# Managing Third-Party Libraries: A Survival Guide for Data Scientists

Francesco P. Lovergine <francesco@lovergine.com>

November 25, 2025

# Table of Contents

# Who am I?

- Researcher at CNR-IREA in Italy
- One of the $\sim 1,000$ Debian developers (as *frankie@debian.org*)
- DebianGis team founder (about 20 years ago)
- Old-school computer scientist and geek
- The culprit of Giuseppe's transformation from a forest scientist to a Linux addict, even

# The Modern Data Science Stack

**The Reality:**

- 100+ dependencies per project
- Multiple languages (Python, R, C/C++)
- Cross-platform requirements
- Rapid ecosystem evolution

**Common Problems:**

- "It works on my machine"
- Version conflicts
- Breaking changes
- Platform-specific issues

**The Goal**

Reproducible, maintainable, and reliable data science environments across Linux, macOS, and Windows.

# What We'll Cover

1. Package management fundamentals
2. System packages vs. language-specific packages
3. Native vs. C/C++-extension packages
4. Version management and discovery
5. Source control best practices
6. Conda vs. Miniforge
7. Containerization strategies
8. Jupyter notebook management
9. Upgrade strategies and dependency resolution

# The Package Management Landscape

**System-Level:**

```
# Debian/Ubuntu
apt install python3-numpy

# RHEL/Fedora
dnf install python3-numpy

# macOS
brew install numpy

# GNU Guix
guix install python-numpy
```

**Language-Level:**

```
# Python
pip install numpy

# R
install.packages("dplyr")

# Julia
using Pkg; Pkg.add("DataFrames")
```

# System Packages: Pros and Cons

**Advantages:**

- System-wide availability
- Security updates
- Dependency resolution
- Tested for your OS
- No compilation needed

**Disadvantages:**

- Often outdated
- Limited selection
- Requires root/admin
- OS-specific
- Can't have multiple versions

**Recommendation**

Use system packages for: base Python/R, system libraries (BLAS, LAPACK, OpenSSL), development tools.

# Language Package Managers

**Python Ecosystem:**

```
pip install package_name            # PyPI
pip install package_name==1.2.3     # Specific version
pip install -r requirements.txt     # From file
```

**Key Files:**

- `requirements.txt` - Pin exact versions
- `setup.py` / `pyproject.toml` - Package metadata
- `constraints.txt` - Version boundaries

**Best Practice**

Always use virtual environments: `python3 [--system-site-packages | --copies] -m venv myenv`

# Understanding Package Types

## Pure Python Packages

- Written entirely in Python
- Platform-independent
- Examples: `requests`, `click`, `pytest`
- Fast installation, slower execution

## C-Extension Packages

- Contains compiled C/C++/Fortran code
- Platform-specific binaries (wheels)
- Examples: `numpy`, `pandas`, `scipy`
- Requires compilation if no wheel available

# Working with C-Extensions

**Pre-compiled Wheels (Preferred):**

```
pip install numpy  # Downloads wheel if available
```

**Source Installation (Fallback):**

```
# Requires build tools
apt install build-essential python3-dev  # Ubuntu
dnf install gcc python3-devel            # Fedora
brew install gcc                         # macOS

pip install numpy --no-binary :all:      # Force source
```

**Warning**

Source builds need: compiler, headers, BLAS/LAPACK libraries, correct environment variables (CFLAGS, LDFLAGS).

# Performance Considerations

| Package Type | Installation | Runtime |
|---|---|---|
| Pure Python | Fast | Slower |
| C-Extension (wheel) | Fast | Fast |
| C-Extension (source) | Slow | Fast |

## CPU Architecture Matters

Modern packages optimize for:

- SSE/AVX instructions (x86_64)
- Apple Silicon (ARM64)
- CUDA for GPU computing

Choose the right wheel for your hardware!

