3 Built-in data types

There are several data types implemented in standard Python library. You should already be familiar with numeric types (`float` and `int`) and have an understanding of the `boolean` type. Just as a reminder, `boolean` has only two available values `True` and `False`. `True` is represented with 1 (or any other non-zero value) and 0 can be interpreted as `False`. Any empty sequence will be treated as False and anything with a non-zero length is considered True. You can manipulate the numeric values using the arithmetic operators (+, -, *, etc.). Mind that in Python, also a `complex` object, representing real and imaginary part of a complex number, and operations on fractions (rational numbers) are also available[1].

In general, Python data types are divided into two groups:

- **mutable** – variables, or part of variables, can be changed, the length can be modified,
- **immutable** – individual parts of variables cannot be changed, the length is constant.

If an object is mutable, it can be changed in place (without the need of assignment to a new variable). Immutable objects require running expressions to create new objects and assigning the results to variables.

Python standard library includes very useful iterable objects. Iterable object can be iterated by in a loop. Examples of iterable objects are strings, tuples, lists, dictionaries, sets and range objects.

3.1 Sequences

A **sequence** of elements is **an ordered collection of values**. An individual element of a sequence is usually accessible using a bracket operator (example 3.1). The expression in brackets is called an **index** and a value of a sequence that it accesses is called an **item**. The index indicates which character in a sequence is requested. Python indexes elements **starting from 0**. The index can be treated as **an offset from the beginning**, and the offset of the first element is zero. The value of an index needs to be an integer. Negative indices are also used. They enable accessing the elements starting from the end (Example in Table 3.1). If an index has a negative value, it is counted backward from the end of the sequence. The last element in every sequence has an index `-1`.

Table 3.1. Indexing sequence elements with positive and negative indices

x =	[	"first",	2,	"third",	4.5	]
positive index		0	1	2	3	
negative index		-4	-3	-2	-1	

3.1.1 Strings

A string is a **sequence** of characters. To define a string, characters must be enclosed in single or double quotes. An individual element of a string variable can be accessed using an index (example 3.1).

Example 3.1. Accessing a string character

```python
b = "Hello, there"
c = 'hello!'
print(b[0]) # H
i = 1
print(b[i+1]) # l
```

Length of the string variable can be checked using a `len` function (example 3.2).

[1] via moduel `fractions`

Example 3.2. Checking the length of a string

```python
word = "apple"
print(len(word))
```

A string is an **immutable** type – its individual elements (characters) cannot be modified with an assignment statement, it also has a fixed length (example 3.3). The methods and functions that operate on a string **return a new variable** derived from the original. However, a string variable can be updated with an assignment statement and some elements of an existing string can be used to create a new one.

Example 3.3. Modyfying string variable in Python

```python
a = "Hello world"
a[1] = "b" # this operation will result in TypeError
a = a + "!"           # this will work
```

Mathematical operations on strings are not possible, with an exception of `+` and `*`. The `+` performs **string concatenation** – joins the strings by linking them end-to-end. The `*` performs repetition: `"a"*3` results with `"aaa"` (example 3.4). A string can be split into several parts with a `split` command, the result is saved to a list.

Example 3.4. String concatenation

```python
a = "Hello"
b = "world!"
print(a + " " + b)
print(4 * a)

c  = "a b 9"
print(c.split())
```

Objects can be converted to a string using `str` function (example 3.5). Objects are automatically converted to string representation for printing with `print` function. In example 3.2, the result of `len(word)` is a number, which was implicitly casted to a string.

Example 3.5. Casting to string

```python
a = "I am "
b = 50
c = " years old"
# number 50 needs to be casted to str in order to create
# one long variable with the rest of the string variables
sentence = a + str(b) + c
```

Some of the characters have special meaning, therefore, if we want to include them in strings, we need an escape character (\\). An example of the special character enclosed in a string variable may be a new line character (\n) or a tab character (\t). To include a literal backslash, we need to precede it with another backslash (example 3.6).

