8 Working with files

Sometimes variables and user input are not enough and programmers need to store data in a more permanent way or read data from files. So far, most of the examples contained programs that run for a short time and give a short output. After restarting, a program works on a clean slate and computes the result again. However, sometimes is necessary to keep the results in a more permanent form. One of the simplest ways to maintain data is by reading and writing data to text files. A **text file** is a sequence of characters stored on a permanent medium (hard disk, USB stick, etc). Text files can contain very different kinds of data and are quite useful when it comes to storing information.

Python has couple of built-in methods to work with files. Before a file can be accessed, either for writing or for reading, a **file object** must be created (example 8.1).

Example 8.1. General template for creating a file object

```
file_object = open(filename, attribute)
```

`filename` is the name of the file that is to be accessed (including its path, if necessary) and `attribute` is the type of the action performed on the `filename`:

- `"r"` – open an existing file for reading,
- `"w"` – open file for writing; if `filename` does not exist, it is created, if it exists – the file is cleared,
- `"a"` – open an existing file for appending – new text will be added at the end of the already existing text in a file,
- `"r+"` – open an existing file for reading and writing,
- `"w+` – same as `"r+"`, if `filename` does not exist, it is created,
- `"a+"` – same as `"w+"`, except that data is appended to the end of the file.

It is a good practice to close a file, when access to it is no longer required, using `file_object.close()` method on a file object. Keeping too many files open may exhaust the system resources, causing a program to abort. If a file is not closed, it gets closed when the program ends. However, if a program raises an error, file might not be closed properly and the contents might be lost. More secure way of opening, using a **context manager**, which does not require using `close()` method, is presented in example 8.2.

Example 8.2. Suggested way to open files using a context manager

```
with open(filename, attribute) as file: # give file object an alias and use it from this point
    to access the contents
   file.read()
```

One of the common problems while working with files is that different systems use different characters to indicate the end of the line: a newline character `\n` (Linux, macOS) or `\r`, some systems use both (Windows). These inconsistencies may cause problems while moving files between different operating systems. To see the string representation character at the end of the line (and other white space characters), use `repr()` function (example `[1]`).

In [1]:
```
s = "new\t line \n"
print(s)
print(repr(s))
```

```
new     line

'new\t line \n'
```

8.1 Reading data from file

There are several methods for reading data from a file (example 8.3).

Example 8.3. Different methods of reading data from file

```
with open(filename, "r") as file:
    n_characters = file.read(n)  # 1
    one_line = file.readline() # 2
```

Example 8.4. Reading a whole file at once

```
with open(filename, "r") as file:
    text = file.read()
```

The `read(n)` method reads `n` characters from a file and returns them as a string. If `n` value is not provided, all the characters in a file are read into one variable including newline characters (example **??**). To read a current line, use `readline()` to load a line until the newline character is encountered, (`\n`) or `readline(n)` to read `n` characters from the current line. All the lines in a file can be read using `readlines()` (Example 8.5) method on a file object. The method returns a **list of strings**, each element of a list representing a line **ending with a newline character** (the newline character is not removed despite the lines being separate elements of a list).

Example 8.5. Reading the whole file and separating the lines

```
with open(filename, "r") as file:
    lines = file.readlines(n)
```

If we want to process the line while reading, it is convenient to use method presented in example 8.6. The `file` object acts as iterator and returns a next line in every subsequent iteration, until the end of file is reached, enabling reading a file one line at a time (iterating over lines). This method is suitable for large files.

Example 8.6. Processing lines of the file on the go

```
file = open(filename, "r")
for line in file:      # implicitly reads a file, line by line
    <do something with line>
file.close()
```

8.2 Writing data to file

Writing string data to a file can be accomplished by using one of two methods from the standard library. `file.write(string_var)` writes a string variable to a file object `file` and `file.writelines(string_var)` is used to save a list of strings to a file object referenced by name `file`. **Neither method** adds a newline character to the end of the line. Therefore, if you want the text split to multiple lines, you should remember to add the newline character yourself. Method `writelines` is the opposite of `readlines` – `readlines` does not remove the newline character and the resulting list of strings can be used in `writelines` to obtain the same results as the original file which was read. It is also important to remember, that the values saved to a file should be of type `string`. Example `[2]` contains a command casting an integer to a string.

