

第 3 章 文本与文档处理

本章将详细介绍如何运用 Python 执行文档读取、编写和修改，文件夹遍历和文件压缩等操作，以及常见的异常处理方法。本章内容侧重于实战应用，让读者学会如何进行问题需求分析，如何拆解问题并进行逻辑梳理，通过文字或流程图等方式清晰表达程序算法。最后利用 ChatGPT 协助进行代码调试与优化，通过实战项目加深读者对文本和文档处理方法的

理解。

本章涉及的主要知识点如下：

- ❑ 文件系统应用：熟悉文件系统的基本属性和操作。
- ❑ 程序算法的表达：使用自然语言和流程图表达程序算法。
- ❑ 提升 Python 编程技巧：熟悉 Python 文档处理相关的代码模块和自定义工具包。
- ❑ 学习 ChatGPT 的提示语：借助 ChatGPT 辅助学习新的模块，提升代码质量等。

3.1 文档读写操作

Python 配备了丰富的内置函数与模块，以便于处理文件的读写操作。本节着重介绍 Python 在文件读写方面的应用，并通过若干实例帮助读者深入理解这些功能。

3.1.1 增、删、改操作

1. 打开文件并读取文件内容

(1) 打开文件：使用 `open()` 函数打开一个文件，并返回一个文件对象。

(2) 操作模式：使用不同的操作模式来指定文件的读写方式，如读取模式（'r'）、写入模式（'w'）和追加模式（'a'）等。

(3) 读取文件内容：使用文件对象的 `read()` 方法读取整个文件的内容，或使用 `readline()` 方法逐行读取文件。

(4) 关闭文件：使用文件对象的 `close()` 方法关闭文件，以释放系统资源。

```
file = open('example.txt', 'r')           # r 为读取模式
content = file.read()
print(content)
file.close()
```

2. 逐行读取文件内容

使用 `readlines()` 方法每次返回一行文本内容，可以使用循环方式读取返回的内容。

```
file = open('example.txt', 'r')
for line in file.readlines():
    print(line)
file.close()
```

3. 写内容到文件中

使用文件对象的 `write()` 方法把一个字符串写到文件里, 或使用 `writelines()` 方法写入内容。

```
file = open('example.txt', 'w') # w 为写入模式
file.write('Hello, world!')
file.close()
```

4. 追加内容到文件末尾

如果不要覆盖原来的内容, 可以选择“a”追加模式, 这样新的内容会添加到文本末尾。

```
file = open('example.txt', 'a') # a 为追加模式
file.write('\nThis is a new line.')
file.close()
```

下面是一个完整的例子, 演示如何使用 Python 读取和写入文件。

```
# 程序 3.1: 打开文件并写入内容
file = open("example.txt", "w")
file.write("Hello, world!")
file.close()
print("文件已写入")
# 打开文件并读取其中的内容
file = open("example.txt", "r")
content = file.read()
# 输出文档内容
print(content)
file.close()
```

在上面的例子中, 首先使用 `open()` 函数打开名为 `example.txt` 的文件, 并使用读取模式 (`r`) 读取其中的内容。然后使用 `read()` 函数读取整个文件的内容, 并将其存储在变量 `content` 中。最后使用 `print()` 函数打印文件的内容。接下来, 再次打开相同的文件, 但这次使用写入模式 (`w`) 写入新的内容。使用 `write()` 函数将字符串 "Hello, world!" 写入文件中, 然后关闭文件。

5. 删除文件

要删除一个文件, 可以使用 `os` 模块中的 `remove()` 函数。下面是删除文件的示例代码:

```
import os
file_path = '/path/to/file.txt' # 替换为实际的文件路径
os.remove(file_path)
```

6. 重命名文件

要重命名一个文件, 可以使用 `os` 模块中的 `rename()` 函数。下面是重命名文件的示例代码:

```
import os
old_file_path = '/path/to/old_file.txt' # 替换为实际的旧文件路径
new_file_path = '/path/to/new_file.txt' # 替换为实际的新文件路径
os.rename(old_file_path, new_file_path)
```

3.1.2 常见的异常处理

以上示例仅为演示目的，并未包含错误处理和最佳实践。在实际应用中，应该考虑异常处理、上下文管理器等方面的细节，以确保代码的健壮性和可靠性。当读取文件时，需要考虑一些常见的异常情况。使用 Python 的 `with` 语句可以确保文件正确关闭，并且能够更好地处理异常情况。以下是一些常见的文件读取异常的处理。

1. 文件不存在异常

在尝试打开文件时，如果文件不存在，则会引发此异常。通过捕获该异常，可以提供友好的错误提示信息。

```
try:
    with open('example.txt', 'r') as file:
        # 进行文件读取操作
        content = file.read()
except FileNotFoundError:
    print("文件不存在，请检查文件路径是否正确。")
```

2. 读取错误

当读取文件出现问题时，如权限不足或磁盘空间已满等情况，会引发 `IOError` 异常。可以通过捕获该异常来处理读取错误。

```
try:
    with open('example.txt', 'r') as file:
        # 进行文件读取操作
        content = file.read()
except IOError:
    print("读取文件时发生错误，请检查文件是否可读或磁盘空间是否充足。")
```

有时候要对文件内容进行解析，如将文本文件中的数字转换为整数。如果文件中包含无法解析的内容，则会引发文件内容解析异常。

```
try:
    with open('example.txt', 'r') as file:
        for line in file:
            try:
                value = int(line)
                # 对解析后的值进行操作
            except ValueError:
                print("文件中包含无效的内容，请检查文件格式。")
except FileNotFoundError:
    print("文件不存在，请检查文件路径是否正确。")
```

通过使用 `with` 语句，确保在文件读取完成后自动关闭文件，即使在出现异常的情况下也能正常工作。同时，通过适当捕获和处理异常，程序可以提供有用的错误信息，并避免程序崩溃。

3. 编码转换问题

有时需要将文本从一个编码转换为另一个编码。例如，从 `UTF-8` 转换为 `GBK`。这通常发生在不同的系统或应用需要共享文本的场景。使用 `codecs` 模块可以完成编码转换。

```
import codecs
with codecs.open('source_file_utf8.txt', 'r', encoding='utf-8') as source:
    content = source.read()
with codecs.open('target_file_gbk.txt', 'w', encoding='gbk') as target:
    target.write(content)
```

