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 output, doctests proceed as follows: the expected output 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 into 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
```

Altough the dictionaries `d1` and `d2` have the same value, the doctests indicates that the two string representations of these two dictionaries are different. The conversion of a set/dictionary of a string may depend on the computer that executes the code or de Python version used on that computer. 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

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

Pythia does not suffer with the same problem, 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.

Rummikub

General information

Assign values to keys in a dictionary

When associate a value to a key of a dictionary, you may have to take into account whether or not the dictionary already associates 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.

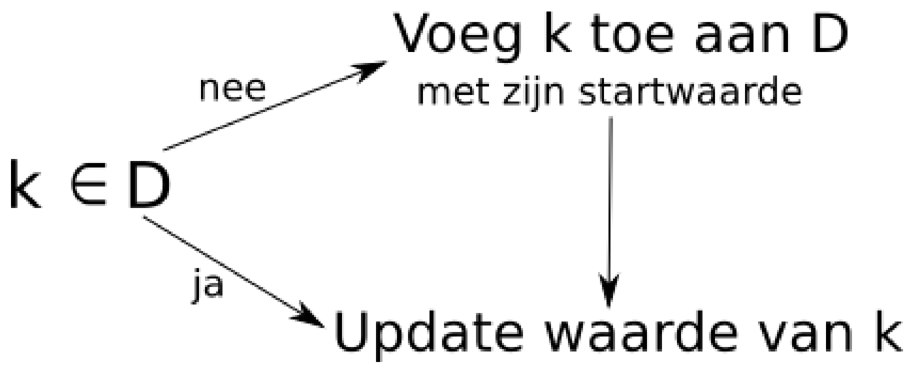

Figure 1: updaten dictionary

This technique has 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 = dict() # 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
...
>>> frequencyTable(['R', 'S', 'E', 'E', 'N', 'T', 'E', 'I', 'L', 'D', 'I'])
{'E': 3, 'S': 1, 'D': 1, 'N': 1, 'T': 1, 'R': 1, 'L': 1, 'I': 2}
```

In this case you could also have used the dictionary `get` method. 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 will return the value `None` by default. 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
```

Consensus sequence

General information

Remarks

If you want to create a list with a fixed size `n` whose elements all have the same value `x`, you don't need to use a `for`-loop or a *list comprehension*. The simple solution is to write `[x] * n`.

```
>>> [' '] * 3
[' ', ' ', ' ']
>>> [1] * 5
[1, 1, 1, 1, 1]
```

Not that this multiplication does not make copies of the object `x`, but results in a list whose elements all point to the same object `x`. This is definitely important in case `x` is a *multable* object.

```
>>> aList = [[1, 2]] * 4
>>> aList
[[1, 2], [1, 2], [1, 2], [1, 2]]
>>> aList[0][1] = 666
>>> aList
[[1, 666], [1, 666], [1, 666], [1, 666]]
>>> aList[3].append(42)
>>> aList
[[1, 666, 42], [1, 666, 42], [1, 666, 42], [1, 666, 42]]
```

Solar system

General information

Sort based on optional parameter `key`

The list method `sort` and the built-in function `sorted` can both be used to sort the elements of a given list. They differ in the fact that the `sort` method rearranges the elements of the list *in place*, whereas the function `sorted` returns a new sorted list, while leaving the orginal list unchanged.

Apart from this difference, both functions have many things in common. They both have an optional parameter `reverse` that takes a Boolean value. The value indicates whether the elements have to be sorted in increasing (value `False`, the default value) or decreasing (value `True`) order. Both functions also have a second optional parameter `key` that can be used to determine the order of the elements. This ordering of the elements will be used when sorting the list.

The parameter `key` takes a function as its argument. This function must take a single argument. In case a function `f` is passed to the parameter `key`, the order of the elements is not determined by the elements themselves, as is the default behaviour, but is based on the values returned by the function `f` for each of the element (each element is this passed individually as an argument to the function `f`).

Say, for example, that you have defined a function `f` and that you pass this function to the parameter `key`. Before the actual sorting takes place, a function call `f(element)` is done for each `element` in the list that needs to be sorted. Afterwards, the elements of the list are sorted based on the values returned by the function `f` for each of the elements in the list. At the first position in the sorted list you will find the `element` that results in the smallest value for `f(element)` (or the largest value in case `reverse=True`), and at the last position in the sorted list you will find the `element` that results in the largest value for `f(element)` (or the smallest value in case `reverse=True`).

The natural order in which tuples are sorted is to sort the tuples first based on their first elements, and in case these elements have equal values sort them further based on successive elements in the tuple. Say, for example, that we have a list of tuples, where each tuple contains two integers. The natural ordering of these tuples results in the following outcome.

```
>>> aList = [(2, 7), (0, 10), (4, 0), (1, 6), (2, 5), (2, 6)]
>>> sorted(aList)
[(0, 10), (1, 6), (2, 5), (2, 6), (2, 7), (4, 0)]
```

If we wanted to sort the tuples first on their second element, and then on their first element, we could do this in the following way.

```
>>> def sortkey(pair):
...     return pair[1], pair[0]
...
>>> aList = [(2, 7), (0, 10), (4, 0), (1, 6), (2, 5), (2, 6)]
>>> sorted(aList, key=sortkey)
[(4, 0), (2, 5), (1, 6), (2, 6), (2, 7), (0, 10)]
```

Please not that this ordering is not the same as the reverse natural ordering of the elements.

Specific information

In Python you can define a function inside another function. As a result, the inner function is only visible inside the scope of the outer function. In addition, the inner function also has access to the local variables of the outer function.

In the following code fragment we define the function `g` inside the function `f`. As a result, the function `g` also has access to the variable `aList` that has been defined within the scope of the function `f`.

```
>>> def f():
...
...     aList = ['a']
...
...     def g():
```

```
...         print(aList)   # access variabele from outer scope
...
...     g()
...     aList.append('b')
...     g()
...
>>> f()
['a']
['a', 'b']
```

For this assignment you can make use of this technique to define a function that is passed to the parameter `key` of a sort function (`sort` or `sorted`). This function has the restriction that it will be called with a single argument (an element of the list), while it might also need access to other information to compute the sorting value of that argument.

```
>>> def sortkey(value, order):
...     return order.index(value)
...
>>> def arrange(aList, order):
...     return sorted(aList, key=sortkey)
...
>>> arrange(['a', 'b', 'c'], 'cbad')
TypeError: sortkey() missing 1 required positional argument: 'order'
```

The problem here is that the built-in function `sorted` calls the function `sortkey` that is passed to its parameter `key` with a single argument (an element of the list), whereas the function `sortkey` expects two arguments. A possible solution is to define the function `sortkey` within the function `arrange`.

```
>>> def arrange(aList, order):
...
...     def sortkey(value):
...         return order.index(value) # use variable order from outer scope
...
...     return sorted(aList, key=sortkey)
...
>>> arrange(['a', 'b', 'c'], 'cbad')
['c', 'b', 'a']
```

Because the variable `order` is a local variable of the function `arrange` (parameters are local variables too), there's no longer a need to pass the value of that variable to the function `sortkey`. Since `sortkey` has been defined inside the function `arrange`, it has direct access to the value of the variable `order`.