Lecture 02

A hitchhiker's guide to coding

High Performance Computing for Data Science - SISSA, 2023-2024

Pasquale Claudio Africa, Konstantin Karchev

23 Apr 2024

Outline

1. The role of Python in modern scientific computing
2. Dependency management
3. Docker
4. Continuous Integration/Continuous Deployment

The role of Python in modern scientific computing

The role of Python in modern scientific computing

Python has emerged as a pivotal language in scientific computing, distinguished by:

- Intuitive and readable syntax, making coding accessible to scientists from various fields.
- A vast array of libraries and tools tailored for scientific applications.

The power of Python in scientific computing is amplified by its extensive library ecosystem:

- NumPy and SciPy are fundamental for numerical computations.
- pandas enhances data manipulation and analysis capabilities.
- Matplotlib and Seaborn excel in creating scientific visualizations.
- TensorFlow and PyTorch are at the forefront of machine learning research and applications.

Python's role in democratizing scientific research is underscored by its open-source nature, fostering collaboration and innovation.

Some real-world applications of Python in scientific research

Python's impact in scientific research is evident through numerous real-world applications:

- In physics, it has been used to analyze data from the Large Hadron Collider.
- In biology, Python is integral in genome sequencing projects like the Human Genome Project.
- Environmental scientists utilize Python in modeling the effects of climate change on different ecosystems.
- In astronomy, it played a key role in processing the first image of a black hole.
- ...

How to get your system ready

Most Python libraries can be installed with `pip`, with `Conda`, with a package manager on Linux and macOS, or from source.

- Using `pip`:

```
pip install numpy scipy matplotlib seaborn pandas
```

- Using `Conda`:

```
conda create -n sci-env
conda activate sci-env
conda install numpy scipy matplotlib seaborn pandas
```

Best practices in setting up a scientific computing environment include creating isolated environments and maintaining updated library versions.

Modules and packages

Modules

Modules: reusable code in Python

In Python, the ability to reuse code is facilitated by modules. A module is a file with a `.py` extension that contains functions and variables. There are various methods to write modules, including using languages like C to create compiled modules.

When importing a module, to enhance import performance, Python creates byte-compiled files (`__pycache__/filename.pyc`). These files, platform-independent and located in the same directory as the corresponding `.py` files, speed up subsequent imports by storing preprocessed code.

Using Standard Library Modules

You can import modules in your program to leverage their functionality. For instance, consider the `sys` module in the Python standard library. Below is an example:

```
# Example: module_using_sys.py
import sys

print("Command line arguments:", sys.argv)
```

When executed, this program prints the command line arguments provided to it. The `sys.argv` variable holds these arguments as a list. For instance, running `python module_using_sys.py we are arguments` results in `sys.argv[0]` being `'module_using_sys.py'`, `sys.argv[1]` being `'we'`, `sys.argv[2]` being `'are'`, and `sys.argv[3]` being `'arguments'`.

The `from... import...` Statement

You can selectively import variables from a module using the `from... import...` statement. However, it's generally advised to use the `import` statement to avoid potential name clashes and enhance readability.

```
from math import sqrt
print("Square root of 16 is", sqrt(16))
```

A special case is `from math import *`, where all symbols exported by the `math` module are imported.

A module's `__name__`

Every module has a `__name__` attribute that indicates whether the module is being run standalone or imported. If `__name__` is `'__main__'`, the module is being run independently.

```
# Example: module_using_name.py
if __name__ == '__main__':
    print("This module is being run independently.")
```

Creating your own modules

Creating modules is straightforward: every Python program is a module!

Save it with a `.py` extension. For example:

```
# Example: mymodule.py
def say_hi():
    print("Hello, this is mymodule speaking.")

__version__ = '1.0'
```

Now, you can use this module in another program:

```
# Example: mymodule_demo.py
import mymodule

mymodule.say_hi()
print("Version:", mymodule.__version__)
```

The `dir` function

The built-in `dir()` function lists all symbols defined in an object. For a module, it includes functions, classes, and variables. It can also be used without arguments to list names in the current module.

```
# Example: Using the dir function.
import sys

# Names in sys module.
print("Attributes in sys module:", dir(sys))

# Names in the current module.
print("Attributes in current module:", dir())
```

Packages

Packages: organizing modules hierarchically

Packages are folders of modules with a special `__init__.py` file, indicating that the folder contains Python modules. They provide a hierarchical organization for modules.

```
<some folder in sys.path>/
└── datascience/
    ├── __init__.py
    ├── preprocessing/
    │   ├── __init__.py
    │   ├── cleaning.py
    │   └── scaling.py
    └── analysis/
        ├── __init__.py
        ├── statistics.py
        └── visualization.py
```

The `__init__.py` files (1/5)

The `__init__.py` file in a Python package serves multiple purposes. It's executed when the package or module is imported, and it can contain initialization code, set package-level variables, or define what should be accessible when the package is imported using `from package import *`.

