Python Package Development

AI starts from here

Pascal Sun

UWA NLP-TLP Group

The University of Western Australia

November 28, 2024

Table of Contents

1 Introduction

2 Virtual Environment

3 Project

4 Development

5 CLI

6 Doc

7 Testing

8 Distribution

9 Docker GPU

10 License

11 Q&A

Introduction

Why we need to deliver Python Package?

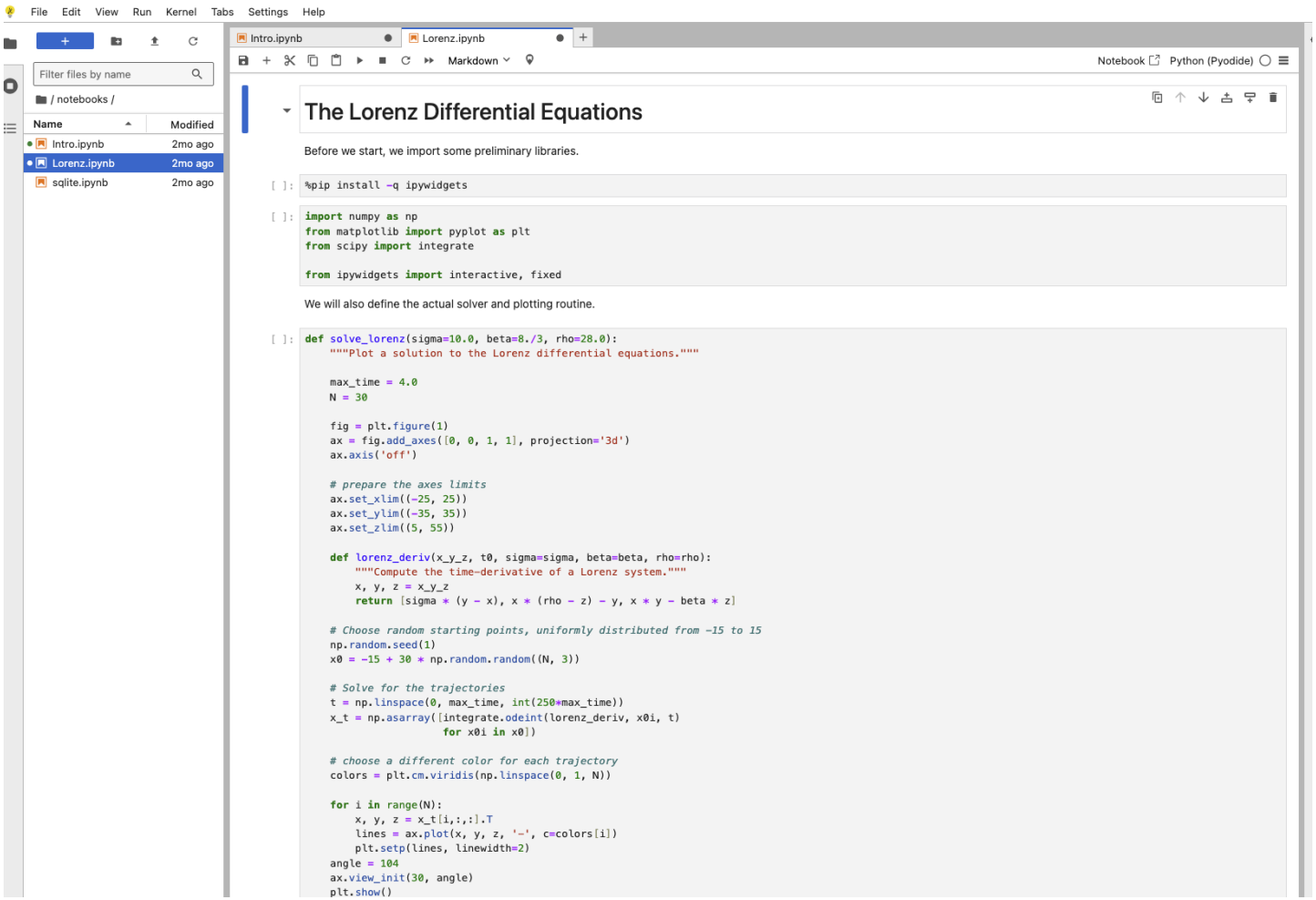

Figure: You can not deploy this to production

Why we need to deliver Python Package?