上面的示例并未涵盖所有可能的异常状况。在实际应用中，根据具体需求和场景，有必要考虑到更多的异常和错误状况。后续将通过实际案例，依据实际情况对异常处理等细节进行深入探讨。

3.1.3 如何利用 ChatGPT 优化代码

若编程经验不足，可以寻求 ChatGPT 的帮助，优化代码，增加代码的安全性和健壮性。

问题 1:

请帮忙优化代码，代码如下：

```
file = open("example.txt", "w")
file.write("Hello, world!")
file.close()
print("文件已写入")
# 打开文件并读取其中的内容
file = open("example.txt", "r")
content = file.read()
# 输出文档内容
print(content)
```

ChatGPT 回答 1:

```
def demol():
    """
    完整的例子，演示了如何使用 Python 读取和写入文件
    """
    try:
        # 打开文件并写入内容
        with open("example.txt", "w") as file:
            file.write("Hello, world!")
        print("文件已写入")
        # 打开文件并读取其中的内容
        with open("example.txt", "r") as file:
            content = file.read()
        # 输出文档内容
        print(content)
    except FileNotFoundError:
        print("文件未找到")
    except IOError:
        print("文件读写错误")
```

从回答中可以看到，ChatGPT 引入了 2.10 小节所学的异常处理机制，代码得以捕获潜在的异常错误，并通过使用文件上下文管理器（with）来确保文件得以正确关闭。

3.2 文件夹操作

文件夹的增、删操作是日常工作中经常需要处理的任务。使用 Python 编程语言，可以

轻松地实现对文件夹的创建、删除和重命名等操作。本节将介绍如何使用 Python 执行这些操作并提供示例代码。

3.2.1 文件夹基本操作

1. 创建文件夹

创建一个新的文件夹，可以使用 `os` 模块中的 `mkdir()` 函数。下面是创建文件夹的示例代码：

```
import os
folder_path = '/path/to/new_folder'          # 替换为实际的文件夹路径
os.mkdir(folder_path)
```

2. 删除文件夹

要删除一个文件夹，可以使用 `os` 模块中的 `rmdir()` 函数。请确保文件夹为空，否则无法成功删除。下面是删除文件夹的示例代码：

```
import os
folder_path = '/path/to/folder'            # 替换为实际的文件夹路径
os.rmdir(folder_path)
```

3. 重命名文件夹

要重命名一个文件夹，可以使用 `os` 模块中的 `rename()` 函数。下面是重命名文件夹的示例代码：

```
import os
old_folder_path = '/path/to/old_folder'    # 替换为实际的旧文件夹路径
new_folder_path = '/path/to/new_folder'    # 替换为实际的新文件夹路径
os.rename(old_folder_path, new_folder_path)
```

 **注意：**在执行文件夹的删除操作时，务必小心谨慎，并确保真正想要删除或更改名称的对象。

通过使用几句 Python 代码，便能够轻松地进行文件夹的增、删操作。这些功能在处理文件系统中的数据、批量处理文件等任务中非常有用。根据具体需要，可以进一步扩展这些示例，以满足个性化需求。

3.2.2 文件路径操作

1. 获取当前文件目录

利用 `os.path.dirname()` 方法可以获取对应文件的目录地址。

```
file_path = "/home/user/documents/example.txt"
directory = os.path.dirname(file_path)
print(directory)          # 输出: /home/user/documents
```

2. 拼接文件路径

当使用 `open()` 函数打开文件时，路径是一个关键参数。如果路径中包含非 ASCII 字符（如中文），则部分系统会遇到问题。建议在不了解系统的情况下，最好使用原始字符串（在字符串前加 `r`）或者使用纯英文名称，以避免路径中的转义字符问题。

```
with open(r'路径\到\中文文档.txt', 'r') as file:
    content = file.read()
```

对于 Windows 和 Linux 中的路径问题，Windows 使用反斜杠（`\`）作为路径分隔符，而 Linux 使用正斜杠（`/`）作为路径分隔符。当代码需要在不同的操作系统上运行时，这可能会成为一个问题。为了避免这种系统兼容性问题，可以使用 `os.path.join()` 来编写支持跨平台的代码。

```
import os
path = os.path.join('路径', '到', '中文文档.txt')
with open(path, 'r') as file:
    content = file.read()
```

总之，当使用 Python 读取中文文档时，正确处理编码、编码转换和路径是关键。此外，考虑到代码的跨平台兼容性，使用 `os.path.join()` 来构建路径也是一个良好的习惯。

3. 获取文件名

利用 `os.path.basename()` 方法可以获取文件名并去掉目录路径。

```
file_path = "/home/user/documents/example.txt"
filename = os.path.basename(file_path)
print(filename) # 输出: example.txt
```

3.2.3 批量文件压缩

文件压缩是在日常工作和数据处理中常见的任务。通过压缩文件，可以减小文件大小，节省存储空间，并方便文件传输和共享。Python 提供了多种库和模块来实现文件的压缩和解压缩操作。下面介绍如何使用 Python 进行文件压缩。

1. 使用 zipfile 库进行压缩

Python 的 `zipfile` 库提供了对 ZIP 文件的创建、读取和修改等功能。下面是使用 `zipfile` 库进行文件压缩的示例代码：

```
import zipfile

def compress_file(file_paths, zip_name):
    with zipfile.ZipFile(zip_name, 'w') as zipf:
        for file in file_paths:
            # 通过循环把每个文件写到压缩文件中
            zipf.write(file)
```

在上述代码中，`compress_file()` 函数接收两个参数：`file_paths` 是要压缩的文件路径列表；`zip_name` 是要生成的 ZIP 文件名。通过调用 `write()` 方法，逐个将文件添加到 ZIP 文件中。

2. 使用shutil库进行压缩

除了 zipfile 库外，Python 的 shutil 库也提供了方便的文件压缩和归档功能。shutil 库支持多种压缩格式，包括 ZIP、TAR 和 GZ 等。下面是使用 shutil 库进行文件压缩的示例代码：

```
import shutil

def compress_file(file_dir_path, zip_name):
    shutil.make_archive(zip_name, 'zip', *file_dir_path)
```

compress_file()函数同样接收两个参数：file_dir_path 是要压缩的文件夹路径；zip_name 是要生成的 ZIP 文件名。通过调用 make_archive()函数，可以指定压缩文件格式为 ZIP，并将多个文件归档到一个 ZIP 文件中。

