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 repeat the same conversion on the same computer using the same Python version. The problem can be solved by making sure doctests do not compare strings, but directly compare sets or dictionaries. For example, if a doctest initially looks like

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

you may better rewrite this doctest as

```
>>> some_function(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.

Variable number of keyword arguments

Sometimes you won't like to fix the arguments of a function beforehand. The use of a double star as an operator in front of a function parameter allows you to accomplish this.

```
>>> def some_function(**kwargs):
...     for key in kwargs:
...         print(f"The parameter '{key}' has value '{kwargs[key]}'.")
...
```

```
>>> some_function(name='Tim', age=8, location='Ghent')
The parameter 'name' has value 'Tim'.
The parameter 'age' has value '8'.
The parameter 'location' has value 'Ghent'.
```

The variable `kwargs` (without the double star) now is a dictionary that maps parameter names onto their corresponding values.

Sometimes it's also handy not to pass a sequence of keyword arguments when calling a function, but instead wrap those arguments in a dictionary whose keys correspond to the parameter names.

```
>>> def some_function(name, age, location):
...     print(f'{name} is {age} years old and lives in {location}.')
>>> kwargs = {'name': 'Tim', 'age': 8, 'location': 'Ghent'}
>>> some_function(**kwargs)
Tim is 8 years old and lives in Ghent.
```

Set operators

In Python it is really easy to compute the union, the intersection and the difference of two sets.

```
>>> set1 = {'A', 'B', 'C'}
>>> set2 = {'C', 'D', 'E'}
>>> set1 & set2               # intersection
{'C'}
>>> set1 | set2               # union
{'A', 'B', 'C', 'D', 'E'}
>>> set1 - set2               # difference
{'A', 'B'}
>>> set2 - set1               # difference is asymmetric
{'D', 'E'}
```

Removing an element from a set

Use the method `set.pop()` to remove an unspecified element from a set. This methode will remove and return an element from a set. Note that it is not known beforehand which element this methode will return. But the element will not be chosen truly random.

```
>>> some_set = {'ABC', 'abc', '123'}
>>> some_set.pop()
'abc'
>>> some_set
{'123', 'ABC'}
```

Use the methode `set.remove()` when you want to remove a particular element from a set.

```
>>> some_set = {'ABC', 'abc', '123'}
>>> some_set.remove('123')
>>> some_set
{'abc', 'ABC'}
```

Autogram

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.

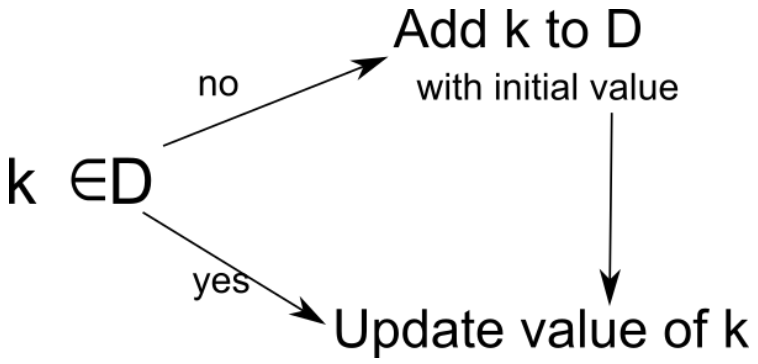

Figure 1: update dictionary

You can use this technique, for example, to construct 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).

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

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

```
def frequency_table(some_list):
    freq = {}                    # create empty dictionary
    for element in some_list:
        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 also have used the method `dict.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 method `dict.get()` will never result in a `KeyError` in case the key does not occur in the dictionary. Instead, the method `dict.get()` 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 method `dict.get()` 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
```

Readouts

Using dictionaries to avoid `if` statements

Sometimes you might need extremely long `if-elif-else` statements where each clause does exactly the same thing, except for a different value of one variable. In these cases is it usually better to look up the value of that variable in a dictionary, and work with its associated value.

```
>>> if x == 'A':
...     y += 3
... elif x == 'B':
...     y += 1
... elif x == 'C':
...     y += 7
```

This code snippet can be written must shorter by using a dictionary, whose keys are the possible values of `x` and whose corresponding values are the values that need to be added to `y`.

```
>>> increment = {'A': 3, 'B': 1, 'C': 7}
>>> y += increment[x]
```