canVODpy

Nicolas Bader

Wednesday 7th January, 2026

Curvenote

1 Documentation

1.1 Documentation Home

1.1.1 canVODpy Documentation

Welcome to the complete technical documentation for canVODpy, a modern monorepo for GNSS vegetation optical depth analysis.

What is canVODpy? canVODpy is a **modular Python package ecosystem** for analyzing GNSS (Global Navigation Satellite System) signals to estimate vegetation optical depth (VOD). The project uses a **monorepo architecture** with **namespace packages** to provide both modularity and a unified API.

Who Should Read This? This documentation is for:

- **New developers** joining the project
- **Contributors** wanting to understand the architecture
- **Users** interested in how the package works
- **Anyone** curious about modern Python monorepo development

No prior knowledge assumed! We explain everything from the ground up.

What You'll Learn

1. Architecture Overview **Read: Architecture →**
Understand the big picture:

- Why a monorepo instead of separate repos?
- How are the seven packages organized?
- What is the dependency flow?
- Why this architecture?

Start here if: You want to understand the overall project structure.

2. Development Tooling **Read: Tooling →**
Master the modern Python toolchain:

- **uv** - Fast package manager
- **uv_build** - Build backend for packages
- **ruff** - Linter and formatter

- **ty** - Type checker
- **Just** - Task runner
- **pytest** - Testing framework
- **MyST** - Documentation system

Start here if: You're new to modern Python development or want to know why we chose these tools.

3. Namespace Packages Deep Dive **Read: Namespace Packages →**

Understand the `canvod.*` namespace:

- What are namespace packages?
- How do they work in Python?
- Why use them vs. regular packages?
- How does `canvod.readers` differ from `canvod_readers`?
- The role of PEP 420

Start here if: You're confused about namespace packages or the project structure.

4. Development Workflow **Read: Development Workflow →**

Learn day-to-day development:

- Setting up your environment
- Working on a package
- Running tests and quality checks
- Adding dependencies
- Building and publishing
- Common tasks and troubleshooting

Start here if: You're ready to contribute code.

5. Build System **Read: Build System →**
Understand package building and distribution:

- What is "building" a package?
- Source distributions vs. wheels
- How uv_build works
- Publishing to PyPI
- Version management
- Build configuration

Start here if: You want to understand how packages are built and published.

Quick Start Guide

For New Developers

1. **Understand the architecture** → *Architecture*
1. **Learn the tools** → *Tooling*
1. **Set up your environment** → *DevelopmentWorkflow*
1. **Start coding!**

For Contributors

1. **Read:** Development Workflow
2. **Clone the repo:** `git clone https://github.com/nfb2021/canvodpy.git`
3. **Setup:** `cd canvodpy && uv sync && just hooks`
4. **Make changes:** Follow the workflow guide
5. **Submit PR:** `git push` and create pull request

For Package Users

1. **Install:** `pip install canvodpy`
2. **Import:** `from canvod.readers import Rnxv3Obs`
3. **Use:** See package-specific documentation

Key Concepts

Monorepo **One repository** containing **multiple packages**:

```
canvodpy/                    # Single repository
  packages/
      canvod -readers/      # Package 1
      canvod -aux/          # Package 2
      canvod -grids/        # Package 3
      ...                  # Packages 4 -6
  canvodpy/                 # Package 7 (umbrella)
```

Benefits:

- Easier to coordinate changes
- Shared tooling and configuration
- Single CI/CD setup
- Better for monolithic-to-modular migration

Namespace Packages **Multiple packages** sharing **one namespace**:

```
# All from different PyPI packages, same namespace:
from canvod.readers import Rnxv3Obs      # canvod -readers
from canvod.grids import HemiGrid         # canvod -grids
from canvod.vod import calculate_vod      # canvod -vod
```

Benefits:

- Professional, unified API
- Clear package relationships
- Users install only what they need
- Extensible by third parties

Workspace **Shared development environment** for all packages:

```
canvodpy/
  .venv/           # Shared virtual environment
  uv.lock          # Shared lockfile
  packages/        # All packages here
```

Benefits:

- One `uv sync` installs everything
- Packages immediately see each other's changes
- Guaranteed compatible versions
- Fast iteration

The Seven Packages

```
canvod-readers    \rightarrow Read GNSS data (RINEX, etc.)
canvod-aux        \rightarrow Handle auxiliary data (meteorology, orbit)
canvod-grids      \rightarrow Spatial grids (HEALPix, hemispheric)
canvod-vod        \rightarrow Calculate vegetation optical depth
canvod-store      \rightarrow Store data (Icechunk, Zarr)
canvod-viz        \rightarrow Visualize results (plots, maps)
canvodpy          \rightarrow Umbrella (imports everything)
```

Dependency flow:

```
readers \rightarrow aux \rightarrow grids \rightarrow vod \rightarrow store \rightarrow viz
```

Technology Stack

Core Technologies

- **Language:** Python 3.13+
- **Package Manager:** uv (Astral)
- **Build Backend:** uv_build (Astral)
- **Namespace:** PEP 420 implicit namespace packages

Development Tools

- **Linter/Formatter:** ruff (Astral)
- **Type Checker:** ty (Astral)
- **Testing:** pytest
- **Task Runner:** Just
- **Pre-commit:** pre-commit hooks
- **CI/CD:** GitHub Actions

Data Technologies

- **Spatial Grids:** HEALPix
- **Storage:** Icechunk, Zarr
- **Data Processing:** NumPy, Pandas, Xarray
- **Formats:** RINEX, NetCDF, Zarr

Project Philosophy

1. Modern Over Legacy We use **modern tools** (uv, ruff, ty) over legacy equivalents (pip, flake8, mypy) for:

- Speed (10-100x faster)
- Better integration
- Simpler configuration
- Active development

2. Modularity Over Monolith **Small, focused packages** instead of one large package:

- Clear responsibilities
- Independent development
- Flexible dependencies
- Easier testing

3. Standards Compliance Following **TU Wien GEO** standards:

- uv-based workflow
- Comprehensive testing
- Quality checks (ruff ALL rules)
- Proper documentation

4. Documentation First **Explain everything:**

- Why decisions were made
- How things work
- What alternatives exist
- Assume no prior knowledge

Getting Help

Documentation

- Read the guides (you are here!)
- Check package-specific READMEs
- See code examples in docs/

Community

- GitHub Issues: Report bugs, request features
- Pull Requests: Contribute code
- Discussions: Ask questions

Resources

- GitHub Repository
- Contributing Guide
- TU Wien GEO

Document Structure This documentation consists of five interconnected guides:

```
index.md (you are here)
      architecture.md        - Overall structure
      tooling.md            - Tools explained
      namespace -packages.md  - Namespace deep dive
      development -workflow.md - Daily development
      build -system.md        - Building & publishing
```

Suggested reading order:

1. Start: index.md (overview)
2. Architecture $\rightarrow$ *tooling* $\rightarrow$ *namespace* $-$ *packages* $\rightarrow$ *development* $-$ *workflow* $\rightarrow$ *build* $-$ *system*

Or jump directly to what you need!

Next Steps Ready to dive in?

- Architecture Overview $\rightarrow$ - Understand the big picture
- Development Tooling $\rightarrow$ - Master the tools
- Namespace Packages $\rightarrow$ - Learn namespace packages
- Development Workflow $\rightarrow$ - Start coding
- Build System $\rightarrow$ - Package building

Or:

- Clone the repo: `git clone https://github.com/nfb2021/canvodpy.git`
- Set up: `cd canvodpy && uv sync`

- Explore: Look around and start coding!