3. 解压缩文件

要解压缩一个 ZIP 文件，可以使用 zipfile 库或 shutil 库中的相应方法。以下是使用 zipfile 库进行解压缩的示例代码：

```
import zipfile

def extract_zip(zip_file, extract_path):
    with zipfile.ZipFile(zip_file, 'r') as zipf:
        zipf.extractall(extract_path)
```

extract_zip()函数接收两个参数：zip_file 是要解压缩的 ZIP 文件路径；extract_path 是要提取文件的目标路径。通过调用 extractall()方法，可以将 ZIP 文件中的所有内容提取到指定的目标路径中。

还有一种方式是使用 shutil 库中的 unpack_archive()函数解压缩文件，其用法与上述示例类似。

```
import shutil

# 要解压缩的文件路径
archive_path = 'path/to/archive.zip'
# 解压缩后保存的目标文件夹路径
target_folder = 'path/to/target_folder'
# 调用 unpack_archive() 函数解压缩文件
shutil.unpack_archive(archive_path, target_folder)
```

通过上述 Python 代码，可以轻松地实现文件的压缩和解压缩操作。读者可以根据具体需求，选择适合的库和方法来处理不同的压缩格式。

 **注意：**压缩函数会根据压缩文件的扩展名自动推断压缩格式。如果无法识别压缩格式或不支持该格式，则会引发异常。在使用该函数之前，请确保已安装好相应的解压缩工具，例如对于 ZIP 文件，需要确保系统中有可用的 unzip 工具。

3.3 实战：反馈意见统计

在日常工作和生活中，我们时常需处理诸多的文件，如文本文件、电子表格或 PDF 等。

例如，小明接到了一项任务，需整理一个包含众多读者来信的文件夹，这些来信均为电子邮件附件，每封文本皆可视为一位读者的反馈。下周的部门会议需要回顾上季度的杂志销售状况，领导希望能拿到这些读者反馈的数据。基于该项任务，小明需要统计共计收到多少读者反馈以及反馈意见的总字数。面对文件夹中的众多文件，小明感到无所适从，毕竟逐一打开文件、记录字数并求和，可能耗时一周亦难以完成。

3.3.1 问题需求分析

在面对重复且有规律的工作时，小明想到可以运用 Python 编程语言编写一个脚本程序来协助完成这项任务。在开始编写代码之前，务必先将思路梳理清晰，将大任务分解成小任务，并用文字或图表的形式将其记录下来。对于初学者而言，这样做有助于将代码拆分成短小且易于编写的小段落。正如在学习写作的初期，老师会让学生先写一个作文大纲。

在处理问题时，可以根据个人偏好选择使用文字描述、思维导图或流程图等方法，对问题进行分析与拆解。针对统计反馈意见包含的数据的任务，对其需求进行深入剖析，要求程序对指定文件夹内的文本进行详细统计，从而精确获取读者反馈的总数量及总字数，实现步骤拆分如下：

- (1) 打开指定的文件夹并获取文件夹中所有的文件名。
- (2) 遍历文件夹中的每个文件。
- (3) 打开每个文本文件并读取其内容。
- (4) 统计读者反馈的数量，每个文本文件都代表一封读者来信，所以每遍历一个文本文件，反馈数量就加 1。
- (5) 统计总字数，将每个文本文件的字数累积到总字数中。
- (6) 关闭所有打开的文件。
- (7) 输出读者反馈的总数和总字数。

3.3.2 代码编写

1. 导入所需的模块

根据前面的需求分析，第一步是打开指定的文件夹并且只处理文本文档。因此需要导入 Python 内置的 `os` 和 `glob` 模块实现文件操作和文档读取。

```
import os
import glob
```

2. 定义遍历文件夹的函数

定义一个函数遍历指定文件夹中的全部文档文件，并返回文件路径列表。

```
def get_documents(folder_path):
    documents = []
    for file in glob.glob(os.path.join(folder_path, '*')):
        if os.path.isfile(file):
            documents.append(file)
    return documents
```

如果遇到不认识的库，记得要做什么吗？当然是咨询无所不知的 ChatGPT。

问题 1:

请简单介绍 Python 中的 glob 的用法

ChatGPT 回答 1:

glob 是 Python 标准库中的一个模块，用于处理文件和目录的通配符匹配。它可以根据指定的规则搜索匹配的文件路径并返回一个列表。

glob 模块支持简单的通配符匹配规则，通配符可以用于文件名的任何部分。下面是一些常用的通配符用法：

*: 匹配任意长度的任意字符，可以匹配空字符串。

?: 匹配任意单个字符。

[characters]: 匹配括号中的任何一个字符，可以使用连字符表示范围。例如，[0-9] 匹配任意数字。

[!characters]: 匹配不在括号中的任意一个字符。

根据上面的规则，如果都是 txt 文档，那么可以这样写：

```
# 获取当前目录下所有以.txt 结尾的文件名
text_files = glob.glob('*.txt')
print(text_files)
```

3. 读取文档内容并提取信息

编写代码读取每个文档的内容，并提取所需的信息进行汇总。

```
def process_documents(documents):
    summary = {}
    for document in documents:
        with open(document, 'r') as file:
            content = file.read()
            # 根据具体需求，提取信息并进行汇总
            # 例如，可以统计文档中的单词数量和行数等
            word_count = len(content) # 中文情况
            line_count = content.count('\n') + 1
            summary[document] = {'word_count': word_count, 'line_count':
line_count}
    return summary
```

📌注意：上面的代码是统计字符的个数，这种方法适合以中文为主的内容，若是英文内容为主，建议对字符以空格进行拆分，因为一个英文单词包含多个字符。

4. 执行脚本并输出结果

调用定义的函数执行脚本并输出结果。

```
if __name__ == '__main__':
    folder_path = '/path/to/folder' # 替换为实际的文件夹路径
    documents = get_documents(folder_path)
    summary = process_documents(documents)
    total_summary = {'word_count': 0, 'line_count': 0}
    for document, info in summary.items():
        print("文件: ", document)
        print("单词数量: ", info['word_count'])
        print("行数: ", info['line_count'])
        print("-" * 20)
        # 汇总
        total_summary["word_count"] += info["word_count"]
```

```
line_count["line_count"] += info["line_count"]
```