Example 3.6. Escape characters

```python
# escape characters:
print("A tab character can be signled with \\t") # prints \t instade of Tab space in a sentence,
     compare with line below
print("A tab character can be signled with \t")
print("Include backslash character \\")
```

If we want to put a quotes inside the string, use the other set of quotes (single quotes inside double quotes or the other way round, like in example 3.7).

Example 3.7. Using quotes in string

```
print("Shakespeare is the author of 'Hamlet'")
print('Shakespeare is the author of "Hamlet"')
```

Multiline strings are delimited with the use of triple quotes, like in example 3.8. In this case, newline characters included in the interpreter or Python string are preserved in a string. Multiline strings are often used for documentation or adding multiline input.

Example 3.8. Defining a multiline string

```
long_string = """First line
               second line
               third line
               """
another_long_string = '''First line
                      second line
                      third line
                      '''
```

An **empty string** is a string that has no charachters at all (`""` or `''`), but is perfectly valid and useful, for example, to build another string – an empty string can be used as a 'base' (example 3.9). An empty string is interpreted as `False` (it has `len` equal to 0).

Example 3.9. Using empty string

```
base = ""
number = 10.6
base += "Today's temperature: "
base += str(number)
```

3.1.1.1 String methods There are several methods which are characteristic of string variables. A **method** is similar to a function – it takes arguments and returns a value, but the syntax is different – it uses a dot notation (example 3.10). A method call is called an **invocation** (invoking a method on an object).

Example 3.10. Using a string method

```
a = "hello"
a = a.upper()  # this method takes no arguments, hence empty parantheses
print(a)
```

Some of the string methods include:

- `upper()`
- `lower()`
- `title()`
- `capitalize()`
- `swapcase()`
- `startswith()`
- `endswith()`
- `strip()`
- `lstrip()`
- `rstrip()`
- `split()`
- `join()`
- `count()`
- `index()`
- `find()`
- `reverse()`
- `replace()`
- `isdigit()`
- `isascii()`
- `isnumeric()`
- `isspace()`
- `islower()`
- `isupper()`

3.1.2 Lists

Lists are one of the Python's most useful types. Like a string, a list is an ordered collection of values. Unlike a string, a list can hold many values of any type, including integers, strings, tuples, file objects etc. A list can also include a list and become a **nested** list. A value in a list is called an **item** or an **element**.

A list is defined by enclosing its elements in square brackets (example 3.11). A list that does not have any elements is called an empty list.

Example 3.11. Creating lists

```python
simple_list = [1, 2, 4]
empty = []
many_elements = ["a", [2, 3], 4, 5, "aba"]
many_elements[1] = 8
print(many_elements[-1])
```

Lists are **mutable**, their elements and length can be modified. The syntax of accessing an item in a list is the same as for characters in a string – an index or a slice in brackets (see section 3.5). Indices start with 0. When a bracket operator appears on a left side of an assignment, it indicates which element of a list will be assigned. A list can be accessed from its front or back.

Indices works in the same way as in strings: an index can only be an integer. Trying to access an element (read or write) that does not exist will result in an error (`IndexError` to be precise).

Example 3.12. Creating new list from an existing list

```python
simple_list = [1, 2, 4]
new_list = simple_list
new_list[0] = "a"
```

An assignment of an already existing list to a new variable **does not create a new object**. It creates a new **reference** to an object already existing in the memory. Any changes made to a newly-created object will by reflected in the original list (example 3.12). To create an independent copy (a so-called **deep copy**), use statement `new_list = simple_list[:]` or `copy` method (example 3.13).

Example 3.13. Deep copy of a list

```python
simple_list = [1, 2, 4]
new_list = simple_list[:]
another = simple_list.copy()
another[1] = 0
print(simple_list)
print(another)
```

Similar to strings, `+` and `*` are defined for lists. `+` concatenates two lists, and the `*` operator repeats a list a given number of times (example 3.14).

