From Scripts to Packages: A Guide to Distributing Scientific Python

AM215 - LECTURE 5

The AM215 Team

Today's Roadmap

▒▒ Part 1: The Problem of Sharing Code

Examine the limitations of ad-hoc code sharing and the anti-patterns of `sys.path` modification, establishing the need for a formal packaging system.

▒▒ Part 2: Anatomy of a Modern Package

Analyze the `pyproject.toml` standard for declaring project metadata, dependencies, entry points, and build requirements for compiled extensions.

▒▒ Part 3: A Practical Packaging Workflow

Demonstrate the end-to-end packaging workflow: local editable installs, building distribution artifacts (`sdist`, `wheel`), and publishing to a package index.

▒▒ Part 4: Professional Practices

Implement professional practices including automated versioning with `setuptools-scm`, software licensing, and the use of modern, integrated toolchains like `uv`.

Part 1: The "Why" — The Problem of Sharing Code

Our Starting Point: A Multi-File Project

We have a working Blackjack simulation from Lecture 4, split across multiple files:

```
lec04/code/blackjack/
├── deck.py
├── game.py
├── simulation.py
└── ...
```

It works perfectly inside its own directory. But it's a script, not a reusable tool.

> How do we share our `Deck` and `BlackjackGame` classes with a collaborator so they can just `import` them?

The Ad-Hoc "Solutions" and Their Failures

Scenario 1: Emailing `.py` files.

- **Problem:** No versioning, no dependency tracking, requires manual placement in the right folder. Highly error-prone.

Scenario 2: `git clone` the whole repository.

- **Problem:** Forces collaborators to work inside your project structure or manually manage their Python path. Still no automated dependency management.

> These methods are fragile and not reproducible. We need a formal, robust system.

Beyond requirements.txt

In Lecture 3, we used `requirements.txt` and `environment.yml` to capture a project's dependencies. So why do we need something new?

- **Application vs. Library:** A `requirements.txt` file defines the exact environment for a specific *application*. A package defines the dependencies for a reusable *library* that can be imported into many different applications.
- **Versioning and Distribution:** A package is a versioned, distributable unit. You can install `my-library==1.0` or `my-library==1.1`, and `pip` will handle resolving its dependencies. This is a more robust and flexible form of reproducibility than just sharing an environment file.

> Packaging is for distributing **reusable components**, not just replicating a single project's environment.

How import Really Works: sys.path

When you write `import my_module`, Python doesn't search your whole computer. It iterates through a specific list of directories.

That list is `sys.path`.

```
>>> import sys
>>> from pprint import pprint
>>> pprint(sys.path)
['',  # Current working directory
 '/usr/lib/python311.zip',
 '/usr/lib/python3.11',
 '/usr/lib/python3.11/lib-dynload',
 '/home/user/.local/lib/python3.11/site-packages', # User installs
 '/usr/local/lib/python3.11/dist-packages',      # System-wide installs
]
```

The "Easy Fix": Hacking sys.path

If Python just searches sys.path, can't we just add our project's path to it?

Method 1: In Python

```
import sys
sys.path.append('/home/user/projects/card_games/src')
```

Method 2: In your Shell

```
export PYTHONPATH="/home/user/projects/card_games/src"
```

> This seems like a simple way to make your card_games library available everywhere.

Why Modifying `sys.path` is an Anti-Pattern

This creates a fragile, implicit dependency on your machine's specific folder structure. It is a direct cause of "it works on my machine" errors.

The Problems:

- **Not Portable:** Your script will fail on a collaborator's machine because their paths are different.
- **Hard to Debug:** The dependency is hidden. Imagine `project_A` needs `cool_lib_v1` and you add it to `PYTHONPATH`. If you later run `project_B`, it might accidentally import `cool_lib_v1` and fail in a confusing way, because the dependency is implicit and not declared in `project_B`.
- **Causes Conflicts:** You can easily create situations where different projects implicitly depend on different versions of a library.

