Python Programming 1

(CP1S02005E)

Białystok University of Technology
Faculty of Electrical Engineering

Industry Digitization, semester II
Academic year 2024/2025

Lecture no. 07 (16.04.2025)

Jarosław Forenc, PhD

Topics

- Functions in Python
  - general structure of a function
  - passing and returning values
  - functions and variables, recursive functions
  - recommendations, docstrings
  - modules

Python - functions (function structure)

- general structure of a function in Python:

```
def function_name(parameters):
    """Function description (optional)"""
    # Function body
    result = expression
    return result
```

- a function definition starts with the keyword def, followed by the function name and its parameters (in parentheses)
- the function definition ends with a colon (:)
- a function description (called a docstring) is a special multi-line string placed after the function definition, used to document the code
- the body of the function is a block of code that defines the operations performed by the function
- the return statement is used to specify the value returned by the function and to terminate the function

Python - functions

- when calling a function, you provide the function name followed by its arguments (values) in parentheses

```
result = function_name(arguments)
```

- a function that calculates and returns the sum of two numbers

```
def add(a, b):
    sum = a + b
    return sum

result = add(5, 10)
print(f"The sum of the numbers is: {result}")
```

```
The sum of the numbers is: 15
```

- the body of the function consists of all the indented lines of code

Python - functions

- when calling a function, you provide the function name followed by its arguments (values) in parentheses

```
result = function_name(arguments)
```

- a shorter version of a function that calculates and returns the sum of two numbers

```
def add(a, b):
    return a + b

print(f"The sum of the numbers is: {add(5, 10)}")
```

```
The sum of the numbers is: 15
```

Python - functions

- a function can be called only after it has been declared
- it is not possible to call a function that has not yet been declared

```
result = add(5, 10)
print(f"The sum of the numbers is: {result}")

def add(a, b):
    sum = a + b
    return sum
```

```
Traceback (most recent call last):
  File "d:\MyApp.py", line 1, in <module>
    result = add(5, 10)
             ^^^
NameError: name 'add' is not defined
```

Python - functions

- calling a function without arguments will cause an error

```
def add(a, b):
    sum = a + b
    return sum

result = add()
print(f"The sum of the numbers is: {result}")
```

```
Traceback (most recent call last):
  File "d:\MyApp.py", line 5, in <module>
    result = add()
             ^^^
TypeError: add() missing 2 required positional
arguments: 'a' and 'b'
```

Python - functions

- calling a function with the wrong number of arguments will also result in an error

```
def add(a, b):
    sum = a + b
    return sum

result = add(5)
print(f"The sum of the numbers is: {result}")
```

```
Traceback (most recent call last):
  File "d:\MyApp.py", line 5, in <module>
    result = add(5)
             ^^^
TypeError: add() missing 1 required positional
argument: 'b'
```

Python - functions

- a function may have no arguments
- such a function usually displays information on the screen

```
def welcome():
    print("Hello world!")

welcome()
```

```
Hello world!
```

- the function does not have to include a return statement (but it can)

```
def welcome():
    print("Hello world!")
    return

welcome()
```

Python - functions

- a function with arguments can also just display information on the screen

```
def welcome(name):
    print(f"Hello, {name}!")
    return

welcome("Kate")
welcome("Paul")
```

```
Hello, Kate!
Hello, Paul!
```

Python - functions (passing arguments)

- positional arguments - the arguments in the function call appear in the same order as the parameters in the function definition

```
def calculate_bmi(weight, height):
    bmi = weight / (height ** 2)
    return bmi

BMI = calculate_bmi(75, 1.80)
print(f"BMI is: {BMI:.2f}")
```

```
BMI is: 23.15
```

Python - functions (passing arguments)

- keyword arguments - arguments are passed to the function as name=value pairs
- the order of arguments in the function call does not matter

```
def calculate_bmi(weight, height):
    bmi = weight / (height ** 2)
    return bmi

BMI = calculate_bmi(weight=75, height=1.80)
print(f"BMI is: {BMI:.2f}")

BMI = calculate_bmi(height=1.80, weight=75)
print(f"BMI is: {BMI:.2f}")
```

```
BMI is: 23.15
BMI is: 23.15
```

Python - functions (passing arguments)

