

# Unit 2.3: Data Structures

August 24, 2021

In the last unit we discussed functions, modules and packages, and had a quick look at the Python's standard library and the Python Package Index.

In this unit we will cover data structures (lists, tuples, dictionaries, sets) in Python that will allow us to work with more powerful data items than just the individual numbers, strings and Booleans that we have used so far. We will also discuss the important difference between call by value and call by reference.

In the next unit time we will start to interact with the "outside world", read from and write to files, and start to use common libraries for analysing data.

## Data Structures in Python

We have already seen the four basic data types that Python provides: string, integer, float and Boolean. Data structures are more than data types, they are used to represent more complex types of data. There are four built-in data structures in Python: lists, tuples, dictionaries and sets. They are all special kinds of variables that can store more than just one value. In lists and tuples the elements have a defined order. Lists, dictionaries and sets are the so-called mutable data structures, meaning that it is possible to add, edit or delete elements. Lists and tuples allow for duplicates, while dictionaries (at least for the keys) and sets contain each value at most once. All these data structures are iterable, that is, a for-loop can automatically go through all their elements.

### Lists

Here is an example of a list in Python, containing the first 12 Fibonacci numbers:

In [1]:

```
fibonacci_numbers = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

That is, lists can simply be defined by comma-separated lists of values in square brackets:

```
<list_name> = [<value1>, <value2>, ..., <valueN>]
```

Lists can also contain values of different types, for example:

In [2]:

```
person_details = ["Bob", "Smith", "22.05.1987", 1.6, 4094379]
```

Individual elements of lists can be accessed by giving their position (index) in the list in square brackets directly behind the name of the variable:

```
<list_name>[<index>]
```

For historical technical reasons the first index of a list is not 1, but 0, and accordingly the last index is its length – 1. For example, we can print the 7th Fibonacci number (13) by:

In [3]:

```
print(fibonacci_numbers[6])
```

13

Python also allows for backward indexing, where the last element in the list is indexed with  $-1$ , and the first with  $-\text{len}(\text{list})$ . For example, in the list of Fibonacci numbers the first 1 has index 0, or alternatively index  $-12$ . The 144 has index 11, or alternatively  $-1$ . The 5 can be indexed by 4 or  $-8$ :

In [22]:

```
print(f"{fibonacci_numbers[0]} == {fibonacci_numbers[-12]}")  
print(f"{fibonacci_numbers[11]} == {fibonacci_numbers[-1]}")  
print(f"{fibonacci_numbers[4]} == {fibonacci_numbers[-8]}")
```

1 == 1

144 == 144

5 == 5

As an example for the other list, we can access Bob's last name by:

In [4]:

```
print(person_details[1])
```

Smith

Conversely, a new value can be assigned to a list element, for example a new last name for Bob:

In [5]:

```
person_details[1] = "Tailor"
```

Resulting in a changed list:

In [6]:

```
print(person_details)
```

['Bob', 'Tailor', '22.05.1987', 1.6, 4094379]

We can delete an element from the list, for example:

In [7]:

```
del person_details[4]
```

Resulting in:

In [8]:

```
print(person_details)
```

```
['Bob', 'Tailor', '22.05.1987', 1.6]
```

The operators "in" and "not in" can be used to check if a particular values is contained in a list (or not):

In [9]:

```
if "Bob" in person_details:  
    print("Bob is there.")
```

```
Bob is there.
```

Using the `len` function we can get the length of a list:

In [10]:

```
print(len(person_details))
```

```
4
```

Lists can also be used as iterable objects for for-loops, which are in fact a convenient way for going through the elements of a list. Here is an example:

In [11]:

```
numbers = [2, 4, 6, 8, 10]  
for n in numbers:  
    print(n)
```

```
2  
4  
6  
8  
10
```

Nesting for-loops into each other is also easy with lists. If we want, for example, to determine all matches to be played in a "Province League" between the provinces in Ireland (Gaelic football or Rugby or ...) the following piece of code is sufficient:

In [12]:

```
teams = ["Connacht", "Ulster", "Munster", "Leinster"]
for home in teams:
    for guest in teams:
        if home != guest:
            print(f"{home} : {guest}")
```

