

Coding environment & reproducibility

How to fail consistently



Coding environment

We are all products of our environment — *C.J. Heck*

The coding environment affects the result of running software

- Operating system
- Software version
- Presence of dependencies



Software version

- Software is released in different versions
- On the command line, you can usually use `--version` or `-V` to print the version info

```
1 bash --version
2 python --version
```

- **Semantic versioning** proposes rules how to assign and increment version numbers

```
1 MAJOR.MINOR.PATCH
```

Dependencies

- Software may depend on other software/libraries
- Efficient/architecture-specific implementations
- Reliability due to larger community
- Dependencies are not always pre-installed:

```
1 python3 -c "import numpy"
```



Hands-on session

- Find out which Python version is available on your machine
- Find out which Python version is available on levante
- Do they provide NumPy?



Environments

The context in which software is created or executed

- Local/development environment
- Testing environment
- Production environment

~~It works on my machine~~ 🙄

It works in **this** environment! 🚀

Package manager

Tools to install packages and their dependencies

- General purpose package manager (`brew`, `apt`, `micromamba`)

```
1 brew install coreutils
```

- Python package manager (`pip`, `micromamba`, `uv`, `pixi`)

```
1 python3 -m pip install numpy
```

Package manager

Tools to install packages and their dependencies

- General purpose package manager (`brew`, `apt`, `micromamba`)

```
1 brew install coreutils@9.7
```

- Python package manager (`pip`, `micromamba`, `uv`, `pixi`)

```
1 python3 -m pip install "numpy==2.0.2"
```



Virtual environments (`venv`)

- Contain all dependencies needed to support a project
- Isolated from other virtual environments (and the OS)
- Not checked into source control systems such as Git
- Considered as disposable
- Not considered as movable or copyable
- See [PEP 405](#) for more info on `python` virtual environments



Specifying a python environment

- `micromamba` is a widely used package manager for python
- Handles virtual environments, general purpose packages (and libraries), and python packages
- Supports `YAML` file to specify environments:

```
environment.yaml
```

```
1 name: my-environment
2 channels:
3   - conda-forge
4 dependencies:
5   - python=3.12
6   - numpy>=2
7   - matplotlib=3.10.*
8   - scipy=1.15.2
```



Hands-on session

- Install `micromamba` on your machine
- Create an `environment.yaml` with some dependencies
- Create the environment using

```
1 micromamba env create -f environment.yaml
```

- Activate the environment (`micromamba --help`)
- Run the following code again in the terminal:

```
1 python --version  
2 python -c "import numpy; print(numpy.__version__)"
```

Reproducibility



- Resolving a set of dependencies is deterministic, *in theory*
- Depends on creation time (new package versions)
- May depend on the machine
- For reproducibility, it is important to export the **resolved environment**, e.g., `micromamba env export`:

```
1 name: my-environment
2 channels:
3   - conda-forge
4 dependencies:
5   - brotli=1.1.0=hd74edd7_2
6   - brotli-bin=1.1.0=hd74edd7_2
7   - bzip2=1.0.8=h99b78c6_7
8   - ca-certificates=2025.1.31=hf0a4a13_0
9   - contourpy=1.3.1=py312hb23fbb9_0
10  - cycler=0.12.1=pyhd8ed1ab_1
11  - fonttools=4.57.0=py312h998013c_0
12  - freetype=2.13.3=h1d14073_0
13  - kiwisolver=1.4.8=py312h2c4a281_0
14  - lcms2=2.17=h7eeda09_0
15  - lerc=4.0.0=h9a09cb3_0
```



Hands-on session

- Run `micromamba env export`
- Compare your output with that of someone else in the class



Consistency

- Predictable structure makes the code easier to follow
- Reduces cognitive load
- Easier collaboration



Code formatting

```
1 import numpy as np
2 def hypot(a,b):
3     """Given the legs of a right triangle, return its hypotenuse."""
4     return np.sqrt(a ** 2+b**2)
5
6 def is_even(a):
7     '''Check if a given number is even.'''
8     answer=42
9     return a % 2==0
```



Code formatting

```
1 import numpy as np
2
3
4 def hypot(a, b):
5     """Given the legs of a right triangle, return its hypotenuse."""
6     return np.sqrt(a**2 + b**2)
7
8
9 def is_even(a):
10    """Check if a given number is even."""
11    answer = 42
12    return a % 2 == 0
```



Automated code formatting

Code formatters are used to apply style guidelines

- They follow standards (e.g. [PEP8](#)) but can be customized
- They format code in a **deterministic** way
- For Python: [ruff](#) or [black](#)

```
1 ruff format test.py
```



Static code analysis

Code linters are used to statically analyse code

- Find unused variables, style violations, and more
- For Python: [ruff](#) or [flake8](#)

```
1 ruff check test.py
```

```
1 F841 Local variable `answer` is assigned to but never used
2   --> test.py:11:5
3     |
4   9 | def is_even(a):
5  10 |     """Check if a given number is even."""
6  11 |     answer = 42
7     |     ^^^^^^^
8  12 |     return a % 2 == 0
9     |
10 help: Remove assignment to unused variable `answer`
```



Take home messages

- The results of software depend on the coding environment
- Virtual environments help to isolate dependencies
- To ensure reproducibility (for you and others), it's essential to share resolved environments



Shotgun buffet



Module system

- DKRZ uses **Environment Modules** to provide different versions of various packages

- List all available modules

```
1 module avail
```

- Activate a certain module (with implicit/or explicit version)

```
1 module load python3  
2 module load python3/unstable
```

- Unload all currently loaded modules (useful at the beginning of scripts)

```
1 module purge
```



Containerization

- Packaging together software code with all its necessary components

It's basically a fully functional and portable computing environment. — *RedHat*

- Most well-known are [docker](#) (de facto standard) and [Apptainer](#) (open-source alternative)
- Can be scaled and orchestrated on larger platforms ([Kubernetes](#))



pixi

- **Pixi** is a promising package management tool for python
- It builds on the the existing conda ecosystem, but **replaces** all it's tools
- Faster resolution and creation of environments
- Built-in reproducibility through lock files (**pixi.lock**)



uv

- `uv` is yet another promising package manager for python
- It builds on the the existing PyPI ecosystem
- Faster resolution and creation of environments
- Built-in reproducibility through lock files (`uv`)