- keyword arguments - arguments are passed to the function as name=value pairs
- if the parameter names don't match, an error will occur

```
def calculate_bmi(weight, height):
    bmi = weight / (height ** 2)
    return bmi

BMI = calculate_bmi(Weight=75, height=1.80)
print(f"BMI is: {BMI:.2f}")
```

```
Traceback (most recent call last):
  File "d:\MyApp.py", line 4, in <module>
    BMI = calculate_bmi(Weight=75, height=1.80)
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: calculate_bmi() got an unexpected keyword
argument 'Weight'
```

Python - functions (default parameters)

- each parameter of a function can have a default value

```
def post_code(code="15-351", city="Bialystok"):
    result = f"{code} {city}"
    return result

print(f"Address 1: {post_code("00-950","Warsaw")}")
print(f"Address 2: {post_code("15-433")}")
print(f"Address 3: {post_code()}")
```

```
Address 1: 00-950 Warsaw
Address 2: 15-433 Bialystok
Address 3: 15-351 Bialystok
```

- if no argument is provided for that parameter when the function is called, the default value will be used

Python - functions (default parameters)

- default parameters should be placed after parameters without default values

```
def post_code(code, city="Bialystok"):
    result = f"{code} {city}"
    return result

print(f"Address 1: {post_code("00-950","Warsaw")}")
print(f"Address 2: {post_code("15-433")}")
```

```
Address 1: 00-950 Warsaw
Address 2: 15-433 Bialystok
```

Python - functions (optional arguments)

- a function can use optional arguments

```
def person(name1, lastname, name2=""):
    if name2:
        result = f"{name1} {name2} {lastname}"
    else:
        result = f"{name1} {lastname}"
    return result

pers = person("John", "Smith")
print(pers)
pers = person("Arthur", "Doyle", "Conan")
print(pers)
```

```
John Smith
Arthur Conan Doyle
```

Python - functions (returning a value)

- the return statement can appear multiple times

```
def grade(pts):
    if  0 <= pts <= 50:
        return 2.0
    elif 51 <= pts <= 60:
        return 3.0
    elif 61 <= pts <= 70:
        return 3.5
    elif 71 <= pts <= 80:
        return 4.0
    elif 81 <= pts <= 90:
        return 4.5
    elif 91 <= pts <= 100:
        return 5.0

pts = int(input("Enter points: "))
print(f"Your grade: {grade(pts)}")
```

```
Enter points: 67
Your grade: 3.5
```

```
Enter points: 100
Your grade: 5.0
```

```
Enter points: -10
Your grade: None
```

Python - functions (returning a value)

- in the simplest case, a function can return a single value

```
def get_text(make, year):
    txt = f"Your {make} is {2025-year} years old"
    return txt

car = get_text("Opel",1988)
print(car)
```

```
Your Opel is 37 years old
```

- the return statement takes a value from inside the function and passes it back to the function call location

Python - functions (returning a value)

- a function can return a value of any type, such as a dictionary or a list

```
def person(name, lastname):
    pers = {"name":name, "lastname":lastname}
    return pers

someone = person("John", "Smith")
print(someone)
```

```
{'name': 'John', 'lastname': 'Smith'}
```

Python - functions (returning a value)

- a function can return a value of any type, such as a dictionary or a list

```
def person(name, lastname, age=None):
    pers = {"name":name, "lastname":lastname}
    if age:
        pers["age"] = age
    return pers

someone = person("John", "Smith", age=23)
print(someone)

someone = person("John", "Smith")
print(someone)
```

```
{'name': 'John', 'lastname': 'Smith', 'age': 23}
{'name': 'John', 'lastname': 'Smith'}
```

- the value None will be used when a variable is not assigned any specific value

Python - functions (returning a value)

- a function can return multiple values

```
def conversion(seconds):
    hour = seconds // 3600
    minute = (seconds % 3600) // 60
    second = seconds % 60
    return hours, minute, second

h, m, s = conversion(8523)
print(f"{h:02d}:{m:02d}:{s:02d}")
result = conversion(8523)
print(result)
```

```
02:22:03
(2, 22, 3)
```

- in reality, the function returns a single tuple, which we unpack into variables

Python - functions (value passing)

- you can pass values of complex types, such as lists, to a function

```
def welcome(people):
    for person in people:
        print(f"Hello, {person}!")

people = {"John","Kate","Peter","Monica"}
welcome(people)
```

```
Hello, John!
Hello, Kate!
Hello, Peter!
Hello, Monica!
```

