**文件路径**

在学习关于路径的各种操作之前，我们首先明确路径是什么，能不能细分。路径保存文件或文件夹所在位置的信息，比如 data/documents/report.txt，分为 2 部分，目录和文件名/文件夹名，第一部分是从根目录开始一直到最后一个斜杠之前的内容，这是目录；第二部分则是文件名（含扩展名），或者是文件夹名，或者为空。路径就好比是某个小区某单元某楼层某房间号，可以确认住户的位置信息，住户的名字即为文件名，男女可类比为扩展名；也可以不精确到人，把房间号作为文件夹名，房间号前面的为目录。

**基本操作**

文件路径也是要从基本的操作学习：

创建：路径（包含文件）与目录（不包含文件）

获取：拆分文件路径

验证：判断路径/目录是否存在；

路径的基本操作在 Python 里有 2 种实现方式，一种是传统的 os.path 模块，另一种是 Python 3.4 之后引入的 pathlib 模块，需要注意的是，os 模块操作的是字符串，数据类型是 str，而 pathlib 模块提供了面向对象的路径操作方式，数据类型是 Path，二者混合使用时要记得进行数据类型转换。

```
import os

from pathlib import Path
```

在 Python 中，处理文件路径的基本操作通常涉及以下几个方面：

拼接路径：把文件所属的各级目录以及文件名和扩展名组合起来，创建路径之后，该路径下并不存在空文件，后面需要对文件进行操作。

```
#os.path.join 函数跨平台安全拼接或使用字符串拼接
file_path1 = os.path.join("data", "documents", "test.txt")
file_path2 = Path("data2")/ "documents" / "report.txt" #  自动处理路径分隔符

 # 输出: data/documents/report.txt（Windows 下为 data\documents\report.txt）
```

需要注意的是各级目录是分开的还是自身就已使用斜杠符号连接起来了。

创建目录：创建文件所在的**目录即文件夹**，不是文件。（这里注意区分 os 模块的 makedirs 和 path 模块下的 mkdir 操作均是递归创建多层）

os.makedirs(dir_path3, exist_ok=True) # exist_ok=True 避免目录已存在时报错

dir_path4.mkdir(parents=True, exist_ok=True)

创建文件

（1）创建空文件

如果只需要创建一个空文件，可以用以下简化方式：

```
# 方法 1：使用 open()
open('file_path1', 'w').close()
# 方法 2：使用 pathlib
Path('file_path2').touch()  # touch()方法模拟 Unix 的 touch 命令
```

（2）使用内置的 open() 函数（最常用）

通过写入模式打开文件，如果文件不存在会自动创建。

```
# 方法 1：文本模式创建（默认）
with open('file_path1', 'w', encoding='utf-8') as f:
        f.write('这是通过 open()创建的文本文件')
# 方法 2：二进制模式创建
with open('image.bin', 'wb') as f:
        f.write(b'\x00\x01\x02')  # 写入二进制数据
```

'w'：文本模式，覆盖已有内容（不存在则创建）

'a'：文本模式，追加内容（不存在则创建）

'x'：文本模式，仅创建新文件（如果文件已存在会报错）

加上 b 后缀（如 'wb'）表示二进制模式

（3）使用 pathlib.Path 的 write_text（）函数

```
# 方法 1：直接写入内容
file_path2.write_text('这是通过 pathlib 创建的文本文件', encoding='utf-8')
# 方法 2：二进制写入
bin_path = Path('data.bin')
bin_path.write_bytes(b'\x03\x04\x05')
```

# 检查路径存在：判断该路径是否存在（包括文件和纯目录）。操作文件前先检查路径是否存在 写文件时目录不存在会触发 FileNotFoundError

exists = os.path.exists(path)

exists = file_path.exists()

# 输出: True

获取文件名：从已知路径中提取文件名（包括扩展名）或仅提取文件名（不含扩展名）。

# 获取文件名（带扩展名）

filename = os.path.basename(path)

filename = dir_path.name

# 输出: report.txt

# 获取文件名（不带扩展名，其本质是将文件名进行切分得到文件名和扩展名的列表，然后取列表的第一个值）

filename_no_ext = os.path.splitext(filename)[0] 或者 os.path.split('.')[0]

filename_no_ext = file_path.stem

# 输出: report

获取目录名：从已知路径中提取目录部分。

dir_path = os.path.dirname(path)

dir_path = file_path.parent

# 输出: data/documents

上面获取目录以及文件名就涉及到路径的拆分，将路径拆分成目录和文件名 2 部分。

路径的拆分和组合：使用 os.path.split、os.path.splitext 等函数拆分路径。

*获取当前工作目录*：根据当前运行代码所在的路径和文件名称获取文件所在的目录；

current_dir = os.getcwd() # *返回当前 Python 脚本的工作路径*

# *获取绝对路径*

relative_path = "demo.txt"

abs_path = os.path.abspath(relative_path) # *将相对路径转为绝对路径*

这个绝对路径前面的目录是当前工作目录，隐式获得的。其本质是将一个路径进行了拆分，一般是将从根目录到当前文件所在的目录得到相对目录，然后获取基于这个目录的相对文件位置。

```
# 将绝对路径改为相对路径：
if os.path.isabs(abs_path2):
    rel_path = Path(abs_path2).relative_to(current_dir1)
    print('rel_path',rel_path)
```

删除路径/目录

在 Python 中，删除文件或目录有两种常见的方法：

删除文件：，不能删除目录

os.remove()&dir_path.unlink()

删除空目录：

os.rmdir()&dir_path.rmdir()

删除目录（包括非空目录）：递归删除整个目录树，即目录下的所有子目录和文件都会被删除

shutil.rmtree()

**关键操作对比表**

| **操作** | **os.path 方式** | **pathlib 方式** |
| --- | --- | --- |
| 路径拼接 | os.path.join("a", "b", "c.txt") | Path("a") / "b" / "c.txt" |
| 获取文件名（带扩展名） | os.path.basename(path) | path.name |
| 获取文件名（无扩展名） | os.path.splitext(name)[0] | path.stem |
| 获取父目录 | os.path.dirname(path) | path.parent |

| | | |
|---|---|---|
| 创建目录 | os.makedirs(path, exist_ok=True) | path.mkdir(parents=True, exist_ok=True) |

**注意事项**

**跨平台兼容性**

使用 os.path.join() 或 pathlib 的 / 运算符可自动处理 Windows（\）和 Linux/macOS（/）的路径分隔符差异,Windows 路径前加个 r 可以识别

避免手动拼接路径（如 "data" + "/" + "file.txt"）

**文件编码**

默认使用系统编码（可能引发乱码） 显式指定编码更安全：Path.read_text(encoding="utf-8")

**路径规范化**

使用 os.path.normpath() 或 Path.resolve() 解析相对路径和符号链接 **推荐：**

新项目优先使用 pathlib，提供更面向对象和简洁的 API，减少代码量 30% 以上。

**文件夹遍历 os.listdir(path)和 os.walk(root)**

在深度学习中，常用的一种操作是遍历很多张图片处理做成数据集，这里就需要遍历文件夹的函数，一般有两个方法，os.listdir(path),会将路径下的文件夹以及文件名称储存到列表里，但不会递归到子文件夹里。

深度学习数据集的文件夹通常是多级的，比如图片是分类存放的，就有主文件夹，类别子文件夹，然后是图片文件，这时更常用的是 os.walk(root)，一般和 for 循环结合使用。会先遍历根文件夹得到根文件夹路径，以及下面的一级子文件夹路径列表（没有则为空）和一级子文件列表（没有则为空），返回的是一个深度递归的三元组（path，dirnames，filenames），然后把每一个子文件夹作为根目录继续深度循环遍历。

举个例子，

-dir1

-1.jpg

-subdir1

--2.jpg

--3.jpg

-subdir2

--4.jpg

--5.jpg

如果是 os.listdir(dir1),输出就是 1.jpg,subdir1,subdir2。

而 for root,dir,file in os.walk(dir1) :

第一次循环遍历输出三元组

（dir1 [subdir1，subdir2]，[1.jpg]）for 循环的 root，dir，file 分别来承接元组的三个变量；

第二次遍历输出三元组

dir/subdir1，[]，[2.jpg，3.jpg]

第三次遍历输出三元组

dir/subdir2，[]，[4.jpg，5.jpg]

**os.mkdir 和 os.makedirs 的联系和区别是什么**

在 Python 的 os 模块中，os.mkdir() 和 os.makedirs() 都是用于创建目录的函数，但它们的功能和使用场景有所不同，具体联系和区别如下：

