File handling

Files can be resources for our programs. We can read data input from files, and also we can write out some results from our code.

You should be aware of some aspects of dealing with files:

- File resources have to be accessed through the operating system.
- File resources need to be explicitly opened and closed in our code, otherwise the OS might run out of its resources.
- File operations are *very slow*, at least compared to in-memory operations.
- A file is a shared resource which means that others may change its content while you are reading it and vice versa.
- Files can be opened in text or in binary mode. Text files try to deal with character encodings while reading and writing. Binary mode deal with raw data.

Jupyter notebook

- File handling

 Journey

- Projects
- Review
- Curriculum
- Profile
- Modules
- Feedback
- My Groups
- Attendance
- Self Assessment

LEARNING PATH
Programming Basics - Python
Web and SQL - Python Flask
OOP - Java
OOP - C#
Advanced - Java Spring
Advanced - ASP.NET
Download Jupyter notebook

File handling¶

File handling is an important part of any application. If you're thinking about uploading your profile picture to social media, or just sending a PDF as an emial attachment, you are dealing with file handling in the backend.

Python has several functions for creating, reading, updating, and deleting files.

Resources during this tutorial¶

All our demo files has the exact same content:

Opening and reading from files¶

The key function for working with files in Python is the **open()** function.

The open() function takes two parameters; **filename**, and **mode**.

There are four different methods (modes) for opening a file:

- "r" - Read - Default value. Opens a file for reading, error if the file does not exist
- "a" - Append - Opens a file for appending, creates the file if it does not exist
- "w" - Write - Opens a file for writing, creates the file if it does not exist
- "x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

- "t" - Text - Default value. Text mode
- "b" - Binary - Binary mode (e.g. images)

To open a file for reading it is enough to specify the name (and the relative directory path) of the file:

In [1]:
```
f = open("files/demofile.txt")
```
It means, there should be a *files* directory next to this notebook or python script, with a file called *demofile.txt*

To read it's content line-by-line into a variable:

In [2]:
```
lines = f.readlines()
```

```
lines
```
Out[2]:
```
['this is a demo file with demo content
',
 'but it also has some extra content!
',
 'and even more...']
```
As you see, this file contains 3 rows and the **readlines()** method reads these into a python list. The first and second line has a '\n' at the end, which is a newline character.

If you want to explicitly specify the **modes**, you could do this:

In [3]:
```
f2 = open("files/demofile.txt", "rt")
lines2 = f2.readlines()
lines2
```
Out[3]:
```
['this is a demo file with demo content
',
 'but it also has some extra content!
',
 'and even more...']
```
As you see, the result is the exact same.

Reading from files - other important methods¶

If you just want to read the whole content, use the **read()** method:

In [4]:
```
f = open("files/demofile.txt", "r")
file_content = f.read()
file_content
```
Out[4]:
```
'this is a demo file with demo content
but it also has some extra content!
and even more...'
```

By default the read() method returns the whole text, but you can also specify **how many characters** you want to return:

In [5]:
```
f = open("files/demofile.txt", "r")
f.read(7)
```
Out[5]:
```
'this is'
```

If you want to loop through all the lines, this is the simplest solution:

In [6]:
```
f = open("files/demofile.txt", "r")
```

```
for x in f:
    print(x)
this is a demo file with demo content

but it also has some extra content!

and even more...
```

In this case, an opened file acts as an **iterator**.

- An iterator is an object that contains a countable number of values (3, right now)
- An iterator is an object that can be iterated upon, meaning that you can traverse through all the values (printed them)

Closing files¶

You should think about files as **opened resources**.

When you are working on too much task, dealing with too many inputs, learning about too many things, you get overwhelmed. The same happens with the computer's memory.

It's recommended to **close these files** when you've finished working with them. (I recommend the same for the problems mentioned above :D)

If you want to close a file, use the **close()** method:

In [7]:
```
f = open("files/demofile.txt", "r")
print(f.readlines())
f.close()
['this is a demo file with demo content
', 'but it also has some extra content!
', 'and even more...']
```

Write to existing files¶

To write to an existing file, you must **add a parameter to the open()** function:

- "a" - Append - will append to the end of the file
- "w" - Write - will overwrite any existing content

Open the file "demofile2.txt" and append content to the file:

In [8]:
```
f = open("files/demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("files/demofile2.txt", "r")
```

```
print(f.read())
this is a demo file with demo content
but it also has some extra content!
and even more...Now the file has more content!
```
As you see, the content is appended to the last (3rd) line! If you want to append it as a new line, you should append a *newline character* first. It's always *worth to put an extra line* after your last line to avoid these kind of situations.

Open the file "demofile3.txt" and overwrite the content:

In [9]:
```
f = open("files/demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the appending:
f = open("files/demofile3.txt", "r")
print(f.read())
Woops! I have deleted the content!
```
The "w" method will overwrite the entire file.

Create a new file¶

To create a new file in Python, use the **open()** method, with one of the following parameters:

- "x" - Create - will create a file, returns an error if the file exist
- "a" - Append - will create a file if the specified file does not exist
- "w" - Write - will create a file if the specified file does not exist

Create a file called *myfile.txt* in the *files* directory:

In [10]:
```
f = open("files/myfile.txt", "x")
```
Create a new file if it does not exist:

In [11]:
```
f = open("files/myfile.txt", "w")
```
Since it was already created in the previous step, nothing really happens :)

Example¶

In this example we create a file from scratch, append a lot of numbers to it, reopen the file and finally print the content of it.

In [12]:
```
example_file = open('files/example.txt', 'w')

for i in range(10):
    example_file.write(f'Line {i+1}.
```

```
')

example_file.close()

reopened_file = open('files/example.txt', 'r')

for line in reopened_file:
    print(line)

reopened_file.close()
Line 1.

Line 2.

Line 3.

Line 4.

Line 5.

Line 6.

Line 7.

Line 8.

Line 9.

Line 10.
```

With statement in Python¶

In Python you need to give access to a file by opening it. You can do it by using the open() function, as you have seen above.

With the **with** statement, you get better syntax and exceptions handling.

The with statement simplifies exception handling by encapsulating common preparation and cleanup tasks. In addition, **it will automatically close the file**. The with statement provides a way for ensuring that a clean-up is always used.

Regular way:

In [15]:
```
file = open("files/demofile.txt")
data = file.read()
print(data)
file.close()  # It's important to close the file when you're done with it
this is a demo file with demo content
but it also has some extra content!
and even more...
```
With **with** statement:

In [17]:

```
with open('files/demofile.txt') as file:
    data = file.read()
    print(data)
this is a demo file with demo content
but it also has some extra content!
and even more...
```

Notice, that we didn't have to write *file.close()*. That will automatically be called.