PHOT 110: Introduction to programming

LECTURE 07: Modules (Ch. 4.9)

Michaël Barbier, Spring semester (2023-2024)

MODULES (CH. 4.9)

USING MODULES

- Import modules with the **`import`** keyword
- Dot-notation to select sub-modules, objects, functions

Example: **`math`** is a **module**

```
1  import math
2  print(type(math))
```

```
<class 'module'>
```

DOT-NOTATION: SELECT SUB-MODULES, OBJECTS, FUNCTIONS

- The `math` module contains **functions** and other **objects**

Contains a variable with a value for π

```
1 math.pi
```

```
3.141592653589793
```

```
1 type(math.pi)
```

```
float
```

Contains function $\sin()$

```
1 type(math.sin)
```

```
builtin_function_or_method
```

DOT-NOTATION: SELECT SUB-MODULES, OBJECTS, FUNCTIONS

- numpy is a module

```
1 import numpy
2 print(type(numpy))
```

```
<class 'module'>
```

- also contains functions, sub-modules, attributes

```
1 print(type(numpy.random))
2 print(type(numpy.random.normal))
3 print(type(numpy.pi))
```

```
<class 'module'>
<class 'builtin_function_or_method'>
<class 'float'>
```

BEWARE: DOT-NOTATION FOR METHODS

- A method is a function bound to an object
- Called on an object: `object.method()`

An example object is a list:

```
1 # list objects: reverse(), sort(), ...
2 my_list = ["apple", 1, True, "five"]
3 print(my_list)
```

```
['apple', 1, True, 'five']
```

Applying the method `reverse()` to change the object:

```
1 my_list.reverse()
2 print(my_list)
```

```
['five', True, 1, 'apple']
```

BEWARE: DOT-NOTATION FOR METHODS

- A method is a function bound to an object
- Called on an object: `object.method()`

Another example object is a string:

```
1  # str objects: upper(), substring() ...
2  my_string = "apple cake"
3  print(my_string)
```

```
apple cake
```

Applying the method `upper()` to obtain the string in all-caps:

```
1  all_caps_string = my_string.upper()
2  print(all_caps_string)
```

```
APPLE CAKE
```

BEWARE: DOT-NOTATION FOR METHODS

- A method is a function bound to an object
- Called on an object: `object.method()`

Another example object is a string:

```
1 # str objects: upper(), substring() ...
2 my_string = "apple cake"
3 print(my_string)
```

```
apple cake
```

The method `replace()` can have arguments:

```
1 changed_string = my_string.replace("apple", "banana")
2 print(changed_string)
```

```
banana cake
```

CREATING MODULES

WHAT IS A MODULE ?

- A **separate** Python script
- Contains variables, **functions**, classes, etc.
- Can be **imported**

Example:

module_print.py

```
1  def print_dollar(text):
2      """
3      Print text surrounded
4      by dollar symbols
5      """
6      print("$" + text + "$")
```

script.py

```
1  # Import the module
2  import module_print as mp
3
4  # Use a function
5  mp.print_dollar("x + 4")
```

```
$x + 4$
```

ANOTHER EXAMPLE OF A MODULE

tranformations.py

```
1  import numpy as np
2
3  def rotate(xys, angle):
4    """ Rotates points around the origin """
5    rot_matrix = np.array(
6      [[np.cos(angle), -np.sin(angle)],
7      [np.sin(angle), np.cos(angle)]])
8    return np.dot(rot_matrix, xys)
9
10 def translate(xys, displacement):
11   """ Translates points with a displacement vector """
12   return xys + displacement
13
14 def scale(xys, scale):
```

USING THE MODULE

script.py

```
1  # import our module
2  import transformations as transfo
3
4  shape = np.array([[1, 2, 1, -1, 0, 1], [0, 1, 2, 2, -3, 0
5  rot_shape = transfo.rotate(shape, 1)
6  plt.plot(shape[0,:], shape[1,:])
7  plt.plot(rot_rot[0,:], rot_shape[1,:])
8  plt.show()
```

WHY USE MODULES ?

- **Re-use** function definitions over multiple scripts
- **Readability**: One very long script is not readable:
  - Break up code in **independent parts**
  - Further **structure your code**
- Structure example, divide into:
  - file input/output
  - calculations
  - plotting
  - AND your main script as a separate file

TESTING THE FUNCTIONS IN YOUR MODULE

- When importing a module, its code is executed
- Function definitions don't have side-effects

But … any code that **prints**, **changes a variable**, **plots**, or outputs something would be executed on **import** !

