Python 编码规范手册

Python Coding Style Guide

姓名：郭政

学号：2023141461076

目 录

引言

本规范旨在为使用 Python 语言进行软件开发的项目提供统一、专业的技术指导和管理标准。通过对 PEP8、Google Python Style Guide 及业界最佳实践的参考整理，指导团队统一编程风格、提高代码质量、简化维护难度，支撑长期可持续开发。

规范等级说明

a. 强制 (Mandatory)： 必须遵守的规则，违反将阻碍代码提交或合并。
b. 推荐 (Recommended)： 建议遵循的规则，提高一致性和可维护性。
c. 允许 (Allowed)： 可选规则，在特定上下文下可使用，需团队共识。

一、 模块与导入（Imports）

1.1 强制要求

1.1.1 独占一行

每条 import 语句必须独立成行，不可在一行中导入多个模块。这种书写方式更易读、更便于版本控制和后期维护。

```
1. import os
2. import sys
3. # import os, sys #非法
```

1.1.2 导入顺序统一

模块导入顺序必须严格遵循三层结构：

Python 标准库（如 os, datetime）

第三方库（如 requests, numpy）

本地应用或自定义模块（如 my_project.utils）

每一层之间用空行分隔，增强视觉清晰度帮助开发者快速辨识模块来源。

1.1.3 禁止使用 import *

使用通配符导入会引入不明确的符号，污染当前命名空间，可能导致名称冲突，尤其在大型项目中极易出错。必须显式列出需要导入的成员。

1.1.4 禁止相对导入

不允许使用相对路径（如 from .. import utils），必须使用项目根目录起始的绝对导入路径。这样在 IDE、静态分析工具和部署环境下都更稳定、清晰。

1.1.5 合理选择导入方式

使用 import module 适用于整体模块导入，保留模块前缀有助于命名空间清晰。

使用 from module import name 可直接使用指定函数、类或变量，适用于简化调用。

需根据实际场景选择，不应滥用简化方式污染命名空间。

1.2 推荐实践

1.2.1 合并类型注解导入

使用 typing 或 collections.abc 等类型支持库时，应将多个类型合并导入，避免冗长重复的多行导入。

示例：from typing import List, Dict, Optional

1.2.2 使用常见别名统一缩写

当引入常见大型库（如 numpy, pandas, matplotlib）时，应使用行业通用的缩写别名，有助于团队协作和代码风格统一。

示例：import numpy as np, import pandas as pd

1.2.3 避免重复导入同一模块

同一个模块只应在文件中导入一次。如果需要在多个函数或类中使用，应放在文件顶部统一导入，避免重复。

1.3 可接受情况

1.3.1 教学或脚本可简化导入风格

在教学示例、一次性脚本或调试脚本中，可以根据场景适当简化导入书写方式。但应保持逻辑清晰，避免影响理解和后续使用。

示例：在 Jupyter Notebook 中可直接使用 from math import * 进行快速实验，但不推荐在生产代码中使用。

二、 命名规范（Naming Conventions）

2.1 强制要求

2.1.1 包与模块命名采用小写加下划线（lower_with_underscores）

所有包（目录）与模块（.py 文件）应使用全小写字母并用下划线分隔，避免使用驼峰或混合命名。这种命名风格有助于统一项目结构，减少在不同系统（如大小写敏感的 Linux）下出现导入错误的风险。

