Modules, Packages, and uv

Lecture 05

Dr. Colin Rundel

Modules

What is a module?

In Python, a *module* is simply a file containing Python code (`.py` extension).

Modules provide:

- A way to organize related code into separate files
- A namespace to avoid naming conflicts
- Reusability of code across projects

Any Python file can be imported as a module,

Lec05_module.py

```python
"Sample module for Lec05"

def greet(name):
    return f"Hello, {name}!"

pi = 3.14159

print("Hello!")
```

Importing modules

The `import` statement loads a module (i.e. runs the code) and creates a namespace for its contents,

```
1  import Lec05_module
```

```
Hello!
```

Contents are then accessed via the namespace using dot notation,

```
1  Lec05_module.greet("Jane")
```

```
'Hello, Jane!'
```

```
1  Lec05_module.pi
```

```
3.14159
```

Import variations

There are several additional ways to import modules or their contents:

```
import Lec05_module                          # import the module
from Lec05_module import pi                  # import specific objects
from Lec05_module import greet as hello      # import with alias
import Lec05_module as module                # import module with alias
```

```
pi
```

```
3.14159
```

```
hello("John")
```

```
'Hello, John!'
```

```
module.pi
```

```
3.14159
```

The `from module import *` pattern imports all public names (does not start with `_`) but is considered bad practice as it can lead to namespace pollution and naming conflicts.

Module search path

When you import a module, Python searches for it in the following order:

1. The current working directory
2. Directories in the `PYTHONPATH` environment variable
3. Standard library directories
4. Site-packages (installed third-party packages)

You can inspect the search path via `sys.path`,

```
import sys
sys.path
```

```
[ /Users/rundel/.local/share/uv/python/cpython-3.14.2-macos-aarch64-none/bin,
  /Users/rundel/.local/share/uv/python/cpython-3.14.2-macos-aarch64-none/lib/python314.zip,
  /Users/rundel/.local/share/uv/python/cpython-3.14.2-macos-aarch64-none/lib/python3.14,
  /Users/rundel/.local/share/uv/python/cpython-3.14.2-macos-aarch64-none/lib/python3.14/lib-dynloa
  /Users/rundel/Desktop/Sta663-Sp26/website/.venv/lib/python3.14/site-packages,
  /Users/rundel/Library/R/arm64/4.5/library/reticulate/python,
  /Users/rundel/Desktop/Sta663-Sp26/website/.venv/lib/python314.zip,
  /Users/rundel/Desktop/Sta663-Sp26/website/.venv/lib/python3.14,
  /Users/rundel/Desktop/Sta663-Sp26/website/.venv/lib/python3.14/lib-dynload ]
```

Module attributes

Modules have several built-in attributes that provide useful metadata,

```
module.__name__
```

```
'Lec05_module'
```

```
module.__file__
```

```
'/Users/rundel/Desktop/Sta663-Sp26/website/stati
```

```
print(module.__doc__)
```

```
Sample module for Lec05
```

```
import json
json.__name__
```

```
'json'
```

```
json.__file__
```

```
'/Users/rundel/.local/share/uv/python/cpython-3.
```

```
print(json.__doc__[:200], "...")
```

```
JSON (JavaScript Object Notation) <https://json.
JavaScript syntax (ECMA-262 3rd edition) used as
interchange format.

:mod:`json` exposes an API familiar to user ...
```

`dir()` function

The `dir()` function lists all names defined in a module (or any object),

```
1 dir(module)
```

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__
```

```
1 dir(json)
```

```
['JSONDecodeError', 'JSONDecoder', 'JSONEncoder', '__all__', '__author__', '__builtins__', '__cach
```

__name__ and "__main__"

Every module has a __name__ attribute. When a module is run directly, __name__ is set to "__main__". When imported, it's set to the module's name.

Lec05_app.py

