STAT243 Lecture 5.3 Modules and Packages

Logic

- 流行的脚本语言通常都有大量在线可用的 add-on packages, Python 的流行很大程度上归功于 PyPI 和 Conda 上丰富的附加包集合, 这些包提供了 Python 的大部分功能
- 要使用一个 package, 需要先在系统上安装 (使用 `pip install` 或 `conda install`), 然后在每次启动新会话时加载 (使用 `import` 语句)
- 有些 modules 默认随 Python 安装 (如 `os` 和 `re`), 但需要用户在特定的 Python 会话中加载

1 Modules 模块

1.1 Module 是什么?

- module 是相关代码的集合, 存储在扩展名为 `.py` 的文件中
- 代码可以包含函数, 类, 变量以及可运行代码
- 要访问 module 中的对象, 需要 import module

从 Shell 创建示例模块:

Bash

```bash
cat << EOF > mymod.py
x = 7
range = 3
def myfun(x):
    print("The arg is: ", str(x), ".", sep = '')
EOF
```

使用模块:

Python

```python
import mymod
print(mymod.x)  # 7
mymod.myfun(99)  # The arg is: 99.
```

1.2 import 语句

- `import` 语句允许我们访问 module 中的代码
- 它将 module 中的 name of object 与 import scope 中的一个 name 关联起来
- name (references) 到 object 的 mapping 称为 **namespace 命名空间** (会在之后详细讲解)

Python

```python
del mymod
# Check if `mymod` is in scope.
try:
    mymod.x
except Exception as error:
    print(error)  # name 'mymod' is not defined
```

理解命名空间:

Python

```python
y = 3
import mymod

mymod  # 这是在 current (global) scope 中的模块对象
# <module 'mymod' from '/accounts/vis/paciorek/teaching/243fall25/fall-2025/units/mymod.py'>

x
# NameError: name 'x' is not defined (不在全局命名空间)

range  # 这是 buildin 函数, 不是来自模块
# <class 'range'>

mymod.x
# 7

mymod.range
# 3

dir(mymod)  # 查看模块中的所有对象
# ['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'myfun', 'range', 'x']
```

`y` 和 `mymod` 在 global namespace (全局命名空间) 中, 而 `range` 和 `x` 在 `mymod` 的 module namespace (模块命名空间中)

1.3 从 module 导入对象

可以使用 `from ... import ...` 语句使 module 中定义的对象直接在 current scope 中访问:

Python

```python
from mymod import x
x  # 现在是全局命名空间的一部分: 7

dir()  # 查看当前命名空间中的所有对象
# ['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__', 'content', 'files', 'io', 'it', 'm', 'math', 'mymod', 'os', 'pattern',
'platform', 'r', 're', 'return_group', 'st', 'stream', 'subprocess', 'sys', 'text', 'time',
'tmp', 'x', 'y']
```

此时**它与模块中的对象是不同的**:

Python

```python
mymod.x = 5
x = 3

mymod.x     # 5
x           # 3
```

Remark

通常不建议使用 `from` 以这种方式导入对象, 因为这可能引入名称冲突并降低模块性

1.4 使用别名简化模块名

可以使用 `import ... as ...` 来简化模块名

Python

```python
import mymod as m
m.x  # 5
```

2 Packages 包

2.1 `__init__.py` 文件

Packages 是一个包含一个或多个模块的 directory, 其中有一个名为 `__init__.py` 的文件, 该文件在导入包时被调用, 用于初始化包

创建基本包:

Bash

```bash
mkdir mypkg

cat << EOF > mypkg/__init__.py
# 使 mymod.py 中的对象可作为 mypkg.foo 使用
from .mymod import *
print("Welcome to my package.")
EOF

cat << EOF > mypkg/mymod.py
x = 7
def myfun(val):
    print(f"Converting {val} to integer: {int(val)}.")
EOF
```

调用包:

Python

```python
import mypkg  # Welcome to my package.
mypkg.x  # 7
mypkg.myfun(7.3)  # Converting 7.3 to integer: 7.
```

由于 `__init__.py` 中设置了 import 模块的方式, 我们在调用 package 之后无需知道特定的 module 的位置即可使用它们

> ⚠ **Remark**
>
> 可以在 `__init__.py` 中设置 `__all__` 来定义 `from ... import *` 所导入的内容

2.2 internal/private objects 内部/私有对象

可以添加用户不直接使用的 internal module (仅供 main module 使用), 按照约定**以下划线** `_` **开头**来表示私有/内部函数:

Bash

```bash
cat << EOF > mypkg/auxil.py
def _helper(val):
    return val + 10
EOF

cat << EOF >> mypkg/mymod.py
from .auxil import _helper
def myfun10(val):
    print(f"Converting {val} to integer plus 10: {int(_helper(val))}.")
EOF
```

2.3 子包

包可以在嵌套目录中包含模块, 通过这类**subpackages**实现额外的模块性:

> ⚠ **Remark: subpackage 的调用方式**
>
> subpackage 一般通过以下两种方式被调用:
>
> - 通过 package 的 main `__init__.py` 在用户使用 package 时自动调用, 即在 `mypkg/__init__.py` 中添加 `from . import mysubpkg`
> - 要求用户手动调用, 如 `import mypkg.mysubpkg`