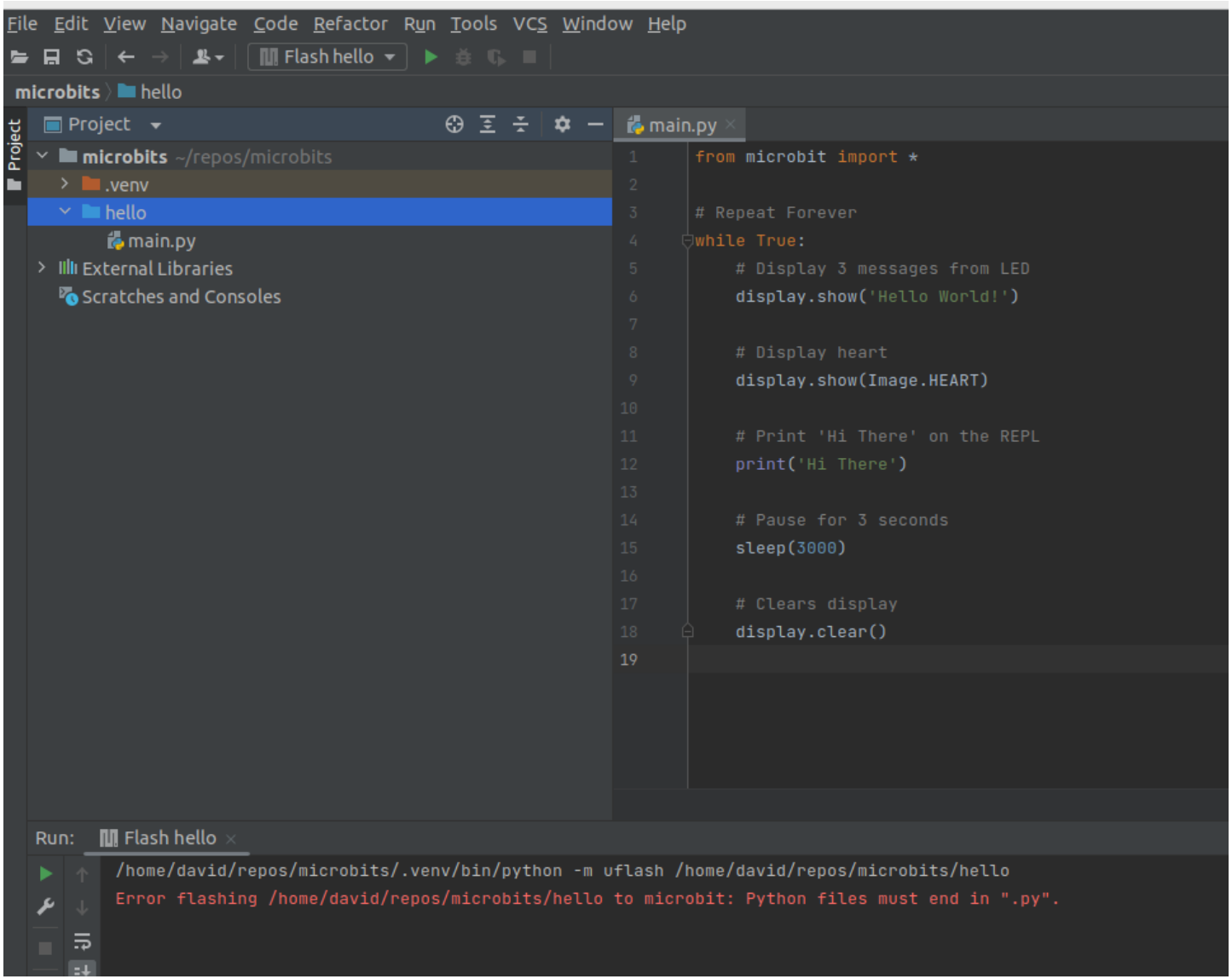

Figure: You also can not deliver this to production

Life is short

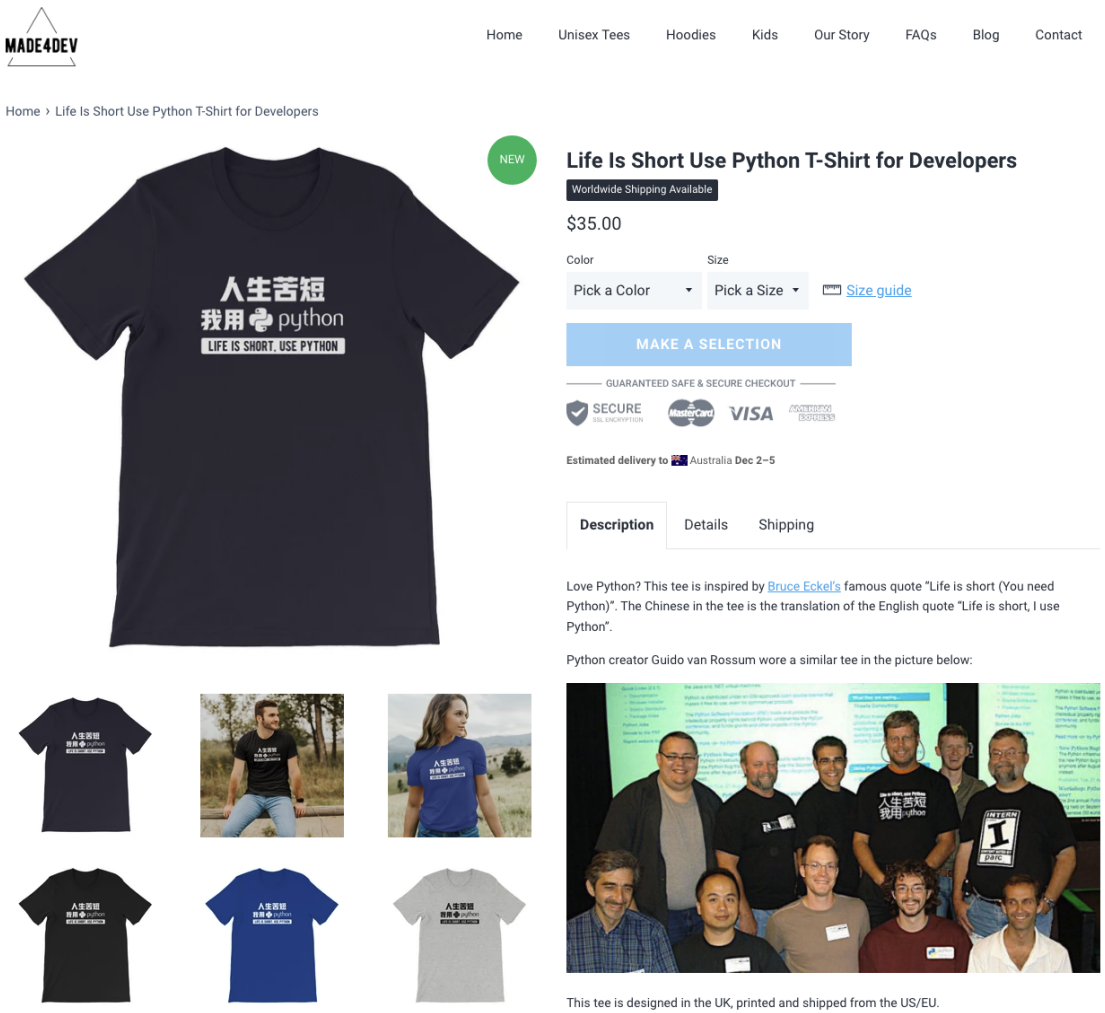

Figure: Life is short, (I use Python/You need Python)

