Python 技术管理规范

刘悦玲 2022141480082 软件工程

引言

本规范根据我在日常 Python 项目开发中遇到的问题与经验总结，同时参考了 Google、Meta 等大厂的技术管理规范，旨在为我今后进行 Python 项目开发提供一套统一的技术管理要求，以提高代码质量、可维护性、团队协作效率和项目整体的健壮性。规范分为三个级别：

- **A 级 (强制 Mandatory):** 必须严格遵守的规则，违反这些规则可能会导致代码审查不通过、构建失败或引发严重问题。
- **B 级 (推荐 Recommended):** 强烈建议遵循的最佳实践，有助于提升代码质量和开发效率。
- **C 级 (允许 Permitted):** 在特定情况下可以接受的做法，但需谨慎使用并确保不影响整体质量。

A. 强制 (Mandatory)

1. **A01. 代码格式 (PEP 8):** 代码必须严格遵守 PEP 8 -- Style Guide for Python Code 的核心规范。
2. **A02. 缩进:** 必须使用 4 个空格进行缩进，禁止使用制表符 (Tab)。
3. **A03. 行长度:** 每行代码长度不得超过 79 个字符，文档字符串和注释不得超过 72 个字符。
    - **说明:** 过长的代码行难以阅读，尤其是在分屏或小屏幕上。
    - **示例:**

      ```
      # 正确 (假设这里的注释或代码在 79/72 字符内)
      my_long_variable_name = another_variable + yet_another_variable - some_other_value
      # 这是一个符合长度限制的注释。


      # 错误 (过长)
      this_is_a_very_long_variable_name_that_exceeds_the_recommended_line_length_limit_for_python_code = "some_value"
      # 这个注释同样也太长了，应该换行或者缩短它以符合规范要求。
      ```

4. **A04. import 语句:** 每个 import 语句必须单独一行。
    - **正确示例:**

      ```
      import os
      import sys
      ```

- **错误示例:**
  import os, sys # 错误: 多个导入在同一行

- **原因:** 单独的导入语句更清晰，易于阅读和管理。

5. **A05. import 顺序:** import 语句必须按照以下顺序分组：标准库、第三方库、本地应用程序/库。组间用空行隔开。
   - **正确示例:**
     import os
     import sys

     import requests # 第三方库

     from my_project import my_module # 本地应用/库
     from . import utils

   - **错误示例:**
     import requests # 第三方库在标准库之前
     import os
     from my_project import my_module # 本地库与标准库混杂
     import sys

   - **原因:** 统一的导入顺序有助于快速理解模块的依赖关系。
6. **A06. 通配符导入:** 禁止使用通配符导入 (from module import *)，除非是内部 API 且明确标记了导出名称（如使用 __all__）。
   - **正确示例 (通常不推荐，除非特殊情况):**
     # module.py
     __all__ = ['public_function']
     def public_function():
       pass
     def _internal_function():
       pass

     # main.py
     from module import public_function # 明确导入

   - **错误示例:**
     from my_module import * # 错误: 使用了通配符导入

     result = some_function_from_my_module() # 不清楚
     some_function_from_my_module 来自哪里

- **原因:** 通配符导入会污染当前命名空间，使得代码难以理解和调试，不知道哪些名称被导入。

7. **A07. 文件编码:** 源文件必须使用 UTF-8 编码。
8. **A08. 错误处理:** 关键操作（如 I/O、外部服务调用）必须使用 try...except 块进行包裹，并捕获具体的异常类型，禁止裸露的 except:。
    - **正确示例:**

```
try:
    with open("config.json", "r") as f:
        config = json.load(f)
except FileNotFoundError:
    print("错误: 配置文件 'config.json' 未找到。")
    config = {}
except json.JSONDecodeError:
    print("错误: 配置文件 'config.json' 格式错误。")
    config = {}
except Exception as e: # 作为最后的保障，捕获其他未知异常
    print(f"发生未知错误: {e}")
    config = {}
```

