer 2
"{'A': 1, 'B': 2, 'C': 3}"
>>> d1 == d2
True
>>> s1 == s2
False
```

Although dictionaries `d1` and `d2` have the same value, the doctest indicates that the string representations of these two dictionaries are different. The conversion of a set/dictionary to a string may depend on the computer that executes the code, the Python version used on that computer, and might even differ if you do repeat the same conversion on the same computer using the same Python version. The problem can be solved by making sure the doctests do not compare strings, but directly compare sets or dictionaries. For example, if a doctest initially looks like

```
>>> aFunction(parameter1, parameter2)
{'A' : 1, 'B': 2, 'C': 3}
```

you may better rewrite this doctest as

```
>>> aFunction(parameter1, parameter2) == {'A' : 1, 'B': 2, 'C': 3}
True
```

This problem never occurs on Dodona, since its way of testing the source code never converts return values or results of expression evaluations into strings, but directly compares the resulting objects.

Isomers

General information

Assign values to keys in a dictionary

When associating a value with a dictionary key, you may have to take into account that the dictionary may or may not already associate a value to this key. It is often the case that you have to add a new key/value pair in case the key was not used in the dictionary, and thay you have to update an existing key/value pair in case the key was already used in the dictionary.

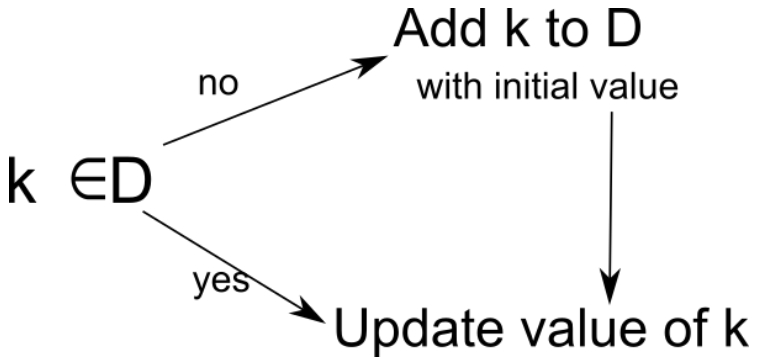

Figure 1: update dictionary

This technique can to be used, for example, for the construction of frequency tables. A frequency table is a dictionary that maps each key onto an integer that indicates how often the key occurs in a container object (e.g. a list, a tuple or a set).

```
>>> aList = ['R', 'S', 'E', 'E', 'N', 'T', 'E', 'I', 'L', 'D', 'I']
>>> frequentietabel(aList)
{'E': 3, 'S': 1, 'D': 1, 'N': 1, 'T': 1, 'R': 1, 'L': 1, 'I': 2}
```

The above technique can be used to implement the function `frequencyTable`.

```
def frequencyTable(aList):
    freq = {}                    # create empty dictionary
    for element in aList:
        if element not in freq:
            freq[element] = 0    # add element to dictionary with initial value 0
        freq[element] += 1       # update value associated with element
```

In this case you could have also used the dictionary method `get`. This method returns the value associated with a given key in the dictionary. In contrast to indexing dictionaries using square brackets to fetch the value associated with a given key, the `get` method will never result in a `KeyError` in case the key does not occur in the dictionary. Instead, the `get` method by default returns the value `None` if the key does not occur in the dictionary. If you pass a value to the second optional parameter, this value will be returned as the default value in case the `get` method does not find the key in the dictionary.

```
>>> d = {'A': 1, 'B': 2, 'C': 3}
>>> d['A']
1
>>> d.get('A')
1
>>> d['D']
Traceback (most recent call last):
```

```
KeyError: 'D'
>>> d.get('D')                    # returns the value None
>>> d.get('D', 0)
0
```

Adjacent numbers

General information

Grouping the elements of a list

Python offers multiple solutions for grouping the elements in a list into group of n elements. For example, you may use a list comprehension to solve this problem.

