Python Programming 1

(CP1S02005E)

Białystok University of Technology
Faculty of Electrical Engineering

Industry Digitization, semester II
Academic year 2024/2025

Lecture no. 04 (26.03.2025)

Jarosław Forenc, PhD

Topics

- Lists
  - implementation, creation methods, list comprehension
  - element indexes, slices
  - functions and methods
- Tuples
  - implementation, creation methods
  - element indexes, slices

Python - lists

- List - a mutable data structure that stores a collection of elements arranged in a specific order

```
numbers = [5, 2, 8, 3, 0, 2, 1]
functions = ["print", "input", "sort", "sqrt"]
```

  - list elements are placed inside square brackets and separated by commas
  - lists can contain elements of different data types (numbers, strings, other lists, tuples, dictionaries)
  - the elements of a list do not have to be related (but usually are)
  - lists are dynamic - elements can be added and removed during program execution

Python - tuples

- Tuple - an immutable data structure that stores a collection of elements arranged in a specific order (a list of elements that cannot be changed)

```
numbers = (5, 2, 8, 3, 0, 2, 1)
functions = ("print", "input", "sort", "sqrt")
```

  - tuple elements are placed inside regular parentheses and separated by commas
  - tuples can contain elements of different data types (numbers, strings, lists, other tuples, dictionaries)
  - they take up less memory than lists, making them more efficient when data is only meant to be read

Python - lists, creation methods

- using a pair of square brackets to denote an empty list

```
my_list = []
print(my_list)
```

```
[]
```

- using square brackets and separating elements with commas

```
names1 = ["Jane"]
names2 = ["Peter", "Joe", "Luke"]
print(names1)
print(names2)
```

```
['Jane']
['Peter', 'Joe', 'Luke']
```

Python - lists, creation methods

- using the no-argument constructor: list()

```
my_list = list()
print(my_list)
```

```
[]
```

- using the constructor with an iterable object as an argument: list(iterable)
- an iterable object can be a sequence, a container supporting iteration, or an iterator object
- the constructor creates a list whose elements are the same and in the same order as the elements of the iterable object

Python - lists, creation methods

- another list as a constructor argument: list(iterable)

```
names = ["Peter", "Joe", "Luke"]
names_list = list(names)
print(names_list)
```

```
['Peter', 'Joe', 'Luke']
```

- other iterable objects as constructor arguments: list(iterable)

```
letters = list("ABCDE")
numbers = list((1, 2, 3, 4, 5))
print(letters)
print(numbers)
```

```
['A', 'B', 'C', 'D', 'E']
[1, 2, 3, 4, 5]
```

Python - lists, creation methods

- using list comprehension
- list comprehension combines a for loop, the creation of a new element, and its automatic addition to the list in a single line of code

```
list_name = [expression for element in sequence]
```

- example: a list of squares of numbers from 1 to 10

```
squares = [x**2 for x in range(1,11)]
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- the for loop generates numbers x for the expression, the values of the expression are added to the squares list, note: there is no colon (:) after the for loop

Python - lists, creation methods

- using list comprehension
- it is also possible to add a condition to a list comprehension to filter elements

```
name = [expression for element in sequence if condition]
```

- example: a list of squares of even numbers from 1 to 10

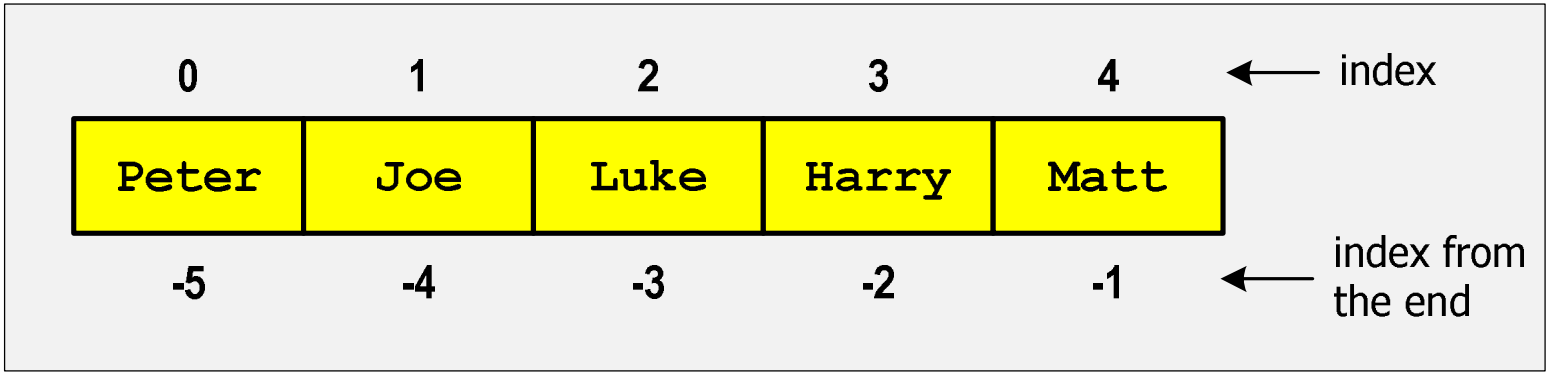
```
squares = [x**2 for x in range(1,11) if x % 2 == 0]
print(squares)
```

