Python: Lists

Introduction to Computer Programming
Bachelor in Data Science

Roberto Guidi

roberto.guidi@supsi.ch

Fall 2021

SUPSI University of Applied Sciences and Arts
of Southern Switzerland

Content created in collaboration with: L. Grossi, F. Landoni

Variables

In Python, in order to create a variable, it is necessary to define the identifier (name) that we want to use.

In contrast to other languages, it is not needed to specify the data type of the variable. Since it is a dynamically typed language, Python will figure out automatically the data type.

When declaring a new variable, it is also necessary to initialize it with a value.

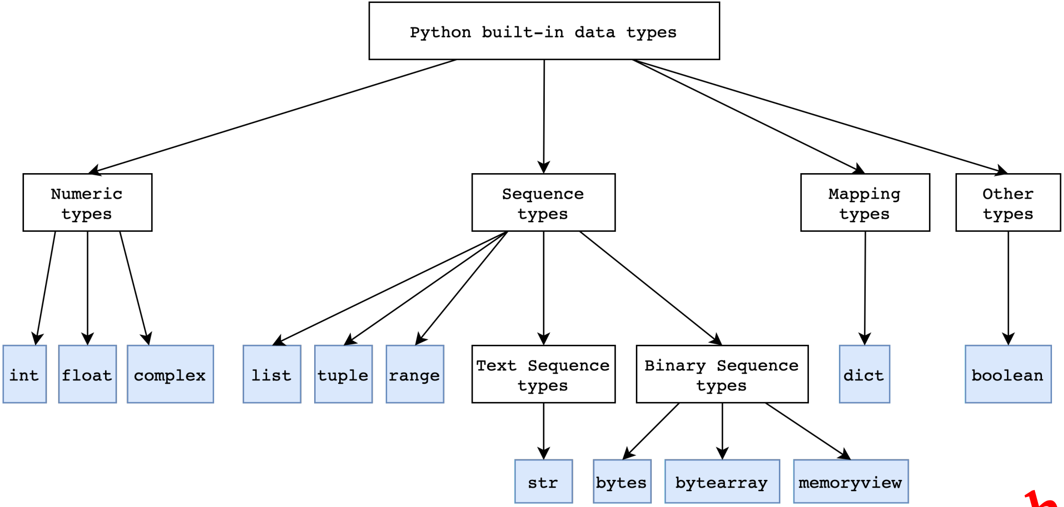
```
message = "my message"
print(message)
```

The variable is then accessible, using the identifier, from the moment of the declaration until the end of the code block in which it is contained.

It is possible to redefine the value held by a variable at any time.

Refresher

Data types



Refresher

Data types

One common way to organize data types is that of dividing them in these categories:

- Non structured data types: each identifier refers to a single and unique element.

  Example: `int, float, complex, boolean, ...`
- Structured data types: using a single identifier it is possible to access to compound elements.

  Example: `lists, dictionaries, tuples, ...`

Lists

One of the most used data types in Python is the List.

It is a structured data type that allows us to store an ordered collection of objects (sequence).

Lists allow to store and manage in an efficient way large amount of heterogeneus information.

Unlike strings, that are immutable sequences of characters, they can store objects of any sort like numbers, strings, other lists, . . .

Lists

The most important properties of lists in Python are:

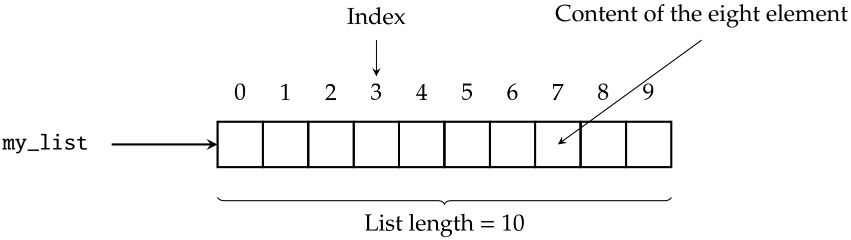
- Ordered collections: list maintain the positional order of the items they contain.
- Items accessed by index: using the offset from the list start as index, it is possible to fetch an item out of a list.
- Content can be any sort of object: it is possible to store as list item any kind of object.
- Content can change (they are mutable): lists can grow or shrink, item can be added, deleted or replaced.