联系

- 两者的核心作用相同：均用于在文件系统中创建目录（文件夹）。
- 都需要传入一个路径参数，指定要创建的目录位置。
- 若目标目录已存在，调用时都会抛出 FileExistsError 异常。

区别

| 特性 | os.mkdir(path) | os.makedirs(path, exist_ok=False) |
| --- | --- | --- |
| 功能 | 仅能创建**单个目录**（最底层目录）。 | 可以创建**多级目录**（包括中间不存在的目录）。 |
| 对父目录的要 | 要求路径中的所有父目录必须已存在，否则抛出 | 自动创建路径中所有不存在的父目录，无需手动提前创建。 |

| 特性 | os.mkdir(path) | os.makedirs(path, exist_ok=False) |
| --- | --- | --- |
| 求 | FileNotFoundError。 | |
| 参数差异 | 无额外重要参数。 | 有 exist_ok 参数（默认 False）：<br>- 若设为 True，目标目录已存在时不会抛出异常；<br>- 若设为 False（默认），目标目录已存在时抛出 FileExistsError。 |
| 适用场景 | 仅创建单级目录，且父目录已存在的场景。 | 需要创建多级目录（如 a/b/c），或不确定父目录是否存在的场景。 |

**os.path.dirname(__file__)**

__file__，指代当前运行的代码文件。

KB_ROOT_PATH=os.path.join(os.path.dirname(os.path.dirname(__file__)),'knowledge-base')这行代码用于**动态获取项目中** knowledge-base **目录的绝对路径**，其含义可拆解为以下几个部分：

__file__：

这是 Python 的内置变量，表示当前脚本文件（即包含这行代码的 .py 文件）的**相对路径或绝对路径**（取决于执行脚本的方式）。例如，若当前脚本路径为 D:/RAGPro/server/config.py，则 __file__ 的值为 'D:/RAGPro/server/config.py'。

os.path.dirname(path)：

返回 path 路径中**目录部分**（即去掉文件名，保留上级目录）。例如：

若 path = 'D:/RAGPro/server/config.py'，则 os.path.dirname(path) 返回 'D:/RAGPro/server'。

os.path.join(path1, path2)：

用于拼接两个路径，自动适配当前操作系统的路径分隔符（Windows 用 \，Linux/macOS 用 /）。

路径计算过程拆解

假设当前脚本（包含这行代码的文件）的路径为：D:/RAGPro/server/config.py

则代码的计算步骤如下：

**第一次** os.path.dirname(__file__)：

对 __file__（'D:/RAGPro/server/config.py'）取目录，得到 'D:/RAGPro/server'（当前脚本所在的目录 server）。

**第二次** os.path.dirname(...)：

对上一步结果（'D:/RAGPro/server'）再次取目录，得到 'D:/RAGPro'（server 的上级目录 RAGPro）。

os.path.join(...)

**拼接路径**：将上一步的结果（'D:/RAGPro'）与 'knowledge-base' 拼接，最终得到：

'D:/RAGPro/knowledge-base'

**as_posix（）**

pathlib 模块中 Path 对象的一个方法 as_posix()，**用于将路径转换为 POSIX 格式（使用正斜杠/作为分隔符的字符串），即**在 Windows 上：将反斜杠 \ 转换为正斜杠 /；在 Linux/macOS 上：保持不变（因为已经是 POSIX 格式），它在跨平台开发中特别有用，保证路径格式的一致性。操作**返回字符串**（不是 Path 对象）

```
# Windows 路径示例
win_path = Path(r"C:\Users\Alice\文档\file.txt")
print(win_path.as_posix())
# 输出: C:/Users/Alice/文档/file.txt
# Linux/macOS 路径示例
linux_path = Path("/home/alice/文档/file.txt")print(linux_path.as_posix())
# 输出: /home/alice/文档/file.txt
# 混合路径示例
mixed_path = Path("C:\\Users/Bob\\文件//report.docx")print(mixed_path.as_posix())
# 输出: C:/Users/Bob/文件/report.docx
```

**模块（类和函数）的导入**

实际的项目中不可能一份代码文件实现所有的功能，因此需要借助外在的力量，这份力量主要有 2 类，一种是系统库集成的，像 numpy 处理数据，另一种是自己项目写的函数和类，但是放到不同的位置。

当前的代码怎么使用这些外在的力量（类和函数）呢？

方法就是 import 导入，将其它外在的类和函数导入到当前运行的代码文件里，下面详细学习一下。

**基本原理**

首先明确几个概念，项目中的文件夹对应 Python 的包/库 package，文件对应模块 module，模块的名称就是 Python 代码文件取消扩展名.py。

包 package 和模块 module 的导入核心就是找到模块并完成导入，问题来了，如何找到模块呢？找不到模块也是经常导入报错的原因，答案就是在设置的路径里寻找。具体来说，寻找导入模块的步骤如下：

1.在 sys.modules 缓存中查找是否已经导入过该模块，如果存在则直接使用。这一步通常不进行人为的操作；

2.如果不在缓存中，则搜索模块文件，这部分是需要重点关注的，排查导入报错问题也是需要先看这块。搜索路径由 sys.path 定义，一般包括当前目录、环境变量 PYTHONPATH 指定的目录以及 Python 安装目录（这个一般是 conda 或 pip 安装指定的目录）。可以手动添加路径。

3.除了直接在路径里寻找之外，还经常使用的是相对路径导入，这个后面会详细介绍分析。

找到模块文件之后，就是将模块的代码整合到运行的代码里，具体如下：

1. 找到模块文件后，将其编译为字节码（如果必要），然后执行模块中的代码（注意：模块顶层的 import 代码在导入时会被执行，下面的代码看是否有 if __name__ == __main__）。

2.将模块对象添加到`sys.modules`缓存中，并创建到当前命名空间的引用（根据导入方式）。

**导入分类**

从导入内容和数量来看，

可以直接导入整个模块，

import module_name;

使用的模块中的内容 ：

module_name.function_name 或 module_name.Class_name

可以导入模块的部分函数 from *module_name* import *function_0*, *function_1*，*此时*直接使用函数名或类名，无需模块名前缀

*也可以导入模块的全部函数*

from *module_name* import *

因为有的模块和函数名字比较长，或者要导入的函数的名称可能与程序中现有的名称冲突，可以在导入的时候使用 as 起个别名

import module_name as new_name

从导入方式来看，

分为相对导入和绝对导入，其区别为是否写出模块的绝对路径。

项目根目录与顶级包

下面我们重点区分一下项目根目录和顶级包这个概念，为学习后面的相对导入和绝对导入做好准备。

项目根目录（project_root，项目名）本身不是一个 Python 包，因为它没有__init__.py，但当我们将项目根目录添加到 sys.path 后，就可以直接导入其下面的子文件夹作为包 package，找不到主目录时，可以手动设置：

```
# 设置项目根目录
sys.path.insert(0, os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
# 在 main 文件里添加项目根目录到 sys.path
sys.path.append(os.path.dirname(os.path.abspath(__file__)))
```

注意一般情况下项目名称不是顶级包，下面的子文件夹才是顶级包。

**项目根目录 vs 顶级包 关键区别**

| **特性** | **项目根目录** | **顶级包** |
| --- | --- | --- |
| 定义 | 包含整个项目的文件夹 | 项目中包含 __init__.py 的顶级目录 |
| Python 识别 | 不在 sys.path 中时不被识别 | Python 将其视为可导入的包 |
| __init__.py | 不需要 | 必须存在 |
| 导入方式 | 不可直接导入 | 可直接导入 |

典型位置　包含多个包的父目录　　项目根目录下的一级子文件夹

绝对导入就是从顶级包开始，每个子包都写上

import package1.package2... .module1

相对导入就是确定要导入的模块和当前运行模块的相对位置关系，相对导入本质上也是转换成绝对导入，其连接点就是当前运行模块所归属的__package__属性。根据这个属性，Python 会得到当前运行模块所属的包，然后根据前面有几个点（.）来推导出要导入模块的父包

from . import module → from my_package import module

以下面的实例详细解释一下**点号含义**和相对导入的使用方法：

```
project_root/                    <-- 项目根目录
├── main.py
├── core/                        <-- 一级子包
│   ├── __init__.py
│   ├── utils.py
│   └── services/                <-- 二级子包
│       ├── __init__.py
│       ├── api.py
│       └── auth.py
└── lib/                         <-- 一级子包（同级目录）
    ├── __init__.py
    ├── network.py
    └── storage/
        ├── __init__.py
        └── cloud.py
```