Adding a newline character to each line is shown in example `[2]`. The return value is a number of characters written.

In [2]:
```python
file = open('example.txt', "w")
for i in range(100):
    file.write(str(i**2))
    file.write("\n")
file.close()
```

File can be written also by using a regular `print` function, the output needs to be redirected to a file, as in example `[3]`.

In [3]:
```python
f = open("example_2.txt", "w")
print("hello", "world", file = f)
f.close()
```

8.3 Other methods for reading and writing files

8.3.1 Text files

If a file has a repeated structure (the same number of columns in each line) it is fairly easy to import, the easiest being numeric files.

`NumPy` provides two methods for reading `txt` files: `loadtxt` and `genfromtxt`. `loadtxt` is a simple and fast text importer. The basic usage is loading the data from a file as floats (example ??). `skiprows` enables to skip the header in a file and `delimiter` separates values in a line with a delimiter. This method can only read purely numeric data (no missing values).

In [4]:
```python
import numpy as np

data = np.loadtxt("filename.txt", delimiter=",", skiprows=1)
```

To deal with data in a more sophisticated way, `genfromtxt` can be used. It works slightly slower, but will not fail if a non-numeric type is encountered during its execution. The function will return `NaN` (not-a-number) instead for the values in a file that it cannot read, unless you provide other data type in the output table (see Section 8.3.2). In example ??, data read is supposed to be tab delimited (\t is a representation of tab character). The method also allows to skip the header (`skip_header`) or footer (`skip_footer`) and define the columns which should be used in the output array.

In [5]:
```python
import numpy as np

data = np.genfromtxt("filename.txt", delimiter="\t",skiprows=1)
```

To export numeric data to a text file, `NumPy` provides a function `savetxt` (example ??).

In [6]:
```python
import numpy as np

x = np.arange(10)
data = np.savetxt("filename.txt", x, delimiter="\t") # use Tab as delimiter
```

8.3.2 Working with csv files

Comma-separated values, `csv` for short is a commonly used type of text file containing columns of values (often both text and numeric values) separated with a comma. Usually the first line of the file includes a header – names of the columns. Another variant of a `csv` file is separating the columns with a semicolon (`;`), used when the decimal separator for numbers is a comma (quite often the case while exporting data to csv from Excel).

Python provides several methods to read and write `csv` files:

- using `read` or `readlines` and splitting each line by the comma character;
- using `csv` module, returns a csv object that can be processed line by line; each line is returned as a list of strings
- using `numpy` functions, setting delimiter to `;`.
- using `read_csv` from the `pandas` module; the data is read to a `DataFrame` object and can be further processed by `pandas`;

The decision, which method to use, depends on the complexity and size of the file, but also on the contents. `Numpy` arrays can store only one type of the data, so files containing both text and numbers cannot be imported that way (or we need to remember to adjust the datatype accordingly).

For the examples below, a very simple `csv` file, containing the characters as presented in Example 8.7 stored in a file named `new_file.csv`. Notice, that the file could have an extension `txt` instead of `csv` and all the methods would work in the same way, as `csv` file is still a text file.

Example 8.7. Example `csv` file

```
name,lname,age
Paul,Smith,23
Anna,Jones,43
```

8.3.2.1 Standard library methods Reading a file with built-in methods was presented in example [7].

In [7]:
```python
with open("new_file.csv", "r") as f:
    lines = f.readlines() # read the whole file into separate lines

# divide the lines into individual elements, get rid of the newline character at the end of the line
lines = [line.strip().split(",") for line in lines]
print(lines)
# can be casted to an np.array if needed
```

```
[['name', 'lname', 'age'], ['Paul', 'Smith', '23'], ['Anna', 'Jones', '43']]
```

Saving a file (example [8]).