Lists: creation

The simplest way to create a list is to enclose values (objects) in square brackets `[]`
A few examples:

```python
# an empty list
L = []

# a list containing integers
L = [1,2,3]

# a list containing strings
L = ["one", "two", "three"]

# a list containing multiple types
L = ["one", 1, True, 3.5]
```

Lists: internal structure



The length of the list is equal to the number of elements that it contains.
The statement `len(my_list)` returns the current length of the list.

In Python, the numbering of the elements always start from 0:

- the index of the first element is 0
- the index of the last element is *N-1* where *N* is the length of the list.

Lists: basic usage

In order to access the items of a list we can use the following notation:

```
list_variable_name[index]
```

Example:

```
my_list = ["How", "are", "you?"]
first_el = my_list[0]
second_el = my_list[1]
third_el = my_list[2]
```

Remember that:

- The numbering of the `index` starts from 0
- The value of the `index` must be an integer value.

Lists: example 1

Without using list, multiple variables:

```python
if __name__ == '__main__':
    x1 = 1000
    x2 = 50
    x3 = 234
    x4 = 12000
    x5 = 70
    sum_result = x1 + x2 + x3 + x4 + x5
    print(f"Sum: {sum_result}")
```

Output:

```
Sum: 13354
```

Lists: example 2

Using a list:

```python
if __name__ == ’__main__’:
    x = [1000, 50, 234, 12000, 70]
    sum_result = x[0] + x[1] + x[2] + x[3] + x[4]
    print(f"Sum: {sum_result}")
```

Output:

```
Sum: 13354
```

Lists: example 3

Using a list and a loop:

```python
if __name__ == '__main__':
    x = [1000, 50, 234, 12000, 70]

    sum_result = 0
    for i in range(len(x)):
        sum_result += x[i]

    print(f"Sum: {sum_result}")
```

Output:

```
Sum: 13354
```

Lists: in-place change (index assignment)

To modify an element at a given index inside a list it is possible to simply use the assignment operator =.

Example:

```
>>> my_list = ["How", "are", "you?"]
>>> my_list[0] = "Who"
>>> my_list
['Who', 'are', 'you?']
```

Lists methods

Lists allow a variety of operations to be carried out.

Some of the most important operations are insertion, deletion and sorting of list items.

This operations are made available by method calls on list objects.

In the following slides we will go through a selection of useful methods.

A description of all the list methods is available on the Official Documentation

Lists methods: append

To add an item at the end of a list we can use the `append(item)` method.

Example:

```
>>> my_list = [1,2,3]
>>> my_list.append(4)
>>> my_list
[1,2,3,4]
```

Lists methods: insert

To insert an item at a given position of a list we can use the `insert(index, item)` method.

Example:

```
>>> my_list = ["a", "b", "c"]
>>> my_list.insert(1, "aa")
>>> my_list
[’a’, ’aa’, ’b’, ’c’]
```

Lists methods: pop

To delete an item at a the end of a list we can use the pop() method.
The deleted item is returned by the method.

Example:

```
>>> my_list = ["a", "b", "c"]
>>> item = my_list.pop()
>>> my_list
['a', 'b']
>>> item
'c'
```

The pop method also optionally allows to specify an index as deletion target.

```
>>> my_list = ["a", "b", "c"]
>>> item = my_list.pop(1)
>>> my_list
['a', 'c']
```

Lists methods: remove

To delete an item by value from a list we can use the `remove(value)` method.

Example:

```
>>> my_list = ["a", "b", "c"]
>>> my_list.remove("a")
>>> my_list
['b', 'c']
```

Lists methods: reverse

To reverse the order of items contained in a list we can use the `reverse()` method.

Example:

```
>>> my_list = ["a", "b", "c"]
>>> my_list.reverse()
>>> my_list
['c', 'b', 'a']
```

Lists methods: sort

To sort the items contained in a list we can use the `sort()` method.