示例：data_loader.py、utils/math_tools.py

2.1.2 文件命名必须使用 .py 后缀，且不能包含连字符 -

Python 模块应以 .py 结尾以确保能够被解释器识别。禁止使用 - 是因为它在 Python 中不是合法的模块名字符，会导致导入失败。

合法：my_script.py；非法：my-script.py

2.1.3 类名使用大驼峰（CapWords）命名格式

所有类必须以每个单词首字母大写的方式命名（又称 PascalCase），以便于与函数、变量区分，同时提升代码的清晰度和一致性。

示例：UserProfile、DataManager

2.1.4 函数与变量名使用小写加下划线（lower_with_underscores）

函数、局部变量和全局变量必须采用这种命名方式，这种风格强调清晰表达，尤其在函数较长或参数多时更加一目了然。

示例：load_data()、user_count

2.1.5 常量使用全大写加下划线（UPPER_CASE_WITH_UNDERSCORES）

常量应以大写字母命名，单词间用下划线分隔。这种写法可以直观地区分常量与普通变量，提醒开发者它的值不应更改。

示例：MAX_RETRY_COUNT、DEFAULT_TIMEOUT

2.1.6 私有成员使用 _ 前缀标识

类或模块中的私有变量和方法应以前缀 _ 开头，表示“内部使用”。这并不会真正限制访问，但能传达明确的开发者意图。

示例：_load_config()、_cache_data

2.2 推荐实践

2.2.1 变量和参数名称应具有描述性

命名应清晰反映其用途或含义。避免使用 val、tmp、data1 等通用或含糊的命名，否则难以理解变量在业务中的作用。

推荐：user_list, retry_interval； 避免：val, tmp

2.2.2 自定义异常类应以 Error 结尾

所有自定义异常必须从 Exception 继承，并命名为 SomethingError，便于在捕获异常时明确识别，提升异常处理的规范性。

示例：FileFormatError, LoginTimeoutError

2.2.3 使用 __all__ 显式声明模块导出接口

在模块中使用 __all__ 可以明确指定哪些函数、类或变量是可被外部导入

的。这能限制 API 的外泄，增强模块的封装性。

示例：

__all__ = ['DataLoader', 'parse_config']

2.3 可接受情况

2.3.1 模块私有变量可以使用封装方式提供访问

对于模块内部的私有变量（如 _secret_key），可以使用函数或属性对其进行封装和访问，这样既能隐藏实现细节，又不影响可用性。

```
1. _secret_value = "xxx"
2. def get_secret():
3.     return _secret_value
```

三、 异常处理（Exceptions）

3.1 强制要求

3.1.1 禁止使用裸 except:

不允许使用不带异常类型的 except:，否则会捕获所有异常（包括系统异常如 KeyboardInterrupt），导致问题隐藏、难以调试。

3.1.2 不得无故捕获 Exception

除非用于最外层“兜底”逻辑，否则不应使用 except Exception: 捕获所有异常。应根据实际情况捕获特定异常类型，如 ValueError, IOError 等。

3.1.3 禁止用 assert 校验用户输入

assert 语句在生产环境中可能被禁用（如通过 python -O），不可靠。应使用 if ...: raise 明确检查用户输入并抛出合适异常。

3.1.4 自定义异常必须继承自 Exception，并以 Error 结尾

保证自定义异常与 Python 异常体系兼容，同时命名清晰易识别。

3.1.5 异常链要保留原始上下文

在重新抛出异常时应使用 raise NewError(...) from original_error 语法，保留原始异常栈信息，便于问题定位。

3.1.6 异常处理代码必须避免隐藏业务错误

捕获异常后应明确处理：记录日志、转换异常、终止流程等，禁止空 except 或简单 pass 掩盖关键问题。

3.2 推荐实践

3.2.1 使用 try/except/finally 正确释放资源

当需要清理资源（如关闭文件、释放锁）时，务必使用 finally 块确保无论是否抛出异常都能安全执行清理操作。

3.2.2 try 块应尽量小

将 try 块范围限定为“可能抛出异常的语句”，避免包裹无关代码，减少误捕异常的风险。

3.2.3 函数中抛出的异常应在 docstring 中注明

通过 Raises: 段说明可能抛出的异常类型，帮助调用者更好地使用和处理该函数。

3.2.4 日志中应包含异常详细信息

捕获异常后应使用 logger.exception() 或 logger.error(..., exc_info=True) 记录完整堆栈，便于排查问题。

3.2.5 优先使用内置异常

能使用标准库中已有的异常类型时，应优先使用。如遇非法参数使用 ValueError，文件找不到用 FileNotFoundError，避免自定义冗余异常。

3.2.6 对用户或外部系统输入要有防御式异常捕捉

网络请求、配置文件、命令行输入等外部数据应有健壮的异常处理逻辑，防止程序中断。

3.2.7 异常处理逻辑应保持简洁

在 except 块中不要嵌套过多业务逻辑，应集中处理异常本身，保持职责单一。

3.3 可接受情况

3.3.1 可静默忽略非关键异常，但需记录日志

某些异常不影响主要业务流程（如日志写入失败、缓存失效等）时，可忽略，但应记录日志用于事后追踪。

3.3.2 可使用 try/except 作条件判断替代 if，但需明确意图

在某些性能敏感或语义更清晰的场景（如操作不存在的字典键），可使用 try 代替 if 判断，但必须确保异常频率很低。

```
1. try:
2.     return cache[key]
3. except KeyError:
4.     return default
```