Here are some common examples of using `__init__.py` files.

The `__init__.py` files (2/5)

1. Initialization code

```
# __init__.py in a package.

# Initialization code to be executed when the package is imported.
print("Initializing my_package...")

# Define package-level variables.
package_variable = 42

# Import specific modules when the package is imported.
from . import module1
from . import module2
```

In this example, the `__init__.py` file initializes the package, sets a package-level variable (`package_variable`), and imports specific modules from the package.

The `__init__.py` files (3/5)

2. Controlling `from package import *`

```
# __init__.py in a package.

# Define what should be accessible when a user writes 'from package import *'.
__all__ = ['module1', 'module2']

# Import modules within the package.
from . import module1
from . import module2
```

By specifying `__all__`, you explicitly control what is imported when using `from package import *`. It's considered good practice to avoid using `*` imports, but if you need to, this can help manage what gets imported.

The `.` symbol means that `module1.py` and `module2.py` are to be located in the same folder as the `__init.py__` file.

The `__init__.py` files (4/5)

3. Lazy loading

```
# __init__.py in a package.

# Initialization code.
print("Initializing my_lazy_package...")

# Import modules only when they are explicitly used.
def lazy_function():
    from . import lazy_module
    lazy_module.do_something()
```

In this example, the module is initialized only when the `lazy_function` is called. This can be useful for performance optimization, especially if some modules are rarely used.

The `__init__.py` files (5/5)

4. Setting package-level configuration

```
# __init__.py in a package.

# Configuration settings for the package.
config_setting1 = 'value1'
config_setting2 = 'value2'
```

You can use the `__init__.py` file to set package-level configuration settings that can be accessed by modules within the package.

Python modules as 1. scripts vs. 2. pre-compiled libraries

In Python, modules and packages can be implemented either as Python scripts or as pre-compiled dynamic libraries. Let's explore both concepts:

1. **Python modules as scripts:**
    - **Extension:** Modules implemented as scripts usually have a `.py` extension.
    - **Interpretation:** The Python interpreter reads and executes the script line by line.
    - **Readability:** Scripts are human-readable and editable using a text editor.
    - **Flexibility:** This is the most common form of Python modules. You can write and modify the code easily.
    - **Portability:** Python scripts can be easily shared and run on any system with a compatible Python interpreter.

2. Python modules as dynamic libraries

- **Compilation:** Modules can be pre-compiled into shared libraries for performance optimization.
- **Execution:** The compiled code is loaded into memory and executed by Python.
- **Protection of intellectual property:** Pre-compiled modules can be used to distribute proprietary code without exposing the source.
- **Performance:** Pre-compiled modules may offer better performance as they are already in machine code.

It's essential to note that Python itself is an interpreted language, and even when using pre-compiled modules, the Python interpreter is still involved in executing the code. The use of pre-compiled modules is more about optimizing performance and protecting source code than altering the fundamental nature of Python as an interpreted language.

You can use tools like Cython or PyInstaller to generate pre-compiled modules or standalone executables, respectively, depending on your specific use case and requirements.

The Python Standard Library

Python's standard library is very extensive, offering a wide range of facilities as indicated by the long table of contents listed on the website. The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs by abstracting away platform-specifics into platform-neutral APIs.

In addition to the standard library, there is an active collection of hundreds of thousands of components (from individual programs and modules to packages and entire application development frameworks), available from the Python Package Index.

Zen of Python:
"Explicit is better than implicit."
Run `import this` in Python to learn more.