Python uses comparison tests to determine the ordering. By default, sorting is made in ascending order.

```
>>> my_list = [3, 8, 4, 11, 1, 9]
>>> my_list.sort()
>>> my_list
[1, 3, 4, 8, 9, 11]
```

It is possible to reverse the sorting ordering using a special argument *reverse=True*.

```
>>> my_list = [3, 8, 4, 11, 1, 9]
>>> my_list.sort(reverse=True)
>>> my_list
[11, 9, 8, 4, 3, 1]
```

Lists: sequence operations

Length of a list: `len(list_variable)`

```
>>> my_list = [1,2,3]
>>> len(my_list)
3
```

Concatenation of lists: +

```
>>> my_list1  = [1,2,3]
>>> my_list2 = [4,5,6]
>>> my_list1 + my_list2
[1, 2, 3, 4, 5, 6]
```

Repetition: *

```
>>> my_list1  = ["Ni!"]
>>> my_list2 * 4
["Ni!", "Ni!", "Ni!", "Ni!"]
```

Lists: sequence operations

Since a string is a sequence of characters, it can be converted to a list.

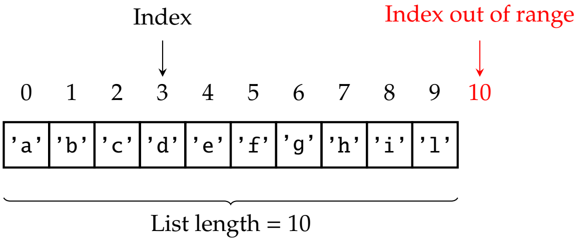
The conversion to a list can be done using the `list(string_variable)` function.

```
>>> my_string = "hello!"
>>> list(my_string)
['h', 'e', 'l', 'l', 'o', '!']
```

Lists: index boundaries

Be careful when accessing the elements of a list:

If we try to access an element k with $k \geq length$, the program will interrupt with an `IndexError`: list index out of range.



Index

Index out of range

0 1 2 3 4 5 6 7 8 9 10

'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'l'

List length = 10

Lists: index boundaries

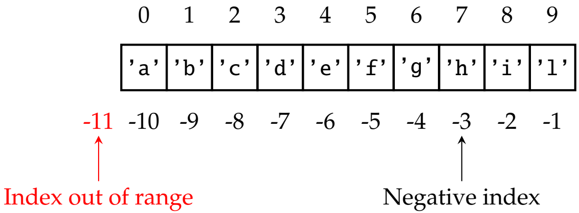
Example:

```
my_list = ["How", "are", "you?"]

# no problem
el = my_list[1]

# IndexError, index is >= len(my_list)
el = my_list[4]

# IndexError, index is >= len(my_list)
el = my_list[len(my_list)]
```

Lists: negative index boundaries

If we use a negative index what happens is that the count will be made from right (end of the list). The last element is has negative index -1.

If we try to access an element k with $k < -length$, the program will interrupt with an `IndexError:` list index out of range.



	0	1	2	3	4	5	6	7	8	9
	'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'	'i'	'l'
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Index out of range

Negative index

Lists: negative index boundaries

Example:

```
my_list = ["How", "are", "you?"]

# el is "you?"
el = my_list[-1]

# el is "how?"
el = my_list[-3]

# IndexError, index is < -len(my_list)
el = my_list[-4]
```

Lists: slices

It is often useful to extract a slice (a portion) from a list.

In python we can use the slice operator :

The syntax of this operators is the following `[start:stop:step]`.

This will allow us to get the piece of list from index n to index m, including the first but excluding the last and with the defined *step*.

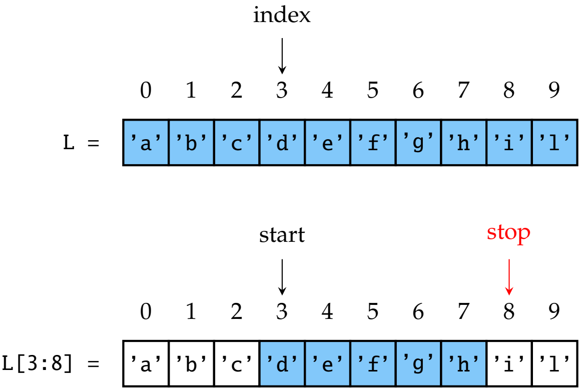
Note that a slice of a list is also a list.

It is possible to slice with both positive and negative indexes. Omitting *start*, *stop*

or *step* has a particular behavior as we will see later.