3.3.3 程序主入口可使用统一的异常捕获

顶层入口（如 main()）可用 try: except Exception: 捕获所有未处理的异常，并统一处理，例如记录日志或友好地退出程序。

四、 注释与文档（Comments & Docstrings）

4.1 强制要求

4.1.1 所有公共类、函数、方法必须编写 docstring

所有供外部调用的类、函数、方法必须使用三重双引号 """ 编写文档字符串，用于说明用途、参数、返回值和异常。

示例：

```
1. def fetch_data(url: str) -> dict:
2.     """Fetch JSON data from a given URL and return it
                                    as a dictionary."""
```

4.1.2 注释前留两个空格

行内注释与前方代码之间应至少有两个空格，便于阅读和统一格式。

示例：

```
1. count += 1  # 累加计数器
```

4.1.3 块注释写在代码前，行注释写在代码右侧

块注释用于解释代码逻辑，应写在相关代码之前。

行注释用于简短说明某一行代码用途。

示例：

```
1. # 读取配置文件
2. config = load_config()
3. db.connect()  # 初始化数据库连接
```

4.1.4 注释内容必须是完整语句，语义清晰

注释不得写成拼音缩写、首字母缩写或“无注释胜有注释”的模糊句子。避免“处理数据”、“逻辑优化”等含糊不清的描述。

4.1.5 Docstring 必须使用英文或中英文混排，禁止仅中文

在国际化团队或开源项目中必须使用英文。若项目为中文团队内部可使用中英文混排，但禁止全中文，避免工具不兼容或非通用。

4.2 推荐实践

4.2.1 函数 docstring 应包含标准结构

建议使用 Google、NumPy 或 reStructuredText 规范格式编写 docstring，并包含：

Args:：参数说明

Returns:：返回值说明

Raises:：可能抛出的异常说明

示例（Google style）：

```
1. def compute_average(scores: List[int]) -> float:
2.     """
3.     Compute the average score from a list of integers.
4.     Args:
5.         scores (List[int]): A list of score values.
6.     Returns:
7.         float: The average score.
8.     Raises:
9.         ValueError: If the list is empty.
10.    """
```

4.2.2 注释应解释目的，而不是代码语法

避免注释只是对代码的逐字翻译，注释应说明为什么这样做。

错误示例：# 设置变量为 1

正确示例：# 初始状态标志，表示未处理

4.2.3 类注释使用 Attributes: 列出公共属性

类级 docstring 应写清楚类的功能、用法，并列出关键属性及含义。

示例：

```
1. class User:
2.     """
3.     A user of the application.
4.     Attributes:
5.         username (str): The user's login name.
6.         email (str): The user's email address.
7.         is_active (bool): Indicates if the user is currentl
   y active.
8.     """
```

4.2.4 4.2.4 多行注释应对齐，使用统一缩进风格

多行块注释需对齐并使用 # 开头，每行前统一缩进，提高可读性。

示例：

```
# 该函数执行以下任务：
# 1. 验证输入
# 2. 执行数据处理
```

3. 返回计算结果

4.2.5 Docstring 应体现设计意图或约束条件

如果函数具有边界条件、性能限制、特定使用方式，应在 docstring 中指出。

"""本函数适用于列表长度不超过 1000 的情况，超出将降低性能。"""