**点号含义：**

. 表示当前文件所在的目录（包），.. 表示上一级目录（父包），... 表示祖父包（但不推荐使用），例如

在 core/services/api.py 中：. = core.services，.. = core

在 lib/network.py 中：. = lib，.. = lib

在 core/utils.py 中：. = core，.. = 项目根目录（无父包）

以上面目录结构为例，相对导入可以导入当前包下的模块，子包下的模块（子包.module），具体到代码里：

假设我们在 core/services/api.py 中：

```
# ☑ 正确用法
from . import auth              # 导入同级的 auth.py
from .. import utils            # 导入父级的 utils.py
```

# ❌ 错误用法

from ... import main　　# 错误！超过两级父目录

from ..lib import network　　# 错误！不能跨兄弟目录

也就是说，相对导入一般最多只导入父包下的模块，和父包同级的，父包上级的模块都不能导入。

相对导入的问题

因为相对导入也是要转换为绝对导入，而且是依赖__package__属性的，就会出现一个相对导入最常出现的问题，找不到包。

相对导入只能在 package 里面的 module 使用，并且使用了相对导入的模块必须由其它模块调用而不能直接运行。

当直接运行包内的模块文件时：

python my_package/submodule.py

会发生下面的问题：

Python 将该文件视为**顶级脚本（类似于 main.py）**而非包的一部分，__package__ 被设置为 None，不知道其属于哪个 package，则相对导入无法确定父包位置，也就找不到其它模块，导入失败并抛出 ImportError

解决方案有下面几种：

方案 1：使用外部入口点（推荐），比如在 main.py 主文件里调用运行

*# 项目根目录 /main.py*

import sys

import os

*# 导入包内的模块*

from my_package import submodule

if __name__ == "__main__":

　　submodule.run()

运行：python main.py

方案 2：使用 -m 参数（模块运行模式）

*# 在项目根目录运行*

python -m my_package.submodule

注意是 module，不是.py，也就是去掉.py 的模块名。

这样 Python 会：识别完整的包路径 my_package.submodule 并正确设置 __package__ = "my_package"启用相对导入

方案 3：动态修复包上下文（高级技巧）

*# submodule.py*

*顶部添加* if __name__ == "__main__" and __package__ is None:

*# 手动设置包上下文*

import os

import sys

*# 计算包路径 (my_package.submodule → my_package)*

file_path = os.path.abspath(__file__)

base_dir = os.path.dirname(os.path.dirname(file_path))

package_name = os.path.basename(base_dir)

*# 添加到 sys.path*

if base_dir not in sys.path:

sys.path.insert(0, base_dir)

*# 设置包上下文*

__package__ = package_name

*# 重新导入当前模块（确保相对导入生效）*

import importlib

importlib.import_module(__name__)

### 注意事项：

1. **循环导入**：两个模块相互导入可能导致问题，应尽量避免。可以通过重构代码（如将导入语句放在函数内部或使用延迟导入）来解决。

2. **重复导入**：多次导入同一个模块，实际上只会执行一次（由于`sys.modules`缓存）。但可以多次使用`import`语句，不过通常只导入一次。

3. **模块执行**：导入模块时，模块顶层的代码（包括 print 语句、函数定义等）会被执行。因此，通常将可执行代码封装在`if __name__ == '__main__':`中，避免在导入时执行。

4. **命名冲突**：使用`from ... import *`可能导致当前命名空间中已有的名字被覆盖。推荐使用导入整个模块并起别名的方式，或者明确导入需要的名字。

**动态导入**

在代码的实际运行中，我们会遇到根据参数导入特定的模块，这时可以使用动态导入模块。importlib.import_module 提供了这种动态导入的能力。import_module 函数内部调用了`__import__()`函数，它返回导入的模块对象，我们可以通过该对象访问模块中的内容。

importlib.import_module（package 名或者模块名）

```
def get_loader(loader_name: str, file_path: str, loader_kwargs: Dict = None):
    '''
    根据 loader_name 和文件路径或内容返回文档加载器。
    '''
    loader_kwargs = loader_kwargs or {}
    try:
        if loader_name in ["RapidOCRPDFLoader", "RapidOCRLoader"]:
            document_loaders_module =
importlib.import_module('document_loaders')
        else:
            document_loaders_module =
importlib.import_module('langchain.document_loaders')
        DocumentLoader = getattr(document_loaders_module, loader_name)
    except Exception as e:
        msg = f"为文件{file_path}查找加载器{loader_name}时出错：{e}"
        logger.error(f'{e.__class__.__name__}: {msg}',
                     exc_info=e if log_verbose else None)
```

document_loaders_module = importlib.import_module('langchain.document_loaders')

DocumentLoader = getattr(document_loaders_module, "UnstructuredFileLoader")

关键要点：

动态导入虽然强大，但应谨慎使用。在大多数场景下，静态导入仍然是首选，因为它更易于理解和调试。动态导入最适合需要运行时灵活性的特定场景。

优先使用 importlib.import_module() 而非 __import__()

始终验证动态导入的输入来源

合理利用缓存避免重复导入开销

为动态导入提供清晰的错误处理

理解 Python 导入机制底层原理

**问题-运行没错，IDE 检查报错**

项目名称为 xirang，一级子文件夹为 RAGPro，下面有子文件夹 server，configs，和子文件 startup，现在 2 个困境，在 RAGPro 目录下 Python 运行 startup.py 时，子文件夹 server 等下面的文件的导入需要 from server import，如果是 from RAGPro.server import 就报错找不到 RAGPro，但如果是 from server import，运行时不会报错，但代码本身检查时会报错找不到 server，configs，也定位不到从这里面导入的模块，如何解决

直接运行时使用 from server import 可以工作，因为 Python 将当前目录添加到 sys.path

但代码检查器（如 IDE）无法识别这种相对导入方式

使用绝对导入 from RAGPro.server import 会报错，因为 RAGPro 不是一个已安装的包

**参数的传递**

代码在运行时，需要传入不同的参数。对于简单程序来说，参数可以直接写在文件里，但是这样修改维护不方便，Python 动态维护参数的常用方法有 2 类，一类是处理命令行的 args 类，另一个就是把参数写在 config 配置文件里作为模

块导入到需使用的文件。

**argparse**

argparse 用来解析命令行参数和选项

分为三个步骤

1 创建参数解析器 ：是参数解析系统的"配置器"，负责定义参数规则，在下面的添加参数后基本不变，ArgumentParser 类实例

parser=argparse.ArgumentParser()

2 解析器添加参数：

parser.add_argument("num1",type,required,default)

parser.add_argument("num2",type,required,default)

3 输出解析结果：是参数解析系统的"输出"，包含解析后的实际值，每次运行根据命令行输入变化，Namespace 类实例

args=parser.parse_args()

三个步骤形成了一个完整的工作流：定义规则 → 解析输入 → 使用结果

注意这里解析参数得到的 args 是一个对象，而前面获取的参数就是它的属性，因此在代码中使用参数就是获取对象的属性，方法为对象名.参数名，即 args.num1 ， args.num2

或者使用 vars()将解析值转换成字典对象。

我们重点关注分析一下第二步的添加参数到底是在做什么？首先要明确，解析器要添加读取的是命令行上的内容，这些内容就是参数的值，而 add_argument()括号里的内容就是在指定参数的名称，参数值的数据类型，数量，帮助信息等。

参数的类型有 2 种，位置参数（参考函数的位置参数）和选项标志，

**添加位置参数（必需参数）**

位置参数（类似于函数的位置参数）：在 add_argument()`中只提供参数名称（不包含前缀`-`或`--`）。例如：`parser.add_argument("filename")`。

```
parser.add_argument(
    "filename",   # 参数名称
    type=str,     # 参数转换函数
    nargs="+",    # 接受一个或多个值
```

help="输入文件路径"  # 帮助信息)

添加位置参数时需要注意什么？

1. 顺序性：位置参数在命令行中出现的顺序必须与程序中定义的顺序一致。

2.必需性：默认情况下，位置参数是必需的。如果用户没有提供，argparse 会报错。如果希望位置参数可选，可以通过设置 nargs 参数（例如 nargs='?'）来实现。如果设置了`nargs='?'`，则可以为位置参数设置默认值。

运行时在命令行输入 python example.py test.docx，这个 test.docx 就是参数 filename 的值。

**添加可选参数**

parser.add_argument(

"-v", "--verbose",  # 短选项和长选项

action="store_true",  # 参数行为

dest="show_details",  # 结果对象中的属性名

help="显示详细输出"  # 帮助信息)