This documentation was written for people who have never worked with monorepos, namespace packages, or modern Python tooling. If anything is unclear, please open an issue!

1.2 Architecture Overview

1.2.1 What is canVODpy?

canVODpy is a **monorepo** containing multiple Python packages for GNSS (Global Navigation Satellite System) vegetation optical depth (VOD) analysis. Instead of having seven separate repositories, we keep all related packages in one place while maintaining their independence.

1.2.2 The Problem We're Solving

Previously, the GNSS VOD analysis code existed as a single large package (`gnssvodpy`). This created several problems:

1. **Tight coupling**: All features were interdependent
2. **Large dependencies**: Installing one feature meant installing all dependencies
3. **Difficult testing**: Hard to test components in isolation
4. **Unclear boundaries**: Code organization was unclear

1.2.3 Our Solution: Modular Monorepo

We split the functionality into **seven independent packages** that work together:

```
canVODpy Monorepo
   canvod -readers    \rightarrow Read GNSS data formats (RINEX, etc.)
   canvod -aux        \rightarrow Handle auxiliary data
   canvod -grids      \rightarrow Manage spatial grids (HEALPix)
   canvod -vod        \rightarrow Calculate vegetation optical depth
   canvod -store      \rightarrow Store data (Icechunk, Zarr)
   canvod -viz        \rightarrow Visualize results
   canvodpy          \rightarrow Umbrella package (imports everything)
```

1.2.4 Key Architectural Decisions

1. Namespace Packages Instead of seven separate top-level packages, we use **namespace packages** so all packages share the `canvod.*` namespace:

```
# All packages share the "canvod" namespace
from canvod.readers import Rnxv3Obs
from canvod.grids import HemiGrid
from canvod.vod import calculate_vod
```

Why? This creates a unified, professional API while keeping packages technically independent.

2. **Monorepo Structure** All packages live in one repository but can be:

   - Developed independently
   - Tested in isolation
   - Published separately to PyPI
   - Versioned individually (if needed)

   Why? Easier to coordinate changes across packages while maintaining modularity.

3. **Workspace Architecture** We use a **workspace** structure where:

   - All packages share one virtual environment (`.venv`)
   - All packages share one lockfile (`uv.lock`)
   - Dependencies are resolved together
   - But each package maintains its own `pyproject.toml`

   Why? Ensures all packages work together with compatible versions.

1.2.5 Directory Structure Explained

```
canvodpy/                              # Repository root

  packages/                           # Independent packages
     canvod -readers/
        src/
           canvod/                  # Namespace (NO __init__.py)
               readers/             # Actual package
                   __init__.py
        tests/
        docs/
        pyproject.toml              # Package config
        Justfile                    # Package commands
        README.md

     canvod -aux/                    # Same structure
         ...

  canvodpy/                          # Umbrella package
     src/
         canvodpy/                  # Regular package
             __init__.py            # Re -exports all subpackages
```

```
.github/                          # CI/CD
   actions/setup/                 # Reusable setup action
   workflows/                     # CI workflows

docs/                             # Documentation (you are here!)
pyproject.toml                    # Workspace configuration
uv.lock                           # Shared lockfile
Justfile                          # Root commands
README.md                         # Project overview
```

1.2.6 Package Independence

Each package can be:

Installed independently:

```
pip install canvod -readers          # Just the readers
pip install canvod -grids canvod -vod # Just grids + VOD
pip install canvodpy                 # Everything
```

Developed independently:

```
cd packages/canvod -readers
just test                            # Test only this package
just build                           # Build only this package
```

Published independently: Each package gets its own PyPI page:

- `pypi.org/project/canvod-readers/`
- `pypi.org/project/canvod-aux/`
- etc.

1.2.7 Dependency Flow

Packages can depend on each other:

```
canvod -readers (no dependencies)
    ↓
canvod -aux (needs readers)
    ↓
canvod -grids (needs aux)
    ↓
canvod -vod (needs grids)
    ↓
canvod -store (needs vod)
    ↓
canvod -viz (needs store)
    ↓
canvodpy (imports all)
```

During development: All packages are installed in "editable mode" so changes to one package immediately affect packages that depend on it.

After publishing: Users can install only what they need.

1.2.8 Why This Architecture?

Advantages

1. **Modularity**: Clear separation of concerns
2. **Flexibility**: Use only what you need
3. **Maintainability**: Easier to understand and modify
4. **Testing**: Test components in isolation
5. **Collaboration**: Different people can own different packages
6. **Dependency management**: Smaller dependency trees per package

Trade-offs

1. **Complexity**: More configuration files to manage
2. **Build time**: Need to build multiple packages
3. **Learning curve**: Developers need to understand the structure

We believe the advantages far outweigh the trade-offs for a project of this size and complexity.

1.2.9 Real-World Example

Before (monolith):

```
# Everything in one package
import gnssvodpy

# Unclear what's what
reader = gnssvodpy.Rnxv3Obs()
grid = gnssvodpy.HemiGrid()
```

After (modular):

```
# Clear, explicit imports
from canvod.readers import Rnxv3Obs
from canvod.grids import HemiGrid

# Or use the umbrella
import canvodpy
reader = canvodpy.readers.Rnxv3Obs()
```

1.2.10 Next Steps

- Understanding the Tooling - Learn about uv, ruff, ty, etc.
- Namespace Packages Deep Dive - How the `canvod.*` namespace works
- Development Workflow - How to work in this monorepo
- Build System - How packages are built and published

1.3 Development Tooling

This document explains every tool we use and why we chose it. Think of this as a comprehensive guide for someone completely new to modern Python development.

1.3.1 The Problem: Traditional Python Tooling

Traditional Python development involves many separate tools:

- `pip` for installing packages
- `virtualenv` or `venv` for environments
- `setuptools` or `distutils` for building
- `flake8` + `black` + `isort` for code quality
- `mypy` for type checking
- `twine` for publishing
- `requirements.txt` or `setup.py` for dependencies

Each tool has its own configuration format, its own commands, and they don't always work well together.

1.3.2 Our Solution: Modern, Integrated Tooling

We use a modern toolchain from Astral that consolidates functionality and works seamlessly together.

