Prof. Dr. Gemma Roig
M.Sc. Alperen Kantarcı
M.Sc. Gamze Akyol

Programmieren für Studierende der Naturwissenschaften

Lecture 6 – Imports and NumPy

Contents

- L6: External Packages, Introduction NumPy and SciPy
  P6: Exercises

- L7:External Packages 2
  P7: Exercises

- L8: Handling external data and visualization
  P8: Exercises

- L9: Design of algorithms
  P9: Exercises (not graded) and independent work in small groups

Modularization

A module is a self-contained part of software, consisting of several sub-programs (procedures and functions) and data structures

• Modules are a means for encapsulation of software, i.e. separation of "Interface" and implementation and protection against "uncontrolled" error propagation.

• Programs or parts of programs become reusable without having to create and maintain redundant code

Modularization

•Larger, complex programs can be organized and structured through the use of modules. Functionalities can be integrated according to the modular principle

• Several developers (groups) can work on and test individual modules independently of each other once the interface has been defined.

Modularization

• Many programming languages support the module concept through built-in language resources, including Python.

• Modules can be compiled separately in many programming languages and provided in the form of program libraries

Modules in Python

- A module is a "container" that contains objects. Modules define a so-called namespace

- A module is created by saving a program in Python with the extension .py under the name

- The import <filename> statement makes modules available for the current program

Modules in Python

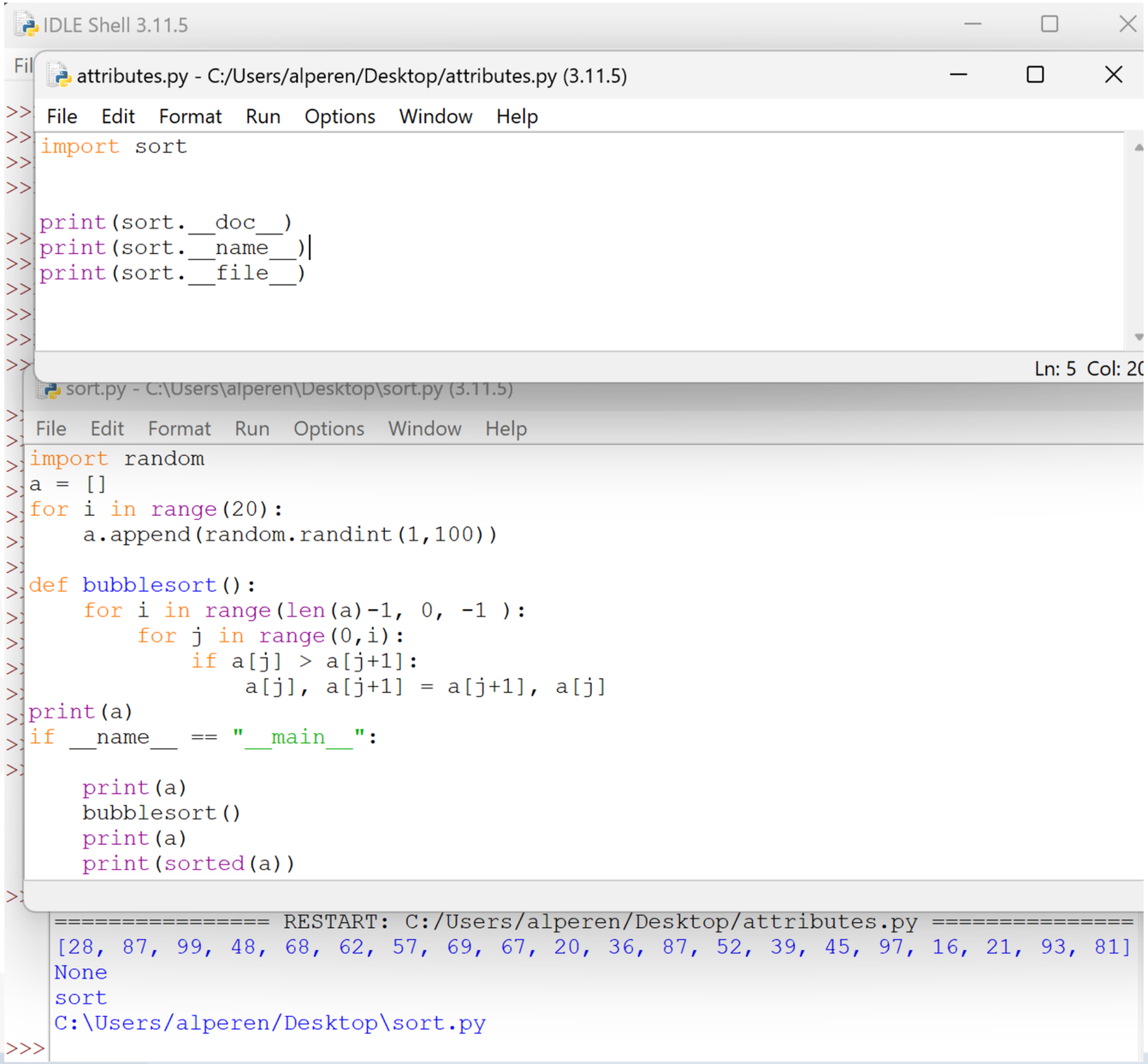
• Modules should have short names, all in lowercase, no special characters (especially no dots!). Underscores only if it increases readability