请将`/path/to/folder` 替换为实际的文件夹路径。通过执行以上代码，能够快速遍历指定文件夹中的全部文档，读取其内容并提取所需的信息，结果如下：

```
文件： 3-2-目录\读者来信 1.txt
字数： 87
行数： 3
-----
文件： 3-2-目录\读者来信 2.txt
字数： 66
行数： 1
-----
文件： 3-2-目录\读者来信 3.txt
字数： 67
行数： 2
-----
文件： 3-2-目录\读者来信 4.docx
字数： 7425
行数： 82
-----
文件： 3-2-目录\读者来信 5.docx
字数： 7433
行数： 85
-----
文件： 3-2-目录\读者来信 6.docx
字数： 7389
行数： 77
-----
字数总计： 22467
行数总计： 22467
```

有这样的脚本帮忙，便能高效地处理大批量文档，节省时间与精力。在实际应用过程中，需要根据具体需求进行相应的调整与扩展。根据不同文件类型选择合适的库来读取文件内容，并根据实际情况添加更多的信息提取与处理功能。借助这个简洁的 Python 程序，仅需 10min 即可完成所有文档的统计工作，从而提高工作效率。

3.3.3 调试与优化

1. 字符编码

处理中文文档时很容易就会出现编码问题，若看到类似如下代码错误提示，就属于编码问题，可询问 ChatGPT，让它帮助解决问题。

```
UnicodeDecodeError: 'gbk' codec can't decode byte 0xaf in position 28:
illegal multibyte sequence
```

当然，在前面介绍异常处理时提过，只需要使用正确的编码读取文档就可以解决。尝试以下几种方法：

- ❑ 更改编码方式：如果知道文件或数据的确切编码，确保使用正确的编码方式来读取或解码。例如，数据是 UTF-8 编码类型，就应该使用 `'utf-8'` 来解码。
- ❑ 忽略非法字符：如果不关心某些非法字符，则可以在解码时忽略它们，这可以通

通过在 `open()` 函数中使用 `errors='ignore'` 参数来实现。

```
# 更改编码方式与忽略非法字符
with open('your_file.txt', 'r', encoding='gbk', errors='ignore') as f:
    content = f.read()
```

- ❑ 替换非法字符：如果想替换非法字符而不是完全忽略它们，那么可以使用 `errors='replace'` 参数，使用一个替代符号（通常是问号）替换所有无法解码的字符。

```
with open('your_file.txt', 'r', encoding='gbk', errors='replace') as f:
    content = f.read()
```

2. 不同类型的文档分开处理

现已完成字数和行数的统计，不妨稍作休息，品尝一杯咖啡，随后来看下屏幕上的计算结果。值得注意的是，Word 文档统计出的字数明显较 txt 文档多出百倍，这个结果显然不合理。为此，打开文档进行核实，发现信件内容仅包含数行文字，字数绝不会超过百字。接下来先对 txt 类型的文档进行统计，针对 Word 文档则需探索其他处理方式。如果没有头绪，可咨询 ChatGPT，看能否协助准确统计 Word 文档数据。以下代码只处理 txt 文档。

```
def get_documents(folder_path):
    documents = []
    # 只处理 txt 文档
    for file in glob.glob(os.path.join(folder_path, '*.txt')):
        if os.path.isfile(file):
            documents.append(file)
    return documents
```

ChatGPT 给出的提示是使用 `python-docx` 库遍历文件夹中的所有 “.doc” 文档，并提醒这个库不是标准库，需要先安装环境。

```
pip install python-docx
```

然后给出实例如下：

```
def count_doc_info(folder_path):
    # 创建存储结果的列表
    doc_info = []
    # 遍历文件夹中的所有文件
    for root, dirs, files in os.walk(folder_path):
        for file in files:
            if file.endswith(".doc") or file.endswith(".docx"):
                # 获取文档的绝对路径
                doc_path = os.path.join(root, file)
                # 打开文档
                document = Document(doc_path)
                # 统计文本字数
                text = ""
                for paragraph in document.paragraphs:
                    text += paragraph.text
                word_count = len(text.split())
                # 统计行数
                line_count = len(document.paragraphs)
                # 将结果添加到列表中
                doc_info.append({"Filename": file, "Word Count": word_count,
                                "Line Count": line_count})
    return doc_info
```

按照这个例子，我们可以升级自己的代码，完美处理 txt 和 Word 文档。部分代码如下：

```
def get_documents(folder_path):
    txt_documents = [] # 保存 txt 文档路径
    doc_documents = [] # 保存 Word 文档路径
    for file in glob.glob(os.path.join(folder_path, '*')):
        if os.path.isfile(file): # 只处理文件而非文件夹
            if file.endswith('.txt'):
                txt_documents.append(file)
            elif file.endswith('.doc') or file.endswith('.docx'):
                doc_documents.append(file)
    return txt_documents, doc_documents

def process_documents(documents, file_type='word'):
    """
    优化后的代码，通过 file_type 判断文件类型，用不同的方法进行处理
    :return: {'word_count': 字数统计值, 'line_count': 行数统计值}
    """
    summary = {}
    if file_type == 'word':
        for document_name in documents:
            # 打开文档
            document = Document(document_name)
            # 统计文本字数
            text = ""
            for paragraph in document.paragraphs:
                text += paragraph.text
            word_count = len(text)
            # 统计行数
            line_count = len(document.paragraphs)
            summary[document_name] = {'word_count': word_count, 'line_count':
line_count}
        else:
            for document in documents:
                with open(document, 'r', encoding='gbk', errors='ignore') as file:
                    content = file.read()
                    # 根据具体需求，提取信息并进行汇总
                    # 例如，可以统计文档中的单词数量、行数等
                    # 中文情况
                    word_count = len(content)
                    line_count = content.count('\n') + 1
                    summary[document] = {'word_count': word_count, 'line_count':
line_count}
    return summary
```

部分输出结果如下：

```
文件： 3-2-目录\读者来信 4.docx
字数： 54
行数： 3
-----
文件： 3-2-目录\读者来信 5.docx
字数： 68
行数： 2
-----
文件： 3-2-目录\读者来信 6.docx
字数： 46
```

```
行数: 1
```

```
-----
```

```
字数总计: 388
```

```
行数总计: 12
```

