Practical Application — Project Initialization Script

Creating a well-organized structure for a Python project is crucial for making the code manageable and scalable. Imagine you frequently start new projects with a similar structure, like the one below. These repetitive actions can be automated with a Python script. Read up about a step-by-step process of making a structured Python project. The acting code example is the script initilalize_project.py

Set Up the Project Root

It is desirable that the project directory name of your project should be descriptive and meaningful, so it can remind you what this project is about.

Example:

```
$ mkdir prj_mkt_data
```

Create a Virtual Environment

Again, it is desirable to have a virtual environment to keep your project dependencies isolated from other projects, preventing version conflicts.

Create the Virtual Environment For Windows:

```
python -m venv venv
venv\Scripts\activate
```

Create the Virtual Environment for Linux (macOS)

```
python3 -m venv venv
source venv/bin/activate
```

Project Initialization Script

```
my_new_project/
|— src/
|   |— main.py
|   └── utils.py
|— tests/
|   |— test_main.py
|   └── test_utils.py
└── docs/
 |   |— readme.md
 |  └── changelog.md
 |-------requirements.txt
```

Script to Create the Project Structure

```
import os
from pathlib import Path

# Define the directory structure and file names
project_structure = {
   "src": ["main.py", "utils.py"],
   "tests": ["test_main.py", "test_utils.py"],
   "docs": ["readme.md", "changelog.md"]
}

# Function to create directories and files
def create_project_structure(base_path: str, structure” dict) -> None:
   base_path = Path(base_path)

   for folder, files in structure.items():
      folder_path = base_path / folder
      try:
         folder_path.mkdir(parents=True, exist_ok=True)
         print(f"Directory {folder_path} created successfully.")
      except Exception as e:
         print(f"Error creating directory {folder_path}: {e}")
         continue

      for file in files:
         file_path = folder_path / file
         try:
            file_path.touch(exist_ok=True)
            print(f"File {file_path} created successfully.")
         except Exception as e:
            print(f"Error creating file {file_path}: {e}")

# specify the base path for the new project
base_path = "my_new_project"

# create the new project structure
create_project_structure(base_path, project_structure)
```

In this example we define the project structure in a dictionary with subdirectories as keys and filenames as values. The create_project_structure() function takes a base path and the structure dictionary, creating the necessary directories and files within the specified base path. The script first ensures the base project directory exists. It then iterates through each folder and file, creating them as specified. Error handling is included for managing any issues that arise during directory or file creation.

Running the Script

Save the script as init_project.py and run it from the command line:

```
python init_project.py
```

This will create the my_new_project directory with specified subdirectories and files.

Some additional elements are very useful if not necessary:

- config.json or config.toml;
- __init__.py;
- requirement.txt;
- .gitignore;
- setup.py;

Configuration Files

Configuration files can help automate testing, linting, and deployment, not to mention improve performance. Python has many ways and techniques to create

configuration files and objects and all have their merits. *.toml is a convenient format because it is also used in PyTest framework, so it can be more usable across the project.
The simplest way is to use JSON:

```
{
    "api": {
        "url": "https://examples.com/v2/all"
    },
    "data": {
        "dir": "/data/",
        "file": "in-data.csv"
    },
    "results": {
        "dir": "/Users/Iamuser/Desktop/results",
        "file": "results.json"
    }
}
```

__init__.py file

The __init__.py file is used to initialize a Python package. Place it in each directory you want to treat as a package. This file can also define what gets imported when the package is imported.

Example:

```
 # my_new_package/  init  .py
from .module1 import foo
from .module2 import bar
```

Now users can use direct import:

```
from my_new_package import foo
```

.gitignore

Add a .gitignore file to specify files and directories that Git should ignore:

```
venv/

  pycache /

*.pyc
```

requirements.txt

For efficient dependency management, list packages in requirements.txt:

```
pip freeze > requirements.txt
```

Create a setup.py for efficient packaging

Although it is optional but if you want to distribute your project as a package, add a setup.py file, which tells Python how to build and install the package. One possible way of making the setup file is using setuptools:

```
from setuptools import setup, find_packages

setup(
        name="my_new_project",
        version="0.1",
        packages=find_packages(),
        install_requires=[ "some_package>=1.0.0",]
)
```

We can use pip to install the package locally:

```
pip install -e
```

This enables importing my_new_package without specifying the full path.

Do not forget to document the project by creating the **README.md** file. READMEs can contain anything however the usual minimalistic TOC includes overview, installation instructions, and usage examples. And if necessary, supplement it with ReadThis.md, MustRead.pdf and others.