```
Connacht : Ulster
Connacht : Munster
Connacht : Leinster
Ulster : Connacht
Ulster : Munster
Ulster : Leinster
Munster : Connacht
Munster : Ulster
Munster : Leinster
Leinster : Connacht
Leinster : Ulster
Leinster : Munster
```

Lists provide several useful object methods, such as:

`<list>.index(<value>)` returns the index of the first occurrence of the value in the list

`<list>.append(<value>)` appends the value to the list as new element

`<list>.remove(<value>)` removes the (first) element with the value from the list

`<list>.sort()` sorts a list (ascending if it contains numbers, alphabetically if it contains strings)

Here is a simple random example:

In [29]:

```
numbers = [4, 31, 34, 2, 3, 13, 53, 54, 2]
print(numbers)
print(f"Index(2): {numbers.index(2)}")
numbers.append(99)
print(numbers)
numbers.remove(2)
print(numbers)
numbers.sort()
print(numbers)
print(f"Index(13): {numbers.index(13)}")
```

```
[4, 31, 34, 2, 3, 13, 53, 54, 2]
Index(2): 3
[4, 31, 34, 2, 3, 13, 53, 54, 2, 99]
[4, 31, 34, 3, 13, 53, 54, 2, 99]
[2, 3, 4, 13, 31, 34, 53, 54, 99]
Index(13): 3
```

Maybe you have wondered if lists can also contain others lists. Yes, there are also lists of lists. Here is an example, doing the opposite of sorting, namely creating a random running order of presentations:

In [19]:

```
import random

presentations = [
    ["Bob", "World Heritage Sites in Montenegro"], \
    ["Elise", "The discography of Lecrae"], \
    ["Evelyne", "Amphibian species in the American state of Texas"], \
    ["Harry", "Notable individuals who have been affiliated with Pomona College"], \
    ["Jack", "27 local nature reserves in Cambridgeshire"], \
    ["Linda", "The 2018 Atlantic hurricane season"], \
    ["Michael", "The chief minister of Jharkhand"], \
    ["Paul", "The cartography of Jerusalem"]
]

random.shuffle(presentations)

i = 0
print("Presentations on Tuesday, April 3:")
while i < len(presentations)/2:
    print(f"\t {presentations[i][0]}: {presentations[i][1]}")
    i += 1
print("Presentations on Thursday, April 4:")
while i < len(presentations):
    print(f"\t {presentations[i][0]}: {presentations[i][1]}")
    i += 1
```

```
Presentations on Tuesday, April 3:
    Michael: The chief minister of Jharkhand
    Evelyne: Amphibian species in the American state of Texas
    Linda: The 2018 Atlantic hurricane season
    Bob: World Heritage Sites in Montenegro
Presentations on Thursday, April 4:
    Elise: The discography of Lecrae
    Paul: The cartography of Jerusalem
    Jack: 27 local nature reserves in Cambridgeshire
    Harry: Notable individuals who have been affiliated with Pomona College
```

We can also use slicing to split the above list in two. Slicing allows to refer to a whole range of indexes instead to just a single one. A slicing expression has the following basic form, referring to all elements from the first index to the one before the last index:

```
<list>[<first_index>:<last_index>]
```

That can for instance be used to replace the while loops in the example above by for-loops:

In [18]:

```
print("Presentations on Tuesday, April 3:")
for presentation in presentations[0:len(presentations)//2]:
    print(f"\t {presentation[0]}: {presentation[1]}")

print("Presentations on Thursday, April 4:")
for presentation in presentations[len(presentations)//2:8]:
    print(f"\t {presentation[0]}: {presentation[1]}")
```

```
Presentations on Tuesday, April 3:
    Jack: 27 local nature reserves in Cambridgeshire
    Evelyne: Amphibian species in the American state of Texas
    Paul: The cartography of Jerusalem
    Michael: The chief minister of Jharkhand
Presentations on Thursday, April 4:
    Linda: The 2018 Atlantic hurricane season
    Elise: The discography of Lecrae
    Bob: World Heritage Sites in Montenegro
    Harry: Notable individuals who have been affiliated with Pomo
na College
```

With slicing also the so-called unspecified index that can be used. If the first index of the slice is left unspecified, it refers to all elements in the list from the beginning to the second index - 1. If the second index is left unspecified, it refers to the first index and all remaining elements after it. An example for the list of Fibonacci numbers:

In [23]:

```
print(fibonacci_numbers[:6])
print(fibonacci_numbers[6:])
```

```
[1, 1, 2, 3, 5, 8]
[13, 21, 34, 55, 89, 144]
```

**[Now is a good time to do Exercise 1.]**

When copying lists, whether completely or partially with the slicing operators, it needs to be taken into account that copying of complex objects like lists behaves a bit differently than the copying of simple variable values. If the usual assignment operator (=) is used, only the reference to the list is copied, meaning that all changes to the original list are also visible in the copied list, because they refer to the same object. When using the slicing operator or a dedicated `copy()` function, the (respective) elements of the list are copied into a new object that is independent from the original. This is sometimes also called a "shallow copy". If the list contains a reference to another list or complex object, however, only the reference will be copied. Therefore, in this case "deep copy" needs to be made in order to copy the whole list completely. The following code illustrates the difference:

In [2]:

```
import copy

short_list = ["1", "2", "3"]
long_list = ["a", "b", "c", "d", "e", "f", "g", short_list]

# assignment
assigned_list = long_list
print(assigned_list)
del short_list[0]
del long_list[3]
print(assigned_list)

# (shallow) copy
copied_list = copy.copy(long_list)
print(copied_list)
del short_list[0]
del long_list[3]
print(copied_list)

# deep copy
deep_copied_list = copy.deepcopy(long_list)
print(deep_copied_list)
del short_list[0]
del long_list[3]
print(deep_copied_list)
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g', ['1', '2', '3']]
['a', 'b', 'c', 'e', 'f', 'g', ['2', '3']]
['a', 'b', 'c', 'e', 'f', 'g', ['2', '3']]
['a', 'b', 'c', 'e', 'f', 'g', ['3']]
['a', 'b', 'c', 'f', 'g', ['3']]
['a', 'b', 'c', 'f', 'g', ['3']]
```

The main reason for not doing deep copies of lists by default is that they are typically slower and in many cases not necessary.

## Tuples

Tuples are actually very similar to lists, just that they are immutable and cannot be changed. Thus, they only support operations that read from them, but no operations that would change or delete the data structure. In practice tuples are frequently obtained as a result from library functions, but they can also be created directly, by using round brackets:

```
<tuple_name> = (<value1>, <value2>, ..., <valueN>)
```

For example:

In [24]:

```
sample = ("Thursday", "lunch", "pasta", 3.95)
print(sample)
```

```
('Thursday', 'lunch', 'pasta', 3.95)
```

All reading operations (such as indexing, slicing, iteration...) work in the same way as on lists, for example:

In [25]:

```
print(sample[0])
print(sample[len(sample)//2:])

for s in sample:
    print(s)
```

```
Thursday
('pasta', 3.95)
Thursday
lunch
pasta
3.95
```

However, writing operations are not possible on tuples, that is, no changing of elements, no deletions, no appending, no sorting, etc. In practice, tuples are frequently used for example by web services or other APIs to return results. You cannot manipulate these directly, but of course access them and copy the contained values to other data structures. Being read-only data structures also makes operations on tuples faster than on lists, so when working with large collections of data that does not change, the use of tuples might be preferred over lists. Furthermore, tuples are also used to make functions return more than one value. For example:

In [26]:

```
def integer_division(a,b):
    quotient = a//b
    remainder = a%b
    return quotient, remainder

print(integer_division(20,6))
```

```
(3, 2)
```

Note that the return statement does not explicitly define the pair of numbers as a tuple, but any comma-separated list of return values as shown here will automatically be turned into a tuple.

## Dictionaries

Dictionaries are another complex data structure in Python that can be used to store several values, or more precisely key-value pairs. Keys must be unique and immutable (to be safe it is best to only use simple data types such as strings or numbers as keys), while values can occur repeatedly and be any kind of data type. Dictionaries can be defined as follows:

$$\langle \text{dictionary\_name} \rangle = \{ \langle \text{key1} \rangle : \langle \text{value1} \rangle, \dots, \langle \text{keyN} \rangle : \langle \text{valueN} \rangle \}$$

For example:

In [27]:

```
person_details = {"First name": "Bob", "Last name": "Smith", "Building": "BBG", "Room"
```

The `print()` function can also print out dictionaries, for example:



In [28]:

```
print(person_details)
```

```
{'First name': 'Bob', 'Last name': 'Smith', 'Building': 'BBG', 'Room': 223}
```

While in lists a numerical index is used to access the element at a certain position, with dictionaries the key is used to access a particular value. The order of the pairs inside the data structure should not bother the programmer. The basic syntax for accessing a value is:

<dictionary\_name>[<key>]

For example, to print out the first name of the person, we can use the following code:

In [29]:

```
print(f"First name:" + person_details["First name"])
```

```
First name:Bob
```

To change the value for a key or to add a new key-value pair to a dictionary, the assignment statement can be used, for example:

In [31]:

```
person_details["Last name"] = "Tailor"  
person_details["Phone"] = 1234
```

Resulting in a changed dictionary:

In [32]:

```
print(person_details)
```

```
{'First name': 'Bob', 'Last name': 'Tailor', 'Building': 'BBG', 'Room': 223, 'Phone': 1234}
```

Elements can be deleted from a dictionary also via their key, for example:

In [33]:

```
del person_details["Building"]  
del person_details["Room"]
```

Resulting in:

In [34]:

```
print(person_details)
```

```
{'First name': 'Bob', 'Last name': 'Tailor', 'Phone': 1234}
```

Just as lists, also dictionaries know their length, that is, the number of key-value pairs in them:

In [35]:

```
print(len(person_details))
```

3

The operators "in" and "not in" can be used to check if a **key** is contained in a dictionary (or not):

In [36]:

```
print("First name" in person_details)
print("Bob" in person_details)
```

True  
False

Now let's look at a more comprehensive example with dictionaries. The following small program lets the user enter a number of term-definition pairs for a glossary, then prints the glossary alphabetically sorted by the terms:

In [1]:

```
glossary = {}

while True:
    new_key = input("Please enter term: ")
    new_value = input("Please enter definition: ")
    glossary[new_key] = new_value
    more_entries = input("Do you want to add another entry? (y/n) ")
    if more_entries != "y":
        break

# version without alphabetic sorting:
#for key in glossary:
#    print(f"{key}: {glossary[key]}")

keys = list(glossary.keys())
keys.sort()

for key in keys:
    print(f"{key}: {glossary[key]}")
```

```
Please enter term: Wikipedia
Please enter definition: popular online encyclopedia
Do you want to add another entry? (y/n) y
Please enter term: Wikipedian
Please enter definition: Wikipedia editor
Do you want to add another entry? (y/n) y
Please enter term: Wikidata
Please enter definition: central storage for data used in Wikipedia
Do you want to add another entry? (y/n) n
Wikidata: central storage for data used in Wikipedia
Wikipedia: popular online encyclopedia
Wikipedian: Wikipedia editor
```

As with lists, also with dictionaries there is a difference between a normal "shallow" copy through the

assignment operator, and a thorough deep copy with the `copy.deepcopy()` function.

[Now is a good time to do Exercise 2.]

## Sets

Sets in Python correspond to sets in mathematics. They contain each element only once, and set operations like union and intersection can be performed on them. Sets support membership tests (`in`, `not in`), but they are unordered and have no index to access individual elements. Sets can be defined as in the following example:

In [38]:

```
set1 = {3,1,2}
set2 = set([5,6,4])
```

That is, a list of elements in curly braces defines a set. An empty pair of curly braces is however already reserved for creating an empty dictionary, so alternatively a set can be created as shown in the second line, but calling the `set` function with a (possibly empty) list to create a new set.

Sets define no order themselves, but commands like `print` might order the elements:

In [39]:

```
print(set1)
print(set2)
```

```
{1, 2, 3}
{4, 5, 6}
```

Elements can be added and removed from sets with the corresponding functions. Adding an element to a set that is already contained in it will simply have no effect:

In [40]:

```
set1.add(1)
print(set1)
set1.add(4)
print(set1)
```

```
{1, 2, 3}
{1, 2, 3, 4}
```

The operators `|`, `&`, `-` and `^` can be used to compute the union, intersection, difference and symmetric difference between sets, respectively:

In [41]:

```
print(set1 | set2)
print(set1 & set2)
print(set2 - set1)
print(set2 ^ set1)
```

```
{1, 2, 3, 4, 5, 6}
{4}
{5, 6}
{1, 2, 3, 5, 6}
```

[Now is a good time to do Exercise 3.]

## Call by Reference vs. Call by Value

There is an important difference between passing complex data objects (like the data structures discussed today) as arguments to a function, compared to passing variables of simple types. Have a look at the following code and try to guess what it does before you (execute it and) check the actual output:

In [42]:

```
# function that manipulates the string passed as argument
def add_to_string(string, addition):
    string = string + addition
    print(f"\t {string}")