可选参数：在 add_argument()中提供了以-或--开头的选项标志。

（1）-v 和--verbose 是用来定义同一个命令行选项的两种形式。它们的主要区别在于：-v 是短选项（一个短横线加一个字母），便于快速输入；而 --verbose 是长选项（两个短横线加一个单词或词组）。长选项含义更清晰，更易阅读和理解。 在同一个 add_argument 方法中，它们指向同一个目标（即同一个属性）。一般使用长选项去掉前面的两个短横线后的名字（即 verbose）作为属性名

在命令行上，用户可以使用以下任意一种形式： python main.py -v 或 python main.py --verbose 。无特殊设置的话，命令行上出现了可选参数，则其对应的参数的值设为 true，否则，设为 false。

如果只定义了一个选项（例如只定义-v 或只定义--verbose），那么用户就只能使用定义的那个选项。

另外，短选项可以合并使用，例如如果有多个短选项，可以合并在一起写：

# 组合多个短选项 (当多个选项都不需要额外参数时)

python script.py -avs  # 等同于 -a -v -s

**参数辅助信息**

1.help 帮助信息：在所有添加的参数后面可以使用 help 来描述参数的作用。运行程序时只需添加 -h 或 --help 参数即可，其它参数均不需要。

2.type：参数类型转换函数，指将命令行输入的字符串（默认所有参数值都是字符串）转换为程序所需的特定数据类型（如整数、浮点数、布尔值、文件对象等）的过程。当基础类型不能满足需求时，可以创建自定义转换函数，然后令type=自定义函数名。

3.metavar：在帮助文档中显示的参数名称。

4.dest="show_details"：可以使用 dest 参数显式指定属性名（赋别名）。

5.required：标记可选参数是否必需，true 为必需。一般用在可选参数，因为位置参数默认是必需的。

6 数量可变的参数：使用 nargs 参数可以指定参数的数量，传入的参数将被放到参数名称的列表里。

- nargs=?：0 个或 1 个参数。

- nargs=*：0 个或多个参数。

- nargs=+：1 个或多个参数。

- 整数：确切个数的参数（如 nargs=2 表示需要两个参数）。

7.action："store"和"store_true"，两者效果完全相同，都会将添加参数的值设置为 True。如果未指定，则存储一个 False 值（或者如果设置了`default`则存储默认值）。

| **action 类型** | **功能描述** | **常用参数** |
| --- | --- | --- |
| "store" | 存储参数值（默认，可省略） | type, default |
| "store_const" | 存储常量值 | const |
| "store_true" | 存储 True | - |
| "store_false" | 存储 False | - |
| "append" | 将值追加到列表 | - |
| "append_const" | 将常量追加到列表 | const |

| action 类型 | 功能描述 | 常用参数 |
| --- | --- | --- |
| "count" | 计算参数出现次数 | - |
| "help" | 显示帮助信息 | - |
| "version" | 显示版本信息 | version |
| "extend" | 扩展列表（Python 3.8+） | nargs |

8. choices：参数允许的值范围，如[1, 2, 3], range(1, 10)
9. default：参数默认值（当未提供时使用）

```
# 添加互斥参数
group = parser.add_mutually_exclusive_group()
group.add_argument(
    "--fast",
    action="store_true",
    help="快速模式（牺牲准确性）")
group.add_argument(
    "--accurate",
    action="store_true",
    help="精确模式（速度较慢）")
```

*获取参数,创建一个空 parser 对象，模拟读取命令行参数（实际读取 kawrgs 的参数）*

**配置文件**

在大型项目中，为了更好地维护和管理参数，通常会把关键的参数放置在配置文件里，常用的配置文件一般包括.py 文件和.json 文件，在开源项目里会有.example 文件指导用户给定初始的参数（为了避免在版本控制中暴露敏感信息（如数据库密码、API 密钥等），我们通常不会直接修改 config.py.example 文件，而是将其复制为 config.py，并在这个新文件中修改参数的值。

值得注意的是，若 configs 是个库，下面有多个配置文件，需要在库下面添加一个 __init__.py 文件，文件里导入各个配置文件，然后在应用文件里再直接

导入库 import configs。配置文件参数的优先级低于命令行参数，即命令行传递的参数是最终使用的参数。

.py 文件

**直接使用 Python 语法：**通过变量、字典或类定义配置。

**灵活性强：**支持动态逻辑（如条件判断、计算表达式）。

**无需解析器：**直接通过 import 导入配置，完全按照 Python 数据结构的正常方法使用。

**潜在安全隐患：**如果配置文件来自不可信来源，执行代码可能带来风险。

示例代码

*#字典*

DATABASE = {

"host": "localhost",

"port": 3306,

"username": "admin",

"password": "secret",

"timeout": 10}

#变量，一般全大写

DEBUG = True # *可动态控制配置项*

#使用方式

from config import DATABASE, DEBUG

print(DATABASE["host"]) # *输出: localhost*

print(DEBUG) # *输出: True*

可以看到，.py 文件将配置参数以字典，变量等数据形式存储，使用时直接 import 导入对应变量，然后就可以按照 Python 语法正常使用。

.json 配置文件

**结构化数据格式：**键值对存储，支持嵌套结构。

**跨语言兼容：**几乎所有编程语言都能解析。

**无代码执行：**纯数据文件，安全性高。

**严格语法：**不允许注释（原生 JSON），需注意格式正确性。

示例文件

```
{"database": {
"host": "localhost",
"port": 3306,
"username": "admin",
"password": "secret"},
"debug": true}
```

使用方式

```
import json
with open("config.json") as f:
config = json.load(f)
print(config["database"]["host"]) # 输出: localhost
print(config["debug"]) # 输出: True
```



**变化的函数**

**内部函数（嵌套函数）**

在 Python 中，内部函数（嵌套函数）可以直接访问外部函数的参数和变量，这种特性称为闭包（Closure）。闭包允许内部函数捕获并记住外部函数的环境，即使外部函数已经执行完毕。

**关键规则：**

1. **读取外部参数 / 变量：**内部函数可以直接读取外部函数的参数和变量。
2. **修改限制：**默认情况下，内部函数不能直接修改外部函数的不可变类型变量（如 int、str、tuple），但可以修改可变类型变量（如 list、dict）。

**强制修改：**如果需要修改外部函数的不可变变量，可以使用 nonlocal 关键字声明。

**装饰器函数**

我们知道，函数是封装了特定功能的代码块，前面也介绍了函数参数的具体传递，不过参数均是简单的变量。如果我们想对函数的功能进行扩展应该怎么做呢？直接修改函数代码必然不是一个好选择，python 提供了装饰器。

装饰器是 Python 中一个非常强大的功能，它允许我们修改或增强函数、方法或类的行为，而不需要改变其本身的代码。装饰器本质上是一个可调用对象（通常是一个函数），它接受一个函数作为参数，并返回一个新的函数。新函数包括 2 部分，核心部分是原函数的功能，第 2 部分就是新增的模式化的通用功能。

其基本结构如下：

```
#1 定义一个接收函数为参数的装饰器函数
def decorator_function(func):
    #2 定义一个包装函数（wrapper），参数使用*args,**kwargs，以适应各种参数的原函数，这里参数和原函数是对应的
    def wrapper(*args,**kwargs):
        语句 1   #3.在原函数前后增加一些通用的功能语句，也可以使用上面传递进来的参数
        result=func(*args,**kwargs)#4.调用原函数
        语句 2
        return result   #5 原函数有返回值则返回值，包装函数结束
    return wrapper     #最后返回扩展函数，装饰器函数结束
# 6 使用方式，@+装饰器函数名，下面跟着接收函数的定义，之后正常调用接收函数
@decorator_function
def func:
    pass
```

举个例子，在调用原函数前后添加了额外日志记录的功能。

```
def log_decorator(func):
    def wrapper(*args, **kwargs):
```

```
        for num in args:      #内部扩展函数的其它语句也可以使用
            print('num:',num)
        print(f"Before function {func.__name__} is called.")
        result = func(*args, **kwargs)
        print(f"After function {func.__name__} is called.")
        return result
    return wrapper

@log_decorator   # 接收函数为参数的函数名前面加@符号即为装饰器。
def add(a, b):     #原函数定义在下方
    return a + b
result = add(3, 5)
print(result)

# 带参数的装饰器需要三层嵌套
def require_role(role):
    # 外层接收装饰器参数
    def decorator(original_func):
        # 中间层接收函数
        def wrapper(*args, **kwargs):
            # 内层实现装饰逻辑
            if role == "admin":
                print("管理员权限验证通过")
                return original_func(*args, **kwargs)
            else:
                raise PermissionError("权限不足!")
        return wrapper
    return decorator
```

# 应用带参数的装饰器@require_role("admin")def delete_database():

print("数据库已删除!")

