程序设计与计算思维

第 23 讲：Python 独立程序编写、版本控制与 AI 编程

韩文弢

清华大学

2026 年 5 月

本讲内容

问题与目标

问题引入

在前面的课程中，所有编程工作都在 Jupyter Notebook 的浏览器界面内完成：写代码、运行、画图、调试，一站式搞定。但当你走出课堂，会遇到这些问题：

- 如何在没有浏览器的服务器上运行 Python 程序？
- 如何把一个 500 行的笔记本变成**可复用**的代码库？
- 如何让别人（或将来的自己）一键安装项目所需的全部依赖？
- 如何在不小心改坏代码后，**回退**到昨天还能运行的版本？
- 如何借助 AI 工具，让编程效率成倍提升？

本讲将带大家走出笔记本，掌握**真实开发环境**中的核心技能。

本讲目标

1. 了解终端的基本概念，掌握常用命令行操作
2. 了解如何将 Jupyter Notebook 中的代码迁移到独立 `.py` 文件
3. 掌握 Python 模块和包的基本概念与使用方法
4. 学会使用 pip、conda 和 uv 管理第三方包
5. 掌握 Git 与 GitHub 的基本工作流
6. 了解 AI 辅助编程的基本概念，学会使用 OpenCode 等工具提升开发效率

终端基础

什么是终端？

终端（Terminal）：通过文字命令与计算机交互的界面，也称**命令行（Command Line）**。

- 在 Jupyter Notebook 中，我们一直在浏览器里写代码
- 在实际开发中，许多操作需要在终端中完成
- macOS：Terminal.app，或 iTerm2, kitty, Ghostty, Alacritty 等第三方终端模拟器
- Windows：Windows Terminal 或 PowerShell
- JupyterHub 中也可以打开终端（Launcher → Terminal）

为什么需要终端?

以下场景需要终端:

- **运行**独立 `.py` 脚本
- **安装**第三方包（`pip install`）
- **管理**代码版本（`git commit`）
- **启动**开发工具（`opencode`）
- **部署**程序到服务器

终端是程序员的**基本功**，几乎所有开发工具都通过终端操作。

常用命令

命令	功能	示例
pwd	显示当前目录	pwd
ls	列出文件和目录	ls / ls -la
cd	切换目录	cd lectures
mkdir	创建目录	mkdir my_project
cp	复制文件	cp a.py b.py
mv	移动/重命名	mv old.py new.py
rm	删除文件	rm temp.py
cat	查看文件内容	cat hello.py

路径

路径（path）：定位文件或目录在文件系统中的位置。

- **绝对路径（absolute path）**：从根目录开始的完整路径

  ```
  /Users/hanwentao/lectures/lec23.typ
  ```

- **相对路径（relative path）**：相对于当前目录的路径

  ```
  lectures/lec23.typ
  ../templates/talk.typ
  ```

- “.”表示当前目录，“..”表示上一级目录
- “~”表示用户主目录（如 `/Users/hanwentao`）

终端中的快捷操作

提高终端效率的常用技巧：

1. **Tab 补全**：输入文件名或命令的前几个字母，按 `Tab` 自动补全
2. **上下箭头**：浏览历史命令，避免重复输入
3. `Ctrl + C`：中断当前正在运行的程序
4. `Ctrl + L`：清屏
5. 命令后面加 `--help`：查看帮助信息

在终端中运行 Python

- `python script.py`：运行 `.py` 文件
- `python`（不带参数）：进入交互式解释器（REPL）
- `python -c "..."`：执行一行 Python 代码

```
python hello.py
python -c "print(2 + 3)"
python
>>> import math
>>> math.sqrt(16)
4.0
>>> exit()  # 或 Ctrl+D 退出，Windows 可能是 Ctrl+Z 然后回车
```

注意：如果没有 `python` 命令，可以尝试使用 `python3` 命令。

从笔记本到独立程序

为什么需要独立程序?