Do not reinvent wheels

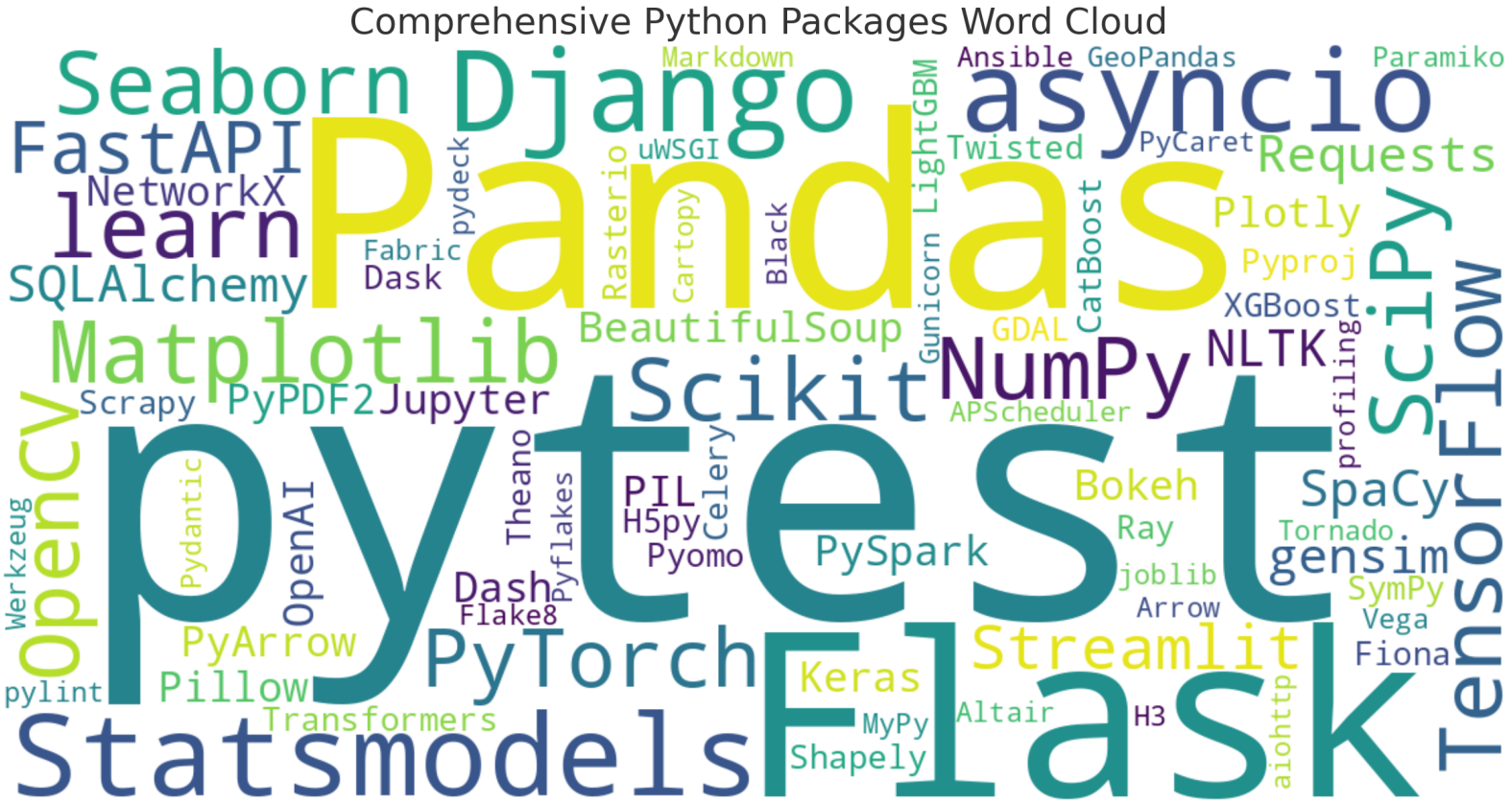

Figure: There are so many useful wheels

Virtual Environment

Why we need virtual environment?

- You will need different Python versions for different projects
- For each project, you will need different packages
- Isolates project dependencies from system-wide Python installation
- Prevents conflicts between package versions

How Python Virtual Environment Works

Core Mechanism

- Creates isolated Python environment by copying:
  - Python interpreter
  - Core libraries
  - pip package manager
- Modifies system path during activation

Activation Process

1. Prepends venv's bin to PATH
2. Updates shell prompt
3. Changes Python path search order

Directory Structure

```
myenv/
    bin/              # Scripts
    include/          # Header files
    lib/              # Libraries
        pythonX.Y/site-packages/
    pyvenv.cfg        # Configuration
```

Key Features

- Lightweight
- No system changes
- Easy cleanup

Understanding Dependency Hell

- **Dependency Hell:** When software dependencies conflict
- Common scenarios:
  - Project A needs Package X version 1.0
  - Project B needs Package X version 2.0
- Even worse:
  Project A needs Package Y, which depends on Package X version 2.0

