Prof. Dr. A-S. Smith
am Lehrstuhl für Theoretische Physik I
Department für Physik
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Advanced Python for Research Projects

Exercise sheet 4: Modules, Documentation and testing

Problem 4.1 *Using files and directories as modules*

We want to work on packaging our code into a python module that can then be installed via pip. For this purpose, we will need to set up the code, then add some tests and documentation and finally set up the packaging. Here, we will first start with setting up our code in an appropriate manner. For this exercise, we will try and package our implementations of `map`, `reduce` and `map_reduce` (the sequential and the parallel versions) as well as our re-implementations of scipy and numpy functions into a module.

a) Set up your project structure. We will first need an empty folder as a base for our project. Then add a folder `fauap_advanced_functions` for the source code, i.e. the python modules, a folder `tests` for our tests, a folder `docs` for putting documentation in, and a file `Readme.md` to put the most relevant general information into.

b) We now want to pack all our prior functions into the `fauap_advanced_functions` package. To do this, put all functions related to `map` and `reduce` in their sequential forms into a module (i.e. a file) `map_reduce.py` in the `fauap_advanced_functions` folder. Then add their parallel versions into a file `map_reduce_parallel.py` in the `fauap_advanced_functions` folder. Finally add your custom reimplementation of numpy/scipy functions into a file `sci_num.py` in the `fauap_advanced_functions` folder. To turn the folder `fauap_advanced_functions` into a package, you also need to add a file `__init__.py` to the `fauap_advanced_functions` folder. This init-script is called/executed first, whenever your import your package. Only if a folder contains an init-script, does the python interpreter consider it a package that can be imported. Within this init-script, you should import all of the functions you want to be available when someone writes `import fauap_advanced_functions` from your other files as well as all packages necessary for your other modules. For now, let us leave the file mostly empty, but add a function `say_hello`, which prints `Hello:` followed by the value of `__name__`. As all of the code in `__init__.py` is executed when you import a package, you could add a call `say_hello()` to the code and it would print `Hello <package_name>` whenever you import the package. This also means that you should be careful not to unnecessarily execute code here as it will be executed whenever the package is loaded.

c) Your files may need to import functions from other files in this directory. You can prefix the module name with a dot (e.g. `.map_reduce`) to import a function from a module within the same directory. Make sure that your imports of all files in the directory are correctly pointing to the local files.

d) Add a module `arithmetics.py` in which you add a custom function `verbose_sqrt`, which effectively just calculates the square root of its argument. However, it will first check whether the argument is negative. If it is negative, output a warning message via a logger that you initialize within that file with the `__name__` label. Also, if the argument is negative, raise/throw a ValueError. Document the function to a reasonable degree.

e) Add documentation to the files in the package directory as the lecture has described for modules. The functions themselves should already have been documented in prior exercises.

f) You can make a package executable via `python -m <package_name>` by adding a `__main__.py` file to it. Use this to add an interface to your module, where the user can pass a floating point number to the `__main__.py` script and it will print the result of your `verbose_sqrt` function. The file should be documented and guarded, i.e. check for `__name__ == "__main__"` before executing the described code. Make sure that you correctly set up logging in the __main__ script and that you correctly import the `verbose_sqrt` function.

g) Create a main script `main.py` in the root directory of your project and import the module `fauap_advanced_functions` (possibly with a leading dot to indicate that it is a local import). What do you notice when you execute the script even without further code if you have added a call to `say_hello` in your init-script?

h) Use the function `say_hello` imported from our package to output the module name. Which module name does it print compared to a `print(__name__)` statement in `main.py`?

i) Consider adding loggers to your package's other modules using the __name__ variable to determine the module name and add them to your modules' functions to log infos, warnings and errors where appropriate. E.g. log a warning if the input to the derivative or integration function do not have the minimum reasonable amount of data points for calculation and add debug outputs of the number of created worker processes to the parallel map and reduce implementations.

Problem 4.2 *Building tests for our modules*

a) Add tests for your package's modules' functions to check that they work. You should have written these tests in previous exercises, now move them to files in the `test` folder of our package and make sure they properly import the functions they are testing from the package. You may have to use relative imports as long as our package has not been installed. You can use two leading dots to go up one directory, e.g. `import ..fauap_advanced_functions.map_reduce` should import the map_reduce module of our package from the `test` directory.

b) Run the tests and verify correct operation. Fix any errors you notice and add comments explaining the intention of your tests where they had not yet been provided.