    - **错误示例:**

```
try:
    # 一些操作
    result = 10 / 0
except: # 错误: 裸露的 except 会捕获所有异常，包括 SystemExit 和 KeyboardInterrupt
    print("发生了一个错误")
```

    - **原因:** 裸露的 except: 会捕获所有类型的异常，包括开发者可能不希望捕获的系统级异常（如 SystemExit, KeyboardInterrupt），并且使得错误定位更加困难。应捕获具体的、预期的异常。
9. **A09. 敏感信息管理:** 密码、API 密钥等敏感信息严禁硬编码在代码中。必须使用环境变量、配置文件或专用的密钥管理服务。
10. **A10. 虚拟环境:** 所有项目开发和部署必须使用虚拟环境 (如 venv, conda) 来隔离依赖。
11. **A11. 单元测试:** 核心业务逻辑和重要功能模块必须编写单元测试，并确保测试通过。
12. **A12. 主程序入口:** 作为可执行脚本运行时，必须使用 if __name__ == "__main__": 来保护主程序入口。
    - **正确示例:**

```
def main_logic():
    print("执行主要逻辑")
```

```
if __name__ == "__main__":
    main_logic()
```

- **错误示例 (直接执行):**
```
print("脚本开始执行...") # 当此文件被其他模块导入时，这行代码也会执行
# ...主要逻辑...
```

- **原因:** if __name__ == "__main__": 确保了当文件被作为模块导入时，其主程序逻辑不会执行，只有当文件作为脚本直接运行时才会执行。

13. **A13. 未使用代码:** 必须移除所有无效、冗余或不可达的代码 (Dead Code)。
14. **A14. 可变默认参数:** 函数定义中严禁使用可变类型 (如 list, dict) 作为默认参数。应使用 None 并配合逻辑判断进行初始化。
    - **正确示例:**
```
def add_item(item, L=None):
    if L is None:
        L = []
    L.append(item)
    return L
```

    - **错误示例:**
```
def add_item_buggy(item, L=[]): # 错误: L 在函数定义时只创建一次
    L.append(item)
    return L

list1 = add_item_buggy(1) # [1]
list2 = add_item_buggy(2) # [1, 2] (预期是 [2])
```

    - **原因:** 可变默认参数在函数定义时只被评估一次。后续调用若不提供该参数，会共享同一个对象，导致意外行为。
15. **A15. 显式相对导入:** 在包（package）内部，模块间的导入必须使用显式的相对导入 (如 from . import sibling 或 from .sibling import specific_thing)。
    - **假设包结构:**
```
my_package/
    __init__.py
    module_a.py
    sub_package/
        __init__.py
        module_b.py
```

- **正确示例 (在 my_package/module_a.py 中):**
  from .sub_package import module_b # 导入同级子包中的模块
  from . import utils # 假设 my_package/utils.py 存在

- **错误示例 (在 my_package/module_a.py 中):**
  import module_b # 错误: 隐式相对导入，可能导致混淆或在不同 Python 版本行为不一
  # 或者
  import my_package.sub_package.module_b # 绝对导入，虽然可行，但在包内部推荐相对导入

- **原因:** 显式相对导入更清晰地表明了导入的模块与当前模块在包结构中的关系，避免了与 Python 搜索路径中其他同名模块的冲突。

B. 推荐 (Recommended)

1. **B01. 命名 - 模块 (Module):** 模块名应简短、全小写，可使用下划线以提高可读性 (如 my_module.py)。
2. **B02. 命名 - 类 (Class):** 类名应使用驼峰命名法 (CapWords convention) (如 MyClass)。
3. **B03. 命名 - 函数/方法 (Function/Method):** 函数和方法名应全小写，单词间用下划线分隔 (snake_case) (如 def my_function():)。
4. **B04. 命名 - 变量 (Variable):** 变量名应遵循与函数名相同的蛇形命名法 (snake_case) (如 my_variable)。
5. **B05. 命名 - 常量 (Constant):** 常量名应全大写，单词间用下划线分隔 (如 MAX_CONNECTIONS = 10)。
6. **B06. 文档字符串 (Docstrings):** 所有公共模块、函数、类和方法都应编写符合 PEP 257 -- Docstring Conventions 规范的文档字符串。
   - **正确示例:**
     ```
     def calculate_area(radius):
       """计算圆的面积。

