

STAT243 Lecture 4.2 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。

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。

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

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

- **示例：使用 `breakpoint()`**



Python

```
1 import run_with_break as run
2 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

1.2 Post-mortem Debugging (事后调试)

- 方法

```
Python
1 import pdb
2 import run_no_break as run
3 run.fit(run.data, run.n_cats)
4 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。

1.3 常用 pdb 命令速查表

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

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

3 防止与捕获错误的技巧

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
1 import warnings
2
3 def mysqrt(x):
4     if isinstance(x, str):
5         raise TypeError(f"What is the square root of '{x}'?")
6     if isinstance(x, (float, int)):
7         if x < 0:
8             warnings.warn("Input value is negative.", UserWarning)
9             return float('nan')
10        return x**0.5
11    else:
12        raise ValueError(f"Cannot take the square root of {x}")
```

3.2 try/except 捕获运行时错误

```
Python
1 try:
2     model = statsmodels.api.OLS(sub['y'], statsmodels.api.add_constant(sub['x']))
3     fit = model.fit()
4     params[i, :] = fit.params.values
5 except Exception as error:
6     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。

3.3 维度保持 (Dimensionality Handling)

- Numpy 常会自动压缩多余维度, 导致错误:

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

- 修复：



Python

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

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

```
1 del x
2
3 def f(z):
4     y = 3
5     print(x + y + z)
6 # NameError: x is not defined
```

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）。