Example 3.14. `+` and `*` operators for lists

```python
list_a = [1, 2, 3]
list_b = [4, 5, 6]
print(list_a + list_b)
print(list_a * 3)
```

Example 3.15. Matrix created as a list of lists

```python
matrix_example = [[1,2, 3], [4, 5, 6], [7, 8, 9]]
```

```python
print(matrix_example[1])            # print second row of the matrix
print(matrix_example[1][0])         # print first element of second row
```

Matrices can be represented as nested lists, where each row is an element of the list (example 3.15). However, matrices created as lists are rarely used for numeric operations. Another type called `ndarray`, an element of `NumPy` module will be discussed later.

3.1.2.1 List methods

Python provides many methods to operate on lists. Since writing to elements with non-existing indices is not possible, there are methods to add elements to a list (apart from the `+` operand): `append` and `extend` (example 3.16).

Example 3.16. Adding elements to list: `append` and `extend`

```python
list_a = ["a", "b"]
t.append("c")         # in-place method, does not require an assignment
list_b = ["d", "e"]
list_a.extend(list_b) # works as list_a + list_b
list_a.append(list_b) # adds one element only, a list (because list_b is a list)
```

`sort` method arranges elements in a list from low to high (example 3.17). Most of the list methods are **void**, which means they modify the list in place (do not create a new variable) and return a `None` value. In other words, using `sort()` modifies a list permanently and we cannot revert the list to the original order. Sorting in reverse order requires passing an argument `reverse=True` to the `sort()` method. The second function, `sorted`, do not sort the elements permanently – it returns a new object (does not affect the actual order of the list). If this object is not assigned to a variable, the result of sorting is lost.

Example 3.17. Sorting list in-place

```python
list_a = ["c", "d", "a", "e", "b"]
list_a.sort()         #in-place method, returns None
print(list_a)
list_a.sort(revers=True) # reverse order

list_a_sort = sorted(list_a) # built-in function, returns sorted list
```

Adding up elements of a list is a common operation, there is a built-in function `sum` to sum all elements in a list (example 3.18).

Example 3.18. Summing the elements of the list

```python
list_a = [1, 2, 5, 7]
list_sum = list_a.sum()
```

To delete an element from a list, several methods can be used. If the index of an item, that needs to be deleted is known, a `pop` method can be used (example 3.19). A `pop` method without an index deletes the last element in a list. This method modifies the list and returns the deleted item. If the deleted value is not needed, a `del` operator can be used.

Example 3.19. Deleting an element with a known index

```python
list_a = [1, 2, 5, 7, 4, 2, 11]
popped = list_a.pop(1)       # pop value 2, an element with a index 1
popped_last = list_a.pop()

del list_a[1]
```

If the index of the element is not known, it is possible to remove from the list the first element of a given value (example 3.20). `remove` method returns `None`.

Example 3.20. Deleting an element with an unknown index but known value

```
list_a = [1, 2, 5, 7, 4, 2, 11, 5]
list_a.remove(5)             # removes a value 5 with an index 2
```

It is important to remember that most list methods modify the argument and return `None`, unlike string methods, which return a new object and do not modify the original (due to mutable and immutable nature of a list and string object).

Reading the documentation or testing bits of code can prevent from errors in your program. If you need to sort your list, but also need the original, make a copy of the list before sorting.

Make sure to familiarise yourself with the list methods listed below:

- `append`
- `extend`
- `insert`
- `pop`
- `remove`
- `clear`
- `copy`
- `count`
- `index`
- `reverse`
- `sort`

3.1.2.2 Creating a list from a string

Creating a list from any other sequence is quite easy: requires only a `list()` function. If used on a string variable, the `list` function breaks a string into individual characters (example 3.21).

