DATA STRUCTURES AND ALGORITHMS

Extra reading 2

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2024 - 2025

Content

- Iterators in Python (iterators in other programming languages will be discussed in the next extra reading(s))

Iterators

- As discussed in Lecture 2 and Seminar 1, an iterator is a structure which helps us loop though the elements of a container in a generic manner, independently of the internal representation of the container.
- The advantage of the iterator is that it gives us a common way of processing the elements of a container, which does not change if the internals of the container change.
- During this course, we will mainly talk about a simple, unidirectional, read-only iterator, which has the following operations: *init*, *first*, *getCurrent*, *next*, *valid*.

Iterators II

- Iterators exist in many programming languages as well, even if they do not look exactly like the simplified version we consider. The goal of this extra reading is to provide you some information about how the iterator looks like in *Python*.

- Without entering in too many details, I will try to present the types of iterators, their operations, what happens if you modify the container why iterating, and other details that I think you might find interesting.

Iterators in Python

- Python uses iterators a lot, all containers in Python can be iterated, but the iterator is pretty hidden: in many cases iterators are created and used without the user seeing it explicitly.
- Whenever an expression of the form *for elem in list:* is used, internally an iterator is created and used for the traversal.

Iterators in Python

- The iterator is created and returned by a function called: *__iter__*, all containers and other classes which want to have an iterator, will need to implement this function.
- The iterator itself contains one single operation, *__next__* which goes to the next element and returns the current one. If there are no more elements, it throws a *StopIteration* exception.
- The two operations together are sometimes called the *iterator protocol* in Python, meaning that if you want to create an iterator, this is what you must implement.
- Since there is only the *next* operation, iterators are unidirectional, you cannot go back with them, and you cannot reset an iterator to the beginning (but obviously you can create another iterator).

Iterators in Python

- As seen in the previous slide, in order to create an iterator, two function need to be implemented: *__iter__* and *__next__*. The first must be implemented in the container class, while the second *might* be implemented as part of the container class or it might be part of a separate class (the actual Iterator class). The difference between the two approaches is that if the iterator is part of the container class, you cannot create two iterators for the container in the same time.

- In order to see the difference, I have included two Python implementations in the corresponding folder on Teams.

File PythonIteratorV1 I

- Contains a Bag with frequencies, implemented using two lists in Python, where both the *__iter__* and *__next__* functions are part of the BagWithFrequencyPI1 class.
- This has the advantage that the *__next__* function has direct access to the lists which contain the elements and frequencies.
- On the other hand, in the *test1()* function you can see that when we want to print pairs or triples of elements, it will not print every possibility, only the iterator for the last (second in case of pairs, third in case of triples) element goes through all elements, when that iterator becomes invalid, the other ones invalid as well.
- Nevertheless, if you want to use two iterators one after the other (different loops), that works without a problem.

File PythonIteratorV1 II

```
1. for elem1 in b:
2.     for elem2 in b:
3.         print(elem1, '-', elem2)
```

- In the above piece of code you can see how we print pairs. On line 1, function *__iter__* of the bag is called, which initializes *currentPosition* and *currentFrequency*, and *elem1* will be the first element from the bag.
- On line 2, *__iter__* is called again, and it initializes the iterator again to the first element (technically nothing changes, it is there currently) and *elem2* will be the first element from the bag.
- The pair is printed and then the "second" iterator (the one with *elem2*) advances and elem2 gets a new value. And this continues until there are no more values for *elem2*, case when the iterator will become invalid.

File PythonIteratorV1 III

- At this point, we return to the first for loop and we want to get another value for *elem1*, but the iterator is invalid, since it is the same iterator that was used in the inner loop. So the outer loop stops as well.
- The same happens when we try to print triples, only the innermost iterator will actually go through the elements, the first two will only take the value of the first element from the bag.