@require_role("user")def view_data():

print("显示数据...")

# 测试

delete_database()  # 成功执行 try:

view_data()  # 触发异常 except PermissionError as e:

print(f"错误: {e}")

总结一下，装饰器函数本质上是一个函数，其主要功能和作用体现在以下几个方面：

**1. 代码复用和模块化**

装饰器可以将一些通用的功能（如日志记录、权限验证、性能测试等）封装起来，然后应用到多个不同的函数上，避免了代码的重复编写，提高了代码的复用性和可维护性。

**2. 功能增强**

在不修改原函数代码的前提下，为原函数添加额外的功能。这符合软件开发中的开闭原则，即对扩展开放，对修改关闭。

**3. 分离关注点**

将与核心业务逻辑无关的功能（如日志、权限控制等）从业务逻辑中分离出来，使代码结构更加清晰，便于维护和管理。

注意事项-

装饰器会改变被装饰函数的元信息（如函数名、文档字符串等），可以使用`functools.wraps`来保留这些信息。

- 装饰器在模块导入时就会执行，因此装饰器内部的代码会在导入时运行。

使用 functools.wraps 保留原始函数的元信息，

import functools

def my_decorator(func):

@functools.wraps(func)

def wrapper(*args, **kwargs):

```
        print("装饰器中...")
        return func(*args, **kwargs)
    return wrapper
@my_decoratordef example():
    print("Hello")
print(example.__name__) # 输出 'example' 而不是 'wrapper'
print(example.__doc__) # 输出 '示例函数的文档字符串' 而不是 '包装函数的文档字符串'
```

**Iterable 和 iterator**

为了节省内存，即把数据一个一个的加载进来，Python 提供了可迭代对象 iterator 和迭代器 iterable2 个概念，可迭代对象就是内部实现了__iter__方法的类，是存储数据的容器，类似于一个空竹竿存了若干豆子；迭代器就是内部实现了__next__方法的类，是获取数据的指针，类似于一个钩子可以把一个豆子取出来，并指向下一个豆子。

terable 是个容器，可以提供 iterator，可以理解为完整的链表或者链表的表头；iterator 是一个可以指向下一个对象的对象，可以理解为链表的一个单元；两者结合弥补了 python 不能用指针的缺陷，非常适合在链表这种数据结构中使用；

具体到代码实现，每次调用内部函数 iter(iterator)，会返回一个迭代器 iterable，这个迭代器会记录当前容器的数据位置在哪里，也就是说返回一个钩子，然后便可以调用内部函数 next(iterable)就可以把指向的数据取出来，然后再指向下一个数据，如果容器里没有数据，再获取就要报错返回。

这就是最基本的可迭代对象和迭代器，从这个概念来看，可迭代对象才是真正可迭代的，存储数据的，而迭代器最初只是获取数据的方法而已（后面还会再完善，迭代器本身也是可迭代对象）。代码如下：

```
#最基础的可迭代对象和迭代器对象
class counter_iteratble():
    def __init__(self,start,end):
        self.start=start
        self.end=end
    def __iter__(self):
        return counter_iterator(self.start,self.end)
```

```
class counter_iterator():
    def __init__(self,current,end):
        self.current=current #记录当前位置
        self.end=end
    def __next__(self):
        if self.current<=self.end:
            value=self.current
            self.current+=1
            return value
        else:
            return StopIteration

if __name__=='__main__':
    counter_iteratble=counter_iteratble(1,3)
    counter_iterator=iter(counter_iteratble)
    print(next(counter_iterator))
    for iter_num in iterable1:     # 可迭代对象使用 for 循环遍历
      print('iter_num',iter_num)
```

（1）可迭代的数据是如何体现的？如上面的数值区间，也就表示可迭代对象区间为[start,end]，而迭代器的初始化是包括最末尾 end，和当前位置 current（注意这只是当前位置索引，用 start 来初始化，不用 0 是因为数值区间 start 可能不从 0 开始）；而可迭代数据类型包括很多种，如字符串，列表，元组等，此时

```
class ListIterable:
    def __init__(self, items):   # items 可以是任意数据集合
        self.items = items
    def __iter__(self):
        return ListIterator(self.items)
class ListIterator:
    def __init__(self, items):
```

```
            self.items = items
            self.index = 0
        def __next__(self):
            if self.index < len(self.items):
                value = self.items[self.index]  # 返回字符串/任意类型数据
                self.index += 1
                return value
            else:
                raise StopIteration  # 正确做法是抛出异常（非返回 ValueError）
    # 使用示例
    iterable = ListIterable(["A", "B", "C"])
    for item in iterable:
        print(item)  # 输出 A → B → C
```

（2）迭代器只能通过 iter（）调用实现吗？能否自己实例化？并不是，只是 for 循环自动调用 iter()来获取迭代器，然后不断调用 next()直到捕获 StopIteration 异常，迭代器本身也可实例化。

（3）iter（），next（）和类的方法 __iter__ 和 __next__有什么关系？上面的 iter()和 next()是 python 的内置函数，与类的特殊方法 __iter__ 和 __next__ 之间是接口与实现的关系，内置函数调用了类的方法，iter(obj) 等价于 obj.__iter__()，是显式地遍历可迭代对象，而 for 循环提供了隐式的遍历方法

（4）for 循环隐式遍历：拿列表，元组，字典，字符串等可迭代对象举例子，遍历取值时常常使用 for 循环，其内部也是经过了两步，以列表 list 为例：

for list in lists：

等价于：

（1）从可迭代对象获取迭代器 iter（lists）；

（2）使用 next 方法获取数据：

```
    while True:
        try:
            item = next(iterator)  # 调用 __next__
```

```
            # 执行循环体
        except StopIteration:
            break
```

值得注意的是，对于自定义的可迭代对象和迭代器对象，两次使用 for 循环遍历的话，这种正常的可迭代对象每次循环都会重新获取一个可迭代器，因为__iter__方法返回的是可迭代器的实例化对象。

迭代器也是可迭代对象

前面说迭代器仅仅是一个工具和获取数据的方法，本身并不能迭代，这有些违反直觉，因为直观上感觉有了一个钩子应该能顺藤摸瓜地把后面的豆子全都取出来，因此，Python 做了一个优化，即从任意一个迭代器也能迭代，也是一个可迭代对象，具体到代码就把迭代器类也加了 iter 方法，iter 下只需要返回 self 自身。相当于钩子返回了一个钩子，数据还是借着原来的容器存储的。

但是注意，这样只能遍历一次，因为遍历完后迭代器状态已经结束（索引值已经到了最后的数据）。而上面的分离方式，每次调用`__iter__`都会返回一个新的迭代器，因此可以多次遍历。

合并的代码（但注意：这样只能遍历一次）：`

```
class MyRange:
    def __init__(self, start, end):
        self.start = start
        self.end = end
        self.current = start
    def __iter__(self):
        return self # 返回自身，因为自身就是迭代器
    def __next__(self):
        if self.current < self.end:
            value = self.current
            self.current += 1
            return value
        else:
```

```
            raise StopIteration# 使用
my_range = MyRange(1,4)
for num in my_range:
    print(num) # 输出 1,2,3
# 再次遍历就不会有输出了，因为迭代器已经到头
print("再次遍历:")
for num in my_range: print(num) # 不会输出```
```

**生成器函数和生成器对象**

调用一般的函数或者完成执行某个任务，或者返回某个结果，函数执行完毕就结束。而调用生成器函数不会直接返回值，而是返回一个生成器对象保存到变量里，使用内置函数 next 时会逐步返回一个值，然后代码继续返回到生成器函数返回生成器对象的位置。

从上面的描述可以看出，生成器对象就是一种特殊的迭代器。生成器对象实现了迭代器协议（即__iter__()和__next__()方法）。 因此，生成器对象也是迭代器，同时也是可迭代对象（因为迭代器本身就是可迭代的）。

怎样创建生成器函数呢？

在 Python 里，yield 关键字用于创建生成器函数。生成器函数使用 yield 逐个返回值，每次调用生成器的 __next__() 方法时，函数会从上次 yield 语句的位置继续执行，直到遇到下一个 yield 语句或者函数结束。

