Reproducibility: Collaboration and Credibility in Scientific Computing

AM215 - LECTURE 3

The AM215 Team

The Collaboration Challenge

A collaborator sends you their new Monte Carlo simulation for a paper.

The email says:

> "Here's the script. It works great on my machine! Just run `python mc_simulation.py` and you should see the results."

You clone the repository and try to run it. This is where our journey begins.

The Stakes: A Cautionary Tale

In 2010, economists Carmen Reinhart and Kenneth Rogoff published "Growth in a Time of Debt," a highly influential paper arguing that high public debt slows economic growth. It was widely cited to justify austerity policies worldwide.

The problem: A graduate student, Thomas Herndon, couldn't reproduce their results while trying to replicate their work for a class project.

The fallout:

- Herndon and his professors discovered critical errors in the original Excel spreadsheet.
- A simple formula error accidentally excluded the first 5 countries from a key average.
- The corrected analysis showed a much weaker link between debt and growth.
- The paper's central conclusion was undermined, sparking a global debate about the evidence behind austerity.

> This wasn't fraud; it was a computational error with massive real-world consequences. Reproducibility is the bedrock of scientific credibility.

Where Would You Start?

The mystery: Same code, same seed, different numbers.

What could cause this?

Our Debugging Journey

The first challenge in collaboration isn't getting the *same* answer; it's getting *any* answer at all.

Today, we'll tackle this from the ground up:

1. **Get it running:** Solve the initial "it won't even start" problem.

2. **Get it right:** Ensure everyone gets the same results.

3. **Get it right, anywhere:** Build a workflow that is portable and robust.

Our goal: Move from "it works on my machine" to "it works, correctly, on **any** machine."

Attempt #1: Just Run It

Following our collaborator's instructions, we run the script. It immediately fails.

```
$ python mc_simulation.py
Traceback (most recent call last):
  File "/home/user/mc-project/mc_simulation.py", line 5, in <module>
    import scipy.stats
ModuleNotFoundError: No module named 'numpy'
```

> The script depends on a package (`numpy`) that isn't installed on our machine. The next logical step is to install it.

Attempt #2: The "Obvious" Fix

We try to install the missing package using pip.

```
$ pip install numpy
error: externally-managed-environment
× This environment is externally managed
╰─> To install Python packages system-wide, try apt
install
    python3-xyz, or break the system packages with
    --break-system-packages.
...
note: If you believe this is a mistake, please contact
your system's
      packager.
```

> We've hit a new wall. Our OS is actively preventing us from installing packages globally to protect its own tools. We need a way to manage dependencies for *this project* in an isolated way.

The Core Problem: A Shared, Global Environment

The `externally-managed-environment` error is a symptom of a larger problem: by default, Python installations are **global**.

This leads to two major issues:

1. **System Integrity:** Installing project-specific packages globally can break OS tools that depend on system Python (the error we just saw).

2. **Project Conflicts:** Project A needs `numpy==1.26`, but Project B needs `numpy==2.0`. You can't have both installed globally at the same time.

> We need a way to create isolated, per-project environments.

Solution: Python Virtual Environments (`venv`)

The standard tool for this is `venv`. It creates a lightweight, self-contained directory with a project-specific Python installation and its own packages.

The Workflow:

1. **Create** the environment (a new directory):

   ```
   python3 -m venv sim_env
   ```

2. **Activate** it (modifies your shell's PATH):

   ```
   source sim_env/bin/activate
   ```

Your shell prompt will change to show `(sim_env)`.

3. **Install** packages into the isolated environment:

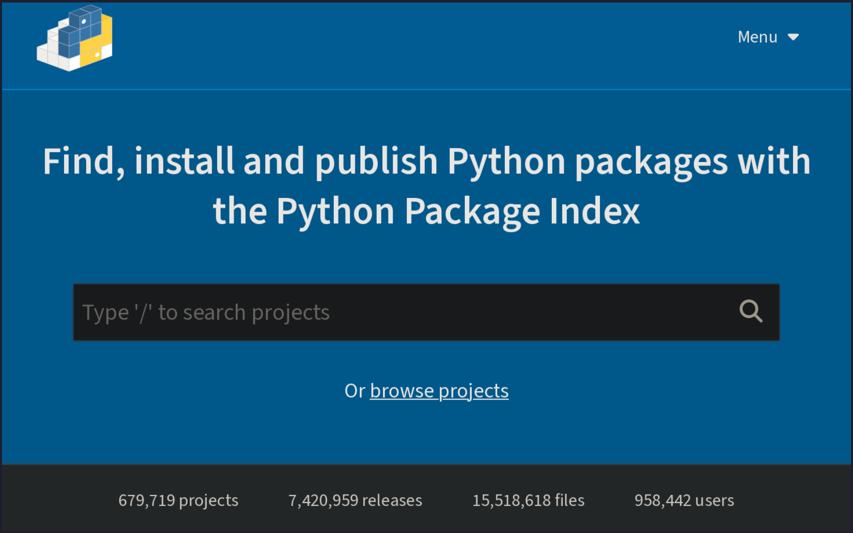
   ```
   pip install scipy numpy matplotlib
   ```

