Python from Zero

Modules

Modules

Much of Python's versatility comes from *modules*

- "Standard" modules and third-party modules
- Plain Python has it's limits, but with additional modules, you can do (almost?) everything with it (and often easy and convenient)

Custom Modules

Also a good way to organize larger projects

- A lot of code in a single Python script (or Notebook) is hard to read and maintain
- Create own modules to organize code

Using Modules

First of all, *load* modules

```python
import module_name
import other_module_name as om # define an alias

module_name.useful_function()  # call `useful_function` from the module `module_name`

om.other_useful_function()     # use the alias to call `other_useful_function`
                               #   from `other_module_name`
```

You can also specify which parts of the module you want to import:

```python
from module_name import useful_function, useless_function
from other_module_name import other_useful_function as of # with alias

useful_function() # now no need for `module_name.` anymore
of()
```

Important Modules

`numpy`

- One of the most widely used libraries, offers fast and efficient handling and storage of large amounts of numerical data, and useful classes to represent high-dimensional arrays

`matplotlib`

- Most common library for data visualization (creating plots)

`pandas`

- Offers the class `DataFrame` that is very similar to the `R` data structure and allows handling large amounts of data

and many more

Environments

Sometimes, certain modules may not play nice with each other

- In some project, you may rely on a very specific version of a module, e.g. Numpy
- But another library that you need for another project will only install with a *different* version of aforementioned module (e.g. Numpy)

Environments

Python package managers (like `pip` or `conda`) allow having multiple *environments*

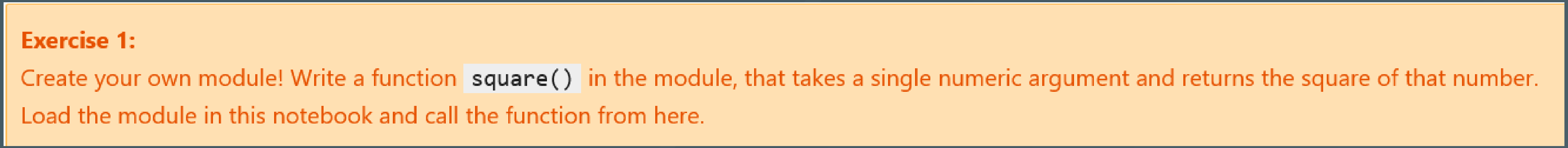
- Think of an environment as an encapsulated installation of Python and modules
- You can create an environment with the certain Numpy version
- And another environment with the other library and the correct Numpy version for that

This can be really useful! However, we will stay in the default environment...

It's Your Turn!

- Open the notebook `Session_3.ipynb` and work through it until before the "Interlude: Plain File Loading in Python" section

Do the exercise!



Exercise 1:

Create your own module! Write a function `square()` in the module, that takes a single numeric argument and returns the square of that number. Load the module in this notebook and call the function from here.

Interlude: Basic File Loading and Writing

So far, we hard-coded the data that we used in our programs

```
x = [1,2,3] # hard coded, usually not done
```

In reality, this is not feasible at all!

Usually, a program reads data to work with from a file

Although there are often better options using additional modules, Python offers some basic functionalities to read and write files

Interlude: Basic File Loading and Writing

Basic syntax:

```
with open("path/to/a/file.txt", "rt") as file_handle:
    # do stuff with the file_handle
```

`"rt"` stands for *read text*, i.e. `open()` knows that it should open the file for reading and that the file is supposed to be a text file (alternative: binary, beyond our scope!)

```
with open("path/to/a/file.txt", "wt") as file_handle:
    # do stuff with the file_handle
```

`"wt"` stands for *write text*, i.e. `open()` knows that it should open a text file for writing

Interlude: Basic File Loading and Writing

File Handle

This object is our connector to the file we just opened

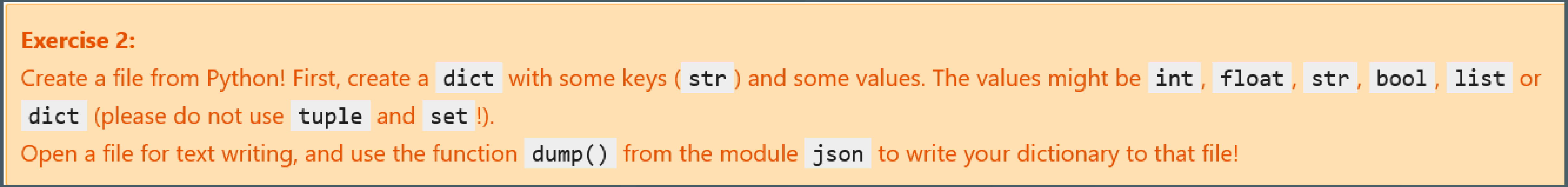
It has methods to read from and to write to a file (but make sure you opened the file with the correct `rt` or `wt` argument!)

- `file_handle.readlines()` - returns the entire file content as a list of `str`, one element per line
- `for line in file_handle:` - iterate line by line through the file (reading)
- `file_handle.write(str)` - write the string `str` to the file
- `file_handle.writelines([str1, str2, ...])` - write all strings in the list to the file

It's Your Turn!

- Open the notebook `Session_3.ipynb` and work through the "Interlude: Plain File Loading in Python" section

Do the exercise!



Exercise 2:
Create a file from Python! First, create a `dict` with some keys (`str`) and some values. The values might be `int` , `float` , `str` , `bool` , `list` or `dict` (please do not use `tuple` and `set` !).
Open a file for text writing, and use the function `dump()` from the module `json` to write your dictionary to that file!

Note that libraries like `json` or `pandas` sometimes offer better ways to read and write files than plain Python

Numpy

One of the most important third party libraries for Python

- If you work with numerical data, you will sooner or later use Numpy
- In machine learning, you will definitely use Numpy

Numpy offers *fast and efficient* ways to work with numerical data

- `numpy.ndarray` the main class of Numpy
- Can be a single value, a 1-dimensional list, 2-dimensional matrix, and even more dimensions
- Although you *can* implement higher-dimensional objects with Python `list`s, Numpy is much faster and uses less memory (due to implementation details)

Numpy

`ndarray` vs. `list`

- List is dynamic, you can append and remove values
- List elements can have any type
- Nested lists of any length
- List is comparatively slow and big

Numpy

`ndarray` vs. `list`

- Size of an `ndarray` is fixed, you should preferably know the final size before you create it
- Only one type per object, e.g. `np.int64` or `np.float64`
- In a matrix (2D array), fixed row and column sizes (`N x M matrix`) (also applies for higher dimensions)
- Very fast and efficient, Numpy also comes with useful operations such as matrix multiplication

Numpy

`ndarray`

```
import numpy as np

np.array([1,2,3])       # create a 1-dim array from a list
```

```
[1 2 3]
```

```
r = 3
c = 2
np.zeros(shape=(r,c))  # create a 2-dim array filled with zeros. Note the shape argument,
                       #   a tuple specifying the number of rows and columns
```

```
[[0. 0.]
 [0. 0.]
 [0. 0.]]
```

Numpy

```
np.full(shape=(2,3,2), fill_value=3.14) # create a 3-dim array filled with the value 3.14
```

```
[[[3.14 3.14]
  [3.14 3.14]
  [3.14 3.14]]

 [[3.14 3.14]
  [3.14 3.14]
  [3.14 3.14]]]
```

```
np.ones(shape=(3)) # create a 1-dim array (list) filled with ones
```

```
[1. 1. 1.]
```

Numpy

See it in action on the Notebook...