```python
print("Loading ...")
print(f"__name__ = {__name__}")
print("")

def main():
    print("Running main()")

if __name__ == "__main__":
    main()
```

```bash
uv run Lec05_app.py
```

```
Loading ...
__name__ = __main__

Running main()
```

```python
from Lec05_app import main
```

```
Loading ...
__name__ = Lec05_app
```

```python
main()
```

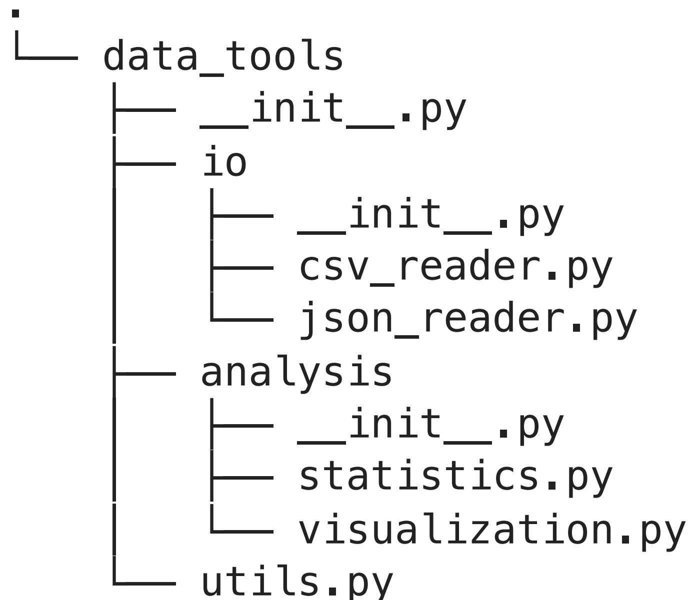
```
Running main()
```

Packages

What is a package?

A *package* is a way to organize related modules into a hierarchal structure using directories. A package is a directory containing:

- An `__init__.py` file (can be empty)
- One or more module files (`.py`)
- Optionally, sub-packages (subdirectories with their own `__init__.py`)



```
.
└── data_tools
    ├── __init__.py
    ├── io
    │   ├── __init__.py
    │   ├── csv_reader.py
    │   └── json_reader.py
    ├── analysis
    │   ├── __init__.py
    │   ├── statistics.py
    │   └── visualization.py
    └── utils.py
```

`__init__.py`

The `__init__.py` file:

- Marks a directory as a Python package
- Runs when the package is imported
- Can be empty or contain initialization code
- Can define `__all__` to control `from package import *`
- Often handles re-exporting from sub-modules to simplify import process

Example base `__init__.py` from our data_tools package,

```
1  from .utils import helper_func
2  from .io.csv_reader import read_csv
3
4  __all__ = ["helper_func", "read_csv"]
```

httpx - `__init__.py` example

```
from .__version__ import __version__

from ._api import delete, get, head, options, patch, post, put, request, stream
from ._auth import BasicAuth, DigestAuth, NetRCAuth
from ._client import AsyncClient, Client
from ._config import Limits, Proxy, Timeout, create_ssl_context
from ._exceptions import (ConnectError, ConnectTimeout, HTTPError, HTTPStatusError, ...)
from ._models import Cookies, Headers, Request, Response
from ._transports import AsyncHTTPTransport, HTTPTransport, MockTransport
from ._urls import URL, QueryParams

__all__ = [
    "__version__",
    "AsyncClient", "Client",
    "delete", "get", "head", "options", "patch", "post", "put", "request",
    "BasicAuth", "DigestAuth", "NetRCAuth",
    "Limits", "Proxy", "Timeout",
    "HTTPError", "ConnectError", "TimeoutException",
    "Request", "Response", "Headers", "Cookies", "URL",
    ...
]
```

Importing from packages

Packages allow for hierarchical imports,

```python
from collections import Counter
Counter("mississippi")
```

```
Counter({'i': 4, 's': 4, 'p': 2, 'm': 1})
```

```python
from urllib.parse import urlparse
urlparse("https://duke.edu/path?x=1")
```

```
ParseResult(scheme='https', netloc='duke.edu', path='/path', params='', query='x=1', fragment='')
```

