# 技术管理专项作业

本规范汇总了Python开发的代码风格和项目管理最佳实践，参考了PEP 8、Google、Facebook、阿里巴巴等大型公司的规范。每条规则按重要程度标记为 a. 强制（必须严格遵守）、b. 推荐（强烈建议遵守，除非有特殊理由）、c. 允许（在特殊情况下可以接受）。本规范覆盖项目结构、命名、代码格式、异常处理、日志、测试、版本控制、依赖管理和文档等方面。

**1.项目结构：**

a1. 模块化组织代码：必须将代码按功能划分为不同模块或包，避免将所有代码堆在单一文件中。

a2. 禁止循环依赖：模块之间不得存在循环导入，确保依赖关系单向且清晰。

b1. 单一职责原则：建议每个模块、类或函数只承担单一职责，保持代码功能内聚、清晰。

b2. 规范项目目录：按照职责分类组织文件夹，例如测试代码放在tests/目录，配置文件放在config/目录，脚本放在scripts/目录等。

**2.命名规范：**

a1. 遵循PEP8命名：类名使用帕斯卡命名（CamelCase，首字母大写，异常类以“Error”结尾）；函数和变量名使用小写加下划线命名（snake\_case）。

b1. 常量命名：常量采用全大写加下划线命名（例如MAX\_RETRIES），以体现不可变属性。

b2. 模块与包命名：模块（文件）名使用短小全小写字母，可加下划线；包（文件夹）名使用简短全小写字母，避免使用下划线。

b3. 名称语义清晰：命名应自解释，避免使用模糊缩写。不要仅通过大小写区分名称，避免与内置名字冲突；尽量不使用单字母名称（除非在局部循环计数等情境）。

**3.代码格式：**

a1. 缩进风格：代码缩进必须使用4个空格，禁止使用制表符(Tab)。

a2. 行宽限制：单行代码长度不应超过80个字符，必要时应换行以提高可读性。

a3. 语句终止：禁止在行末添加多余的分号；不得在同一行书写多个独立语句。

b1. 空行分隔：使用空行分隔代码块。函数和类定义之间空两行，类内部方法之间空一行，以增强代码可读性。

b2. 空格使用：在二元运算符两侧、逗号和冒号后应当加空格，但逗号前、括号内侧不加空格，避免为了对齐额外添加多余空格。

b3. 导入顺序：import按标准库、第三方库、本地模块的顺序分组导入；尽量使用绝对导入，避免相对导入造成模块命名冲突。

b4. 换行规则：多行语句优先使用括号实现隐式换行，避免使用反斜杠\续行；拆分长表达式时，在操作符前断行而非后断行以提高可读性。

c1. 允许尾逗号：在多行排列的列表、字典等结构中，可以在最后一个元素后保留一个逗号，以方便后续添加元素时的版本对比。

**4.异常处理：**

a1. 禁止裸异常捕获：禁止使用空的except:或不指明异常类型的通用异常捕获，避免无处理地拦截所有异常。

b1. 使用恰当异常：针对不同错误使用准确的异常类型来处理。优先使用内置异常类表达具体问题，必要时可自定义异常（命名以“Error”结尾），并提供有意义的错误消息方便调试。

b2. 确保资源清理：使用finally子句或上下文管理器(with语句)来确保资源释放（如关闭文件、网络连接），即使发生异常也会执行清理操作。

c1. 顶层统一捕获：仅允许在程序入口等高层位置使用广泛的异常捕获（如except Exception）来记录错误日志。在记录后应采取适当措施（如安全退出或重新抛出），避免程序静默失败。

**5.日志记录：**

a1. 使用日志模块：在生产代码中必须使用Python内置的logging模块进行日志输出，禁止使用print打印进行日志记录。

b1. 配置日志系统：根据运行环境配置适当的日志设置，包括日志级别、格式、输出渠道（控制台、文件等），在开发调试时使用较详细日志级别，在生产环境降低日志详细程度。并确保日志配置包含必要的上下文信息但不泄露敏感数据。

b2. 合理设置级别：划分日志严重级别——调试细节使用DEBUG，一般运行信息使用INFO，警示内容使用WARNING，错误和异常使用ERROR或CRITICAL。日志内容应包含必要的上下文信息以协助定位问题，但避免记录如密码等敏感数据。

c1. 调试输出限定：开发过程中允许用print做临时调试输出，但提交代码前应移除或改用正式的日志记录方式。

**6.测试策略：**

a1. 单元测试编写：必须为核心模块和功能编写单元测试。使用pytest、unittest等框架，并遵循测试命名规范（测试文件以test\_开头放在tests/目录，测试函数以test\_开头命名）。

b1. 测试覆盖率：建议测试覆盖率达到80%以上，重点覆盖关键的业务逻辑和边界情况，但不过分追求百分比而忽略测试质量。

b2. 持续集成测试：将测试集成到持续集成流程中，在每次代码提交或合并时自动运行全部测试，用于及时发现回归问题。

c1. 扩展测试类型：允许根据需要编写集成测试、性能测试等额外测试来验证系统整体行为，但应将此类测试与单元测试区分开（例如放在不同目录或使用不同标记），以免影响单元测试的独立运行**。**

**7.版本控制：**

a1. 强制使用VCS：所有代码必须使用版本控制系统管理（优选Git），并且勤于提交；每次提交都要包含清晰的提交说明，准确描述更改内容。

b1. 分支开发与审核：采用合理的分支模型进行协作开发，例如Git Flow或功能分支工作流。功能开发在独立分支进行，通过Pull Request进行代码审核后再合并到主分支，避免直接推送代码到主分支。

b2. 忽略不必要文件：配置并维护好项目的.gitignore文件，避免将编译生成的文件、敏感配置或无关文件纳入版本库。

c1. 历史记录管理：允许在个人开发分支中使用rebase、amend等操作来整理提交历史，但禁止修改共享分支的历史记录（避免影响他人同步代码）。

**8. 依赖管理：**

a1. 虚拟环境隔离：必须使用虚拟环境（如venv或conda）来隔离项目依赖。项目所需的第三方库及其版本必须在requirements.txt、Pipfile等依赖文件中明确列出。

b1. 依赖版本锁定：建议固定关键依赖的版本或使用锁定文件（如Pipfile.lock、poetry.lock）来确保环境可重复构建。

b2. 定期更新清理：定期更新依赖以获得最新的安全补丁和功能改进，并移除不再使用的依赖库，保持项目依赖精简。

c1. 工具和特殊情况：可以使用pipenv、Poetry等工具进行依赖管理，但团队应统一使用其中一种方案。特殊情况下也允许将体积小且重要的依赖库直接内置在项目中（vendoring）。

**9.文档：**

a1. 完善代码文档：所有公共模块、类、函数必须编写文档字符串（docstring），并遵循统一的格式风格（例如reST或Google风格）清晰描述其功能和用法。

b1. 项目说明文件：在项目根目录提供清晰的README文档，简介项目概览、安装使用方法和示例等，方便他人了解和使用，并应随着代码演进及时更新相关文档。

b2. 注释规范：代码注释应当简洁明了。对于复杂逻辑或重要代码段，使用块注释进行说明（每行均以#开头）；避免大量使用行内注释，必要的行内注释应与代码以两个空格隔开，并确保表达清楚。

c1. 内部接口文档：对于内部使用的且非常简单明了的辅助函数，允许不撰写文档字符串；但对外提供的接口仍必须具备完善的注释和文档说明。