In [8]:
```python
# writing files
data = [['name', 'lname', 'age'], ['Paul', 'Smith', '23'], ['Anna', 'Jones', '43']]
# join each line with a comma
# writelines does not include a newline character, so first, we need to add it to data
data_to_save = [",".join(line) + "\n" for line in data]

with open("new_file_standard.csv", "w") as f:
    lines = f.writelines(data_to_save)
```

8.3.2.2 NumPy Reading and writing `csv` files using NumPy can be done using `genfromtxt` / `loadtext` and `savetext` functions. `loadtxt` and `genfromtxt` return an array, so if the data in the file contains both letters and numbers, by default, the non-numeric characters are changed to `nan` (not-a-number, value that represents an incorrect value) (example `[9]`). Since our example data contain two text columns, the first and second column in a resulting array contain `nans`. The `skip_header` parameter allows to skip `n` lines of the header – in example there is only one header line that contains the names of columns. More information on `np.genfromtxt` parameters can be found here: `https://numpy.org/doc/stable/reference/generated/numpy.genfromtxt.html`.

In [9]:
```python
import numpy as np

data = np.genfromtxt("new_file.csv", delimiter=",", skip_header=1)
print(data)
```

```
[[nan nan 23.]
[nan nan 43.]]
```

We can change the default type of data in a `ndarray` and, by this simple adjustment, read all the data as string data (example `[10]`).

In [10]:
```python
import numpy as np

# read every value as str
data = np.genfromtxt("new_file.csv",
                     delimiter=",",
                     dtype="str",
                     skip_header=1,
                     encoding="utf8")
print(data)
```

```
[['Paul' 'Smith' '23']
['Anna' 'Jones' '43']]
```

`np.genfromtxt` allows to return every column as a single variable (in this case, a nd.array), which sometimes can be useful, especially, if we aim to process string data and numeric data separately.

In [11]:
```python
first_names, last_names, age = np.genfromtxt("new_file.csv",
                                             delimiter=",",
                                             unpack=True,
                                             skip_header=1,
                                             dtype="str",)
print(first_names)
```

```
['Paul' 'Anna']
```

An example of saving the tabular data, with both string values and numeric values is shown in listing `[12]`. Notice, that in case of mixed data (or, generally, saving string data), `fmt` parameter describing the format of the saved file is required. The format description follows the rules for string formatting (included in section 8.4).

In [12]:
```python
# saving tabular data to a file
# providing the format: fmt, since we save mixed data
np.savetxt("new_file_np.csv", data, # data is the array that we want to save
           fmt="\%s \%s \%d",
           delimiter=",",
           newline="\n",
           header="name,lname,age",
           comments="")
# by default, numpy adds # to header, redefine comments to skip it;
# also, by default line with # in the beginning is skipped while reading with
    genfromtxt
```

8.3.2.3 `csv` module `csv` module is dedicated to work with a csv file. Reading and writing a file is done using an iterator object, which returns one line at a time (example `[13]`). The line can be further processed before it is saved in some data structure.

In [13]:
```python
import csv

with open("new_file.csv", "r") as f:
    csvr = csv.reader(f)
    for line in csvr:
        # the line here is not saved, just printed, but we can append it to a list
            for example
        print(line)
```

```
['name', 'lname', 'age']
['Paul', 'Smith', '23']
['Anna', 'Jones', '43']
```

8.3.2.4 `pandas` During this course we do not work with `pandas` module, but there are two methods worth mentioning in terms of reading and writing `csv` files (example `[14]` and `[15]`). `Pandas` reads the data into its own data structure called `DataFrame`, which is not discussed during the course. At this point, if we need a column from the `DataFrame`, we can cast it to another object, like a list or a `ndarray`.

In [14]:
```python
import pandas as pd

data = pd.read_csv("new_file.csv")
# take column called "name" and cast it to a list
names = list(data["name"])
print(name)
```

```
['Paul', 'Anna']
```

Saving data to `csv` using `pandas` requires using a method of a `DataFrame` object, hence, we need to create one, before we can store the data in a file. Necessery code is provided in example `[15]`.

In [15]:
```
# first, the data needs to be transformed to DataFrame, as saving to csv is a method
    of a DataFrame object
data = [['name', 'lname', 'age'], ['Paul', 'Smith', '23'], ['Anna', 'Jones', '43']]
# parameter columns define names of columns
data_to_pd = pd.DataFrame(data[1:], columns=data[0])
data_to_pd.to_csv("new_file_pd.csv",
                  index=False, #skip index of rows
                  )
```

Apart from a method for reading csv files mentioned in the next subsection, `pandas` provides also a method to read data stored in an Excel file (both `xls` and `xlsx`), it is called `read_excel`.