1.3.3 Core Tools

1. **uv - The Package Manager** **Website:** https://docs.astral.sh/uv/
   What it does:
   - Manages Python versions
   - Creates virtual environments
   - Installs packages (like `pip`)
   - Resolves dependencies
   - Locks dependencies
   - Runs commands

- Builds packages

Why we chose it:

- **10-100x faster** than pip
- **Written in Rust** for speed and reliability
- **All-in-one tool** replaces pip, venv, pip-tools, and more
- **Compatible with pip** - uses PyPI, respects requirements.txt
- **From Astral** - same team as ruff, actively maintained

Common commands:

```
# Install dependencies
uv sync

# Add a new dependency
uv add numpy

# Run a command in the environment
uv run python script.py
uv run pytest

# Build the package
uv build

# Create environment with specific Python version
uv venv - -python 3.13
```

How it works:

1. **Dependency Resolution:** uv reads `pyproject.toml` and figures out all dependencies
2. **Lockfile Creation:** Creates `uv.lock` with exact versions
3. **Installation:** Downloads and installs packages blazingly fast
4. **Environment Management:** Creates `.venv/` directory with everything installed

Configuration:
All configuration lives in `pyproject.toml`:

```
[project]
name = "canvod -readers"
dependencies = [
    "numpy>=1.24",
    "pandas>=2.0",
]

[dependency -groups]
dev = [
    "pytest>=8.0",
    "ruff>=0.14",
]
```

---

2. uv_build - The Build Backend **Website:** https://docs.astral.sh/uv/concepts/build-backend/

What it does:

- Builds Python packages (creates `.whl` and `.tar.gz` files)
- Handles package metadata
- Supports namespace packages
- Integrates with uv

Why we chose it:

- **Native namespace package support** (critical for our `canvod.*` structure)
- **Fast** - built in Rust
- **Simple configuration** - minimal boilerplate
- **Pure uv ecosystem** - no need for setuptools or hatchling

How it works:
In `pyproject.toml`:

```
[build -system]
requires = ["uv_build>=0.9.17,<0.10.0"]
build -backend = "uv_build"

[tool.uv.build -backend]
module -name = "canvod.readers"  # The dot creates a namespace package!
```

The dot in `"canvod.readers"` tells uv_build:

- This is a **namespace package**
- Don't create `canvod/__init__.py`
- Allow multiple packages to share the `canvod` namespace

Building:

```
uv build              # Creates dist/canvod_readers -0.1.0.tar.gz and .whl
```

---

3. **ruff - The Linter and Formatter** **Website:** https://docs.astral.sh/ruff/
   What it does:

   - **Lints** Python code (finds errors, style issues, bad practices)
   - **Formats** Python code (makes it pretty and consistent)
   - **Replaces:** flake8, pylint, black, isort, and 10+ other tools

   Why we chose it:

   - **10-100x faster** than traditional tools
   - **Written in Rust**
   - **All-in-one** - no need for black + isort + flake8
   - **Configurable** - thousands of rules
   - **Auto-fixes** many issues

   What makes it special:
   Ruff implements **700+ linting rules** from many different tools:

   - Pyflakes (F)
   - pycodestyle (E, W)
   - isort (I)
   - pep8-naming (N)
   - flake8-bugbear (B)
   - flake8-comprehensions (C4)
   - ... and many more

Our configuration:

```
[tool.ruff]
line -length = 88
target -version = "py313"

[tool.ruff.lint]
select = ["ALL"]  # Enable ALL rules
ignore = [
    "D",      # pydocstyle (we'll add docstrings later)
    "ANN",    # flake8 -annotations
    "COM812", # trailing comma (conflicts with formatter)
]
```

Common commands:

```
# Check for issues
ruff check .

# Check and auto -fix
ruff check . - -fix

# Format code
ruff format .

# Check if formatted
ruff format . - -check
```

Example:
Before ruff:

```
import os, sys  # Multiple imports on one line
x=1+2  # No spaces around operators
def badname( ): pass  # Bad style
```

After ruff:

```
import os
import sys

x = 1 + 2


def badname():
    pass
```

4. **ty - The Type Checker** **Website:** Part of Astral's ecosystem (new tool)
   What it does:

   - Checks Python type hints
   - Finds type errors before runtime
   - **Replaces:** mypy

   Why we chose it:

   - **Faster** than mypy
   - **From Astral** - same ecosystem as uv and ruff
   - **Better error messages**
   - **Modern** - designed for modern Python

   How type checking works:
   Python supports optional type hints:

```
# Without types (old way)
def add(a, b):
    return a + b

# With types (modern way)
def add(a: int, b: int) -> int:
    return a + b
```

   The type checker verifies:

```
add(1, 2)       #  OK
add("a", "b")   #  Error: expected int, got str
```

   Configuration:

```
[tool.ty]
strict = false  # Start relaxed, become strict later
```

   Common commands:

```
# Check types
ty check .

# Check specific file
ty check src/canvod/readers/__init__.py
```

1.3.4 Supporting Tools

5. **Just - Task Runner** **Website:** https://github.com/casey/just
   What it does:

   - Runs common development commands
   - Like `make` but for any language
   - **Replaces:** Makefile, shell scripts, npm scripts

   Why we chose it:

   - **Simple syntax** - easier than Makefiles
   - **Cross-platform** - works on Windows, Mac, Linux
   - **Commands with arguments** - flexible
   - **Aliases** - shortcuts for common tasks

   Example Justfile:

```
# Run tests
test:
    uv run pytest

# Format and lint
check:
    uv run ruff format .
    uv run ruff check . - -fix
    uv run ty check

# Clean build artifacts
clean:
    rm -rf dist/ build/
```

   Usage:

```
just test         # Run tests
just check        # Check code quality
just              # List all commands
```

---

6. **pytest - Testing Framework** **Website:** https://docs.pytest.org/
   What it does:

   - Runs tests
   - Generates coverage reports
   - Provides fixtures and mocking

   Why we chose it:

   - **Industry standard** for Python testing
   - **Simple** - just write `test_*.py` files
   - **Powerful** - fixtures, parametrization, plugins
   - **Good integration** with coverage tools

   Example test:

```
# tests/test_readers.py
def test_rnx_reader():
    from canvod.readers import Rnxv3Obs
    reader = Rnxv3Obs()
    assert reader is not None
```

   Running:

```
uv run pytest                    # Run all tests
uv run pytest tests/test_file.py # Run specific test
uv run pytest - -cov=canvod       # With coverage
```

---

7. **pre-commit - Git Hooks** **Website:** https://pre-commit.com/
   What it does:

   - Runs checks before you commit code
   - Prevents bad code from being committed
   - Auto-formats code on commit

   Why we chose it:

   - **Catches issues early** - before they reach CI
   - **Configurable** - run any checks you want

- **Standard tool** - widely used

Our configuration:

```
repos:
  - repo: https://github.com/astral-sh/ruff-pre-commit
    hooks:
      - id: ruff-check
        args: [--fix]
      - id: ruff-format

  - repo: https://github.com/astral-sh/uv-pre-commit
    hooks:
      - id: uv-lock  # Update lockfile
```

Installation:

```
just hooks        # Installs pre-commit hooks
# OR
uvx pre-commit install
```

Now every time you `git commit`, the hooks run automatically!

---

8. **MyST - Documentation** **Website:** https://mystmd.org/
   What it does:
   - Modern documentation system
   - **Markdown-based** (easier than reStructuredText)
   - **Supports Jupyter notebooks**
   - **Beautiful output**

   Why we chose it:
   - **Markdown** - easier to write than RST
   - **Jupyter integration** - include notebooks
   - **Modern** - better than Sphinx for new projects
   - **Interactive** - live preview

   Usage:

```
uv run myst        # Preview docs locally
```

1.3.5 Tool Comparison

Task	Traditional	Modern (Our Choice)
Package manager	pip	**uv**
Environment	venv	**uv** (built-in)
Linting	flake8 + pylint	**ruff**
Formatting	black + isort	**ruff**
Type checking	mypy	**ty**
Building	setuptools	**uv_build**
Task runner	make / shell	**just**
Testing	pytest	**pytest** (same)
Documentation	Sphinx	**MyST**

1.3.6 The Workflow

Here's how all these tools work together:

1. **Write code** in your editor
2. **pre-commit** runs ruff and checks on commit
3. **uv** manages dependencies automatically
4. **just test** runs pytest
5. **just check** runs ruff + ty
6. **GitHub Actions** runs everything in CI
7. **uv build** creates packages
8. **MyST** generates documentation

Everything is **fast**, **integrated**, and **modern**.