> We need a formal system to manage installations into a known, reliable location (`site-packages`).

The src/ Layout: Preventing Import Ambiguity

A common source of "it works on my machine" errors is accidentally importing local code instead of the installed package.

Problem: Without a src/ layout, if you are in your project root, import card_games might import the local card_games/ directory. This can hide installation problems, as your local tests will pass even if the package is not correctly installed.

```
my_project/
├── card_games/  <-- Ambiguous import
│   └── ...
└── tests/
    └── test_deck.py
```

Solution: The src/ layout forces you to test against the *installed* version of your code, as the source is no longer on the top-level sys.path.

```
my_project/
└── src/
    └── card_games/
        └── ...
```

> This structure ensures that what you test is what your users will get.

A Better Way: The Python "Package"

The first step toward a formal, distributable unit is to structure our code as a **package**.

- **Module:** A single `.py` file.
- **Package:** A directory of modules that can be imported as a single entity.

The key is a special file: `__init__.py`. This file tells Python that a directory should be treated as a package.

Using __init__.py to Define a Public API

The `__init__.py` file can be empty, but its real power is defining a clean public interface for your package by controlling what is exposed to the user.

```
# src/card_games/__init__.py

# Expose these classes at the top level of the package
from .deck import Deck, Card
from .game import BlackjackGame, JokerBlackjackGame
from .simulation import main as run_simulation
```

Before (messy): `from card_games.deck import Deck`
After (clean): `from card_games import Deck`

> This hides the internal file structure from your users, creating a more stable and professional API.

The Missing Piece: Dependencies & Metadata

Our `card_games` package is now well-structured, but we still have unanswered questions:

- What if `game.py` required `numpy` to run an analysis? How would we declare that?
- How do we give our package a version number, like `card_games-v1.0`?
- How do we list ourselves as the authors?

> This motivates the need for a packaging system that handles more than just code structure.

The First Attempt: setup.py

The classic solution to the metadata problem was a file called setup.py.

```
# The OLD way: setup.py
from setuptools import setup

setup(
    name="card_games",
    version="0.1.0",
    packages=["card_games"],
    install_requires=["numpy"],
)
```

The Problem: setup.py was a Python script. It was **executable**, not **declarative**. To know the package's dependencies, a tool like pip had to run the script, which was inefficient and a potential security risk.

The PEP Revolution: A Declarative Approach

The Python community uses **PEPs (Python Enhancement Proposals)** to formally propose and agree on new standards.

Several key PEPs modernized the packaging ecosystem:

- **PEP 518:** Introduced `pyproject.toml` as a standard file to declare build system requirements.
- **PEP 517:** Defined a standard way for build tools (like `pip`) to communicate with build backends (like `setuptools`).

> This revolution moved the ecosystem from executable scripts to a declarative standard. Because `pyproject.toml` is static, a tool like `pip` can determine build dependencies (e.g., `setuptools`, `cython`) *before* executing any project code. This is faster, more secure, and avoids complex dependency resolution problems.

The Core Principle: Separating "What" from "How"

The modern system separates the *declaration* of your project from the *implementation* of the build process.

1. `pyproject.toml` **(The "What")**
    - A static, declarative file.
    - It states **WHAT** your project is: its name, version, and dependencies.
2. **Build Backend (The "How")**
    - A tool like `setuptools` or `flit`.
    - It reads `pyproject.toml` and handles **HOW** to build the package.
    - For our examples, we will use `setuptools`, the most common build backend.

> This separation is the central, most important principle of modern Python packaging.

What is a Build Backend?

The build backend is the engine that turns your source code into distributable packages (`sdist` and `wheel`). Its responsibilities include:

- **Collecting Files:** Finding all your source files, `pyproject.toml`, `README.md`, and `LICENSE`.
- **Compiling Code:** If your project includes extensions written in C, C++, Fortran, or Cython, the backend manages the compilation process. This is its most critical and complex job.
- **Generating Metadata:** Processing the `[project]` table from `pyproject.toml` into a standard format.
- **Creating Artifacts:** Packaging everything into the final `.tar.gz` (sdist) and `.whl` (wheel) files according to official specifications.

