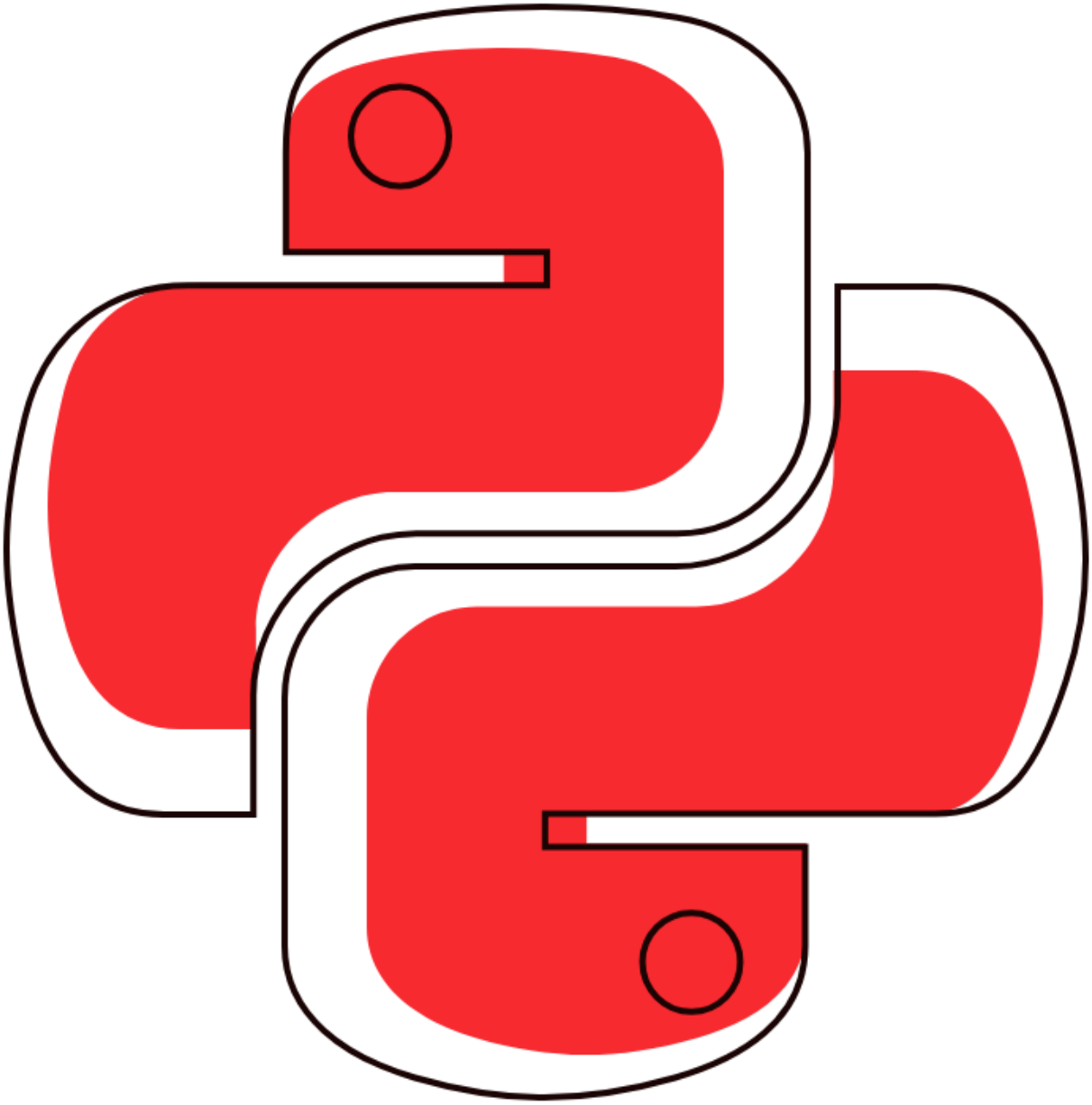
PYTHON DEVKIT

ARIHARASUDHAN



Contents

VIRTUALENV

A Python environment refers to an isolated and self-contained space where Python and its associated libraries and packages can be installed and executed. The purpose of creating separate environments is to manage dependencies, avoid conflicts between different projects. Python environments are particularly crucial when working on multiple projects or when dealing with projects that have specific version requirements for libraries. There are two types of environments in Python. The global environment refers to the system-wide installation of Python and its packages. It is the default environment when we install Python on our machine.

However, using the global environment for all projects can lead to dependency conflicts and version mismatches. A virtual environment is a self-contained directory that contains a specific Python interpreter along with its own set of libraries and packages. It allows to create an isolated environment for each project, ensuring that dependencies are independent of each other.

Tool to Create A Venv

The most common tool for creating virtual environments in Python is virtualenv, though Python 3 also includes the built-in venv module.

Creating a virtual env

We can use a tool like virtualenv or venv to create a new virtual environment in a project directory.

#CREATE

> virtualenv myenv (or)

> python3 -m venv myenv

Activating a virtual env

Activating the virtual environment simply modifies the shell's PATH to use the isolated Python interpreter and associated tools.

#ACTIVATE In Windows,

> myenv\Scripts\activate

#ACTIVATE In Linux,

> source myenv/bin/activate

Installing Packages

We can use pip within the virtual environment to install project specific dependencies.

These dependencies are then isolated from the global environment.

#INSTALL PACKAGES

> pip install package_name

Deactivatig Venv

When the project is finished or we want to switch to a different project, we can deactivate the virtual environment.

#DEACTIVATE

> deactivate

PIP

PIP is a package installer for Python. It is used to install and manage Python packages. PIP can install packages from the Python Package Index (PyPI) and other sources.

A Package Manager

A package manager or installer is a tool that automates the process of installing, upgrading, configuring, and removing software packages. PIP is a popular package manager in Python Programming Language.

Software Repository

A software repository is a storage location where software packages are stored and maintained for distribution.

Examples include PyPI for Python, Maven for Java, and npm for JavaScript.

Python Package Index

PyPI is the official Python package repository. It hosts a vast collection of Python packages that can be easily installed using pip.

Creating Own PyPI Package

To create a PyPI package, we need to structure our project correctly and include a setup.py file with metadata.

Below are the steps to create a simple PyPI-compatible package.

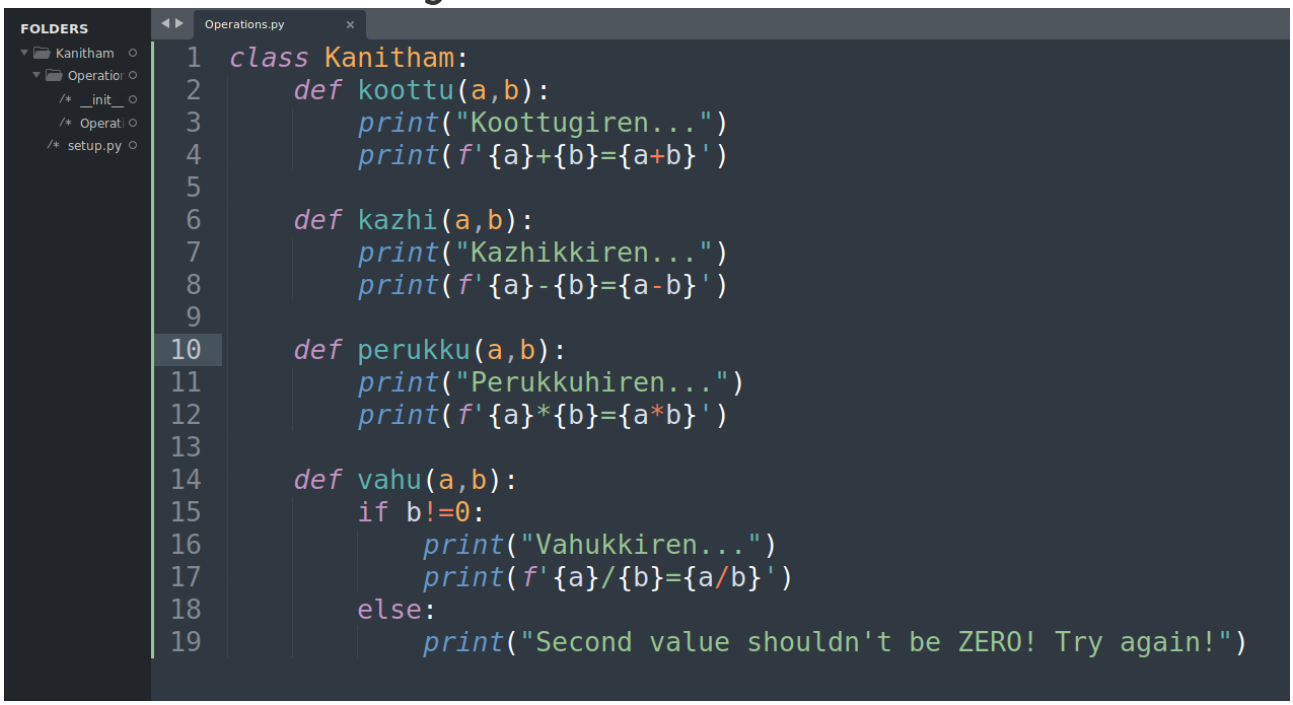
1. Create a directory for your package. The tree structure would be like the following:

```
ari-18308@ari-18308:~/Desktop/Kanitham$ tree
.
├── Operations
│   ├── __init__.py
│   └── Operations.py
└── setup.py

1 directory, 3 files
ari-18308@ari-18308:~/Desktop/Kanitham$
```

Here, Operations is the name of the package. It holds the Operations.py file. __init__.py is to make the folder a python package.

