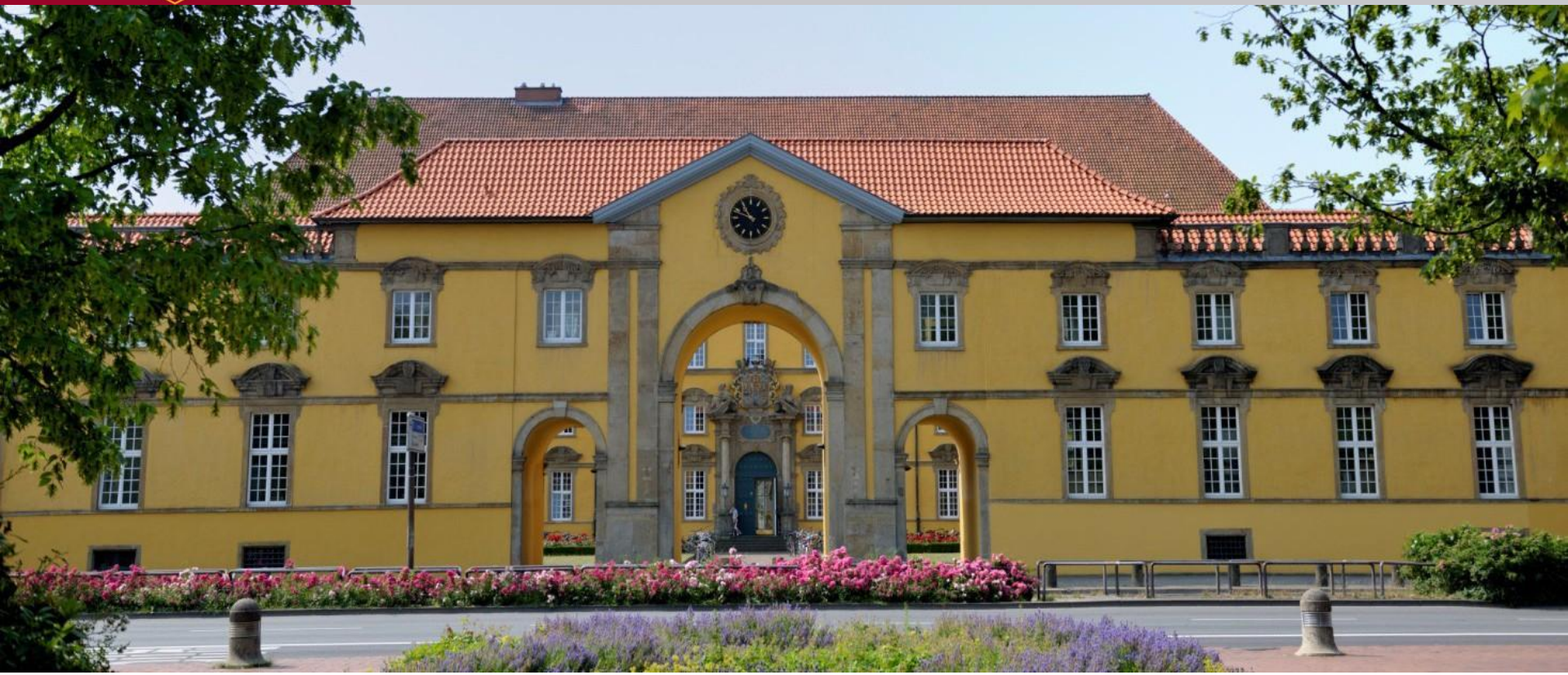

Basic Programming in Python

4. Chapter: Functions

Lecturer: Daniel Weinhardt

Summer Term 2024

Slides were created by Nohayr Muhammad Abdelmoneim
Thank you very much for sharing!

