Python Project Organization Guide: Conda + Git

Table of Contents

Project Structure Overview

Recommended Directory Structure

```
voc-validation-pipeline/
├── README.md
├── .gitignore
├── environment.yml          # Conda environment specification
├── setup.py                 # Makes project pip-installable
├── pyproject.toml           # Modern Python packaging (alternative to setup.py)
├── requirements.txt         # Pip dependencies (backup/supplement to conda)
├── .env.example             # Environment variable template
│
├── data/                    # Local data (usually gitignored)
│   ├── raw/                 # Original, immutable data
│   ├── interim/             # Intermediate transformed data
│   ├── processed/           # Final, canonical datasets
│   └── external/            # Data from third party sources
│
├── notebooks/               # Jupyter notebooks for exploration
│   ├── 01_exploratory_analysis.ipynb
│   └── 02_validation_testing.ipynb
│
├── src/                     # Source code for the project
│   └── voc_validation/      # Main package (use underscores, not hyphens)
│       ├── __init__.py
│       ├── config.py        # Configuration management
│       ├── data/           # Data loading and processing
│       │   ├── __init__.py
│       │   ├── loaders.py
│       │   └── processors.py
│       ├── validation/     # Validation logic
│       │   ├── __init__.py
│       │   ├── qc_checks.py
│       │   └── validators.py
│       ├── pipeline/       # Pipeline orchestration
│       │   ├── __init__.py
│       │   └── pipeline.py
│       └── utils/          # Utility functions
│           ├── __init__.py
│           └── helpers.py
│
├── tests/                  # Unit and integration tests
│   ├── __init__.py
│   ├── test_data_loaders.py
│   ├── test_validators.py
│   └── fixtures/           # Test data
│
├── scripts/                # Standalone scripts
│   ├── run_pipeline.py
│   └── generate_report.py
│
├── docs/                   # Documentation
│   ├── methodology.md
│   └── api_reference.md
│
└── outputs/                # Generated outputs (usually gitignored)
    ├── reports/
    ├── figures/
    └── models/
```

Understanding Python Packages and Modules

Module vs Package vs Distribution

Module: A single Python file (e.g., `validators.py`)

- Contains functions, classes, variables

- Imported with `import validators` or `from validators import my_function`

Package: A directory containing `__init__.py` and other modules

- Allows hierarchical structuring
- Example: `voc_validation/` is a package, `voc_validation/data/` is a subpackage

Distribution: What you install with pip/conda

- Can contain multiple packages
- Defined in `setup.py` or `pyproject.toml`

How Import Paths Work

Python finds modules/packages in paths listed in `sys.path`, which includes:

1. The directory containing the script being run
2. PYTHONPATH environment variable directories
3. Site-packages directories where pip/conda install packages
4. Standard library directories

Making Your Project Importable

Option 1: Editable Install (Recommended for Development)

```bash
# From project root directory
pip install -e .
```

This makes your package importable from anywhere without copying files. Changes to source code are immediately reflected.

Option 2: Add to PYTHONPATH (Quick but not recommended)

```bash
export PYTHONPATH="${PYTHONPATH}:/path/to/voc-validation-pipeline/src"
```

Why Use `src/` Layout?

The `src/` layout prevents accidentally importing from the local directory instead of the installed package, ensuring tests run against the installed version.

Import Patterns

```python
# Absolute imports (preferred)
from voc_validation.data.loaders import load_autogc_data
from voc_validation.validation.validators import VOCValidator

# Relative imports (within package only)
# In voc_validation/pipeline/pipeline.py:
from ..data.loaders import load_autogc_data  # Go up one level
from ..validation import validators  # Import submodule

# Avoid wildcard imports
# Bad:
from voc_validation.validation import *
# Good:
from voc_validation.validation import VOCValidator, validate_calibration
```

Package Initialization

Use `__init__.py` to control what's exported:

```python
# src/voc_validation/__init__.py
from .validation.validators import VOCValidator
from .data.loaders import load_autogc_data

__version__ = "0.1.0"
__all__ = ["VOCValidator", "load_autogc_data"]
```

This allows users to do:

```python
from voc_validation import VOCValidator  # Instead of from voc_validation.validation.validators
```

Conda Environment Management

Why Conda for Data Science?