```
[4, 16, 36, 64, 100]
```

Python - lists, element indexes

- a list is an ordered collection, and any of its elements can be accessed by specifying its position (index)
- the first element of a list has an index of 0, the second element has an index of 1, and so on
- the last element of a list has an additional index of -1, the second-to-last element has an index of -2, and so on

```
names = ["Peter", "Joe", "Luke", "Harry", "Matt")]
```



Python - lists, element indexes

- to reference list elements, provide the list name followed by the element number in square brackets: list_name[index]

```
names = ["Peter", "Joe", "Luke", "Harry", "Matt"]
print(f"Hello {names[1]}!")
print(f"Hello {names[-1]}!")
```

```
Hello Joe!
Hello Matt!
```

- using an invalid index will cause a runtime error

```
print(f"Hello {names[5]}!")
```

```
Traceback (most recent call last):
  File "d:\MyApp.py", line 2, in <module>
    print(f"Hello {names[5]}!")
                   ~~~~~~^^^
IndexError: list index out of range
```

Python - lists, element indexes

- a colon (:) allows working with parts of a list, called slices

```
list_name[start_index : end_index : step]
```

- the slice starts from the element at start_index and ends at the element before end_index
- each of the values inside the square brackets can be omitted, and default values will be assumed
- start_index - default: 0 (first element)
- end_index - default: len(list) (total number of elements in the list)
- step - default: 1

Python - lists, element indexes

- examples of creating slices

```
names = ["Peter", "Joe", "Luke", "Harry", "Matt"]

print(names[0:3])
print(names[:4])
print(names[-3:])

print(names[::2])

print(names[:])
```

```
['Peter', 'Joe', 'Luke']
['Peter', 'Joe', 'Luke', 'Harry']
['Luke', 'Harry', 'Matt']

['Peter', 'Luke', 'Matt']

['Peter', 'Joe', 'Luke', 'Harry', 'Matt']
```

Python - lists, element indexes

- displaying a list using a for loop

```
names = ["Peter", "Joe", "Luke"]
for name in names:
    print(f"Hello {name}!")
```

```
Hello Peter!
Hello Joe!
Hello Luke!
```

- displaying a slice of a list using a for loop

```
names = ["Peter", "Joe", "Luke"]
for name in names[0:2]:
    print(f"Hello {name}!")
```

```
Hello Peter!
Hello Joe!
```

Python - lists, functions and methods

Function / Method	Result
`len(list_name)`	returns the number of elements in the list
`min(list_name)`	returns the smallest element in the list
`max(list_name)`	returns the largest element in the list
`sum(list_name)`	returns the sum of the elements in the list
`sum(list_name,start=0)`	returns the sum of the elements in the list, where start is an optional parameter specifying the initial value of the sum (default is 0)
`sorted(list_name)`	returns a sorted version of the list without modifying the original list

Python - lists, functions and methods

Function / Method	Result
`del list_name[i:j]`	removes elements with indices from i to j-1, equivalent to: list_name[i:j] = []
`del list_name[i:j:k]`	removes elements from i to j-1 with a step of k
`list_name.append(x)`	adds element x to the end of the list
`list_name.clear()`	removes all elements from the list, equivalent to: del list_name[:]
`list_name.copy(x)`	creates a copy of the list, equivalent to: list_name[:]
`list_name.extend(t)`	adds the contents of the iterable t to the end of the list t

Python - lists, functions and methods

Function / Method	Result
`list_name.insert(i,x)`	inserts element x at index i
`list_name.pop()`	removes and returns the last element of the list
`list_name.pop(i)`	removes and returns the element at index i
`list_name.remove(x)`	removes the first occurrence of element x in the list
`list_name.reverse()`	reverses the order of elements in the list
`list_name.sort()`	sorts the elements in the list

Python - lists, operations (in)

- checking if a value is in a list - the in keyword