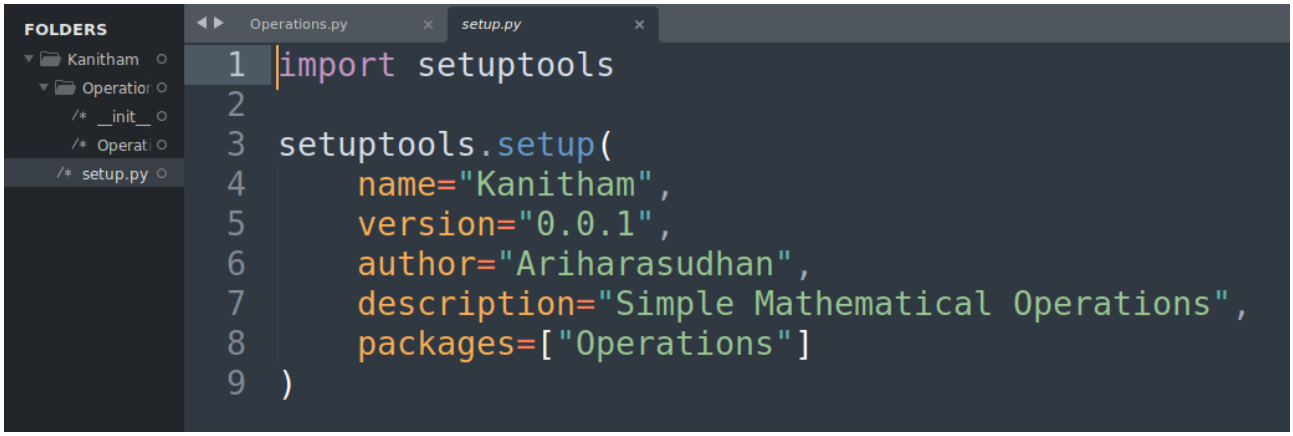
2. Play with Your Code

It's your package. Write whatever you want.



3. Set The Code Up

setup.py is a configuration file.



4. Locally Install

Now that, your package is ready. Let's install it locally.

```
ari-18308@ari-18308:~/Desktop/Kanitham$ pip install .
Processing /home/ari-18308/Desktop/Kanitham
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: Kanitham
  Building wheel for Kanitham (setup.py) ... done
  Created wheel for Kanitham: filename=Kanitham-0.0.1-py3-none-any.whl si
ze=1584 sha256=f0f7ac41a521d38feb411dcfefc9a39459a49d43d95e381faa52222b70
8164d6
  Stored in directory: /tmp/pip-ephem-wheel-cache-sxounm8y/wheels/63/40/2
4/1ccc903fb6e4d4b03e559f34f00c656d95ff716f72d926159f
Successfully built Kanitham
DEPRECATION: pandas 0.23.4 has a non-standard dependency specifier pytz>=
2011k. pip 24.0 will enforce this behaviour change. A possible replacemen
t is to upgrade to a newer version of pandas or contact the author to sug
gest that they release a version with a conforming dependency specifiers.
 Discussion can be found at https://github.com/pypa/pip/issues/12063
Installing collected packages: Kanitham
```

Once installed, check out whether it works.

```
ari-18308@ari-18308:~/Desktop/Kanitham$ python
Python 3.7.0 (default, Jun 28 2018, 13:15:42)
[GCC 7.2.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from Operations.Operations import Kanitham as ka
>>> ka.koottu(2,3)
Koottugiren...
2+3=5
>>> ka.kazhi(4,3)
Kazhikkiren...
4-3=1
>>> ka.perukku(4,2)
Perukkuhiren...
4*2=8
>>> ka.vahu(3,0)
Second value shouldn't be ZERO! Try again!
```

5. Time to distribute

Once we have all the needed files, use the following command to install the latest version of the setuptools package (we used in the setup.py file).

```
ari-18308@ari-18308:~/Desktop/Kanitham$ python -m pip install --user --upgrade setuptools
Requirement already satisfied: setuptools in /home/ari-18308/anaconda3/lib/python3.7/site-packages (40.2.0)
Collecting setuptools
  Downloading setuptools-68.0.0-py3-none-any.whl.metadata (6.4 kB)
Downloading setuptools-68.0.0-py3-none-any.whl (804 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 804.0/804.0 kB 2.4 MB/s eta 0:00:00
DEPRECATION: pandas 0.23.4 has a non-standard dependency specifier pytz>=2011k. pip 24.0 will enforce this behavio
ur change. A possible replacement is to upgrade to a newer version of pandas or contact the author to suggest that
 they release a version with a conforming dependency specifiers. Discussion can be found at https://github.com/pyp
a/pip/issues/12063
Installing collected packages: setuptools
Successfully installed setuptools-68.0.0

[notice] A new release of pip is available: 23.3.1 -> 23.3.2
[notice] To update, run: pip install --upgrade pip
ari-18308@ari-18308:~/Desktop/Kanitham$
```

Make sure we are at the same directory where setup.py is located and run this command.

```
ari-18308@ari-18308:~/Desktop/Kanitham$ python setup.py sdist
running sdist
running egg_info
writing Kanitham.egg-info/PKG-INFO
writing dependency_links to Kanitham.egg-info/dependency_links.txt
writing top-level names to Kanitham.egg-info/top_level.txt
reading manifest file 'Kanitham.egg-info/SOURCES.txt'
writing manifest file 'Kanitham.egg-info/SOURCES.txt'
warning: sdist: standard file not found: should have one of README, README.rst, README.txt, README.md

running check
creating Kanitham-0.0.1
creating Kanitham-0.0.1/Kanitham.egg-info
creating Kanitham-0.0.1/Operations
copying files to Kanitham-0.0.1...
copying setup.py -> Kanitham-0.0.1
copying Kanitham.egg-info/PKG-INFO -> Kanitham-0.0.1/Kanitham.egg-info
copying Kanitham.egg-info/SOURCES.txt -> Kanitham-0.0.1/Kanitham.egg-info
copying Kanitham.egg-info/dependency_links.txt -> Kanitham-0.0.1/Kanitham.egg-info
copying Kanitham.egg-info/top_level.txt -> Kanitham-0.0.1/Kanitham.egg-info
copying Operations/Operations.py -> Kanitham-0.0.1/Operations
copying Operations/__init__.py -> Kanitham-0.0.1/Operations
Writing Kanitham-0.0.1/setup.cfg
creating dist
Creating tar archive
removing 'Kanitham-0.0.1' (and everything under it)
ari-18308@ari-18308:~/Desktop/Kanitham$
```

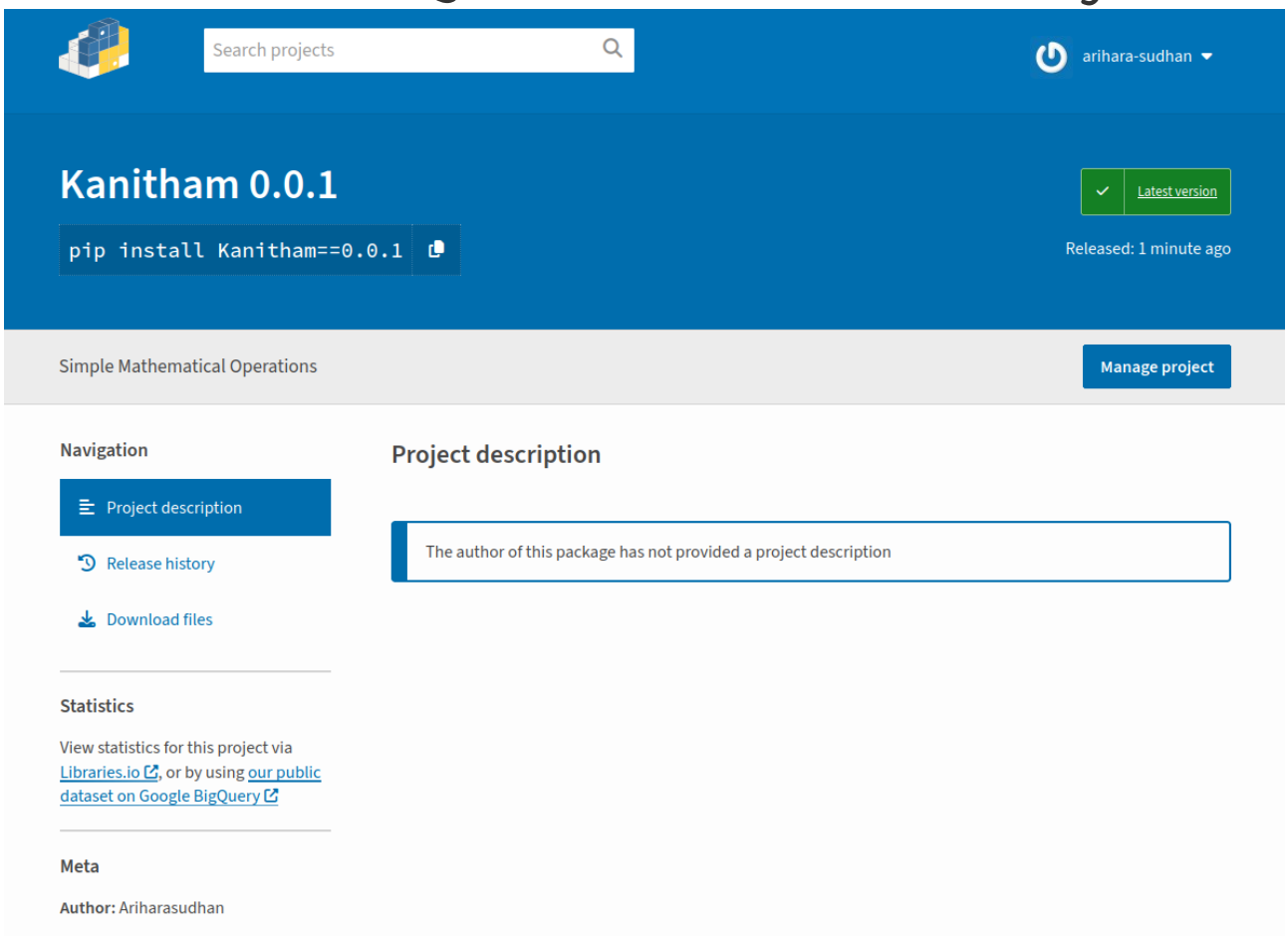
You will see a new folder dist containing the tar.gz file that provides metadata and the essential source files needed.

```
ari-18308@ari-18308:~/Desktop/Kanitham$ ls
build  dist  Kanitham.egg-info  Operations  setup.py
ari-18308@ari-18308:~/Desktop/Kanitham$ tree
.
├── build
│   ├── bdist.linux-x86_64
│   └── lib
│       └── Operations
│           ├── __init__.py
│           └── Operations.py
├── dist
│   └── Kanitham-0.0.1.tar.gz
├── Kanitham.egg-info
│   ├── dependency_links.txt
│   ├── PKG-INFO
│   ├── SOURCES.txt
│   └── top_level.txt
├── Operations
│   ├── __init__.py
│   └── Operations.py
└── setup.py

7 directories, 10 files
ari-18308@ari-18308:~/Desktop/Kanitham$
```

Make sure you have an account in PyPI. Use twine to distribute your package to the community.

```
ari-18308@ari-18308:~/Desktop/Kanitham$ twine upload dist/*
Uploading distributions to https://upload.pypi.org/legacy/
Enter your username: arihara-sudhan
Enter your password:
Uploading Kanitham-0.0.1.tar.gz
100%|████████████████████████████████████████| 3.33k/3.33k [00:02<00:00, 1.26kB/s]

View at:
https://pypi.org/project/Kanitham/0.0.1/
ari-18308@ari-18308:~/Desktop/Kanitham$
```

Our package is there in PyPI.