Solution: using the `__name__` variable

USING THE `__name__` VARIABLE

- `__name__` is always defined
- `__name__` contains:
  - name of the module (file name) if imported
  - `"__main__"` if run as a script

my_module.py

```
1  ... module functions ...
2
3  if __name__ == "__main__":
4    <statements>
```

USING THE `__name__` VARIABLE

- `__name__` is always defined
- `__name__` contains:
  - name of the module (file name) if imported
  - `"__main__"` if run as a script

my_module.py

```
1  def increment(x):
2      return x + 1
3
4  if __name__ == "__main__":
5      # Run the test functions
6      print(f"The value of 3 + 1 = {increment(3)}")
```

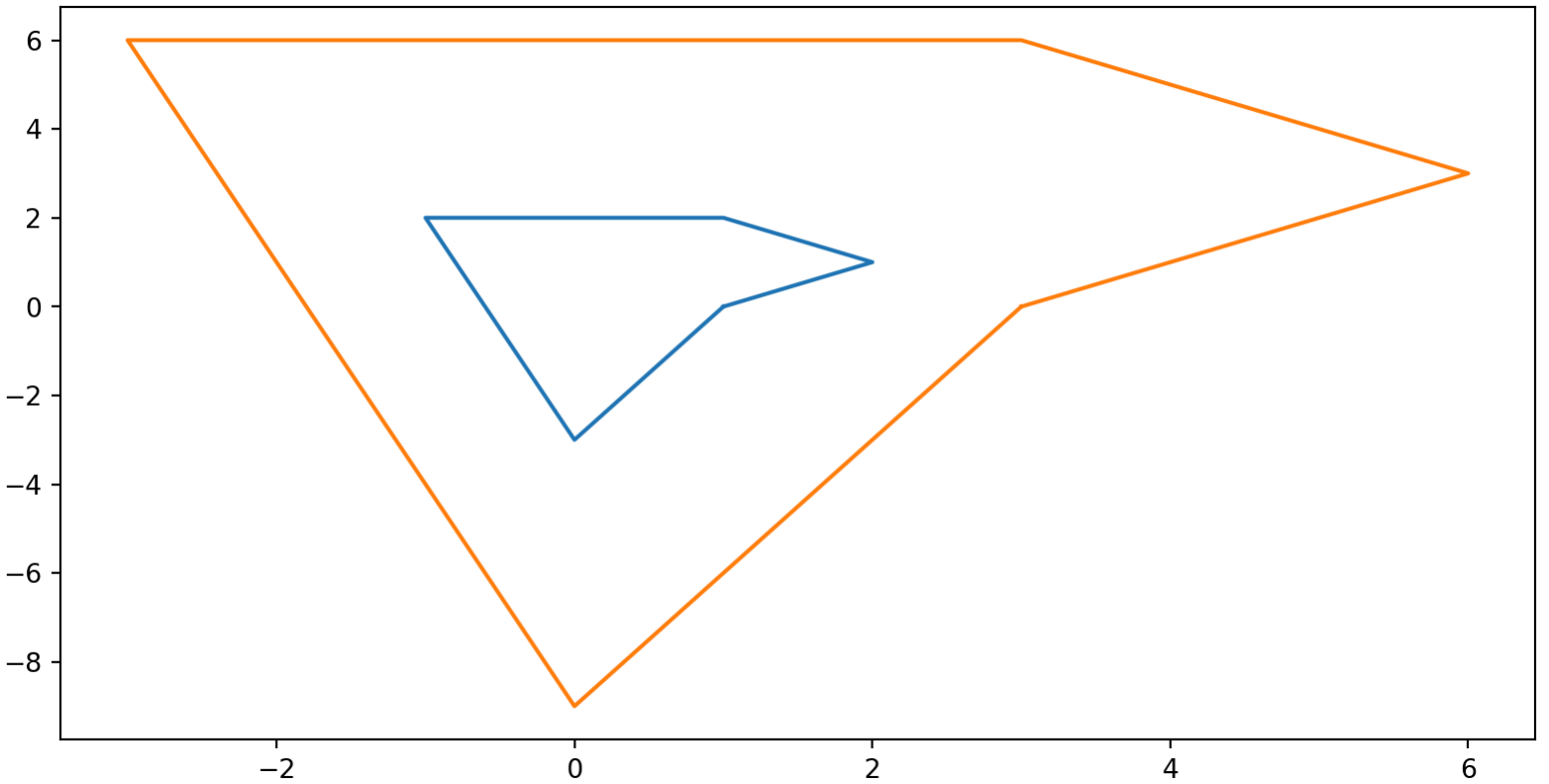
```
The value of 3 + 1 = 4
```

IMPORTING WITHOUT `__name__ == "__main__"`

tranformations.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  def scale(xys, scale):
5    """ scales points by scale """
6    return xys * scale
7
8  # Testing the scale function
9  shape = np.array([[1, 2, 1, -1, 0, 1], [0, 1, 2, 2, -3, 0
10 scaled_shape = scale(shape, 3)
11 plt.plot(shape[0,:], shape[1,:])
12 plt.plot(scaled_shape[0,:], scaled_shape[1,:])
13 plt.show()
```

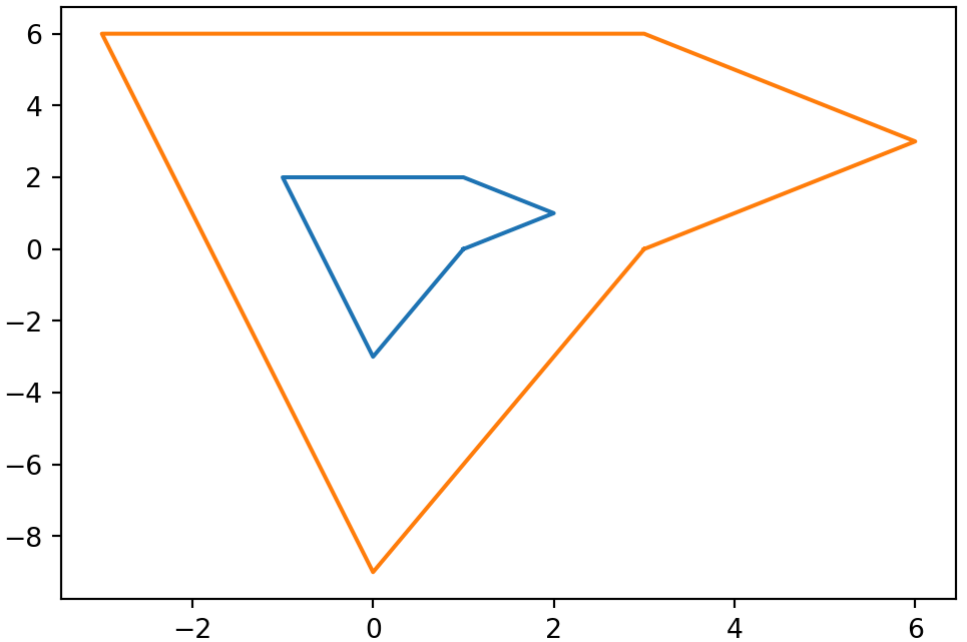
IMPORTING WITHOUT `__name__ == "__main__"`



IMPORTING WITHOUT `__name__ == "__main__"`

main_script

```
1 import transformations
2 print("Not doing anything except printing this !")
```




```
Not doing anything except printing this !
```

AVOIDING SIDE EFFECTS

tranformations.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  def scale(xys, scale):
5    """ scales points by scale """
6    return xys * scale
7
8  # Only execute tests if run as script
9  if __name__ == "__main__":
10   shape = np.array([[1, 2, 1, -1, 0, 1], [0, 1, 2, 2, -3,
11   scaled_shape = scale(shape, 3)
12   plt.plot(shape[0,:], shape[1,:])
13   plt.plot(scaled_shape[0,:], scaled_shape[1,:])
14   plt.show()
```

DOCUMENTING MODULES

transformations.py

```
""" Geometrical transformations on points in 2D.

Provides functionality for rotation, translation,
or scaling of (x, y)-coordinates. Coordinates
should be given as (2 x N) Numpy arrays where N