8.3.3 MATLAB files

Reading MATLAB data files (`mat` files) can be done with one of the methods from `SciPy` library. The data is loaded into dictionary, the variable names are treated as keys in this dictionary (example `[16]`).

In [16]:
```
import scipyio as sio

data = sio.loadmat("data.mat")
```

Matlab data files, starting from V7.3 cannot be read by `loadmat`, PyTables should be used instead.

Saving data to a `mat` file requires two arguments: a filename and a dictionary containing the names of the variables as keys (given as string objects) and their actual variables (example `[17]`).

In [17]:
```
import scipyio as sio

x = 5
y = list(range(10))
variables = {"x": x, "y": y}
data = sio.savemat("data.mat", variables)
```

8.3.4 Any kind of data

Any kind of data can be stored in a file and then loaded in a few simple commands. This capability is provided in Python by a `pickle` module. Let's assume that we want to save few variables, called `a`, `b` and `c` being of an arbitrary data type. They can be saved to a file as in example `[18]` and later loaded as in an example `[19]`.

In [18]:
```
import pickle

a = 5
b = list(range(10))
c = {a: b}
file = open("file_data", "wb")
pickle.dump(a, file)
pickle.dump(b, file)
pickle.dump(b, file)
file.close()
```

In [19]:
```python
import pickle

file = open("file_data", "rb")
a = pickle.load(file)
b = pickle.load(file)
c = pickle.load(file)
file.close()
```

Most of basic Python data types (strings, lists, dictionaries, numeric data, shared objects) can be pickled, however system resources (files or sockets) and code objects (byte-compiled code) cannot be saved to a file with that method. Pickling is neither fast, nor space-efficient. Saving to `JSON` files might be faster and results in smaller files. Another drawback is that **pickling is not very secure**, and may result in loading malicious content. Therefore it should not be used for files that might have been altered by someone else.

8.4 Text formatting

In Python, several methods of text formatting are available:

- %-formatting – the oldest available method, similar to text formatting in C, not recommended.
- **format** (`str.format`) ;
- **f-string** – `f-string` provides the same formatting options as `str.format`, but the method of coding is more clear ;
- (starting from Python 3.8:) self-documenting with f-strings: `f"The length is length="` (example ??)
- using `Template` (not described in this document).

8.4.1 % operator

Formatting with `%` is the oldest available method. It allows to do simple positional formatting. As in example `[20]`, a string, which includes a format specifier (`%s` and `%.4f`), is followed with a `%` operator and variable names given as a tuple (because it takes only one argument). The first variable is put in place of the first formatter, the second – in place of the second formatter and so on.

Instead of positional matching, variables can be matched using aliases, see more details here: `https://docs.python.org/3/library/stdtypes.html#old-string-formatting`.

In [20]:
```python
name, price = "Alice", 32.52
print("Hello, %s! You need to pay %2.2f dollars!" % (name, price))
print("Hello, %s! You need to pay %2.1f dollars!" % (name, price))
print("Hello, %s! You need to pay %2.0f dollars!" % (name, price))
```

Out[20]:
```
Hello, Alice! You need to pay 32.52 dollars!
Hello, Alice! You need to pay 32.5 dollars!
Hello, Alice! You need to pay 33 dollars!
```

8.4.2 `format` method

Python 3 introduced a new approach to string formatting - a `format` method. Formatting is done by calling the format method **on a string object** (example `[21]`). As with `%` operator, the arguments can be taken positionally, you can refer to them using indices or aliases, as well as using `**` to unpack dictionaries.