We can install it and use it in our code.

6. Install and Use

First uninstall the locally installed package. Then, install from the cloud and check it out.

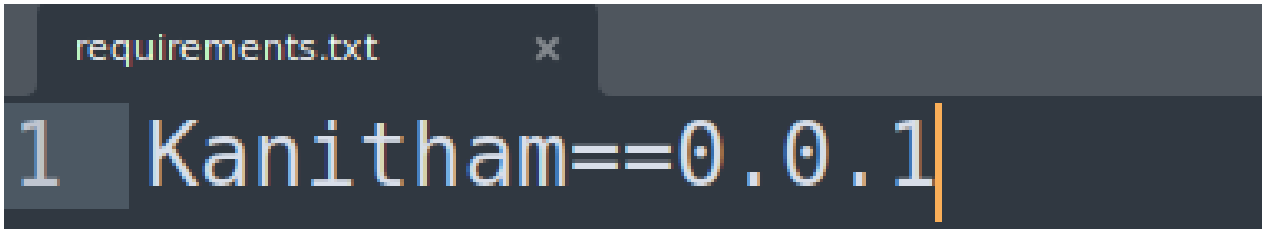
```
ari-18308@ari-18308:~/Desktop/Kanitham$ pip uninstall Kanitham==0.0.1
Found existing installation: Kanitham 0.0.1
Uninstalling Kanitham-0.0.1:
  Would remove:
    /home/ari-18308/anaconda3/lib/python3.7/site-packages/Kanitham-0.0.1.dist-info/*
    /home/ari-18308/anaconda3/lib/python3.7/site-packages/Operations/*
Proceed (Y/n)? Y
  Successfully uninstalled Kanitham-0.0.1
ari-18308@ari-18308:~/Desktop/Kanitham$ pip install Kanitham==0.0.1
Collecting Kanitham==0.0.1
  Downloading Kanitham-0.0.1.tar.gz (858 bytes)
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: Kanitham
  Building wheel for Kanitham (setup.py) ... done
  Created wheel for Kanitham: filename=Kanitham-0.0.1-py3-none-any.whl size=1543 sha256=
bdd67958e0ab9417c3d600ebda452378949350
  Stored in directory: /home/ari-18308/.cache/pip/wheels/b3/f3/d5/3d9e22b1222a250c47a25
9d
Successfully built Kanitham
```

```
>>> from Operations.Operations import Kanitham as kan
>>> kan.koottu(2,3)
Koottugiren...
2+3=5
>>> kan.kazhi(22,3)
Kazhikkiren...
22-3=19
>>>
```

That's it!

Installing PyPI Package

We can manually create a requirements.txt, a file that contains a list of packages or libraries needed to work on a project and that can all be installed using the the file.



We can also create it using pip freeze. I am in a virtual environment where I've installed some packages. Let's generate one requirements file using pip freeze and check it out.

```
(myenv) ari-18308@ari-18308:~/Desktop/myenv$ pip freeze > requirements.txt
You are using pip version 10.0.1, however version 23.3.2 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
(myenv) ari-18308@ari-18308:~/Desktop/myenv$ cat requirements.txt
aravindari==0.1.0
(myenv) ari-18308@ari-18308:~/Desktop/myenv$
```

We can use the requirements file to install the packages specified in it.

It can be done as following:

```
(myenv) ari-18308@ari-18308:~/Desktop/myenv$ pip install -r requirements.txt
Requirement already satisfied: aravindari==0.1.0 in ./lib/python3.7/site-packages (from -r requirements.txt
1)) (0.1.0)
You are using pip version 10.0.1, however version 23.3.2 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
(myenv) ari-18308@ari-18308:~/Desktop/myenv$
```

Another way of installing packages is to use the setup.py file. (pip install .) It is generally preferred in packaging and distribution.

```
ari-18308@ari-18308:~/Desktop/Kanitham$ ls
build  dist  Kanitham.egg-info  Operations  setup.py
ari-18308@ari-18308:~/Desktop/Kanitham$ pip install .
Processing /home/ari-18308/Desktop/Kanitham
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: Kanitham
  Building wheel for Kanitham (setup.py) ... done
  Created wheel for Kanitham: filename=Kanitham-0.0.1-py3-none-any.whl si
ze=1543 sha256=158e422b96b69e203659c6aaaa29e2d88040bea326ef907464689d7ffd
4ab414
  Stored in directory: /tmp/pip-ephem-wheel-cache-iorp4o9v/wheels/63/40/2
4/1ccc903fb6e4d4b03e559f34f00c656d95ff716f72d926159f
Successfully built Kanitham
DEPRECATION: pandas 0.23.4 has a non-standard dependency specifier pytz>=
2011k. pip 24.0 will enforce this behaviour change. A possible replacemen
t is to upgrade to a newer version of pandas or contact the author to sug
gest that they release a version with a conforming dependency specifiers.
 Discussion can be found at https://github.com/pypa/pip/issues/12063
Installing collected packages: Kanitham
  Attempting uninstall: Kanitham
    Found existing installation: Kanitham 0.0.1
    Uninstalling Kanitham-0.0.1:
      Successfully uninstalled Kanitham-0.0.1
Successfully installed Kanitham-0.0.1
```

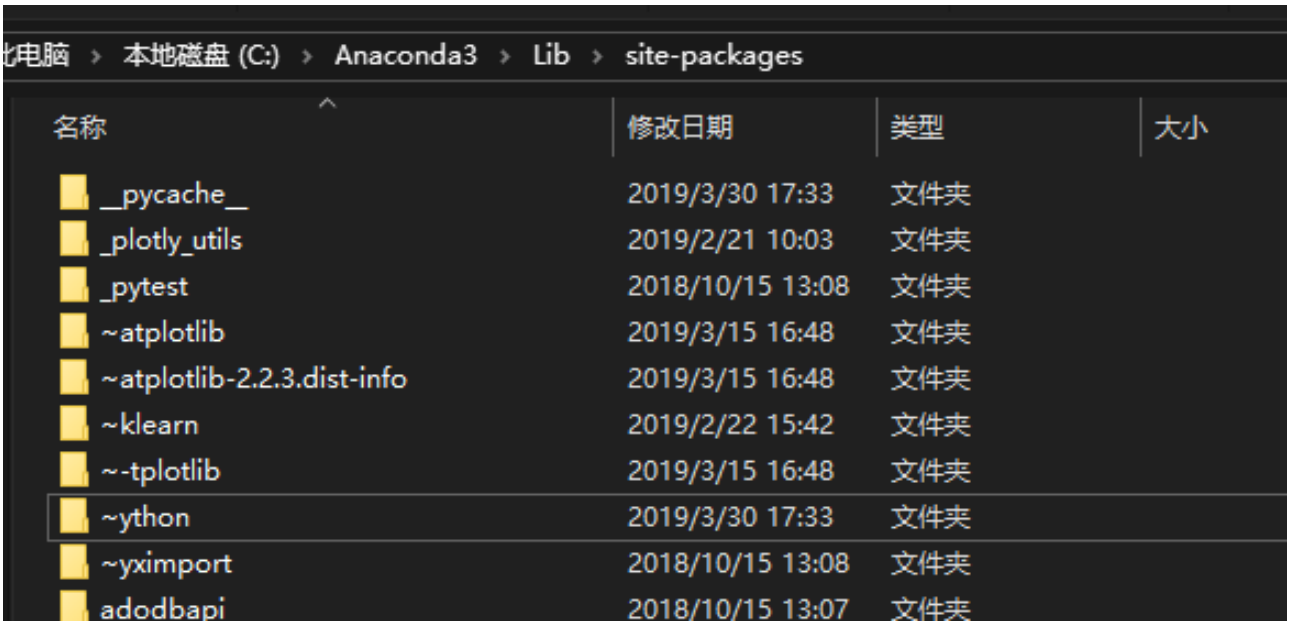
When to use what?

requirements.txt and setup.py are both used in Python projects, but they serve different purposes and are used at different stages of the software development lifecycle. Developers use requirements.txt to declare the dependencies required for their project. setup.py is a Python script that includes information about the project, such as the project name, version, author, and dependencies. It is used during the distribution and installation of the project. When we create a package for distribution, the information in setup.py is used by tools like setuptools to build and package the project.

setup.py is used by tools like setuptools to create distributable packages (source distributions, whl), while requirements.txt is used by pip to install dependencies.

Site-packages

It is a directory where Python installs packages that are intended to be used globally. It's a system-wide location for third-party packages. When we install a package using tools like pip, the package is typically placed in the site-packages directory.



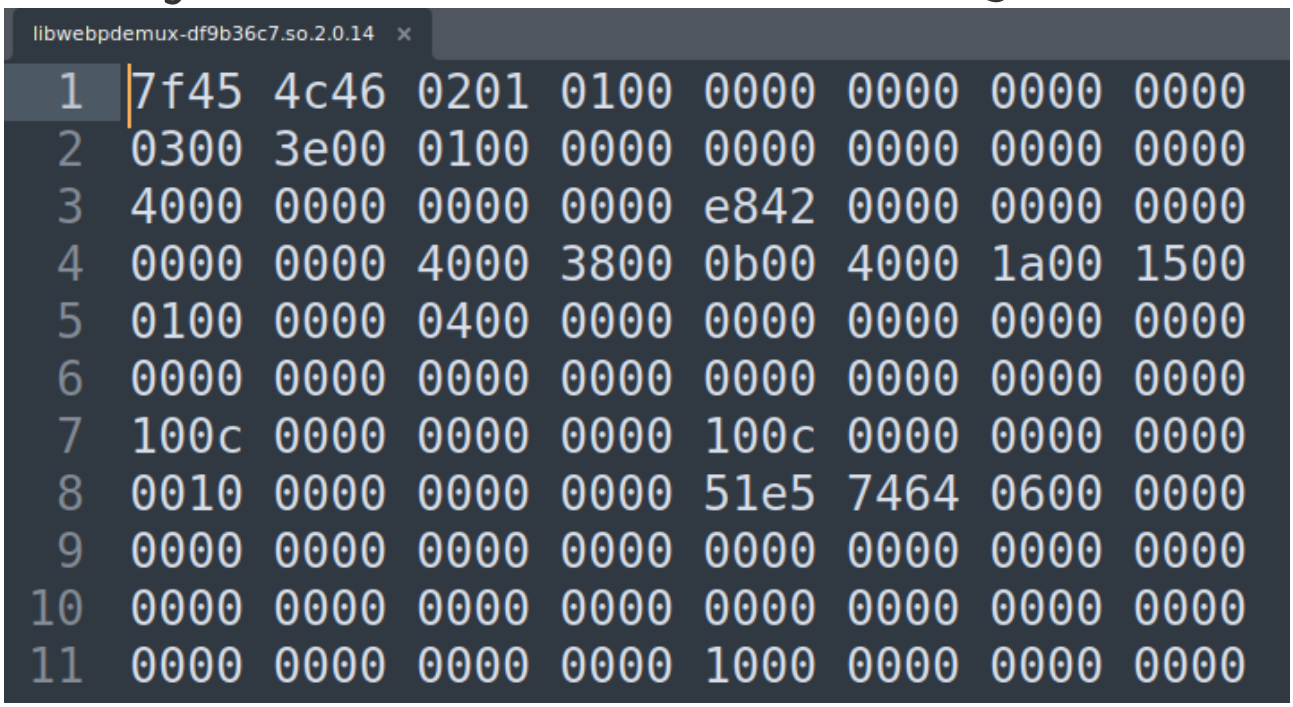
Development Directory

This refers to the directory where we are actively developing our Python project. It contains our source code, scripts & so on.