• Larger Python programs are often organized as a package of modules. Python package names use only lowercase letters (no underscore)

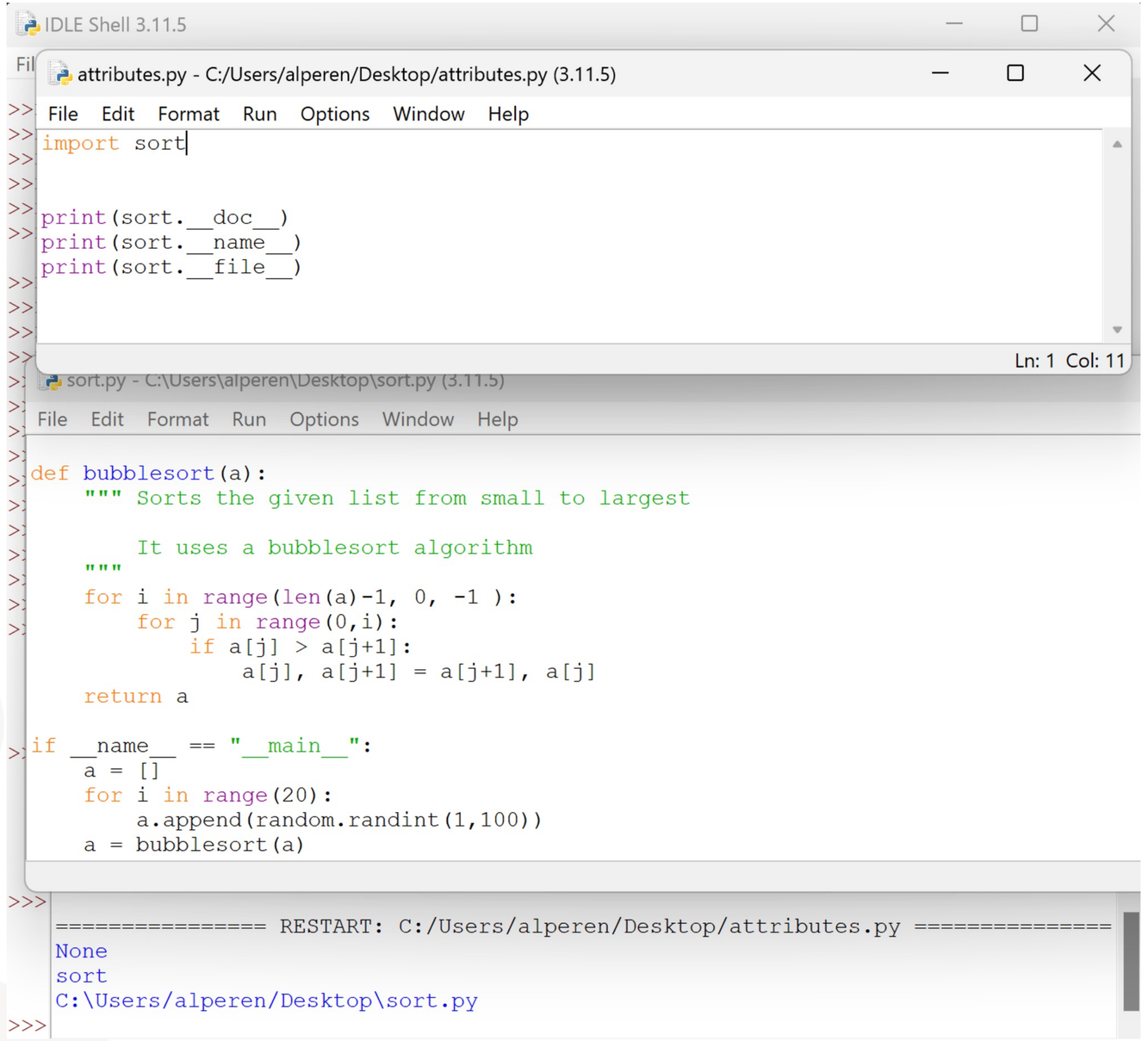
Automatically generated attributes of a module m