下面是一个简单的例子，展示了 yield 的迭代方法：

```
def simple_generator():
    yield 1
    yield 2
    yield 3
    # 创建生成器对象
gen=simple_generator()
    #开始迭代
print(next(gen))
print(next(gen))
```

```
print(next(gen))
try:
    print(next(gen))
except
    StopIteration: print("生成器已经耗尽")
```

在这个例子中，simple_generator 是一个生成器函数，它使用 yield 关键字返回三个值。每次调用 next(gen) 时，函数会从上次 yield 的位置继续执行，直到遇到下一个 yield 语句或者函数结束。当生成器耗尽后，再次调用 next 会引发 StopIteration 异常。

同样地，生成器函数也可以使用 for 循环遍历：

```
for x in simple_generator():
    print(x)
```

每次的循环都会调用一次 next 内置函数，输出一个结果。

生成器函数遍历树结构对象

下面是一个使用 yield 遍历树结构对象的例子：

```
#tree yield
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []
    def add_child(self, child_node):
        self.children.append(child_node)
    def traverse(self):
    # 先返回当前节点的值
        yield self.value
    # 递归遍历子节点
        for child in self.children:
```

```
            yield from child.traverse()
# 创建树结构
root = TreeNode(1)
child1 = TreeNode(2)
child2 = TreeNode(3)
root.add_child(child1)
root.add_child(child2)
grandchild1 = TreeNode(4)
grandchild2 = TreeNode(5)
child1.add_child(grandchild1)
child1.add_child(grandchild2)
# 遍历树
for value in root.traverse():
    print(value)
```

代码功能说明

**TreeNode 类：**

__init__ 方法：初始化树节点，每个节点包含一个 value 属性和一个 children 列表，用于存储子节点。

add_child 方法：用于向当前节点添加子节点。

traverse 方法：这是一个生成器函数，用于遍历树结构。首先，使用 yield 返回当前节点的值。然后，使用 yield from 递归调用子节点的 traverse 方法，将子节点及其子树的所有值逐个返回。

**树结构的创建：**创建了一个简单的树结构，根节点的值为 1，有两个子节点 2 和 3，子节点 2 又有两个子节点 4 和 5。

**树的遍历：**使用 for 循环遍历根节点的 traverse 生成器，逐个打印树中所有节点的值。

通过这种方式，使用 yield 可以方便地实现树结构的遍历，并且避免了一次性将所有节点的值存储在内存中，提高了内存使用效率。

**enumerate（）函数**

enumerate() 函数是 Python 内置函数，用于将一个可迭代对象（如列表、元组、字符串、字典、集合、文件对象等作为参数）组合为一个索引序列（即枚举对象）。它返回的是一个迭代器（枚举对象），每次迭代返回一个元组，包含两个元素：索引（从 0 开始计数）和原可迭代对象对应索引的元素。

for i,j in enumerate（）函数将字符串，元组，列表等可迭代对象返回数据下标和对应数据的元组，用 for 循环中的 i，j 来接收。

1.惰性求值：与生成器类似，enumerate() 不会立即生成所有元组，而是在迭代过程中逐个产生，节省内存。

2. 可设置起始索引

list(enumerate(fruits, start=1)) # [(1, 'apple'), (2, 'banana'), (3, 'cherry')]

3. 与可迭代对象解耦：枚举对象独立于原始可迭代对象，对原始可迭代对象的修改不会影响枚举对象（因为枚举对象在创建时已经基于原始可迭代对象创建了自己的迭代器）

为什么 enumerate() 返回的是迭代器？

- 效率：不需要预先计算所有索引-值对，特别适合处理大型数据。

- 通用性：可以与任何可迭代对象配合工作，包括无限序列（如 `itertools.count()`）。

深度学习中经常用的地方是遍历读取数据集，推而广之，dataloader 作为一个可迭代对象，其数据下标是 batch（在创建 dataloader 时会把 batch size 作为参数传入），从 0 开始，最大数为样本总数除以 batch size 大小，下标对应的数据是一 batch 的数据 x 和标签 y（这里的 x 和 y 是列表）。比如总共有 320 个样本，batch_size 大小为 16，则是一个大小为 320/16=20 的列表，列表中的每个元素则是 16 个样本数据 x[batch_size, ...]和标签 y[batch_size, ...]。

for batch,(x,y) in enumerate(dataloader):batch 从 0 到 19

**try 和 except**

在 Python 中，try、except 和 finally 是异常处理机制的关键组成部分。

它们的基本功能原理是：尝试执行一段代码（try 块），如果这段代码中发生了异常，则根据异常类型执行相应的处理代码（except 块），最后无论是否发生异常，都会执行清理代码（finally 块）。

基本组成结构：

1. try 块：包含可能引发异常的代码。

```
try:
    risky_operation()
```

2. except 块：一个或多个，用于捕获并处理特定类型的异常。可以指定异常类型，也可以不指定（捕获所有异常）。永远不要使用空的 except:（它会捕获包括 KeyboardInterrupt 的所有异常），至少应记录异常：

```
except ExceptionType1:
    # 处理特定异常类型
    handle_exception1()
except ExceptionType2 as e:
    # 处理另一种异常（可获取异常对象）
    handle_exception2(e)
except:
    # 捕获所有未处理的异常（不推荐常规使用）
    handle_unknown()
```

3. else 块（可选）：当`try`块中的代码没有引发异常时执行。

```
else:
    # 当没有异常发生时执行（可选）
    safe_operation()
```

4. finally 块：无论是否发生异常，都会执行的代码块。常用于资源清理（如关闭文件、释放锁等）。

```
    finally:
        # 无论是否发生异常都会执行
        cleanup_resources()
```

再举个常用的文件读取的例子：

```
    file = None
    try:
        file = open('data.txt', 'r')
```

```
        content = file.read()
        print(content)
    except FileNotFoundError:
        print("文件不存在！")
    finally:
        if file:  # 检查文件对象是否存在
            file.close()  # 确保文件总是关闭
            print("文件已安全关闭")
```

**try-except 块的变量作用域问题**

Try 和 except 代码块的变量是否通用？在 Python 中，try 和 except 块共享**相同的作用域**，就像它们属于同一个代码块一样。这意味着：两块**变量是通用的**，在 try 块中创建的变量可以在 except 块中访问；在 except 块中创建的变量可以在 try 块之后使用；但如果 try 块中途发生异常，except 块及后面模块变量可能处于未定义状态

**关键注意事项：**

```
try:
    x = 10  # 定义在 try 块中
    y = 5 / 0  # 这里会引发异常
except ZeroDivisionError:
    print(x)  # 可以访问 x（因为异常发生在赋值之后）
    # print(y)  # 错误！y 未定义（因为赋值未完成）
    z = 20  # 在 except 块中定义新变量
print(z)  # 可以访问 z（值为 20）
```

**最佳实践：**

推荐：在 try 之前初始化可能需要的变量

```
result = None
try:
    data = load_data()  # 可能失败的操作
```

```
    result = process(data)   # 可能失败的后续操作 except DataLoadError:
        print("数据加载失败")
except ProcessingError:
        print("数据处理失败")
# 安全访问 result
if result is not None:
        use_result(result)
```

**Traceback 问题排查溯源**

有时候 except 捕获的异常比较笼统，不能精确确定出错的位置，这时可以使用 traceback 模块排查溯源。

异常种类有哪些？

**上下文管理与 with 语句**

上下文管理是一种用于管理资源（如文件、网络连接、锁等）的机制，它确保在代码块执行前后进行正确的资源分配和释放，即使代码块中发生了异常也是如此。 在 Python 中，上下文管理是通过 with 语句来实现的。

with 语句允许我们定义一个执行上下文，在该上下文中，我们可以确保资源的获取和释放是自动完成的。 即在 with 语句中，我们只需要关注资源或对象本身的操作（这点和普通方法一致），系统自动释放资源和对象（这点在普通方法中需要通过 try except 和 finally 实现，即 try 实现功能，except 处理异常，finally 释放资源。）

**上下文管理器的基本结构**

上下文管理的基本结构由两个特殊方法组成：__enter__和__exit__。这两个方法定义在上下文管理器对象中。

1. __enter__(self):

当进入 with 语句块时，会调用这个方法。该方法返回的资源（是类吗？）会被赋值给 as 后面的变量（如果有的话）。 通常在这个方法中获取资源（如打开文件）。

2. __exit__(self, exc_type, exc_value, traceback)`:

当离开 with`语句块时（无论是正常结束还是因为异常），都会调用这个方法。

参数 exc_type, exc_value, traceback 分别表示异常类型、异常值和调用栈。如果没有异常发生，这三个参数都是 None。 如果该方法返回 True，则表示异常已经被处理，不会继续向外传播；如果返回 False，则异常会被重新抛出（如果没有异常，返回什么都可以）。