File Types of Packages

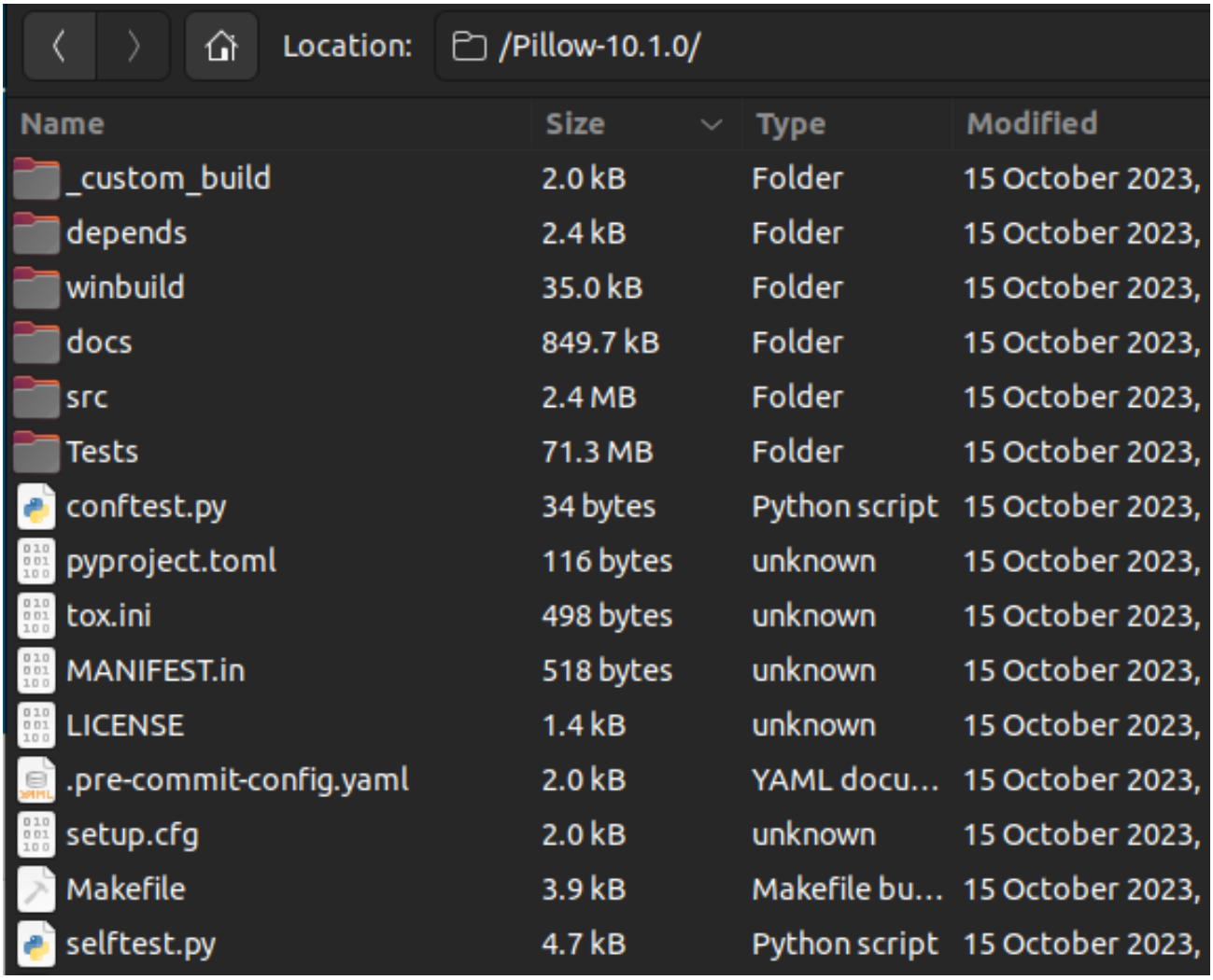
1. whl (Wheel) File Format

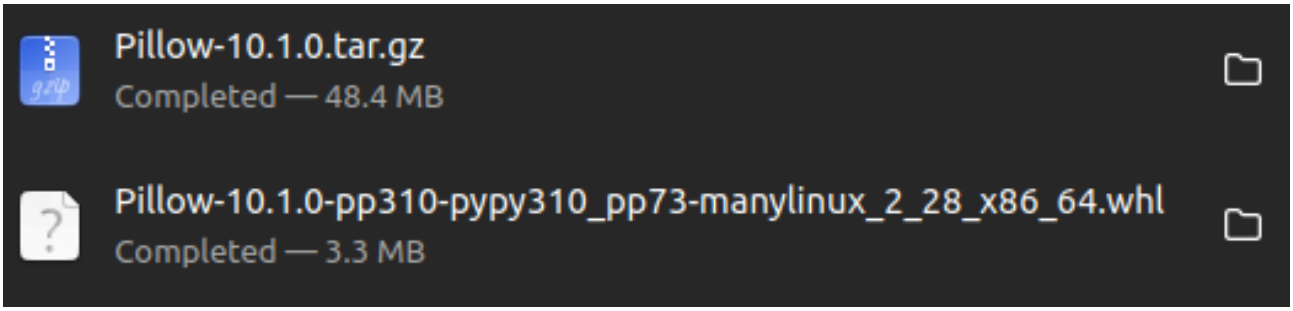
It is a binary distribution format used for packaging Python projects. whl files contain the compiled bytecode of the project and metadata. They are installed using pip.



2. Other File Types

Other file types include source distributions (tar.gz or zip files) that contain the source code of the project. These can be used for installation but require compilation.





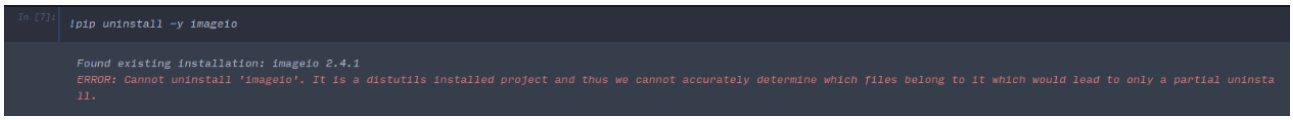
Editable Mode in Installation

Editable mode, often specified using the -e flag with pip, allows to install a package in editable mode. In this mode, the package is not copied to the site-packages directory. Instead, a link or reference to development directory is used. Changes made in the development directory immediately affect the installed package without requiring a reinstallation. Say we have two modules where one depends on another. If we don't install the module A in editable mode, we have to install it again and again whenever we change the module A. Editable mode makes it easy!

```
ari-18308@ari-18308:~/Desktop/learn/A$
ari-18308@ari-18308:~/Desktop/learn/A$
ari-18308@ari-18308:~/Desktop/learn/A$ pip install -e .
```

Dependency Conflict

A dependency conflict occurs when different parts of a software project specify conflicting requirements for a particular library or package. For example, one module may require version 1.0 of a package, while another module requires version 2.0.



Some ways to handle dependency conflicts : Using a virtual environment to isolate dependencies for different projects; Clearly specifying version ranges or constraints for dependencies in our project's configuration files; Using tools like pip, pipenv or poetry to check for and resolve dependencies.

CONDA

PIP is package manager for Python that installs packages from the Python Package Index (PyPI). It is Python-specific and doesn't manage non-Python packages. Conda is a cross-language package manager that can install packages written in any language. It is not limited to Python and can manage environments, dependencies, and packages from different sources.

Install

From a web browser, go to the Anaconda Distribution page, available via the following link: https://www.anaconda.com/distribution/

Find the latest Linux version and copy the link to the installer bash script. Logged into your Ubuntu server as a sudo non-root user, move into the /tmp directory and use curl to download the link you copied from the Anaconda website:

```
> cd /tmp
> curl -O https:// repo.anaconda.com/archive/Anaconda3-2021.05-Linux-x86_64.sh
```

Run the bash script.

```
> bash Anaconda3-2019.03-Linux-x86_64.sh
```

You'll receive the following output to review the license agreement by pressing ENTER until you reach the end.

Welcome to Anaconda3 2021.05
In order to continue the installation process, please review the license agreement.
Please, press ENTER to continue
>>>
...
Do you approve the license terms? [yes|no]

When you get to the end of the license, type yes then press ENTERas long as you agree to the license to complete installation. Once you agree to the license, you will be prompted to choose the location of the installation. You can press ENTER to accept the default location, or specify a different location.

```
Anaconda3 will now be installed into this location:
/home/ari-18308/anaconda3
  - Press ENTER to confirm the location
  - Press CTRL-C to abort the installation
  - Or specify a different location below
[/home/ari-18308/anaconda3] >>>
```

At this point, the installation will proceed. Note that the installation process takes some time. Once installation is complete, you'll receive the following output:

```
...
installation finished.
Do you wish the installer to initialize Anaconda3
by running conda init? [yes|no]
[no] >>>
```

It is recommended that you type yes to use the conda command.

