4 Flow control in sequences. Code style

Looping allows to take the same set of actions with every item in a collection of data. In Python, looping through a sequence, set or dictionary can be done without an looping variable, because lists, dictionaries and tuples are **iterable**. As a result working with list of any length can be efficient, despite the number of elements.

4.1 Traversing with a `for` loop

A lot of computations require processing one element of a collection at a time, often in a loop. The processing starts at the beginning, selects a character, uses it somehow and continues to the next item – this pattern is called a **traversal**. In many programming languages such operations are defined with a `while` or `for` loop as in example 4.1. Notice, that each loop requires an additional variable that would indicate the current iteration and, at the same time, index the sequence.

Example 4.1. Iterating through a collection using an iteration variable `i`

```
word = "hello, there!"
i = 0
while i < len(word):
    letter = word[i]
    print(letter)
    i += 1

# for loop with an additional variable
for i in range(len(word)):
        print(word[i])
```

However, in Python it is possible to write a traversal in a more concise way, using a keyword `in`. Some of the built-in objects are iterable and can be used as an iterator in a loop, e.g. strings, lists, dictionaries and tuples as well as sets. Then, the example 4.1 can be rewritten as in example 4.2. Each time a new loop iteration starts, the next item in a collection is assigned to the variable `letter`. The loop continues until no items are left to process. Creating a `range` object dependable on a length of a variable is not necessary. Each iteration produces one element of the iterated collection.

Example 4.2. Iterating through a string in a more Pythonic way

```
word = "hello, there!"
for letter in word:
    print(letter)
```

You do not need to know the implementation of the data to be able to go through all its elements. It is also possible to iterate elements on the fly, allowing processing data streams that might be to big to fit into computer's memory all at once.

Another approach, using the built-in Python function, requires using an `enumerate` function that can iterate through pairs: an item and its index in a sequence (example 4.3). `enumerate` creates an iterator that returns a sequence of tuples – an index and an item. The item can be accessed without the index.

Example 4.3. Iterating through a tuple with index and value

```
example_tuple = (1, 2, 3, 4, 5)
for index, el in enumerate(example_list):
        print("index:", index, ", value: ", el)
```

`enumerate` is similar to going through the element using `range` and iteration variable, but makes the code clearer and easier to understand.

Using methods presented above will work well, if the task does not require an update of the elements in a sequence. Otherwise, an index is needed to reassign the variable. One way to do that, is to use two built-in functions: `range` and `len` (example 4.4). Example is not very *Pythonic*, but in order to modify the element at the chosen index, it is necessary to **index the original sequence** and not to reassign the element resulting from the `enumerate`.

Example 4.4. Assigning values to an iterated variable

```
example_list = [1, 2, 3, 4, 5]
for i in range(len(example_list)):
    example_list[i] = example_list *3
```

Traversing through keys and values in a dictionary can be executed using one of the methods presented in the previous chapter (example 4.5), `items`, `values` or `keys`. Looping through keys is the default behaviour, so typing `for key in example_dictionary` instead of `for key in example_dictionary.keys()` will give exactly the same results.

Example 4.5. Iterating through a dictionary

```
example_dictionary = {"a":1, "b":2, "c":3, "d":4, "e":5}
for key in example_dictionary.keys():
    print("key:", key)

for value in example_dictionary.values():
    print("value: ", value)

for key, value in example_dictionary.items():
    print("key:", key, "value: ", value)

#same effect with packed values
for item in example_dictionary.items():
    print("key:", item[0], "value: ", item[1])
```

4.1.1 Modifying the objects inside a loop

`For` loops are very good for working with iterable objects, however sometimes the object cannot be modified or it is generally not advised to add elements or remove elements from a currently traversed iterable variable inside a `for` loop, because Python can have a problem with keeping track of the items. To modify the object as you work through, it is better to use a `while` loop.

