Python tutorial 1

Virtual environments

A virtual environment (VE) is a space where you can install Python libraries separate from your installation of Python. The benefit of using a VE is it allows you to download specific libraries for different projects you are working on. This is particularly useful since it prevents you from downloading multiple versions of the same library in a single place (in the event that this happens, Python would be unable to distinguish between the different versions). Moreover, there is no limitation on the number of VEs you can have (including for a single project), making them a powerful tool for any Python developer.

To create a VE, we can execute the following line of code in the command line (make sure you are in the desired project directory):

In [ ]:
```
python -m venv VE_name # Change the name of VE_name to
                       # the name you want for the VE
```

We can then activate the VE we have created by running the following:

In [ ]:
```
VE_name\Scripts\activate
```

When this VE is activated we can proceed to install the relevant libraries for the Python project as follows (the selection of libraries is just an example):

In [ ]:
```
python -m pip install numpy pandas matplotlib scipy statsmodels seaborn plotly
```

Once this has been achieved, we can proceed to write a code file that has access to the installed libraries:

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy as sc
import statsmodels.api as sm
import seaborn as sns
import plotly.express as px
```

Poetry

A useful package manager for VEs is Poetry. Using Poetry means you do not need to manually create your VEs or pip install your Python libraries. Moreover, it is a great tool to manage you project dependencies, as it works to find a combination of installations that work well together (these are stored in a so-called lock file). Furthermore, it provides an intuitive alternative to building Python packages and distributing them.

The installation of Poetry is recommended through the use of the Poetry installer (as opposed to pip), since this isolates Poetry from you other installations. The installation can be executed in the command line as follows:

In [ ]:
```
# Windows device:
(Invoke-WebRequest -Uri https://install.python-poetry.org -UseBasicParsing).Content | py -

# Apple device:
curl -sSL https://install.python-poetry.org | python3 - # Mac
```

Alternatively, this can be done using pip:

In [ ]:
```
python -m pip install --user poetry
```

To test that the installation was successful, we can check the version of the Poetry we have installed:

In [ ]:
```
poetry --version
```

Depending on how Poetry was installed, we can update the version in the following way:

In [ ]:
```
# Poetry installer method:
poetry self update

# pip install method:
python -m pip install --upgrade poetry
```

To create a new project with Poetry, we can simply write in the command line:

In [ ]:
```
poetry new demo
```

This will create a directory called demo with the following file structure:

In [ ]:
```
demo
├── demo
│   └── __init__.py
├── pyproject.toml
├── README.rst
└── tests
    ├── __init__.py
    └── test_demo.py
```

The demo directory is where the project is. Inside, we have a demo folder, with an **init**.py file for convenience. In addition, pyproject.toml stores the metadata for the project, the README.rst file should provide a brief description of the project, and the tests folder stores the unit tests for the Python project.

The TOML file is a simple file type that is easy to read and write. In our case, the pyproject.toml file contains the following:

In [ ]:
```
[tool.poetry]

# Metadata for the project:
name = "demo"
```

```
version = "0.1.0"
description = ""
authors = ["Your name <your@e-mail.address>"]

# Project dependencies (required to run software):
[tool.poetry.dependencies]
python = "^3.10"

# Dependencies belonging to project developers
# (needed for development and testing):
[tool.poetry.dev-dependencies]
pytest = "^5.2"

# Settings for the build system:
[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"
```

To add and install packages, we can either edit the pyproject.toml file, or use the command poetry:

In [ ]:
```
poetry add <package name> # e.g., <package>=requests
```

The second approach additionally looks for a suitable version that does not conflict with other dependencies, directly installs in the project VE and creates/updates the lock file called poetry.lock. Furthermore, this command creates a VE outside of the project directory if this is the first time we have used the project. If we want to move the VE inside the project directory (including for future projects), we can execute:

In [ ]:
```
poetry config virtualenvs.in-project true
```

All dependencies of the requests package (all versions) will be installed and can be found in the lock file, however, when inspecting the pyproject.toml, only the requested package will be visible. The purpose of the lock file is that all versions of the requests package are the same when recreating the VE.

Despite developer dependencies not being needed to run the project, we can add them via:

In [ ]:
```
poetry add --dev <package name>
```

We can remove packages/libraries from a project with

In [ ]:
```
poetry remove <name>

poetry remove --dev <package name> # removes developer dependencies
```

If we already have an existing project that has used Poetry, we can install all dependencies at onces using:

In [ ]:
```
poetry install # installs all packages from the lock file (if existing),
               # otherwise, installs packages in a newly created lock file
```

We can run a program in the project VE as well:

In [ ]:
```
poetry run python run.py

poetry run pytest # to run pytest as a developer dependency
```

We can also activate the VE in a separate shell if we execute:

In [ ]:
```
poetry shell
```

Even though it is useful to have packages stored in a lock file so that our projects always work as expected (due to packages remaining the same version as at the inception of the project), there might still be instances where we want to update packages for project use. To do this, we can simply use the command:

In [ ]:
```
poetry update # updates dependencies in the VE and the poetry.lock file

poetry update package_name # updates a specific package

poetry add package_name@latest # updates the package to the latest version
                               # (or replace latest with version number)
```

The command:

In [ ]:
```
poetry build
```

creates two files in a new directory called dist. The first file is a compiled package, while the second file contains the source code for the package. Following this, we can publish with an access token with the sequence of commands:

In [ ]:
```
poetry config pypi-token.pypi <token> # creates access token instead
                                      # of username and password
poetry publish # publish project
```

Some common Poetry commands include:

In [ ]:
```
poetry --version # shows current version of Poetry

poetry new <name> # creates new Poetry project

poetry init # convert existing project to new project

poetry add <package> # add package

poetry remove <package> # remove package

poetry show # list packages installed

poetry export -f <filename> # export dependency list

poetry install # install all dependecies of current Poetry project

poetry run <command> # run command in project VE

poetry shell # activate VE in new shell
```

Sometimes, it is useful to export project dependencies to a .txt file when collaborating with those who do not use Poetry:

In [ ]:
```
poetry export -f requirements.txt > requirements.txt
```