Example 3.21. Casting string to a list

```
word = "hello"
list_w = list(word) # returns ["h", "e", "l", "l", "o"]
```

Another method to create a list from a string is `split`, which separates the characters in a string into items of a list, based on a defined delimiter (default is whitespace). The inverse of `split` is `join` method – using a list of strings (or characters), it merges the elements and puts a delimiter in between (example 3.22). However, counterintuitive, the `join` method is invoked on the **delimiter** and the merged elements are passed as a parameter.

Example 3.22. Splitting and joining a string

```
word = "hello.how.are.you"
list_w = word.split(".")

# reverse
merged = "+".join(list_w)
```

3.1.3 Tuples

Similarly to a list, a tuple is a sequence of arbitrary objects (of various data types) separated by commas and enclosed in parentheses (although, the parentheses are not necessary). Items in a tuple are indexed with integers, starting with 0. Unlike lists, **tuples are immutable**, which means they cannot be modified after they are created. Use tuples to store a set of values that should not be changed throughout the program run.

If a tuple contains only one object, a final comma is required (to signalize a tuple) (example 3.23). A list (or any other object) can be converted to a tuple using a `tuple` function. A tuple can be cast to a list using a `list` function.

Example 3.23. Creating tuples

```
empty_tuple = tuple() # or ()
single_element = (1, )
another_single = "a",
many_elements = "a", (2, 3)  # tuple packing, both strings and a tuple
letter, numbers = many_elements # unpacking the tuple
tuple_from_list = tuple([3, 4, 5, 6]) # create a tuple from a list
list_from_tuple = list(tuple_from_list) # create a list from a tuple
```

If a sequence (string or a list) is passed as an argument to `tuple` function, a tuple containing individual elements will be created (example 3.24).

Example 3.24. Creating tuples from other sequences

```
string_tuple = tuple("apple")
list_tuple = tuple(["a", 1, 2, ["b", "g"]])
```

Tuples support most of the list methods, the elements are indexed with a bracket operator. Since tuples are immutable, an assignment to one of the elements with a bracket operator will raise an error. However, a new value can be assigned to a variable that holds a tuple – to change the values in the tuple, a redefinition (reassignment) of the variable is needed. Indices can only be used to access the element, not to modify it.

Due to their immutability, tuples have only two methods: `count` and `index` (example 3.25).

Example 3.25. Tuple methods

```
# define a tuple
t = (4, 5, 6, 9, 4, 4, 5)
t.count(4) # returns a number of occurences of number 4
t.index(9) # returns an index of the value
```

A useful feature of tuples is **tuple assignment** (or **tuple unpacking**) – an elegant solution that enables, among others, swapping values of two variables without using a temporary assignment (example 3.26 and example 3.27). Each variable on the right side of the equal sign is assigned to its correspondent on the left side. All the expressions on the right side are evaluated before assignment. The number of the variables on both sides must be equal.

Example 3.26. Unpacking a tuple

```
letter, numbers = ("b", (3, 4))
```

Example 3.27. Tuple assignment

```
b, a = a, b
```

In general, the right side can be any kind of sequence (even a string or a list).

3.2 Dictionaries

A dictionary contains a collection of indices, called **keys**, each associated with **a value**. This association is called **key-value pair** or an item. A dictionary represents a **mapping** from keys to values – each key maps to a value. A mapping can be interpreted as a relationship in which each element of one set corresponds to an element in another set. Both numbers and strings can be used as keys. Basically, immutable types should be used as keys (that includes tuples and boolean values). Keys need to be hashable, which means they need can be transformed to an integer value by a **hash** function. If a key changes, then the hash function result changes and finding a key may become impossible. **Keys has to be unique.** If a key is used more than once, the last entered value is remembered. Values in the dictionary can be of any type (numbers, strings, tuples, lists, dictionaries).

A dictionary is also an iterable type, but instead of integer indices (as in a list), the elements are accessed using keys.

To create a new dictionary you can use a function `dict()`. The {} represent an empty dictionary. To add items to a specific key in a dictionary, even a non-existing one, use square brackets with a key (example 3.28) and an assignment operator. Passing a tuple list to `dict` function is another way to create a dictionary.