is the number of points.

Typical usage example:
  points = np.array([[1, 4, 6],[2, 5, 3]])
  scaled_points = scale(points, 3)
"""
import numpy as np

# ... code of the module ...
```

CREATING AN EXECUTABLE OF YOUR SCRIPT

HOW TO CREATE AN EXECUTABLE ?

- Run a script as an **executable**
  - without Python interpreter
  - Directly runnable (by “double clicking”)
- Two good options (as of 2024)
  - **Pyinstaller**
  - **Nuitka** (up to Python version 3.11)
- These compilers **incorporate the Python interpreter** within the executable $\longrightarrow$ the resulting **executables are large**.

USING PYINSTALLER

Pyinstaller works for (our) **Python version 3.12**.

Follow the instructions below to create an executable:

1. Make a new project for your script
2. Copy your script in the new project (or write it)
3. Install the dependencies (required packages)
4. Install the pyinstaller package
5. Execute below line (change the script file name to your own file name) in the **terminal**:

```
pyinstaller --onefile your_script.py
```

PYTHON PACKAGES VS. MODULES

WHAT IS A PYTHON PACKAGE ?

- **Collection of modules**
- You can **import** the top-module
- Prepared for **distribution**
  - **Sharing** your application
  - Installing
- Can be shared on the Python Package Index: PyPI

CREATING A PYTHON PACKAGE ?

- **setup.py** as advised in the book is the old way, new way:
- Write a pyproject.toml file to package a Python project
- Easiest way: use Package and Dependency Managers:
  - Poetry: works in PyCharm out-of-the-box, see the PyCharm JetBrains page
  - Rye: has a plugin for PyCharm
  - PDM: works not well together with PyCharm yet

EXAMPLE .TOML FILE

pyproject.toml

```toml
[build-system]
requires = ["setuptools", "setuptools-scm"]
build-backend = "setuptools.build_meta"

[project]
name = "my_package"
authors = [{name = "Michael Barbier"},]
description = "My package description"
readme = "README.md"
license = {text = "BSD-3-Clause"}
dependencies = ["numpy",]

[project.scripts]
my-script = "my_package.module:function"
```

Lecture 07: Modules, script documentation, and executables (Ch. 4.9)