# Discovering Package Versions

**Installed packages:**

```
pip list                    # All installed
pip show numpy              # Details for one
pip list --outdated         # Update candidates
```

**Available versions:**

```
pip index versions numpy    # All PyPI versions
pip install numpy==4        # Error shows versions
```

**Dependency tree for installed packages:**

```
pip install pipdeptree
pipdeptree                  # Show dependencies
pipdeptree -p numpy         # Reverse dependencies
```

# Version Pinning Strategies

**Exact Pinning:**

```
numpy==1.24.3
pandas==2.0.1
scikit-learn==1.2.2
```

**Pros:** Fully reproducible
**Cons:** No security updates

**Compatible Release:**

```
numpy~=1.24.0    # >=1.24.0, <1.25.0
pandas>=2.0,<3.0
scikit-learn>=1.2
```

**Pros:** Patch updates
**Cons:** Can break

## Recommended Approach

Development: use compatible releases
Production: pin exact versions with `pip freeze > requirements.txt`

# Lock Files for Reproducibility

**Problem:** `requirements.txt` doesn't capture transitive dependencies.
**Solution:** Use lock files:

```
# Poetry
poetry lock
poetry install

# Pipenv
pipenv lock
pipenv install --deploy

# pip-tools
pip-compile requirements.in
pip-sync requirements.txt
```

## Key Benefit

Lock files capture the entire dependency graph with exact versions, ensuring identical environments across machines.

# GitHub Repository: Releases vs. Branches

**Releases/Tags (Stable):**

```
pip install git+https://
  github.com/user/repo.git
  @v1.2.3

# Or download release
pip install https://github.
  com/user/repo/archive/
  v1.2.3.tar.gz
```

**Branches (Development):**

```
pip install git+https://
  github.com/user/repo.git
  @main

# Specific commit
pip install git+https://
  github.com/user/repo.git
  @abc1234
```

**Warning**

Never use @main or @master in production! These are moving targets and break reproducibility.

# When to Use Git Installations

## Appropriate Use Cases

- Testing unreleased bug fixes
- Contributing to development
- Using features not yet in PyPI release
- Private repositories (with authentication)

## Better Alternatives

- Wait for official release
- Fork and create your own release
- Build a wheel and host internally
- Use commit SHA, not branch names

## Production Rule

If it's not on PyPI with a version number, it's not production-ready.

# Handling Private Packages

### Option 1: Private PyPI Server

```
pip install --index-url https://pypi.company.com/simple/ \
    company-package
```

### Option 2: Git with SSH Keys

```
pip install git+ssh://git@github.com/company/private@v1.0.0
```

### Option 3: Build and Share Wheels

```
python -m build           # Creates dist/*.whl
pip install dist/package-1.0.0-py3-none-any.whl
```

# The Conda Ecosystem

## What is Conda?

Cross-platform, language-agnostic package and environment manager. Not just for Python!

**Anaconda:**

- Full distribution (3+ GB)
- 250+ pre-installed packages
- Commercial use restrictions
- defaults channel

**Miniconda:**

- Minimal installer
- Only conda + Python
- Same license as Anaconda
- defaults channel

# Introducing Miniforge

## What is Miniforge?

Community-driven conda installer using conda-forge channel by default.

**Key Advantages:**

- Completely free and open source
- No commercial restrictions
- conda-forge channel (more packages, faster updates)
- Better ARM64/Apple Silicon support
- Faster package builds
- Mamba included (faster resolver)

## Recommendation

Use Miniforge for new projects. It's the future of conda.

# Conda vs. Pip: Key Differences

| Feature | Conda | Pip |
|---|---|---|
| Languages | Any (Python, R, C++) | Python only |
| Binary packages | Yes, always | Wheels when available |
| System libraries | Included | System-dependent |
| Repository | conda-forge | PyPI |

**Important**

You can use both conda and pip in the same environment, but install conda packages first, then pip packages.

# Conda Environment Management

## Creating Environments:

```
conda create -n myproject python=3.11
conda activate myproject
```

## Environment Files (environment.yml):

```
name: myproject
channels:
  - conda-forge
  - defaults
dependencies:
  - python=3.11
  - numpy=1.24
  - pandas=2.0
  - pip:
    - some-pip-only-package==1.0.0
```

# Mamba: The Fast Alternative

**What is Mamba?**

- C++ reimplementation of conda
- Parallel downloads
- Better dependency resolution
- Drop-in replacement: `mamba` instead of `conda`

**Installation:**

```
conda install -n base mamba  # In regular conda
# Or use Miniforge3 (includes mamba by default)
```

**Usage:**

```
mamba install numpy pandas scikit-learn  # Much faster!
mamba create -n myenv python=3.11 numpy
```

**Performance**

Mamba can be 10-100x faster than conda for complex environments.