- Manages both Python packages and system-level dependencies (C libraries, etc.)
- Better for scientific computing packages (numpy, scipy, pandas)
- Isolates environments completely
- Handles binary dependencies that pip struggles with

Creating environment.yml

```yaml
name: voc-validation
channels:
  - conda-forge
  - defaults
dependencies:
  - python=3.11
  - numpy>=1.24
  - pandas>=2.0
  - scipy>=1.10
  - matplotlib>=3.7
  - seaborn>=0.12
  - jupyter
  - pytest>=7.0
  - black  # Code formatter
  - flake8  # Linter
  - pip
  - pip:
    # Packages only available via pip
    - python-dotenv>=1.0
    - your-special-package==1.2.3
```

Environment Management Commands

```bash
```

```
# Create environment from file
conda env create -f environment.yml

# Activate environment
conda activate voc-validation

# Update environment from file
conda env update -f environment.yml --prune

# Export current environment (for sharing exact versions)
conda env export > environment_lock.yml

# Deactivate environment
conda deactivate

# Remove environment
conda env remove -n voc-validation

# List environments
conda env list

# Install new package and update environment.yml manually
conda install -c conda-forge some-package
```

Best Practices for Conda

1. **Pin Python version** but be flexible with package versions (use `>=` not `==`)
2. **Use conda-forge** channel for most packages (better maintained)
3. **Keep environment.yml minimal** - only list direct dependencies
4. **Create environment_lock.yml** for exact reproducibility
5. **One environment per project** - avoid modifying base environment
6. **Use pip within conda** sparingly - conda dependencies first, pip for gaps

Conda vs Requirements.txt

Keep both for flexibility:

```bash
```

```
# Generate requirements.txt from conda environment
pip list --format=freeze > requirements.txt
```

Or maintain manually:

```txt
# requirements.txt - for pip-only environments
pandas>=2.0.0
numpy>=1.24.0
scipy>=1.10.0
python-dotenv>=1.0.0
```

Git Best Practices

Essential .gitignore

```gitignore
```

```
# Python
__pycache__/
*.py[cod]
*$py.class
*.so
.Python
env/
venv/
ENV/
build/
develop-eggs/
dist/
downloads/
eggs/
.eggs/
lib/
lib64/
parts/
sdist/
var/
wheels/
*.egg-info/
.installed.cfg
*.egg

# Jupyter Notebook
.ipynb_checkpoints
*.ipynb_checkpoints/

# Conda
*.conda
*.tar.bz2

# Environment files
.env
.venv
environment_lock.yml  # Only commit environment.yml

# Data (commit small examples, not full datasets)
data/raw/*
data/interim/*
data/processed/*
!data/raw/.gitkeep
```

```
!data/raw/sample_data.csv
data/external/*

# Outputs
outputs/
*.log

# IDEs
.vscode/
.idea/
*.swp
*.swo
.DS_Store

# Testing
.pytest_cache/
.coverage
htmlcov/
```

Git Workflow for Data Projects

Initial Setup

```bash
# Initialize repository
git init
git add README.md .gitignore environment.yml
git commit -m "Initial commit: project structure"

# Create and switch to development branch
git checkout -b develop
```

Branching Strategy

```
main (stable releases)
 └── develop (integration branch)
     ├── feature/data-loader
     ├── feature/voc-validation
     └── feature/pipeline-orchestration
```