Example 4.6. Reassigning element from one set to another

```
# create an example set and an empty set to reassign elements to
example_set = set(range(10))
set_2 = set()

# the loop works until example_set is True <- is not empty
while example_set:
       set_2.add(example_set.pop())
```

One example is moving one set of data to another: use an iterable variable as a boolean expression (the loop will stop execution when the object will be empty) and pop the elements to other variable

(example 4.6). To remove every instance of a specified value `element` from an iterable `collection`, define the header of a loop as `while element in collection:` (example 4.7).

Example 4.7. Removing all occurences of an element in collection

```
example = [1, 2, 3, 4, 1, 1, 1, 1, 1, 3, 2]
value = 1
i = 0  # counter for removed values

while value in example:
       # remove gets rid of only one occurence, so if we want to keep all the other duplicated
           values
       # we need to remove each value one by one
       example.remove(value)
       # keeping track of how many values we removed
       i += 1
print("Example without value", value, ":", example, "Removed", i, "occurences")
```

4.2 zip

`zip` is a built-in function that takes two (or more) sequences and interweaves them, like a zipper. The result is a zip object, that can iterate through pairs (or triples if three sequences are used, or even more values at once). The most common use is in a `for` loop (example 4.8). A zip object is an **iterator** which is an object that iterates through a sequence. If sequences are not of the same length, zip stops, when the shorter sequence is finished.

Example 4.8. Using `zip` in a function

```
list_a = [1, 2, 3]
list_b = ["a", "b", "c"]
for pair in zip(list_a, list_b):
    print(pair)       # pair is a tuple

for a, b in zip(list_a, list_b):
    print(a, b)       # individal values -> tuple unpacking
```

`zip` method can be also used to create a list of pairs in two objects (example 4.9).

Example 4.9. Using zip to create a list of pairs

```
list_a = [1, 2, 3]
list_b = ["a", "b", "c"]
pairs = list(zip(list_a, list_b))
```

Since `zip` returns a zip object, the best way to see the created pairs or triples (which are tuples), is casting the result either to a list or to tuple.

`zip` can also be used to create pairs of keys and values that would be used in a dictionary (example 4.10).

Example 4.10. Creating a dictionary using a `zip` function

```
list_a = [1, 2, 3]
list_b = ["a", "b", "c"]
d = dict(zip(list_a, list_b))
```

4.3 Additional functions used in loops

Sometimes there is a need to go through the elements in an ordered manner, not necessarily by index, but somehow sorted. Two functions might be useful to approach this task. `sorted` is a function that sorts an iterable variable (sequences but also sets and dictionaries – by sorting its keys) and returns a new list of items. The result of a `sorted` function can be used to traverse through (example **??**). By default, the the values are returned in an ascending order.

Example 4.11. Sorting the input for a `for` loop

```
example = [2, 3, 5, 1, 0, -8, 4]

# print the elements in a ascending manner using sorted
for element in sorted(example):
        print(element)
```

Another useful function is `reversed`, which returns a reverse iterator.

Example 4.12. Exit condition for float equality

```
example = [2, 3, 5, 1, 0, -8, 4]

# print the elements in a ascending manner using sorted
for element in sorted(example):
        print(element)
```

4.4 Iterations vs accuracy

Throughout this course, loops will be used to compute roots of the function or solutions to linerar equations. Usually, the number of steps needed to reach the right answer will not be known in advance – the loop will run until a solution is found or rather, until the estimated value will stop changing. It seems natural, that one would want to check, whether the current estimation and previous estimation are the same using an equality operator (`y == x`) (example 4.13). However, testing float equality is very risky and definitely **not recommended**. Floating-points are only approximately correct, most of the numbers, like 1/3, **cannot be represented exactly with a float**.

Example 4.13. Exit condition for float equality

```
while True:
        print(x)
        y = (x + a/x) / 2
        if y == x:
                break
        x = y
```

Instead of testing equality, it is safer to test if the result has acceptable accuracy or magnitude. In other words, test if the difference between the previous and current estimate is lower than accepted difference level (tolerance), ε (example 4.14).