```
>>> aList = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> [(aList[i], aList[i + 1]) for i in range(0, len(aList), 2)]
[('a', 'b'), ('c', 'd'), ('e', 'f'), ('g', 'h')]
```

You may also use the built-in function `zip` that simultaneously iterates two or more iterable objects. Each iteration step, the function returns a tuple containing the i-th elements of the iterable objects that are passed to it.

If you simultaneously iterate over the list containing all elements at even positions (aList[::2]) and the list containing all elements at odd positions (aList[1::2]), you obtain exactly the same result.

```
>>> aList = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> zip(aList[::2], aList[1::2])
[('a', 'b'), ('c', 'd'), ('e', 'f'), ('g', 'h')]
```

Relative positions of the cells in a grid

When processing cells that are arranged in a grid, you often need to process the environment of a given cell: for example, the four cells that share a side with the given cell. You can do this in a naive way:

```
>>> grid = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> total_sum = 0
>>> for r, row in enumerate(grid):
...     for k, element in enumerate(row):
...         total_sum = grid[r - 1][k]      # top neighbor
...         total_sum += grid[r][k + 1]     # right neighbor
...         total_sum += grid[r + 1][k]     # bottom neighbor
...         total_sum += grid[r][k - 1]     # left neighbor
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
IndexError: list index out of range
```

As you can see, this does not work because we never check to see if all four neighbors of the current cell exist. After all, cells at the edge of the grid only have two or three neighboring cells. If we would explicitly check if each of these four neighbors exists, we would get quite repetitive code. Something we definitely would like to avoid. However, there is a more generic approach.

You can use a container (a set for example) to keep track of the relative coordinates of the neighboring cells. In the above example, these are

- $(-1, 0)$ for the top neighbor (row index is decremented by 1, column index is the same)
- $(0, 1)$ voor de rechterbuur (row index is the same, column index is incremented by 1)
- $(1, 0)$ voor de onderbuur (row index is incremented by 1, column index is the same)
- $(0, -1)$ voor de linkerbuur (row index is the same, column index is decremented by 1)

When we make use of this trick, we can rewrite the above code snippet in the following way:

```
>>> grid = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> total_sum = 0
>>> neighbors = {(-1, 0), (0, 1), (1, 0), (0, -1)}  # relative coordinates of neighbors
>>> for r, row in enumerate(grid):
...     for c, element in enumerate(row):
...         for dr, dc in neighbors:
...             # check if neighboring cell is within grid
...             if 0 <= r + dr < len(grid) and 0 <= c + dc < len(row):
...                 total_sum += grid[r + dr][c + dc]
...
>>> total_sum
120
```

Equidivision

Specific information

You can make use of a *flood fill* algorithm for the implementation of the function `connected`.

We start by randomly chosing an element e from the container S and putting that element in a list L. Here, L represents the list of all elements that we have labeled as reachable, but whose neighbors may not have been labeled as such. Next, we repeat the following procedure until L is empty.

- randomly choose an element f from L
- for all non-labeled neighbors b of f in S
  - add b to L
  - label b as reachable
- remove f from L

If L is empty, all elements in S that are reachable from the element e will be labeled. If there still are elements in S that have not been labeled as reachable, the container is not connected.

Tip:

- instead of labeling an element in S, the element can also be removed from S; in that case, you should preferably use a set instead of a list as the data structure for L

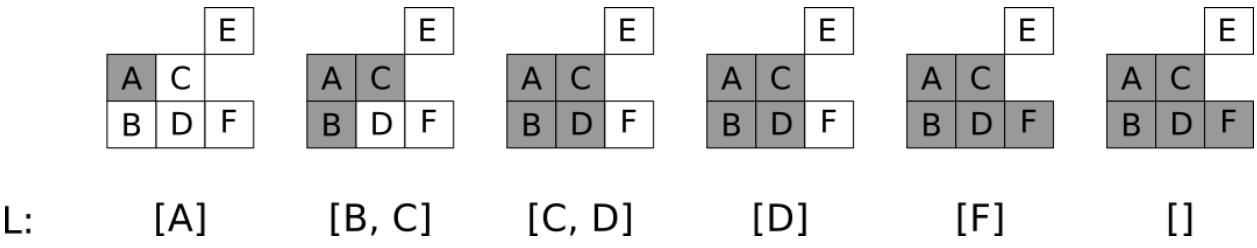

Figure 2: Flood fill