> Now, `python` and `pip` commands are local to this project. Deactivating (`deactivate`) or deleting the `sim_env` folder restores the global state.

What are pip and PyPI?

- **PyPI** (*https://pypi.org*) **(Python Package Index):** The official, community-hosted repository for nearly all open-source Python packages. Think of it as the App Store for Python.
- pip**:** The standard package installer for Python. It's the tool that downloads packages from PyPI and installs them into your environment.

When you run pip install scipy, pip connects to PyPI, finds the scipy package, resolves its dependencies (like numpy), and installs them.



Find, install and publish Python packages with the Python Package Index

679,719 projects 7,420,959 releases 15,518,618 files 958,442 users

Step 1: Documenting Dependencies

Now that our script runs, we must document its dependencies so others can replicate our environment. The standard is a `requirements.txt` file.

A first attempt (`requirements.txt`):

```
# Our project's dependencies
numpy
scipy
matplotlib
```

To install from the file:

```
pip install -r requirements.txt
```

> **The Problem:** This installs the *latest* versions of these packages. Six months from now, "latest" will be different, and our code might break. This is not reproducible.

Step 2: Pinning Versions for Reproducibility

To ensure everyone uses the exact same code, we "pin" the versions using `==`.

A reproducible `requirements.txt`:

```
# Pinned versions for reproducibility
numpy==1.26.4
scipy==1.12.0
matplotlib==3.8.3
```

Now, `pip install -r requirements.txt` will always install these exact versions.

How to generate this file? After getting your code to work, run `pip freeze`:

```
pip freeze > requirements.txt
```

> `pip freeze` automatically captures the exact versions of all packages currently in your environment. This is the key to basic reproducibility.

The Limit of venv and pip

We have a reproducible requirements.txt, but a new collaborator using Python 3.12 tries to install it...

```
(sim_env) $ pip install -r requirements.txt
```

The Problem: Something in our pinned packages is not compatible with Python 3.12.

venv only isolates packages; it **cannot** change the version of Python itself. It just uses whatever python3 you started with.

> We need a tool that can manage Python versions *for us*.

A Modern Solution: uv

uv is an extremely fast, all-in-one Python package manager developed by Astral (the creators of ruff).

Why is it so popular?

- **Blazing Fast:** Written in Rust, it's often 10-100x faster than pip.
- **All-in-One:** It's a single binary that replaces pip, venv, pip-tools, and more.
- **Drop-in Replacement:** You can use it with your existing requirements.txt files.

> uv solves many of the pain points of traditional Python packaging with a focus on speed and user experience.

uv in Action

uv can create a virtual environment with a specific Python version, solving our previous problem.

1. Create an environment with Python 3.11:

```
uv venv --python 3.11
```

uv will download and install Python 3.11 for you if it's not already available.

2. Install dependencies (now it works!):

```
uv pip install -r requirements.txt
```

> uv streamlines the workflow by managing both the Python interpreter and the packages within the environment.

Ensuring Exact Environments: `uv pip sync`

While `uv pip install` adds packages, a more robust command is `sync`.

```
uv pip sync requirements.txt
```

What `sync` does:

- **Installs** any packages from `requirements.txt` that are missing.
- **Uninstalls** any packages in the environment that are *not* in `requirements.txt`.

> `uv pip sync` ensures the environment is an **exact** reflection of the requirements file, preventing issues from stray, manually-installed packages.

A Deeper Problem: The Dependency Iceberg

Even with uv managing our Python environment, a new problem emerges: our numerical results *still* differ between machines.

The Visible Dependencies (managed by pip/uv):

- numpy, scipy, matplotlib

The Hidden Dependencies (the rest of the iceberg):

- **Numerical Libraries:** MKL, OpenBLAS, LAPACK. The linear algebra routines that numpy calls. Different libraries can give slightly different results.
- **System Libraries:** glibc, compilers (gcc), etc.
- **Hardware:** CPU architecture (x86 vs. ARM).