在前面的课程中，所有代码都在 Jupyter Notebook 中运行。笔记本适合交互式探索和教学演示，但在实际开发中：

- 生产环境需要**独立运行**的程序
- 团队协作需要**可复用**的代码模块
- 自动化部署需要**命令行**工具
- 代码需要**版本管理**

创建 .py 文件

将笔记本中的代码保存为独立的 .py 文件：

```
print("Hello, World!")
```

在终端中运行：

```
python hello.py
```

从笔记本导出

Jupyter Notebook 可以导出为 `.py` 文件：

方法一： 菜单 File → Download as → Python (.py)

方法二： 命令行工具 `nbconvert`

```
jupyter nbconvert notebook.ipynb --to script
```

命令行参数

使用 sys.argv 接收命令行参数：

```
import sys

name = sys.argv[1]
count = int(sys.argv[2])

for _ in range(count):
    print(f"Hello, {name}!")
```

运行方式：

```
python greet.py Alice 3
```

独立运行模式

这是 Python 中非常重要的一个惯用写法：

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

if __name__ == "__main__":
    result = add(3, 5)
    print(f"3 + 5 = {result}")
    assert add(1, 2) == 3
    print("All tests passed!")
```

- 当文件被**直接运行**时，`__name__` 变量的值为 `"__main__"`
- 当文件被**导入**时，`__name__` 为模块名，不会执行测试代码
- 这样同一个文件既可以作为脚本运行，也可以作为模块被导入

模块与包

什么是模块?

模块（Module）：一个 `.py` 文件就是一个模块。

模块化编程的好处：

1. **代码复用**：避免重复编写相同功能
2. **命名空间**：通过 `module.function()` 避免命名冲突
3. **可维护性**：将代码按功能组织到不同文件中
4. **可测试性**：每个模块可以独立测试

导入模块的方式

- 推荐使用 `import module` 或 `from module import name` 的方式
- 避免 `from module import *`：会污染命名空间，且不清楚导入了哪些名称

```
import math  # 导入整个模块，使用时需要 `math.` 前缀
import math as m  # 导入模块并起别名，使用时需要 `m.` 前缀
from math import pi, sin  # 从模块中导入特定名称，直接使用 `pi` 和 `sin`
from math import factorial as fact  # 从模块中导入特定名称并起别名，使用时需要 `fact` 前缀
from math import *  # 导入模块中的所有名称，不推荐使用，可能会导致命名冲突
```

常用标准库模块

模块	功能	常用函数/类
`math`	数学函数	`sqrt`, `sin`, `cos`, `log`, `pi`
`os`	操作系统接口	`getcwd`, `listdir`, `path.join`
`sys`	系统相关	`argv`, `path`, `exit`
`json`	JSON 处理	`load`, `dump`, `loads`, `dumps`
`random`	随机数	`random`, `randint`, `choice`, `seed`
`datetime`	日期时间	`date`, `datetime`, `timedelta`
`pathlib`	路径操作	`Path`, `read_text`, `write_text`

什么是包?

包（Package）：包含 `__init__.py` 文件的目录。

```
mypackage/
├── __init__.py
├── math_tools.py
├── string_tools.py
└── data/
    ├── __init__.py
    └── loader.py
```

- `__init__.py` 可以为空文件，也可以包含包的初始化代码
- 包可以嵌套，形成层级结构

创建自己的包

以一个简单的数学工具包（mypackage/math_tools.py）为例：

```python
def fibonacci(n):
    fib = [0, 1]
    for i in range(2, n):
        fib.append(fib[-1] + fib[-2])
    return fib
```

使用时：

```python
from mypackage.math_tools import fibonacci
print(fibonacci(10))
```

包管理工具

pip：Python 包管理器

`pip` 是 Python 的标准包管理工具，从 **PyPI**（Python Package Index）安装包。

```
pip install numpy
pip install numpy==1.24.0
pip list
pip show numpy
pip uninstall numpy
pip freeze > requirements.txt
pip install -r requirements.txt
```

- `requirements.txt`：记录项目依赖，格式如 `numpy>=1.20\nmatplotlib>=3.4`
- 使用 `pip install -r requirements.txt` 一键安装所有依赖

conda：环境与包管理

conda 是一个更全面的工具，可以管理 Python 本身和 C/C++ 库：

```
conda create -n myproject python=3.12
conda activate myproject
conda install numpy matplotlib
conda install -c conda-forge fenics
conda env list
conda deactivate
conda env remove -n myproject
```

uv：新一代 Python 包管理器

uv 是用 Rust 编写的极速 Python 包管理器，集环境管理、依赖解析、脚本执行于一体：