Example 3.28. Creating a dictionary

```
dictionary = dict()          # creating an empty dictionary, also dictionary = {}
# many element at once
dictionary = {"el1" : 12.5, "el2" : 2.1, 3 : "a"} # keys may be numbers or strings
dictionary["el4"] = (12.5, 8) # adding elements to a dictionary: assign a value to a new key
```

Dictionaries are not ordered alphabetically and generally do not preserve the order in which the elements were added. The order is not relevant, since the elements are not accessed via integer indices interpreted as the offset from the beginning, but via keys. Referring to a key that is not in a dictionary, will raise an error (`KeyError`). To avoid this error, you may find a `get()` method particularly useful. If a key appears in a dictionary, `get` returns the corresponding value or else the default value (example 3.29).

Example 3.29. `get` method

```
dictionary = {"a" : 12.5, "b" : 2.1, "c" : 10.0}
x = dictionary.get("a", 0)
y = dictionary.get("d", 0)
```

`len()` function works on a dictionary, it returns a number of key–value pairs. To see the list of a values stored in a dictionary, invoke `values()` method on the object (example 3.30). All the keys may be acquired with a `keys()` method. `items()` returns sequence of tuples containing key–value pairs.

Example 3.30. Getting a list of keys, values and items stored in a dictionary

```
dictionary = {"a" : 12.5, "b" : 2.1, "c" : 10.0}
print(dictionary.values())
print(dictionary.keys())
print(dictionary.items())
print(len(dictionary))
```

Removing an element in a dictionary requires using a `del` statement (example 3.31), which has to be provided with the name of the dictionary variable and a key.

Example 3.31. Removing an item in a dictionary

```
dictionary = {"a" : 12.5, "b" : 2.1, "c" : 10.0}
del dictionary["b"]
```

3.3 Sets

A **set** is used to contain **an unordered collection of unique objects**. When a set is created from a list of values, unique values are identified and duplicates are not added to a set. The values placed into a set must be immutable. Set itself is a mutable type[2], elements can be added and removed. Hence, a set cannot be included in a set. Sets are usually used to test membership or uniqueness.

[2]There is also an immutable equivalent in the standard library called **frozenset**

To create a set, either curly braces or a `set()` function can be used (example 3.32). However, to create an empty set, only `set()` would give desired result as a pair of curly braces creates an empty dictionary. Because sets are unordered, the elements cannot be indexed by numbers.

Example 3.32. Creating a set

```
set_1 = {"a", "b", "b", "c"} # set creation, duplicates will be removed
print(set_1)
set_empty = set()
set_string = set("unique set of characters") # set from individual characers
"s" in set_string            # fast membership testing
```

Example 3.33. Set operations

```
a = {"a", "b", "b", "c"}     # duplicates will be removed
b = {"c", "d", "a"}
a + b
```

Basic usage include membership testing and eliminating duplicate entries. Sets support also operations known from mathematics: union, intersection, difference and symmetric difference. The following set-specific operators are available (`a` and `b` are sets in example 3.33):

- `a | b` or `a.union(b)` – union: all elements from both `a` and `b`,
- a - b or `a.difference(b)` – set difference: elements in `a` and not in `b`,
- a ˆ b or `a.symmetric_difference(b)` – symmetrical difference: elements in `a` or `b`, but not both,
- a & b or `a.intersection(b)`– intersection: elements in both `a` and `b`;
- `a.isdisjoint(b)` – returns `True` if `a` and `b` have no items in common;
- `a.issubset(b)` – returns `True` if `a` is a subset of `b`;
- `a.issuperset(b)` – returns `True` if `a` is a superset of `b`.

New items can be added to a set using `add` or `update` (example 3.34). Removing an item requires, as usual, `remove` function.