# When to Containerize

**Use Containers When:**

- Deploying to production
- Complex system dependencies
- Ensuring reproducibility
- Multiple conflicting projects
- Team collaboration
- CI/CD pipelines

**Skip Containers When:**

- Simple exploration
- Interactive development
- GPU access is tricky
- Resource constraints
- Learning/prototyping

**Golden Rule**

Use virtual environments for development, containers for deployment.

# Docker for Data Science

**Basic Dockerfile:**

```
FROM python:3.11-slim

WORKDIR /app

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

# Python dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["python", "train_model.py"]
```

# Multi-Stage Builds

**Optimize image size:**

```
# Build stage
FROM python:3.11 as builder
WORKDIR /app
COPY requirements.txt .
RUN pip install --user --no-cache-dir -r requirements.txt

# Runtime stage
FROM python:3.11-slim
WORKDIR /app
COPY --from=builder /root/.local /root/.local
COPY . .
ENV PATH=/root/.local/bin:$PATH
CMD ["python", "app.py"]
```

**Result**

Build stage has compilers and build tools. Runtime stage is minimal.

# Docker Compose for Data Pipelines

```yaml
version: '3.8'
services:
  jupyter:
    build: .
    ports:
      - "8888:8888"
    volumes:
      - ./notebooks:/app/notebooks
      - ./data:/app/data
    environment:
      - JUPYTER_ENABLE_LAB=yes

  database:
    image: postgres:15
    environment:
      - POSTGRES_PASSWORD=secret
    volumes:
      - pgdata:/var/lib/postgresql/data

volumes:
  pgdata:
```

# Alternative Container Tools

**Docker Alternatives:**

- **Podman**: Daemonless, rootless containers
- **Singularity/Apptainer**: HPC-focused, no root required
- **LXC/LXD**: System containers (not just applications)

**Singularity for HPC:**

- Designed for multi-user clusters
- Better GPU integration
- Can run Docker images
- Single-file containers

**HPC Recommendation**

Use Singularity if deploying to academic/research clusters.

# The Jupyter Challenge

**Common Problems:**

- Kernel doesn't match environment
- Package imports fail mysteriously
- "Works in notebook, fails in script"
- Notebooks aren't reproducible
- Hard to version control

**Root Causes:**

- Global Jupyter vs. project-specific kernels
- Hidden state and execution order
- Cell outputs in version control
- Implicit dependencies

# Setting Up Jupyter Properly

**Install Jupyter in Each Environment:**

```
# Create environment
python -m venv myproject
source myproject/bin/activate

# Install Jupyter + kernel registration
pip install jupyter ipykernel
python -m ipykernel install --user --name=myproject \
    --display-name="Python (myproject)"

# Launch Jupyter
jupyter lab
```

**Key Point**

Each environment gets its own kernel. Select the right kernel in Jupyter.

# Jupyter with Conda/Mamba

```
# Create conda environment
mamba create -n datasci python=3.11 jupyter numpy pandas
mamba activate datasci

# The kernel is automatically registered
jupyter lab

# To see all kernels
jupyter kernelspec list

# To remove a kernel
jupyter kernelspec uninstall myproject
```

**Common Mistake**

Installing Jupyter globally and trying to import project packages. Always install Jupyter in your project environment!

# Version Control for Notebooks

**Problem:** Notebooks contain outputs, metadata, execution counts.

**Solution 1: Strip outputs before committing**

```
# Using nbconvert
jupyter nbconvert --clear-output --inplace notebook.ipynb

# Using nbstripout (recommended)
pip install nbstripout
nbstripout notebook.ipynb

# Git hook (automatic)
nbstripout --install
```

**Solution 2: Use Jupytext**

```
pip install jupytext
jupytext --to py:percent notebook.ipynb  # Sync with .py
```

Commits the .py file instead of .ipynb!