观察输出结果看到，Word 文档的字数和行数都正常了，总字数从几万变为几百。

3. 优化保存结果

将成果展示在屏幕上虽便于调试代码，但若呈现给领导则显得不够正式。通常应将数据保存至电子表格中。如果不知怎样保存为电子表格，不用担心，马上咨询 ChatGPT。

问题 1:

现在有一个字典数据如下{'3-2-目录\读者来信 4.docx': {'word_count': 54, 'line_count': 3}, '3-2-目录\读者来信 5.docx': {'word_count': 68, 'line_count': 2}, '3-2-目录\读者来信 6.docx': {'word_count': 46, 'line_count': 1}}, 请问怎样保存为 CSV 文件

ChatGPT 回答如下:

```
import csv
data = {
    '3-2-目录\读者来信 4.docx': {'word_count': 54, 'line_count': 3},
    '3-2-目录\读者来信 5.docx': {'word_count': 68, 'line_count': 2},
    '3-2-目录\读者来信 6.docx': {'word_count': 46, 'line_count': 1}
}
output_path = "path/to/output.csv"

# 要保存的字段名
fieldnames = ['Filename', 'Word Count', 'Line Count']
with open(output_path, mode='w', newline='') as file:
    writer = csv.DictWriter(file, fieldnames=fieldnames)
    # 写入列名
    writer.writeheader()
    # 写入数据
    for filename, counts in data.items():
        writer.writerow({'Filename': filename, 'Word Count': counts['word_count'],
                        'Line Count': counts['line_count']})
```

根据 ChatGPT 给出的回答，建议使用 CSV 来操作电子表格，它是一个比 Excel 轻量的电子表格格式，操作简单。我们根据示例代码增加一个 save_data()函数。

```
def save_data(data, output_path):
    """
    把数据记录到电子表格里
    """
    # 要保存的字段名
    fieldnames = ['文件名称', '字数', '行数']
    with open(output_path, mode='w', newline='') as file:
        writer = csv.DictWriter(file, fieldnames=fieldnames)
        # 写入列名
        writer.writeheader()
        # 写入数据
        for filename, counts in data.items():
            writer.writerow(
                {'文件名称': filename, '字数': counts['word_count'], '行数':
                 counts['line_count']})
```

最后一起打开保存的 CSV 文件察看效果，表格内容如图 3-1 所示。

A	B	C	D
文件名称	字数	行数	
3-3-目录\读者来信1.txt	87	3	
3-3-目录\读者来信2.txt	66	1	
3-3-目录\读者来信3.txt	67	2	
3-3-目录\读者来信4.docx	54	3	
3-3-目录\读者来信5.docx	68	2	
3-3-目录\读者来信6.docx	46	1	

图 3-1 保存为 CSV 文件

3.4 实战：摄影集文件整理

一位摄影师最近完成了一次令人难忘的旅行拍摄活动，期间他拍摄了大量照片，现在需要对这些照片进行批量重命名，以便更好地整理、归档和分享它们。摄影师希望按照城市名称对照片进行命名，并采用城市_XXXX.jpg 的格式，其中，XXXX 是一个数字编号。

3.4.1 问题需求分析

针对上面的需求使用流程图进行分析，根据问题描述给出的流程图如图 3-2 所示。

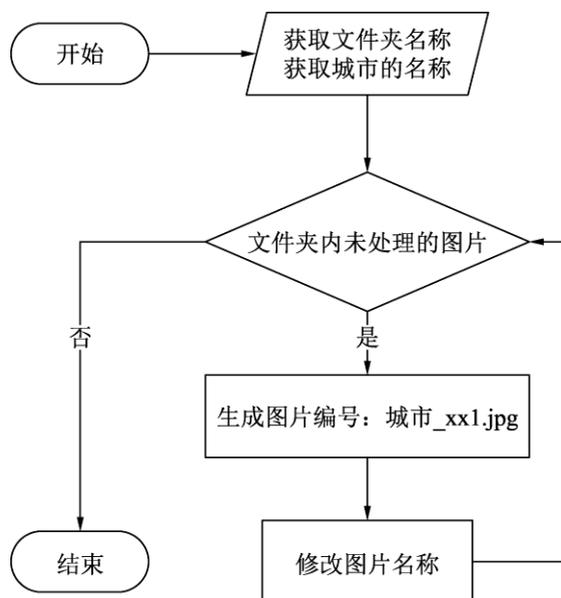


图 3-2 修改图片名称流程

3.4.2 代码编写

现在编写脚本将文件名中的数字编号替换为更有意义的名称。首先需要导入必要的模块。

```
import os
```

接下来定义一个函数来批量更改文件名称。

```
def rename_photos(folder_path, prefix):
    files = os.listdir(folder_path)
    count = 1
    for file in files:
        if file.endswith('.jpg'):
            old_name = os.path.join(folder_path, file)
            new_name = os.path.join(folder_path, f'{prefix}_{count}.jpg')
            os.rename(old_name, new_name)
            count += 1
```

在上述代码中，`rename_photos()`函数接收两个参数：`folder_path` 是照片文件夹的路径；`prefix` 是想要添加的前缀名称。

然后调用定义的函数来执行脚本。假设照片文件夹的路径为 `/path/to/photos`，并且想要将它们的前缀更改为“成都”。

```
if __name__ == '__main__':
    folder_path = '/path/to/photos'
    city = '成都'
    rename_photos(folder_path, city)
```

运行脚本后，照片文件名将从原始的 `IMG_XXXX.jpg` 更改为 `成都_1.jpg`、`成都_2.jpg` 的名称。在实际应用中可能需要根据具体需求进行修改和扩展。例如，根据拍摄日期、景点名称或其他标识符来更改文件名称，只需要简单修改脚本中的逻辑即可满足不同的需求。

3.4.3 调试与优化

通过 3.4.2 节的程序，成功地使用 Python 脚本批量更改了照片文件的名称。现在，能够轻松地整理、归档和分享照片了！此时需要将照片打包发给朋友，那么要用到批量压缩文件进行处理。因为文件名称已经修改，现在的文件名非常有规律，通过关键词选中需要打包压缩的文件然后发送给朋友即可。例如，把成都拍的照片发给成都的朋友，可以利用 `glob` 来筛选文件。

```
# 获取当前目录下所有成都的图片
img_files = glob.glob('成都_*.jpg')
print(img_files)
```