Example 3.34. Adding elements to a set

```
a = {"a", "b", "b", "c"}     # duplicates will be removed
a.add()"e") # add a single item
a.update(["f", "g", "g"]) # add multiple items
a.remove("g")
```

3.4 Iterable variable methods

There are some methods that work on all basic sequence types as well as on dictionaries and sets. It is useful to remember at least couple of those:

- `len(x)` – returns length of the variable `x` (numbers of items in x),
- `max(x)` – returns a maximum item included in `x`,
- `min(x)` – returns a minimum item included in `x`,
- `sum(x)` – sums all values in `x`,

- `sorted(x)` – returns a sorted list of the elements in sequence `x` (by default, in ascending order),
- `del x` or `del x[n]` – delete a variable / an item from a variable with index `n`,
- `all(x)` and `any(x)` – checks weteher all / any value in `x` is `True`.

3.5 Indexing and slices

A segment of a collection, whether it is a string variable, a list or a tuple, is called a **slice**. Selecting a slice is not much different from selecting a single element. The operator `[m:n]` returns the part of a variable from the `m`-th character to the `n`-th, **including** the character with an index `m`, but **excluding** the element with index `n` (example 3.35). If the first index is omitted (before the colon), the slice starts at the beginning of a variable (index 0). If the second index is omitted, it goes to the end of the sequence.

Example 3.35. Slicing

```python
word = "workout"
# choose part "work" in workout
w_slice = word[0:4]          # the same as word[:4]
# choose part "out" in workout
w_slice = word[4:7]          # the same as word[4:], last index is 6, but len(word)=7
w_slice = word[4:-1] # choose elements from index 4, but leave the last element out: "ou"
```

If the first index is greater than the second, the result is an empty item (a variable of the same type with a length 0).

Slices can take the third optional argument which is a size of a step between successive characters: `[m:n:step]` (example 3.36). A step with a negative value returns elements of a sequence in reversed order starting from `m` to `n+1`, in this case $m > n + 1$. The slice `[::-1]` inverses the sequence.

Example 3.36. Slicing with an interval

```python
word = "workout"
w_slice = word[0:4:2]        # the same as word[:4:2] -> returns "wr"
# choose every third letter in "workout"
w_slice2 = word[::3]         # returns "wkt"
w_slice2 = word[::-2]        # choose every second letter, but backwards -> "torw"
w_slice3 = word[:]       # select all elements, creates a copy
```

All slice operations return a new object (list, string) containing the requested elements (**shallow copy**). For lists, assignments to slices are also possible (example 3.37).

Example 3.37. Slicing assignment

```python
word_list = list("workout")
word_list[2:5] = "a" # results in ['w', 'o', 'a', 'u', 't'], all three elements are substituted
    with one element only
word_list[1:len(word)] = [] # clears elements from index 1 to the end of the list, after the
    operation, only one element left
```

3.6 The `in` operator

The keyword `in` is a boolean operator that takes two elements, checks whether the first one is an element of the second variable: returns `True` if the element is found and `False` otherwise (example 3.38).

Example 3.38. Using the `in` operator

```python
word = "workout"
print("work" in word)

example_list = [1, 2, 4, 5, 6]
print(3 in example_list)
print(6 in example_list)
```

In case of dictionaries, `in` operator returns `True` if a value is one of the keys in the dictionary. Python dictionaries use a structure called a **hashtable**, the `in` operator takes the same amount of time to find an element, regardless of the number of keys stored. The dictionary keys are in no particular order. To browse keys in a sorted matter, use a `sorted` function (example 3.39).

Example 3.39. Sorting a dictionary

```python
dictionary = {"a" : 12.5, "b" : 2.1, "c" : 10.0}
for key in sorted(dictionary):
    print(key, dictionary[key])
```

3.7 `range` function

A `range` function generates a sequence of numbers in an arithmetic progression. **The given endpoint is never part of the generated sequence** (example 3.40). The range object is **iterable**. Ranges are useful in `for` loops. Also, combining them with a `len` function enables an iteration the same number of times as elements in a list (example 3.41). The advantage of the `range` is that it does not use much of computer memory, but still enables creating huge ranges of numbers. The only required value is a stop value (endpoint). As with slices, the step of the sequence can be either a negative or a positive value (negative goes backwards).