Dependency management

Questions

1. **Code longevity:** How do you ensure that your code will still function as expected in one year, or even five? Consider the implications of using libraries such as Numpy, TensorFlow, or packages from sources like GitHub.
2. **Consistent results:** How can you guarantee that both your current and future collaborators are able to achieve the same computational results as you?
3. **Easy installation:** What steps can you take to simplify the process for collaborators to set up your code with all required dependencies?
4. **Reproducibility:** What measures can you implement to make it straightforward for colleagues to reproduce your results accurately?
5. **Managing multiple projects:** How can you effectively manage multiple projects that have differing and potentially conflicting dependencies?

Objectives

1. **Recording dependencies:** Master the process of documenting all dependencies required by your project.
2. **Communicating dependencies:** Develop the skill to clearly list and explain these dependencies in reports, theses, or publications.
3. **Using isolated environments:** Learn to utilize isolated environments to manage different projects without interference from conflicting dependencies.
4. **Simplifying script usage:** Enhance the ease of using and reusing scripts and projects to streamline workflows and improve efficiency.

PyPI (The Python Package Index)

- **Installation tool**: pip
- **Usage summary**: PyPI is primarily used for Python-only packages or Python interfaces to external libraries. It also hosts packages that include bundled external libraries, such as numpy.
- **Number of packages**: Extensive, with long-term support for older versions.
- **Handling libraries**: Dependencies on external libraries require either inclusion in the package or installation through other means (e.g., OS installers or manual setup).
- **Pros**:
  - User-friendly interface.
  - Simplified package creation process.
- **Cons**:
  - Installation of packages with external dependencies can be complex.

Conda

- **Installation tool**: Conda or its faster alternative, Mamba.
- **Usage summary**: Conda serves as a broader distribution tool, aimed at managing not just Python packages but also the libraries and tools they depend on. This is particularly valuable for scientific software that relies on external libraries for enhanced performance.
- **Number of packages**: A curated selection in the default channel, with a vast array in community-managed channels. Non-Conda packages can also be installed using pip.
- **Handling libraries**: External libraries are installed as distinct Conda packages.
- **Pros**:
  - Relatively easy to use.
  - Facilitates management of complex dependencies involving external libraries.
- **Cons**:
  - More complex package creation process.

Isolated environments

Isolated environments are crucial for managing software dependencies in project development, especially when working with complex or conflicting package requirements. Here's how they help:

- **Version specificity**: Install exact versions of packages required for your projects, ensuring consistency and compatibility across development stages.
- **Project isolation**: Each project can have its own separate environment. This prevents conflicts that arise when different projects need different versions of the same package.
- **Error recovery**: Mistakes during installation (such as installing incorrect packages) are easily rectifiable. Simply remove the compromised environment and set up a new one, starting afresh with correct specifications.
- **Result replication**: By exporting a list of packages from an environment, you can share a consistent setup with collaborators or replicate it in different setups. This facilitates consistency in results across different systems.

Isolated environments: examples

Conda

```
conda create --name python310-env python=3.10 numpy=1.24.3 matplotlib=3.7.2
conda activate python310-env
# Develop and run your Python code...
conda deactivate
```

Virtual environments

```
python3 -m venv scicomp
source scicomp/bin/activate
pip install numpy==1.24.3
pip install matplotlib==3.7.2
# Develop and run your Python code...
deactivate
```

Recording dependencies (1/2)

For effective management and replication of Python environments, recording dependencies is essential. There are two primary methods used to achieve this: `requirements.txt` for pip and `environment.yml` for Conda.

`requirements.txt`

This is a straightforward text file used by pip with virtual environments. It lists the packages your project depends on. Here's a basic example:

```
numpy
matplotlib
pandas
scipy
```

Recording dependencies (2/2)

`environment.yml`

Used by Conda, this YAML file provides a structured format to specify the name of the environment, the channels from which packages should be sourced, and the dependencies themselves. Example:

```yaml
name: my-environment
channels:
  - defaults
dependencies:
  - numpy
  - matplotlib
  - pandas
  - scipy
```

Pinning dependencies

- `requirements.txt`:

```
numpy==1.18.5
matplotlib==3.3.1
pandas==1.1.3
scipy==1.5.2
```

- `environment.yml`:

```yaml
name: my-environment
channels:
  - defaults
dependencies:
  - numpy=1.18.5
  - matplotlib=3.3.1
  - pandas=1.1.3
  - scipy=1.5.2
```

Additional resources

- Dependency management .
- Packaging .

Introduction to Docker

What is Docker?