4.2.6 对复杂正则表达式、算法逻辑必须注释说明

如果某段代码难以一眼理解，特别是涉及正则、数学逻辑、边界判断等，必须提供说明注释。

4.3 接受情况

4.3.1 私有或极短函数可用行内注释代替 docstring

对于内部使用、实现简单的私有函数，使用简洁的行注释代替 docstring。

示例：

```
1. def _reset():  # 重置缓存状态
2.     ...
```

4.3.2 测试代码或脚本中 docstring 可简化

在测试函数（如 test_something()）或一次性脚本中，docstring 可简略，但仍应标明作用。

4.3.3 生成代码可省略注释

对于通过模板、工具自动生成的代码文件，可以省略注释。但应在文件开头注明生成来源及不建议手动修改。

4.3.4 在交互式命令或调试中允许省略注释

交互式环境（如 Jupyter Notebook）中用于临时调试的代码块可以不注释，但若长期保留应补充说明。

五、 代码格式（Code Style）

5.1 强制要求

5.1.1 使用 4 个空格进行缩进，禁止使用 tab

Python 官方 PEP 8 明确规定每一层缩进使用 4 个空格。使用 tab 会在不同编辑器或平台下造成不一致，影响可读性和可维护性。大多数代码格式化工具（如 Black、YAPF）默认也遵循此规则。

5.1.2 每行代码长度不超过 80 个字符

为了提升可读性、便于多窗口编辑和版本管理，建议每行代码长度控制在 80 字符以内。某些例外场景可放宽（见 5.3.1），但不应作为常态。

5.1.3 禁止在行尾使用分号 ;

Python 并不依赖 ; 来分隔语句。使用分号会显得冗余，甚至误导他人以为有 C/C++ 风格的控制结构，违背 Python 的简洁哲学。

5.1.4 运算符两侧必须添加空格

所有二元运算符（如 =, +, -, ==）应在两边加空格，以增强可读性。

推荐写法：a = b + 1

错误示例：a=b+1

5.1.5 默认参数禁止使用可变类型（如 []、{}）

Python 中函数默认参数是函数对象的一部分，仅在定义时计算一次。使用可变类型会导致多个调用共享一个对象，产生意料之外的副作用。

正确示例：

```
1. def func(data=None):
2.     if data is None:
3.         data = []
```

5.2 推荐实践

5.2.1 使用括号换行，避免使用反斜杠 \

使用圆括号、方括号、花括号包裹内容时自动支持换行，安全性和可读性都优于反斜杠。

推荐：

```
1. result = (
2.     long_expression_1
3.     + long_expression_2
4.     + long_expression_3
5. )
```

5.2.2 容器（列表、字典、元组）多行定义时，最后一项后加逗号

这样可减少修改时产生的差异，提高版本控制友好性，也避免因漏写逗号造成语法错误。

示例：

```
1. settings = {
2.     "host": "localhost",
3.     "port": 8080,
4. }
```

5.2.3 函数与类之间空两行，类中方法之间空一行

在模块级保持结构清晰：函数与类定义间空两行，类内方法之间空一行，增强可读性与逻辑层次感。

5.2.4 优先使用 f"..." 字符串格式化

f-string 是 Python 3.6+ 提供的现代字符串格式化方式，语法简洁、性能更优，推荐替代 % 和 .format() 方法。

示例：

```
1. name = "Alice"
2. print(f"Hello, {name}!")
```

5.2.5 使用空行分隔逻辑块

在一个函数体内部，如果有明显的逻辑分段，应使用空行划分，提升结构清晰度，避免“代码墙”。

5.2.6 按需添加括号提高表达式可读性

当表达式较复杂、优先级不明显时，使用括号可以消除歧义，提升直观性，即使在语法上不是必须的。

5.3 可接受情况

5.3.1 特殊情况下允许超过 80 字符

以下情况可以合理地突破 80 字符限制：

长 URL 或路径字符串

错误堆栈追踪信息

使用格式化工具（如 yapf、black）自动生成的不可拆行表达式

保持注释或文档字符串格式整洁时

这些例外应控制数量，不应影响整体风格一致性。

5.3.2 可对齐赋值语句或结构体定义

在字典、参数表、配置项等结构中，为了可读性可使用对齐方式书写，但不要过度追求对齐导致频繁修改。

示例：

```
1. config = {
2.     "timeout"  : 30,
3.     "retries"  : 3,
4.     "loglevel" : "INFO",
5. }
```

5.3.3 调试/临时代码中允许格式放宽，但不可提交

本地调试过程中的代码格式可适度放宽要求，但在正式提交代码前必须整理格式，保证代码质量。

六、 其他规范（Other Guidelines）

6.1 强制要求

6.1.1 使用主函数保护结构 if __name__ == "__main__"

所有脚本的主执行入口必须用该结构保护，防止在模块被导入时误执行主程序逻辑。

```
1. def main():
2.     ...
3. if __name__ == "__main__":
4.     main()
```