You can now activate the installation with the following command:

> source ~/.bashrc

Use the conda command to test the installation and activation:

> conda list

Create An Environment

```
ari-18308@ari-18308:~/Desktop$ conda create -n py33 python=3.3 anaconda
```

-n py33: This specifies the name of the environment, which in this case is "py33".
python=3.3: This specifies the Python version for the new environment. In this case, it is Python 3.3.
anaconda: This specifies that the Anaconda distribution should be used as the base set of packages for the new environment. The Anaconda distribution includes a wide range of data science and scientific computing packages, making it convenient for users in those domains.

In Environment

To Activate the environment,

```
ari-18308@ari-18308:~/Desktop$ conda activate py33
```

We can set environment variables during activation of an environment as following.

```
ari-18308@ari-18308:~/Desktop$ conda activate py33 export MY_VARIABLE=val
```

When we activate a Conda environment and set environment variables, those variables are specific to that activated environment. They are not global system variables but rather local to the activated environment. When we deactivate the Conda environment, these environment variables are unset, meaning they are no longer active or accessible.
We can install any packages inside an environment as following:

```
ari-18308@ari-18308:~/Desktop$ conda install pygame
```

To list out the packages installed in an environment, we can use conda list.

```
ari-18308@ari-18308:~$ conda list
# packages in environment at /home/ari-18308/anaconda3:
#
# Name                    Version                   Build  Channel
_ipyw_jlab_nb_ext_conf    0.1.0                     py37_0
alabaster                 0.7.11                    py37_0
anaconda                  5.3.1                     py37_0
anaconda-client           1.7.2                     py37_0
anaconda-navigator        1.9.2                     py37_0
anaconda-project          0.8.2                     py37_0
appdirs                   1.4.3            py37h28b3542_0
asn1crypto                0.24.0                    py37_0
astroid                   2.0.4                     py37_0
astropy                   3.0.4            py37h14c3975_0
atomicwrites              1.2.1                     py37_0
attrs                     18.2.0           py37h28b3542_0
```

Exporting an environment to a file means creating a file that contains a specification of all the packages and dependencies installed in a particular Conda environment. This file is typically in YAML format and can be used to recreate the exact environment on another system. It captures the configuration of the

environment, including package names and versions.

```
(base) ari-18308@ari-18308:~$ conda env export > environment.yml
(base) ari-18308@ari-18308:~$ cat environment.yml
name: base
channels:
  - defaults
dependencies:
  - _ipyw_jlab_nb_ext_conf=0.1.0=py37_0
  - alabaster=0.7.11=py37_0
  - anaconda=5.3.1=py37_0
  - anaconda-client=1.7.2=py37_0
  - anaconda-navigator=1.9.2=py37_0
  - anaconda-project=0.8.2=py37_0
  - appdirs=1.4.3=py37h28b3542_0
  - asn1crypto=0.24.0=py37_0
  - astroid=2.0.4=py37_0
  - astropy=3.0.4=py37h14c3975_0
```

To recreate the environment on another system,
>conda env create -f environment.yml
This command reads the environment.yml file and recreates the exact environment, installing the specified versions of each package. It's a convenient way.

To remove a specific package from an activated environment, conda remove can be used.

```
(base) ari-18308@ari-18308:~$ conda remove spyder
```

To deactivate a conda environment, conda deactivate name can be used. It can also be done by using source deactivate in Linux and deactivate in Windows.

```
(base) ari-18308@ari-18308:~$ source deactivate
ari-18308@ari-18308:~$ conda activate base
(base) ari-18308@ari-18308:~$
```

To delete an environment

```
ari-18308@ari-18308:~$
ari-18308@ari-18308:~$ conda env remove --name myenv
```

To update conda

```
ari-18308@ari-18308:~$
ari-18308@ari-18308:~$ conda update conda
```

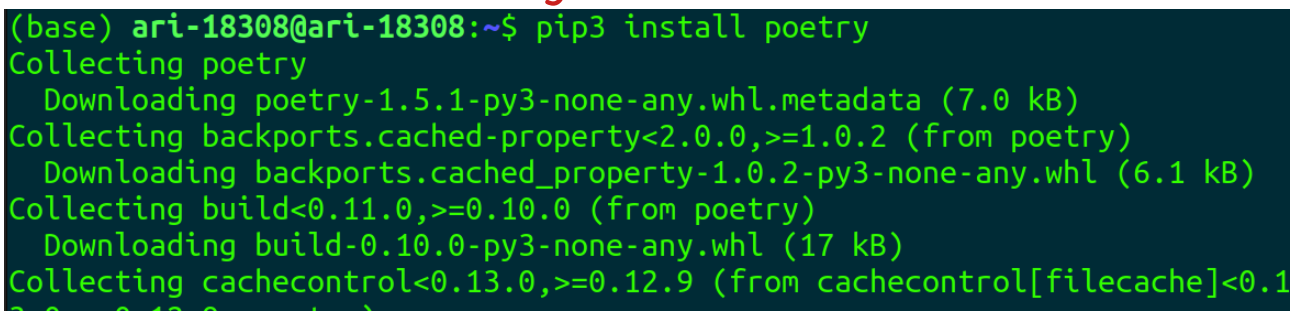
To uninstall an environment, we can directly use the uninstaller.

Advantages

> Conda is not limited to Python packages. It supports packages from various programming languages.
> Conda allows the creation of isolated environments, each with its own set of packages and dependencies.
> Conda excels at handling complex dependency trees.
> Conda installs precompiled binary packages, which can significantly speed up the installation process compared to source-based installations.
>Conda allows users to create and distribute environments and packages without an internet connection.

POETRY

Poetry is a Python dependency management and packaging tool. It simplifies the process of managing project dependencies and packaging by providing a clear and consistent interface. Just as other packages, it can also be installed using pip install poetry.



To install a specific version of Poetry, we can use the following command, pip install poetry== version_number.

To upgrade poetry, we can use pip upgrade poetry or by using poetry self update.

To add packages, if the package is from github, we can simply use `poetry add git+https://github.com/user/repo.git#branch_name;`
If the package is from a local directory, we can do `poetry add /path/to/package`

Poetry Lock File

The poetry.lock file is automatically generated by Poetry and contains the specific versions and hashes of all the dependencies listed in the `pyproject.toml` file. It ensures that all team members or deployments use the same dependency versions.

```
(rangen-py3.7) ari-18308@ari-18308:~/Desktop/rangen$ cat poetry.lock
# This file is automatically @generated by Poetry 1.5.1 and should not be
 changed by hand.
package = []

[metadata]
lock-version = "2.0"
python-versions = "^3.7"
content-hash = "1cecdb5c4f27a43e0bfaed3c063b275782ac693a803e2b2ecd482fcb6
a737d5e"
```

To create a poetry.lock file and install dependencies inside a conda environment using Poetry,

1.Create a new Python virtual environment (conda)

```
ari-18308@ari-18308:~$ conda create --name testenv python=3.8
Solving environment: done
```

2.Activate the environment

```
ari-18308@ari-18308:~$ conda activate testenv
```

3.Navigate to your project directory and Initialize Poetry in your project.

```
(testenv) ari-18308@ari-18308:~$ mkdir PROJ
(testenv) ari-18308@ari-18308:~$ cd PROJ
(testenv) ari-18308@ari-18308:~/PROJ$ poetry init
```

Answer the questions prompted by Poetry to generate a pyproject.toml file.

4.Add your project dependencies to the pyproject.toml file under the

[tool.poetry.dependencies] section.

```
(testenv) ari-18308@ari-18308:~/PROJ$ ls
pyproject.toml
(testenv) ari-18308@ari-18308:~/PROJ$ nano pyproject.toml
(testenv) ari-18308@ari-18308:~/PROJ$ cat pyproject.toml
[tool.poetry]
name = "proj"
version = "0.1.0"
description = ""
authors = ["ARI"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.7"
requests = "^2.26.0"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

5.Run the following command to install the dependencies and generate the poetry.lock file.

```
(testenv) ari-18308@ari-18308:~/PROJ$ poetry install
Updating dependencies
Resolving dependencies... (1.0s)

Package operations: 5 installs, 0 updates, 0 removals

  • Installing certifi (2023.11.17)
  • Installing charset-normalizer (3.3.2)
  • Installing idna (3.6)
  • Installing urllib3 (2.0.7)
  • Installing requests (2.31.0)
```

This will install the specified dependencies and create a poetry.lock file in your project directory.

```
(testenv) ari-18308@ari-18308:~/PROJ$ ls
poetry.lock  pyproject.toml
(testenv) ari-18308@ari-18308:~/PROJ$ cat poetry.lock
# This file is automatically @generated by Poetry 1.5.
 changed by hand.

[[package]]
name = "certifi"
version = "2023.11.17"
description = "Python package for providing Mozilla's
optional = false
```

To generate poetry.lock without installing packages,

```
(testenv) ari-18308@ari-18308:~/PROJ$ poetry lock
Updating dependencies
Resolving dependencies... (0.8s)
(testenv) ari-18308@ari-18308:~/PROJ$ cat poetry.lock
# This file is automatically @generated by Poetry 1.5.1
```

Dependencies Update

The command, poetry update will analyze pyproject.toml file and determine the latest compatible versions for your dependencies, and update the poetry.lock file. If you want to install the updated dependencies immediately, you can run poetry install.