Common Workflow

```bash
# Start new feature
git checkout develop
git pull origin develop
git checkout -b feature/autogc-parser

# Make changes and commit frequently
git add src/voc_validation/data/loaders.py
git commit -m "Add AutoGC data parser for VOC compounds"

# Push feature branch
git push -u origin feature/autogc-parser

# Merge feature when complete
git checkout develop
git merge feature/autogc-parser
git push origin develop

# Delete feature branch
git branch -d feature/autogc-parser
git push origin --delete feature/autogc-parser
```

Commit Message Best Practices

Use conventional commits format:

```
type(scope): description

[optional body]

[optional footer]
```

Types:

- `feat`: New feature
- `fix`: Bug fix
- `docs`: Documentation changes
- `style`: Code style changes (formatting)
- `refactor`: Code refactoring
- `test`: Adding tests

- `chore`: Maintenance tasks

Examples:

```bash
git commit -m "feat(validation): add calibration curve validation"
git commit -m "fix(data): handle missing timestamps in AutoGC files"
git commit -m "docs: add VOC compound reference table"
```

Handling Large Data Files with Git LFS

```bash
# Install Git LFS
conda install git-lfs
git lfs install

# Track large file types
git lfs track "*.csv"
git lfs track "*.nc"
git lfs track "data/external/*"

# This creates/updates .gitattributes
git add .gitattributes
git commit -m "chore: configure Git LFS for data files"
```

Example: VOC Data Validation Pipeline

Project Setup

```bash
```

```
# Create project directory
mkdir voc-validation-pipeline
cd voc-validation-pipeline

# Initialize git
git init
git checkout -b develop

# Create directory structure
mkdir -p src/voc_validation/{data,validation,pipeline,utils}
mkdir -p tests/{data,validation,pipeline}
mkdir -p {notebooks,scripts,docs,data/{raw,interim,processed,external},outputs/{reports,figures}}

# Create __init__.py files
touch src/voc_validation/__init__.py
touch src/voc_validation/{data,validation,pipeline,utils}/__init__.py
touch tests/__init__.py
```

setup.py Example

```python
```

```python
# setup.py
from setuptools import setup, find_packages

setup(
    name="voc-validation",
    version="0.1.0",
    description="Data pipeline for VOC validation from AutoGC",
    author="Your Name",
    author_email="your.email@example.com",
    packages=find_packages(where="src"),
    package_dir={"": "src"},
    python_requires=">=3.9",
    install_requires=[
        "numpy>=1.24",
        "pandas>=2.0",
        "scipy>=1.10",
        "python-dotenv>=1.0",
    ],
    extras_require={
        "dev": ["pytest>=7.0", "black", "flake8", "jupyter"],
    },
)
```

Modern pyproject.toml Alternative

```
toml
```

```toml
# pyproject.toml
[build-system]
requires = ["setuptools>=61.0", "wheel"]
build-backend = "setuptools.build_meta"

[project]
name = "voc-validation"
version = "0.1.0"
description = "Data pipeline for VOC validation from AutoGC"
readme = "README.md"
requires-python = ">=3.9"
dependencies = [
    "numpy>=1.24",
    "pandas>=2.0",
    "scipy>=1.10",
    "python-dotenv>=1.0",
]

[project.optional-dependencies]
dev = ["pytest>=7.0", "black", "flake8", "jupyter"]

[tool.setuptools.packages.find]
where = ["src"]
```

Configuration Management

```
python
```

```python
# src/voc_validation/config.py
from pathlib import Path
from dotenv import load_dotenv
import os

# Load environment variables
load_dotenv()

# Project paths
PROJECT_ROOT = Path(__file__).parent.parent.parent
DATA_DIR = PROJECT_ROOT / "data"
RAW_DATA_DIR = DATA_DIR / "raw"
PROCESSED_DATA_DIR = DATA_DIR / "processed"
OUTPUT_DIR = PROJECT_ROOT / "outputs"

# VOC compound list
VOC_COMPOUNDS = [
    "benzene", "toluene", "ethylbenzene", "xylenes",
    "acetone", "isopropanol", "methanol"
]

# Validation thresholds
CALIBRATION_R2_THRESHOLD = 0.995
MAX_BLANK_CONCENTRATION = 0.1  # ppb
MAX_RSD_PERCENT = 15.0

# AutoGC settings
AUTOGC_SAMPLING_RATE = int(os.getenv("AUTOGC_SAMPLING_RATE", "60"))  # seconds
```