Overview

- Recap on methods
- Functions vs. Methods
- How to define Functions
- How to call Functions
- Variable Scope
- Recursive Functions

String Methods

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case

https://www.w3schools.com/python/python_strings_methods.asp

List Methods

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

https://www.w3schools.com/python/python_lists_methods.asp

Built in functions

Function	Description
abs()	Returns the absolute value of a number
all()	Returns True if all items in an iterable object are true
any()	Returns True if any item in an iterable object is true
ascii()	Returns a readable version of an object. Replaces none-ascii characters with escape character
bin()	Returns the binary version of a number
bool()	Returns the boolean value of the specified object
bytearray()	Returns an array of bytes
bytes()	Returns a bytes object
callable()	Returns True if the specified object is callable, otherwise False
chr()	Returns a character from the specified Unicode code.
classmethod()	Converts a method into a class method
compile()	Returns the specified source as an object, ready to be executed
complex()	Returns a complex number
delattr()	Deletes the specified attribute (property or method) from the specified object
dict()	Returns a dictionary (Array)
dir()	Returns a list of the specified object's properties and methods

https://www.w3schools.com/python/python_ref_functions.asp

What is a function?

- block of code with a certain name
- performs a specific task
- can take one or more arguments as an input.
- can return one or more variables as an output.
- is defined once and can be used many times.
- is only executed when it is “called”

What is a function?

- block of code with a certain name
- performs a specific task
- can take one or more arguments as an input.
- can return one or more variables as an output.
- is defined once and can be used many times.
- is only executed when it is “called”

Let’s break that down!

Defining and calling a function

- block of code with a certain name
- the **syntax** for defining a function looks like this:

  def function_name():
- performs a specific task
- Python knows which block of code belongs to the function through the **correct indentation**

  → **one tab more** than the function definition
- Call function with name and "()"

```python
# Definition of a simple function
def greet():
    print("Hello")

# Call the defined function
greet()
```

```
Hello
```

Passing input arguments

- can take one or more **arguments as an input**

```python
# Definition of a simple function
def greet(name):
    # task which is performed by the function
    print("Hello", name)

# Call the defined function
greet("Daniel")
```

✓ 0.0s

```
Hello Daniel
```

One input

Multiple inputs

```python
# Definition of a simple function
def greet(name, greeting_formula):
    # task which is performed by the function
    print(greeting_formula + ", " + name + "!\nHow are you today?")

# Call the defined function
greet("Daniel", "What's up")
```

✓ 0.0s

```
What's up, Daniel!
How are you today?
```

Returning values

- can return one or more variables as an output.

Use “return” keyword to return one value or a tuple of values

Assign output values to a variable
(or use them directly)

```python
# Definition of a simple function
def greet(name):
    # task which is performed by the function
    greeting = "Hello " + name
    return greeting

# Call the defined function
output = greet("Daniel")
print(output)
```

✓ 0.0s

```
Hello Daniel
```

Returning values

- Functions can return one or more variables as an output.

Return multiple variables

Return a tuple of variables

Access the single output elements through indexing

```python
# Definition of a simple function
def greet(name):
    # task which is performed by the function
    greeting = "Hello " + name
    return greeting, len(greeting)

# Return the outputs by assigning the values to variables
greeting_out, greeting_len = greet("Daniel")
print(output)
print(f"The length of the greeting is: {greeting_len}")

# Return an tuple of the outputs
output_tuple = greet("Daniel")
print(output_tuple[0])
print(f"The length of the greeting is: {output_tuple[1]}")
```

✓ 0.0s

```
Hello Daniel
The length of the greeting is: 12
Hello Daniel
The length of the greeting is: 12
```

Reusability

- is defined once and can be used many times

Calling the same function with different input arguments

```python
# Definition of a simple function
def greet(name):
    # task which is performed by the function
    greeting = "Hello " + name
    return greeting, len(greeting)

list_of_names = ["Daniel", "Alexandra", "Rob", "Arthur", "Silvia"]
for name in list_of_names:
    print(greet(name))
```

✓ 0.0s

```
('Hello Daniel', 12)
('Hello Alexandra', 15)
('Hello Rob', 9)
('Hello Arthur', 12)
('Hello Silvia', 12)
```

Advanced function handling - Default arguments

- Default arguments can be defined in a function
- Useful if many cases can be covered with one value
- No need to specifically pass a value to a default argument