```
(testenv) ari-18308@ari-18308:~/PROJ$ poetry update
Updating dependencies
Resolving dependencies... (0.9s)

No dependencies to install or update
(testenv) ari-18308@ari-18308:~/PROJ$ poetry install
Installing dependencies from lock file

No dependencies to install or update
```

Config Values

To show all configuration settings, we can use poetry config --list

```
(testenv) ari-18308@ari-18308:~/PROJ$ poetry config --list
cache-dir = "/home/ari-18308/.cache/pypoetry"
experimental.system-git-client = false
installer.max-workers = null
installer.modern-installation = true
installer.no-binary = null
```

It displays a list of all configuration settings currently configured in our Poetry environment. It provides information about various settings such as the location of the virtual environment, cache directory, repositories, and more.

To show a specific configuration setting,

```
(testenv) ari-18308@ari-18308:~/PROJ$ poetry config virtualenvs.create
true
```

This command specifically shows the current value of the virtualenvs.create configuration setting. In this case, it checks whether Poetry is configured to automatically create a virtual environment when a new project is created.

To set a new value for a configuration setting,

```
(testenv) ari-18308@ari-18308:~/PROJ$ poetry config virtualenvs.create fa
lse
```

This command changes the value of the virtualenvs.create setting to false. It means that Poetry will no longer automatically create a virtual environment when you create a new project.

To verify the change,

```
(testenv) ari-18308@ari-18308:~/PROJ$ poetry config virtualenvs.create
false
```

To Uninstall, pip uninstall is enough while working with pip. For conda, first deactivate the environment if active. Then, simply type, conda deactivate poetry.

BLACK AND ISORT

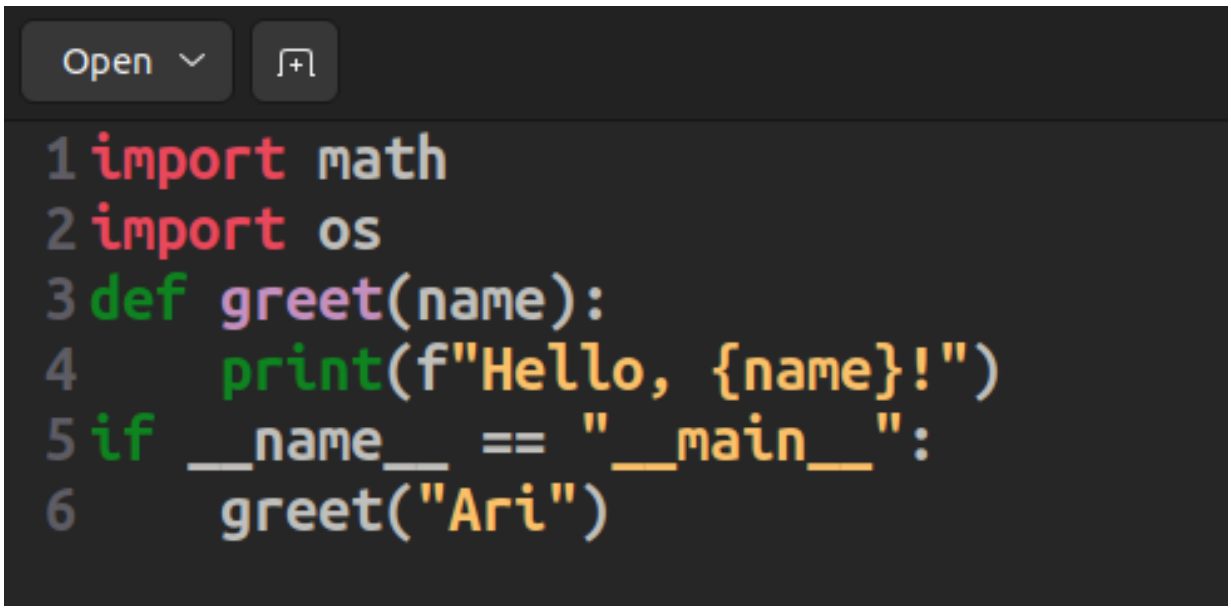
A Formatting Tool helps ensure consistent and readable code by automatically applying a set of coding style rules to the given source code. Two popular formatting tools for Python are Black and isort.

To Install Black and isort,

> pip install black

> pip install isort

Let's create a simple Python script (test.py) with intentionally unformatted and disordered code:



Use the following commands to format the code using Black and isort.

```
(base) ari-18308@ari-18308:~/PROJ$ isort test.py
Fixing /home/ari-18308/PROJ/test.py
(base) ari-18308@ari-18308:~/PROJ$
```

Now, let's check our test.py

Open

```python
import math
import os


def greet(name):
    print(f"Hello, {name}!")
if __name__ == "__main__":
    greet("Ari")
```

To do the same with black,

```
(base) ari-18308@ari-18308:~/PROJ$ black test.py
reformatted test.py

All done! ✨ 🍰 ✨
1 file reformatted.
(base) ari-18308@ari-18308:~/PROJ$
```

Open

```python
import math
import os


def greet(name):
    print(f"Hello, {name}!")


if __name__ == "__main__":
    greet("Ari")
```

We can note that, the code formatted by black is not same as that made by isort.

If the code is already formatted, these commands will not produce any output

or will show a success message. If there are formatting issues, they will be displayed in the console.

Imports properly ordered refers to the practice of

```
(base) ari-18308@ari-18308:~/PROJ$ black --check test.py
All done! ✨ 🍰 ✨
1 file would be left unchanged.
(base) ari-18308@ari-18308:~/PROJ$ isort --check test.py
(base) ari-18308@ari-18308:~/PROJ$
```

arranging import statements in a consistent and organized manner within a Python source code file. The goal is to enhance readability and maintainability of the code. Properly ordered imports typically follow a set of conventions or rules, and tools like isort are used to automatically enforce these conventions. Imports are often grouped into sections based on their type, such as standard library modules,

third-party packages, and local (project-specific) modules. Within each group, imports are usually ordered alphabetically. This makes it easy to locate a specific module or package.

Blank lines may be used to separate different groups of imports, improving visual separation and making the code more readable.

```
# Standard Library Imports

import os

import sys

# Third-Party Imports

from flask import Flask, render_template

import requests

# Local Imports
```

```
from mymodule import my_function
```

To check if all imports are properly ordered,

```
(base) ari-18308@ari-18308:~/PROJ$ isort --check-only test.py
(base) ari-18308@ari-18308:~/PROJ$
```

To check for unwanted imports, you might want to use a linter like pylint or flake8. These tools can identify unused imports and other potential issues.

Install PyLint

```
> pip install pylint
```

Run PyLint

```
> pylint test.py
```

PyLint will provide information about code quality, including any unused imports or other issues.

```
(base) ari-18308@ari-18308:~/PROJ$ pylint test.py
************* Module test
test.py:1:0: C0111: Missing module docstring (missing-docstring)
test.py:5:0: C0111: Missing function docstring (missing-docstring)
test.py:1:0: W0611: Unused import math (unused-import)
test.py:2:0: W0611: Unused import os (unused-import)
```

MyPy

Type checking is a process in programming where the types of variables or expressions are verified to ensure that they adhere to the expected types. The goal is to catch errors related to data types early in the development process, ideally before the code is executed. Type checking can be performed either statically or dynamically. Static Type Checking occurs at compile-time, and the types of

variables are checked before the code is run. Languages like Java, C++, and some versions of Python (with tools like Mypy) use static type checking.

Dynamic Type Checking occurs at runtime, and type information is checked during the execution of the program. Languages like Python, JavaScript, and Ruby use dynamic type checking.

A Type Checker is a tool or component of a programming language that performs type checking. It analyzes the code to ensure that variables and expressions are used in a manner consistent with their declared types. Type checkers

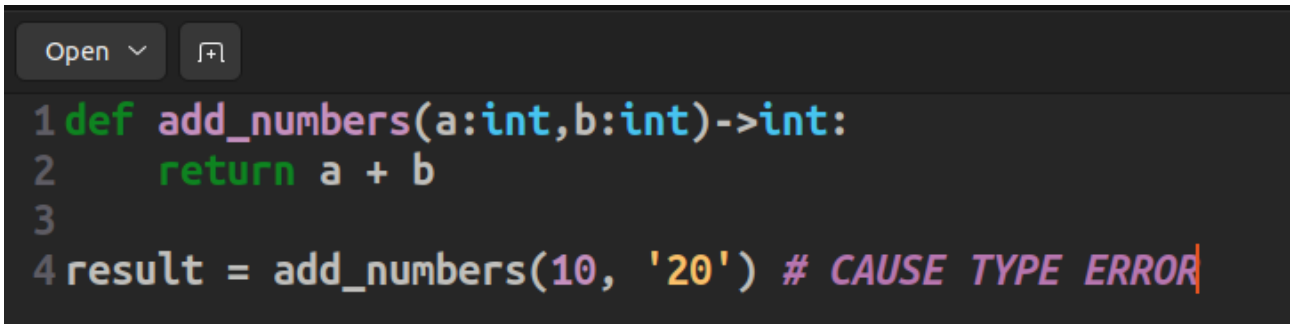
can catch potential errors early in the development process, helping developers write more robust and bug-free code.

Why We Need One for Python

Python is dynamically typed, which means that variable types are determined at runtime. While this flexibility is convenient, it can lead to runtime errors that could have been caught earlier with static type checking. Tools like Mypy bring static typing to Python, allowing developers to catch type-related errors during development.

Static Typing using Mypy

Let's consider a simple Python script, test.py:



Mypy is an open-source third-party static type checker for the Python programming language. It aims to bring optional static typing to Python, allowing developers to catch type-related errors early in the development process. Mypy is not a part of the Python standard library, but it has gained popularity as a tool for improving code quality and maintainability. Comment like #type: ignore at the error-prone line to ignore .

```
(base) ari-18308@ari-18308:~/PROJ$ mypy test.py
Success: no issues found in 1 source file
(base) ari-18308@ari-18308:~/PROJ$ cat test.py
def add_numbers(a:int,b:int)->int:
    return a + b

result = add_numbers(10, '20') #type: ignore
(base) ari-18308@ari-18308:~/PROJ$
```

To ignore an error in a line in Mypy, we can use this # type: ignore comment at the end of the line where the error occurs.

To skip file or code block from being checked by mypy, we can skip entire files or code blocks using #type: ignore at the beginning of the file or block.

```
(base) ari-18308@ari-18308:~/PROJ$ cat test.py
#type: ignore
def add_numbers(a:int,b:int)->int:
    return a + b

result = add_numbers(10, '20')
(base) ari-18308@ari-18308:~/PROJ$ mypy test.py
Success: no issues found in 1 source file
(base) ari-18308@ari-18308:~/PROJ$
```

mypy.ini

mypy.ini is a configuration file for Mypy, allowing to customize its behavior.

Create a file named mypy.ini with the specified content and place it in the root of your project repository. This file can be used to configure Mypy according to your project's needs.

```
  GNU nano 6.2                          mypy.ini *
[mypy]
# Specify the Python version (e.g., 3.8)
python_version = 3.8

# Enable strict mode (catches more type errors)
strict = True

# Ignore missing imports (can be useful for gradual type checking)
ignore_missing_imports = True

# Specify additional directories to check (comma-separated)
# e.g., check only files in the 'src' directory
check_untyped_defs = True
disallow_untyped_defs = True

# Exclude files or directories from type checking
exclude = ^(tests|docs)/
```

To add mypy.ini to a git repo,

```
(base) ari-18308@ari-18308:~/PROJ$ git add mypy.ini
(base) ari-18308@ari-18308:~/PROJ$ git commit -m "Add mypy.ini for type c
hecking configuration"
[master (root-commit) 259bb59] Add mypy.ini for type checking configurati
on
 Committer: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 1 file changed, 25 insertions(+)
 create mode 100644 mypy.ini
(base) ari-18308@ari-18308:~/PROJ$ git push
```

Linting

Linting, or static code analysis, is the process of analyzing source code for potential errors, bugs, and style issues without actually executing the code. It helps ensure code quality, maintainability, and adherence to coding standards. Pylint is a popular Python tool for linting that checks for various aspects such as coding standards, potential errors, and code smells.