> The backend does the heavy lifting, while `pyproject.toml` just provides the instructions.

Choosing a Build Backend

Different backends are optimized for different kinds of projects. You specify your choice in `pyproject.toml`.

- `setuptools`: The classic, most powerful, and feature-rich backend. It's the safe default and handles complex projects with C/Cython extensions well.
  - `build-backend = "setuptools.build_meta"`
- `flit-core`: A simple, fast backend for pure-Python packages with no complex build steps.
  - `build-backend = "flit_core.buildapi"`
- `hatchling`: A modern, extensible, and standards-based backend that is part of the `hatch` toolchain.
  - `build-backend = "hatchling.build"`
- `scikit-build-core` / `meson-python`: Specialized backends for projects that use external build systems like CMake or Meson, common in scientific computing.

> **Recommendation:** Start with `setuptools`. If your project is simple and pure-Python, consider `flit` or `hatch`.

Part 2: The "What" — Anatomy of a Modern Package

A Minimal Package

Let's start with the simplest possible package structure and configuration.

The Structure:

```
card_games_project/
├── pyproject.toml
└── src/
    └── card_games/
        ├── __init__.py
        ├── deck.py
        └── game.py
```

The Configuration (pyproject.toml):

```toml
# The build system declaration (PEP 518)
[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"

# The project metadata (PEP 621)
[project]
name = "card_games"
version = "0.1.0"
```

> This is a complete, valid, and buildable Python package.

Declaring Dependencies & Metadata

Now we can add more information to our `[project]` table.

```
[project]
name = "card_games"
version = "0.1.0"
authors = [
  { name="Your Name", email="you@example.com" },
]
description = "A simple library for card game simulations."
requires-python = ">=3.9"
dependencies = [
    # Our core Deck and Game classes have no external dependencies!
]
```

- `authors` / `description`: Essential metadata for PyPI.
- `requires-python`: Specifies the Python versions your code works with.
- `dependencies`: A list of packages required for the core library to function.

Optional Dependencies ("Extras")

Optional dependencies allow users to install extra functionality on demand. This is perfect for dependencies only needed for development, testing, or optional features.

```
[project.optional-dependencies]
plotting = [
    "matplotlib>=3.5.0",
]
test = [
    "pytest",
]
dev = [
    "build",
    "twine",
]
```

- We define groups like `plotting`, `test`, and `dev`.
- A user can install them via `pip install .[plotting]` or `pip install -e '.[dev,plotting,test]'`.

> This is a powerful pattern for separating core logic from heavier dependencies for visualization, testing, or machine learning. (We'll cover `pytest` in a future lecture!)

Making Scripts Executable: Entry Points

How does a user run our simulation after installing the package? We create a command-line "entry point".

```
[project.scripts]
run-blackjack-sim = "card_games.simulation:main"
```

- This creates a command-line script named `run-blackjack-sim`.
- When a user runs it, it will execute the `main` function inside the `card_games.simulation` module.

> This is the standard, robust way to distribute executable applications as part of an installable Python package.

Handling Compiled Code with Cython

In Lecture 4's notes.md, we identified get_hand_value as a performance bottleneck. We can speed this up with **Cython**, a language that translates Python-like code into optimized C extensions.

- **How it works:** Cython compiles .pyx files into .c files, which are then compiled into fast machine code.
- **Why it helps:** By adding static C type declarations (cdef int), we can convert slow Python loops into fast C loops, dramatically improving performance for numerical code.
- **Build Dependency:** The build process now requires cython to perform the .pyx -> .c translation *before* the C compiler runs. This is why it's a build-time dependency.