Lists: slices with positive indexes

```
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'l']
print(L[3:8])
# prints ['d', 'e', 'f', 'g', 'h']
```



index

0 1 2 3 4 5 6 7 8 9

L = 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'l'

start stop

0 1 2 3 4 5 6 7 8 9

L[3:8] = 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'l'

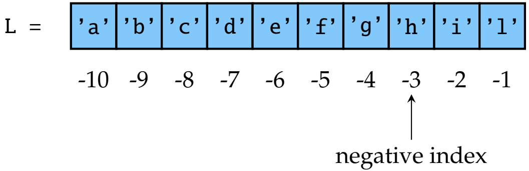
Lists: slices with positive indexes example

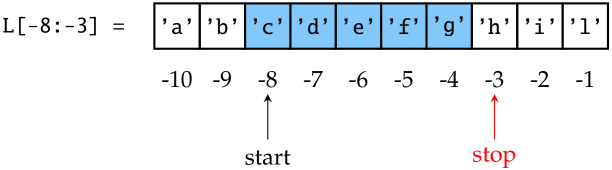
A few examples:

```
my_list = ["How", "are", "you?"]
print(my_list[0:1])
# prints [’How’]

print(my_list[0:2])
# prints [’How’, ’are’]

print(my_list[1:2])
# prints [’are’]

print(my_list[0:3])
# prints [’How’, ’are’, ’you?’]
```

Lists: slices with negative indexes

```
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'l']
print(L[-8:-3])
# prints ['c', 'd', 'e', 'f', 'g']
```





Lists: slices with negative indexes example

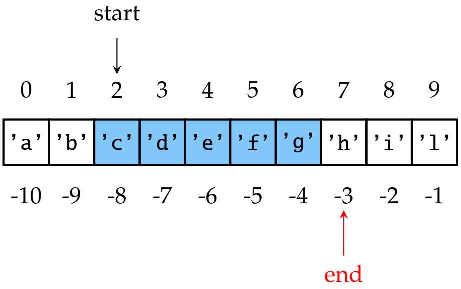
A few examples:

```
my_list = ["I", "am", "fine", "thanks!"]
print(my_list[-3:-1])
# prints [’am’, ’fine’]

print(my_list[-3:-2])
# prints [’am’]

print(my_list[-4:-2])
# prints [’I’, ’am’]

print(my_list[-2:-1])
# prints [’fine’]
```

Lists: slices with positive and negative indexes

It is possible to specify both positive and negative indexes at the same time.

```
L = [’a’, ’b’, ’c’, ’d’, ’e’, ’f’, ’g’, ’h’, ’i’, ’l’]
print(L[2:-3])
# prints [’c’, ’d’, ’e’, ’f’]
```



start

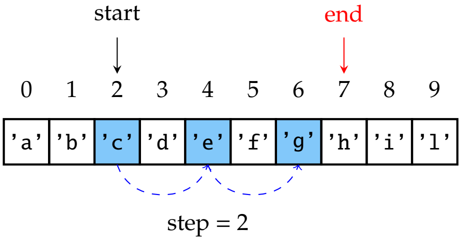
0	1	2	3	4	5	6	7	8	9
’a’	’b’	’c’	’d’	’e’	’f’	’g’	’h’	’i’	’l’
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

end

Lists: slices with step

As seen in the examples so far, it is possible omit the step, which is 1 by default. The step can be used as follows in order to control the increment used when slicing.

```
L = [’a’, ’b’, ’c’, ’d’, ’e’, ’f’, ’g’, ’h’, ’i’, ’l’]
print(L[2:7:2])
# prints [’c’, ’e’, ’g’]
```

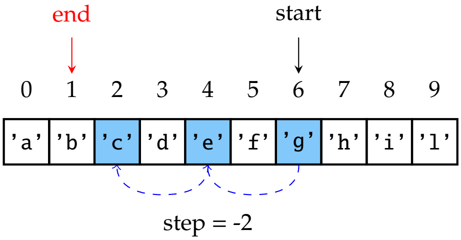


0	1	2	3	4	5	6	7	8	9
’a’	’b’	’c’	’d’	’e’	’f’	’g’	’h’	’i’	’l’

Lists: slices with negative step

It is also possible to specify a negative step value. In this case the list will be sliced from right to left with the desired step.

```
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'l']
print(L[6:1:-2])
# prints ['g', 'e', 'c']
```



Lists: slice at beginning and end

As mentioned before, if we omit the *start* and/or *end* values, the slice will behave in a particular way.

If we omit the *start* index, the default 0 will be used.

Therefore, writing `L[0:5]` or `L[:5]` is equivalent.

```
L = [’a’, ’b’, ’c’, ’d’, ’e’, ’f’, ’g’, ’h’, ’i’, ’l’]
print(L[:5])
# prints [’a’, ’b’, ’c’, ’d’, ’e’]
```

Lists: slice at beginning and end

When omitting the *end* index, the behavior is very similar as the case seen before.