```
uv init my_project
cd my_project
uv add numpy matplotlib
uv add --dev pytest
uv run script.py
uv remove numpy
```

- 使用 `pyproject.toml` 管理项目配置和依赖（现代 Python 标准）
- 内置虚拟环境管理，无需手动创建
- 速度比 pip 快 10–100 倍

pip、conda 与 uv 对比

特性	pip	conda	uv
实现语言	Python	Python	Rust
速度	基准	较慢	极快（10–100x）
语言范围	仅 Python	Python + C/C++	仅 Python
环境管理	需配合 venv	内置	内置
依赖锁定	无（需 pip-tools）	无	内置（uv.lock）
包来源	PyPI	Anaconda + conda-forge	PyPI
适用场景	通用	科学计算、跨语言	现代 Python 项目

虚拟环境

虚拟环境：为每个项目创建独立的 Python 环境，避免包版本冲突。

```
python -m venv myenv
source myenv/bin/activate
pip install numpy matplotlib
python my_script.py
deactivate
```

- 在 JupyterHub 中，通常已经配好了环境，不需要自己创建
- 在自己的电脑上开发时，建议每个项目使用独立虚拟环境

Git 与版本控制

什么是版本控制?

版本控制：记录文件内容的变化，以便将来查阅、回溯和协作。

你是否有过这样的经历?

论文_v1.doc
论文_v2.doc
论文_最终版.doc
论文_最终版2.doc
论文_最终版_老师修改.doc
论文_真的最终版.doc
论文_打死不改版.doc

版本控制系统可以优雅地解决这个问题。

为什么不用云盘?

需求	云盘	Git
同步粒度	每次按键 / 每次保存	手动提交（commit）
并行开发	文件副本冲突	分支（branch）机制
变更追踪	有限	完整历史记录
自动化	无	CI/CD 流水线
代码审查	无	Pull Request

- 代码对变化**敏感**：一个字符的改变可能影响整个程序
- 需要以**逻辑单位**（而非按键）同步变更

Git 核心概念

Git 工作流程：工作区 → 暂存区 → 本地仓库 → 远程仓库

```
远程仓库 (Remote)
    ↕  push / pull
本地仓库 (Repository)
    ↕  commit
暂存区 (Staging Area)
    ↕  git add
工作区 (Working Directory)
```

GitHub：代码托管平台

GitHub（github.com）是全球最大的代码托管平台，基于 Git 提供云端协作服务。

- **仓库（Repository）**：存放一个项目的所有文件和历史记录
- **Pull Request**：提交代码变更请求，支持在线审查和讨论
- **Issues**：追踪 bug、功能需求和讨论
- **Actions**：自动化测试、构建和部署（CI/CD）
- **开源社区**：数百万开源项目，如 Linux、TensorFlow、VS Code

我的 GitHub 账号：https://github.com/hanwentao

Git 基本操作

初始化与首次提交：

```
mkdir my_project
cd my_project
git init
echo "# My Project" > README.md
git add README.md
git commit -m "Initial commit"
```

Git 基本操作（续）

日常开发流程：

```
git status
git add hello.py
git add .
git commit -m "Add hello world script"
git log --oneline
git diff
```

分支操作

分支是 Git 最强大的特性之一：

```
git branch feature-login
git checkout feature-login
git checkout -b feature-signup
git branch
git checkout main
git merge feature-login
git branch -d feature-login
```

- main（或 master）：主分支，稳定版本
- 功能分支：开发新功能时创建，完成后合并回主分支
- 分支让多人并行开发成为可能

远程协作

```
git clone https://github.com/user/repo.git
git pull origin main
git push origin main
git remote -v
git fetch
```

- `clone`：首次获取远程仓库
- `pull`：获取并合并远程更新
- `push`：将本地提交推送到远程
- **Pull Request**：请求将你的变更合并到别人的项目

合并冲突

当两个人同时修改了同一文件的同一位置时，会产生**合并冲突**：

```
<<<<<<< HEAD
def greet(name):
    print(f"Hello, {name}!")
=======
def greet(name, greeting="Hi"):
    print(f"{greeting}, {name}!")
>>>>>>> feature-login
```

解决方法：

1. 手动编辑文件，选择要保留的内容
2. 删除冲突标记（`<<<<<<<`、`=======`、`>>>>>>>`）
3. `git add` + `git commit` 提交解决后的文件

.gitignore 文件

指定哪些文件不需要被 Git 追踪：