找到全部文件后，然后用 `zipfile` 压缩打包，完整代码如下：

```
import os
import zipfile

def rename_photos(folder_path, prefix):
    """
    重命名照片函数
    Args:
        folder_path (str): 照片所在文件夹的路径
```

```

    prefix (str): 照片名称前缀
Returns:
    list: 被重命名后的照片路径列表
"""
# 获取文件夹中的所有文件
files = os.listdir(folder_path)
# 计数器, 用于生成新的照片名称
count = 1
# 存储被重命名后的照片路径
images = []
for file in files:
    # 根据实际情况选中文件后缀, 如, jpg、png 和 mp4 等
    if file.endswith('.png'):
        # 获取原始照片路径
        old_name = os.path.join(folder_path, file)
        # 生成新的照片路径
        new_name = os.path.join(folder_path, f'{prefix}_{count}.jpg')
        # 重命名照片
        os.rename(old_name, new_name)
        count += 1
        # 将新的照片路径添加到列表中
        images.append(new_name)
return images

def compress_file(file_paths, zip_name):
    """
    创建 ZIP 压缩文件和解压
    :param file_paths: 需要压缩的文件路径列表
    :param zip_name: 生成的 ZIP 压缩文件名
    """
    with zipfile.ZipFile(zip_name, 'w') as zipf:
        for file in file_paths:
            # 压缩文件
            zipf.write(file)

if __name__ == '__main__':
    # 定义城市名称
    city = '成都'
    # 图片文件夹的目录地址
    source_path = "3-4-目录"
    image_paths = rename_photos(source_path, city)
    # 压缩文档
    compress_file(image_paths, f'{city}.zip')

```

编写程序也是一种创作, 创作是没有终点的, 可以不断优化, 根据实际需求可以增加功能, 如把压缩好的文件放到邮箱然后发送给相关的人, 或者把压缩好的文件上传到云盘, 然后清理已经压缩的文件, 节约磁盘空间等。

3.5 实战：重要文档定期备份

在工作和生活中, 我们时常需要对关键文档和文件进行定期备份, 以防数据丢失或意外损坏。通过运用 Python 编程, 可实现自动化备份过程, 从而确保重要文档得到妥善保管,

提高数据安全性。本节将以实际工作场景为例，介绍如何运用 Python 完成定期备份重要文档的任务。身为项目经理，每日均需处理大量项目文档和报告。这些文档对项目具有至关重要的意义，在共享目录中文档有被破坏的风险，如同事误操作删除或覆盖了源文件、因计算机故障造成源文件丢失等。因此你决定实施每日定期备份策略，确保在文件出现问题时能够及时恢复，减少损失。

3.5.1 问题需求分析

关于备份文档，最简单的实现方式是定期将共享文档中“/项目/你的文件目录”的文件夹全部复制到另一个安全的位置，如“/项目/备份文件夹”。

这是最关键的步骤，若想备份更安全、有效，就需要增加辅助功能。

- ❑ 备份日志功能：监控备份过程是否执行完成，是全部备份，还是只有部分文件备份。
- ❑ 检验备份数据：验证备份的数据和源数据是否完全一致。
- ❑ 配置备份计划：确保备份频率和时间，能够控制备份的执行时间和周期。

3.5.2 代码编写

下面编写 Python 脚本实现自动备份重要文档的功能。首先需要导入所需的模块。

```
import os
import shutil
import datetime
```

接下来定义一个函数来执行备份操作。

```
def backup_documents(source_folder, backup_folder):
    """
    备份文档函数
    Args:
        source_folder (str): 源文件夹路径
        backup_folder (str): 备份文件夹路径
    """
    # 获取当前日期
    backup_date = datetime.datetime.now().strftime("%Y-%m-%d")
    # 构建备份文件夹路径
    backup_folder_path = os.path.join(backup_folder, backup_date)
    # 创建备份文件夹，如果已存在则不报错
    os.makedirs(backup_folder_path, exist_ok=True)
    # 遍历源文件夹中的文件
    for file_name in os.listdir(source_folder):
        # 检查文件是否以'.docx'或'.pdf'结尾
        if file_name.endswith('.docx') or file_name.endswith('.pdf'):
            source_file = os.path.join(source_folder, file_name)
            backup_file = os.path.join(backup_folder_path, file_name)
            # 打印备份文件信息
            print("备份文件: ", source_file)
            # 复制文件到备份文件夹
            shutil.copy2(source_file, backup_file)
```

在上述代码中，`backup_documents()`函数接收两个参数：第一个 `source_folder` 是要备份的源文件夹路径；第二个 `backup_folder` 是备份文件夹的路径。使用 `datetime` 模块来获取当前日期，并以“年-月-日”的格式创建一个新的备份文件夹。然后，遍历源文件夹中的每个文件，只选择“.docx”和“.pdf”文件进行备份。使用 `shutil` 库的 `copy2()`函数复制文件到备份文件夹中。

调用定义的函数执行备份操作。假设源文件夹的路径为 `/path/to/documents`，备份文件夹的路径为 `/path/to/backup`。

```
if __name__ == '__main__':
    source_path = '3-3-目录'
    backup_path = '3-5-目录/备份文件目录'
    # 创建备份文件夹（如果不存在）
    if not os.path.exists(backup_path):
        os.makedirs(backup_path)
    backup_documents(source_path, backup_path)
```

运行脚本后，重要文档将被自动备份到指定的备份文件夹中并按照日期进行归类，如图 3-3 所示。



图 3-3 修改图片名称

3.5.3 调试与优化

代码优化的策略已在需求分析中详细介绍过了，可根据实际情况逐一实施。以定期备份功能为例，此功能并非仅通过代码层面就能实现，而是需要根据操作系统来确定。在 Linux 系统中，可使用 `cron` 命令进行定时任务设定；在 Windows 系统中，可以通过任务计划程序来定期执行备份脚本。相关操作这里不展开介绍，有兴趣的读者可自行查阅相关资料。