```
import random

numbers = [random.randint(0,9) for _ in range(10)]
print(numbers)

num = int(input("Enter a number: "))

if num in numbers:
    print(f"The number {num} is in the list")
else:
    print(f"The number {num} is not in the list")
```

```
[1, 4, 1, 8, 5, 4, 0, 9, 0, 1]
Enter a number: 1
The number 1 is in the list
```

- an underscore after the keyword for is often used when the loop variable is not needed inside the loop

Python - lists, operations (not in)

- checking if a value is not in a list - the not in keyword

```
import random

numbers = [random.randint(0,9) for _ in range(10)]
print(numbers)

num = int(input("Enter a number: "))

if num not in numbers:
    print(f"The number {num} is in the list")
else:
    print(f"The number {num} is not in the list")
```

```
[2, 5, 8, 2, 0, 3, 9, 1, 7, 4]
Enter a number: 6
The number 6 is not in the list
```

Python - lists, operations (is the list empty?)

- checking if a list is not empty

```
passive = ["resistor", "coil", "capacitor"]

if passive:
    print("The 'passive' list is not empty")
else:
    print("The 'passive' list is empty")
```

```
The 'passive' list is not empty
```

- if a list's name appears in an if statement, it returns True if the list contains at least one element
- for an empty list, it returns False

Python - lists, operations (copy of the list)

- assigning a new name to an existing list does not create a copy

```
passive = ["resistor", "coil"]

new_passive = passive
passive.append("capacitor")

print(f"passive =     {passive}")
print(f"new_passive = {new_passive}")
```

```
passive =     ['resistor', 'coil', 'capacitor']
new_passive = ['resistor', 'coil', 'capacitor']
```

- both variables refer to the same object in memory (the list)

Python - lists, operations (copy of the list)

- creating a copy of a list requires using a slice that covers the entire original list

```
passive = ["resistor", "coil"]

new_passive = passive[:]
passive.append("capacitor")

print(f"passive =     {passive}")
print(f"new_passive = {new_passive}")
```

```
passive =     ['resistor', 'coil', 'capacitor']
new_passive = ['resistor', 'coil']
```

- this gives you two separate objects in memory (two lists)

Python - lists, operations (moving elements)

- to transfer elements from one list to another, it is better to use a while loop than a for loop (inside a for loop, you should not modify the list that is being iterated over)

```
passive = ["resistor", "coil", "capacitor"]
new_passive = []

while passive:
    element = passive.pop()
    print(f"Moving element: {element}")
    new_passive.append(element)

print(new_passive)
```

```
Moving element: capacitor
Moving element: coil
Moving element: resistor
['capacitor', 'coil', 'resistor']
```

Python - lists, operations (removing elements)

- removing from a list values that occur multiple times

```
names = ["Peter", "Joe", "Luke", "Peter", "Matt", "Joe"]
print(names)

element = input("Enter the element to remove: ")
while element in names:
    names.remove(element)

print(names)
```

```
['Peter', 'Joe', 'Luke', 'Peter', 'Matt', 'Joe']
Enter the element to remove: Joe
['Peter', 'Luke', 'Peter', 'Matt']
```

Python - tuplets, creation methods

- using a pair of regular parentheses to denote an empty tuple

```
my_tuple = ()
print(my_tuple)
```

```
()
```

- using regular parentheses and separating elements with commas

```
names = ("Peter", "Joe", "Luke")
print(imiona)
```

```
('Peter', 'Joe', 'Luke')
```

Python - tuplets, creation methods

- defining a single-element tuple requires adding a comma at the end

```
name1 = ("Mary",)
name2 = "Sarah",
print(name1)
print(name2)
```

```
('Mary',)
('Sarah',)
```

- omitting the comma results in a regular variable instead of a tuple