The Code (fast_rules.pyx):

```
# cython: language_level=3
def get_hand_value_cython(list hand):
    cdef int value = 0, num_aces = 0
    for card in hand:
        if card.rank in 'JQK': value += 10
        elif card.rank == 'A':
            num_aces += 1; value += 11
        else: value += int(card.rank)
    while value > 21 and num_aces:
        value -= 10; num_aces -= 1
    return value
```

Configuring the Build Backend

How do we tell setuptools to compile our Cython code? We use the [tool.setuptools] table in pyproject.toml.

```
[build-system]
requires = ["setuptools>=61.0", "cython"]

[tool.setuptools]
ext-modules = [
    {name = "card_games.fast_rules", sources = ["src/card_games/fast_rules.pyx"]},
]
```

- The [tool] table is the standard place for tool-specific configuration.
- ext-modules tells setuptools to build a list of extension modules.
- Other tools like pytest and ruff (a linter) also use the [tool] table for their configuration.

> This reinforces the "What vs. How" principle: `[project]` is standard, while `[tool]` is for backend-specific instructions.

Including Data Files

Scientific packages often need to bundle non-Python data (e.g., sample datasets, config files). We can tell `setuptools` to include these.

The Setup (`pyproject.toml`):

```
[tool.setuptools.package-data]
# In the 'card_games' package, include all .json files
card_games = ["*.json"]
```

This ensures that if we have a file like `src/card_games/strategies.json`, it will be included in the installed package.

> This is the standard way to bundle data files with your code.

Accessing Package Data

Once data files are bundled, you should **not** use standard file I/O (`open()`) to access them, as their location depends on how the package was installed.

The modern, correct way is to use the `importlib.resources` module.

```
import json
from importlib.resources import files

def load_strategy(name):
    # Traverses the package to find the data file
    path = files('card_games').joinpath(f'{name}.json')
    with path.open('r') as f:
        return json.load(f)

# This works whether the package is in a .zip, installed, or editable
strategy_data = load_strategy('default_strategy')
```

`importlib.resources` provides a reliable way to access package data regardless of installation method.

pyproject.toml: Richer Metadata for PyPI

This optional metadata makes your project more discoverable and professional on PyPI.

```toml
[project]
# ...
readme = "README.md"
keywords = ["cards", "simulation", "blackjack"]
classifiers = [
    "Programming Language :: Python :: 3",
    "Operating System :: OS Independent",
]

[project.urls]
"Homepage" = "https://github.com/your-username/card_games"
"Bug Tracker" = "https://github.com/your-username/card_games/issues"
```

- `readme`: Points to the file that PyPI will render as your project's homepage.
- `keywords`: A list of terms to help users find your package via search.
- `classifiers`: Standardized "tags" from a list curated by PyPI. The `::` syntax creates a browsable hierarchy. You can find the full list on **pypi.org/classifiers** (*https://pypi.org/classifiers/*).

Distribution Formats: Source Distribution (sdist)

When you build your package, you create distribution artifacts. The first is the **sdist**.

- **What it is:** A compressed archive (`.tar.gz`) containing your raw source code and `pyproject.toml`.
- **Purpose:** It's the universal format. It allows users on any platform to download your code and build it themselves.
- **Downside:** Requires the user to have a compatible build environment, which can be slow or fail if compiled code is involved.

Distribution Formats: The Wheel

The second, and preferred, format is the **wheel** (`.whl`), a "built" distribution. It can be installed very quickly without a build step on the user's machine.

The filename contains compatibility tags: `{name}-{version}-{python_tag}-{abi_tag}-{platform_tag}.whl`

- **Pure Python Wheel:** `card_games-0.1.0-py3-none-any.whl`
  - `py3`: Works on any Python 3.
  - `none`: No specific ABI (Application Binary Interface).
  - `any`: Works on any platform.
- **Platform-Specific Wheel (with C/Cython code):** `my_scipy_lib-1.0-cp311-cp311-manylinux_x86_64.whl`
  - `cp311`: CPython version 3.11 only.
  - `cp311`: CPython 3.11's ABI.
  - `manylinux_x86_64`: Linux, 64-bit Intel/AMD.