通常在这个方法中释放资源（如关闭文件）。

**上下文管理器的种类**

除了使用类定义上下文管理器，Python 还提供了 contextmanager 装饰器（在 contextlib 模块中），可以快速创建基于生成器的上下文管理器。这种上下文管理器使用 yield 语句将函数分成两部分：yield 之前的部分相当于__enter__，之后的部分相当于__exit__。

**方式 1：类实现（完整控制）**

**特点**：显式定义__enter__和__exit__，拥有完整异常处理能力，支持状态管理

```
class FileManager:
    def __init__(self, filename):
        self.filename = filename
        self.file = None
    def __enter__(self):
        print("打开文件...")
        self.file = open(self.filename)
        return self.file   # 返回资源对象
    def __exit__(self, exc_type, exc_val, exc_tb):
        print("关闭文件...")
        if self.file:
            self.file.close()
        # 处理异常（返回 True 则阻止异常传播）
        if exc_type:
```

```
            print(f"发生异常: {exc_type}")
        return False  # 不屏蔽异常
# 使用示例
with FileManager("test.txt", "w") as f:
    f.write("Hello Context!")  # 即使发生异常，文件也会关闭
    # 触发异常测试: 1/0
print("操作完成")
```

f 是返回的资源名称（可任意取），f.write 是该资源自己拥有的方法。

**方式 2：生成器实现（简洁方案）**

**特点**：使用 contextlib.contextmanager 装饰器，用 yield 分隔资源获取/释放，代码更简洁

例 1：

```
from contextlib import contextmanager
@contextmanager
def file_manager(filename, mode):
    try:
        print("打开文件...")
        f = open(filename, mode)
        yield f  # 返回资源
    except Exception as e:
        print(f"发生异常: {type(e).__name__}")
    finally:
        print("关闭文件...")
        f.close()
# 使用示例
with file_manager("test.txt", "a") as f:
    f.write("\nAppend content")
    # 触发异常测试: int('abc')
```

```
# 传统方式（易出错）
file = open('data.txt')
try:
    data = file.read() # 若此处抛出异常，
    file.close() 可能不被执行！
finally:
    file.close() # 必须手动确保关闭
```

例 2：

```
from contextlib import contextmanager
@contextmanagerdef session_scope():
    # 创建新的数据库会话（相当于 __enter__ 方法）
    session = SessionLocal()
    try:
        # 将会话传递给 with 代码块使用
        yield session   # 暂停执行，将会话对象传递给 with 块
        # 代码块无异常时提交事务（yield 之后的部分相当于 __exit__ 中的正常退出逻辑）
        session.commit()
    except:
        # 发生异常时回滚事务
        session.rollback()
        # 重新抛出异常
        raise
    finally:
        # 始终关闭会话释放连接（相当于 __exit__ 中的清理逻辑）
        session.close()
```

**with 语句用法**

with 上下文管理器 as 变量:

代码块

# 等价于：

manager = 上下文管理器()

变量 = manager.__enter__()

try:

代码块

finally:

manager.__exit__(异常信息)

With 语句的格式是：with 类名（参数）/函数名（参数） as cls：

类名（参数）/函数名（参数）上面已经介绍了就是上下文管理器的 2 种方式，那么后面这个 as 有什么作用呢？

**as 关键字的使用场景**

| 情况 | 是否使用 as | 说明 |
| --- | --- | --- |
| 需要访问管理器返回值 | 必须使用 | 当 __enter__() 返回的值需要在代码块中使用时 |
| 仅需资源管理 | 可省略 | 不需要使用管理器返回值时 |

比如当需要直接操作上下文管理器返回的资源对象时：

# 文件操作：需要文件对象进行读写

with open('test.txt', 'w') as file:  # as 接收 open() 返回的文件对象

file.write("Hello, World!")  # 操作文件对象

# 退出 with 后文件自动关闭

当只需确保资源清理，不需要操作管理器返回值时，比如锁操作：

# 锁操作：只需确保锁被释放，无需直接操作锁对象

import threading

```
lock = threading.Lock()
with lock:  # 不需要 as，因为不需要操作锁对象
    print("临界区代码执行中")# 退出 with 后锁自动释放


class Timer:
    def __enter__(self):
        self.start = time.time()
    def __exit__(self, *args):
        elapsed = time.time() - self.start
        print(f"代码块耗时: {elapsed:.2f}秒")
with Timer():  # 不需要 as，因为不关心返回值
    time.sleep(1.5)  # 模拟耗时操作
```

**输出：**

```
代码块耗时: 1.50 秒
```

2. 异常处理更健壮

**问题解决：**在 __exit__() 中可统一处理异常（如回滚事务、日志记录）。

**原理：**__exit__() 接收异常信息，可通过返回 True 抑制异常。

```
class DatabaseTransaction:
    def __enter__(self):
        self.conn = sqlite3.connect('db.db')
        return self.conn.cursor()
    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type: # 发生异常时回滚
            self.conn.rollback()
        else: # 否则提交
            self.conn.commit()
            self.conn.close()
with DatabaseTransaction() as cursor:
```

cursor.execute("DELETE FROM users") # *若此处失败，自动回滚*

**Log 日志**

```
#logger.debug('create', create)
# 方式 1：添加占位符，打印变量值（推荐）
logger.debug('创建向量库参数: create=%s', create)  # 假设 create 是布尔值或其他变量
```

**是什么？有什么功能作用？**

Python 内置的 logging 模块来记录日志。它提供了灵活的事件日志系统，可以用于调试、信息记录、警告、错误等。简单来说，不再一个个手打 print 输出信息，而是直接把关键信息记录到日志里。而且会对信息进行分类，使用不同的方法如 warn()，info()记录不同等级的信息。

基本概念：

- 日志级别：（级别递增）有什么区别？如何定义级别？

  DEBUG：调试细节

  INFO：正常运行信息

  WARNING：潜在问题

  ERROR：功能错误

  CRITICAL：严重错误

**核心组件**

- 记录器（Logger）：记录日志的主入口，用于直接执行日志记录的对象。有哪些属性和方法？

- 处理器（Handler）：决定日志输出位置，将日志记录发送到不同的地方（如控制台、文件等）。在哪里定义，如何发送？

- 格式器（Formatter）：定义日志记录的格式。

  **Filter**：过滤日志（可选）

**工作流程：**

Logger → 过滤 → Handler → Formatter → 输出

**怎么用？**

使用步骤：

1. 创建记录器（Logger）并设置日志级别。

2. 创建处理器（Handler）并设置处理器的日志级别。

3. 创建格式器（Formatter）并设置到处理器上。

4. 将处理器添加到记录器。 注意：记录器和处理器都可以设置日志级别，只有不低于（即严重程度不低于）该级别的日志才会被处理。

**简单示例**

```
import logging


# 1. 创建记录器 Logger 实例（命名为 'my_app'）
logger = logging.getLogger('my_app')
logger.setLevel(logging.DEBUG)  # 设置最低处理级别（DEBUG 及以上都会处理）


# 2. 创建控制台处理器
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.WARNING)  # 控制台只输出 WARNING 及以上级别


# 3. 创建文件处理器
file_handler = logging.FileHandler('app.log', mode='w')  # 写入模式
file_handler.setLevel(logging.DEBUG)  # 文件记录所有 DEBUG 及以上级别


# 4. 创建格式化器
formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    datefmt='%Y-%m-%d %H:%M:%S'
```

)

# 5. 将格式化器添加到处理器

console_handler.setFormatter(formatter)

file_handler.setFormatter(formatter)

# 6. 将处理器添加到记录器 Logger

logger.addHandler(console_handler)

logger.addHandler(file_handler)

# ========== 测试日志输出 ==========

logger.debug('这是一条调试信息')  # 仅写入文件

logger.info('程序正常启动')  # 仅写入文件

logger.warning('磁盘空间不足 80%!')  # 同时输出到控制台和文件

logger.error('无法连接数据库!')  # 同时输出到控制台和文件

logger.critical('系统崩溃!!!')  # 同时输出到控制台和文件

输出结果

控制台输出（WARNING 及以上）：

2024-08-03 14:20:35 - my_app - WARNING - 磁盘空间不足 80%!

2024-08-03 14:20:35 - my_app - ERROR - 无法连接数据库!

2024-08-03 14:20:35 - my_app - CRITICAL - 系统崩溃!!!