> pip and uv only manage the tip of the iceberg. For scientific code, the hidden part matters.

Solution: Conda for Full-Stack Management

Conda is a language-agnostic package and environment manager.

Conda's Superpower: It manages the *entire* software stack, not just Python packages.

- Python interpreters (like `uv`)
- Python packages (like `pip`)
- **And** non-Python libraries like MKL, CUDA, GCC, and more.

> Conda solves the "dependency iceberg" problem by managing dependencies from the hardware drivers all the way up to your Python script.

Conda Environments: environment.yml

Conda uses an environment.yml file to define the full stack.

```yaml
name: mc-sim-reproducible
channels:
  - conda-forge
dependencies:
  - python=3.11
  - numpy=1.26
  - scipy=1.12
  - mkl=2023.1 # Explicitly pin the BLAS library!
```

- channels: Locations where Conda looks for packages. conda-forge is the community-driven channel, like PyPI for the Conda ecosystem.
- dependencies: A list of all packages (Python and non-Python).

Conda in Practice: Using `micromamba`

`micromamba` is a lightweight, fast, standalone implementation of the Conda package manager. We'll use it for our demos as it's included in our development container.

Workflow:

1. **Create the environment from the file:**

   ```
   micromamba create -f environment.yml
   ```

2. **Activate the environment:**

   ```
   micromamba activate mc-sim-reproducible
   ```

> `micromamba` uses the same `environment.yml` files as Conda but provides a much faster and more focused experience.

The Final Challenge: Ultimate Portability

Even with Conda, we can face issues:

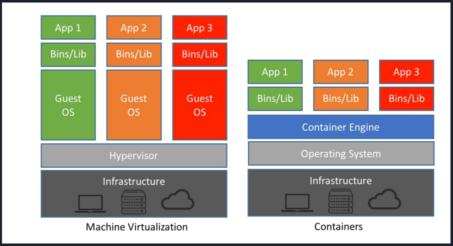
- A collaborator is on Windows and struggles with setup.
- The HPC cluster doesn't have Conda installed.
- A cloud service requires a specific deployment format.

We need to package our *entire* environment—OS, libraries, and code—into a single, runnable unit that works anywhere.

Two approaches:

1. **Virtual Machines (VMs):** Heavyweight, slow, but fully isolated.
2. **Containers:** Lightweight, fast, and the modern standard.

Virtual Machines vs. Containers



Virtual Machine (VM)	Container
Heavyweight (GBs)	**Lightweight** (MBs)
Full Guest OS	Shares Host OS Kernel
Slow to boot (minutes)	Fast to start (seconds)
Hardware virtualization	OS-level virtualization

> Containers provide the isolation of a VM without the overhead, making them perfect for shipping scientific code.

What is a Container?

A container is a standard process on your OS, but with a crucial difference: it's **isolated** using Linux kernel features (namespaces and cgroups).

- **Image:** A read-only template containing an application and its dependencies (e.g., `ubuntu:22.04`, `python:3.11-slim`).
- **Container:** A live, running instance of an image.
- **Dockerfile:** A text file recipe for building an image.
- **Registry:** A place to store and share images (like Docker Hub).

> The **OCI (Open Container Initiative)** standard ensures that images built with one tool (like Docker) can be run by another (like Podman).

The Dockerfile: A Recipe for an Image

This file defines the steps to build our environment.

```
# 1. Start from an official, minimal base image
FROM python:3.11-slim

# 2. Set the working directory inside the container
WORKDIR /app

# 3. Copy requirements and install dependencies
# (This step is cached if requirements.txt doesn't change)
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# 4. Copy the rest of the application code
COPY . .

# 5. Define the default command to run
CMD ["python", "mc_simulation.py"]
```

> Each instruction creates a new "layer" in the image. This allows for efficient caching and rebuilding.

The Core Container Workflow

1. **Build:** Create an image from a Dockerfile.

   ```
   docker build -t mc-sim-app .
   ```

2. **Run:** Execute a command in a new container from the image.

   ```
   docker run --rm mc-sim-app
   ```

3. **Push:** Upload your image to a registry to share it.

   ```
   docker push your-username/mc-sim-app
   ```

4. **Pull:** A collaborator downloads the image.

   ```
   docker pull your-username/mc-sim-app
   ```

> This workflow ensures that everyone is running the exact same bits, from the OS up.

Interacting with Containers

Sometimes you need an interactive shell inside your container for debugging.