可以根据自身需求调整脚本中的源文件夹与备份文件夹的路径，并根据实际情况进一步个性化设置与扩展功能。例如，可在备份文件名中添加时间戳以区分不同时间点的备份，或应用压缩技术压缩备份文件的大小，从而节省存储空间（具体可参见 3.2.3 节的例子）。

3.6 实战：文件定期清理

在实际工作中，时常需要对文件系统进行定期清理和整理，以确保文件有序且存储空间得以高效利用。借助 Python 编程语言，能编写脚本批量删除老旧文档、日志文件和空文

文件夹。以下为一个实际工作场景中的例子，展示如何运用 Python 完成此任务。

假设你是一家公司的系统文档管理员，负责维护和管理公司内部的文件系统。你在工作中注意到文件系统中存在许多旧文档、日志文档和空文件夹，它们占据了大量的存储空间。为了优化文件系统并释放存储空间，你希望用脚本定期执行批量删除操作。

3.6.1 问题需求分析

根据问题需求，可以将问题拆解为几个小任务。

- ❑ 定义删除策略：确定哪些类型的文件可以被删除，如多久的文档属于旧文档、哪些文档属于日志文档等。
- ❑ 遍历文件系统：递归遍历文件系统中的所有文件和文件夹。如果文件夹里还有文件夹，则同样需要遍历。
- ❑ 对每个文件或文件夹执行两步操作。第一步，检查是否符合删除策略。对于旧文档，可以根据最后的修改日期进行判断；对于日志文档，可以根据文件名或内容特征进行判断；对于空文件夹，可以检查其是否没有子文件或子文件夹。第二步，如果符合删除策略，则执行删除操作。
- ❑ 记录删除操作：可以将已删除的文件路径、文件大小等信息记录到日志文件中，以备参考。
- ❑ 监测删除过程：确保删除过程正常运行并没有错误。可以设置日志记录或发送通知跟踪删除进度和检测潜在的问题。

 **注意：**在执行删除操作之前，应确保已经备份了重要的文档和数据，以防止意外删除数据或丢失数据。

3.6.2 代码编写

(1) 定义删除策略，然后执行删除操作。

由于在前面的任务中已经实现遍历文件夹的功能，因此优先实现批量删除功能。首先引入必要的模块。

```
import os
import time
```

接下来定义一个函数执行删除操作。

```
def delete_files(folder_path, days_threshold):
    """
    删除指定文件夹中指定天数之前的文件和空目录。
    参数：
    folder_path (str): 文件夹路径
    days_threshold (int): 天数阈值
    """
    current_time = time.time()
    # 遍历指定文件夹下的所有文件和子文件夹
    for root, dirs, files in os.walk(folder_path, topdown=False):
        # 遍历文件夹下的所有文件
```

```

for file_name in files:
    # 获取文件的完整路径
    file_path = os.path.join(root, file_name)
    # 获取文件的修改时间
    file_modified_time = os.path.getmtime(file_path)
    # 判断文件是否超过指定天数阈值
    if current_time - file_modified_time > days_threshold:
        # 如果超过阈值，则删除文件
        os.remove(file_path)
# 遍历文件夹下的所有子文件夹
for dir_name in dirs:
    # 获取子文件夹的完整路径
    dir_path = os.path.join(root, dir_name)
    # 判断子文件夹是否为空
    if not os.listdir(dir_path):
        # 如果为空，则删除子文件夹
        os.rmdir(dir_path)

```

在上述代码中，`delete_files()`函数接收两个参数，其中，`folder_path`参数是要删除文件的根文件夹路径，`days_threshold`参数用于判断文件是否为旧文件的阈值天数。使用 `time` 模块的 `time()`函数获取当前时间。然后，使用 `os.walk()`方法遍历文件夹中的所有子文件夹和文件。对于每个文件，检查其最后修改时间是否超过了设定的阈值天数，如果超过，则使用 `os.remove()`删除文件。对于每个文件夹，检查其是否为空，如果是空文件夹，则使用 `os.rmdir()`删除文件夹。

(2) 记录删除操作。

使用 `print()`函数在适当的位置输出被删除的文档信息，部分代码如下：

```

# 如果超过阈值，则删除文件
os.remove(file_path)
# 执行删除文件函数后打印信息
print(f"文件: {file_path}被删除")

```

但这样记录操作信息量比较少，引入 `logging` 标准库可以解决问题。它是由 `Python` 标准库提供的，用于记录程序运行时的信息、错误信息和调试信息。使用 `logging` 标准库能够将程序的日志记录到文件、控制台或其他输出目标中，并能够根据不同的日志级别过滤和格式化日志信息。

使用 `logging` 也很简单，只需要在程序开始时候初始化 `logging`，后续使用和 `print()`函数类似，代码如下：

```

import logging
# 初始化日志记录
logging.basicConfig(
    filename='app.log', # 日志文件名
    level=logging.INFO, # 日志级别为 INFO
    format='%(asctime)s - %(levelname)s - %(message)s' # 日志格式
)

# 记录日志的例子
logging.debug("这是一个调试信息")
logging.info("这是一个信息")
logging.warning("这是一个警告")
logging.error("这是一个错误")
# 输出

```

```
2024-01-16 20:41:11,497 - INFO - 这是一个信息
2024-01-16 20:41:11,498 - WARNING - 这是一个警告
2024-01-16 20:41:11,498 - ERROR - 这是一个错误
```

首先通过调用 `basicConfig()` 函数来配置日志记录。通过指定日志文件名、日志级别和日志格式，将日志记录到文件 `app.log` 中，并将日志级别设置为 `INFO`，即只记录 `INFO` 级别及以上的日志信息，因此在终端调试信息时没有记录。然后通过调用 `debug()`、`info()`、`warning()` 和 `error()` 等函数记录不同级别的日志信息。这些函数的参数是一个字符串，表示要记录的日志信息。通过使用不同的日志级别，可以灵活地控制记录的日志信息。例如，将日志级别设置为 `DEBUG`，则会记录所有级别的日志信息；将日志级别设置为 `WARNING`，则只会记录警告和错误级别的日志信息。日志级别从低到高是 `DEBUG`、`INFO`、`WARNING` 和 `ERROR`。最后用 `logging.info()` 代替 `print()` 函数，输出的字符串也不需要替换。

```
# 如果超过阈值，则删除文件
os.remove(file_path)
# 执行删除文件函数后打印信息
logging.info(f"文件: {file_path}被删除")
```