Shell

```shell
mkdir mypkg/mysubpkg

cat << EOF > mypkg/mysubpkg/__init__.py
from .values import *
print("Welcome to my package's subpackage.")
EOF

cat << EOF > mypkg/mysubpkg/values.py
x = 999
b = 7
_my_internal_var = 9
EOF
```

使用子包:

Python

```python
import mypkg.mysubpkg  # __init__.py 被调用
# Welcome to my package's subpackage.

mypkg.mysubpkg.b
# 7

mypkg.x
# 7
```

> ⚠ **Remark**
>
> 一般来说，我们不会把 **子包（mysubpkg）** 里的东西，直接“搬运”到 **主包（mypkg）** 的命名空间里。也就是说，通常用户要用子包的内容时，需要写 `mypkg.mysubpkg.xxx`。
>
> 但是，有些情况下会有例外。举个例子：
>
> - 在 **NumPy** 里，函数 `linspace` 实际上是定义在文件 `numpy/core/function_base.py` 里的。
> - 可是用户在用的时候，并不需要写 `numpy.core.linspace`，只需要写 `numpy.linspace` 就行。
> - 这是因为 NumPy 在它的 **`__init__.py` 文件** 里，提前帮你把 `core` 子包里的 `linspace` 导入到了 `numpy` 的顶层命名空间。
>
> 对比一下：
>
> - 有些函数 NumPy 没有帮你“搬运”到顶层，比如线性代数的函数。
> - 它们必须通过子包来访问，例如 `numpy.linalg.<函数名>`，而不是直接 `numpy.<函数名>`。

3 Installing Packages 安装包

Logic

- 如果包在 PyPI 或通过 Conda 可用但不在系统上, 可以轻松安装
- 通常不需要机器上的 root 权限来安装包, 但可能需要使用 `pip install --user` 或设置新的 Conda 环境
- 包通常依赖于其他包, pip 或 conda 通常会自动安装依赖项
- Conda 的一个优点是它还可以安装 Python 包依赖的非 Python 包, 而使用 pip 时有时需要安装一个 system package 来满足依赖项

3.1 包管理工具

mamba:

- `mamba` 是 conda 的直接替代品, 通常在依赖项解析方面做得更好
- 在最近的 conda 版本中, 可以通过运行 `conda config --set solver libmamba` 在使用 conda 命令时使用 `mamba` 的依赖项解析器

conda-forge channel:

通常建议在使用 Conda 安装包时使用 **conda-forge** 通道, conda-forge 提供了由社区维护的各种最新包

3.2 使包可安装 (optional)

可以配置包以便通过 pip 构建和安装, 需要以下文件:

- `pyproject.toml`: 打包工具使用的配置文件
- `setup.py`: 使用 setuptools 时构建和安装包时运行
- `setup.cfg`: 使用 setuptools 时提供包的元数据
- `environment.yml`: 提供应使用包的完整环境信息
- `LICENSE`: 指定包的许可证

3.3 可重现性和包管理

Logic

为了可重现性, 了解使用的包版本很重要

pip 和 conda 使这变得容易, 可以创建一个 **requirements** 文件来捕获当前使用的包及其版本

使用 pip:

Shell

```shell
pip freeze > requirements.txt
pip install -r requirements.txt
```

使用 conda:

Shell

```shell
conda env export --no-builds > environment.yml
conda env create -f environment.yml
```

`--no-builds` flag 确保不包含系统特定的依赖项的信息 (例如那些只在 MacOS 上工作而在 Windows 上不工作的依赖项)

3.4 Conda 环境

Conda 环境提供了额外的模块化/可重现性层, 允许为计算设置完全可重现的环境:

Shell

```shell
conda create -n myenv python=3.13
source activate myenv
conda install numpy
```

Remark

如果我们使用 `conda activate` 而不是 `source activate` , Conda 会提示我们运行 `conda init` , 这将对我们的 `~/.bashrc` 进行更改，其中之一是在启动 shell 时自动激活 Conda 基础环境

这可能没什么问题，但了解这一点很有帮助。

3.5 Package location

Logic

Python 中的包可能安装在文件系统的各个位置, 有时找出包从文件系统的哪个位置加载是很有用的

使用 `__file__` 和 `__version__` 对象查看包在文件系统上的安装位置和版本:

Python

```python
import numpy as np
np.__file__  # '/system/linux/miniforge-3.13/lib/python3.13/site-packages/numpy/__init__.py'
np.__version__  # '2.1.3'
```

Remark

- `pip list` 或 `conda list` 也会显示所有包的版本号
- `sys.path` 显示 Python 在系统上查找包的位置

3.6 源码包与二进制包

源码包和二进制包的区别

- 源码包包含原始的 Python 代码 (有时也包含 C/C++ 和 Fortran 代码)
- 二进制包将所有非 Python 代码以二进制格式储存, C/C++ 和 Fortran 代码已经被编译

使用源码包和二进制包进行安装的区别

- 如果从源码安装包,
  - C/C++/Fortran 代码将在我们的系统上编译
  - 需要系统上有可用的编译器并且配置正确, 但编译后的代码一般能确保在我们的系统上运行
- 如果从二进制安装包,
  - 不需要在系统上进行编译
  - 某些情况下, 代码可能无法在我们的系统上运行, 因为它是以一种与系统不兼容的方式编译的

Remark

- Python `wheels` 是 Python 包的二进制包格式

- 某些包的 wheel 会因 operating system 的不同而不同, 以便包能在安装的系统上正确安装