By default, the *end* value used is *len(L)*.

Therefore, writing `L[3:len(L)]` or `L[3:]` is equivalent.

```
L = [’a’, ’b’, ’c’, ’d’, ’e’, ’f’, ’g’, ’h’, ’i’, ’l’]
print(L[3:])
# prints [’d’, ’e’, ’f’, ’g’, ’h’, ’i’, ’l’]
```

Lists and loops

One of the most common task that is performed on a list is that of iterating through its items, one by one, from the first to the last.

This can be achieved in many different ways.

In Python, since a list is a sequence type, the most simple approach to loop items is using the following syntax:

```python
for item in my_list:
    # do something with the item
    print(item)
```

This approach is comparable with the `foreach` concept in other languages.

Lists and loops: basic example

Print all the items present in a list:

```
>>> my_list = ["a", 5, "b", 7,  "c", 10]
>>> for item in my_list:
>>>     print(item)

a
5
b
7
c
10
```

Lists and loops: enumeration

With the basic approach seen we can loop through the items but we do not have the index available at each iteration.

However sometimes it is useful to have access both to the item and its index (offset) at the same time.

For this purpose we can use the built-in function `enumerate(list_variable)` which will return us at each iteration the current index and item as in the next example.

```
>>> my_list = ["a", 5, "b", 7,  "c", 10]
>>> for index, item in enumerate(my_list):
>>>     print(f"item {item} at index {index}")

item a at index 0
item 5 at index 1
item b at index 2
item 7 at index 3
```

Lists and loops: loop with manual index

Another possibility to loop items in a list is by manually generating the indexes.

In this case we can use the `range()` function as already seen.

Example:

```
for i in range(len(my_list)):
        # do some cool stuff
        print(my_list[i])
```

This approach is similar to how we would iterate array items in other languages like Java or C.

Anyway it is not the preferred and usual way to iterate items in python.

Lists and loops: examples loops with manual index

Code that sums all the numbers contained in a list:

```python
my_list = [1, 5, 3, 10, 7]
# cycle the elements of the list
sum_result = 0
for i in range(len(my_list)):
    sum_result += my_list[i]
```

Code to find the highest value in a list:

```python
x = [1, 7, 6, 9, 11, 5, 4]

max_val = x[0]
for i in range(1, len(x)):
    if x[i] > max_val:
        max_val = x[i]
```

Lists: copy

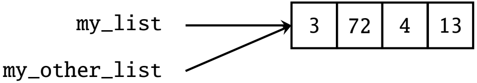
Let suppose that we have created a list as follows:

```
my_list = [3, 72, 4, 13]
```

If we want to copy the list we might think to do:

```
my_other_list = my_list
```

What we have done is simply:



Basically we just copied the reference to the list, not the actual list.
The operation done is only an assignment statement.

Lists: deep copy

If we really want a copy of the list and its content, we need to perform what is known as deep copy.

For this purpose, if the items in our list are not compound objects (not other lists, dictionaries, . . .), we can use for example the slicing operator.

```
my_list = [3, 72, 4, 13]
my_other_list = my_list[:]
print(my_other_list)
# prints [3, 72, 4, 13]
print(my_other_list is my_list)
# prints False
```

There are also other methods to perform a deep copy.
We will look at them later when talking about classes and objects.

Lists: comparison

In order to find out if the content of a list is the same as another one you can use the == operator.

Lists are compared in Python by comparing each item from left to right.

This is done recursively in case of nested structures, until the end of the first mismatch.

Do not use the operator is for this purpose since it just checks for reference equality (identity).

Please note that in other programming languages the == does the same thing as Python's is operator.

Lists: comparison example

```
if __name__ == ’__main__’:
    my_list = [3, 72, 4, 13]
    my_list_2 = [4, 5, 46, 13]
    print(my_list == my_list_2)
    # prints False
    my_list_2 = [3, 72, 4, 13]
    print(my_list == my_list_2)
    # prints True
```

Lists: check if an element is contained in a list

In order to find out if an element is contained in a list, it is possible to use the `in` keyword.

The syntax is as follows `value_to_find in my_list`.

