Python Programming 1

(CP1S02005E)

Białystok University of Technology
Faculty of Electrical Engineering

Industry Digitization, semester II
Academic year 2024/2025

Lecture no. 09 (30.04.2025)

Jarosław Forenc, PhD

Topics

- Files in Python
  - format JSON
  - pathlib module
- Exceptions
  - try-except
  - try-except-finally
  - try-except-else

Python - JSON format

- JSON (JavaScript Object Notation) is a format for storing and exchanging computer data, originally developed for the JavaScript language (files with the .json extension)
- it stores data in key-value pairs
- example of a JSON file:

```
{
    "name": "John",
    "age": 21,
    "hobby": ["sport", "dance", "travel"]
}
```

- JSON is language-independent - most programming languages have built-in support for this format (e.g., Python, Java, C++, JavaScript)
- it is very popular in communication between applications

Python - JSON format

- data types used by JSON:
  - integer and floating-point numbers (float, compliant with double-precision floating-point format)

```
{
    "age" : 25,
    "height" : 1.85
}
```

  - string - as a key, diacritical characters should be avoided

```
{
    "name" : "Paul",
    "surname" : "Walker"
}
```

Python - JSON format

- data types used by JSON:
  - boolean values - true, false

```
{
    "ready" : true,
    "finished" : false
}
```

  - null - a special value indicating the absence of data

```
{
    "data" : null
}
```

  - array

```
{
    "cities" : ["Paris", "Berlin", "Tokio"]
}
```

Python - JSON format

- data types used by JSON:
  - object - sets of key-value pairs enclosed in curly braces

```
{
    "car" :
    {
        "brand" : "Opel",
        "model" : "Kadet",
        "year" : 1998,
        "registered" : true
    }
}
```

- advantages of the JSON format include high data readability
- other popular formats: CSV, XML, YAML

Python - JSON format

- in Python, the json module is used to handle JSON data
- json.dumps() is used to encode data to JSON format

```
import json
data = {"name": "John", "age": 30, "city": "Berlin"}
json_string = json.dumps(data)
print(json_string)
```

```
{"name": "John", "age": 30, "city": "Berlin"}
```

```
import json
numbers = [2, 3, 5, 7, 11, 13]
json_string = json.dumps(numbers)
print(json_string)
```

```
[2, 3, 5, 7, 11, 13]
```

Python - JSON format

- json.loads() is used to decode JSON data into Python objects

```
import json
json_string = '{"name": "John", "age": 30, "city": "Berlin"}'
data = json.loads(json_string)
print(data)
```

```
{'name': 'John', 'age': 30, 'city': 'Berlin'}
```

Python - JSON format

- these functions (json.dumps() and json.loads()) also enable direct reading from and writing to files

```
import json

data = [1, 2, 3, 4, 5]

# Writing data to a JSON file
with open("data.json", "w") as json_file:
    json.dump(data, json_file)

# Reading data from a JSON file
with open("data.json", "r") as json_file:
    loaded_data = json.load(json_file)

print(loaded_data)
```

```
[1, 2, 3, 4, 5]
```

Python - pathlib module

- the pathlib module is used for handling file and directory paths (independent of the operating system)
- paths are represented as Path objects, which can be manipulated using various methods
- these methods allow reading data from a file and writing data to a file
- reading and displaying the contents of a text file on the screen:

```
from pathlib import Path
file = Path("data.txt")
data = file.read_text()
print(data)
```

- we create a Path object and assign it to a variable named file
- the read_text() method reads the entire content of the file and stores it in a single, long text string called data, and then automatically closes the file

Python - pathlib module

- when the end of the file is reached, the read_text() method returns an empty string, which is displayed as a blank line
- the blank line can be removed using the rstrip() method:

```
from pathlib import Path
file = Path("data.txt")
data = file.read_text()
data = data.rstrip()
print(data)
```

- you can also use so-called method chaining:

```
from pathlib import Path
file = Path("data.txt")
data = file.read_text().rstrip()
print(data)
```

Python - pathlib module

- as the file path, you can provide:
  - a relative path - the file's location is specified relative to the directory in which the program is being executed

```
file = Path("folder/subfolder/file.txt")
```