```python
from pathlib import Path
Path(".").resolve()
```

```
PosixPath('/Users/rundel/Desktop/Sta663-Sp26/website/static/slides')
```

```python
from os.path import join, exists
join("static", "slides")
```

```
'static/slides'
```

```python
exists("Lec05.qmd")
```

```
True
```

The standard library

Python comes with a large collection of built-in packages called the *standard library* - these are available without installing anything extra.

Some useful examples:

- `json` - JSON encoding/decoding
- `pathlib` - filesystem paths
- `os` - operating system interface
- `sys` - system-specific parameters
- `re` - regular expressions
- `datetime` - dates and times
- `collections` - specialized containers
- `itertools` - iteration utilities
- `functools` - higher-order functions
- `math` - mathematical functions
- `random` - random number generation
- `typing` - type hints

Exercise 1

Given the following package structure (available in `exercises/`):

```
.
└── analytics
    ├── __init__.py
    ├── data
    │   ├── __init__.py
    │   ├── cleaner.py     # contains: remove_nulls(), normalize()
    │   └── loader.py      # contains: load_csv(), load_json()
    └── models
        ├── __init__.py
        └── regression.py  # contains: LinearModel class
```

Write import statements to:

1. Import `load_csv` from `loader.py`
2. Import both functions from `cleaner.py`
3. Import `LinearModel` with the alias `lm`
4. From within `regression.py`, import `remove_nulls` using a relative import

Package Management with uv

Python packaging landscape

Python's packaging ecosystem has historically been fragmented:

- Multiple tools: `pip`, `virtualenv`, `venv`, `conda`, `poetry`, `pipenv`, etc.
- Multiple config files: `requirements.txt`, `setup.py`, `pyproject.toml`, etc.
- Version management often handled separately (`pyenv`)

uv is a modern tool that aims to unify these concerns with a fast, Rust-based implementation.

What is uv?

uv is a Python package and project manager developed by Astral (creators of `ruff`).

Key features:

- Extremely fast (10-100x faster than pip)
- Manages Python versions
- Creates and manages virtual environments
- Installs packages
- Handles project dependencies via `pyproject.toml`
- Drop-in replacement for `pip` and `virtualenv`
- Directly supported by Positron and Reticulate

Installing uv

uv is already installed on the departmental servers, for local installs:

On MacOS/Linux:

```
1  curl -LsSf https://astral.sh/uv/install.sh | sh
```

or with homebrew:

```
1  brew install uv
```

or with pip / pipx

```
1  pipx install uv
2  pip install uv
```

Verify installation

Once installed you should be able to run the following,

```
uv --version
```

```
uv 0.9.26 (Homebrew 2026-01-15)
```

As long as you have version `0.9.*` you should be fine.

Managing Python versions

uv can install and manage multiple Python versions,

```
1  uv python list
```

```
cpython-3.15.0a5-macos-aarch64-none                    <download av
cpython-3.15.0a5+freethreaded-macos-aarch64-none       <download av
cpython-3.14.2-macos-aarch64-none                      /opt/homebre
cpython-3.14.2-macos-aarch64-none                      /opt/homebre
cpython-3.14.2-macos-aarch64-none                      /Users/runde
cpython-3.14.2-macos-aarch64-none                      /Users/runde
cpython-3.14.2+freethreaded-macos-aarch64-none         <download av
cpython-3.13.11-macos-aarch64-none                     /opt/homebre
cpython-3.13.11-macos-aarch64-none                     <download av
cpython-3.13.11+freethreaded-macos-aarch64-none        <download av
cpython-3.12.12-macos-aarch64-none                     /opt/homebre
cpython-3.12.12-macos-aarch64-none                     <download av
cpython-3.11.14-macos-aarch64-none                     <download av
cpython-3.10.19-macos-aarch64-none                     <download av
cpython-3.9.25-macos-aarch64-none                      <download av
cpython-3.9.6-macos-aarch64-none                       /usr/bin/pyt
cpython-3.8.20-macos-aarch64-none                      <download av
pypy-3.11.13-macos-aarch64-none                        <download av
pypy-3.10.16-macos-aarch64-none                        <download av
pypy-3.9.19-macos-aarch64-none                         <download av
pypy-3.8.16-macos-aarch64-none                         <download av
graalpy-3.12.0-macos-aarch64-none                      <download av
graalpy-3.11.0-macos-aarch64-none                      <download av
graalpy-3.10.0-macos-aarch64-none                      <download av
graalpy-3.8.5-macos-aarch64-none                       <download av
```