Example Data Loader

```
python
```

```python
# src/voc_validation/data/loaders.py
import pandas as pd
from pathlib import Path
from typing import Union
from ..config import RAW_DATA_DIR, VOC_COMPOUNDS

def load_autogc_data(
    filepath: Union[str, Path],
    compounds: list[str] = None
) -> pd.DataFrame:
    """
    Load AutoGC VOC data from CSV file.

    Parameters
    ----------
    filepath : str or Path
        Path to AutoGC CSV file
    compounds : list of str, optional
        List of compounds to load. If None, loads all VOC_COMPOUNDS

    Returns
    -------
    pd.DataFrame
        DataFrame with timestamp index and compound concentrations
    """
    filepath = Path(filepath)
    if not filepath.is_absolute():
        filepath = RAW_DATA_DIR / filepath

    if compounds is None:
        compounds = VOC_COMPOUNDS

    # Load data
    df = pd.read_csv(filepath, parse_dates=["timestamp"])
    df.set_index("timestamp", inplace=True)

    # Validate expected columns exist
    missing_compounds = set(compounds) - set(df.columns)
    if missing_compounds:
        raise ValueError(f"Missing compounds in data: {missing_compounds}")

    return df[compounds]
```

Example Validator

python

```python
# src/voc_validation/validation/validators.py
import pandas as pd
import numpy as np
from typing import Dict, Tuple
from ..config import (
    CALIBRATION_R2_THRESHOLD,
    MAX_BLANK_CONCENTRATION,
    MAX_RSD_PERCENT
)

class VOCValidator:
    """Validator for VOC measurement quality control."""

    def __init__(self, strict_mode: bool = False):
        self.strict_mode = strict_mode
        self.validation_results = {}

    def validate_calibration(
        self,
        standards: pd.DataFrame,
        concentrations: np.ndarray
    ) -> Dict[str, bool]:
        """
        Validate calibration curves for each compound.

        Parameters
        ----------
        standards : pd.DataFrame
            Measured peak areas for calibration standards
        concentrations : np.ndarray
            Known concentrations for calibration standards

        Returns
        -------
        dict
            Validation results for each compound
        """
        results = {}

        for compound in standards.columns:
            peak_areas = standards[compound].values

            # Linear regression
            coeffs = np.polyfit(concentrations, peak_areas, 1)
            predicted = np.polyval(coeffs, concentrations)

            # Calculate R²
            ss_res = np.sum((peak_areas - predicted) ** 2)
            ss_tot = np.sum((peak_areas - np.mean(peak_areas)) ** 2)
            r2 = 1 - (ss_res / ss_tot)

            results[compound] = {
                "passed": r2 >= CALIBRATION_R2_THRESHOLD,
                "r2": r2,
                "slope": coeffs[0],
                "intercept": coeffs[1]
            }

        self.validation_results["calibration"] = results
        return results

    def validate_blanks(self, blank_data: pd.DataFrame) -> Dict[str, bool]:
        """Validate blank measurements are below threshold."""
        results = {}

        for compound in blank_data.columns:
            mean_blank = blank_data[compound].mean()
            results[compound] = {
                "passed": mean_blank <= MAX_BLANK_CONCENTRATION,
                "mean_concentration": mean_blank
            }

        self.validation_results["blanks"] = results
        return results

    def validate_precision(
        self,
        replicate_data: pd.DataFrame
    ) -> Dict[str, bool]:
        """Validate measurement precision using replicate samples."""
        results = {}

        for compound in replicate_data.columns:
            values = replicate_data[compound]
            mean_val = values.mean()
            std_val = values.std()
            rsd = (std_val / mean_val) * 100 if mean_val > 0 else np.inf
            results[compound] = {
                "passed": rsd <= MAX_RSD_PERCENT,
                "rsd_percent": rsd,
                "mean": mean_val,
                "std": std_val
            }

        self.validation_results["precision"] = results
        return results

    def get_summary_report(self) -> str:
        """Generate summary report of all validations."""
        report = ["VOC Validation Summary", "=" * 50]

        for validation_type, compounds in self.validation_results.items():
            report.append(f"\n{validation_type.upper()}:")
            for compound, results in compounds.items():
                status = "✓ PASS" if results["passed"] else "✗ FAIL"
                report.append(f"  {compound:20s}: {status}")

        return "\n".join(report)
```