Get an interactive shell:

```
# -it gives you an interactive terminal
docker run -it --rm mc-sim-app /bin/bash
```

Manage running containers:

```
# List currently running containers
docker ps

# Stop a running container by name or ID
docker stop <container_id>
```

> The `-it` flag is your key to exploring and debugging your containerized environment interactively.

Managing Images

Your local machine can fill up with images.

List local images:

```
docker images
```

Tag an image before pushing:

```
# Tag your local image with your Docker Hub username
docker tag mc-sim-app your-username/mc-sim-app:v1.0
```

Remove an image:

```
docker rmi mc-sim-app
```

> Tagging is essential for versioning and sharing images on a registry.

Container Housekeeping

Docker can consume a lot of disk space with old images, stopped containers, and build caches.

The magic command to clean up:

```
docker system prune
```

This command will remove:

- All stopped containers
- All networks not used by at least one container
- All dangling images (untagged)
- All build cache

For a deeper clean (use with caution!):

```
docker system prune -a --volumes
```

> Run `docker system prune` periodically to reclaim disk space.

Automating with Scripts

For any non-trivial project, encapsulate your Docker commands in scripts.

`build.sh`:

```bash
#!/usr/bin/env bash
set -euo pipefail
IMAGE_NAME="your-username/mc-sim-app:latest"
docker build -t "$IMAGE_NAME" .
echo "Built image: $IMAGE_NAME"
```

`run.sh`:

```bash
#!/usr/bin/env bash
set -euo pipefail
IMAGE_NAME="your-username/mc-sim-app:latest"
# Mount current directory's 'output' folder into
container's /app/output
docker run --rm -v "$(pwd)/output:/app/output" "$
IMAGE_NAME"
```

> Scripts make your workflow reproducible and easier for others to use.

Advanced Environments: Spack for HPC

Spack is a package manager designed for High-Performance Computing (HPC). You will encounter it on the Harvard cluster.

What is it?

- A flexible package manager for building and installing software from source.
- Manages massive, complex software stacks with thousands of versions and configurations.

Why is it used on clusters?

- It allows multiple versions of the same software (e.g., `gcc@10`, `gcc@11`) to coexist.
- It can optimize builds for the specific hardware of the cluster.

```
# Example: Load a specific version of a tool on a cluster
spack load python@3.10.4
```

> Think of Spack as a system-level tool for cluster admins, while Conda/uv are for users to manage their own project environments.

The Ultimate Reproducibility: Nix

Nix is a package manager and build system that takes reproducibility to its theoretical limit.

The Core Idea:

- It treats package builds like pure functions.
- The inputs (source code, dependencies, build scripts, compiler flags) are cryptographically hashed.
- The output is stored in a unique path derived from the hash (e.g., `/nix/store/s2...-python-3.11.5`).

> If the inputs are identical, the output is **guaranteed** to be bit-for-bit identical, across any machine.

Nix in Practice: Declarative Environments

The most powerful feature of Nix for developers is creating temporary, on-demand environments.

No installation needed:

```
# This command gives you a shell with numpy and scipy
# without installing them permanently.
nix-shell -p python311Packages.numpy python311Packages.scipy
```

You can also define an entire development environment in a `shell.nix` file.

> Nix allows you to pop into a perfectly defined, isolated environment for a specific task and then have it vanish. It's the ultimate "clean room."

Summary: Choosing the Right Tool

Start simple and escalate as your project's needs grow. It's about choosing the appropriate tool for the job, not always reaching for the most complex one.

▒ venv + pip: For solo projects.

- **Feature**: Isolates Python packages.
- **Trade-off**: Fast & small, but Python-only.

▒ uv: For most Python projects.

- **Feature**: Fast, all-in-one, manages Python versions.
- **Trade-off**: Python-only, still uses system libraries.

▒ conda / mamba: For numerical/scientific teams.

- **Feature**: Manages full stack (inc. MKL).
- **Trade-off**: Solves numerical dependencies, but larger environments.

▒ container: For publication/deployment.

- **Feature**: Packages entire OS for maximum portability.
- **Trade-off**: More setup required.

▒ Spack: For HPC/shared clusters.

- **Feature**: Manages complex compiled software.
- **Trade-off**: System-level tool, not for user projects.

▒ Nix: For ultimate reproducibility.

- **Feature**: Bit-for-bit identical builds.
- **Trade-off**: Highest reproducibility, but steep learning curve.

> **The Workflow:** Start with `uv`. If you hit system library issues, move to conda. For ultimate portability and publication, use a `container`.