The hell tricks the AI

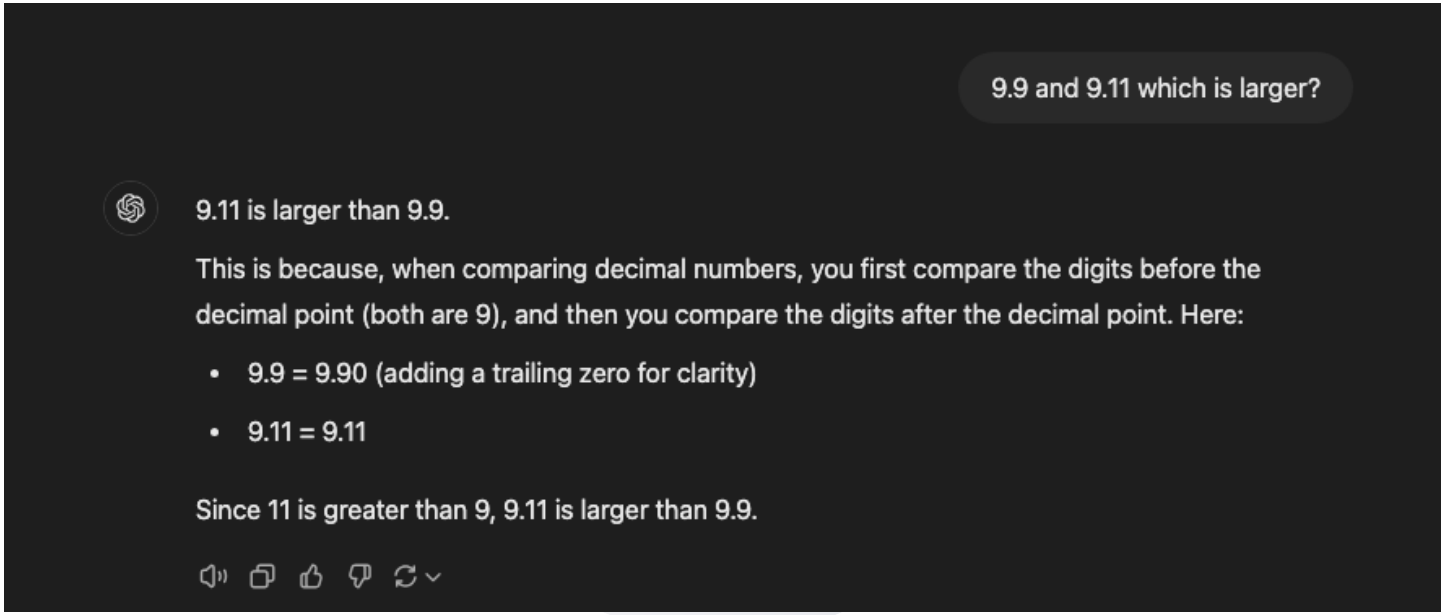

Figure: Math calculation or Version Comparison

Problematic issue calling for your solution

Core Problem

Package version resolution is essentially a graph search problem:

- Need to satisfy all version constraints simultaneously

Critical Edge Cases

- **High-Frequency Updates:**
  - Example: AWS CLI with near-daily releases
  - Thousands of versions increase computational complexity
  - Which means it will take "forever" to resolve the versions
- **Research Software Issues:**
  - Research packages often lack industry-standard versioning
  - Inconsistent maintenance and version specifications
  - Which means the resolve of different packages will fail

Dependency Management Tools

Key Management Needs

- **Python Version:** Multiple versions across projects
- **Dependencies:** Project-specific packages and versions

Evolution of Tools

- **Early Days:** pyenv (Python versions) + virtualenv (isolated environments)
- **pipenv:** Combined pip and virtualenv, introduced Pipfile
- **conda:** Handles non-Python dependencies, scientific computing focus
- **poetry:** Modern dependency resolution, simplified packaging, uses pyproject.toml

Why Different Tools?

- **conda:** Solves scientific computing needs (C/C++ dependencies)
- **poetry:** Try to address the dependency hell problem

Virtual Environment Solutions

Practical Approaches

No one-size-fits-all solution in my opinion, but here are reliable approaches:

1. **Conda-based Solution**
   - Use when conda installation is feasible
   - `conda install` or `pip install` both work
   - Manage via `requirements.txt`
   - Consider disk space
2. **Traditional Python Setup**
   - Python via `brew`/`pyenv`
   - Create env: `python -m venv venv`
   - Use `pip` + `requirements.txt`
3. **Docker-based Approach**
   - Recommended for production
   - Configure in Dockerfile
   - Manage via `requirements.txt`
4. **Poetry Workflow**
   - Modern dependency management
   - For stable, non-cutting-edge projects

1. Conda Method

Setup - Scientific Computing Choice

```
# Create environment with Python 3.9
conda create -n myenv python=3.9

# Activate environment
conda activate myenv

# Install packages
conda install numpy pandas
pip install requests  # also works

# Save dependencies, I actually prefer doing this mannually
pip freeze > requirements.txt
```

- **Pros:** Best for scientific packages (numpy, pandas)
- **Cons:** Large disk space, slower than pip
- **Use when:** Working with data science projects

2. Traditional venv Method

Setup - Lightweight Choice

```
# Install Python (if needed)
brew install python@3.9  # or pyenv install 3.9.0

# Create and activate environment
python -m venv venv
source venv/bin/activate  # Unix
.\venv\Scripts\activate   # Windows

# Install packages
pip install numpy pandas
pip freeze > requirements.txt
```

- **Pros:** Lightweight, built-in, fast
- **Cons:** Python binary management needed
- **Use when:** Simple Python projects, limited disk space

3. Docker Method

Setup - Production Choice

```
# Create Dockerfile
FROM python:3.9-slim
WORKDIR /app
# Install other binary dependencies if required
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .

# Build and run
docker build -t myproject .
docker run myproject
```

- **Pros:** Production-ready, fully isolated, reproducible
- **Cons:** More complex setup, sometimes worse dev experience
- **Use when:** Deploying to production, team projects

4. Poetry Method

Setup - Modern Python Choice

```
# Install Poetry
curl -sSL https://install.python-poetry.org | python3 -

# Start project
poetry new myproject
cd myproject

# Install and manage packages
poetry add numpy pandas
poetry install
poetry run python script.py
```

- **Pros:** Modern dependency resolution, good package management
- **Cons:** Resolve conflict forever

Best Practices

- Create one virtual environment per project
- Always include requirements.txt
- Inside the requirements.txt, local your version
- Use version control (.gitignore for venv folder)
- Document environment setup in README.md
- Regularly update dependencies

Project

Project Structure Overview

Basic Python Project Structure

```
my_project/
    .git/
    .gitignore
    README.md
    requirements.txt     # or pyproject.toml
    setup.py             # if publishing package
    my_project/          # main package directory
        __init__.py
        main.py
        utils.py
    tests/
        __init__.py
        test_main.py
```

Common Additional Directories

Optional but Recommended

- **docs/** - Documentation files
- **scripts/** - Utility scripts, tools
- **examples/** - Example usage
- **data/** - Project data files
- **notebooks/** - Jupyter notebooks
- **configs/** - Configuration files