Python - functions (value passing)

- a list passed to a function can be modified
- all changes made are permanent

```
def mr_mrs(people):
    for i, person in enumerate(people):
        people[i] = f"Mrs {person}" if person[-1] == "a"
                                    else f"Mr {person}"

people = ["Jan","Kasia","Piotr","Magda"]
print(people)
mr_mrs(people)
print(people)
```

```
['Jan', 'Kasia', 'Piotr', 'Magda']
['Mr Jan', 'Mrs Kasia', 'Mr Piotr', 'Mrs Magda']
```

Python - functions (value passing)

- a copy of the list can be passed to the function (using slice notation [:])

```
def mr_mrs(people):
    for i, person in enumerate(people):
        people[i] = f"Mrs {person}" if person[-1] == "a"
                                    else f"Mr {person}"
people = ["Jan","Kasia","Piotr","Magda"]

print(people)
mr_mrs(persons[:])
print(people)
```

```
['Jan', 'Kasia', 'Piotr', 'Magda']
['Jan', 'Kasia', 'Piotr', 'Magda']
```

- this should only be done in justified cases, as creating a copy takes time and memory

Python - functions (value passing)

- any number of arguments can be passed to a function

```
def welcome(*people):
    for person in people:
        print(f"Hello, {person}!")

welcome("Kate")
welcome("Mike", "Peter")
```

```
Hello, Kate!
Hello, Mike!
Hello, Peter!
```

- an asterisk in the parameter name causes the creation of a tuple named people and places the received values in it
- a tuple is created even when only one argument is given
- an arbitrary number of arguments is often referred to as *args

Python - functions (value passing)

- if a function has parameters of different types, the parameter that accepts multiple values must be placed at the end
- Python first matches positional arguments and keyword arguments, and only then gathers the remaining arguments

```
def welcome(text, *people):
    for person in people:
        print(f"{text}, {person}!")

welcome("Bonjour", "Kate")
welcome("Hello", "Mike", "Peter")
```

```
Bonjour, Kate!
Hello, Mike!
Hello, Peter!
```

Python - functions (value passing)

- any number of keyword arguments can also be passed

```
def phone(brand, model, **data):
    data["brand"] = brand
    data["model"] = model
    return data

Samsung = phone("Samsung", "Galaxy", screen="AMOLED")
print(Samsung)
```

```
{'screen': 'AMOLED', 'brand': 'Samsung', 'model': 'Galaxy'}
```

- two asterisks before the parameter name cause the creation of an empty dictionary named data and place all received key-value pairs into it

Python - functions and variables

- variables declared inside a function are local, meaning they only exist within that function

```
def my_function():
    val = 10
    print(val)

my_function()
print(val)
```

```
10
Traceback (most recent call last):
  File "d:\MyApp.py", line 6, in <module>
    print(val)
          ^^^
NameError: name 'val' is not defined
```

- the variable val cannot be used outside the function my_function()

Python - functions and variables

- variables declared outside functions are global variables
- they can be used throughout the entire file

```
val = 10

def my_function():
    print(val)

my_function()
```

```
10
```

- inside the function my_function(), the variable val can only be read

Python - functions and variables

- variables declared outside functions are global variables
- assigning a new value to the variable val inside the function my_function() will create a local variable

```
val = 10                # global variable

def my_function():
    val = 0             # local variable
    print(val)          # print local variable

my_function()
print(val)              # print global variable
```

```
0
10
```

Python - functions and variables

- variables declared outside functions are global variables
- assigning a new value to the variable val inside the my_function() function requires the use of the global keyword

```
val = 10                # global variable

def my_function():
    global val          # global – global variable
    val = 0
    print(val)          # print global variable

my_function()
print(val)              # print global variable
```

```
0
0
```

Python - recursive functions

- in Python, recursive functions can be used - that is, functions that call themselves

```
# factorial

def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

for n in range(0,16):
    print(f"{n}! = {factorial(n)}")
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
```

Python - recursive functions

- in Python, recursive functions can be used - that is, functions that call themselves

```
# Fibonacci sequence

def Fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return Fib(n - 1) + Fib(n - 2)

for n in range(0,16):
    print(f"F{n} = {Fib(n)}")
```

```
F0 = 0
F1 = 1
F2 = 1
F3 = 2
F4 = 3
F5 = 5
F6 = 8
F7 = 13
F8 = 21
F9 = 34
F10 = 55
F11 = 89
F12 = 144
F13 = 233
F14 = 377
F15 = 610
```