> The tags tell `pip` if the wheel is compatible with the user's system.

Wheels, Compilers, and the pip vs. conda Story

- **What is an ABI?** The **Application Binary Interface** is a low-level contract that defines how compiled code communicates with the Python interpreter. It can change between Python versions.
- **Benefit of Platform Wheels:** They ship pre-compiled code. This is why you don't need a C or Fortran compiler to `pip install numpy`—you're just downloading the binaries.
- `pip` **vs.** `conda` **revisited:**
  - `pip` can install pre-compiled code *from a wheel*. It cannot compile code itself or manage non-Python dependencies like MKL or CUDA.
  - `conda` is a general-purpose package manager that *can* install compilers and external libraries, making it more powerful for complex scientific stacks.

> Wheels are how the Python-native packaging system handles compiled code distribution.

An Entry Point with Optional Dependencies

A quick note on our package design: the `run-blackjack-sim` entry point calls a function that generates a plot. This means it requires `matplotlib`, which we've defined in the `plotting` optional dependency group.

```
[project.scripts]
run-blackjack-sim = "card_games.simulation:main"

[project.optional-dependencies]
plotting = ["matplotlib>=3.5.0"]
```

This is a valid design choice, but it means a user must install the package with `pip install .[plotting]` to use the command-line script. A more robust design might make the plotting conditional, checking if `matplotlib` is installed at runtime.

The Complete pyproject.toml

```toml
# Build system configuration
[build-system]
requires = ["setuptools>=61.0", "setuptools-scm[toml]>=6.2", "cython"]
build-backend = "setuptools.build_meta"

# Core project metadata
[project]
name = "card_games_am215"
dynamic = ["version"]
authors = [{ name = "AM215 Team", email = "am215@example.com" }]
description = "A simple library for card game simulations."
readme = "README.md"
license = "MIT"
requires-python = ">=3.9"
classifiers = ["Programming Language :: Python :: 3", "Operating System :: OS
Independent"]
dependencies = []

# Optional dependencies (extras)
[project.optional-dependencies]
plotting = ["matplotlib>=3.5.0"]
test = ["pytest"]
dev = ["build", "twine"]

# Command-line scripts
[project.scripts]
run-blackjack-sim = "card_games.simulation:main"

# Backend-specific configuration
[tool.setuptools_scm]

[tool.setuptools]
ext-modules = [
    {name = "card_games.fast_rules", sources = ["src/card_games/fast_rules.pyx"]},
]

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

Part 3: The "How" — A Practical Packaging Workflow

DEMO 1 (Goal): Local Development

Our first goal is to make our `card_games` library usable on our own machine, from any directory, just like `numpy` or `matplotlib`.

The Objective: Install our package in **"editable" mode**.

- This allows us to `import card_games` from anywhere.
- Any changes we make to the source code will be immediately reflected, without needing to reinstall.

This is the standard workflow for developing a Python package.

DEMO 1 (Action): The Editable Install

We'll perform three steps:

1. **Arrange** our files into the `src/card_games/` directory.
2. **Create** our `pyproject.toml` with the `plotting` extra and `scripts` entry point.
3. **Install** the package in editable mode, including the optional dependencies.

```
# From the project root directory (card_games_project/)
pip install -e '.[dev,plotting,test]'
```

Under the Hood: The `.` syntax tells `pip` to install the package in the current directory.

- The `-e` flag makes it "editable" (for developers).
- A user would just run `pip install .` for a standard install.
- `[plotting,test]` includes the optional dependency groups.

How Editable Installs Work

An editable install doesn't copy your code to `site-packages`. Instead, it places a special path configuration (`.pth`) file there that points back to your project's source directory.