1.3.7 Learning Resources

- uv documentation
- ruff documentation
- Just manual
- pytest documentation
- MyST documentation

1.3.8 Next Steps

- Namespace Packages - How `canvod.*` works
- Development Workflow - Day-to-day development
- Build System - How building works

1.4 Namespace Packages

1.4.1 Namespace Packages Explained

The Problem: Package Naming Imagine you're building a data science toolkit. The traditional approach creates separate packages:

```
import mycompany_readers
import mycompany_processors
import mycompany_writers
```

This works but has issues:

- **Verbose**: Long package names
- **Disorganized**: Packages aren't clearly related
- **Cluttered**: Every package takes a top-level name

The Solution: Namespace Packages Namespace packages let multiple packages share a common prefix:

```
from mycompany.readers import CSVReader
from mycompany.processors import DataCleaner
from mycompany.writers import ExcelWriter
```

Notice the pattern: `mycompany.{subpackage}`. This creates a **professional, hierarchical API** while keeping packages independent.

How We Use Them In canVODpy, we have seven packages that all share the `canvod` namespace:

```
from canvod.readers import Rnxv3Obs      # canvod-readers package
from canvod.aux import AuxData            # canvod-aux package
from canvod.grids import HemiGrid         # canvod-grids package
from canvod.vod import calculate_vod      # canvod-vod package
from canvod.store import IcechunkStore    # canvod-store package
from canvod.viz import plot_hemisphere    # canvod-viz package
```

Each import comes from a **different package**, but they all use the `canvod.*` namespace.

Understanding the Structure

Regular Package (Before) A traditional package structure:

```
canvod_readers/              # Package name
  pyproject.toml
  src/
      canvod_readers/      # Module name (matches package)
          __init__.py
```

Import: `from canvod_readers import Rnxv3Obs`
Problem: The package name `canvod_readers` pollutes the global namespace.

Namespace Package (After) Our namespace package structure:

```
canvod -readers/              # Package name (with dash)
  pyproject.toml
  src/
      canvod/              # Namespace (NO __init__.py!)
          readers/         # Module name
              __init__.py
```

Import: `from canvod.readers import Rnxv3Obs`
Key differences:

1. Package name uses **dashes**: `canvod-readers`
2. Namespace directory has **NO `__init__.py`**: `canvod/` (empty)
3. Module directory has **`__init__.py`**: `canvod/readers/`

The Magic: No `__init__.py` The critical part is that **`src/canvod/` has NO `__init__.py` file**.

```
src/
  canvod/                   # \leftarrow NO __init__.py here!
      readers/
          __init__.py       # \leftarrow __init__.py ONLY here
```

Why?
When Python sees a directory without `__init__.py`, it treats it as a **namespace package**. Multiple packages can contribute to the same namespace.

Visual Comparison Regular package:

```
src/
  canvod/
      __init__.py          # \leftarrow This makes it a regular package
      readers/
          __init__.py
```

Result: **Only one package can use the name `canvod`**
Namespace package:

```
src/
  canvod/                  # \leftarrow NO __init__.py = namespace
      readers/
          __init__.py
```

Result: **Multiple packages can share the `canvod` namespace**

How Python Finds Imports When you write:

```
from canvod.readers import Rnxv3Obs
```

Python searches:

1. Look for `canvod` (finds it's a namespace - no `__init__.py`)
2. Look for `readers` within `canvod` (finds regular package with `__init__.py`)
3. Import `Rnxv3Obs` from `canvod/readers/__init__.py`

With Multiple Packages Installed If you have both `canvod-readers` and `canvod-grids` installed:

```
from canvod.readers import Rnxv3Obs    # From canvod -readers package
from canvod.grids import HemiGrid       # From canvod -grids package
```

Python finds both because:

- Both packages contribute to the `canvod` namespace
- Python merges them seamlessly
- No conflict because each has its own submodule (`readers` vs `grids`)

PEP 420: Implicit Namespace Packages Our approach follows PEP 420 "Implicit Namespace Packages" (Python 3.3+).

Before PEP 420 (Python 2): You needed special code in `__init__.py`:

```
# canvod/__init__.py (OLD WAY - DON'T DO THIS)
__path__ = __import__('pkgutil').extend_path(__path__, __name__)
```

After PEP 420 (Python 3.3+): Just... don't create `__init__.py` in the namespace directory. That's it!

```
src/canvod/              # NO __init__.py needed!
```

Why is this better?

- Simpler
- Standard Python
- Faster
- Better tool support

Configuring uv_build To tell `uv_build` we're creating a namespace package, we use a **dotted module name**:

```
# packages/canvod-readers/pyproject.toml
[build-system]
requires = ["uv_build>=0.9.17,<0.10.0"]
build-backend = "uv_build"

[tool.uv.build-backend]
module-name = "canvod.readers"  # \leftarrow The dot indicates namespace!
```

The dot in `"canvod.readers"` tells uv_build:

- `canvod` is a **namespace** (don't package it as a regular module)
- `readers` is the **actual module** (package this)

What Gets Built When you run `uv build`, uv_build creates a wheel containing:

```
canvod_readers -0.1.0 -py3 -none -any.whl
  canvod/
      readers/
          __init__.py
          ... (your code)
```

Notice:

- No `canvod/__init__.py` in the wheel
- Only `canvod/readers/` content is included
- This allows other packages to add to `canvod/`

Multiple Packages, Same Namespace Here's how all seven packages co-exist:

Package: canvod-readers

```
wheel: canvod_readers -0.1.0.whl
  canvod/
      readers/
          __init__.py
```

Package: canvod-aux

```
wheel: canvod_aux -0.1.0.whl
  canvod/
      aux/
          __init__.py
```

Package: canvod-grids

```
wheel: canvod_grids -0.1.0.whl
  canvod/
      grids/
          __init__.py
```

When all installed:

```
site -packages/
  canvod/
      readers/      # From canvod -readers
      aux/          # From canvod -aux
      grids/        # From canvod -grids
      vod/          # From canvod -vod
      store/        # From canvod -store
      viz/          # From canvod -viz
```

All packages contribute to the same `canvod/` directory without conflict!

Common Pitfalls

Pitfall 1: Creating `__init__.py` in namespace

```
src/
  canvod/
      __init__.py        # \leftarrow DON'T DO THIS!
      readers/
          __init__.py
```

Problem: Creates a regular package, breaks namespace sharing.

Pitfall 2: Wrong module-name syntax

```
[tool.uv.build -backend]
module -name = "canvod_readers"    # \leftarrow Wrong! No dot = regular package
```

Problem: Won't create proper namespace structure.

