STAT243 Lecture 4 Good Practices

1 Good Coding Practices

1.1 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。

1.2 Code Syntax and Style

1.2.1 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**。

1.2.2 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。

1.2.3 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

```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。

1.3 Assertions, Exceptions, and Testing

1.3.1 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。

1.3.2 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。

1.3.3 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。

1.3.4 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
```

```yaml
      - 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。

1.4 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。

1.5 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**。

2 Debugging and Recommendations for Avoiding Bugs

> 🔥 **Logic: Overview**
>
> - **Common Debuggers in Python**
>   - `pdb`：Python 标准调试器（内置模块）。
>   - `ipdb`：IPython 封装的增强版本，支持自动补全与高亮。
>   - **JupyterLab**：自带交互式调试器。
>   - **VS Code**：最常用的图形化调试器，集成度高，推荐使用。

2.1 基本调试策略

- **阅读 traceback**
  - 从错误信息底部开始阅读；底部显示触发错误的实际行。
  - 用双引号包裹错误信息进行网页搜索（尤其在 Stack Overflow 上）常能快速找到解决方案。
- **理解调用栈（call stack）**
  - 错误通常发生在多层函数调用中。
  - 调试重点是：**出错时被执行的函数及其参数**。
  - 若错误出现在外部库函数中，应检查你自己代码传入的参数。
- **修错顺序**
  - 当有多个错误时，**从最早出现的错误开始修复**，后续错误往往由它引起。
- **检查可复现性**
  - 重新启动 Python 环境，看错误是否仍存在，以排查全局变量或作用域污染问题。
- **隔离错误源**
  - 逐步删除或添加代码（类似“二分查找”策略）以定位问题所在。
  - 模块化设计（函数化）可单独测试各部分逻辑。
- **传统方法**
  - 插入 `print()` 输出变量值以追踪执行流。
  - 尽管有更好工具，这一策略仍在简单情况下有效。
- **逐行执行**
  - Python 可逐行执行代码，适合快速排查简单问题。
  - 若错误涉及复杂嵌套函数或变量作用域，建议使用调试器。

2.1.1 Using `pdb`

- **激活调试器的方法**
  1. 插入 `breakpoint()` 或 `import pdb; pdb.set_trace()`。
  2. 运行后出错时使用 `pdb.pm()` 进入调试模式。
  3. 在 Jupyter 中使用 `%debug` 魔法命令。
  4. 用 `pdb.run("function_call")` 控制函数执行。
  5. 启动时直接进入调试模式：

  Shell
  ```
  python -m pdb file.py
  ```

- **示例：使用 `breakpoint()`**

  Python
  ```
  import run_with_break as run
  run.fit(run.data, run.n_cats)
  ```

  运行后进入交互模式 `(Pdb)`，可使用：
  - `n`：执行当前行并到下一行
  - `c`：继续到下一个断点
  - `p var`：打印变量

2.1.2 Post-mortem Debugging（事后调试）

- **方法**

  Python

```python
import pdb
import run_no_break as run
run.fit(run.data, run.n_cats)
pdb.pm()
```

在错误发生点自动进入调试器。

- **导航堆栈**
  - `u`：上移一层到用户代码。
  - `l`：查看周围代码。
  - `p var`：打印变量值。

2.1.3 常用 `pdb` 命令速查表

命令	含义
`h` / `help`	显示所有命令
`l` / `list`	查看当前行附近代码
`p` / `print`	打印变量
`n` / `next`	执行当前行
`s` / `step`	进入函数内部
`r` / `return`	跳出当前函数
`c` / `continue`	继续执行至下个断点
`b` / `break`	设置断点
`tbreak`	临时断点
`u` / `d`	在调用栈中上/下移动
`where`	查看调用栈
`q`	退出调试
`<Enter>`	重复上一命令

2.2 常见错误来源

- 括号不匹配（parenthesis mismatch）
- == 与 = 混淆
- 浮点比较（== 不可靠）
- 返回值类型或形状与预期不符
- 错用函数或变量名
- 无名参数顺序错误
- 变量未定义，Python 从外层作用域取到错误值（典型作用域错误）
- Python 自动降维导致维度不一致

2.3 防止与捕获错误的技巧

2.3.1 Defensive Programming（防御式编程）

- 检查函数输入的有效性。

- 提供合理默认值。
- 对输入/输出进行范围验证。
- 用 `assert`、`try` 或 `raise` 抛出带信息的错误。
- 示例：

Python

```python
import warnings

def mysqrt(x):
    if isinstance(x, str):
        raise TypeError(f"What is the square root of '{x}'?")
    if isinstance(x, (float, int)):
        if x < 0:
            warnings.warn("Input value is negative.", UserWarning)
            return float('nan')
        return x**0.5
    else:
        raise ValueError(f"Cannot take the square root of {x}")
```

2.3.2 try/except 捕获运行时错误

Python

```python
try:
    model = statsmodels.api.OLS(sub['y'], statsmodels.api.add_constant(sub['x']))
    fit = model.fit()
    params[i, :] = fit.params.values
except Exception as error:
    print(f"Regression cannot be fit for stratum {i}.")
```

- 若某分层无数据导致回归失败，程序仍能继续运行。

2.3.3 维度保持（Dimensionality Handling）

- Numpy 常会自动压缩多余维度，导致错误：

Python

```python
mat = np.array([[1, 2], [3, 4]])
mat2 = mat[1, :]       ## mat2 维度变为 (2,)
```

- 修复：

Python

```python
if len(mat2.shape) != 2:
    mat2 = mat2.reshape(1, -1)
```

2.3.4 避免使用全局变量

- 全局变量使代码不可预测，易受污染。
- 清理环境变量或在新 session 中运行函数以检测依赖。

Python

```
del x

def f(z):
    y = 3
    print(x + y + z)
## NameError: x is not defined
```

2.3.5 其他调试与编码建议

- 优先使用标准库或成熟算法包。
- 模块化设计：小函数易测试与复用。
- 先保证**正确性与可读性**，再考虑优化。
- **逐步构建代码**并测试中间结果。
- 为 `if`、`for` 等逻辑写健壮条件判断。
- 避免硬编码数值（如 `3e8` 应定义为 `speed_of_light`）。
- 提早编写测试（unit tests）。