In [21]:
```python
name, price = "Alice", 32.52
print("Hello, {}! You need to pay {} dollars!".format(name, price))
print("Hello, {0:s}! You need to pay {1:.3f} dollars!".format(name, price))
print("Hello, {0:s}! You need to pay {1:.1f} dollars!".format(name, price))
print("Hello, {0:s}! You need to pay {1:.0f} dollars!".format(name, price))
```

Out[21]:
```
Hello, Alice! You need to pay 32.52 dollars!
Hello, Alice! You need to pay 32.520 dollars!
Hello, Alice! You need to pay 32.5 dollars!
Hello, Alice! You need to pay 33 dollars!
```

Since we can refer to elements using their indices, we can modify their order in any way we want (example `[22]`).

In [22]:
```python
# switch elements
variable = 0.1
name = "Alice"
print("Name of person: {1}, fraction: {0}".format(name, variable))
```

```
Name of person: 0.1, fraction: Alice
```

`format` allows to define the keyword pairs to reference the values included in the string (example `[23]`). Example `[24]` contains code for referencing dictionary.

In [23]:
```python
# using substitutions for variable names
name = "Alice"
print("Name of person: {name:.20s}, fraction: {var:16.2f}".format(name=name, var
    =0.87)) # notice the additional white space before the value
```

```
Name of person: Alice, fraction:             0.87
```

In [24]:
```python
countries = {'country': 'Poland', 'capital': 'Warsaw'}
print("The capital of {country} is {capital}.".format(**countries))
```

```
The capital of Poland is Warsaw.
```

8.4.3 f-string

Starting with Python 3.6, a new method of string formatting is available called **formatted string literals** or **f-strings**. This approach allows to use embedded Python expressions inside string constants (examples `[25]` and `[26]`). **To use the f-string, prefix the string with letter f.**

In [25]:
```python
name, price = "Alice", 32.45
print(f"Hello, {name}! You're paying {price:3.2f} today!")
```

Out[25]:
```
Hello, Alice! You're paying 32.45 today!
```

In [26]:
```python
print(f"Name of person: {name:^10s}, fraction: {variable:08.4f}")
```

```
Name of person: Alice   , fraction: 000.1000
```

The syntax inside a string is similar to `format` method, but f-strings are faster than the previously mentioned methods. Apart from including variables, we can do some arithmetic or call functions inside the expression (example `[27]`) – everything that is enclosed in braces is treated as an expression.

In [27]:
```python
import math as m

print(f"2 + 2 = {2+2}")
print(f"sin(pi) = {m.sin(m.pi/2)}")
name = "ANNA SMITH"
print(f"{name.title()}")
```

```
2 + 2 = 4
sin(pi) = 1.0
Anna Smith
```

Limitations of f-strings:

- the braces can't be empty (`f"aabb"` won't work),
- f-strings cannot contain a backslash ('\') characters – which is sometimes used to escape some characters or to introduce a newline character, or tab space; a workaround uses a variable which includes this character (examples `[28]` and `[29]`).

In [28]:
```python
# careful with quotation marks
countries = {'country': 'Poland', 'capital': 'Warsaw'}
print(f"The capital of {countries['country']} is {countries['capital']}.")
# print(f"The capital of {countries["country"]} is {countries["capital"]}.") #won't
    work
#print(f"The capital of {countries[\"country\"]} is {countries[\"capital\"]}.") #won
    't work
```

```
The capital of Poland is Warsaw.
```

In [29]:
```python
# workaround fro backslash limitations:
nl = '\n'
print(f'line_1{nl}line_2')
```

```
line_1
line_2
```

From version 3.8, an adjustment to f-string is available – a named f-string. The idea is to include the name of the variable along its value. New syntax requires an additional `=` operator in the string constant, but the rest remains the same (example `[30]`).

In [30]:
```python
variable = 0.1
name = "Alice"
print(f"Name of person: {name=}, {variable=}")
print(f"Name of person: {name=:>10s}, {variable=:4.2f}")
```

Out[30]:
```
Name of person: name='Alice', variable=0.1
Name of person: name=   Alice, variable=0.10
```

8.4.4 Format specifiers

In the previous subsections you might have noticed, that some braces included digits, letters and special characters that seemed to modify the looks of the output string. Those symbols werer part of a **format specifier**. **Format specifiers** define the result of the formatting. They let control the fill character, alignment, width, precision and type of data that is inserted in the replacement field. The same format specifiers work for the `format` method and a f-string. The general from of the format specifier is:

```
:[[<fill>]<align>][<sign>][#][0][<width>][<group>][.<prec>][<type>]
```

The brackets in the format specifier rule suggest, that the element is optional – in fact, all the components are, some of the previous examples did not define any of the formatting rule and still output the variables correctly.