Attribute	Description
m.__dict__	The dictionary, which belongs to the module and supports especially the name management
m.__doc__	Doc string of the module
m.__name__	Module name
m.__file__	File from which the module was loaded

Attributes of a module



Attributes of a module



Imports of a module

`import module`	Zugriff auf Funktionen des Moduls durch Qualifizierung: **`module.Funktionsname`**
`from module import name [,name]*`	Zugriff auf die **Funktion** `name` des Moduls einfach durch `name`
~~`from module import *`~~	~~Alle benutzten Namen werden importiert und können unqualifiziert benutzt werden~~

- **Do not use the last variant because of potential name conflicts!**
- **Alternative:** `import module as <nickname>`

What happens during the import?

- The module file is searched in the computer/path and hopefully found

- The code is translated into bytecode (if necessary)

- The module is executed once i.e. the code on "top level” of the module (but only once, at the first import !)

- Thereby the names of the module are made known (like a "RunModule” or F5 in IDLE)

Main function

• Often you can find in Python programs/modules the statement

```
if __name__ == "__main__":
   <code>
```

• What is it for? Why do you need that?

• Any Python code can query at runtime what is called by using the variable `__name__` reads out

• This is

- `__name__ == "__main__"`, if it was started from the operating system, from the console or in IDLE with Run Module (F5)
- `__name__ == "<own module name>"` if it was imported

Main example

```
print('Hallo, hier ist das Modul testfile.')
print('Bei mir ist __name__ auf',__name__,'gesetzt.')

def sr_in_test ():
  print('Hier läuft sr_in_test .' )
  print('Bei mir ist __name__ auf',__name__,'gesetzt.')



def main():
  print('Hier könnte z.B.Initialisierungscode für das Programm testfile \
        stehen, wenn es als selbstständiges Programm aufgerufen wird.')


if __name__ =='__main__':
  main()
```

Main example

```
print('Hallo, hier ist das Modul testfile.')
print('Bei mir ist __name__ auf',__name__,'gesetzt.')

def sr_in_test ():
  print('Hier läuft sr_in_test .' )
  print('Bei mir ist __name__ auf',__name__,'gesetzt.')



def main():
  print('Hier könnte z.B.Initialisierungscode für das Programm testfile \
        stehen, wenn es als selbstständiges Programm aufgerufen wird.')


if __name__ =='__main__':
  main()
```

```
Hallo, hier ist das Modul testfile.
Bei mir ist __name__ auf __main__ gesetzt.
Hier könnte z.B.Initialisierungscode für das Programm testfile
stehen, wenn es als selbstständiges Programm aufgerufen wird.
>>> import testfile
Hallo, hier ist das Modul testfile.
Bei mir ist __name__ auf testfile gesetzt.
>>>
```

Why do we write this in a separate function instead of the if condition?