```
  GNU nano 6.2                          test.py *
def add_numbers(a, b):
    result = a + b
    print(result)
add_numbers(5, '10')  # Intentional error: adding int and str
```

Let's install pylint as below:

```
(base) ari-18308@ari-18308:~/PROJ$ pip install pylint
Requirement already satisfied: pylint in /home/ari-18308/anaconda3/lib/py
thon3.7/site-packages (2.1.1)
```

Now, let's check out.

```
(base) ari-18308@ari-18308:~/PROJ$ pylint test.py
************* Module test
test.py:1:0: C0111: Missing module docstring (missing-docstring)
test.py:1:0: C0103: Argument name "a" doesn't conform to snake_case namin
g style (invalid-name)
test.py:1:0: C0103: Argument name "b" doesn't conform to snake_case namin
g style (invalid-name)
test.py:1:0: C0111: Missing function docstring (missing-docstring)

-------------------------------------------------------------------
Your code has been rated at 0.00/10 (previous run: 3.33/10, -3.33)
```

Pylint will output messages and a score for the code quality.

To skip or ignore a specific error raised by Pylint, you can use the #pylint: disable=<error-code> comment in your code.

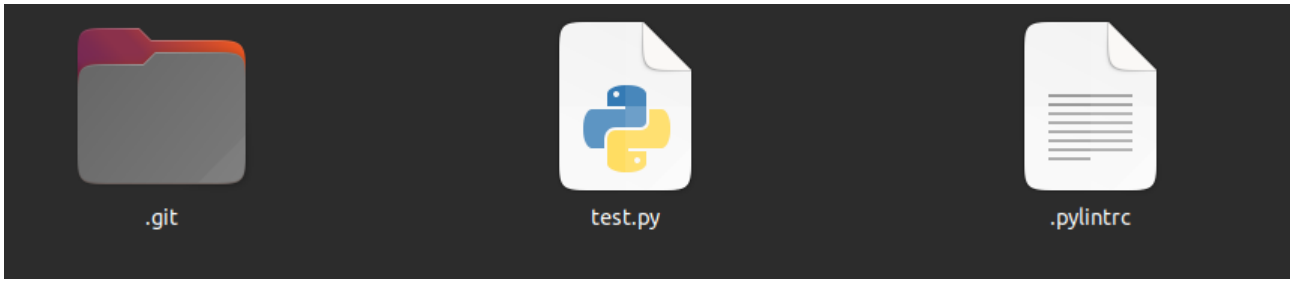
```
(base) ari-18308@ari-18308:~/PROJ$ cat test.py
# test.py

def add_numbers(a, b):
    result = a + b  # pylint: disable=unsupported-operand-type
    print(result)
```

To skip a File from Pylint Checking, we can use the # pylint: skip-file comment at the beginning of the file.

```
(base) ari-18308@ari-18308:~/PROJ$ cat test.py
# test.py
# pylint: skip-file

def add_numbers(a, b):
    result = a + b
    print(result)
(base) ari-18308@ari-18308:~/PROJ$ pylint test.py
(base) ari-18308@ari-18308:~/PROJ$
```

pylintrc

We can create a pylintrc file to configure Pylint settings. This file can be used to set various options, disable/enable specific checks, etc.



Let's define the contents in pylintrc file.

.pylintrc

```
# pylintrc

[MASTER]
disable = C0103, C0114, C0115, C0116  # Disable specific checks (comma-separated error codes)

[MESSAGES CONTROL]
disable = W1203  # Disable specific warning

```

Now, let's run.

```
(base) ari-18308@ari-18308:~/PROJ$ pylint test.py
(base) ari-18308@ari-18308:~/PROJ$
```

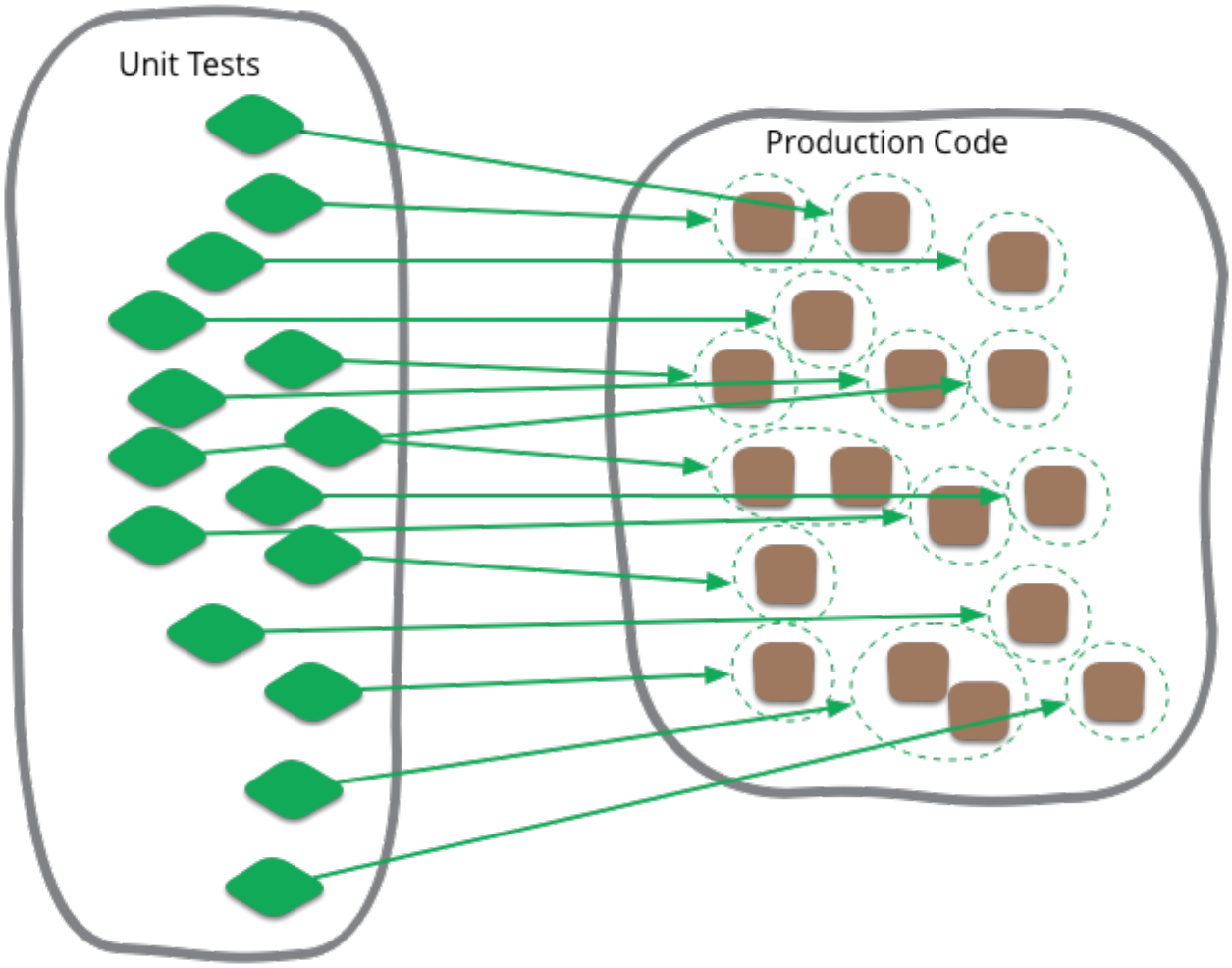
To add pylintrc to git repo, simply create a file named .pylintrc in the root of your project and add the desired configurations. It will automatically be picked up by Pylint when you run it. Make sure to include this file in your version control system (e.g., Git) so that other developers can benefit from the same linting configurations.

```
(base) ari-18308@ari-18308:~/PROJ$ git add .pylintrc
(base) ari-18308@ari-18308:~/PROJ$ git commit -m "Add .pylintrc file for
Pylint configuration"
[master f385dc9] Add .pylintrc file for Pylint configuration
 Committer: Ariharasudhan S <ari-18308@ari-18308.cbe.zohocorpin.com>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 1 file changed, 8 insertions(+)
 create mode 100644 .pylintrc
(base) ari-18308@ari-18308:~/PROJ$
```

PyTest

A Unit Test is a type of software testing where individual units or components of a program are tested in isolation to ensure they work as intended. These tests focus on specific functionalities or units of code and aim to validate that each unit behaves correctly. Unit tests are typically written and executed by developers during the

development phase to ensure the correctness of their code.



Behavior Tests, on the other hand, are higher-level tests that focus on the overall behavior of the software from an end-user perspective. These tests evaluate how different components interact with

each other and whether the software meets the specified requirements. Unlike unit tests, behavior tests are more concerned with the system's external behavior rather than individual units of code.

Unit Tests with PyTest

Let's create a simple Python module operations.py with some functions and write unit tests for them using Pytest.

operations.py

```python
# operations.py

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    if y == 0:
        raise ValueError("Cannot divide by zero")
    return x / y
```

Now, let's write unit tests using Pytest.

test_operations.py

```python
# test_operations.py
import operations as math_operations
import pytest

def test_add():
    assert math_operations.add(2, 3) == 5
    assert math_operations.add(-1, 1) == 0
    assert math_operations.add(0, 0) == 0

def test_subtract():
    assert math_operations.subtract(5, 2) == 3
    assert math_operations.subtract(0, 0) == 0
    assert math_operations.subtract(-1, -1) == 0

def test_multiply():
    assert math_operations.multiply(2, 3) == 6
    assert math_operations.multiply(0, 5) == 0
    assert math_operations.multiply(-1, 4) == -4

def test_divide():
    assert math_operations.divide(6, 2) == 3
    assert math_operations.divide(0, 5) == 0
    assert math_operations.divide(-8, -2) == 4

def test_divide_by_zero():
    with pytest.raises(ValueError):
        math_operations.divide(10, 0)

```

To run the tests, save the above code in two separate files (operations.py and test_operations.py) respectively.

Then, just run!