The components of the format specifier are:

- `:` – separates the format specifier from the rest of the replacement field, e.g. the name of the variable, alias or index,
- `fill` – defines how to pad values that do not take the entire `width`, ignored, if the output occupied the entire field width,
- `align` – defines how to justify values that do not take the entire `width`,
- `sign` – specifies if the sign of the number is included for numeric values,
- `#` – defines an alternate output form depending on the presentation types,
- `0` – values are padded on the left with `0` instead of another ASCII characters,
- `width` – defines the **minimum** width of the output field,
- `group` – defines a grouping character for numeric output (like a thousands separator),
- `.prec` – specifies: (1) the number of decimal digits in a floating point number, (2) the maximum output width for string types,
- `type` – defines the presentation type (type of conversion performed on the argument);

The order of the elements in the format specifier is important. Some of the possible values for the above-mentioned components are listed in Table 8.1.

8.5 Filenames and paths

Files are organized into folders (also called directories). Every running program has a `current directory` – a folder which is default for program operations. If we try to open a file for reading or writing, unless specified otherwise, Python looks for it in a `current working directory`. Working with files and paths is easier with the `os` module (`os` stands for *operating system*), which provides functions, e.g. to check if a file exists.

A string like `C://user/example_directory` that identifies a file or directory is called a path. A single filename is also considered a path, but it is a **relative path** – it relates to current directory. To check current directory, use `getcwd` function from `os` module (example `[31]`).

In [31]:
```python
import os

cwd = os.getcwd()
print(cwd)
```

Table 8.1. Values used for format specifier

component	value	description
align	<	left-aligned (default for strings)
align	>	right-aligned
align	^	centered
align	=	only for numeric values when sign is included
fill		any character can be used to fill the extra space (except for curly braces)
sign	+	the displayed value is always displayed with a leading sign (both positive and negative values)
width	<number>	the minimum width of the output field, defined with a number
group	,	separate groups of numbers with a comma (each group of three digits is separated)
group	_	separate groups of numbers with an underscore
.prec	<number>	number of digits after decimal point for floating point representation
type	s	string
type	d	integer number
type	b	binary integer
type	o	octal integer
type	x or X	hexadecimal integer
type	e	floating point: exponent notation. Prints the number in scientific notation using the letter ’e’ to indicate the exponent.
type	E	floating point: exponent notation. Same as ’e’ except it converts the number to uppercase.
type	f or F	floating point: fixed point. Displays the number as a fixed-point number.
type	F	floating point: fixed point. Same as f except it converts the letter to uppercase.
type	g	floating point: general format. This prints the number as a fixed-point number, unless the number is too large, in which case it switches to e exponent notation.
type	G	floating point: general format. Same as g except switches to E, if the number gets to large.
type	n	floating point: number. This is the same as g, except that it uses the current locale setting to insert the appropriate number separator characters.
type	%	Percentage. Multiplies the number by 100 and displays in fixed (f) format, followed by a percent sign.
type	” (None)	similar to ’g’, except that it prints at least one digit after the decimal point

Out[31]:
```
...21Z/LNM/class\_08\_io\_str\_formatting
```

A path, that begins with a hard drive letter (in Windows) or with `/` character in Linux, is an **absolute path** – it does not depend on a current directory. To find an absolute path of a file, use `os.path.abspath` (example `[32]`).

In [32]:
```python
import os

filename = "example.txt" # file in current directory
f_path = os.path.abspath(filename)
```

Out[32]:
```
...21Z/LNM/class\_08\_io\_str\_formatting/example.txt
```

Many other functions for working with files and folders are provided in `os` module:

`os.chdir(path)` – changes directory,

`os.mkdir(name)` – creates directory,

`os.rmdir(path)` – removes directory,

`os.listdir(path)` – lists all the items in the given directory,

`os.path.join(dirname1, dirname2)` – creates a system-specific path by joining elements given as arguments,

`os.path.exists(path)` – checks if the path is correct (leads to a file or directory),

`os.path.isdir(path)` – checks, whether a path is a directory,

`os.path.isfile(path)` – checks, whether a path is a file,

`os.path.basename(path)` – returns filename without extension,

`os.path.splitext(path)` – returns `basename` and extention of a file,

`os.path.getsize(path)` – returns filesize,

`os.path.abspath(path)` – returns absolute path of a relative path given

8.6 Exercises

Exercise 8.1.