Best Practices

- Keep related files together
- Separate code from data
- Clear import paths
- Follow naming conventions

Example .gitignore

Common Python Ignores

```
# Python
**pycache**/
*.py[cod]
*.so
build/
dist/
# Environment
venv/
env/
.env/
# IDE
.vscode/
.idea/
# Tests
.coverage
.pytest_cache/
# Project
data/raw/
*.log
```

Example: Docs2KG

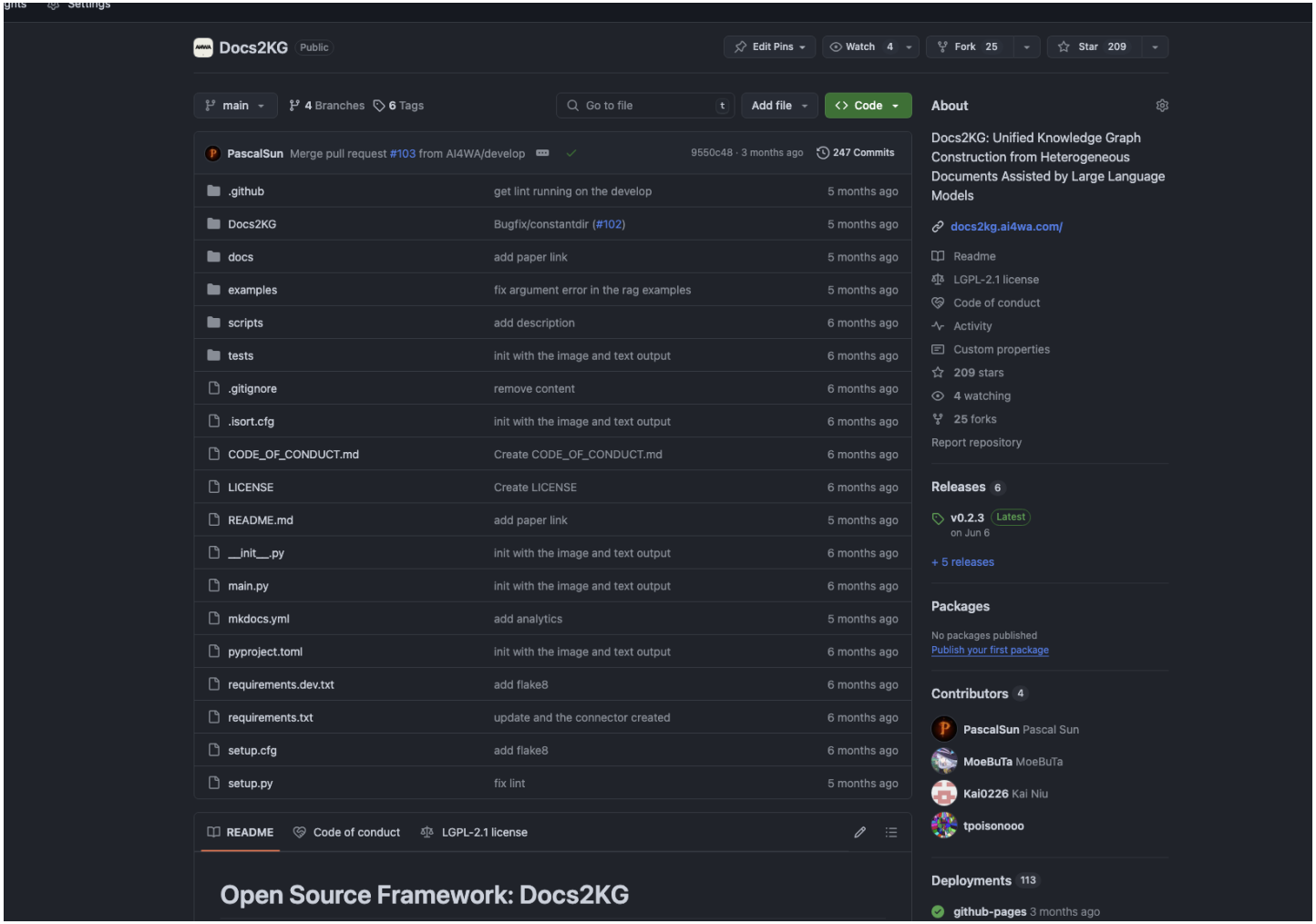

Figure: An example Python package project[1]

[1] https://github.com/AI4WA/Docs2KG

Development

How to Execute Your Code

Understanding if __name__ == "__main__"

```
# file: my_package/main.py
def main():
    print("Running main function")

if __name__ == "__main__":
    main()  # Only runs if file executed directly
```

Execution Methods

1. **Module-style execution:**

   ```
   python -m my_package.module.script
   ```

2. **IDE Integration:**
   - PyCharm: Right-click → Run 'script'
   - Configurable run configurations

3. **Test-Driven Development (TDD):**

   ```
   pytest tests/test_main.py
   ```

How to Execute your code?