# function that manipulates the dictionary passed as argument
def add_to_dictionary(dictionary, key, value):
    dictionary[key] = value
    print(f"\t {dictionary}")

#main program
a_string = "Hello!"
print(a_string)
add_to_string(a_string, " Hello World!")
print(a_string)
a_dictionary = {}
print(a_dictionary)
add_to_dictionary(a_dictionary, "greeting", "Hello World!")
print(a_dictionary)
```

```
Hello!
      Hello! Hello World!
Hello!
{}
      {'greeting': 'Hello World!'}
{'greeting': 'Hello World!'}
```

A string and a dictionary are defined and then passed to a string and dictionary manipulation function, respectively. The printouts within the functions show the effects of the manipulation. However, the printouts after the function calls show a difference: The string is still the same as before the manipulation, while the dictionary has changed. The reason for this lies in the way that parameters are passed to functions. Passing variables of the basic data types (such as strings, integers, floats and booleans) happens as call-by-value, that is, the current value of the variable is copied to create a new local variable with the same value for use inside the

function. Because it is only visible in the scope of the function, however, changes to it will not be visible after the function. To achieve that, the new values would have to be returned by the function and, e.g., assigned to another variable by the calling code.

In contrast, passing variables of complex data types (such as lists and dictionaries) happens as call-by-reference. As with shallow copies, only the reference to the (address of the) object in the memory is copied to a new variable and passed as argument, but no (deep) copy of the object itself is created. Thus, manipulations to the object that happen inside a function will also be visible afterwards, also without returning and re-assigning the result of the function. If a function is not supposed to be able to change the object passed as an argument, a (deep) copy needs to be made before it is passed to the function.

## Exercises

### 1. List of Places Visited

Write a program that repeatedly asks the user to enter the name of a place they have visited today, and appends these place names to an initially empty list. And empty input (hitting Enter without typing in any text) causes the program to stop asking for next places visited. It then prints out the list of places entered.

This should look something like:

```
Which place have you visited today? Amersfort
Which place have you visited today? Utrecht
Which place have you visited today? Amsterdam
Which place have you visited today? Weesp
Which place have you visited today? Den Haag
Which place have you visited today? Zwolle
Which place have you visited today? Amsterdam
Which place have you visited today? Lelystad
Which place have you visited today? Almere
Which place have you visited today? Deventer
Which place have you visited today?
['Amersfort', 'Utrecht', 'Amsterdam', 'Weesp', 'Den Haag', 'Zwolle', 'Amsterdam', 'Lelystad', 'Almere', 'Deventer']
```

Then let the program perform different operations on this list. Start with:

- print out the number of places visited
- print out the first/last place visited
- print out the first/last three places visited
- delete the place at a certain index
- delete a certain place name
- update the place at a certain index to another place name
- print out the list in prettier formatting

You can repeatedly print out the list to observe what happens. This could then look something like:

```
Number of places visited: 10
First place visited: Amersfort
First three places visited: ['Amersfort', 'Utrecht', 'Amsterdam']
Last place visited: Deventer
Last three places visited: ['Lelystad', 'Almere', 'Deventer']
```

```
Removing the third from last place.
['Amersfort', 'Utrecht', 'Amsterdam', 'Weesp', 'Den Haag', 'Zwolle', 'Amsterdam', 'Almere', 'Deventer']
```

```
Removing Den Haag.
['Amersfort', 'Utrecht', 'Amsterdam', 'Weesp', 'Zwolle', 'Amsterdam', 'Almere', 'Deventer']
```

```
Updating the fifth entry to De Bilt.
['Amersfort', 'Utrecht', 'Amsterdam', 'Weesp', 'De Bilt', 'Amsterdam', 'Almere', 'Deventer']
```

```
Places visited:
    Amersfort
    Utrecht
    Amsterdam
    Weesp
    De Bilt
    Amsterdam
    Almere
    Deventer
```

When the operations above work, try other things that come to your mind.

## 2. Dictionary of Places Visited

This is a variation of the previous exercise, but with using a dictionary instead of a list. As above, the program repeatedly asks the user to enter the name of a place they have visited today. If a place name is entered for the first time, it is added to the (initially empty) dictionary as key, and its value is initialized with 1 (counter). If a place name has been entered before, the counter at the respective key is incremented. An empty input (hitting Enter without typing in any text) causes the program to stop asking for next places visited. It then prints out the places and how many times they were visited.

This should look something like:

```
Which place have you visited today? Utrecht
Which place have you visited today? Amersfort
Which place have you visited today? Amsterdam
Which place have you visited today? Weesp
Which place have you visited today? Hoorn
Which place have you visited today? Den Haag
Which place have you visited today? Hoorn
Which place have you visited today? Leiden
Which place have you visited today? Amsterdam
Which place have you visited today? Hoorn
Which place have you visited today?
{'Utrecht': 1, 'Amersfort': 1, 'Amsterdam': 2, 'Weesp': 1, 'Hoorn': 3, 'Den
Haag': 1, 'Leiden': 1}
```

Then let the program perform different operations on this dictionary. Start with:

- remove a place from the dictionary
- decrement the number of times a place was visited
- ask the user to enter a place name and look up how many times it was visited
- print out the dictionary in prettier formatting (with or without alphabetical ordering of the place names)

You can repeatedly print out the dictionary to observe what happens. This could then look something like:

Removing Leiden.

```
{'Utrecht': 1, 'Amersfort': 1, 'Amsterdam': 2, 'Weesp': 1, 'Hoorn': 3, 'Den
Haag': 1}
```

One less visit in Amsterdam.

```
{'Utrecht': 1, 'Amersfort': 1, 'Amsterdam': 1, 'Weesp': 1, 'Hoorn': 3, 'Den
Haag': 1}
```

Which place are you interested in? Utrecht

You visited this place 1 times.

Places visited:

```
Utrecht (1 times)
Amersfort (1 times)
Amsterdam (1 times)
Weesp (1 times)
Hoorn (3 times)
Den Haag (1 times)
```

Places visited:

```
Amersfort (1 times)
Amsterdam (1 times)
Den Haag (1 times)
Hoorn (3 times)
Utrecht (1 times)
Weesp (1 times)
```

When the operations above work, try other things that come to your mind.

### 3. Set of Places Visited

This is another variation of the previous two exercises, this time using sets. Write a program that repeatedly asks the user to enter the name of a place they have visited today, and adds these place names to an initially empty set. And empty input (hitting Enter without typing in any text) causes the program to stop asking for next places visited. It then prints out the set of places entered.

This should look something like:

```
Which place have you visited today? Amsterdam
Which place have you visited today? Hoorn
Which place have you visited today? Enkhuizen
Which place have you visited today? Hoorn
Which place have you visited today? Medemblik
Which place have you visited today? Hoorn
Which place have you visited today? Den Helder
Which place have you visited today? Den Oever
Which place have you visited today? Enkhuizen
Which place have you visited today? Amsterdam
Which place have you visited today?
{'Medemblik', 'Enkhuizen', 'Amsterdam', 'Den Helder', 'Hoorn', 'Den Oever'}
```

Then let the program perform different operations on this set. Start with:

- print out the number of places in the set
- remove a place from the set
- let a second user enter their places visited, and then print out the places visited by both

You can repeatedly print out the set to observe what happens. This could then look something like:

There are 7 places in the set.

Removing Den Oever.

```
{'Medemblik', 'Enkhuizen', 'Amsterdam', 'Den Helder', 'Hoorn'}
```

```
User 2, which place have you visited today? Amsterdam
User 2, which place have you visited today? Enkhuizen
User 2, which place have you visited today? Stavoren
User 2, which place have you visited today? Workum
User 2, which place have you visited today? Harlingen
User 2, which place have you visited today? Leeuwarden
User 2, which place have you visited today? Franeker
User 2, which place have you visited today? Stavoren
User 2, which place have you visited today? Lemmer
User 2, which place have you visited today? Amsterdam
User 2, which place have you visited today?
Places both users have visited: {'Enkhuizen', 'Amsterdam'}
```

When the operations above work, try other things that come to your mind.

### 4. Extra: Anagram Test



An anagram is a word or phrase that is made by rearranging the letters of another word or phrase. For example, "secure" is an anagram of "rescue". Write a function `is_anagram(word1,word2)` that checks if the two words are anagrams of each other. If so, the function should return `True`, and `False` otherwise. You can use the following code to test your function:

```
# Test program
print(is_anagram("rescue", "secure"))
print(is_anagram("Rescue", "Secure"))
print(is_anagram("Rescue", "Anchor"))
print(is_anagram("Ship", "Secure"))
```

The output should be:

```
True
True
False
False
```

That is, the function should **not** distinguish between upper- and lower-case letters.

Hint: Remember that strings can be interpreted as lists of characters.