Response:

• Writing this as a subroutine is a bit safer because this way no "global” variables for the test file can be created accidentally

• Also, in case the program was called from outside, you can revisit the code.

Summary

- To summarize: With a.py – file you can do the following:
  - Import as module and then use in an other program
    - all statements (except after def and class) are executed (once)

  - start and use as a script (program):
    - Here you may have initialization code, which you only need if it is executed as a script/main program

- `if name == " main ":`
  `main()`

- So is a switch that detects in which environment the code is executed

Where should I store my own modules so that the interpreter can find them?

When you import a module, for example module_1, the interpreter searches for module_1.py on the so-called module search path in the following order:

1-) in the current directory of the interpreter (where it loaded the calling module)
2-) in PYTHONPATH,
3-) in the default path "PATH
- This is the search path for the standard libraries
- It is installation dependent, here especially the folder for sitepackages.

4-) In the content of .pth files located in the default path.

1-) Search path: In the current directory of the interpreter

• This is practical, especially during program development

• Pay attention: The current working directory of the interpreter is valid - not the one of IDLE.

• However, built-in libraries are stored somewhere else.

```
>>> import os
>>> os.getcwd()
'/Users/alexanderwolodkin/Documents'
>>>
```

2-) PYTHONPATH

- If these environment variables are present and have system-specific file paths, then the Python interpreter looks to see if it can find the module name in these folders.

- After the default installation, PYTHONPATH is not present.

- However,these environment variables have to be set in a comparatively complicated way under Windows (please look up your operating system if you want to know more).

- It is interesting that this path is then taken over into the path (see 3.) when booting.

3-) In the default PATH

•The path variable in the sys module , i.e.sys.path, is a list of strings that specifies the default module search path.
•This search path is system and installation dependent – check it!

• Under Windows, ready-made modules belong in the folder [installation path to Python]\lib\site- packages (depending on the version)

```
>>> import sys
>>> sys.path
['', '/Users/alexanderwolodkin/Documents', '/Library/Frameworks/Python.framework
/Versions/3.9/lib/python39.zip', '/Library/Frameworks/Python.framework/Versions/
3.9/lib/python3.9', '/Library/Frameworks/Python.framework/Versions/3.9/lib/pytho
n3.9/lib-dynload', '/Library/Frameworks/Python.framework/Versions/3.9/lib/python
3.9/site-packages']
```

Frequently used modules from built-in packages

- Module `sys` Accesses to environment components like command line etc.
- Module `os` Tools of the operating system environment: processes, files, shell commands
- Module `re` Pattern Recognition, Regular Expressions
- Module `math` Basic mathematical functionalities
- Module `time` Measuring time

Namespaces

•Already seen in the last lectures and exercises!

•A subroutine (a function in Python) opens (when called) its own namespace for name encapsulation.

•Therefore, each time a function is called, a new local namespace is created.

•This namespace contains the names that are defined here: the (formal) function parameters as well as the names of variables to which values are assigned in the body of the function.

Summary: Structured Programming

• We divide our programs into main program (the calling program, e.g. the interactive interpreter in IDLE), if necessary into modules and into functions.

• If we are "careful”, i.e.restrictive, with the import `statement`, we can largely avoid naming conflicts (same name for different objects written by different programmers, for example).