Example 3.40. Defining ranges with different step

```python
# example_range = range([start_point], endpoint, [step])
range(0, 10) # numbers from 0 to 9 (included)
range(5) # numbers from 0 to 4 (indcluded)
range(10, 20, 2) # (every second number starting from 10: 10, 12, 14, ..., 18)
range(-3, -200, -20) # [-3, -23, -43, -63, -83, -103, -123, -143, -163, -183]
```

Example 3.41. Combining `range` with `len`

```python
list_a = ["today", "is", "a", "good", "day"]
for i in range(len(list_a)):
    print(i, a[i])
```

To see all the elements of the range, a range have to be casted to a different data type, e.g. a list, using a `list` function (example 3.42).

Example 3.42. Casting a range to a list

```python
a = list(range(0, 10))
```

3.8 Comprehensions

List and dictionary comprehensions provide a concise way to create new objects according to a defined rule. The common usage is to process each element of an iterable with some operations, or to create a subsequence of the elements that satisfy a certain condition.

The bracket operator (or curly bracket operator in case of creating a new dictionary) indicates that a new list is created. The expression inside, specifies the elements of the list, and a `for` clause indicates which sequence is traversed (examples ?? and 3.44). The `for` clause can be followed by zero or more `for` or `if` clauses (example 3.45).

Example 3.43. List comprehension

```
squares = [x ** 2 for x in range(10)]

word = "skeleton"
word_cap = [letter.upper() for letter in word]
```

Example 3.44. Dictionary comprehension

```
squares = {x : x ** 2 for x in range(10)}
```

Example 3.45. More complex list comprehension

```
pairs = [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
# result: [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

List and dictionary comprehensions can be used to filter elements from a sequence. In example 3.46, conditional expression (`if` statement) selects only elements with keys longer than 4 characters and returns a new dictionary.

Example 3.46. Dictionary comprehension filtering the results

```
dictionary = {"spam": 1, "spamspam": 2, "spamspamspam": 3}

long_keys = {}
for k, v in dictionary.items():
    if len(k) > 4:
        long_keys[k] = v

# the same dictionary using a dictionary comprehension
new_keys = {k : v for k,v in dictionary.items() if len(k) > 4}
```

Python comprehensions are easy to read and usually faster (and more compact) than their equivalents written with `for` loops. However, they are harder to debug, because it is not possible to put a `print` statement inside.

Nested comprehensions are permitted (example 3.47), usually based on a nested data structure.

Example 3.47. Nested list comprehension

```
matrix = [
        [1, 2, 3, 4],
        [5, 6, 7, 8],
        [9, 10, 11, 12],
        ]
# transpose rows and columns
transpose = [[row[i] for row in matrix] for i in range(4)]
```

```python
#result: [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

# traditional traversing with double loop
transpose = []
for i in range(4):
        new_row = []
        for row in matrix:
                new_row.append(row[i])
        transpose.append(new_row)
```

Set comprehensions are also available. They use the same curly brackets as dictionary comprehensions. Both the standard and longer versions (including `if` tests or multiple `for clauses`) are valid (example 3.48).

Example 3.48. Set comprehension

```python
new_set_3 = set()
for i in range(10, 100):
        if i % 3 == 0:
                new_set_3.add(i)