  - an absolute path - contains the full path to the file, starting from the drive name (Windows) or the root of the file system (Linux)

```
file1 = Path("C:/folder/subfolder/file.txt")
file2 = Path("/home/user/folder/file.txt")
```

- when specifying file paths, forward slashes (/) are used to separate individual elements (e.g., directories)

Python - pathlib module

- the text read using read_text() can be split into lines and then processed using a for loop

```
from pathlib import Path
file = Path("data.txt")
data = file.read_text()
rows = data.splitlines()
for row in rows:
    print(row)
```

- the splitlines() method splits a string into lines, using the newline character (\n) by default
- the splitting into lines can be done directly within a for loop

```
...
data = file.read_text()
for row in data.splitlines():
    print(row)
```

Python - pathlib module

- if we want to work with the file data as numbers, we need to convert them from text to integers (using the int() function) or floating-point numbers (using the float() function)
- example: sum of floating-point numbers stored in a file

```
from pathlib import Path
file = Path("numbers.txt")
rows = file.read_text().splitlines()
sum = 0
for row in rows:
    sum = sum + float(row)
print(f"Sum of numbers in file: {sum}")
```

```
12.34
15.67
21.36
45.12
```

```
Sum of numbers in file: 94.49
```

- Python does not impose a limit on the amount of data it can work with - the limitation is determined by the system's memory

Python - pathlib module

- the write_text() method allows writing a single line of text to a file

```
from pathlib import Path
file = Path("output.txt")
file.write_text("Hello world!\n")
```

```
Hello world!
```

- if a file with the given name does not exist, it will be created
- if a file with the given name already exists, its previous content will be deleted
- the write_text() method ensures the file is properly closed after the write operation is completed

Python - pathlib module

- if the text consists of multiple lines, it should be prepared in advance, and the write_text() method should be called only once

```
from pathlib import Path
text =  "---------------------------\n"
text += "|  Name  | Code  |  Rate  |\n"
text += "---------------------------\n"
text += "| euro   | 1 EUR | 4.2789 |\n"
text += "| dollar | 1 USD | 3.7599 |\n"
text += "---------------------------\n"
file = Path("table.txt")
file.write_text(text)
```

```
---------------------------
|  Nama  | Code  |  Rate  |
---------------------------
| euro   | 1 EUR | 4.2789 |
| dollar | 1 USD | 3.7599 |
---------------------------
```

Python - pathlib module (methods)

Method	Description
`cwd()`	returns a Path object representing the current directory
`home()`	returns a Path object representing the user's home directory
`exists()`	returns True if the given file or directory path physically exists on the disk
`is_dir()`	returns True if the given path represents a directory
`is_file()`	returns True if the given path represents a file
`iterdir()`	iterates through all elements (files, directories, etc.) in a given directory, returning a generator containing Path objects representing those elements

Python - pathlib module (methods)

Method	Description
`mkdir()`	creates a new directory on the disk
`read_bytes()`	reads the contents of a file as binary data
`rename()`	renames a file or directory
`replace()`	replaces a file or directory on the disk; similar to the rename() method, but if the target path already exists, it will be replaced by the current file or directory
`rmdir()`	removes an empty directory from the file system
`write_bytes()`	writes data to a file as binary data

Python - pathlib module

- checking if a file exists, displaying file contents, displaying the current and home directory

```
from pathlib import Path
file = Path("data.txt")
if file.exists():
    if file.is_file():
        print(f"{file} exists, content: ")
        data = file.read_text()
        print(data)
    else:
        print(f"{file} - not a file")
else:
    print(f"No file: {file}")

print(f"Current directory: {Path.cwd()}")
print(f"Home directory: {Path.home()}")
```

Python - exceptions

- exceptions are special objects used by Python to manage errors that may occur during program execution.
- if an exception is raised and not handled, the program is interrupted and a traceback is displayed showing the exception that occurred

```
x = float(input("Enter x: "))
y = float(input("Enter y: "))
z = x / y
print(f"The result is: {z}")
```

```
Enter x: 3
Enter y: 0
Traceback (most recent call last):
  File "d:\MyApp.py", line 3, in <module>
    z = x / y
        ~~^~~
ZeroDivisionError: float division by zero
```