```
ari-18308@ari-18308:~/PROJ$ pytest test_operations.py
============================= test session starts ==============================
platform linux -- Python 3.7.0, pytest-3.8.0, py-1.6.0, pluggy-0.7.1
rootdir: /home/ari-18308/PROJ, inifile:
plugins: arraydiff-0.2, doctestplus-0.1.3, openfiles-0.3.0, remotedata-0.
3.0
collected 5 items

test_operations.py .....                                                 [100%]

========================== 5 passed in 0.02 seconds ===========================
ari-18308@ari-18308:~/PROJ$
```

This will execute the tests, and you will get a report of the test results.

To save the report, we can redirect the output to a file.

```
ari-18308@ari-18308:~/PROJ$ pytest test_operations.py > report.txt
ari-18308@ari-18308:~/PROJ$ cat report.txt
============================= test session starts ==============================
=======
platform linux -- Python 3.7.0, pytest-3.8.0, py-1.6.0, pluggy-0.7.1
rootdir: /home/ari-18308/PROJ, inifile:
plugins: arraydiff-0.2, doctestplus-0.1.3, openfiles-0.3.0, remotedata-0.
3.0
collected 5 items

test_operations.py .....
[100%]
```

Pytest Fixtures

Fixtures are a way to set up and tear down resources needed (preconditions for tests). Fixtures are defined using the @pytest.fixture decorator.

operations.py

```python
# operations.py

class Calculator:
    def add(self, x, y):
        return x + y

    def subtract(self, x, y):
        return x - y

    def multiply(self, x, y):
        return x * y

    def divide(self, x, y):
        if y == 0:
            raise ValueError("Cannot divide by zero")
        return x / y

```

Instead of duplicating setup code in every test, you can define a fixture to encapsulate the setup logic.

This makes the test functions cleaner.

test_operations.py

```python
# test_operations.py

import pytest
from operations import Calculator

# Fixture to create a Calculator instance for testing
@pytest.fixture
def calculator():
    return Calculator()

# Test functions using the calculator fixture
def test_add(calculator):
    result = calculator.add(2, 3)
    assert result == 5

def test_subtract(calculator):
    result = calculator.subtract(5, 2)
    assert result == 3

def test_multiply(calculator):
    result = calculator.multiply(2, 3)
    assert result == 6

def test_divide(calculator):
    result = calculator.divide(6, 2)
    assert result == 3

    with pytest.raises(ValueError):
        calculator.divide(10, 0)

```

This example illustrate how to use fixtures to set up and

reuse common data or resources in your tests.

```
ari-18308@ari-18308:~/PROJ$ pytest test_operations.py
============================= test session starts ==============================
platform linux -- Python 3.7.0, pytest-3.8.0, py-1.6.0, pluggy-0.7.1
rootdir: /home/ari-18308/PROJ, inifile:
plugins: arraydiff-0.2, doctestplus-0.1.3, openfiles-0.3.0, remotedata-0.
3.0
collected 4 items

test_operations.py ....                                                  [100%]

=========================== 4 passed in 0.01 seconds ===========================
ari-18308@ari-18308:~/PROJ$
```

We can also use one fixture inside another as shown below.

test_operations.py

```python
# test_operations.py

import pytest
from operations import Calculator

# Fixture to create a Calculator instance for testing
@pytest.fixture
def calculator():
    return Calculator()

# Fixture to perform an operation using the calculator fixture
@pytest.fixture
def operation_result(calculator):
    return calculator.add(10, 5)

# Fixture that depends on another fixture (calculator)
@pytest.fixture
def dependent_fixture(calculator):
    return calculator.subtract(8, 4)

# Test function using the operation_result fixture
def test_operation_result(operation_result):
    assert operation_result == 15

# Test function using the dependent_fixture
def test_dependent_fixture(dependent_fixture):
    assert dependent_fixture == 4

```

In pytest, you can use the tmp_path fixture to create temporary directories and files for testing purposes. This is a simple file reader.

filereader.py

```python
# filereader.py

def read_file(file_path):
    with open(file_path, 'r') as file:
        return file.read()

```

Let's check if it works correctly or not.

test_filereader.py

```python
# test_filereader.py
import pytest
import os
from filereader import read_file

def test_read_file(tmpdir):
    # Create a temporary file with content
    file_content = "Hello, this is a temporary file!"
    temp_file_path = tmpdir.join("test_file.txt")

    with open(temp_file_path, 'w') as temp_file:
        temp_file.write(file_content)

    # Call the function with the temporary file path
    result = read_file(str(temp_file_path))

    # Assert that the function returns the correct content
    assert result == file_content

    # Assert that the temporary file has been created
    assert temp_file_path.check(file=True)

# This test will automatically use a temporary directory provided by pytest
# and clean up the temporary files and directories after the test runs.

```

YES! It reads correctly..

```
ari-18308@ari-18308:~/PROJ$ pytest test_filereader.py
========================== test session starts ==========================
platform linux -- Python 3.7.0, pytest-3.8.0, py-1.6.0, pluggy-0.7.1
rootdir: /home/ari-18308/PROJ, inifile:
plugins: arraydiff-0.2, doctestplus-0.1.3, openfiles-0.3.0, remotedata-0.
3.0
collected 1 item

test_filereader.py .                                                [100%]

======================== 1 passed in 0.01 seconds ========================
```

What's happening? There will be a temporary folder created, which is used to store test files.

The conftest File

conftest.py is a special file in pytest that can be used to define fixtures, hooks, and other configurations that are shared across multiple test files within a directory or package. It allows to centralize common configurations and utilities for your tests.

A simple Example is as following:

conftest.py

```python
# conftest.py

import pytest

# Define a fixture that returns a sample data
@pytest.fixture
def sample_data():
    return {"name": "ARI", "age": 21, "city": "In between Nellai and Tenkasi"}

# Define a hook that runs before each test function
def pytest_runtest_setup(item):
    print(f"\nRunning test: {item.nodeid}")

```

Now, let's create a sample file to perform test on it.

test_example.py

```python
# test_example.py

def test_sample_data(sample_data):
    assert sample_data["name"] == "ARI"
    assert sample_data["age"] == 21
    assert sample_data["city"] == "Trichy or Tirupur"

```

Let's test!

```
ari-18308@ari-18308:~/PROJ$ pytest
============================= test session starts ==============================
platform linux -- Python 3.7.0, pytest-3.8.0, py-1.6.0, pluggy-0.7.1
rootdir: /home/ari-18308/PROJ, inifile:
plugins: arraydiff-0.2, doctestplus-0.1.3, openfiles-0.3.0, remotedata-0.
3.0
collected 1 item

test_example.py .                                                      [100%]

=========================== 1 passed in 0.01 seconds ===========================
```

Mock Objects

Mock objects are a crucial concept in testing, particularly when we want to isolate the code under test from external dependencies. They are used to simulate the behavior of real objects and can be controlled to return specific values or simulate certain behaviors. In Python, the unittest.mock module provides a Mock class that is commonly used for creating mock objects. Here is an example of how you might use the pytest-mock plugin to mock an HTTP request:

```
import pytest
import requests
from unittest.mock import Mock

@pytest.fixture
def mock_get(mocker):
    mock = Mock()
    mocker.patch('requests.get', return_value=mock)
    return mock
def test_get_request(mock_get):
    mock_get.return_value.status_code = 200
    mock_get.return_value.json.return_value = {'key': 'value'}
    response = requests.get('http://example.com')
    assert response.status_code == 200
    assert response.json() == {'key': 'value'}
```

In this example, the `mock_get` fixture is used to mock the `requests.get` function. The fixture returns a mock object that is patched into the `requests.get` function using the `mocker.patch` method.

In the `test_get_request` function, the mock object is configured to return a specific status code and JSON response when the `requests.get` function is called. The test function then makes an HTTP request using the requests.get function, and it verifies that the correct status code and JSON response are returned.

Catching Exceptions

The pytest framework's raises() method asserts that a block of code or a function call raises the specified exception. If it does not, the method raises a failure exception, indicating that the intended exception was not raised. Let's create a simple program.

test.py

```python
import pytest

def func(x):
  if x == 5:
    raise ValueError("x must be a value other than 5")
  return x

def test_func():
  with pytest.raises(ValueError, match="x must be a value other than 5"):
    func(5)

# func(5)     # Raises an exemption

test_func()
```

```
================================== ERRORS ==================================
_____________________ ERROR collecting test.py _____________________
test.py:12: in <module>
    func(5)     # Raises an exemption
test.py:5: in func
    raise ValueError("x must be a value other than 5")
E   ValueError: x must be a value other than 5
!!!!!!!!!!!!!!!! Interrupted: 1 errors during collection !!!!!!!!!!!!!!!!
========================= 1 error in 0.13 seconds ==========================
ari-18308@ari-18308:~/PROJ$
```

DVC

DVC is a version control system designed for machine learning (ML) projects. Traditional version control systems like Git are excellent for code versioning, but they are not well-suited for handling large datasets and model files that are common in ML projects. DVC addresses this limitation by providing a way to version control and manage the data, dependencies, and experiments associated with ML projects. DVC allows to version control datasets without duplicating them, saving storage space and facilitating efficient collaboration. DVC ensures that your experiments are reproducible by capturing dependencies, hyper

parameters, and data versions. DVC makes it easier for teams to collaborate on ML projects by tracking changes in data and facilitating sharing of experiments. As datasets and models grow in size, DVC handles them efficiently without affecting the performance of traditional version control systems.

Working with DVC

To demonstrate DVC, let's assume we have a project with some data files.

1.Initialize DVC in our project

```
> dvc init
```

2. Add data to DVC:

```
> dvc add datafile.csv
```

3. Commit changes:

```
> git add .
> git commit -m "Added DVC"
> dvc push
```

In this example, we've initialized DVC, added a data file to it, committed the changes to Git, and pushed the data to a DVC remote (could be a cloud storage or another server).

To remove data from the local DVC cache without affecting the remote storage:

```
> dvc gc
```

This command removes unnecessary files from the DVC cache, freeing up space on our local machine.

Workflow for Maintaining Data in an ML Experiment

Data Versioning: We can use DVC to version control our datasets.

Experiment Tracking: Use a tool like MLflow or TensorBoard for tracking experiments. Log parameters, metrics, and artifacts to keep a record of your experiments.

Model Versioning: Version control your models using Git or a similar system. Track changes to the model architecture and parameters.

Dependency Management: Use a requirements.txt file or a similar approach to manage your project's dependencies. This ensures that others can recreate our environment easily.

Documentation: Maintain clear documentation for our experiments, including details on data sources, preprocessing steps, and model architectures. This helps others understand and reproduce our work.