STAT243 Lecture 4.1 Good Coding Practices

1 Editors

- **选择合适的编辑器/IDE**
  使用支持目标语言的现代编辑器（VS Code、Emacs、Sublime、vim、RStudio 等）。
  优点包括：
  - 语法高亮与代码补全；
  - 自动缩进与括号匹配；
  - 行号显示（便于定位错误）；
  - 集成运行/调试功能；
  - 集成 AI 辅助编程（如 GitHub Copilot、Gemini Code Assist 等）。

> **Logic**
>
> 在 VS Code 或 RStudio 中使用 Quarto，可以直接运行、调试并格式化代码，适合课程作业与科研分析。

---

2 Code Syntax and Style

2.1 General Rules

- **参考标准**：遵循 Python 的官方风格指南 PEP 8。
- **文件头部信息**：用注释说明作者、日期、用途、Python 版本、主要依赖库。
- **Docstrings**：为公共模块、类、函数编写文档字符串；非公共项使用行内注释。
- **缩进**：每层 4 个空格（避免使用制表符）。
- **空白与可读性**：
  - 操作符两侧加空格；
  - 函数参数与元素之间加空格；
  - 使用空行分隔逻辑块；
  - 使用括号分组长表达式以便换行与注释。

> **Remark**
>
> 避免超过 **79 字符的代码行** 和 **72 字符的注释行**。
> 可用 `ruff format` 或 `black` 自动格式化。

- **命名约定（PEP8）**：
  - 类名 → `UpperCamelCase`
  - 函数、变量 → `snake_case`
  - 非公共方法 → `_leading_underscore`
  - 函数名使用动词：如 `calc_mean()`、`compute_error()`。
- **注释原则**：
  - 说明复杂逻辑或魔法常数；
  - 简洁明确，完整句式；
  - 例如：

Python

```python
newdf = (
```

```
    pd.read_csv('file.csv')
    .rename(columns={'STATE': 'us_state'})  # adjust column names
    .dropna()                               # remove NA rows
)
```

Logic

编写可读代码的核心：**清晰的结构 + 自文档化命名 + 简明注释**。

2.2 Coding Style Suggestions

- **模块化设计**：
  - 拆解为小函数，每个函数执行单一任务。
  - 避免重复代码，核心功能应封装复用。
  - 每个函数需：
    - 明确职责；
    - 有 docstring；
    - 具备单元测试。
- **可重用与透明**：
  - 函数应通过输入参数获取数据，而非操作全局变量。
  - 函数输入输出明确定义，方便复现分析。
- **可维护性与防御性编程**：
  - 检查输入有效性，提供默认参数；
  - 对潜在错误发出警告或抛出异常；
  - 使用断言（assert）检测假设。
- **其他建议**：
  - 不硬编码常数（建议使用如 `speed_of_light = 3e8`）。
  - 用 list/tuple 打包相关数据；
  - 避免平台相关代码；
  - 让他人审阅（code review）。

Logic

小函数、显式参数、单一职责 → 易调试、易测试、易复用。

2.3 Linting

- **Linting 定义**：自动检测代码风格与潜在错误的工具。
- **推荐工具**：`ruff`（轻量快速）或 `black`。

Shell

```
pip install ruff
ruff check test.py   # 检查语法
ruff format test.py  # 自动格式化
```

`ruff` 自动修复空格、缩进与注释风格。

Remark

Linting 不仅改善可读性，也能防止潜在逻辑错误。

---

3 Assertions, Exceptions, and Testing

3.1 Exceptions

- **异常（Exception）**：运行时错误的处理机制。
- 可通过 `raise` 主动抛出异常：

Python

```python
def myfun(val):
    if val <= 0:
        raise ValueError("`val` should be positive")
```

- **try-except 结构**：捕获并处理异常。

Python

```python
try:
    with open("file.txt", "r") as f:
        text = f.read()
except Exception as err:
    print(f"{err}\nCheck path: {os.getcwd()}")
    return None
```

- **re-raise 异常**：

Python

```python
try:
    requests.get(url)
except Exception as err:
    print("Problem accessing URL.")
    raise
```

Logic

异常处理逻辑应：
1️⃣ 捕获可预见错误；
2️⃣ 提供明确提示；
3️⃣ 在必要时重新抛出异常。

3.2 Assertions

- **断言（assert）** 用于验证程序状态是否符合预期：

Python

```python
number = -42
assert number > 0, f"Expected positive, got {number}"
```

若条件不满足，抛出 `AssertionError`。

- **常用断言形式**：

Python

```python
assert x in y
assert isinstance(x, float)
assert all(arr)
```

Logic

断言主要用于开发阶段调试和“sanity checks”；
在生产环境可禁用以提高性能。

3.3 Testing

- **单元测试（Unit Tests）**：验证单个函数的行为。
- 推荐使用 `pytest`：

Python

```python
import pytest
import dummy

def test_numeric():
    assert dummy.add_one(3) == 4

def test_bad_input():
    with pytest.raises(TypeError):
        dummy.add_one('hello')
```

运行测试：

Shell

```shell
pytest test_dummy.py
```

Logic

测试不仅验证正确输出，也应验证错误处理行为。

3.4 Automated Testing (CI)

- **持续集成（Continuous Integration, CI）**：
  自动在代码变更时运行测试。
- **GitHub Actions 示例**：
  `.github/workflows/test.yml`

YAML

```yaml
on:
  push:
    branches: [main]

jobs:
  CI:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
```

```
            python-version: 3.12
        - run: |
            pip install pytest
            pip install --user .
            pytest
```

> **Remark**
>
> CI 确保测试在远程环境中自动运行，
> 防止“本地能跑、服务器报错”的问题。

---

4 Version Control

- 使用版本控制（Git/GitHub）记录修改历史。
- 建议：
  - 使用 GitHub Issues 管理任务；
  - 维护清晰的 commit 信息；
  - 保留运行日志或开发笔记。

---

5 AI-Assisted Coding Tools

- **集成式 AI 编程助手**（如 Copilot、Gemini、Cursor）：
  - 代码自动补全与建议；
  - Chat/Agent 模式可自动修改代码；
  - 可基于上下文（文件/目录）生成代码。

> **Remark**
>
> - 对 AI 生成的代码保持警惕：需理解其逻辑。
> - 可用于简单语法或可验证任务（如绘图格式化）。
> - 对复杂算法仍需手动验证。

> **Logic**
>
> 适度使用 AI 辅助工具以加速开发，但**理解优先于生成**。