6.1.2 禁止模块导入时产生副作用

顶层导入模块时不得自动执行函数调用、打开文件、打印日志、连接数据库等行为，所有操作必须封装到函数中并由调用方显式触发。

6.1.3 禁止模块间循环导入

循环依赖会导致导入失败或运行时异常，必须通过结构重构、延迟导入或抽象公共模块解决。

6.1.4 禁止在导入时执行耗时操作

如模型加载、API 调用、数据库连接等必须延后至主流程执行，避免模块导入变慢或崩溃。

6.2 推荐实践

6.2.1 多字符串拼接使用 str.join() 替代 +

特别是在循环或列表拼接场景中，join() 性能和可读性都优于 +。

```
1. names = ["Alice", "Bob", "Charlie"]
2. result = ", ".join(names)
```

6.2.2 布尔判断避免使用 == True/False

推荐使用 if x:、if not x: 替代 if x == True，更清晰、语义更自然。

```
1. if is_active:
2. if not is_deleted:
```

6.2.3 对 None 的判断使用 is 或 is not

因为 None 是单例，推荐使用身份判断而非相等判断。

```
1. if value is None:
2.     ...
```

6.2.4 使用 argparse 或 typer 解析命令行参数

避免手动解析 sys.argv，使用专用库能提供更好的帮助信息与错误处理。

6.2.5 统一使用 logging 模块输出日志

替代 print()，并设置日志等级、格式和输出位置。

6.2.6 使用 with 管理资源生命周期

对于文件、锁、连接等资源使用上下文管理器，确保资源释放。

```
1. with open("file.txt") as f:
2.     content = f.read()
```

6.2.7 使用类型注解（Type Hints）

为函数参数和返回值添加类型，有助于静态检查工具如 mypy。

```
1. def get_name(user_id: int) -> str:
2.     ...
```

6.2.8 避免魔法数字，使用具名常量或枚举

所有业务相关的硬编码值都应定义为命名常量，提升代码可读性与可维护性。

6.2.9 文件应以一个空行结尾

保证兼容 Unix 工具链和版本控制工具，不可省略结尾换行符。

6.2.10 合理实现魔法方法

实现 __str__, __repr__, __eq__, __lt__ 等魔法方法应与类的业务语义一致，不可滥用。

6.3 可接受情况

6.3.1 使用 Cython、Numba 优化性能瓶颈

可用于对计算密集型函数进行加速，但应明确注释原因、范围，并保留原始逻辑可读性。

6.3.2 开发阶段可使用 print()，但提交前应替换为日志系统

快速调试可用 print()，最终应统一切换为 logging 或 stderr 输出。

6.3.3 可使用 pass、TODO、FIXME 占位，但需后续处理

应配合注释说明占位原因，确保后续开发者能够识别并处理。

```
1. # TODO: 添加分页逻辑
2. def fetch_data():
3.     pass
```

6.3.4 使用 lambda 简化短函数，但勿滥用

简短函数或临时表达式可用 lambda，复杂逻辑应定义具名函数。

sorted(data, key=lambda x: x.id)

6.3.5 可临时禁用 lint 规则，但必须说明理由

使用如 # noqa 或 # pylint: disable=... 时，需在旁边加注释说明禁用目的。

6.3.6 使用环境变量控制配置，但需设置默认值并编写文档

应避免程序因环境变量缺失而崩溃，推荐使用 os.getenv("KEY", default) 并在 README 中注明支持的变量。

结语

本规范旨在通过统一的编码风格、清晰的命名体系、严格的异常处理、规范的文档注释和一致的格式控制，提升 Python 项目的整体质量和可维护性。规范并不是限制创新的工具，而是帮助团队建立清晰沟通、协作顺畅、高效交付的基础。

编码是一种沟通形式。良好的代码应该像文字一样易于理解、审阅与传承。希望团队成员不仅能遵守本规范，更能从中理解背后的设计哲学和工程价值，在实践中不断完善、优化，并根据实际需求适时演进。

如有特殊业务需求或风格偏好，可通过团队共识形成本地扩展规则，并在不违背本规范核心原则的前提下进行适度调整。

让我们一起写出更整洁、更清晰、更专业的 Python 代码。