This statement will return a `boolean` value.

```
>>> characters = [’darth vader’, ’obi-one’, ’yoda’]
>>> ’darth vader’ in characters
>>> True

>>> ’luke skywalker’ in characters
>>> False
```

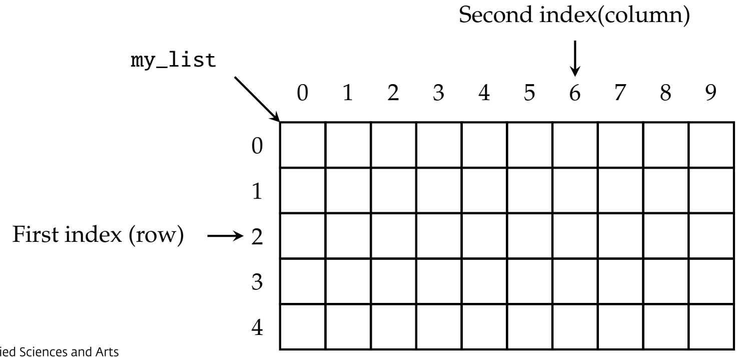
Another way to achieve the same result would be to iterate through the list and compare the current item manually.

Nested Lists

By nesting lists (putting them one inside the other), it is possible to define multi-dimensional structures such as matrixes.

To access data in such structures, multiple indices will be needed.

Example: a bi-dimensional list (two indexes)



Bi-dimensional Lists

In Python, a multi-dimensional (or nested) list is created by declaring a list that contains other lists as items.

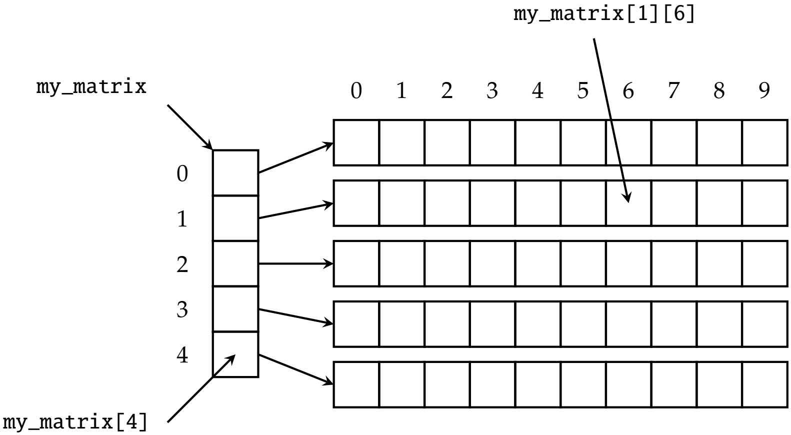
To access a particular element in the bi-dimensional list we need to use two indices.

The syntax is `my_list[i][j]`

In the following example we create a nested list with 3 rows and 3 columns.

```
my_list = [[1, 2, 3], [4, 5, 6], [6, 7, 8]]
print(my_list[0][0])
# prints 1
print(my_list[1][1])
# prints 5
print(my_list[2][2])
# prints 8
```

Bidimensional Lists



Bidimensional Lists: example with indices

```python
if __name__ == '__main__':

    my_matrix = [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]

    for i in range(len(my_matrix)):
        for j in range(len(my_matrix[i])):
            print(my_matrix[i][j])
```

In order to traverse the multi-dimensional list, we use nested loops

Bidimensional Lists without indexes

As discussed previously, when using the statement `range(n)` we are generating a sequence of numbers that we can use in our loops.

Loops can be used in general with any iterable (sequence-like) object.

Since the list is an iterable object, we can avoid indexes to cycle through rows and columns as in the following example:

```python
if __name__ == '__main__':

    my_matrix = [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]

    for row in my_matrix:
        for col in row:
            print(col)
```

Summary

- Lists
- Lists methods
- Sequence operations (`+, *, ...`)
- Index boundaries (`positive index, negative index`)
- Slices statement (`positive index, negative index, step, beginning and end`)
- List and loops
- List copy and comparison
- Nested lists (`bi-dimensional lists`)

Bibliography

- Learning Python 5th edition, Oreilly - Mark Lutz: Chapters 4, 8
- Python Crash Course, no starch press - Eric Matthes: Chapters 3, 4
- Python Official Documentation: https://docs.python.org/3/tutorial/
- LearnByExample: https://www.learnbyexample.org/python/