Example Pipeline Script

```
python
```

```python
# scripts/run_pipeline.py
"""Run complete VOC validation pipeline."""
import sys
from pathlib import Path

# Add src to path if not installed
sys.path.insert(0, str(Path(__file__).parent.parent / "src"))

from voc_validation.data.loaders import load_autogc_data
from voc_validation.validation.validators import VOCValidator
from voc_validation.config import OUTPUT_DIR
import pandas as pd

def main():
    # Load data
    print("Loading AutoGC data...")
    sample_data = load_autogc_data("sample_measurements.csv")
    blank_data = load_autogc_data("blanks.csv")
    calibration_data = load_autogc_data("calibration_standards.csv")

    # Initialize validator
    validator = VOCValidator(strict_mode=True)

    # Run validations
    print("Validating calibration curves...")
    cal_concentrations = [0, 1, 5, 10, 50, 100]  # ppb
    validator.validate_calibration(calibration_data, cal_concentrations)

    print("Validating blank measurements...")
    validator.validate_blanks(blank_data)

    print("Validating measurement precision...")
    replicates = sample_data.iloc[:5]  # First 5 samples are replicates
    validator.validate_precision(replicates)

    # Generate report
    report = validator.get_summary_report()
    print("\n" + report)

    # Save report
    report_path = OUTPUT_DIR / "reports" / "validation_report.txt"
    report_path.parent.mkdir(parents=True, exist_ok=True)
    with open(report_path, "w") as f:
        f.write(report)

    print(f"\nReport saved to: {report_path}")

if __name__ == "__main__":
    main()
```

Testing Example

```
python
```

```python
# tests/test_validators.py
import pytest
import numpy as np
import pandas as pd
from voc_validation.validation.validators import VOCValidator

@pytest.fixture
def sample_calibration_data():
    """Create sample calibration data for testing."""
    concentrations = np.array([0, 1, 5, 10, 50, 100])
    # Perfect linear response: area = 1000 * concentration
    benzene = 1000 * concentrations + np.random.normal(0, 10, len(concentrations))

    df = pd.DataFrame({
        "benzene": benzene,
        "toluene": 1200 * concentrations + np.random.normal(0, 15, len(concentrations))
    })
    return df, concentrations

def test_calibration_validation_pass(sample_calibration_data):
    """Test calibration validation with good data."""
    data, concentrations = sample_calibration_data
    validator = VOCValidator()

    results = validator.validate_calibration(data, concentrations)

    assert results["benzene"]["passed"] is True
    assert results["benzene"]["r2"] > 0.99
    assert results["toluene"]["passed"] is True

def test_blank_validation():
    """Test blank validation."""
    blank_data = pd.DataFrame({
        "benzene": [0.05, 0.03, 0.04, 0.06],  # Below threshold
        "toluene": [0.15, 0.20, 0.18, 0.22]   # Above threshold
    })

    validator = VOCValidator()
    results = validator.validate_blanks(blank_data)

    assert results["benzene"]["passed"] is True
    assert results["toluene"]["passed"] is False
```