# Notebook to Production Code

## The Workflow

1. Explore in notebooks
2. Extract functions to .py modules
3. Import modules in notebooks
4. Test modules independently
5. Use notebooks for visualization only

**Project Structure:**

- project/
- notebooks/ (exploration.ipynb)
- src/ (data.py, models.py)
- tests/
- requirements.txt

# Jupyter Extensions and Tools

## Useful Extensions:

```
# JupyterLab extensions
pip install jupyterlab-git            # Git integration
pip install jupyterlab-lsp            # Code completion
pip install jupyterlab-code-formatter # Auto-format

# Variable inspector
pip install lckr-jupyterlab-variableinspector

# Table of contents
pip install jupyterlab-toc
```

## Papermill for Notebook Automation:

```
pip install papermill
papermill input.ipynb output.ipynb -p param1 value1
```

# The Upgrade Challenge

## Murphy's Law of Dependencies

"If you upgrade any package, something will break."

**Types of Upgrades:**

1. **System OS upgrade** (Ubuntu 22.04 → 24.04)
2. **Python version upgrade** (3.10 → 3.11)
3. **Major package upgrade** (pandas 1.x → 2.x)
4. **Transitive dependency update**

**Cascade Effects:**

- System upgrade → Python upgrade → rebuild all packages
- NumPy upgrade → Pandas, SciPy, Scikit-learn need updates
- One security patch → entire tree needs resolution

# Safe Upgrade Strategy

### Step 1: Assess Current State

```
pip list --outdated
pip check  # Find conflicts
```

### Step 2: Create Backup

```
pip freeze > requirements-backup.txt
conda env export > environment-backup.yml
```

### Step 3: Test in Isolation

```
python -m venv test-upgrade
source test-upgrade/bin/activate
pip install -r requirements-backup.txt
pip install --upgrade package-to-upgrade
# Run tests!
```

# Handling Dependency Conflicts

**Understanding Conflicts:**

```
$ pip install package-a package-b
ERROR: package-a requires numpy>=1.24
       package-b requires numpy<1.24
```

**Resolution Strategies:**

1. Check if newer versions of both packages work together
2. Use a compatible release constraint: `numpy>=1.23,<1.25`
3. Find alternative packages
4. Contact maintainers or file issues
5. Fork and patch (last resort)

**Pro Tip**

Use `pipdeptree` to understand why a specific version is required.

# Automated Dependency Management

**Dependabot (GitHub):**

```yaml
# .github/dependabot.yml
version: 2
updates:
  - package-ecosystem: "pip"
    directory: "/"
    schedule:
      interval: "weekly"
    open-pull-requests-limit: 10
```

**Renovate Bot:** More powerful, supports multiple ecosystems, custom rules.

**pip-tools:**

```
pip-compile --upgrade requirements.in
# Review changes in requirements.txt
pip-sync requirements.txt
```

# Python Version Upgrades

**Python 3.10 → 3.11 → 3.12 → 3.13**
**Considerations:**

- Check package compatibility on PyPI
- Some C-extensions need time to release wheels
- Performance improvements in newer versions
- New language features (pattern matching, etc.)
- Deprecation warnings matter

## Recommended Approach

1. Wait 3-6 months after Python release
2. Check critical packages first (numpy, pandas, pytorch)
3. Test thoroughly with warnings enabled: `python -W all`
4. Keep one environment on old Python during transition

# Dependency Resolution: Understanding the Algorithms

**Why Resolution Matters:**

- Finding compatible versions across 100+ packages
- Handling conflicting requirements
- Speed vs. correctness trade-offs

**Four Different Approaches:**

1. **Pip**: Backtracking (The Detective)
2. **Conda**: SAT Solver (The Mathematician)
3. **Poetry**: Exhaustive Backtracking (The Planner)
4. **uv**: PubGrub in Rust (The Speedrunner)