Example 4.14. Exit condition for float equality

```
epsilon = 0.000001
while True:
        print(x)
        y = (x + a/x) / 2
        if abs(y-x) < epsilon:
```

```
            break
        x = y
```

4.5 Styling the code

Python is famous for its readability. One of the reasons that programs of each Python programmer are structured similarly and are easy to read, is the set of styling guidelines and conventions included in the document called **PEP-8**[1]. The Python style guide was written with the intention, that code is read more often than it is written.

Most commonly used rules included in PEP-8 are:

- consistency is important;
- indentation: 4 spaces per indentation level;
- line length: each line should be less than 80 characters; comment lines: 72 characters;
- blank lines: use a blank line (only one) to separate sections of code; two blank lines to separate top-level functions, one blank line for methods in classes
- operators: mathematical operators, assignment operator, comparison operators should be surrounded with a space character (except for more complicated operations, then only the operator with highest priority),
- parentheses: do not use spaces before and after the parenthesis;
- element separation: separate arguments of functions or values in a list with a comma followed by space;
- imports: put each imported module in a separate line at the top of the file (before module globals and constants);
- comments: inline comments should be separated at least two lines from the statement, comment start with a `#` sign followed by a space;

More guidelines can be found in the official document: `https://python.org/dev/peps/pep-0008/`.

4.6 Exercises

Exercise 4.1. Compute a sum of numbers stored in a list. Solve the task using `while` and `for` loops.

Exercise 4.2. Construct a nested `for` loop to print elements of a nested list.

Exercise 4.3. Write a function called `nested_sum` that takes a list of lists of integers and adds up the elements from all of the nested lists (example 4.3).

Example 4.15. Exercise 4.3

```
t = [[1, 2], [3], [4, 5, 6]]
nested_sum(t) #result: 21
```

Exercise 4.4. Write a function called `is_sorted` that takes a list as a parameter and returns `True` if the list is sorted in ascending order and `False` otherwise (Example 4.16). Remember, that the list may contain more than 3 elements.

[1] PEP stands for **Python Enhancement Proposal** and it is a proposal to make a change in Python language

Example 4.16. Exercise 4.4

```
is_sorted([1, 2, 2])  # True
is_sorted(['b', 'a']) # False
```

Exercise 4.5. Given a list of roots (stored in a list), compute the value of a polynomial $P(x)$ for $x = 2$.

$$P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

Exercise 4.6. Find prime numbers lower than 100 using a `for` or a `while` loop.

Exercise 4.7. Given the values of coordinates in the lists below, compute 3d distances between the adjusting points:

```
X = [10, 1, 21, 4.8, 23, 5.6, 98.4] # X coordinates
Y = [2, 1.5, 4.8, 12.8, 17, 16.8, 98.4] # Y coordinates
Z = [0, 11, 13, 14.8, 3, 5, 98.4] # Z coordinates
```

Exercise 4.8. Write a program that takes a string and prints the letters in decreasing order of frequency. Find text samples from several different languages and see how letter frequency varies between languages[2].

Exercise 4.9. Using list comprehensions, create a list of odd values from 0 to 20.

Exercise 4.10. Create a dictionary containing products and their prices. Create a list with names of some of the products (shopping list). Compute the cost of the products on the shopping list.

Exercise 4.11. Verify if the code above is complicit with the guidelines in PEP-8. Correct accordingly.

4.7 Useful links

- looping: `https://docs.python.org/3/tutorial/datastructures.html#looping-techniques`
- more on sorting `https://docs.python.org/3/howto/sorting.html#sorting-basics`
- PEP-8 `https://www.python.org/dev/peps/pep-0008/`

References

Beazley, D. M. (2009). *Python essential reference*. Addison-Wesley Professional.

Ceder, N. (2018). *The quick Python book*. Simon and Schuster.

Downey, A. B. (2016). *Think Python. How to Think Like a Computer Scientist*. O'Reilly Media, 2nd edition.

Lubanovic, B. (2014). *Introducing Python: Modern Computing in Simple Packages*. O'Reilly Media, Inc.

Lutz, M. (2013). *Learning Python*. O'Reilly Media, 5th edition.

Python (2021). Python 3.10.0 documentation. `https://docs.python.org/3/` [Accessed 17 October 2021].

[2]Exercise adapted from Downey (2016)