Python - exceptions (try-except)

- to catch and handle exceptions, a try-except block is used

```
try:
    # code that may raise an exception
except ExceptionType:
    # exception handling
```

- after try: we place code that may raise an exception
- after except: we place code that should run if an exception occurs
- you can handle a specific type of error, e.g.
  - except ZeroDivisionError:
  - except ValueError:
- you can also use a general exception if you're not sure what type might occur:
  - except Exception:

Python - exceptions (try-except-finally)

```
try:
    # code that may raise an exception
except ExceptionType:
    # exception handling
finally:
    # code that will always be executed
```

- the try-except-finally statement is used when you want to ensure that certain instructions are executed regardless of whether an exception occurs or not
- the finally block is optional, but if it is present, it will always be executed, regardless of whether an exception occurred
- the finally block is used to clean up resources, such as files or network connections, that should always be released regardless of exceptions

Python - exceptions (try-except-else)

```
try:
    # code that may raise an exception
except ExceptionType:
    # exception handling
else:
    # code that will be executed only
    # if no exception occurred
```

- the try-except-else statement is used when we want to execute certain instructions only if no exception occurred in the try block
- the else block is optional and will be executed only if no exception was raised in the try block
- the else block is useful when we want to perform some operations - for example, calculations on data that are expected to work correctly

Python - exceptions

- protecting the program against division by zero

```
x = float(input("Enter x: "))
y = float(input("Enter y: "))
try:
    z = x / y
except ZeroDivisionError:
    print("Division by zero error!")
else:
    print(f"The result is: {z}")
```

```
Enter x: 3
Enter y: 0
Division by zero error!
```

```
Enter x: 3
Enter y: 7
The result is: 0.42857142857142855
```

Python - exceptions

- protecting the program against division by zero and invalid user input

```
x = float(input("Enter x: "))
y = float(input("Enter y: "))
try:
    z = x / y
except ZeroDivisionError:
    print("Division by zero error!")
except ValueError:
    print("Invalid number format!")
else:
    print(f"The result is: {z}")
```

```
Enter x: 3
Enter y: 7,0
Invalid number format!
```

Python - exceptions

- letter statistics in a text file

```
try:
    with open("text.txt", "r", encoding="utf-8") as file:
        data = file.read()
except FileNotFoundError:
    print("Missing file text.txt")
else:
    statistics = {}
    for chr in data:
        if chr.isalpha():
            statistics[chr] = statistics.get(chr, 0) + 1
    print("Letter statistics:")
    for letter, number in sorted(statistics.items()):
        print(f"{letter}: {number}")
```

- the get() method returns the value for the given key (a character) or a default value (0), if the key does not exist in the dictionary

Python - exceptions

- letter statistics in a text file

```
Litwo! Ojczyzno moja! ty jesteś jak zdrowie:
Ile cię trzeba cenić, ten tylko się dowie,
Kto cię stracił. Dziś piękność twą w całej ozdobie
Widzę i opisuję, bo tęsknię po tobie.
```

```
D: 1
I: 1
K: 1
L: 1
O: 1
W: 1
a: 5
b: 4
c: 6
d: 4
```

```
e: 11
i: 16
j: 6
k: 4
l: 2
m: 1
n: 5
o: 14
p: 3
r: 3
```

```
s: 5
t: 11
u: 1
w: 5
y: 3
z: 7
ą: 1
ć: 2
ę: 8
ł: 2
ś: 3
```

Python - exceptions

- if we don't want to take any action after an exception occurs, but still want to handle it, we can use the pass statement in the except block

```
x = float(input("Enter x: "))
y = float(input("Enter y: "))
try:
    z = x / y
except ZeroDivisionError:
    print("Division by zero error!")
except ValueError:
    pass
else:
    print(f"The result is: {z}")
```

- the pass statement is usually used when we are not yet writing any code, but plan to add it in the future

End of lecture no. 9

Thank you for your attention!