```
name = ("Mary")
print(name)
```

```
Mary
```

Python - tuplets, creation methods

- using the no-argument constructor: tuple()

```
my_tuple = tuple()
print(my_tuple)
```

```
()
```

- using the constructor with an iterable object as an argument: tuple(iterable)
- an iterable object can be a sequence, a container supporting iteration, or an iterator object
- the constructor creates a tuple whose elements are the same and in the same order as the elements of the iterable object

Python - tuplets, creation methods

- another tuple (or list) as a constructor argument: tuple(iterable)

```
names = ("Peter", "Joe", "Luke"))
names_tuple = tuple(names)
print(names_tuple)
```

```
('Peter', 'Joe', 'Luke')
```

- other iterable objects as constructor arguments: tuple(iterable)

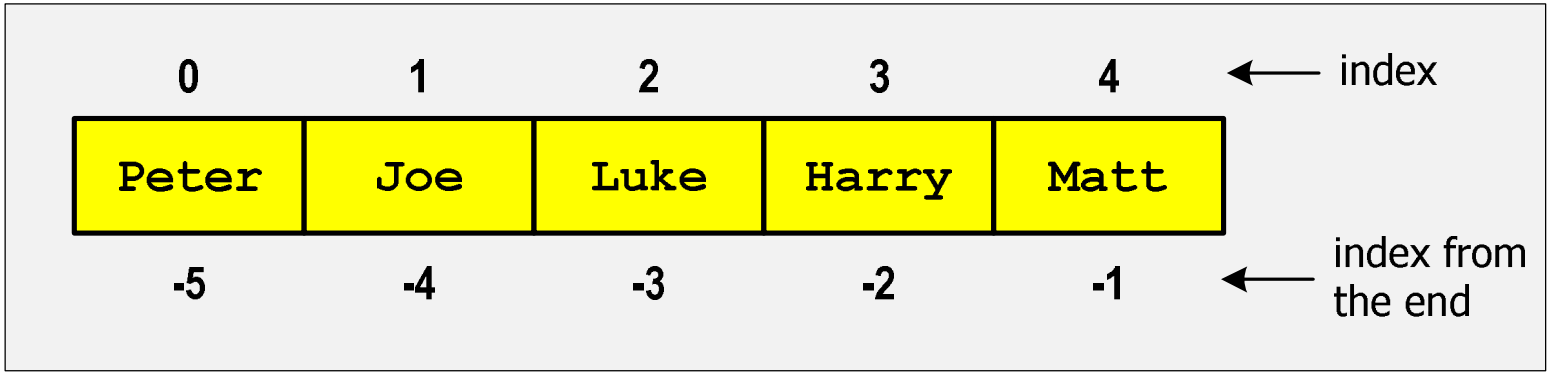
```
letters = tuple("ABCDE")
numbers = tuple((1, 2, 3, 4, 5))
print(letters)
print(numbers)
```

```
('A', 'B', 'C', 'D', 'E')
(1, 2, 3, 4, 5)
```

Python - tuplets, element indexes

- a tuple, like a list, is an ordered collection, and any of its elements can be accessed by specifying its index

```
names = ("Peter", "Joe", "Luke", "Harry", "Matt")
```



- to reference tuple elements, provide the tuple's name followed by the element number in square brackets: tuple_name[index]
- using a colon (:) allows working with slices of the tuple

Python - tuplets, element indexes

- examples of referencing tuple elements

```
names = ("Peter", "Joe", "Luke", "Harry", "Matt")

print(names[0])
print(names[1])
print(names[-1])

print(names[2:])
print(names[::2])
```

```
Peter
Joe
Matt

('Luke', 'Harry', 'Matt')
('Peter', 'Luke', 'Matt')
```

Python - tuplets

- attempting to modify a tuple element will result in an error

```
names = ("Peter", "Joe", "Luke", "Harry", "Matt")
names[0] = "Paul"
```

```
Traceback (most recent call last):
  File "d:\MyApp.py", line 2, in <module>
    imiona[0] = "Paul"
    ~~~~~~^^^
TypeError: 'tuple' object does not support item assignment
```

- a tuple can be overwritten with new values

```
names = ("Peter", "Joe", "Luke", "Harry", "Matt")
print(names)

names = ("Linda", "Mary", "Sarah", "Janet")
print(names)
```

End of lecture no. 4

Thank you for your attention!