Problem 4.3 *Generating documentation for your module*

A good package has some documentation to go with it. Of course we have already written documentation within the code files, but for a new user, it is preferrable to have a version of the documentation that can be read either in the browser (as html) or as a pdf. Here, we want to generate such a documentation from our code files.

a) Use pdoc3 (see `https://pdoc3.github.io/pdoc/` and `https://pypi.org/project/pdoc3/`) to generate a documentation page for your entire project as html. Output the documentation into the directory `docs`. The documentation page `https://pdoc3.github.io/pdoc/doc/pdoc/` has examples for generating the html source

b) Also familiarize yourself with generating documentation for a single module/file.

c) Generate documentation in PDF format and put it into the `docs`-directory as well. You should not specify the `--html` format specifier, but an appropriate other format instead. Look at the documentation of pdoc if necessary to figure out the correct settings

Problem 4.4 *Making our module installable with pip*

We want to use `pip install -e .` to do a quick install of our local module `fauap_advanced_functions` from the current directory so that any python script on our user can access it. In this exercise, we do not want to fully build, bundle and upload the package to pypi using test-pypi as described in the lecture. You should, however, consider doing a full test run of publishing your package to test-pypi following the steps as described in the lecture slides. Within all steps of this exercise, make sure you appropriately replace placeholders with your appropriate data.

a) Add content to the Readme.md file explaining the function of your module, where to find and run tests as well as where to find the documentation. Also add your own contact information and an appropriate license. You can look up a list of open source licenses online, any of them will do, however it may benefit you figuring out the slight differences between options.

b) Create a script `setup.py` in your project's root directory. This is required for an editable installation that we are trying to perform here. It is a legacy feature but due to backwards compatibility always a good idea to provide it. The file should contain the following:

```
import setuptools
if __name__ == "__main__":
    setuptools.setup(
            name='fauap_advanced_functions',
            # Use a semantic version number.
            # If your upload to pypi, no two uploads may have the same version
            version='0.1.0',
            description='<provide␣a␣reasonable␣description>',
            url='<add␣url␣to␣the␣git␣repository␣once␣you␣have␣it>',
            author='<add␣your␣name>',
            author_email='<add␣your␣email␣address>',
            license='BSD␣2-clause', # This may be any license you choose
            packages=['fauap_advanced_functions'],
            # This needs all packages your code depends on
            install_requires=['numpy', 'scipy'],

            # You can put whichever classifiers you want
            # This is just a suggestion
            classifiers=[
                'Development␣Status␣::␣1␣-␣Planning',
                'Intended␣Audience␣::␣Science/Research',
                'Operating␣System␣::␣POSIX␣::␣Linux',
                'Programming␣Language␣::␣Python␣::␣3',
            ],
    )
```

c) Create a `pyproject.toml` file in the root directory of your project. You need to specify the build backend first. Here, we use setuptools as we did in `setup.py`.

```
[build-system]
     requires = ["setuptools>=61.0.0", "wheel"]
     build-backend = "setuptools.build_meta"
```

Afterwards, you need to add project specification and meta-data similar to our `setup.py` script. Make sure your information matches the `setup.py` script. Also make sure you have a fitting python version requirement (end of file). When in doubt, use the python version `python --version` tells you on your development machine.

```
[project]
     name = "fauap_advanced_functions"
     version = "0.1.0"
     description = "<provide␣a␣reasonable␣description>"
```

```
    readme = "README.md"
    authors = [{ name = "<add␣your␣name>",
    email = "<add␣your␣email␣address>" }]
    license = { file = "LICENSE" }

    classifiers = [
            'Development Status :: 1 - Planning',
            'Intended Audience :: Science/Research',
            'Operating System :: POSIX :: Linux',
            'Programming Language :: Python :: 3',
    ]
    keywords = ["map", "reduce", "scientific␣processing"]

    dependencies = [ "numpy", "scipy", ]
requires-python = ">=3.5"
```

d) In our project manifest `pyproject.toml` we referred to a `LICENSE` file. Create that file in the root of your directory and put the appropriate license text for the license you have chosen in there. If you are struggling, use `https://choosealicense.com/`, which for example suggests `https://choosealicense.com/licenses/mit/` for simple and permissive code licensing.

e) Install the current package for the current user using `pip install -e .`; Afterwards, you can test that you can now import from the module `fauap_advanced_functions` with no relative leading dots.

f) Remove the module from the user by running `pip uninstall fauap_advanced_functions` to clean up the installation.