General

Working with *floating point* numbers in doctests

If a function returns a *floating point* number, this might give trouble when testing the correctness of the function using a doctest. This is caused by the fact that doctests perform an exact match between the string that represents the result in the doctest, and the result that is printed or returned by the function. In order to do this, the result of the function is first converted into a string. In comparing these two strings, doctests thus do not take into account the possibility of rounding errors that might occur when working with *floating point* numbers. These rounding errors are a consequence of the limited precision with which computers can represent real-valued numbers.

```
>>> 0.1
0.1
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

If in executing a doctest the expected output (a string) does not exactly match the string representation that is returned by the function, the doctest will consider the result as incorrect.

The Pythia environment does take into account rounding errors when working with *floating point* numbers. Unless otherwise stated in the assignment, Pythia will check if the result is correct up to six decimal digits for functions that return *floating point* numbers (either directly or as elements of compound data types). This more or less comes down to rewriting a doctest according to the following strategy.

```
def multiply(x):

    """
    >>> abs(multiply(0.1) - 0.3) < 1e-6
    True
    """

    return 3 * x
```

Buy 3 get 1 for free

General information

Sorting lists

Python supports two ways to rearrange the elements of a list from the smallest to the largest. You can either call the list method `sort` on the list, or you can pass the list to the built-in function `sorted`. However, there is an important different between these two alternatives. The list method `sort` modifies the list *in place* (and does not return a new list), whereas the built-in function `sorted` returns a new list whose elements are sorted from the smallest to the largest.

```
>>> aList = [4, 2, 3, 1]
>>> aList.sort()
>>> aList
[1, 2, 3, 4]
>>>
>>> aList = [4, 2, 3, 1]
```

```
>>> sorted(aList)
[1, 2, 3, 4]
```

Remarks

By default the list method `sort` and the built-in function `sorted` sort the elements of a list in ascending order. Both function also have an optional parameter `reverse` to which the value `True` can be passed to have the elements arranged in descending order.

```
>>> aList = [1, 2, 3, 4]
>>> aList.sort(reverse=True)
>>> aList
[1, 2, 3, 4]
>>>
>>> aList = [4, 2, 3, 1]
>>> sorted(aList, reverse=True)
[1, 2, 3, 4]
```

Lineup

General information

Remarks

The list method `append` can be used to add an element to the end of a list. The list method `insert` allows to add an element at a given position in a list. in case the position that is passed to the list method `insert` is greater than or equal to the length of the list, the element is appended at the end of the list.

```
>>> aList = []
>>> aList.insert(0, 'a')
>>> aList
['a']
>>> aList.insert(0, 'b')
>>> aList
['b', 'a']
>>> aList.insert(1, 'c')
>>> aList
['b', 'c', 'a']
>>> aList.insert(10, 'd')
>>> aList
['b', 'c', 'a', 'd']
```

Hermit crabs

General information

Passing mutable objects to functions

If you pass a mutable object to a function, the function may modify the object *in place*. This might be an explicit goal of the function, but sometimes it is not desirable to modify values that are passed to a function

while the function is being executed.

Say, for example, that we pass a list to a function. What we actually pass to the function is a reference to that list and not a copy of the list (*call by reference* instead of *call by value*). As a result, the parameter to which the list is assigned becomes an alias for the list, and the function is able to modify the list itself (after all, lists are mutable data structures).

```
>>> def modify(aList, element):
...     aList.append(element)
...     return aList
...
>>> aList = ['a', 'b']
>>> modified = modify(lijst, 'c')
>>> modified
['a', 'b', 'c']
>>> aList
['a', 'b', 'c']
```

In the example below, we first make a copy of the list that is passed to the function. Then we modify the copy, but not the original list. Making a copy of a list can be done for example by using *slicing* (`aList[:]`) or by using the built-in function `list` (`list(aList)`).

```
>>> def modify(aList, element):
...     copy = aList[:]
...     copy.append(element)
...     return copy
...
>>> aList = ['a', 'b']
>>> modified = modify(aList, 'c')
>>> modified
['a', 'b', 'c']
>>> aList
['a', 'b']
```

The Online Python Tutor gives a graphical representation of the difference between the above examples: - example without copying - example with copying

Because we no longer need the reference to the original list that was passed to the function, we may rewrite the function `modify` from the above example in the following way.

```
def modify(aList, element):
    aList = aList[:]
    aList.append(element)
    return aList
```

In this, the reference of the variable `aList` to the original list that is passed to the function `modify`, is replaced by a reference to a copy of the list. This replacement is only visible inside the function, because the variable `aList` is a local variable of the function `modify`. You can also inspect this example using the Online Python Tutor.

Drunken ant

General information

Modify strings

An important difference between strings and lists is that strings are *immutable* and lists are *mutable*. This means that you may replace the elements of a list with other elements without the need to make a new list, or that you can add new elements to the list or remove elements from the list. This is not possible with strings. As soon as a string has been created, you can no longer modify the string. You can only make a new string out of the existing string.

```
>>> s = 'darwin'
>>> l = ['d', 'a', 'r', 'w', 'i', 'n']
>>> s[0] = 'D'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> l[0] = 'D'
>>> l
['D', 'a', 'r', 'w', 'i', 'n']
```

If you want to modify some of the characters of a string, it might turn out to be very handig to convert the string into a list of its characters (using the built-in function `list`), modify that list *in place*, and finally string all characters of the list together (using the built-in string method `join`).

```
>>> s = 'darwin'
>>> l = list(s)
>>> l
['d', 'a', 'r', 'w', 'i', 'n']
>>> l[0] = 'D'
>>> l
['D', 'a', 'r', 'w', 'i', 'n']
>>> s = ''.join(l)
>>> s
'Darwin'
```