文件输出（app.log 内容）：

2024-08-03 14:20:35 - my_app - DEBUG - 这是一条调试信息

2024-08-03 14:20:35 - my_app - INFO - 程序正常启动

2024-08-03 14:20:35 - my_app - WARNING - 磁盘空间不足 80%!

2024-08-03 14:20:35 - my_app - ERROR - 无法连接数据库!

2024-08-03 14:20:35 - my_app - CRITICAL - 系统崩溃!!!

**常用函数**

==None,is None

**copy 与 deepcopy**

在 Python 中，copy()和 deepcopy()都是用于创建对象副本的函数，核心差别在于创建的副本和原来对象之间的关系，即改变副本影不影响原来对象。

浅拷贝（copy()）：只拷贝父对象，不会拷贝对象的内部的子对象。拷贝后的对象与原对象共享子对象，所以如果原始对象中的子对象是可变的，改变它将会影响拷贝后的对象。

深拷贝（deepcopy()）：拷贝父对象和所有子对象。因此，深拷贝后的对象与原始对象没有任何关联。

使用 copy 模块中的 copy()和 deepcopy()函数。需要先导入 copy 模块。

```
import copy
# 原始列表
original = [1, 2, [3, 4]]
print("原始列表:", original)
# 浅拷贝
shallow = copy.copy(original)
shallow[0] = 10  # 修改第一层元素
shallow[2][0] = 30  # 修改第二层元素
print("浅拷贝后修改:")
print("原始列表:", original)  # [1, 2, [30, 4]] - 内部子对象列表被修改，数值 1 不变
print("浅拷贝列表:", shallow)  # [10, 2, [30, 4]]-
original = [1, 2, [3, 4]]# 重置
# 深拷贝
deep = copy.deepcopy(original)
```

```
deep[0] = 10   # 修改第一层元素
deep[2][0] = 30   # 修改第二层元素
print("\n 深拷贝后修改:")
print("原始列表:", original)   # [1, 2, [3, 4]] - 完全不受影响
print("深拷贝列表:", deep)   # [10, 2, [30, 4]]
```

**海象运算符-冒号等号（:=）**

冒号等号（:=）是 Python 3.8 版本引入的“海象运算符”，用于将表达式的值赋给变量并返回该值。它可以在条件表达式、循环条件和列表解析中使用，

elif worker_class := kwargs.get("worker_class")

使用海象运算符 (:=) 实现 **条件判断与赋值同时进行**

语义等价于：

worker_class = kwargs.get("worker_class")if worker_class is not None:

...

**getattr 和 setattr**

getattr 和 setattr 是 Python 内置函数，用于动态地获取和设置对象的属性。它们允许在运行时通过字符串来操作属性。

这 2 个函数是在代码运行过程中，手动地获取和设置类的属性，为什么不直接通过类名.属性操作呢？其中一个可能的原因是：比如通过 config 批量传递字典参数时，便可以通过 setattr 批量设置类的属性。

实用场景示例

场景 1：动态配置对象

class Settings:

pass

*# 从配置文件读取的配置项*

config_data = {

"theme": "dark",

"font_size": 14,

"auto_save": True}

settings = Settings()

# *动态设置属性* for key, value in config_data.items():

setattr(settings, key, value)

# *验证设置* print(settings.theme) # *输出: dark*print(settings.font_size) # *输出: 14*print(settings.auto_save) # *输出: True*

场景 2：安全访问嵌套属性

class Profile:

def __init__(self):

self.contact = Contact()

class Contact:

def __init__(self):

self.email = "user@example.com"

user_profile = Profile()

# *安全获取嵌套属性* def safe_getattr(obj, path, default=None):

parts = path.split('.')

for part in parts:

if hasattr(obj, part):

obj = getattr(obj, part)

else:

return default

return obj

# *正常访问*

email = safe_getattr(user_profile, "contact.email")print(f"Email: {email}") # *输出: Email: user@example.com*

# *访问不存在的路径*

invalid = safe_getattr(user_profile, "contact.phone", "N/A")print(f"Phone: {invalid}")
# *输出: Phone: N/A*

1. getattr 函数

**功能**：动态获取对象的属性值 **语法**：getattr(object, attribute_name[, default])

object：目标对象

attribute_name：属性名字符串

default：可选，属性不存在时返回的默认值

```
class User:
def __init__(self, name, age):
self.name = name
self.age = age
# 创建对象
user = User("Alice", 30)
# 获取存在的属性
name: str = getattr(user, "name") # 等效于 user.name
print(f"Name: {name}") # 输出: Name: Alice
# 获取不存在的属性（提供默认值）
gender: str = getattr(user, "gender", "Unknown")
print(f"Gender: {gender}") # 输出: Gender: Unknown
# 获取不存在的属性（无默认值）
try:
getattr(user, "address") # 触发 AttributeError
except AttributeError as e:
print(f"Error: {e}") # 输出: Error: 'User' object has no attribute 'address'
```

2. setattr 函数

**功能**：动态设置对象的属性值 **语法**：setattr(object, attribute_name, value)

object：目标对象

attribute_name：属性名字符串

value：要设置的值

```
class Config:
pass
```

```
# 创建空对象
config = Config()
# 设置新属性
setattr(config, "timeout", 10) # 等效于 config.timeout = 10
print(f"Timeout: {config.timeout}") # 输出: Timeout: 10
# 修改现有属性
setattr(config, "timeout", 20)
print(f"Updated Timeout: {config.timeout}") # 输出: Updated Timeout: 20
# 动态添加方法
def log_message(self, msg):
return f"LOG: {msg}"
setattr(Config, "log", log_message) # 添加到类（所有实例共享）
print(config.log("Test")) # 输出: LOG: Test
```

Sys

**isinstance()**

isinstance() 函数来判断一个对象是否是一个已知的类型，类似 type()。

isinstance() 与 type() 区别：

- type() 不会认为子类是一种父类类型，不考虑继承关系。
- isinstance() 会认为子类是一种父类类型，考虑继承关系。

如果要判断两个类型是否相同推荐使用 isinstance()。

isinstance(object, classinfo)

**参数**

- object -- 实例对象。
- classinfo -- 可以是直接或间接类名、基本类型或者由它们组成的元组。

**返回值**

如果对象的类型与参数二的类型（classinfo）相同则返回 True，否则返回 False。

**Zip**

把两个可迭代对象按照对应顺序组成元组，然后返回一个可迭代对象。

在 Python 中，zip() 函数用于将多个可迭代对象（如列表、元组等）中的元素**按位置配对**，生成一个元组构成的迭代器（逐次生成数据，如需直接查看结果需用 list() 转换）。如果两个对象元素数量不相同，按照少的数量匹配，多余的会被忽略。语法为：

zip(iterable1, iterable2, ...)

假设 _splits 是一个列表，zip(_splits[0::2], _splits[1::2])：

**切片提取元素得到 2 个列表**：

_splits[0::2] → 取索引为 0, 2, 4, ... 的元素（偶数索引）

_splits[1::2] → 取索引为 1, 3, 5, ... 的元素（奇数索引）

**配对元素**：

zip() 将两个切片列表按顺序配对，形成 (第 0 个元素, 第 1 个元素), (第 2 个元素, 第 3 个元素), ...

示例 1：列表长度为偶数

```
_splits = [10, 20, 30, 40, 50, 60]
# 切片结果：
#     _splits[0::2] = [10, 30, 50]   ← 偶数索引
#     _splits[1::2] = [20, 40, 60]   ← 奇数索引
result = list(zip(_splits[0::2], _splits[1::2]))print(result)   # 输出: [(10, 20), (30, 40), (50, 60)]
```

示例 2：列表长度为奇数（多余元素被忽略）

```
_splits = [10, 20, 30, 40, 50]
# 切片结果：
#     _splits[0::2] = [10, 30, 50]   ← 3 个元素
#     _splits[1::2] = [20, 40]       ← 2 个元素

result = list(zip(_splits[0::2], _splits[1::2]))print(result)   # 输出: [(10, 20), (30, 40)]（50 被忽略）
```

这种操作常用于：

**将相邻元素配对：**

```
nums = [1, 2, 3, 4, 5, 6]
pairs = list(zip(nums[0::2], nums[1::2]))# 结果: [(1, 2), (3, 4), (5, 6)]
```

**构建键值对：**

```
keys = ["a", "b", "c"]
values = [1, 2, 3]
dict(zip(keys, values))  # → {'a': 1, 'b': 2, 'c': 3}
```

在 rag 中，可以将返回的距离和 id 绑定元组同时返回。

```
for i, (dist, idx) in enumerate(zip(distances[0], indices[0])):
```