1. Open a blank file and write a few lines about yourself or about your day. Save the file.
2. Write a program that reads the file and prints what you wrote three times. Use methods below:
   - reading the entire file at once
   - reading the file while looping the file object
   - reading the lines of the file and process them outside the with block.

Exercise 8.2. Create two sets of coordinates, x and y. Save them to a `csv` file. Then, read the file, convert lists to arrays and plot the results. The file `xy.txt` contains two columns of numbers corresponding to x, y coordinates. Read the values to two lists, convert lists to arrays and plot the points.

Exercise 8.3. Define a polynomial $P(x)$. Compute its values for 100 points in the defined interval. Store the results (x and $P(x)$) in a text file with two columns, separated with a tab. Each pair should be in a separate line.

Exercise 8.4. Read data in the file `new_york.csv`. Skip the header. Change missing values (-99) to NaN. Compute an average of temperature and precipitation over the years.

Exercise 8.5. Using the file `sonnets.txt`, save each sonnet to a separate file. Luckily for you, each sonnet has 14 lines :)

Exercise 8.6. Count the words in a `sonnets.txt`.

Exercise 8.7. Read the file `sonnets.txt`. Count the number of appearance of each word in a file. You can skip punctuation (`word` and `word!` is the same word). Save results to a text file.

Exercise 8.8. Using one of the formatting methods, print the table as in example below and save it to a file `oscars.txt`.

```
-----------------------------------------------
|Year  |   Movie title                  | Oscars |
***********************************************
|1959  |Ben Hur                         |......11|
|1997  |Titanic                         |......11|
|2003  |LOTR: The Return of the King    |......11|
|1961  |West Side Story                 |......10|
-----------------------------------------------
```

Exercise 8.9. Read a file `pi_million.txt` including million digits of a pi number. Check if those digits contain your date of birth sequence given as 'yymmdd'.

8.7 Additional material

- Methods of File objects, Python tutorial: `https://docs.python.org/3/tutorial/inputoutput.html#methods-of-file-objects`
- More on `NumPy` data loading methods: `https://likegeeks.com/numpy-loadtxt-tutorial/`
- Using `csv` to read and write `csv` files: `https://docs.python.org/3/library/csv.html`
- Reading and writing `mat` files: `https://docs.scipy.org/doc/scipy/reference/tutorial/io.html`

- Text formatting:
  - str.format: `https://docs.python.org/3/library/string.html#string-formatting`
  - More information on formatting: `https://realpython.com/python-formatted-output/`
  - f-string: `https://www.python.org/dev/peps/pep-0536/`, `https://www.geeksforgeeks.org/formatted-string-literals-f-strings-python/`, `https://realpython.com/python-f-strings/#simple-syntax`
  - basics of string formatting: `https://docs.python.org/3/library/stdtypes.html#printf-style-string-f`
  - = operator: `https://www.geeksforgeeks.org/new-operator-in-python3-8-f-string/`
  - More information on f-strings: `https://docs.python.org/3/reference/lexical_analysis.html#formatted-string-literals`

References

Beazley, D. M. (2009). *Python essential reference.* Addison-Wesley Professional.

Ceder, N. (2018). *The quick Python book.* Simon and Schuster.

Downey, A. B. (2016). *Think Python. How to Think Like a Computer Scientist.* O'Reilly Media, 2nd edition.

Lubanovic, B. (2014). *Introducing Python: Modern Computing in Simple Packages.* O'Reilly Media, Inc.

Lutz, M. (2013). *Learning Python.* O'Reilly Media, 5th edition.

Python (2021). Python 3.10.0 documentation. `https://docs.python.org/3/` [Accessed 17 October 2021].