

Bread crumbs

Een element uit een set halen

To remove an unspecified element from a set, the `.pop()` method can be used. This function will remove and return an element from a set. Note that it is not known beforehand which element this function will return. But it will not be truly random.

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

When you want to remove the a certain element from a set, then it can be done with `.remove()`.

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

Specific information

We will expand on the described algorithm by giving some explanation on each step.

1. initialization: $\mathcal{B} = \emptyset, \mathcal{R} = \{(0,0)\}$

In Python we can use the following code fragment:

```
B = set()
R = {(0,0)}
```

2. remove a random point p from \mathcal{R}

For this the `.pop()` function, as described in these tips and tricks, can come in handy.

3. add p to \mathcal{B}

For this the `.add()` function is your friend.

4. add all neighbors of p to \mathcal{R} that are reachable and are not already in \mathcal{B}

Your own function `neighbors(...)` can be used. For each element in the result, you want to check if it is contained in \mathcal{B} . If not, add this neighbor to \mathcal{R} .

5. repeat from step 2 until $\mathcal{R} = \emptyset$

In Python this means a `while`-loop around steps 2 to 4.

Sacred Cat of Burma

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 that you have to update an existing key/value pair in case the key was already used in the dictionary.

This technique can 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).

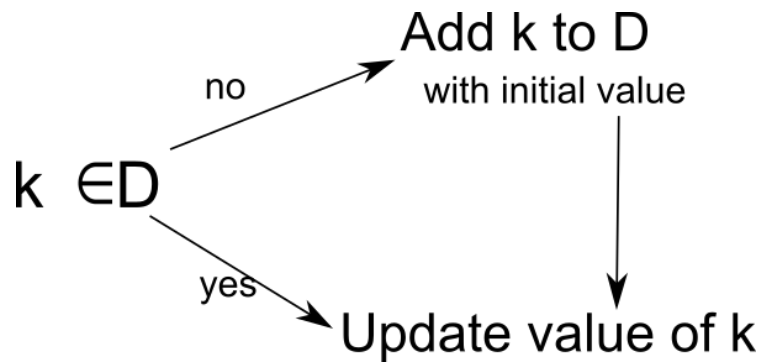


Figure 1: update dictionary

```
>>> 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
```

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 on 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.

Dictionaries gebruiken om if statements te vermijden

Sometimes you might need extremely long `if-elif-else` statements where each clause exactly the same thing, except for a different value of one variable. In these cases it is usually better to lookup 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 much 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`.

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

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 assymetric
{'D', 'E'}
```