- Python has a recursion depth limit, which by default is around 1000 levels

Python - functions (recommendations)

- the name of a function should clearly indicate its purpose
- a function name should consist of lowercase letters and underscores
- the function code should include a comment explaining the purpose of the function, placed after the function definition in the form of a docstring
- if a parameter has a default value, no spaces should be placed on either side of the equals sign
- if a parameter is written in key-value form, no spaces should be placed on either side of the equals sign

Python - functions (docstrings)

- a docstring is a string used to document code
- a docstring is placed after the function definition, enclosed in triple quotes or triple apostrophes

```
def add(a, b):
    """
    A function that adds two numbers

    Parameters
    a (float): the first number
    b (float): the second number

    Returns:
    float: sum of two numbers
    """
    sum = a + b
    return sum
```

a docstring contains:

- a description of the function
- a description of the parameters (name, type, description)
- a description of the return value
- information about exceptions (if any are handled),
- an example of how to use the function

Python - functions (docstringi, PEP 257 style)

- PEP 257-style docstring (classic Python style)

```
def compute_bmi(weight, height):
    """Calculates the body mass index (BMI) value.

    Weight should be in kilograms and height in meters.

    :param weight: Body weight in kilograms.
    :type weight: float
    :param height: Height in meters.
    :type height: float
    :return: Calculated BMI value.
    :rtype: float
    """
    bmi = weight / (height ** 2)
    return bmi
```

- https://peps.python.org/pep-0257/

Python - functions (docstrings, Google style)

- Google-style docstring

```
def calculate_bmi(weight, height):
    """
    Calculates the body mass index (BMI) value.

    Args:
        weight (float): Body weight in kilograms.
        height (float): Height in meters.

    Returns:
        float: Calculated BMI value.
    """
    bmi = weight / (height ** 2)
    return bmi
```

- https://google.github.io/styleguide/pyguide.html

Python - functions (modules)

- functions that have been written can be placed in a separate file (module) and then imported into the program where they are to be used
- a module is a file with the .py extension that contains code to be imported

perimeter.py

```
def square(a):
    result = 4 * a
    return result

def rectangle(a,b):
    result = 2 * (a + b)
    return result
```

myapp.py

```
import perimeter

a = float(input("Side of square: "))
perim = perimeter.square(a)
print(f"Perimeter is: {perim}")
```

```
Side of square: 3
Perimeter is: 12.0
```

Python - functions (modules)

- the import perimeter statement imports the entire module, meaning all functions from the perimeter.py file are copied into the current program

myapp.py

```
import perimeter

a = float(input("Side of square: "))
perim = perimeter.square(a)
print(f"Perimeter is: {perim}")
```

- when calling a function, you provide the module name, a dot, the function name, and its arguments in parentheses

```
module_name.function_name(arguments)
```

Python - functions (modules)

- it is possible to import only selected functions from a module

```
from module_name import function_name
```

or

```
from module_name import name1, name2, name3, ...
```

- in this case, when calling the function, you don't need to use the dot notation

myapp.py

```
from perimeter import square

a = float(input("Side of square: "))
perim = square(a)
print(f"Perimeter is: {perim}")
```

Python - functions (modules)

- when importing a function from a module, you can assign it a new name (alias)

```
from module_name import old_name as new_name
```

Example:

myapp.py

```
from perimeter import square as sqr

a = float(input("Side of square: "))
perim = sqr(a)
print(f"Perimeter is: {perim}")
```

- this is useful when the imported function name conflicts with an existing function in your program, or when the original name is too long

Python - functions (modules)

- the as keyword can also be used to define an alias for the module itself

```
import module_name_old as modul_name_new
```

Example:

myapp.py

```
import perimeter as p

a = float(input("Side of square: "))
perim = p.square(a)
print(f"Perimeter is: {perim}")
```

Python - functions (modules)

- using the * operator, you can import all functions from a module

```
from module_name import *
```

- this allows you to call functions without the module name and dot

myapp.py

```
from perimeter import *

a = float(input("Side of square: "))
perim = square(a)
print(f"Perimeter is: {perim}")
```

End of lecture no. 7

Thank you for your attention!