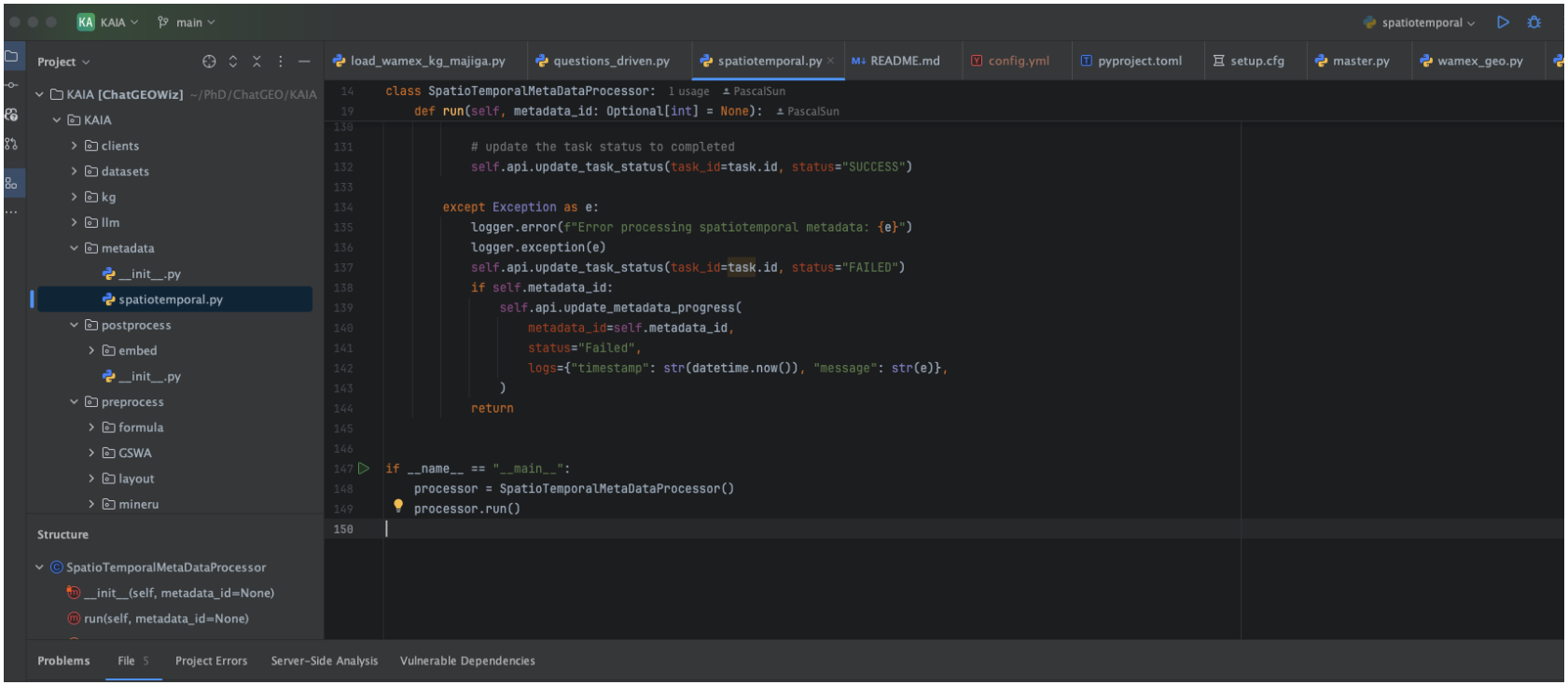

Figure: An example of how to execute your code?

Module Import Strategies

Project Structure

```
my_project/
   my_project/
       __init__.py
       main.py
       utils/
           __init__.py
           helper.py
```

Module Import Strategies

Relative Import

```
# in main.py
from .utils.helper import func
from ..utils import helper
```

- Complex with deep nesting
- Error-prone

Absolute Import

```
# in main.py
from my_project.utils.helper import func
from my_project.utils import helper
```

- Clear and explicit
- Easier to maintain

Import Best Practices

Relative Import Issues

- Hard to understand with deep nesting
- More prone to errors when moving files
- Can break if running scripts directly
- Confusing with multiple dot notation

Why Prefer Absolute Imports?

- Clear and explicit paths
- Easier to understand and maintain
- More reliable when refactoring
- IDE friendly (better autocompletion)

Recommendation

Always use absolute imports unless you have a specific reason not to.

Import Circular, and How to Solve It?

Problem Example

```
[python]
# a.py
from b import function_b

def function_a():
    return function_b() + 1

# b.py
from a import function_a

def function_b():
    return function_a() + 1
```

Import Circular, and How to Solve It?

Solutions:

1. **Restructure Code**

```python
# common.py
def shared_logic():
    pass
```

2. **Import Inside Function**

```python
def function_b():
    from a import function_a  # Local import
    return function_a() + 1
```

3. **Use Dependency Injection**

```python
def function_b(func_a=None):
    if func_a is None:
        from a import function_a
        func_a = function_a
    return func_a() + 1
```

Use Class, Not Functions

We taught regarding how you write the code in function.
However, object oriented is the best practice for large projects.

Example: Functions vs Class Approach

```
[python]
# Function approach
def process_data(data):
    return data * 2

# Class approach
class DataProcessor:
    def __init__(self, data):
        self.data = data

    def process(self):
        return self.data * 2

# Usage
processor = DataProcessor(10)
result = processor.process()
```

Python @staticmethod

Static methods don't require access to class or instance state.
They are utility functions that live in the class namespace.

Example: Using @staticmethod

```
[python]
class Calculator:
    @staticmethod
    def add(x, y):
        # No 'self' or 'cls' parameter needed
        return x + y

    @staticmethod
    def is_positive(num):
        return num > 0

# Usage
result = Calculator.add(5, 3)      # Returns 8
is_pos = Calculator.is_positive(5) # Returns True
```

- No implicit first argument (no self, no cls)
- Cannot access/modify class or instance attributes
- Best for utility functions related to class purpose

Python @classmethod

Class methods receive the class itself as the first argument.
They are commonly used for alternative constructors or factory methods.

Example: Using @classmethod

```
[python]
class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    @classmethod
    def from_string(cls, date_str):
        # cls is the class itself
        year, month, day = map(int, date_str.split('-'))
        return cls(year, month, day)

# Usage
date = Date.from_string('2024-11-28')
```

- Receives class as first argument (cls)
- Can access and modify class state

Code Style Tools: isort, flake8, and black

Tool Purposes & Relationships

These tools work together to ensure consistent, readable, and maintainable Python code:

- **isort**: Organizes and sorts imports systematically
- **flake8**: Enforces PEP 8 style guide and checks code quality
- **black**: Formats code with opinionated, consistent styling

isort

- Groups imports by type
- Sorts alphabetically
- Separates standard, third-party, local
- Removes unused imports

flake8

- Checks line length
- Detects syntax errors
- Finds style issues
- Reports complexity

black

- Enforces formatting
- Removes style choices
- Makes code deterministic
- Focuses on readability

Code Style

Recommended Usage Order

1. Run **isort** first to organize imports
2. Apply **black** to format the code
3. Use **flake8** to catch remaining style issues

CI/CD Integration

- Tools integrated into CI/CD pipeline
- Automated code style checks on each commit/PR
- Fails builds that don't meet standards
- Maintains consistent code quality across team

Why Loguru over Default Logger?