The Unreliable Results

We've containerized our environment. Everything matches perfectly.

New experiment: Run a bootstrap analysis (1000 resamples) to get confidence intervals.

```
$ docker run mc-sim-bootstrap
Bootstrap CI: [8.76, 9.03]
$ docker run mc-sim-bootstrap  # Run again
Bootstrap CI: [8.52, 9.21]
```

> Same container, different confidence intervals. **Why?**

The Culprit: Uncontrolled Randomness

Let's trace through what happens without explicit seed management.

```python
# In mc_simulation.py (original version)
import numpy as np

def simulate_walks(n_walks, n_steps):
    # Uses global numpy random state!
    steps = np.random.randint(0, 4, size=(n_walks,
n_steps))
    # ... rest of simulation
```

> Every run starts from a different random state → different results.

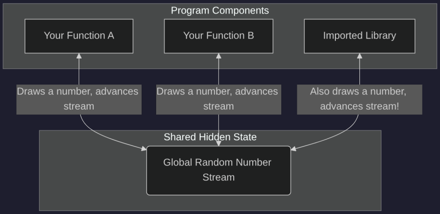
Demo: Global State Chaos

Demo: ./01_demo_reproducibility_fail.py

```python
# The OLD way - global state
np.random.seed(42)
value1 = np.random.random()  # 0.374540

# Another function uses randomness
def unrelated_function():
    return np.random.random()  # Advances global state!

x = unrelated_function()  # Advances the stream

# Try to reproduce, but the state has changed
np.random.seed(42)
_ = unrelated_function()
value2 = np.random.random() # Different from value1!
```

> Global state makes reproducibility fragile and order-dependent.

Why Global State Fails


Program Components
Your Function A
Your Function B
Imported Library
Draws a number, advances stream
Draws a number, advances stream
Also draws a number, advances stream!
Shared Hidden State
Global Random Number Stream

The fundamental problem:

- Single hidden stream shared by a whole program
- Any function can advance it
- Order of operations matters
- Library updates can change call patterns

> In complex projects, global state is impossible to control.

The Modern Solution: Isolated Generators

Demo: ./02_demo_reproducibility_fix.py

```python
# The NEW way - explicit generators
from numpy.random import default_rng

def monte_carlo_pi(n_samples, rng=None):
    """Estimate π using Monte Carlo."""
    if rng is None:
        rng = default_rng()  # Non-reproducible if not provided

    x = rng.uniform(-1, 1, n_samples)
    y = rng.uniform(-1, 1, n_samples)
    inside = np.sum(x**2 + y**2 <= 1)
    return 4 * inside / n_samples

# Reproducible usage
rng = default_rng(42)
pi_estimate = monte_carlo_pi(10000, rng)  # Always 3.1408
```

> Generators are objects you control, not hidden global state.

The Complete Pattern

This ensures your paper's Figure 3 is always reproducible:

```python
# main_analysis.py
SEED = 12345  # Document this in your paper!
rng = np.random.default_rng(SEED)
print(f"Using random seed: {SEED}")

# Pass to all stochastic functions
distances = simulate_walks(N_WALKS, N_STEPS, rng)
bootstrap_ci = compute_bootstrap_ci(distances, rng)
```

> One seed, explicitly passed everywhere = perfect reproducibility.

Gotcha #2: Environment Variables

Some environment variables silently affect reproducibility:

```
# Python 3.3+ dictionary/set iteration order
export PYTHONHASHSEED=0  # Reproducible
export PYTHONHASHSEED=random  # Default, non-reproducible
```

> For complete reproducibility, control `PYTHONHASHSEED` in your scripts.

Gotcha #3: Shell Script Portability

Email from cluster admin:

> "Your script fails with: `/bin/bash: bad interpreter: No such file or directory`"

The problem: Hardcoded paths

```
#!/bin/bash  # Assumes bash is at /bin/bash
```

The solution: Use `env` to find in PATH

```
#!/usr/bin/env bash
```

> Small change, huge portability improvement.