Advanced Topics

Package Installation Modes

```bash
# Development mode - changes reflect immediately
pip install -e .

# Install with optional dependencies
pip install -e ".[dev]"

# Install from git repository
pip install git+https://github.com/yourusername/voc-validation.git

# Install specific version
pip install voc-validation==0.1.0
```

Managing Multiple Related Projects

For larger systems with multiple related projects:

```
ambient-air-monitoring/
├── voc-validation/         # Separate git repo
├── data-acquisition/       # Separate git repo
└── reporting-dashboard/    # Separate git repo
```

Each can be its own package:

```python
# In reporting-dashboard project
from voc_validation import VOCValidator
from data_acquisition import AutoGCConnector
```

Pre-commit Hooks for Code Quality

```yaml
```

```yaml
# .pre-commit-config.yaml
repos:
  - repo: https://github.com/psf/black
    rev: 23.3.0
    hooks:
      - id: black
        language_version: python3.11

  - repo: https://github.com/pycqa/flake8
    rev: 6.0.0
    hooks:
      - id: flake8
        args: [--max-line-length=88]

  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.4.0
    hooks:
      - id: trailing-whitespace
      - id: end-of-file-fixer
      - id: check-yaml
      - id: check-added-large-files
        args: [--maxkb=1000]
```

Install with:

```bash
pip install pre-commit
pre-commit install
```

Documentation with Sphinx

```bash
```

```
# Install documentation tools
conda install sphinx sphinx_rtd_theme

# Initialize docs
cd docs
sphinx-quickstart

# Build documentation
make html
```

Continuous Integration Example

```
yaml
```

```
# .github/workflows/tests.yml
name: Tests

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: ["3.9", "3.10", "3.11"]

    steps:
      - uses: actions/checkout@v3

      - name: Setup Conda
        uses: conda-incubator/setup-miniconda@v2
        with:
          python-version: ${{ matrix.python-version }}
          environment-file: environment.yml
          activate-environment: voc-validation

      - name: Install package
        shell: bash -l {0}
        run: pip install -e ".[dev]"

      - name: Run tests
        shell: bash -l {0}
        run: pytest tests/ -v --cov=voc_validation
```

Environment Variables for Configuration

```python
# .env.example (commit this)
AUTOGC_SAMPLING_RATE=60
DATA_PATH=/path/to/data
LOG_LEVEL=INFO
DATABASE_URL=postgresql://user:pass@localhost/vocdata

# .env (gitignored, create locally)
# Copy from .env.example and fill in actual values
```

Load in your code:

```python
from dotenv import load_dotenv
import os

load_dotenv()
sampling_rate = int(os.getenv("AUTOGC_SAMPLING_RATE", "60"))
```

Quick Reference Commands

Daily Workflow

```bash
# Start working
conda activate voc-validation
git checkout develop
git pull origin develop
git checkout -b feature/new-validation-check

# Make changes, test, commit
pytest tests/
git add .
git commit -m "feat(validation): add new validation check"

# Push and merge
git push -u origin feature/new-validation-check
# Create PR on GitHub/GitLab
# After merge:
git checkout develop
git pull origin develop
git branch -d feature/new-validation-check
```

Troubleshooting

```bash
```

```
# Package not found after install
pip install -e .  # Reinstall in editable mode
python -c "import sys; print(sys.path)"  # Check Python path

# Import errors
python -c "import voc_validation; print(voc_validation.__file__)"  # Verify install

# Conda environment issues
conda deactivate
conda activate voc-validation
which python  # Verify correct Python

# Git conflicts
git status
git diff
# Resolve conflicts in editor, then:
git add resolved_file.py
git commit
```

This guide provides a solid foundation for organizing your VOC validation pipeline project. Adapt the structure to your specific needs as the project evolves!