Modularity

Julián Jiménez Cárdenas

Scientific Computing II

Fundación Universitaria Konrad Lorenz

What is modularity?

Modularity refers to the practice of breaking down a complex system into smaller, manageable, and independent components or modules.

- Each module encapsulates a specific set of functionality, and these modules can be developed, tested, and maintained separately.
- Modularity is a fundamental principle in software design, engineering, and architecture that promotes code reusability, maintainability, and scalability.

Key aspects of modularity

- *Encapsulation*: Each module hides its internal details and exposes a well-defined interface for interacting with the outside world. This concept is also known as information hiding.
- *Abstraction*: Modules present a high-level view of their functionality to the rest of the system, allowing other parts of the system to use the module without needing to understand its internal complexities.
- *Independence*: Modules are designed to be independent of each other, reducing the impact of changes in one module on others. This isolation minimizes the risk of unintended side effects when modifying code.
- *Reusability*: Modular design encourages the development of reusable components. Modules that provide specific functionality can be reused in different projects or parts of the same project.

Key aspects of modularity

- *Maintainability*: Because modules are isolated, debugging, testing, and maintaining code becomes easier. Changes and updates can be confined to specific modules without affecting the entire system.
- *Scalability*: A modular system can be scaled by adding or modifying modules to accommodate new features or requirements.
- *Collaboration*: Modularity facilitates collaboration among developers working on different parts of the system. Teams can work on separate modules simultaneously, reducing development time.

Python example

Suppose you're building a simple program to perform mathematical operations. You can create a modular design by organizing your code into separate functions and modules for different mathematical operations.

```python
# math_operations.py

def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

```python
# main.py
import math_operations

num1 = 10
num2 = 5

# Use functions from the math_operations module
sum_result = math_operations.add(num1, num2)
difference_result = math_operations.subtract(num1, num2)

print(f"Sum: {sum_result}")
print(f"Difference: {difference_result}")
```

Both files should be in the same path.

Python Packages

- A package is a way to organize related modules into a directory hierarchy.
- Packages allow you to create a structured and organized environment for your code, making it easier to manage and distribute.
- Packages are particularly useful when you're working on larger projects that involve multiple modules and functionalities.
- Packages are created by organizing related modules within a directory hierarchy. The top-level directory becomes the package name, and subdirectories become sub-packages.

Package example

Consider the package named `mypackage` with the following directory structure:

```
mypackage/
├── __init__.py
├── math_operations.py
└── geometry/
    ├── __init__.py
    ├── area.py
    └── volume.py
```

Here is how you can use this package:

```
# Alternatively, use shorter names with 'as'
import mypackage.math_operations as math_ops
import mypackage.geometry.area as geometry_area

# Access functions from the modules
result = math_ops.add(5, 10)
area_result = geometry_area.calculate_area(10, 20)
```

Where should I put my package?

In principle, it should be at the same location of the main code. Nevertheless, you can change the `PYTHONPATH` to include the location of your package.

Changing it locally

```
import sys
print(sys.path) # Shows the actual path
sys.path.append('/path/to/your/package')  # Append the path of your package

import mypackage.module1
```

Where should I put my package?

Changing it globally

On a terminal prompt (or in your `.bashrc`) write

```
export PYTHONPATH="/path/to/your/package:$PYTHONPATH"
```

There is a third way of doing this, but we will cover it in detail in the following section.

Environments

- Python virtual environments are an important concept in Python programming that allow you to create isolated environments for your projects.
- These environments ensure that dependencies for one project do not interfere with other projects, and they help manage package versions effectively.
- To create a virtual environment, you can use the `venv` module or third-party tools like `virtualenv` or `conda`. We will use the first.

`venv` usage

1. Open a terminal and navigate to the directory where you want to create your project's environment. Then write:

```
# Replace 'myenv' with the desired name of your environment
python -m venv myenv
```

2. Activate the virtual environment using:

```
# On Windows
myenv\Scripts\activate
# On macOS and Linux:
source myenv/bin/activate
```

`venv` usage

3. Now you can install packages using `conda` or `pip`. Packages installed in the virtual environment are isolated from the global Python installation.

```
pip install package-name
conda install package-name
```

Sometimes `pip` or `anaconda` hang out. The solution is to use the `-v` (verbose) flag.

4. You can close the environment using

```
deactivate
```

Example

Let us install `numpy` in our new environment.

```
pip -v install numpy
```

This is the unique library that our environment has. To check this, write

```
pip freeze
```

and you will get the libraries (and their respective versions) installed in your env.

Requirements

Suppose that you need a specific version (let's say `1.0`) of a library called `library`. Then you should write

```
pip install library==1.0
```

For example, if I want `matplotlib` version `3.5`, I use

```
pip install matplotlib==3.5
```

Using `pip freeze` again will reflect this change.

Requirements

We can write a list of the current libraries installed in our environment writing

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

so you can install them in a new environment using

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

Enviroments and packages

When using an environment, the packages's path is local, and can be checked using

```
python -c "import sys; print(sys.path)"
```

The name of the directory containing the packages is `site-packages/`. Therefore, you should move any package of yours there in order to be detected by your environment.

Nevertheless, there is a cleaner way to deploy your packages in an environment using a `setup.py` file.

setup.py

```
from setuptools import setup, find_packages

setup(
    name="my_package",
    version="1.0.0",
    packages=find_packages(),
    description="A package for mathematical operations and geometry",
    author="Your Name",
    author_email="your.email@example.com",
    url="https://github.com/yourusername/my_package",
    classifiers=[
        "License :: OSI Approved :: MIT License",
        "Programming Language :: Python :: 3.9"
    ],
    python_requires=">=3.7",
    install_requires=['wheel', 'bar', 'greek']
)
```

`setup.py`

Putting the `setup.py` file in the main directory of your project suffices to install it using `pip`:

```
pip install -e my_package/
```

There are multiple advantages of this approach, namely

- **Dependency management:** `setup.py` allows you to specify and manage package dependencies.
- **Version control**.

`setup.py`

- **Clean installation and removal:** you can use `pip uninstall your_package`.
- **Automatic script creation:** You can use `entry_points` in `setup.py` to specify scripts that should be created and added to your system's PATH during installation.
- **Packaging for Distribution**.
- **Metadata**.
- **Consistent Development Workflow:** When working on a project, using `setup.py` to install and manage dependencies becomes part of the standard development workflow.