Default Logger

- Complex configuration
- Basic formatting
- No color support
- Manual exception handling

Loguru

- Zero configuration
- Better formatting
- Colored output
- @logger.catch decorator

```python
# Default logger - Complex setup
import logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Loguru - Simple setup
from loguru import logger

```

CLI

Introduction to CLI

What is CLI?

- Command Line Interface for program interaction
- Two main approaches:
  - Script execution: `python script.py`
  - Direct CLI: `mycli`

Basic - sys.argv

- Access arguments through `sys.argv` list
- `argv[0]`: script name, `argv[1:]`: arguments

```python
import sys

def main():
    print(f"Script: {sys.argv[0]}")
    print(f"Arguments: {sys.argv[1:]}")

if __name__ == '__main__':
    main()

```

Usage

```
$ python script.py arg1 arg2

```

Intermediate - argparse

- Standard library for argument parsing
- Automatic help and validation

```python
import argparse

parser = argparse.ArgumentParser(description='Example')
parser.add_argument('--sum', dest='accumulate',
                    action='store_const',
                    const=sum, default=max,
                    help='sum the integers')
parser.add_argument('integers', type=int, nargs='+')
args = parser.parse_args()

```

Usage

```
$ python script.py --sum 1 2 3

```

Advanced - Click Framework

- Modern CLI framework with decorator interface
- Direct command execution after installation

```python
import click

@click.command()
@click.option('--count', default=1)
@click.option('--name', prompt='Your name')
def hello(count, name):
    for x in range(count):
        click.echo(f'Hello {name}!')

```

Usage

```
$ mycli --count=3 --name=John

```

Making CLI Executable

```
from setuptools import setup

setup(
    name='mycli',
    version='0.1',
    py_modules=['mycli'],
    install_requires=['click'],
    entry_points={
        'console_scripts': ['mycli=mycli:hello'],
    },
)

```

Installation

```
$ pip install --editable .

```

Cli Summary

Script Execution

- sys.argv: Basic parsing
- argparse: Structured CLI
- Requires `python` command

Direct CLI

- Click: Modern interface
- Direct execution
- Professional distribution

Doc

Why Documentation?

- Documentation = Communication
- Code tells what it does, documentation tells why
- Modern approach: Documentation lives with code
- Automatic generation reduces maintenance burden

Writing Documentation in Code

Example: NumPy Style

```
def calculate_metric(data: np.ndarray) -> float:
    """Calculate performance metric from input data.

    Parameters
    ----------
    data : np.ndarray
        Input time series data

    Returns
    -------
    float
        Computed metric value
    """
```

Writing Documentation in Code

Example: Google Style

```
def calculate_metric(data: np.ndarray) -> float:
    """Calculate performance metric from input data.

    Args:
        data (np.ndarray): Input time series data

    Returns:
        float: Computed metric value
    """
```

MkDocs: Modern Documentation Generator

- Why MkDocs?
  - Modern, clean appearance
  - Markdown-based (easier than RST)
  - Material theme support
  - Fast build times
- Key features:
  - Auto-navigation
  - Search functionality
  - Code highlighting
  - Math equation support
- Demo
  - `https://docs2kg.ai4wa.com/`
  - `https://openomnidocs.ai4wa.com/`

Hosting via GitHub Pages through CI/CD

Workflow

```
name: docs
on:
  push:
    branches:
      - main
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Python
        uses: actions/setup-python@v4
      - run: pip install mkdocs-material
      - run: mkdocs gh-deploy --force
```

Testing

Python Testing Overview

Testing is a crucial part of package development that ensures code reliability.

Key Testing Principles

- Write tests before or alongside code (Test-Driven Development)
- Each test should be independent and isolated
- Test both valid and invalid cases
- Keep tests simple and focused
- Aim for high test coverage

Project Structure

```
my_package/
   src/
      my_package/
         __init__.py
         module.py
   tests/
      __init__.py
      test_module.py
```

pytest Basics

pytest is the most popular testing framework for Python.

Simple Test Example

```
[python]
# src/my_package/calculator.py
class Calculator:
    def add(self, x, y):
        return x + y

# tests/test_calculator.py
def test_add():
    calc = Calculator()
    assert calc.add(2, 3) == 5
    assert calc.add(-1, 1) == 0
    assert calc.add(0, 0) == 0
```

- Test files and functions should start with "test_"
- Uses simple assert statements
- Automatically discovers and runs tests

pytest Features

pytest provides powerful features for testing.

Fixtures and Parametrization

```
[python]
import pytest

@pytest.fixture
def calculator():
    return Calculator()

@pytest.mark.parametrize("x, y, expected", [
    (2, 3, 5),
    (-1, 1, 0),
    (0, 0, 0)
])
def test_add_params(calculator, x, y, expected):
    assert calculator.add(x, y) == expected
```

- Fixtures: Reusable test resources/setup
- Parametrize: Test multiple inputs easily
- Marks: Categorize and select tests

Testing Best Practices

Follow these practices for effective testing.

Example: Testing Exceptions

```
[python]
def test_division_by_zero():
    calc = Calculator()
    with pytest.raises(ValueError) as exc_info:
        calc.divide(10, 0)
    assert str(exc_info.value) == "Cannot divide by zero"

@pytest.mark.skip(reason="feature not implemented yet")
def test_future_feature():
    pass
```

- Test edge cases and exceptions
- Use meaningful test names
- One assertion per test (when possible)
- Skip or mark tests appropriately
- Keep tests fast and deterministic

Test Coverage

Monitor your test coverage to ensure thorough testing.