Docker is a platform for developing, shipping, and running applications inside containers. Docker provides an isolated environment for your application and its dependencies, packaged into a Docker container. This container can run on any machine that has Docker installed, making it easy to ensure consistency across development, testing, and production environments.

Docker images vs. containers

- **Docker image**: A Docker image is a lightweight, standalone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and config files.
- **Docker container**: A container is a runtime instance of an image. When you run an image, you create a container from it. This is the process where the Docker engine takes the image, adds a writable layer on top, and initializes various settings (network ports, storage options, etc.).

Benefits of using Docker

- **Consistency across environments**: Docker containers ensure that your application behaves the same way in production as it does in development and testing.
- **Isolation**: Containers are isolated from each other and the host system, ensuring that processes do not interfere with each other.
- **Portability**: Containers can run on any desktop, traditional IT, or cloud infrastructure.
- **Microservices architecture**: Docker is ideal for microservices architecture, allowing each service to be contained in its own environment with its dependencies.
- **Scalability and efficiency**: Docker uses resources more efficiently, allowing you to quickly scale out your application as needed.

Using a Docker image (1/2)

```
# Pull the image.
docker pull python:latest

# Create a container.
docker run --name my_container -v /path/to/host/folder:/shared-folder -it -d python:latest

# Enable the container.
docker start my_container

# Use the container.
docker exec -it my_container /bin/bash
```

Using a Docker image (2/2)

You can leave the container and return to your OS with `exit`. You can check your containers and their status with the command

```
docker ps -a
```

If the status of the container is `Up`, you can stop it with

```
docker stop my_container
```

Once you have created your container remember to **do not** use again the commad `run` but just `start`. Otherwise you will create every time a new container. If you want to remove a container you can run:

```
docker rm <name-of-the-container>
```

Basic Docker commands

Here are some basic Docker commands to get you started:

- `docker run`: Run a container from an image.
- `docker build`: Build an image from a Dockerfile.
- `docker images`: List all locally stored Docker images.
- `docker ps`: List running containers.
- `docker pull`: Pull an image or a repository from a registry.
- `docker push`: Push an image or a repository to a registry.
- `docker rm`: Remove one or more containers.
- `docker rmi`: Remove one or more images.
- `docker logs`: Fetch the logs of a container.

Creating a Docker image/container

A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using `docker build` users can create an automated build that executes several command-line instructions in succession.

Structure of a Dockerfile

- **FROM**: Sets the base image for subsequent instructions. A valid Dockerfile must start with a FROM instruction.
- **RUN**: Executes commands in a new layer on top of the current image and commits the results.
- **COPY**: Copies new files or directories from source and adds them to the filesystem of the container at the destination.
- **CMD**: Provides defaults for executing a container. There can be only one CMD instruction in a Dockerfile.
- **ENV**: Sets the environment variable `<key>` to the value `<value>`.

Building a Docker image: example

```
# Start from a Python base image.
FROM python:3.8

# Install required libraries.
RUN pip install numpy pandas scikit-learn

# Add the project code to the container.
COPY . /app

# Set the working directory.
WORKDIR /app

# Specify the command to run the application.
CMD ["python", "./my_script.py"]
```

Then:

```
docker build -t my_image:version .
docker push my_image:version # If needed.
```

Best practices for efficient Docker images

- **Minimize the number of layers**: Combine similar commands into a single `RUN` statement.
- **Clean up after installs**: Remove unnecessary cache files.
- **Use `.dockerignore`**: Exclude files not relevant to the build (like data files).
- **Multi-stage builds**: Use multi-stage builds to keep the image size down by building in one stage and copying the necessary artifacts to another.

Continuous Integration/Continuous Deployment

What is CI/CD?

CI/CD stands for Continuous Integration and Continuous Deployment or Continuous Delivery. These concepts are fundamental to modern software development practices and are aimed at improving the quality of software and the speed of its delivery.

CI/CD is a method to frequently deliver apps to customers by introducing automation into the stages of app development.

Continuous Integration (CI)

Continuous Integration is the practice of merging all developers' working copies to a shared mainline several times a day. The main goal of CI is to provide quick feedback so that if a defect is introduced into the code base, it can be identified and corrected as soon as possible. CI helps in reducing the time and effort required for integrating changes made by different team members over time. In CI, automated tools are used to assert the new code’s correctness before integration. A source code version control system is the crux of the CI process. The version control system is also supplemented with other checks like **syntax style** review tools, other **code analyzers**, and **code tests**.