```
# Inside .../lib/python3.11/site-packages/
# __editable__.card_games_project.pth

/home/user/projects/card_games_project/src
```

This adds your `src` directory to `sys.path` at runtime, making your package importable while allowing you to edit the code in place.

> It's the principled, tool-managed way to achieve the `sys.path` modification we earlier identified as an anti-pattern when done manually.

DEMO 1 (Verification): Using the Package

The installation is complete. Now for the payoff.

We can now run our simulation from anywhere using the command we created in [project.scripts].

```
# From any directory
$ run-blackjack-sim
Running simulation: Standard Game, Blind Strategy...
Running simulation: Joker Game, Blind Strategy...
...
Generating plot...
Plot saved to blackjack_comparison.png
```

We can also import our library from any directory in a Python session:

```
>>> import card_games
>>> deck = card_games.Deck()
>>> len(deck)
52
```

> Our project is now both an importable library and a distributable command-line tool.

DEMO 2 (Goal & Action): Building the Distributables

Now that we can use our package locally, we need to prepare it for sharing with others. This means creating the standard distribution artifacts.

The Objective: Create the sdist and wheel files.

The Command:

```
# Assumes `build` was installed via `pip install -e '.[dev]'`
python -m build
```

DEMO 2 (Verification): Inspecting the dist/ Directory

The build command creates a new directory called dist/.

```
card_games_project/
├── dist/
│   ├── card_games-0.1.0-py3-none-any.whl  <-- The Wheel
│   └── card_games-0.1.0.tar.gz          <-- The sdist
├── src/
│   └── ...
└── pyproject.toml
```

- The .tar.gz is the source distribution.
- The .whl is the built distribution (wheel).

> These two files are what you would upload to PyPI for others to download and install.

The Challenge of Portable Linux Wheels

When we built our package with a Cython extension, it created a wheel with a platform tag like `linux_x86_64`.

If you try to upload this to PyPI, it will be **rejected** with a `400 Bad Request` error.

Why? A wheel built on your specific Linux machine is linked against your specific versions of system libraries (like `glibc`). This wheel is not guaranteed to work on another Linux machine with different library versions.

PyPI enforces portability by requiring that Linux wheels adhere to the `manylinux` **standard**. A `manylinux` wheel is built in a special, minimal environment and bundles any required external libraries, making it compatible with nearly all modern Linux distributions.

The Magic of auditwheel

So how does the manylinux container turn a linux_x86_64 wheel into a portable manylinux wheel? It uses a tool called auditwheel.

The auditwheel Workflow:

1. **Build:** First, python -m build creates a normal, platform-specific wheel (e.g., ...-linux_x86_64.whl).
2. **Repair:** Then, auditwheel repair inspects this wheel.
   - It finds all the external shared libraries (.so files) that the compiled code depends on.
   - It copies these libraries *into* the wheel itself.
   - It rewrites the compiled code to look for the bundled libraries in a special location inside the wheel.
3. **Re-tag:** Finally, it renames the wheel with the correct manylinux tag (e.g., ...-manylinux_2014_x86_64.whl).
4. **Clean Up:** The original, non-portable wheel (...-linux_x86_64.whl) is removed, leaving only the manylinux wheel for publishing.

> auditwheel is the essential final step that bundles dependencies and makes Linux wheels truly portable.

Solutions for Building Wheels

So how do we create distributable packages if we have compiled code?

1. The Easy Way Out: Source-Only Distribution You can choose to build and upload only an sdist.

```
python3 -m build --sdist
```

- **Pro:** Simple and universal.
- **Con:** The user must have a C compiler and all necessary build tools installed on their machine, which is a significant burden.

2. The Right Way: Build a manylinux Wheel Use a pre-built Docker container provided by the Python Packaging Authority (PyPA) to build your wheels. This is what we do in our course demos.