```
__pycache__/
*.pyc
.env
myenv/
.ipynb_checkpoints/
.DS_Store
data/large_files/
```

- 不要追踪：编译产物、敏感信息、虚拟环境、大型数据文件
- 应该追踪：源代码、配置文件、文档

AI 辅助编程

AI 如何改变编程?

大语言模型（LLM）正在深刻改变编程方式：

- **代码生成**：根据自然语言描述生成代码
- **代码解释**：阅读和理解不熟悉的代码
- **调试辅助**：定位错误并提供修复建议
- **重构建议**：改善代码结构和性能
- **文档生成**：自动生成注释和文档

AI 不是要取代程序员，而是让程序员专注于**更高层次**的思考和设计。

AI 编程工具概览

工具	类型	特点
GitHub Copilot	IDE 插件	实时补全，集成在 VS Code/JetBrains
Cursor	AI IDE	基于 VS Code 的 AI 原生编辑器
ChatGPT	Web 对话	通用对话，支持代码生成与解释
Claude	Web 对话	长上下文，适合大型项目分析
OpenCode	**终端工具**	**开源，多模型，多会话，LSP 支持**

OpenCode 简介

OpenCode 是一个开源的 AI 编程代理，直接在终端中运行。

安装方式：

```
curl -fsSL https://opencode.ai/install | bash
```

核心特点：

1. **开源**：代码托管在 GitHub，社区驱动（160K+ stars）
2. **多模型**：支持 Claude、GPT、Gemini 等 75+ 模型
3. **多会话**：可以同时运行多个 AI 代理处理不同任务
4. **LSP 集成**：自动加载语言服务器，理解代码结构
5. **隐私优先**：不存储代码或上下文数据

OpenCode 的使用方式

启动 OpenCode：

```
cd my_project
opencode
```

在终端中进入交互式对话界面，可以：

- 用自然语言描述需求，AI 自动编辑文件、运行命令
- 指定文件和代码范围，让 AI 在上下文中工作
- 让 AI 执行搜索、分析代码库、生成测试

AI 编程的工作流

传统编程 vs AI 辅助编程：

传统流程	AI 辅助流程
查阅文档	描述需求，AI 生成代码
手动编写代码	审查 AI 生成的代码
手动调试	描述问题，AI 定位错误
手动写测试	AI 自动生成测试用例
手动重构	描述目标，AI 执行重构

- AI 辅助并不等于“全盘接受”——**审查和理解** AI 生成的代码仍然至关重要
- 你需要**足够的基础知识**才能判断 AI 输出的正确性

提示词工程基础

与 AI 交流的质量决定了输出的质量：

好的提示词（prompt）：

```
我有一个 Python 列表，包含 1000 个浮点数。
请帮我写一个函数，计算它们的均值和标准差，
要求不使用 NumPy，只用标准库的 math 模块。
函数签名：def stats(data: list[float]) -> tuple[float, float]
```

差的提示词：

```
帮我算均值和标准差
```

提示词编写技巧

1. **明确上下文**：告诉 AI 你在使用什么语言、库、框架
2. **分步描述**：将复杂需求拆分为多个小步骤
3. **提供示例**：给出输入输出的期望样例
4. **指定约束**：说明性能要求、代码风格、限制条件
5. **迭代改进**：对不满意的输出追问和修正

记住：AI 是**工具**，最终决策权在你手中。

AI 编程的局限与伦理

局限性：

- 可能生成**看似正确但有 bug** 的代码
- 对最新版本的库可能**信息过时**
- 复杂逻辑需要**人工审查和测试**
- 不理解项目整体的**业务逻辑**

最佳实践：

1. 始终**审查** AI 生成的代码
2. 编写**测试**验证功能正确性
3. 使用 Git **版本控制**，随时可以回退
4. 保护**敏感信息**，不要将密钥等传给 AI

本讲小结

知识要点

1. **终端基础**：理解绝对路径与相对路径，熟悉命令行操作方式，掌握常用命令
2. **独立程序**：将笔记本代码迁移到 `.py` 文件，使用独立运行模式
3. **模块与包**：`.py` 文件即模块，目录加 `__init__.py` 即包，通过 `import` 使用
4. **包管理**：pip 通用安装，conda 适合科学计算，uv 是新一代极速工具
5. **版本控制**：Git 实现本地版本管理，GitHub 提供云端协作，分支支持并行开发
6. **AI 辅助编程**：以 OpenCode 为例，了解 AI 编程工具的使用方式和最佳实践

我的工作环境

- **操作系统**：MacOS（主要），Linux（服务器环境），Windows（迫不得已）
- **终端**：kitty（终端模拟器），tmux（终端复用器）
- **命令行**：fish（shell，命令行解释器） + starship（命令行提示符） + atuin（命令历史管理）
- **编辑器**：Zed（主要）, VS Code（次要）, Neovim（终端下）
- **AI 工具**：OpenCode（AI 编程代理）, Ollama（本地模型部署）