       参数:
         radius (float): 圆的半径。

       返回:
         float: 圆的面积。
       """
       import math
       return math.pi * radius ** 2
     ```

- **错误示例:**
  ```
  def calculate_area(radius): # 错误: 缺少文档字符串
    import math
    return math.pi * radius ** 2
  ```
- **原因:** 文档字符串是代码自文档化的重要组成部分，有助于他人（和未来的你）理解代码功能。

7. **B07. 注释 (Comments):** 对代码中非显而易见的部分（如复杂算法、特定业务逻辑）应添加注释进行解释。注释需保持与代码同步更新。
8. **B08. 类型提示 (Type Hinting):** 推荐在函数签名和关键变量上使用类型提示 (PEP 484)，尤其是在复杂或大型代码库中，以增强代码可读性和可维护性。
   - **正确示例:**
     ```
     def greet(name: str) -> str:
       return f"Hello, {name}"

     user_id: int = 123
     ```
   - **错误示例 (无类型提示):**
     ```
     def greet(name): # 错误: 缺少类型提示，降低可读性
       return f"Hello, {name}"
     ```
   - **原因:** 类型提示可以帮助静态分析工具检查类型错误，并提高代码的可理解性。
9. **B09. Linter 和 Formatter:** 推荐使用 Flake8, Pylint, Black, Ruff 等工具进行代码风格检查和自动格式化。
10. **B10. 列表推导式 (List Comprehensions):** 对于简单的列表创建，推荐优先使用列表推导式，而非 map() 和 filter() 函数。
    - **正确示例 (列表推导式):**
      ```
      numbers = [1, 2, 3, 4, 5]
      squares = [x**2 for x in numbers if x % 2 == 0] # [4, 16]
      ```
    - **可接受但通常不如推导式简洁 (map/filter):**
      ```
      numbers = [1, 2, 3, 4, 5]
      squares = list(map(lambda x: x**2, filter(lambda x: x % 2 == 0, numbers)))
      ```
    - **原因:** 列表推导式通常更简洁易读。
11. **B11. 生成器 (Generators):** 处理大数据集或流式数据时，推荐使用生成器以节省内存。
    - **正确示例 (生成器表达式):**
      ```
      large_range = (x * x for x in range(1000000)) # 不会立即创建百万个元素的列表
      for i in large_range:
        if i < 100: # 举例
      ```

```
        print(i)
    else:
        break
```

- **错误示例 (创建完整列表，可能消耗大量内存):**
  ```
  large_list = [x * x for x in range(1000000)] # 错误: 如果数据量极大，会占用大量内存
  for i in large_list:
      if i < 100:
          print(i)
      else:
          break
  ```

- **原因:** 生成器按需产生值，而不是一次性在内存中创建所有值，适用于处理大数据。

12. **B12. 上下文管理器 (Context Managers):** 对于需要管理资源（如文件、锁、数据库连接）的操作，推荐使用 with 语句。
    - **正确示例:**
      ```
      with open("my_file.txt", "w") as f:
          f.write("Hello, world!")
      # 文件会自动关闭，即使发生异常
      ```

    - **错误示例 (忘记关闭或异常处理不当):**
      ```
      f = open("my_file.txt", "w")
      try:
          f.write("Hello, world!")
      finally: # 错误: 如果 open() 本身失败，f 可能未定义，finally 中的 f.close() 会报错
               # 并且代码更冗长
          if f: # 需要检查 f 是否成功打开
              f.close()
      ```

    - **原因:** with 语句确保资源（如文件句柄）在使用完毕后被正确释放，即使在发生异常的情况下也是如此，代码更简洁安全。
13. **B13. 日志记录 (Logging):** 推荐使用 Python 内置的 logging 模块进行应用日志记录，而非 print() 语句。
    - **正确示例:**
      ```
      import logging
      logging.basicConfig(level=logging.INFO)
      logger = logging.getLogger(__name__)

      def my_function(data):
      ```

```
    logger.info(f"开始处理数据: {data}")
    try:
        # ... 处理逻辑 ...
        result = data / 0 # 示例错误
    except Exception as e:
        logger.error(f"处理数据时发生错误: {e}", exc_info=True) # exc_info=True 会记录堆栈信息

my_function(10)
```