```
1  uv python install 3.14
```

```
Python 3.14 is already installed
```

```
1  uv python pin 3.14
```

```
Pinned `.python-version` to `3.14`
```

The pinned version is stored in `~/.python-version` and will be used automatically.

Initializing a project

Use `uv init` to create a new project,

```
mkdir my-project
cd my-project
uv init
```

```
Initialized project `my-project`
```

```
ls
```

```
total 32
drwxr-xr-x@   8 rundel  staff   256 Jan 21 13:22 .
drwxr-xr-x@ 135 rundel  staff  4320 Jan 21 13:23 ..
drwxr-xr-x@   9 rundel  staff   288 Jan 21 13:22 .git
-rw-r--r--@   1 rundel  staff   109 Jan 21 13:22 .gitignore
-rw-r--r--@   1 rundel  staff     5 Jan 21 13:22 .python-version
-rw-r--r--@   1 rundel  staff    82 Jan 21 13:22 main.py
-rw-r--r--@   1 rundel  staff   150 Jan 21 13:22 pyproject.toml
-rw-r--r--@   1 rundel  staff     0 Jan 21 13:22 README.md
```

This creates a `pyproject.toml`, a sample `main.py` script, and basic git infrastructure. Generally, we only really care about the `pyproject.toml` which we can exclusively generate via `uv init --bare`.

`pyproject.toml`

Modern project metadata file, tracks python version and package dependencies among other details.

```toml
[project]
name = "my-project"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.14"
dependencies = []
```

Adding dependencies

Once we have our project setup we can add (and install) dependencies directly via uv. `uv add` updates `pyproject.toml` and installs the package (creating a venv if needed).

```
uv add numpy
```

```
Using CPython 3.14.2
Creating virtual environment at: .venv
Resolved 2 packages in 157ms
Installed 1 package in 27ms
 + numpy==2.4.1
```

```
uv add pandas matplotlib scikit-learn
```

```
Resolved 18 packages in 490ms
Prepared 5 packages in 9.18s
Installed 15 packages in 134ms
 + contourpy==1.3.3
 + cycler==0.12.1
 + fonttools==4.61.1
 + joblib==1.5.3
 + kiwisolver==1.4.9
 + matplotlib==3.10.8
 + packaging==25.0
 + pandas==3.0.0
 + pillow==12.1.0
 + pyparsing==3.3.2
 + python-dateutil==2.9.0.post0
 + scikit-learn==1.8.0
 + scipy==1.17.0
 + six==1.17.0
 + threadpoolctl==3.6.0
```

```
uv add "pydantic<2"
```

```
Resolved 26 packages in 336ms
Prepared 1 package in 238ms
Installed 2 packages in 3ms
 + pydantic==1.10.26
 + typing-extensions==4.15.0
```

```
uv add --dev pytest ruff
```

```
Resolved 24 packages in 337ms
Prepared 4 packages in 859ms
Installed 5 packages in 27ms
 + iniconfig==2.3.0
 + pluggy==1.6.0
 + pygments==2.19.2
 + pytest==9.0.2
 + ruff==0.14.13
```

Updated `pyproject.toml`

```toml
[project]
name = "my-project"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.14"
dependencies = [
    "matplotlib>=3.10.8",
    "numpy>=2.4.1",
    "pandas>=3.0.0",
    "pydantic<2",
    "scikit-learn>=1.8.0",
]

[dependency-groups]
dev = [
    "pytest>=9.0.2",
    "ruff>=0.14.13",
]
```

Virtual environments

Virtual environments isolate project dependencies from the system Python and other projects. Packages are installed in a local folder in your project.

As we just saw, using `uv add` will create a new virtual environment in `.venv` by default if there is not an existing venv.

To explicitly create your own venv you can use,

```
uv venv                      # Create a virtual environment
uv venv --python 3.13  # Or specify Python version
uv venv .venv2               # Or specify directory name
```

Activating environments

To use the virtual environment certain environmental variables need to be set correctly (e.g. `PATH`, `PYTHONPATH`, etc.) so that the correct Python binary and libraries are used.

From the command line / terminal you can run the following in your project directory:

```
source .venv/bin/activate  # macOS/Linux (bash/zsh)
.venv\Scripts\Activate.ps1 # Windows (PowerShell)
.venv\Scripts\activate.bat # Windows (cmd)
```

Alternatively (strongly suggested), use `uv run` to execute commands in the environment without activating,

```
uv run python script.py
uv run pytest
uv run quarto render test.qmd
```

uv and Positron

Positron automatically detects virtual environments in your project directory. When you open a folder containing a `.venv` directory (created by uv), Positron will:

- Detect the environment and offer to use it
- Show the active Python interpreter in the status bar
- Use the environment for the Python console and when running scripts

If not automatically detected, you can manually select the interpreter via the Command Palette (`Cmd+Shift+P` / `Ctrl+Shift+P`) and searching for “Python: Select Interpreter”.

`uv sync`

Since the `.venv` folder is system specific (and large) it is not typically committed to git. Instead you will likely clone a repository that just has a `pyproject.toml` file.

Use `uv sync` to construct the venv and install all dependencies for the project

```
uv sync
```

```
Using CPython 3.14.2
Creating virtual environment at: .venv
Resolved 26 packages in 8ms
Installed 23 packages in 96ms
 + contourpy==1.3.3
 + cycler==0.12.1
 + fonttools==4.61.1
 + iniconfig==2.3.0
 + joblib==1.5.3
 + kiwisolver==1.4.9
 + matplotlib==3.10.8
 + numpy==2.4.1
 + packaging==25.0
 + pandas==3.0.0
 + pillow==12.1.0
 + pluggy==1.6.0
...
```

Listing packages

```
1  uv pip list
```

```
Package            Version
------------------ -----------
contourpy          1.3.3
cycler             0.12.1
fonttools          4.61.1
...
```

```
1  uv pip show numpy
```

```
Name: numpy
Version: 2.4.1
Location: /Users/rundel/my-project/.venv/lib/python3.14/site-packages
Requires:
Required-by: contourpy, matplotlib, pandas, scikit-learn, scipy
```

`uv pip install` and `uv pip freeze` are also available as lower-level commands for working outside of a project context or with `requirements.txt` files.

Dependency tree

```
1 uv tree
```

```
Resolved 26 packages in 8ms
my-project v0.1.0
├── matplotlib v3.10.8
│   ├── contourpy v1.3.3
│   │   └── numpy v2.4.1
│   ├── cycler v0.12.1
│   ├── fonttools v4.61.1
│   ├── kiwisolver v1.4.9
│   ├── numpy v2.4.1
│   ├── packaging v25.0
│   ├── pillow v12.1.0
│   ├── pyparsing v3.3.2
│   └── python-dateutil v2.9.0.post0
│       └── six v1.17.0
├── numpy v2.4.1
├── pandas v3.0.0
│   ├── numpy v2.4.1
│   └── python-dateutil v2.9.0.post0 (*)
├── pydantic v1.10.26
│   └── typing-extensions v4.15.0
├── scikit-learn v1.8.0
│   ├── joblib v1.5.3
│   ├── numpy v2.4.1
```

Common workflows

New project setup:

```
mkdir my-project
cd my-project
uv init --bare
uv add pandas numpy
uv run python project.py
```

Clone existing project:

```
git clone <repo-url>
cd <repo>
uv sync
uv run python script.py
```

Exercise 2

calculate_ci.py

```python
import numpy as np
from scipy import stats

data = [23, 25, 28, 30, 26, 27, 29, 24, 31, 28]

mean = np.mean(data)
sem = stats.sem(data)
ci = stats.t.interval(0.95, len(data)-1, loc=mean, scale=sem)

print(f"Sample mean: {mean:.2f}")
print(f"95% CI: ({ci[0]:.2f}, {ci[1]:.2f})")
```

The script above computes a confidence interval but requires `numpy` and `scipy`.

1. Create a new directory and copy the script into it
2. Initialize a uv project and the required dependencies
3. Check that you are able to successfully run the script and see the calculation results