Correct: Use dot notation

```
[tool.uv.build -backend]
module -name = "canvod.readers"    # \leftarrow Correct! Dot = namespace
```

Testing Namespace Packages You can verify namespace packages work:

```
# Test that imports work
from canvod.readers import Rnxv3Obs
from canvod.grids import HemiGrid

# Test that canvod is a namespace (has no __file__)
import canvod
print(canvod.__file__)  # Should raise AttributeError

# Test that submodules are real packages (have __file__)
from canvod import readers
print(readers.__file__)  # Should show the file path
```

Advantages of Namespace Packages

1. **Professional API**: Clean, hierarchical imports
2. **Modularity**: Users install only what they need
3. **Independence**: Each package can be developed separately
4. **No conflicts**: Packages don't step on each other's toes
5. **Extensibility**: Anyone can add to the namespace
6. **Organization**: Related packages clearly grouped

Disadvantages

1. **Complexity**: Slightly more complex setup
2. **Understanding**: Developers need to understand the concept
3. **Tooling**: Some old tools don't support them well (we use modern tools that do)

Real-World Examples Many major Python projects use namespace packages:

- **Azure SDK**: `azure.storage`, `azure.compute`, `azure.ai`
- **Google Cloud**: `google.cloud.storage`, `google.cloud.compute`
- **Zope**: `zope.interface`, `zope.component`, `zope.schema`
- **Sphinx extensions**: `sphinxcontrib.*`

Summary Namespace packages let multiple independent packages share a common prefix.

In canVODpy:

- Seven packages all use `canvod.*` namespace
- Each package is independent and installable separately
- No `__init__.py` in `src/canvod/` (that's the key!)
- uv_build configured with dotted `module-name`
- Result: Clean, professional API

```
# Users import from a unified namespace
from canvod.readers import Rnxv3Obs
from canvod.grids import HemiGrid

# But behind the scenes, these are separate packages!
```

Next Steps

- Architecture Overview - Overall project structure
- Build System - How packages are built
- Development Workflow - Working with namespace packages

1.5 Development Workflow

This guide explains how to work effectively in the canVODpy monorepo, from setup to publishing.

1.5.1 Initial Setup

1. Clone the Repository

```
git clone https://github.com/nfb2021/canvodpy.git
cd canvodpy
```

2. Install Dependencies

```
# Install all packages in the workspace
uv sync

# This creates .venv/ and installs everything
```

What happens:

- uv reads `pyproject.toml` (workspace root)
- uv reads all package `pyproject.toml` files
- uv resolves all dependencies together
- uv creates a virtual environment (`.venv/`)
- uv installs all packages in editable mode
- Creates/updates `uv.lock` with exact versions

3. Install Pre-commit Hooks

```
just hooks
# OR
uvx pre -commit install
```

This ensures code quality checks run automatically before each commit.

1.5.2 Daily Development

Understanding the Workspace In a monorepo workspace:

- **One virtual environment** (`.venv/`) shared by all packages
- **One lockfile** (`uv.lock`) for all dependencies
- **Editable installs**: Changes to any package immediately affect others

```
canvodpy/
  .venv/                  # Shared virtual environment
  uv.lock                 # Shared lockfile
  packages/               # Your packages
      canvod -readers/
      canvod -aux/
```

Activating the Environment **Option 1: Use `uv run`** (recommended)

```
uv run python script.py
uv run pytest
uv run ruff check .
```

Option 2: Activate manually

```
source .venv/bin/activate    # Linux/Mac
# OR
.venv\Scripts\activate       # Windows

# Now you can use commands directly
python script.py
pytest
```

1.5.3 Working on a Package

Example: Adding a Feature to canvod-readers

1. Navigate to the package

```
cd packages/canvod -readers
```

2. Create a new module

```
# Create a new Python file
touch src/canvod/readers/rinex_v4.py
```

3. **Write code**

```
# src/canvod/readers/rinex_v4.py
"""RINEX version 4 reader."""

class Rnxv4Obs:
    """Read RINEX 4 observation files."""

    def __init__(self, filepath: str):
        self.filepath = filepath

    def read(self) -> dict:
        """Read the file and return data."""
        # Implementation here
        pass
```

4. **Update `__init__.py`**

```
# src/canvod/readers/__init__.py
"""GNSS data format readers."""

from canvod.readers.rinex_v3 import Rnxv3Obs
from canvod.readers.rinex_v4 import Rnxv4Obs  # Add new import

__all__ = ["Rnxv3Obs", "Rnxv4Obs"]
```

5. **Write tests**

```
# tests/test_rinex_v4.py
from canvod.readers import Rnxv4Obs

def test_rnxv4_creation():
    """Test that Rnxv4Obs can be created."""
    reader = Rnxv4Obs("test.rnx")
    assert reader is not None
    assert reader.filepath == "test.rnx"
```

6. **Run tests**

```
# From package directory
just test

# OR from workspace root
just test -package canvod -readers

# OR run specific test
uv run pytest tests/test_rinex_v4.py
```

7. Check code quality

```
# Format, lint, and type check
just check

# OR individually
uv run ruff format .
uv run ruff check . - -fix
uv run ty check
```

8. Commit your changes

```
git add .
git commit -m "Add RINEX v4 reader"

# Pre -commit hooks run automatically:
# - ruff format
# - ruff check - -fix
# - uv lock (updates lockfile)
```

1.5.4 Working Across Packages

Scenario: canvod-grids needs new features from canvod-readers Because packages are installed in **editable mode**, changes propagate immediately.

1. Add feature to canvod-readers

```
# packages/canvod -readers/src/canvod/readers/utils.py
def parse_metadata(file):
    """Parse RINEX metadata."""
    return {"version": "4.0", "interval": 30}
```

2. Use it immediately in canvod-grids

```
# packages/canvod -grids/src/canvod/grids/loader.py
from canvod.readers.utils import parse_metadata  # Works instantly!

def load_grid(rinex_file):
    metadata = parse_metadata(rinex_file)
    # ... use metadata
```

No reinstalling needed! Editable mode means Python loads code directly from source.

1.5.5 Adding Dependencies

To a Specific Package

```
# Navigate to the package
cd packages/canvod -readers

# Add dependency
uv add numpy pandas

# This updates:
# - packages/canvod -readers/pyproject.toml (adds numpy, pandas)
# - uv.lock (locks versions for entire workspace)
```

To Development Dependencies

```
# Add dev dependencies (for testing, linting, etc.)
cd packages/canvod -readers
uv add - -group dev pytest -mock ipython
```

To Workspace Root

```
# Dependencies needed by ALL packages
cd ~/path/to/canvodpy
uv add - -group dev ruff ty pytest
```

1.5.6 Testing Strategies

Test a Single Package

```
cd packages/canvod -readers
just test

# OR from root
just test -package canvod -readers
```

Test All Packages

```
# From workspace root
just test
```

Test with Coverage

```
uv run pytest - -cov=canvod.readers - -cov -report=html
# Opens htmlcov/index.html in browser
```

Test Specific Features

```
# Run tests matching a pattern
uv run pytest -k "rinex"

# Run specific test file
uv run pytest tests/test_rinex_v4.py

# Run specific test function
uv run pytest tests/test_rinex_v4.py::test_rnxv4_creation
```

1.5.7 Code Quality Workflow

Manual Checks

```
# From any package or root
just check

# This runs:
# 1. ruff format .    (format code)
# 2. ruff check . - -fix (lint and auto -fix)
# 3. ty check         (type check)
```

Pre-commit (Automatic) When you commit, pre-commit hooks run automatically:

```
git commit -m "Add feature"

# Runs automatically:
# - ruff format
# - ruff check - -fix
# - uv lock update

# If checks fail, commit is rejected
# Fix the issues and commit again
```

CI/CD (GitHub Actions) On every push, GitHub Actions runs:

1. **Code Quality** (`.github/workflows/code_quality.yml`):
    - Lock file check
    - Linting
    - Formatting