The Bulletproof Pipeline

Demo: run_analysis.sh - Your computational lab notebook

```bash
#!/usr/bin/env bash
set -euo pipefail  # Fail fast on errors

# Create timestamped results directory
RESULTS_DIR="results_$(date +%Y%m%d_%H%M%S)"
mkdir -p "$RESULTS_DIR"

# Log EVERYTHING
{
    echo "Git hash: $(git rev-parse HEAD)"
    pip list
} >> "$RESULTS_DIR/run.log"

# Run analysis, appending output to log
python3 mc_simulation.py 2>&1 | tee -a "$RESULTS_DIR/run.log"
```

When Reproducibility Fails: Debugging

Demo: setup.sh - Systematic environment checking

```
# Check Python version
PYTHON_VERSION=$(python3 -c 'import sys;
print(f"{sys.version_info.major}.{sys.version_info.minor}"
)')
if [ "$PYTHON_VERSION" != "3.10" ]; then
    echo "Warning: Expected Python 3.10, got $
PYTHON_VERSION"
fi

# Verify package versions match requirements
pip list --format=freeze > current.txt
diff requirements.txt current.txt || echo "Package
mismatch detected!"
```

> Automate your debugging - don't rely on memory.

Your Reproducible Workflow

Let's synthesize our debugging journey into a robust, reusable workflow.

> This is the structure that emerged from fixing our simulation.

The Structure That Emerged

What we built while fixing our simulation:

```
mc-project/
├── README.md              # Start here - always!
├── requirements.txt # For quick setup
├── environment.yml        # For numerical work
├── Dockerfile             # For publication
├── setup.sh               # Checks everything
├── run_analysis.sh        # Runs everything
└── mc_simulation.py       # Our actual work
```

> This structure captures our entire reproducible process.

Progressive Enhancement Strategy

You don't need everything at once. Start simple and add complexity as needed.

1. **Exploration:** `requirements.txt` (unpinned)
2. **Sharing with a colleague:** `requirements.txt` + `venv`
3. **Numerical work:** `environment.yml` (Conda)
4. **Publication/collaboration:** `Dockerfile`
5. **Repeated runs:** `run_analysis.sh`

> Match the tool to the project's current stage.

Testing Before Sharing

The "clean machine" test:

1. `git clone` your project into a new directory
2. Run `setup.sh` → Does the environment check pass?
3. Create a `venv` → Does it work cleanly?
4. Run `run_analysis.sh` → Does the full pipeline complete?
5. Check `results/` → Are the outputs correct?

> The golden rule: If you can't reproduce it, neither can they.

Version Control Integration

What to track with `git`:

- All environment files (`requirements.txt`, `environment.yml`, `Dockerfile`)
- All scripts (`setup.sh`, `run_analysis.sh`)
- Your source code (`mc_simulation.py`)

What NOT to track:

- The virtual environment itself (`sim_env/`)
- Python cache files (`__pycache__/`)
- Output directories (`results_*/`)

Example .gitignore

This file tells git what to ignore. We've added one to lec03/code/.gitignore:

```
# Python virtual environments created by demo scripts
demo_env/
venv_demo/

# Python cache files
__pycache__/
*.pyc

# Output directories and files from analysis scripts
results_*
walk_analysis.png
*.log
```

> Keep your repository clean and focused on source files.

Tagging for Publication

When you generate results for a paper, tag that exact version of the code.

```
# After generating final figures
git add -A
git commit -m "Generate final figures for Nature paper"
git tag -a v1.0-nature -m "Code for Nature submission, Jan 2025"
git push origin v1.0-nature
```

In your paper:

> "Code is available at github.com/user/repo, tag `v1.0-nature`."

From Chaos to Confidence

Where we started:

> "It works on someone's machine!"

Where we are now:

> "It works on ANY machine!"

The tools we mastered: venv/uv → conda/micromamba → Docker
The practices we adopted: Document → Isolate → Automate → Test

Resources

Course Materials

- All lecture code: `am215_lectures/lec03/code/`

Tools

- `uv` (*https://astral.sh/uv*): The fast Python package installer.
- `conda` (*https://conda.io/en/latest/*): Full-stack environment management.
- `micromamba` (*https://mamba.readthedocs.io/en/latest/user_guide/micromamba.html*): A fast `conda` alternative.
- `Docker` (*https://docs.docker.com/*): Containerization for portability.
- `Nix` (*https://nixos.org/*): The ultimate reproducible build system.

Further Reading

- **The Python Packaging Guide** (*https://packaging.python.org/*)
- **The Turing Way: Reproducibility** (*https://book.the-turing-way.org/reproducible-research/overview*)
- **NumPy's Random Number Generation** (*https://numpy.org/doc/stable/reference/random/index.html*)
- **Reinhart-Rogoff: The Excel Error** (*https://www.bbc.com/news/magazine-22223190*)
- **Floating Point Math** (*https://docs.python.org/3/tutorial/floatingpoint.html*)

> Start small, build up as needed.