  → Makes function handling easier

```python
# Definition of a simple function
def greet(name, greeting_formula="Hello"):
    # task which is performed by the function
    print(greeting_formula + ", " + name + "!\nHow are you today?")

# Call the defined function
greet("Daniel")
```

✓ 0.0s

default argument

no extra input needed

```
Hello, Daniel!
How are you today?
```

Advanced function handling - Default arguments

- Default arguments can be defined in a function
- Useful if many cases can be covered with one value
- No need to specifically pass a value to a default argument

  → Makes function handling easier

```python
# Definition of a simple function
def greet(name, greeting_formula="Hello"):
    # task which is performed by the function
    print(greeting_formula + ", " + name + "!\nHow are you today?")

# Call the defined function
greet("Daniel", "What's up")
```

✓ 0.0s

```
What's up, Daniel!
How are you today?
```

overwriting default value

Advanced function handling - Type definitions

- Input and output types can be defined as well
- Less confusion about possible inputs and outputs
- No hard type-checking but easier readability of your code

Type definitions of inputs and outputs

```python
def my_math_function(a: float, b: float, operator: str = "+") -> str:
    if operator == "+":
        c = a + b
    elif operator == "-":
        c = a - b
    elif operator == "*":
        c = a * b
    elif operator == "/":
        c = a / b
    else:
        c = "Operator is not defined. Please choose one among these options: [+, - , *, /]"
    string_equation = str(a) + operator + str(b) + "=" + str(c)
    return string_equation

my_math_function(5, 3, "+")
```

✓ 0.0s

```
'5+3=8'
```

Advanced function handling - Assert statement

- With assert statements you can make sure that input arguments fulfill certain conditions
- Conditions must be Boolean

Control instance through assert statement

```python
# Definition of a simple function
def greet(name: str):
    # assert that the input is a string
    assert type(name) == str, "The input must be a string."

    # assert that the input is a string with only alphabetical characters
    assert name.isalpha(), "The input must be a string with only alphabetical characters"

    # task which is performed by the function
    greeting = "Hello " + name
    return greeting

str1 = "Daniel"
print(greet(str1))
```

✓ 0.0s

```
Hello Daniel
```

Advanced function handling - Assert statement

- With assert statements you can make sure that input arguments fulfill certain conditions

```
str1 = "Daniel123"
print(greet(str1))
```

0.1s

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[29], line 11
      8     return greeting
     10 str1 = "Daniel123"
---> 11 print(greet(str1))

Cell In[29], line 4
      2 def greet(name):
      3     # assert that the input is a string with only alphabetical characters
----> 4     assert name.isalpha(), "The input must be a string with only alphabetical characters"
      6     # task which is performed by the function
      7     greeting = "Hello " + name

AssertionError: The input must be a string with only alphabetical characters
```

Advanced function handling - Docstrings

- Docstrings serve as documentation for functions
- Provide necessary information to understand

  → the inner mechanisms, input arguments, return values

Descriptions of:

Function →

Inputs →

Return →

```python
def toy_function(a, b, c):
    """This function takes
    a string, a list and a float number and
    returns a lower case version of the string,
    the list repeated twice and the square of the float number.

    Args:
        a (str): a string containing only alphanumeric characters
        b (list): a list of strings
        c (float): a float number

    Returns:
        str: lower case version of a
        list: b repeated twice
        float: c squared
    """


    a_lower = a.lower()
    b_double = b * 2
    c_squared = c ** 2

    return a_lower, b_double, c_squared
```

Advanced function handling - Docstrings

- Docstrings serve as documentation for functions
- Provide necessary information to understand
- Can be looked up with the **help()** function

```
help(toy_function)
```

✓ 0.0s

```
Help on function toy_function in module __main__:

toy_function(a: str, b: list, c: float) -> tuple[str, list, float]
    This function takes
    a string, a list and a float number and
    returns a lower case version of the string,
    the list repeated twice and
    the square of the float number.