# Resolver Comparison: Algorithms and Trade-offs

| Tool | Algorithm | Analogy | Strength | Weakness |
|---|---|---|---|---|
| Pip | Backtracking | The Detective. Follows leads. If a lead goes cold, goes back to the last clue. | Standard. Installed everywhere. | Can be slow on bad paths; resolves only for current machine. |
| Conda | SAT Solver | The Mathematician. Translates requirements into a giant logic formula and solves it all at once. | Handles complex non-Python binaries (C libraries, compilers). | Heavy memory usage; "Solving environment..." can hang. |
| Poetry | Exhaustive Backtracking | The Planner. Meticulously maps out the entire plan before doing anything. | Correctness. Guarantees reproducible builds across teams. | Slow. poetry lock is notoriously time-consuming on large projects. |
| uv | PubGrub (Rust) | The Speedrunner. Sprints through the maze, remembering every dead end so it never double-checks. | Speed. Instant resolution; true universal locking. | Newer tool; ecosystem is still maturing (though rapidly). |

# Resolution Strategies: When to Use Each

**Use Pip when:**

- Simple, pure-Python projects
- Quick prototyping
- Standard environments

**Use Conda when:**

- Scientific computing
- Non-Python dependencies
- Complex C/C++ libraries

**Use Poetry when:**

- Team collaboration
- Publishing packages
- Guaranteed reproducibility

**Use uv when:**

- Speed is critical
- CI/CD pipelines
- Large dependency trees
- Modern Python projects

# Using Poetry for Project Management

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

# Create project
poetry new myproject
cd myproject

# Add dependencies
poetry add numpy pandas scikit-learn
poetry add --group dev pytest black

# Install everything
poetry install

# Run in environment
poetry run python script.py
poetry shell  # Activate environment
```

# Using uv: The Fast Alternative to Poetry

**What is uv?**

- Written in Rust (extremely fast)
- 10-100x faster than pip/poetry
- Drop-in pip replacement
- Built-in virtual env management
- Compatible with standard tools

**Key Features:**

- Single binary install
- Works with existing projects
- Supports pyproject.toml
- Fast dependency resolution
- No separate venv step needed

**When to Use uv**

Choose uv for speed-critical workflows, CI/CD pipelines, or when you want pip-like simplicity with modern performance.

# uv in Practice

**Installation:**

```
# Linux/macOS
curl -LsSf https://astral.sh/uv/install.sh | sh
# Windows
powershell -c "irm https://astral.sh/uv/install.ps1 | iex"
# With pip (fallback)
pip install uv
```

**Basic Usage:**

```
# Create project with virtual environment
uv venv
source .venv/bin/activate  # or .venv\Scripts\activate on Windows
# Install packages (much faster than pip)
uv pip install numpy pandas scikit-learn
# Install from requirements.txt
uv pip install -r requirements.txt
# Compile requirements with dependencies
uv pip compile requirements.in -o requirements.txt
```

# uv vs. Poetry vs. pip: Quick Comparison

| Feature | pip | Poetry | uv |
|---|---|---|---|
| Speed | Baseline | Slow | 10-100x faster |
| Virtual envs | Manual | Built-in | Built-in |
| Lock files | Manual | Yes | Via compile |
| Project init | No | Yes | No |
| Learning curve | Low | Medium | Low |
| Compatibility | 100% | Good | 100% pip |
| Dependency resolution | Good | Excellent | Excellent |
| Production ready | Yes | Yes | Yes |

**Recommendation:** Use uv for speed and pip compatibility, Poetry for full project management with opinionated structure.

# Pyproject.toml (PEP 621)

```
[project]
name = "my-data-project"
version = "0.1.0"
requires-python = ">=3.10"
dependencies = [
    "numpy>=1.24,<2.0",
    "pandas>=2.0",
    "scikit-learn>=1.3",
]

[project.optional-dependencies]
dev = ["pytest>=7.0", "black>=23.0"]
viz = ["matplotlib>=3.7", "seaborn>=0.12"]

[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"
```

# Cross-Platform Considerations

**Linux → macOS → Windows differences:**

```
# File paths
os.path.join('data', 'file.csv')  # Platform-agnostic
pathlib.Path('data') / 'file.csv' # Even better

# Line endings
# Use 'newline=""' in open() for CSV files
```

**Package availability:**

- Some packages are Linux-only (e.g., some system tools)
- Windows requires pre-built wheels for C-extensions
- macOS ARM64 (M1/M2/M3) support still catching up

# Testing Your Environment