- Type checking

2. **Test Coverage** (`.github/workflows/test_coverage.yml`):
   - Run tests
   - Generate coverage report
   - Post coverage comment on PR
3. **Multi-platform Tests** (`.github/workflows/test_platforms.yml`):
   - Test on Ubuntu, Windows, macOS
   - Test on Python 3.13

1.5.8 Building Packages

Build a Single Package

```
cd packages/canvod -readers
just build

# Creates:
# dist/canvod_readers -0.1.0 -py3 -none -any.whl
# dist/canvod_readers -0.1.0.tar.gz
```

Build All Packages

```
# From workspace root
just dist

# OR manually
for pkg in packages/*/; do
  cd "$pkg"
  uv build
  cd ../..
done
```

Verify Build

```
# Check wheel contents
unzip -l dist/canvod_readers -0.1.0 -py3 -none -any.whl

# Should see:
# canvod/readers/__init__.py
# canvod/readers/rinex_v3.py
# ... etc
```

1.5.9 Version Management

Bump Version

```
# From workspace root
just bump patch    # 0.1.0 \rightarrow 0.1.1
just bump minor    # 0.1.0 \rightarrow 0.2.0
just bump major    # 0.1.0 \rightarrow 1.0.0

# This:
# 1. Updates pyproject.toml
# 2. Updates uv.lock
# 3. Creates git commit
# 4. Creates git tag
```

Manual Version Update

```
# packages/canvod-readers/pyproject.toml
[project]
name = "canvod-readers"
version = "0.2.0"  # \leftarrow Change this
```

Then:

```
uv lock  # Update lockfile
git commit -am "Bump canvod-readers to 0.2.0"
git tag v0.2.0
```

1.5.10 Common Tasks

Adding a New Package to the Workspace

```
# 1. Create package structure
mkdir -p packages/canvod-newpackage/src/canvod/newpackage
mkdir -p packages/canvod-newpackage/tests
mkdir -p packages/canvod-newpackage/docs

# 2. Create pyproject.toml
cat > packages/canvod-newpackage/pyproject.toml << 'EOF'
[project]
name = "canvod-newpackage"
```

```
version = "0.1.0"
description = "New package description"
requires -python = ">=3.13"
dependencies = []

[build -system]
requires = ["uv_build>=0.9.17,<0.10.0"]
build -backend = "uv_build"

[tool.uv.build -backend]
module -name = "canvod.newpackage"
EOF

# 3. Create __init__.py
echo '"""New package."""' > packages/canvod -newpackage/src/canvod/newpackage/__init__.py

# 4. Sync workspace
uv sync

# 5. Verify
python -c "from canvod.newpackage import *"
```

Cleaning Build Artifacts

```
# From workspace root
just clean

# This removes:
# - build/
# - dist/
# - *.egg -info
# - __pycache__
# - .pytest_cache
# - .coverage
```

Updating Dependencies

```
# Update all dependencies to latest compatible versions
uv lock - -upgrade

# Update specific package
uv lock - -upgrade -package numpy
```

1.5.11 Troubleshooting

"Module not found" after adding new code **Problem:** Python can't find your new module.

Solution: Make sure package is installed in editable mode:

```
uv sync  # Reinstalls all packages
```

"Lock file out of date" **Problem:** `uv.lock` doesn't match `pyproject.toml`.

Solution:

```
uv lock  # Regenerate lockfile
```

Pre-commit hooks fail **Problem:** Code doesn't pass quality checks.

Solution:

```
# Run checks manually to see errors
just check

# Fix issues
uv run ruff check . - -fix
uv run ruff format .

# Try commit again
git commit -m "..."
```

Import errors between packages **Problem:** Package A can't import from Package B.

Solution:

1. Check Package B is a dependency of Package A:

```
# packages/canvod -grids/pyproject.toml
[project]
dependencies = [
    "canvod -readers",  # \leftarrow Must be listed
]
```

[resume]

1. Run `uv sync` to install dependencies

1.5.12 Best Practices

1. Always Run Tests Before Committing

```
just test && just check
git commit -m "..."
```

2. Keep Packages Focused Each package should have a **single, clear responsibility**:

- `canvod-readers`: Read GNSS data
- `canvod-readers`: Read data + process + visualize (too much)

3. Document New Features Add docstrings to all public functions:

```
def parse_rinex(filepath: str) -> dict:
    """Parse a RINEX observation file.

    Args:
        filepath: Path to RINEX file

    Returns:
        Dictionary containing observations

    Raises:
        FileNotFoundError: If file doesn't exist
    """
```

4. Write Tests for New Features Every new feature needs tests:

```
def test_new_feature():
    """Test that new feature works."""
    result = new_feature()
    assert result == expected
```

5. Use Type Hints

```
# Good
def process(data: dict) -> list[str]:
    ...

# Bad
def process(data):
    ...
```

1.5.13 Workflow Summary

Daily development cycle:

1. `git checkout -b feature-branch`
2. Make changes to code
3. `just test` (run tests)
4. `just check` (code quality)
5. `git commit -m "..."` (pre-commit runs automatically)
6. `git push`
7. Create pull request
8. GitHub Actions run CI
9. Merge when green

Key commands to remember:

```
uv sync          # Install/update dependencies
just test        # Run tests
just check       # Check code quality
just clean       # Clean artifacts
just hooks       # Install pre -commit
```

1.5.14 Next Steps

- Architecture - Understanding the project structure
- Tooling - Deep dive into tools
- Build System - How packages are built

1.6 Build System

1.6.1 Build System Deep Dive

This document explains how Python packages are built, distributed, and installed. We'll go from basic concepts to our specific implementation.

What is "Building" a Package? **Building** a Python package means creating distribution files that can be:

- Uploaded to PyPI (Python Package Index)
- Installed with `pip` or `uv`
- Shared with users

Distribution Formats Python has two main distribution formats:

1. Source Distribution (sdist) **File:** `canvod_readers-0.1.0.tar.gz` **Contains:**

- Source code (`.py` files)
- `pyproject.toml`
- `README.md`, `LICENSE`
- Everything needed to build the package

When used:

- No prebuilt wheel available
- Source-only packages (rare)
- Building from source

Install process:

1. Download `.tar.gz`
2. Extract files
3. Run build process
4. Install resulting files

2. Wheel (Built Distribution) **File:** `canvod_readers-0.1.0-py3-none-any.whl`
Contains:

- Pre-built package ready to install
- Just copy files to site-packages
- No compilation needed

When used:

- Preferred format (fastest)
- Most packages on PyPI have wheels
- Cross-platform (pure Python)

Install process:

1. Download `.whl`
2. Extract directly to site-packages
3. Done! (very fast)

Filename breakdown:

```
canvod_readers -0.1.0 -py3 -none -any.whl

                              Platform (any = all platforms)
                            ABI tag (none = pure Python)
                          Python version (py3 = Python 3)
                        Version (0.1.0)
                  Package name
```

The Build System: pyproject.toml All build configuration lives in `pyproject.toml`:

```
[build -system]
requires = ["uv_build>=0.9.17,<0.10.0"]  # Build tool needed
build -backend = "uv_build"                # Which backend to use
```

What is a Build Backend? A **build backend** is a tool that knows how to:

1. Read your source code
2. Package it correctly
3. Create distribution files

Common build backends:

- `uv_build` (modern, fast, Rust-based)
- `hatchling` (popular, feature-rich)
- `setuptools` (traditional, widely used)
- `flit` (simple, lightweight)
- `poetry-core` (used by Poetry)