Using pytest-cov

```
# Install pytest-cov
pip install pytest-cov

# Run tests with coverage
pytest --cov=my_package tests/

# Generate HTML coverage report
pytest --cov=my_package --cov-report=html tests/
```

- Aim for high coverage (typically >80%)
- Coverage isn't everything - focus on quality
- Test critical paths thoroughly
- Use branch coverage for better insight
- Include coverage in CI/CD pipeline

Distribution

Package Distribution Overview

Different distribution strategies serve different purposes.

Distribution Types

- Open Source: Public PyPI release
- Private Package: Git repository or private PyPI
- Service: Containerized application

Package Configuration (setup.py)

```python
import os
import re

from setuptools import find_packages, setup

setup(
    name="Docs2KG",
    author="AI4WA",
    author_email="admin@ai4wa.com",
    description="Unified Knowledge Graph Construction from Heterogeneous Documents Assisted by Large Language Models",
    long_description=long_description,
    long_description_content_type="text/markdown",
    version=version,
    packages=find_packages(),  # Adjust the location where setuptools looks for packages
    include_package_data=True,  # To include other types of files specified in MANIFEST.in or found in your packages
    install_requires=read_requirements(),
    python_requires=">=3.8",  # Specify your Python version compatibility
    classifiers=[
        # Classifiers help users find your project by categorizing it
        "Development Status :: 3 - Alpha",
        "Intended Audience :: Developers",
        "Intended Audience :: Science/Research",
        # license: GNU LESSER GENERAL PUBLIC LICENSE
        "License :: OSI Approved :: GNU Lesser General Public License v3 (LGPLv3)",
        # topic
        "Topic :: Scientific/Engineering :: Artificial Intelligence",
        "Programming Language :: Python :: 3.8",
        "Programming Language :: Python :: 3.9",
        "Programming Language :: Python :: 3.10",
    ],
    url="https://docs2kg.ai4wa.com",
    project_urls={
        "Source": "https://github.com/AI4WA/Docs2KG",
    },
)

```

Figure: Example of setup.py

Publishing to PyPI

```yaml
name: Publish Docs2KG

on:
  push:
    tags:
      - 'v*.*.*'

jobs:
  publish:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.x'

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install setuptools wheel twine

      - name: Build package
        run: python setup.py sdist bdist_wheel

      - name: Publish package
        env:
          TWINE_USERNAME: __token__
          TWINE_PASSWORD: ${{ secrets.PYPI_API_TOKEN }}
        run: twine upload dist/*
```

Figure: CI/CD publish to pypi

- Register on PyPI (pypi.org)
- Test on test.pypi.org first
- Set up CI/CD for releases

Private Package Distribution

Distributing closed-source packages within organizations.

Install from Git

```
[bash]
# Install directly from Git
pip install git+ssh://git@github.com/org/package.git

# Or in requirements.txt
my-package @ git+ssh://git@github.com/org/package.git@v1.0.0

# Private PyPI configuration
[global]
index-url = https://private.pypi.org/simple
```

- Use private Git repositories
- Set up private PyPI server (e.g., DevPI)
- Implement access controls
- Version control with Git tags

Service-based Distribution

Packaging your code as a standalone service.

Dockerfile Example

```
[docker]
FROM python:3.9-slim

WORKDIR /app
COPY . .
RUN pip install .

EXPOSE 8000
ENTRYPOINT ["python", "-m", "my_package.server"]

# Build and run
docker build -t my-service .
docker run -p 8000:8000 my-service
```

- Containerize with Docker
- Define clear service interfaces
- Include configuration management
- Plan for scalability

Deployment Options

Different deployment strategies for different needs.

Serverless

- AWS Lambda
- Google Cloud Functions
- Azure Functions
- Event-driven architecture
- Pay-per-use model

Traditional Hosting

- Docker containers
- Kubernetes clusters
- Virtual machines
- Dedicated servers
- Continuous availability

Distribution Checklist

Essential considerations for package distribution.

Documentation

- Installation instructions
- Usage examples
- API documentation
- Deployment guides
- Troubleshooting tips

Quality Assurance

- Automated tests
- Version compatibility
- Security considerations
- Performance benchmarks
- Dependencies management

Docker GPU

Docker with GPU for Python

Docker containers can leverage GPU capabilities for ML/DL applications.

Prerequisites

- NVIDIA GPU drivers installed
- NVIDIA Container Toolkit (nvidia-docker2)
- Docker installed

Base Images

- nvidia/cuda: Basic CUDA support
- pytorch/pytorch: PyTorch with CUDA
- tensorflow/tensorflow: TensorFlow with GPU

Dockerfile Example

GPU Dockerfile

```
FROM nvidia/cuda:11.8.0-runtime-ubuntu22.04

WORKDIR /app

# System dependencies
RUN apt-get update && apt-get install -y \
    python3-pip \
    && rm -rf /var/lib/apt/lists/*

# Python packages
COPY requirements.txt .
RUN pip3 install --no-cache-dir -r requirements.txt

# Copy package
COPY . .
RUN pip3 install -e .
# Test GPU availability
CMD ["python3", "-c", "import torch; \
    print(f'GPU Available: {torch.cuda.is_available()}, \
    Device Count: {torch.cuda.device_count()}')"]

```

Dockerfile Example

Run Container

```
docker build -t my-gpu-app .
docker run --gpus all my-gpu-app

```

License

Why License Matters

- No license = No open source
- Affects how others can use your code
- Impacts project adoption and contributions
- Legal implications for both maintainers and users

Popular Licenses

Permissive

- MIT License
  - Simple and short
  - Allows commercial use
  - Minimal restrictions
- Apache 2.0
  - Patent protection
  - Explicit trademark rights

Copyleft

- GPL v3
  - Must share modifications
  - Strong copyleft
- LGPL
  - Library-friendly
  - Allows proprietary linking

Important Considerations

Before Choosing

- Check dependencies' licenses
- Consider commercial compatibility
- Review company policies
- Understand license obligations

Key Questions

- Do you want to allow commercial use?
- Should derivatives be open source?
- Need patent protection?
- Corporate requirements?

Q&A

Homework

- Setup your package with instruction above
- and implement some modules for your project.

Q&A

Questions?

Feel free to ask anything