```python
# test_environment.py
import sys
import numpy as np
import pandas as pd
import sklearn

def test_imports():
    """Test that critical packages import correctly."""
    assert sys.version_info >= (3, 10)
    print(f"Python: {sys.version}")
    print(f"NumPy: {np.__version__}")
    print(f"Pandas: {pd.__version__}")
    print(f"Scikit-learn: {sklearn.__version__}")

def test_numerical_stability():
    """Test basic numerical operations."""
    arr = np.random.rand(1000, 1000)
    result = np.linalg.inv(arr @ arr.T)
    assert result.shape == (1000, 1000)

if __name__ == "__main__":
    test_imports()
    test_numerical_stability()
    print("Environment OK!")
```

# Quick Reference: Decision Tree

1. **Need reproducibility?** → Lock files + containers
2. **Scientific computing?** → Use conda/miniforge
3. **Pure Python project?** → venv + pip + pip-tools
4. **Production deployment?** → Docker + pinned versions
5. **Jupyter notebook?** → Install in each environment
6. **System libraries needed?** → Use conda or system packages
7. **Bleeding edge features?** → Git + commit SHA
8. **Team collaboration?** → Environment files + documentation

# Best Practices Checklist

- ☐ Use virtual environments (always!)
- ☐ Pin dependencies in production
- ☐ Document system requirements
- ☐ Test on target platforms
- ☐ Keep environment files in version control
- ☐ Automate environment creation
- ☐ Regular security updates
- ☐ Use CI/CD to test environments
- ☐ Monitor for breaking changes
- ☐ Document upgrade procedures

# Common Pitfalls to Avoid

## Don't

- Install packages globally with sudo/admin (with a grain of salt)
- Use `pip install --user` (confusing!)
- Mix conda and pip randomly
- Ignore warnings during installation
- Use latest/master in requirements
- Skip testing after upgrades (because they fail...)
- Commit notebooks with outputs/data
- Upgrade everything at once

# Tools Comparison Matrix

| Feature | venv+pip | uv | conda | poetry | docker |
|---|---|---|---|---|---|
| Learning curve | Low | Low | Medium | Medium | High |
| Speed | Fast | Very Fast | Slow | Fast | Medium |
| Reproducibility | Good | Good | Good | Excellent | Excellent |
| System libs | No | No | Yes | No | Yes |
| Multi-language | No | No | Yes | No | Yes |
| Lock files | Manual | Via compile | Yes | Yes | N/A |
| Python only | Yes | Yes | No | Yes | No |
| Production | Yes | Yes | Yes | Yes | Yes |

# Recommended Workflow

## Daily Development

1. Use conda/mamba OR venv+pip
2. Keep requirements/environment file updated
3. Test regularly
4. Document system dependencies

## Pre-Production

1. Pin all versions exactly
2. Create lock file
3. Build Docker image
4. Test on target platform
5. Document upgrade procedure

# Essential Resources

**Official Documentation:**

- Python Packaging Guide: `https://packaging.python.org`
- Conda Documentation: `https://docs.conda.io`
- Docker Documentation: `https://docs.docker.com`
- PyPI: `https://pypi.org`

**Community Resources:**

- conda-forge: `https://conda-forge.org`
- Python Packaging Authority: `https://www.pypa.io`

**Tools:**

- Poetry: `https://python-poetry.org`
- uv: `https://github.com/astral-sh/uv`
- pip-tools: `https://github.com/jazzband/pip-tools`
- Miniforge: `https://github.com/conda-forge/miniforge`

# Getting Help

**When Things Go Wrong:**

1. Check error messages carefully
2. Search for the error on Stack Overflow/AI overview (with a grain of salt for fake solutions)
3. Check package issue trackers on GitHub
4. Verify your Python/package versions
5. Try in a fresh environment
6. Ask on community forums

**Key Debugging Commands:**

- `pip list` - See what's installed
- `pip check` - Find conflicts
- `pip show package` - Package details
- `python -m site` - Python paths
- `which python` / `where python` - Python location

# Conclusions

## Key Takeaways

- Dependency management is crucial for reproducible data science
- Choose tools based on needs (conda for science, venv for pure Python)
- Always use isolated environments
- Pin versions for production
- Test upgrades carefully
- Document everything
- Containerize for deployment
- Document everything and keep order

**Questions? Confused?**

Happy environment managing!