File PythonIteratorV2 I

- Contains a Bag with frequencies, implemented using two lists in Python, where the Iterator is a separate class, so function *__iter__* is in class BagWithFrequencyPI2 and function *__next__* is in the BagIterator class.
- *__iter__* simply calls the constructor of the BagIterator class, passing itself (*self*) as parameter to it (remember, iterator needs to have a reference to the container it iterates over) and returns the result.
- BagIterator has only two functions: its constructor, where the initializations happen (for the bag, the current position and the current frequency) and the *__next__* function which is almost the same as the one from the previous example.

File PythonIteratorV2 II

- However, in this case we have a problem: since BagIterator and BagWithFrequencyPI2 are two different classes, BagIterator only has access to the public members from the BagWithFrequencyPI2, but not the private representation (the two lists containing the elements and frequencies). This is not good, since the *__next__* operation needs to return an element from the container.

File PythonIteratorV2 III

- The solution is something which is not elegant at all, but works:
    - You can access the private attributes of a class (the ones with start with __) through the name of the class, in the following way.
    - Below you have the code to check if the iterator is valid. To do this, we compare the current positions (*self.__currentPos*) to the length of the list containing the unique elements, it is a list called *__elems* in the the *BagWithFrequencyPI2* class.
    - *__bag* is the name of the attribute in the *BagIterator* class, which contains the bag over which we are iterating.

```
if self.__currentPos == len(self.__bag._BagWithFrequencyPI2__elems):
    raise StopIteration
```

- On the other hand, we can see that in this way, we can print pairs and triples of elements without any problem (*test1()* function is the same as for the first version), all the iterators are independent of each other.

- I have tried printing pairs of elements from a *list* and a *dict*, with two for loops like in the example above, and it worked without a problem.

- The code is in function *test1ListDict* in the PythonIteratorV2 file.

Iterators in Python - variants I

- In the previous examples the iterator was created, managed and pretty much hidden by the *for* loops in Python. But obviously, we might want to use iterators in other contexts as well, for example in a *while loop*. In such cases, we need to create the iterator "manually", by calling the *__iter__* and *__next__* functions. This is possible, and you can see examples of doing this in the *test2* functions of the previously mentioned two classes.

- In the *BagWithFrequencyPI1* class, where the iterator and the container are in the same class, there are two *test2* functions, in the second there is not even a variable called iterator, everything is done through the container.

Iterators in Python - variants II

- Nevertheless, it is not a good practice to call directly functions starting with __ . This is why, it is suggested to use the built in *iter* and *next* functions, which call the corresponding functions, but provide a nicer interface.
- This can be seen in function *test3*, in both files.

Iterators in Python - variants III

- Another interesting issue is what happens when you change the container while iterating over it. Changing can mean adding elements or removing element.
- Most resources say that this is an unsafe practice, which might lead to unexpected behaviour. And this should be enough to not try something like this in real code.
- Nevertheless, to see some examples, I have tested the following cases for our BagWithFrequencyPI2, in the function *test4()*.

Iterators in Python - variants IV

- Case 1 - Iterate over the bag and if the current element is even, add element 111 to the bag.
  - Works OK (does not crash, does not go into infinite loop and the iteration prints the newly added elements as well).
  - For the bag this behaviour makes sense, in the implementation *currentPos* is compared to the length of the list containing the unique elements. When we add a new element, this length increases by one (we add the same element over and over again, it gets added once in the list of unique elements, next adds will increment the frequency). Also, since 111 is odd, and we only add in case of even numbers, by the time the iterator gets to the new element, no more add operations will happen, and it can iterate though all of its occurrences.

Iterators in Python - variants V

- Case 2 - Iterate over the bag and if the current element is even, add the same element again.
    - Leads to infinte loop (as long as the container contains even elements, obviously).
    - This makes sense, if we look at the implementation for our bag: when we add an already existing element, we increase its frequency. This way, the *currentFr* value from the iterator will never be equal to the actual frequency (they both increase in parallel) so once we get to an even element, we will never leave that *currentPos*.