```
# Inside a manylinux container...
python3 -m build
```

- **Pro:** Produces a portable wheel that can be uploaded to PyPI and installed easily by users.
- **Con:** Requires using Docker or a similar container technology.

DEMO 3 (Goal & Action): Publishing to TestPyPI

It's time to share our package with the world. We'll use **TestPyPI**, a sandbox for practicing uploads.

The Objective: Upload our package to a public repository.

The Command:

```
# Assumes `twine` was installed via `pip install -e '.[dev]'`
twine upload --repository testpypi dist/*
```

DEMO 3 (Verification): Installing from the Internet

The final test: can a user install our package and its optional features from the internet?

We'll create a new, completely empty virtual environment to simulate a user's machine.

```
# Create and activate a new, clean environment
python -m venv test_env
source test_env/bin/activate

# Install from TestPyPI with the plotting extras
pip install --index-url https://test.pypi.org/simple/ "card_games_am215[plotting]"
```

After installation, the user can run the simulation:

```
$ run-blackjack-sim
```

> This confirms our package is correctly configured for both core functionality and optional features.

Part 4: Professional Practices and the Ecosystem

What is a git tag?

While a branch is a pointer that moves as you add commits, a **tag** is a permanent, human-readable pointer to a *single, specific commit*.

```
# Create an "annotated" tag for release v1.0.0
git tag -a v1.0.0 -m "First stable release"

# Push the tag to the remote repository
git push origin v1.0.0
```

- The `-a` flag creates an **annotated tag**, which is a full object in Git that stores the author, date, and a message. This is best practice for releases. The `-m` flag provides the message.
- By default, `git push` does not send tags to the remote. You must push them explicitly with `git push origin <tagname>` or `git push --tags`.

> Tags are the standard way to mark official release points in your project's history, making them perfect for versioning.

Versioning with Semantic Versioning (SemVer)

A version number is a contract with your users. **Semantic Versioning (**MAJOR.MINOR.PATCH**)** is the standard for communicating the nature of your changes.

- MAJOR **(e.g., 1.0.0 -> 2.0.0):** For incompatible API changes. Users will expect their code to break and will need to make changes to upgrade.
- MINOR **(e.g., 1.0.0 -> 1.1.0):** For adding new functionality in a backward-compatible way.
- PATCH **(e.g., 1.0.0 -> 1.0.1):** For backward-compatible bug fixes.

> Adhering to SemVer builds trust with your users.

Automated Versioning with setuptools-scm

Manually bumping the version string in pyproject.toml is tedious and error-prone. We can automate this by deriving the version directly from git tags.

The Tool: setuptools-scm

The Setup (pyproject.toml):

```toml
[build-system]
requires = ["setuptools>=61.0", "setuptools-scm"] # Add setuptools-scm

[project]
name = "card_games"
dynamic = ["version"] # Change version to dynamic

[tool.setuptools_scm]
# This section enables the tool. Usually no config is needed.
```

> Your git history becomes the single source of truth for your version numbers.

The setuptools-scm Workflow

The Workflow:

1. Finalize code and commit.
2. Tag the commit: `git tag -a v0.1.0 -m "Version 0.1.0 release"`.
3. Push the tag: `git push --tags`.

Under the Hood: During the build, `setuptools-scm` generates a file (e.g., `_version.py`) inside your package. To access this version at runtime, you can import it in your package's `__init__.py`:

```
from ._version import version as __version__
```

Crucial Gotcha: "Dirty" Builds `setuptools-scm` will add a "local version" identifier (e.g., `+d20241003`) if your git repository has uncommitted changes or untracked files. **PyPI rejects these versions.**

> **Solution:** Always add build artifacts (`dist/`, `build/`) and cache files (`__pycache__/`) to a `.gitignore` file to ensure a clean build.

The Importance of a License

This is one of the most critical and misunderstood aspects of sharing code.

If your code does not have a license, it is NOT open source.

By default, all creative work (including code) is under exclusive copyright. Without a license, no one can legally use, copy, distribute, or modify your work.

> For science to be open and reproducible, and for others to build upon your work, you **must** include a license.

The Licensing Spectrum: Permissive vs. Copyleft

There are two main families of open-source licenses.

Permissive (e.g., MIT, Apache 2.0, BSD)

- **Philosophy:** "Do what you want, just keep my name on it and don't sue me."
- **Rules:** Allows others to use your code in any project (open-source or proprietary) as long as they give you credit.
- **Use Case:** The most common choice for academic and scientific code, as it encourages the widest possible adoption.

Copyleft (e.g., GPL, AGPL)

- **Philosophy:** "Software should be free, and derivatives of it should also be free."
- **Rules:** If someone uses your GPL-licensed code in their project, their project must also be licensed under the GPL.
- **Use Case:** Ensures that the code and its derivatives remain in the open-source ecosystem.

The GPL, "Viral" Licensing, and Modern Debates

The "copyleft" philosophy is championed by Richard Stallman and the Free Software Foundation. The GPL is designed to ensure that software, and any derivative works, remain free.

- **"Viral" Nature:** Because the GPL's requirements "spread" to any software that uses GPL-licensed code, some companies have historically viewed it as a "virus" to be avoided in proprietary products.
- **Real-World Impact:** Apple's macOS ships with a very old version of `bash` (v3.2) because newer versions are licensed under GPLv3, which has stricter terms Apple wishes to avoid. They switched the default shell to `zsh` (which uses a permissive MIT-like license) partly for this reason.
- **Modern Debates:** The use of GPL-licensed code from GitHub to train large language models has sparked intense legal and ethical debates about what constitutes a "derivative work" in the age of AI.

How to Add a License

Adding a license is simple.

1. **Choose a license.** For most scientific projects, the **MIT License** is an excellent default.
2. **Create a** `LICENSE` **file** in your project's root directory.
3. **Copy the full text** of your chosen license into that file.
4. (Optional but good practice) **Declare the license** in your `pyproject.toml` using its SPDX identifier:

```
[project]
# ...
license = "MIT"
```

A Modern, Integrated Workflow with uv

We've seen the standard workflow using separate tools like pip, build, and twine.

The uv tool, which we introduced for fast environment management, also provides an integrated, all-in-one solution for packaging. It can handle initialization, building, and publishing with a single, consistent command-line interface.

> This simplifies your toolchain and provides a fast, modern experience.

The uv Packaging Workflow

uv streamlines the entire process, replacing multiple tools.

1. Initialize the project:

```
# Creates an interactive prompt to generate pyproject.toml
uv init
```

This guides you through creating a standards-compliant pyproject.toml, preventing common errors and lowering the barrier to entry.

2. Build the distributables:

```
# Replaces `python -m build`
uv build
```

3. Publish to a repository:

```
# Replaces `twine upload`
uv publish --publish-url https://test.pypi.org/legacy/ dist/*
```

> uv provides a unified experience for the entire packaging lifecycle, from environment management to publishing.

Summary & Resources

We've gone from a collection of scripts to a fully distributable Python package.

The Core Workflow:

1. **Structure:** Use the `src/` layout.
2. **Configure:** Define metadata and dependencies in `pyproject.toml`. Use tools like `setuptools-scm` for automated versioning.
3. **Install:** Use `pip install -e .` for local development.
4. **Build:** Use `python -m build` (or `uv build`) to create `sdist` and `wheel` artifacts.
5. **Publish:** Use `twine` (or `uv publish`) to upload to TestPyPI, then PyPI.

Further Reading:

- **Python Packaging User Guide** (*https://packaging.python.org/*) (The official guide)
- `uv` **Documentation** (*https://astral.sh/uv*)
- **setuptools-scm Documentation** (*https://github.com/pypa/setuptools_scm/*)
- **Cython Documentation** (*https://cython.readthedocs.io/en/latest/*)
- **choosealicense.com** (*https://choosealicense.com/*) (An easy-to-use license chooser)
- **PEP 621** (*https://peps.python.org/pep-0621/*) (The standard for the `[project]` table)