- **错误示例:**
```
def my_function(data):
    print(f"开始处理数据: {data}") # 错误: 使用 print 进行日志记录
    # ...
```

- **原因:** logging 模块提供了更灵活和强大的日志管理功能，如日志级别、日志格式、输出到不同目标（文件、控制台、网络）等。

14. **B14. 代码审查 (Code Reviews):** 所有代码变更（尤其是新功能和重要修复）都应经过代码审查流程。
15. **B15. 依赖版本固定:** 推荐在 requirements.txt 或 pyproject.toml (配合 Poetry/PDM 等工具) 中固定依赖包的具体版本号，以保证构建的可复现性。
16. **B16. 测试覆盖率:** 推荐关键模块的单元测试覆盖率达到较高水平 (例如 >80%)。
17. **B17. f-string 格式化:** 在 Python 3.6+ 版本中，推荐使用 f-string (formatted string literals) 进行字符串格式化，因其简洁高效。
    - **正确示例 (f-string):**
```
name = "Alice"
age = 30
greeting = f"你好, 我是 {name}, 今年 {age} 岁。"
```

    - **可接受但较旧的方式 (.format()):**
```
greeting = "你好, 我是 {}, 今年 {} 岁。".format(name, age)
```

    - **不推荐的方式 (旧式 % 格式化):**
```
greeting = "你好, 我是 %s, 今年 %d 岁。" % (name, age)
```

    - **原因:** f-string 更易读，性能也通常更好。
18. **B18. 避免 is 比较字面量:** 比较字面量 (如数字、字符串) 时，应使用 == 或 !=，而非 is 或 is not。is 用于检查对象身份。
    - **正确示例:**
```
a = "hello"
```

```
b = "hello"
if a == b: # 正确
    print("字符串内容相同")

x = 10
if x == 10: # 正确
    print("数字值相同")
```

- **错误示例 (可能在某些 CPython 实现下偶然正确，但不保证):**

```
a = "a long string that is not interned" * 10
b = "a long string that is not interned" * 10
if a is b: # 错误: 比较的是对象身份，对于非 interned 的字符串可能为 False
    print("字符串对象相同 (不推荐的比较方式)")

val1 = 257
val2 = 257
if val1 is val2: # 错误: 对于超出小整数缓存范围的整数，is 可能为 False
    print("整数对象相同 (不推荐的比较方式)")
```

- **原因:** is 检查两个变量是否指向内存中的同一个对象，而 == 检查它们的值是否相等。对于字面量，尤其是字符串和较大的整数，Python 并不保证它们总是指向同一个对象（尽管 CPython 有小整数池和字符串驻留等优化）。

19. **B19. 显式 self:** 实例方法的第一个参数推荐总是命名为 self。
20. **B20. 显式 cls:** 类方法的第一个参数推荐总是命名为 cls。
21. **B21. 单一职责原则 (SRP):** 函数和类应尽可能遵循单一职责原则，即一个单元只做一件事。
22. **B22. DRY (Don't Repeat Yourself):** 避免代码重复。将通用逻辑抽象为函数或类。
23. **B23. 可读性优先:** 编写易于阅读和理解的代码。如果简洁性与清晰性冲突，优先选择清晰性。
24. **B24. 版本控制:** 所有项目代码推荐使用版本控制系统 (如 Git) 进行管理。
25. **B25. 异常链:** 在捕获并重新抛出异常时，推荐使用 raise NewException from original_exception 来保留原始异常的上下文。
    - **正确示例:**

```
class CustomError(Exception):
    pass

def process_data():
    try:
        # ... 某些可能引发 ValueError 的操作 ...
        data = int("abc")
    except ValueError as e:
```

```
        raise CustomError("处理数据时发生自定义错误") from e # 保留原始异常信息

try:
    process_data()
except CustomError as ce:
    print(f"捕获到自定义错误: {ce}")
    if ce.__cause__:
        print(f"  原始错误: {ce.__cause__}")
```