    Args:
        a (str): a string containing only alphanumeric characters
        b (list): a list of strings
        c (float): a float number

    Returns:
        str: lower case version of a
        list: b repeated twice
        float: c squared
```

Difference between functions and methods

- A function doesn't need any object and is independent
- A method is a function that is linked with an object
- We can directly call the function with its name
- A method is called by **object.method()**

Function

```
x=[1,2,3,4,5]
print(len(x))
```

```
5
```

```
x=[1,2,3,4,5]
print(x.len())
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (m
Cell In[9], line 2
      1 x=[1,2,3,4,5]
----> 2 print(x.len())

AttributeError: 'list' object has no attribute 'len'
```

Method

```
y="HeLLo"
print(lower(y))
```

```
---------------------------------------------------------------------------
NameError                                 T
Cell In[7], line 2
      1 y="HeLLo"
----> 2 print(lower(y))

NameError: name 'lower' is not defined
```

```
y="Hello"
print(y.lower())
```

```
hello
```

Recursive functions - recursive factorial

- $n! = n \times (n-1) \times (n-2) \times \cdots \times 1$
- $n! = n \times (n-1)!$

```python
def RecFact(n):
    if n==1:
        return 1
    return n*(RecFact(n-1))
```

```python
print(RecFact(5))
```

```
120
```

Tracing back $RecFact(5)$:

$$RecFact(5) = 5 \times RecFact(4) = 120$$

Is n==1? No

$$RecFact(4) = 4 \times RecFact(3) = 24$$

Is n==1? No

$$RecFact(3) = 3 \times RecFact(2) = 6$$

Is n==1? No

$$RecFact(2) = 2 \times RecFact(1) = 2$$

Is n==1? Yes

$$RecFact(1) = 1$$

Recursive functions

- A recursive function is a function that calls itself inside its body.
- A recursive function can be considered as a loop. Hence, a condition must be specified to stop execution.

```
def MyFunction():
    MyFunction()
```

Recursive functions - recursive factorial

- $n! = n \times (n-1) \times (n-2) \times \cdots \times 1$
- $n! = n \times (n-1)!$

```python
def RecFact(n):
    return n*(RecFact(n-1))
```

```python
RecFact(5)
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent c
Cell In[41], line 1
----> 1 RecFact(5)

Cell In[40], line 2, in RecFact(n)
      1 def RecFact(n):
----> 2     return n*(RecFact(n-1))

Cell In[40], line 2, in RecFact(n)
      1 def RecFact(n):
----> 2     return n*(RecFact(n-1))

    [... skipping similar frames: RecFact at line 2 (2970 times)]

Cell In[40], line 2, in RecFact(n)
      1 def RecFact(n):
----> 2     return n*(RecFact(n-1))

RecursionError: maximum recursion depth exceeded
```

Variable scope - local vs. global

- Variable scope determines which part of the program it's defined and can be used at.
- A variable which is created inside a function, is only defined within the function.

Local variable

```
def my_sum():
    x=5
    y=7
    print(x+y)
```

```
my_sum()
print(x,y)
```

```
12

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[8], line 2
      1 my_sum()
----> 2 print(x,y)

NameError: name 'x' is not defined
```

Global variable

```
x=5
y=7
def my_sum():
    print(x+y)
```

```
my_sum()
print(x,y)
```

```
12
5 7
```

global keyword

The global keyword is used to define a global variable that is defined throughout the program.

```
def my_sum():
    global x
    x=5
    y=7
    print(x+y)
```

```
my_sum()
print(x)
print(y)
```

```
12
5

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[4], line 3
      1 my_sum()
      2 print(x)
----> 3 print(y)

NameError: name 'y' is not defined
```

Recap: functions

- $n! = n \times (n-1) \times (n-2) \times \cdots \times 1$
- $n! = n \times (n-1)!$

```
def RecFact(n):
    if n==1:
        return 1
    return n*(RecFact(n-1))
```

```
print(RecFact(5))
```

```
120
```

Questions:

- What would happen if we replace return with print?
- Can you modify the function so that it only accepts positive numbers?
- Does the order within the function matter?
- What would happen if we switch the order of an if-statement and return?

QUESTIONS?