We use `uv_build` because:

- Native namespace package support
- Extremely fast (Rust)
- Simple configuration
- Part of the uv ecosystem

Our Build Configuration

Standard Package Configuration

```
# packages/canvod-readers/pyproject.toml
[project]
name = "canvod-readers"              # PyPI package name
version = "0.1.0"                    # Semantic version
description = "GNSS data readers"    # Short description
requires-python = ">=3.13"           # Python version requirement
dependencies = [                     # Runtime dependencies
    "numpy>=1.24",
    "pandas>=2.0",
]
```

```
[dependency -groups]
dev = [                                # Development dependencies
    "pytest>=8.0",
    "ruff>=0.14",
]

[build -system]
requires = ["uv_build>=0.9.17,<0.10.0"]
build -backend = "uv_build"

[tool.uv.build -backend]
module -name = "canvod.readers"       # Namespace package config
```

The Critical Part: module-name

```
[tool.uv.build -backend]
module -name = "canvod.readers"
```

This single line tells uv_build:

- Create a namespace package structure
- `canvod` is the namespace (shared)
- `readers` is the module (unique to this package)

Without the dot:

```
module -name = "canvod_readers"  # Regular package
```

Result: `canvod_readers/__init__.py`
With the dot:

```
module -name = "canvod.readers"  # Namespace package
```

Result: `canvod/readers/__init__.py` (no `canvod/__init__.py`)

The Build Process

Manual Build

```
cd packages/canvod -readers
uv build
```

What happens:

1. **Read Configuration**
   - uv reads `pyproject.toml`
   - Finds `build-backend = "uv_build"`
   - Loads `uv_build` package
2. **Prepare Source**
   - Scans `src/` directory
   - Finds `canvod/readers/` based on `module-name`
   - Excludes test files, pycache, etc.
3. **Create Source Distribution**
   - Packages source code
   - Includes metadata
   - Creates `dist/canvod_readers-0.1.0.tar.gz`
4. **Create Wheel**
   - Copies Python files
   - Generates metadata files
   - Creates `dist/canvod_readers-0.1.0-py3-none-any.whl`

What's Inside the Wheel

```
unzip -l dist/canvod_readers -0.1.0 -py3 -none -any.whl
```

```
canvod/                            # Namespace directory
  readers/                         # Module directory
    __init__.py
    rinex_v3.py
    rinex_v4.py
    utils.py
canvod_readers -0.1.0.dist -info/   # Metadata
  METADATA                         # Package metadata
  WHEEL                            # Wheel format metadata
  RECORD                           # File checksums
  entry_points.txt                 # CLI scripts (if any)
  top_level.txt                    # Top -level imports
```

Key observations:

- `canvod/` has NO `__init__.py` (namespace!)
- `canvod/readers/` has `__init__.py` (regular package)
- Metadata in separate `.dist-info` directory

Building the Entire Workspace

Build All Packages

```
# From workspace root
for pkg in packages/*/; do
  cd "$pkg"
  uv build
  cd ../..
done

# OR use Just
just dist
```

This creates:

```
packages/
   canvod -readers/
       dist/
           canvod_readers -0.1.0.tar.gz
           canvod_readers -0.1.0 -py3 -none -any.whl
   canvod -aux/
       dist/
           canvod_aux -0.1.0.tar.gz
           canvod_aux -0.1.0 -py3 -none -any.whl
   ...
```

Collect All Wheels

```
# Copy all wheels to a single directory
mkdir -p dist -all
find packages -name "*.whl" -exec cp {} dist -all/ \;

# Result:
# dist -all/
#      canvod_readers -0.1.0 -py3 -none -any.whl
#      canvod_aux -0.1.0 -py3 -none -any.whl
#      ...
```

Installing Built Packages

From Local Wheel

```
# Install specific package
pip install dist/canvod_readers -0.1.0 -py3 -none -any.whl

# OR with uv
uv pip install dist/canvod_readers -0.1.0 -py3 -none -any.whl
```

From PyPI (after publishing)

```
pip install canvod -readers

# OR install specific version
pip install canvod -readers==0.1.0
```

Installing All Packages

```
# Install all from local wheels
pip install dist -all/*.whl

# OR from PyPI
pip install canvodpy  # Umbrella package that depends on all
```

Publishing to PyPI

Prerequisites

1. **PyPI Account**
   - Sign up at https://pypi.org
   - Create API token
2. **Configure Credentials**

   ```
   # Store token securely
   uv publish - -token $PYPI_TOKEN

   # OR use .pypirc
   cat > ~/.pypirc << EOF
   [pypi]
   username = __token__
   password = pypi -...your -token...
   EOF
   ```

Publishing Process

Test on TestPyPI First

```
cd packages/canvod-readers

# Build
uv build

# Upload to TestPyPI
uv publish --repository testpypi

# Test installation
pip install --index-url https://test.pypi.org/simple/ canvod-readers
```

Publish to Production PyPI

```
# Build
uv build

# Upload to PyPI
uv publish

# Now anyone can install:
pip install canvod-readers
```

Publishing All Packages

```
for pkg in packages/*/; do
  cd "$pkg"
  uv build
  uv publish
  cd ../..
done
```

Version Management

Semantic Versioning We follow Semantic Versioning:
Format: `MAJOR.MINOR.PATCH` (e.g., `1.2.3`)

- **MAJOR**: Breaking changes (1.0.0 → 2.0.0)
- **MINOR**: New features, backwards compatible (1.0.0 → 1.1.0)

- **PATCH**: Bug fixes (1.0.0 → 1.0.1)

Pre-release: `1.0.0-alpha`, `1.0.0-beta`, `1.0.0-rc1`

Version Bumping

```
# From workspace root
just bump patch    # 0.1.0 \rightarrow 0.1.1
just bump minor    # 0.1.0 \rightarrow 0.2.0
just bump major    # 0.1.0 \rightarrow 1.0.0
```

This:

1. Updates `version` in `pyproject.toml`
2. Updates `uv.lock`
3. Creates git commit
4. Creates git tag (`v0.1.1`)

Version Constraints in Dependencies When one package depends on another:

```
# packages/canvod -grids/pyproject.toml
[project]
dependencies = [
    "canvod -readers>=0.1.0",    # Any version \geq 0.1.0
    "canvod -aux>=0.1.0,<0.2.0", # Pin to 0.1.x
    "numpy~=1.24.0",            # Compatible with 1.24.x
]
```

Constraint types:

- `>=0.1.0` - At least 0.1.0
- `<2.0.0` - Below 2.0.0
- `~=1.2.0` - Compatible release (1.2.0 to <1.3.0)
- `==1.2.3` - Exact version (avoid unless necessary)