输出的信息记录在文件中。从输出结果看，记录多了一个时间，是输出日志的时间，重复运行代码，会发现日志不会清空，而是按顺序记录每次运行的日志。

现在调用定义的函数来执行删除操作。假设文件系统路径为 `/path/to/files`，希望删除 60 天前的旧文件。

```
if __name__ == '__main__':
    folder_path = '/path/to/files'
    # 定义要删除的文件扩展名和最大文件年龄（以秒为单位）
    days_threshold = 60*60*24*60 # 60天
    delete_files(folder_path, days_threshold)
```

运行脚本后，旧文档、日志文档和空文件夹将被自动删除。

3.6.3 调试与优化

通过前面的多个实战练习，相信读者已经对文件和文件夹的操作驾轻就熟了。一些细心的读者可能已察觉到，在每个任务中都有几个常用的小功能，如获取当前文档的绝对路径、遍历文件夹及创建文件夹等。实际上，可以将这些常用功能封装成一个函数并纳入个人开发工具包，然后在需要时通过导入方式将其应用到相关的脚本程序中，从而提高代码的利用率。

1. 创建文件夹

以创建文件夹为例，之前的代码和转变为函数的代码对比如下：

```
# 创建备份文件夹（如果不存在）
if not os.path.exists(backup_path):
    os.makedirs(backup_path)

# 转为函数
def create_folder(folder_path):
    if not os.path.exists(folder_path):
        os.makedirs(folder_path)
```

不要小看上面的函数代码，觉得没有必要写成函数，只节省了一行代码量。然而，这只是当前的情况，仅检测文件夹是否存在，以避免程序发生错误。随着功能的升级，可添加 `try-except` 模块增加错误处理功能，增设日志记录，并在创建同名文件夹时允许用户选择是否覆盖旧文件夹。代码升级如下：

```
import os
import shutil
import logging

def create_folder_v2(folder_path, ensure_overwrite=False):
    try:
        if os.path.exists(folder_path):
            if ensure_overwrite:
                # 文件夹已经存在且允许覆盖
                # 例如，覆盖现有文件夹或重命名文件夹
                shutil.rmtree(folder_path)          # 删除现有文件夹
                logging.info(f"删除同名文件夹成功: {folder_path}")
                os.makedirs(folder_path)            # 创建新文件夹
                logging.info(f"创建文件夹成功: {folder_path}")
            else:
                logging.warning(f"文件夹已经存在: {folder_path}")
        else:
            os.makedirs(folder_path)
            logging.info(f"创建文件夹成功: {folder_path}")
    except OSError as e:
        logging.error(f"创建文件夹失败: {e}")
```

随着问题越来越复杂，函数也会不断优化。使用包模块可以让全部项目的同一个函数保持一致，避免遗漏更新，所有使用它的程序都能马上使用到最新的功能。

2. 遍历文件夹的方法

前面使用了几种方式实现遍历文件夹里的所有文件，如 `os.listdir()` 可以读取指定文件夹内的全部文件，`glob.glob()` 也可以实现相同的功能，而且还支持使用通配符来寻找文件。读者可以根据自己的喜好，把其中一种方式改写为通用的函数。下面使用 `os.listdir()` 改写一个遍历文件夹文档的函数，代码如下：

```
def get_all_files_by_prefix(folder_path: str, prefix: str) -> List[str]:
    """
    根据文件名前缀获取指定文件夹下的所有文件路径列表
    Args:
        folder_path (str): 文件夹路径
        prefix (str): 文件名前缀
    Returns:
        list: 包含所有满足条件的文件路径的列表
    """
    files = os.listdir(folder_path)          # 获取文件夹中的所有文件和文件夹
    output_files = []                       # 存储满足条件的文件路径
    for file in files:
        if file.endswith(prefix):          # 判断文件名是否以指定前缀结尾
            # 将满足条件的文件路径添加到列表中
            output_files.append(os.path.join(folder_path, file))
    return output_files                    # 返回满足条件的文件路径列表
```

上面是一个相对规范的函数编写例子，参数带上了类型说明，函数第一部分也有详细的使用说明，而且整个函数有非常详细的代码注释。若想进一步优化代码，但又没有具体方向，可以请教 ChatGPT。

问题 1:

你是一位 Python 研发工程师，我输入一个函数，你给出优化函数的建议。我的第一个指令是：

【把函数放在这里】

ChatGPT 回答 1:

这个函数看起来已经很不错了，但还有一些优化的建议可以考虑：

使用列表推导式：你可以使用列表推导式来简化 `for` 循环和条件判断的部分。这将使代码更加简洁和可读。

```
files = [file for file in os.listdir(folder_path) if file.endswith(prefix)]
```

使用 `os.scandir()` 替代 `os.listdir()`：`os.scandir()` 比 `os.listdir()` 更高效，因为它返回一个迭代器而不是一个列表。这可以减少内存消耗和文件系统的访问次数。

```
with os.scandir(folder_path) as entries:
    files = [entry.name for entry in entries if entry.is_file() and
            entry.name.endswith(prefix)]
```

考虑处理大型文件夹：如果文件夹中包含大量的文件，可能需要分批处理以避免内存问题。可以在每次迭代时对结果进行处理，而不是将所有满足条件的文件路径都存储在一个列表中。

以上是一些建议供参考，根据具体情况，在性能、代码可读性和维护性之间作出权衡。记得修改后测试代码，确保其正确性和性能的改进。

读者可以按照 ChatGPT 的建议继续优化代码，这里就不一一展开了。若有其他函数，可以继续询问，只需要输入“第二个指令：（完整的函数代码）”即可。

3.7 总 结

通过本章的学习，读者深入了解了使用 Python 进行文档读写操作的技巧，包括文件的增、删、改操作和异常处理，还学会了文件夹的基本操作和文件路径操作，并掌握了批量文件压缩的方法。通过实战案例的学习，锻炼了问题需求分析、代码编写、代码调试和优化的能力。同时借助 ChatGPT 的能力可以提高编程效率，提升编写代码的能力，拓展优化代码的思路。

第 4 章将深入介绍如何利用 Python 高效地进行数据分析，如何通过编写程序提高数据收集的效率，避免在数据分析过程中可能会出现错误并迅速生成分析报告。