- **错误示例 (丢失原始异常上下文):**

```
class CustomError(Exception):
    pass

def process_data_bad():
    try:
        data = int("abc")
    except ValueError as e:
        raise CustomError("处理数据时发生自定义错误") # 错误: 丢失了原始 ValueError 的信息

try:
    process_data_bad()
except CustomError as ce:
    print(f"捕获到自定义错误: {ce}") # 无法直接看到原始的 ValueError
```

- **原因:** from e 将原始异常 e 设置为新异常的 __cause__ 属性，这在调试时非常有用，可以追溯到问题的根源。

C. 允许 (Permitted)

1. **C01. 空行:** 允许使用空行来分隔代码的逻辑区块，但应避免过度使用。
2. **C02. 行内注释:** 允许少量使用行内注释 (# comment) 来解释单行代码的特定部分。
3. **C03. 尾随逗号:** 在列表、元组、字典的最后一项后面允许使用尾随逗号，尤其是在多行定义时，有助于版本控制和减少合并冲突。
    - **正确示例:**

```
my_list = [
    "apple",
    "banana",
    "cherry", # 尾随逗号
]
my_dict = {
```

```
    "key1": "value1",
    "key2": "value2", # 尾随逗号
}
```

    - **说明:** 当添加新元素时，只需要新增一行，而不会修改前一行的内容，从而使版本控制的差异更清晰。
4. **C04. Lambda 函数:** 对于简单的、单表达式的匿名函数，允许使用 lambda 函数。
5. **C05. 单行多语句 (条件性):** 如果多个语句逻辑紧密相关且能清晰地放在一行（如简单的 if 条件赋值），允许使用，但通常为了可读性不推荐。例如：`if x > 0: y = 1`。
6. **C06. pass 语句:** 允许使用 pass 语句作为占位符，用于尚未实现的代码块或空的类/函数定义。
7. **C07. 自定义异常:** 允许定义继承自标准异常类的自定义异常，用于处理特定的错误场景。
8. **C08. 元类 (Metaclasses):** 在进行高级框架开发或需要深度定制类创建行为时，允许谨慎使用元类。普通应用开发中应避免。
9. **C09. assert 语句:** 允许在测试代码或开发阶段使用 assert 语句进行条件断言，但不应用于生产环境的逻辑校验或错误处理。

结语

在编写这份技术管理规范的过程中，我时常回想起自己以往在项目中因缺乏统一准则而走过的弯路：从命名不一致导致的查找困难，到团队协作时因代码风格分歧而产生的额外沟通成本，这些都让我深刻体会到规范的重要性。之前一直没时间系统梳理个人代码规范，这次刚好借助《软件过程与管理》课程作业的机会，我归纳与总结了日常开发中需要注意的问题形成次文档。在这一过程中，我不仅加深了对 Python 开发实践的理解，也重新审视了自己过去的编程习惯。我相信有了这些清晰的指导能够帮助我在之后新项目开发中提高代码质量，和团队成员快速达成共识，减少因风格不统一带来的摩擦。

本规范旨在提供指导，而非束缚。在实际项目中，可以根据具体情况进行适当调整和补充。持续学习和改进是提升技术水平的关键，希望这份文档能为我之后的 Python 开发之路带来更多积极影响。

参考文献

- **PEP 8 -- Style Guide for Python Code:** https://www.python.org/dev/peps/pep-0008/
- **PEP 257 -- Docstring Conventions:** https://www.python.org/dev/peps/pep-0257/
- **PEP 484 -- Type Hints:** https://www.python.org/dev/peps/pep-0484/
- **Google Python Style Guide:** https://google.github.io/styleguide/pyguide.html
- **The Hitchhiker's Guide to Python - Style Guide:** https://docs.python-guide.org/writing/style/