Key components of CI include:

- **Automated testing**: Running automated tests to ensure each integration meets the required standards.
- **Version control**: All code changes are integrated into a shared version control repository which helps in tracking changes and managing codebase.

Continuous Deployment (CD)

Continuous Deployment refers to the release into production of software that passes the automated tests. Essentially, every change that passes all stages of your production pipeline is released to your customers with no manual intervention, and only a failed test will prevent a new change to be deployed.

Key components of CD include:

- **Automated release process**: Enabling developers to deploy their changes to a production environment or to release to end-users at any time by clicking a button.
- **No human intervention**: Software changes are automatically deployed to production without explicit approval.
- **Instant rollback**: Capabilities to quickly revert to a previous version in case of a problem.

Benefits of CI/CD

1. **Faster release rate**: Frequent releases mean features reach customers quicker.
2. **Improved product quality**: Regular code testing and deployment reduce the risk of major issues in production.
3. **Efficient handling of issues**: Immediate feedback allows for quick fixes to bugs and issues.
4. **Reduced manual work**: Automation in testing and deployment reduces the workload on team members and reduces human errors.
5. **Enhanced team productivity**: Streamlined processes enable team members to focus on other productive activities.

CI/CD is a transformative practice that improves the speed and quality of software development and deployment. It emphasizes the importance of automation in building, testing, and deployment processes, which helps teams to release software changes more quickly and with confidence.

GitHub Actions

GitHub Actions is a CI/CD platform that allows automation of workflows based on GitHub repository events like push, pull requests, or issue creations. It's directly integrated into GitHub, making it an excellent tool for automating the testing and deployment of code hosted on GitHub.

Assumptions

- You have a Python application with a `requirements.txt` file.
- Your application incorporates a testsuite to make sure that new changes do not break existing functionalities.
- Your application's documentation is generated using Sphinx or another documentation tool.
- You have a GitHub repository set up for your project.

Workflow setup

1. **Choose a pre-existing Docker image:**
   For Python applications, the official Python Docker images can be used. These are available on Docker Hub and include various tags corresponding to Python versions. Of course, you can even use your custom built Docker image.

2. **Create a GitHub Actions workflow:**
   This example will use a GitHub Actions workflow to automate the testing with pytest and documentation generation with Sphinx.

3. **Workflow file:**
   Create a workflow file `.github/workflows/python-app.yml`, specifying the conditions under which to execute the action(s) and which command(s) define an action.

Workflow file (1/2)

```yaml
name: Python Application CI

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test-and-document:
    runs-on: ubuntu-latest

    container:
      image: python:3.8-slim
      options: --user 1001:1001

    steps:
    - name: Checkout repository
      uses: actions/checkout@v2

    - name: Set up Python environment
      run: |
        python -m venv venv
        . venv/bin/activate
```

Workflow file (2/2)

```
    - name: Install dependencies
      run: |
        pip install -r requirements.txt
        pip install pytest sphinx

    - name: Run tests
      run: |
        . venv/bin/activate
        pytest

    - name: Generate documentation
      run: |
        . venv/bin/activate
        cd docs
        sphinx-build -b html . _build/html

    - name: Upload documentation
      uses: actions/upload-artifact@v2
      with:
        name: documentation
        path: docs/_build/html/
```

Explanation of key components

1. **Container image**: The workflow runs in a container based on the `python:3.8-slim` image. This ensures that Python and any necessary system dependencies are already installed.
2. **Setup Python environment**: A virtual environment is set up within the container to isolate our project dependencies.
3. **Install dependencies**: Dependencies listed in `requirements.txt` are installed, along with `pytest` for testing and `sphinx` for documentation.
4. **Run tests**: Tests are executed using `pytest`. This step can be expanded with more specific commands depending on the structure of your test suite.
5. **Generate documentation**: Sphinx is used to generate HTML documentation from source files located in the `docs` directory. Adjust the `sphinx-build` command according to your Sphinx configuration.
6. **Upload documentation**: The generated HTML files are uploaded as an artifact on GitHub. These can be downloaded from the Actions tab after the workflow runs or published as static

Additional resources

- Testing and documenting Python code: see lecture notes.
- Productivity tools.
- More about unit testing in Python.