Metadata in pyproject.toml

Full Example

```
[project]
name = "canvod -readers"
version = "0.1.0"
description = "GNSS data format readers for canVODpy"
readme = "README.md"
license = {text = "Apache -2.0"}
authors = [
    {name = "Nicolas Bader", email = "nicolas.bader@geo.tuwien.ac.at"}
]
maintainers = [
    {name = "Nicolas Bader", email = "nicolas.bader@geo.tuwien.ac.at"}
]
keywords = ["gnss", "rinex", "geodesy"]
classifiers = [
    "Development Status :: 3 - Alpha",
    "Intended Audience :: Science/Research",
    "License :: OSI Approved :: Apache Software License",
    "Programming Language :: Python :: 3",
    "Programming Language :: Python :: 3.13",
    "Topic :: Scientific/Engineering :: GIS",
]
requires -python = ">=3.13"
dependencies = [
    "numpy>=1.24",
    "pandas>=2.0",
]

[project.urls]
Homepage = "https://github.com/nfb2021/canvodpy"
Documentation = "https://canvodpy.readthedocs.io"
Repository = "https://github.com/nfb2021/canvodpy"
Issues = "https://github.com/nfb2021/canvodpy/issues"

[project.optional -dependencies]
viz = ["matplotlib>=3.7"]
ml = ["scikit -learn>=1.3"]

[dependency -groups]
dev = [
    "pytest>=8.0",
    "pytest -cov>=5.0",
    "ruff>=0.14",
    "ty>=0.0.9",
]
```

```
[build -system]
requires = ["uv_build>=0.9.17,<0.10.0"]
build -backend = "uv_build"

[tool.uv.build -backend]
module -name = "canvod.readers"
```

Metadata shows up on PyPI:

- Package name, description
- Author information
- License
- Links (homepage, docs, issues)
- Python version requirements
- Dependencies
- Keywords and classifiers

Build Customization

Including Extra Files By default, only Python files in `src/` are included. To include extra files:

```
[tool.uv.build -backend]
module -name = "canvod.readers"
include = [
    "src/canvod/readers/data/*.dat",  # Include data files
    "LICENSE",
    "README.md",
]
exclude = [
    "src/canvod/readers/tests/",      # Exclude tests
    "*.pyc",
    "__pycache__",
]
```

Package Data To include non-Python files that should be installed:

```
[tool.uv.build -backend]
module -name = "canvod.readers"

# Include data files in package
package -data = {"canvod.readers" = ["data/*.json", "schemas/*.yaml"]}
```

Access in code:

```
from importlib.resources import files

data = files("canvod.readers").joinpath("data/config.json").read_text()
```

Advanced: Entry Points

Console Scripts Create command-line tools from your package:

```
[project.scripts]
canvod -read = "canvod.readers.cli:main"
```

Now users can run:

```
canvod -read file.rnx
```

Which calls:

```
# canvod/readers/cli.py
def main():
    import sys
    print(f"Reading {sys.argv[1]}")
```

Troubleshooting Builds

Common Issues **1. Module not found**

```
ModuleNotFoundError: No module named 'canvod.readers'
```

Fix: Check `module-name` in `pyproject.toml`

2. Wrong package structure in wheel

```
# Wrong: canvod_readers/...
# Right: canvod/readers/...
```

Fix: Use dotted `module-name = "canvod.readers"`

3. Missing dependencies in wheel

```
# dependencies not in dependencies list
```

Fix: Add to `[project.dependencies]`

4. Build fails

```
Build backend returned an error
```

Fix: Check `pyproject.toml` syntax with:

```
uv build - -verbose
```

Summary **Build system hierarchy:**

```
pyproject.toml
  ↓
[build -system] \rightarrow uv_build
  ↓
[tool.uv.build -backend] \rightarrow module -name config
  ↓
Source code \rightarrow Wheel + Source dist
  ↓
PyPI \rightarrow pip install
  ↓
User's computer
```

Key files:

- `pyproject.toml` - Configuration
- `uv.lock` - Locked dependencies
- `dist/*.whl` - Built wheels

- `dist/*.tar.gz` - Source distributions

Key commands:

```
uv build          # Build package
uv publish        # Publish to PyPI
just bump patch   # Version bump
just dist         # Build all packages
```

Next Steps

- Architecture - Project structure
- Namespace Packages - How namespaces work
- Development Workflow - Daily development
- Tooling - All tools explained

2 Packages

2.1 canvod-readers

GNSS data format readers

2.1.1 Installation

```
uv pip install canvod -readers
```

2.1.2 Usage

```
from canvod.readers import ...
```

2.1.3 Development

See the main repository README for workspace development setup.

2.2 canvod-aux

Auxiliary data handling

2.2.1 Installation

```
uv pip install canvod -aux
```

2.2.2 Usage

```
from canvod.aux import ...
```

2.2.3 Development

See the main repository README for workspace development setup.

2.3 canvod-grids

HEALPix and hemispheric grid operations

2.3.1 Installation

```
uv pip install canvod -grids
```

2.3.2 Usage

```
from canvod.grids import ...
```

2.3.3 Development

See the main repository README for workspace development setup.

2.4 canvod-vod

Vegetation Optical Depth calculations

2.4.1 Installation

```
uv pip install canvod -vod
```

2.4.2 Usage

```
from canvod.vod import ...
```

2.4.3 Development

See the main repository README for workspace development setup.

2.5 canvod-store

Icechunk and Zarr storage backends

2.5.1 Installation

```
uv pip install canvod -store
```

2.5.2 Usage

```
from canvod.store import ...
```

2.5.3 Development

See the main repository README for workspace development setup.

2.6 canvod-viz

Visualization and plotting utilities

2.6.1 Installation

```
uv pip install canvod -viz
```

2.6.2 Usage

```
from canvod.viz import ...
```

2.6.3 Development

See the main repository README for workspace development setup.

2.7 canvodpy (umbrella)

2.7.1 canvodpy

Umbrella package providing unified access to all canVOD packages.

Installation

```
uv pip install canvodpy
```

Usage

```
import canvodpy

# Access all subpackages
from canvod.readers import Rnxv3Obs
from canvod.grids import HemiGrid
```

Development See the main repository README for workspace development setup.

3 Development

3.1 Contributing Guide

3.1.1 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

Types of Contributions

Report Bugs Report bugs at https://github.com/nfb2021/canvodpy/issues. If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation canVODpy could always use more documentation, whether as part of the official docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback The best way to send feedback is to file an issue at https://github.com/nfb2021/canvodpy/issues.
If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started! Ready to contribute? Here's how to set up `canvodpy` for local development.

1. Fork the `canvodpy` repo on GitHub.
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/canvodpy.git
cd canvodpy
```

3. Install your local copy into a virtualenv. Assuming you have uv and Just installed:

```
# Setup virtual environment
uv sync

# Install pre -commit hooks
just hooks
```

4. Create a branch for local development:

```
git checkout -b name -of -your -bugfix -or -feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes are formatted correctly and pass tests:

```
# Lint, format, typecheck and run tests
just check
just test

# Or run full CI suite
just ci 3.13
```

6. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name -of -your -bugfix -or -feature
```

7. Submit a pull request through the GitHub website.

Pull Request Guidelines Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.md.
3. The pull request should work for Python 3.13+. Tests run in GitHub Actions on every pull request to the main branch.

Workspace Development This project uses a monorepo structure with multiple packages. When developing:

- Work on individual packages in `packages/` or `canvodpy/`
- Run package-specific commands: `just check-package canvod-readers`
- Run workspace-wide commands: `just check`, `just test`
- All packages share a single lockfile and virtual environment

Code Quality Standards We use:

- **ruff** for linting and formatting (ALL rules enabled)
- **ty** for type checking
- **pytest** for testing with coverage

Run `just check` before committing to ensure code quality.

Deploying A reminder for the maintainers on how to deploy. Make sure all your changes are committed. Then run:

```
just bump minor  # possible: major / minor / patch
git push
git push - -tags
```

GitHub Actions will automatically publish to PyPI when you push a new tag.