• The key to this is the qualified name, which uses dot notation to separate the different namespaces: `import <module>` gives access to functions of the module by qualifying: `module.functionname`

Important packages

- Packages we still want to use:
    - NumPy
    - SciPy
    - Matplotlib
- Installation: https://www.scipy.org/install.html
- Documentation: https://docs.scipy.org/doc/
- Try it out afterwards!

numpy package

- Must be imported as

```
import numpy
```

- everyone on the planet uses like this:

```
import numpy as np
```

NumPy Features - Arrays

- The most important feature numpy provides are the so-called multi-dimensional arrays
- These work similar to the lists in Python
- Integration of C/C++ (not considered in detail)
- Fast generation of generic data

NumPy Features - Arrays

- Important differences to lists in Python:
- The entries may only be of one (numeric) data type (No more strings with integers at the same list)
- Access to multidimensional arrays works "as you wish" (i.e. as via indices of a matrix)
- Subsequent change of the number of entries is not possible
- With a concatenation a new object is created!

Examples

```
 9  import numpy as np
10
11  test = np.array([1, 15, 30])
12  print(test, end="\n\n")
13
14  print(type(test), end="\n\n")
15
16  print(test[0],
17        test[1],
18        test[:1],
19        test[:2],
20        test[-1],
21        end="\n\n")
22
23  print(test[0],
24        type(test[0]),
25        test[:1],
26        type(test[:2]),
27        end="\n\n")
```

```
In [8]: runfile('/Users/alexanderwolodkin/Documents/
Python/temp.py', wdir='/Users/alexanderwolodkin/
Documents/Python')
[ 1 15 30]

<class 'numpy.ndarray'>

1 15 [1] [ 1 15] 30

1 <class 'numpy.int64'> [1] <class 'numpy.ndarray'>


In [9]:
```

Example 2

```
 9  import numpy as np
10
11  test = np.array([
12                   [1, 2, 3],
13                   [4, 5, 6],
14                  ],
15                  float)
16  print("Array, Datentyp float:",
17        test, end="\n\n")
18
19  print("Ein bestimmtes Element:",
20        test[0, 1], end="\n\n")
21
22  print("Und was passiert hier?",
23        test[:,2])
```

```
In [12]: runfile('/Users/alexanderwolodkin/Documents/
Python/temp.py', wdir='/Users/alexanderwolodkin/
Documents/Python')
Array, Datentyp float: [[1. 2. 3.]
 [4. 5. 6.]]

Ein bestimmtes Element: 2.0

Und was passiert hier? [3. 6.]

In [13]:
```

Example 2

```
 9  import numpy as np
10
11  test = np.array([
12                   [1, 2, 3],
13                   [4, 5, 6],
14                  ],
15                  float)
16  print("Array, Datentyp float:",
17        test, end="\n\n")
18
19  print("Ein bestimmtes Element:",
20        test[0, 1], end="\n\n")
21
22  print("Und was passiert hier?",
23        test[:,2])
```

```
In [12]: runfile('/Users/alexanderwolodkin/Documents/
Python/temp.py', wdir='/Users/alexanderwolodkin/
Documents/Python')
Array, Datentyp float: [[1. 2. 3.]
 [4. 5. 6.]]

Ein bestimmtes Element: 2.0

Und was passiert hier? [3. 6.]

In [13]:
```

Example 3

```
 9  import numpy as np
10
11  temp = np.array([
12                   [1, 2, 3],
13                   [4, 5, 6],
14                  ],
15                  float)
16  print("Wie lang ist unser temp:",
17        len(temp), end="\n\n")
18
19  print("Wie sieht temp aus",
20        temp, end="\n\n")
21
22  print("Welche Form hat temp?",
23        temp.shape, end="\n\n")
24
25  print("Dann ändern wir das:",
26        temp.reshape(3,2))
```

```
In [25]: runfile('/Users/alexanderwolodkin/Documents/
Python/temp.py', wdir='/Users/alexanderwolodkin/
Documents/Python')
Wie lang ist unser temp: 2

Wie sieht temp aus [[1. 2. 3.]
 [4. 5. 6.]]

Welche Form hat temp? (2, 3)

Dann ändern wir das: [[1. 2.]
 [3. 4.]
 [5. 6.]]

In [26]:
```

Lets try some examples