Iterators in Python - variants VI

- **Obs:** Technically, when you implement an iterator over a bag with frequencies, you have two options to work with the current frequency attribute of the iterator: start from 1 and increment until it gets equal to the actual frequency (this is what we have implemented now); the alternative is to start from the actual frequency and decrement until it gets to 0. If the second case is implemented, this test would not lead to an infinite loop.

Iterators in Python - variants VII

'

- Case 3 - Iterate over the bag and remove the current element after it was printed.
    - For the bag with frequencies, the code crashes, with a *list index out of range* exception. Before crashing, it does print a few elements, roughly half of them.
    - The bag contains 15 unique elements and the last few elements all have frequency 1. When the iterator is at the end and the last element is removed, the iterator should become invalid.
    - This does not happen because our condition for invalidating the iterator is to check if the current position is equal to the length of the elems array, but since we removed an element, current position will be greater than the length.
    - Using $\geq$ instead of $=$ would solve this problem.

Iterators in Python - variants VIII

- Nevertheless, the discussion about how to *solve the problem* is a purely theoretical one, because it shows that the behaviour of the code depends on the internal implementation and this should not happen. It is not OK if the same piece of code behaves differently, depending on the implementation of the container (bag with or without frequencies or whether we increment or decrement the current frequency). This is what unexpected behaviour means, and this is why such modifications should be avoided.

Iterators in Python - variants IX

- I also wanted to see how Python's own containers behave in such situations, so I have tried a few scenarios with them as well.
- The tests for the *list* container are in function *test4List()*

Iterators in Python - variants X

- **Case 1** - Iterate over the list and if an even element is found, the value 111 is appended to the end of the list.
  - It works just like in case of the Bag.
- **Case 2** - Iterate over the list and if an even element is found, the value 111 is added at the beginning of the list.
  - Infinite loop, just like in case of the Bag.
  - While I do not know the exact implementation details of the list, I guess that the following thing happens: list being a dynamic array its iterator is almost certainly a single current position. Assume that current position at a moment is i and $list[i]$ is even: we insert 111 to position 0, which means that $list[i]$ will be moved to position $list[i+1]$. In the iterator current position is incremented, from i to $i+1$, but now on position $i+1$ we have again an even element and we need to insert to position 0 again $\rightarrow$ infinite loop

Iterators in Python - variants XI

- **Case 3** - Iterate over the list and after printing the current element, remove it.
  - The code prints every second element only (no matter if the list has an odd or an even number of elements). Assuming standard dynamic array iterator, this behaviour makes sense.

Iterators in Python - variants XII

- Finally, I have tried some cases with a *dictionary* as well, the code is in the *test4Dict()* function.
  - **Case 1**: Iterate over the dictionary (which means to iterate over the keys) and if the current key is even, change the value to 111.
    - Works without any problems since changing the value of a key is not considered to be a modification in the container, this change is allowed.
  - **Case 2**: Iterate over the dictionary and if the current element *elem* is even, add to the dictionary the pair *(elem+1, 1111)*.
    - Throws a *dictionary changed size during iteration* exception, right after the first modification.
    - Unlike the *list*, the *dict* actually checks for size modifications before continuing the iteration, and throws an exception if a change was detected.

Iterators in Python - variants XIII

- What happens if, when modifying the dictionary, you do both a remove and an add operation? So technically, the size does not change:
- It actually prints elements, but once it printed as many elements as we had in the dictionary initially, it throws a *dictionary keys changed during iteration*.
- I did not find any information about why there are two different exceptions and how the verification is implemented :(

Resources used

- In creating this extra reading, I have used the following resources:
  - https://realpython.com/python-iterators-iterables/
    - A long description of many features. Not necessarily focused on the container part (i.e., iterator is something that you create over a container), probably because in Python iterators are a lot more versatile. It is an interesting read, with many practical examples. If you intend to continue working in Python, it is really useful.
  - https://python.land/deep-dives/python-iterator
    - A shorter, but nice explanation about iterators, in case you do not want to know everything, but would like to know the basics.