# the same set using a set comprehension
new_set = {number for number in range(10, 100)} # all numbers in range 10-99
new_set_3 = {number for number in range(10, 100) if number % 3 == 0} # all numbers divisible by
    3
```

Tuples do not have comprehensions. Parentheses mark generator comprehensions, but this object will not be discussed at this time.

3.9 Choosing the best data structure for the task

When choosing a type of variable for the task, it is important to remember about their limitations. For example, strings can hold only characters and mathematical operations are rather impossible to perform. Lists are mutable and it is easy to add a new item or change the existing one. If a sequence of elements must be used as a key in a dictionary, tuple is the right choice. If passing a sequence as an argument to a function, using tuples reduces chances for unexpected behaviour due to aliasing (tuples cannot be modified).

Choice of data structure requires thinking about operations that will be performed on the variable (ease of implementation). Other aspect might be run time – how much time it takes to find an element in a sequence, or how long it takes to run through a sequence using an `in` operator (which is faster for dictionaries than for lists). The other factor might be storage space.

3.10 Exercises

Exercise 3.1. Using a `dir` function, check out the string, list, tuple and dictionary methods. Write single statements to see how each of them works. Try out various methods to create a variable of each type. Documentation for string methods can be found at `http://docs.python.org/3/library/stdtypes.html#string-methods`. Things to try:

- creating a sequence,
- adding elements to a sequence, adding elements to a specified index,
- mathematical operations (if possible),

- removing elements from a sequence,
- copying,
- checking if the element is included / not included in the sequence,
- change the order of the elements (reverse, indexing, sorting)
- getting the value of the element
- checking equality of the sequences (==, is) [3]

Exercise 3.2. Try to write a single-line version of a program checking if a word is a palindrome. A palindrome is a word that is is spelled the same forwards and backwards.

Exercise 3.3. Try out examples listed below in interactive mode. Can you tell before running those statements, which are correct and which are illegal (why?)?

- `t.append(x)`,
- `t.append([x])`,
- `t = t.append(x)`
- `t = t + [x]`,
- `t += [x]`,
- `t = t + x`,
- `t + [x]`

Exercise 3.4. Using a `range` function, create a list containing only odd numbers in range $\langle 0,\ 20)$.

- add 1000 to the penultimate element,
- change the first element (with index 0) to a string value `"index_0"`,
- compute the results of the basic arithmetic operations (+, -, /, *, * *, //) with a variable `b=2.5`,
- change the fourth element to a list: `["is", "today", "good", "a", "day"]`,
- using the modified list from the previous task, return a string (concatenated) `"good day"`.

Exercise 3.5. Using a `range` function, create a list containing 20 integer values. Use index and slicing methods to print:

- 5 elements at the beginning of the list,
- every second element,
- last 5 elements,
- every third element between indices 6 and 15,
- every fourth element between indices 6 and 15 in the reversed order.

-

[3]information on comparison:
`https://docs.python.org/3/tutorial/datastructures.html#comparing-sequences-and-other-types`

Exercise 3.6. Create a nested list. Print each element of the list using a `while` loop.

Exercise 3.7. Write a program computing the sum and the mean of the values provided by the user. Read the values from the user until '0' is provided. Print the results.

Exercise 3.8. Print the even numbers (starting from two) which sums to a value lower than n (n provided by the user).

Example: $n = 10$ result: 2, 4

Exercise 3.9. Compute the series sum S_m:

$$S_m = \sum_{i=1}^{m} ((i+1)^2 - 5)$$

Stop the computation, when the sum is over the limit (provided by the user or defined by the programmer). How many elements were added?

3.11 Useful links

- Strings: `https://docs.python.org/3/tutorial/introduction.html#strings`
- Lists (3.1, 5.1, 5.1.1, 5.1.2):
  - `https://docs.python.org/3/tutorial/introduction.html#lists`
  - `https://docs.python.org/3/tutorial/datastructures.html#more-on-lists`
- Dictionaries (5.5): `https://docs.python.org/3/tutorial/datastructures.html#dictionaries`
- Sets: `https://docs.python.org/3/tutorial/datastructures.html#sets`
- `range`: `https://docs.python.org/3/tutorial/controlflow.html#the-range-function`

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].

https://docs.python.org/3/tutorial/datastructures.html