Claude Code 国内用户完整使用手册

专业技术文档系列 💙

👤 AI 技术博主 ✨

Python自动化办公AI应用实战手册

让AI帮你写代码，办公效率提升10倍

目录

第1章 Python自动化办公基础

1.1 为什么选择Python进行办公自动化?

在日常办公中，我们每天都要处理大量的重复性工作：整理Excel表格、处理PDF文件、发送邮件、管理文件等等。这些工作不仅耗时耗力，还容易出错。

Python作为一门简洁优雅的编程语言，配合AI助手，能够完美解决这些问题：

痛点	传统方式	Python+AI方案
重复性工作	手动操作，耗时耗力	一键自动化，效率提升10倍
容易出错	人工操作难免疏漏	程序精准执行，零错误
编程门槛	需要学习复杂语法	AI生成代码，零基础可用
成本	购买专业软件	完全免费开源

1.1.1 Python的优势

1. **简单易学**：语法接近自然语言，阅读代码就像读英文
2. **功能强大**：拥有丰富的第三方库，几乎能做任何事情
3. **生态完善**：Excel、PDF、邮件、文件处理都有成熟方案

4. **AI友好**：ChatGPT、Claude等AI助手对Python支持最好

1.1.2 AI时代的编程新方式

不会写代码？没关系！

现在的AI助手（如Claude Code、ChatGPT）可以根据你的描述自动生成Python代码。

示例对话：

> *你：请帮我写一个Python脚本，批量合并文件夹中的所有Excel文件*
>
> *AI：（生成完整代码）*
>
> *你：复制代码 → 保存 → 运行 → 完成！*

1.2 环境搭建（5分钟搞定）

1.2.1 安装Python

1. 访问 Python官网
2. 下载最新版本（建议3.8或以上）
3. 安装时勾选 **“Add Python to PATH”**
4. 点击安装，等待安装完成

1.2.2 验证安装

打开命令行（Windows按 `Win+R` 输入 `cmd`，Mac打开终端），输入：

```
python --version
```

如果显示版本号（如 `Python 3.10.0`），说明安装成功！

1.2.3 安装VS Code（推荐编辑器）

虽然可以用记事本写代码，但VS Code更好用：

1. 访问 VS Code官网
2. 下载并安装
3. 安装Python扩展（插件市场搜索"Python"）

1.3 第一个Python程序

打开VS Code，新建文件 `hello.py`，输入：

```
print("Hello，办公自动化！")
```

保存后，在终端运行：

```
python hello.py
```

看到输出 `Hello，办公自动化！`，恭喜你，第一个程序运行成功！

1.4 如何使用AI生成代码

1.4.1 向AI描述需求

清晰描述你的需求，包含：

- 要处理什么文件?
- 要完成什么操作?
- 输出什么结果?

示例提示词:

```
请帮我写一个Python脚本，实现以下功能：
1. 读取文件夹中的所有Excel文件
2. 提取每个文件的"销售额"列
3. 汇总到一个新的Excel文件中
4. 生成柱状图展示各文件销售额对比
```

◆ 1.4.2 运行AI生成的代码

1. 复制AI生成的代码
2. 保存为 `.py` 文件
3. 根据提示修改文件路径等参数
4. 运行: `python 文件名.py`

◆ 1.4.3 调试和优化

- 如果报错，把错误信息发给AI，让它帮你修复
- 如果结果不对，告诉AI具体哪里不对，让它调整
- 逐步迭代，直到满足需求

1.5 本书使用指南

1.5.1 章节结构

每章包含：

1. **场景介绍**：为什么需要这个功能
2. **环境准备**：需要安装什么
3. **实战案例**：完整的代码示例
4. **代码解析**：关键代码的详细说明
5. **练习作业**：巩固所学知识

1.5.2 学习建议

1. **先通读**：了解每章的功能和应用场景
2. **再实践**：复制代码运行，观察效果
3. **后修改**：根据自己的需求修改代码
4. **多提问**：遇到问题随时向AI助手提问

1.6 本章小结

核心要点：

1. Python+AI是办公自动化的最佳组合
2. 环境搭建简单，5分钟即可完成
3. 不会编程也能用，AI帮你写代码
4. 本书提供完整的实战案例，即学即用

下一步： 进入第2章，学习如何用Python自动化处理Excel文件！

第2章 Excel自动化：让数据处理效率提升10倍

2.1 为什么用Python处理Excel？

2.1.1 传统Excel操作的痛点

在日常办公中，Excel是我们最常用的数据处理工具。但当面对大量数据时，传统的手工操作方式往往让人力不从心：

- **重复性工作耗时**：每月需要合并几十个Excel文件，复制粘贴到手酸
- **容易出错**：人工操作难免有疏漏，一个数据错误可能导致整个报表失真
- **难以自动化**：复杂的VBA编程门槛高，维护困难
- **处理速度慢**：打开一个包含几十万行数据的Excel文件，电脑直接卡死

◆ 2.1.2 Python处理Excel的优势

Python作为一门简洁优雅的编程语言，配合强大的数据处理库，能够完美解决上述问题：

对比维度	传统Excel操作	Python自动化
处理速度	打开大文件卡顿	秒级处理百万行数据
重复工作	手动复制粘贴	一键批量处理
准确性	人工易出错	程序精准执行
可复用性	每次重新操作	脚本永久复用
功能扩展	受限于Excel功能	无限扩展可能

实际案例：某财务同事每月需要汇总30个分公司的销售报表，手工操作需要4小时，使用Python脚本后只需3分钟，效率提升80倍！

◆ 2.1.3 本章学习目标

通过本章学习，你将掌握：

- 使用`openpyxl`处理Excel文件的读写操作
- 使用`pandas`进行高效的数据分析和处理
- 批量处理多个Excel文件
- 数据清洗、去重、格式化
- 自动生成统计报表和可视化图表

2.2 环境搭建

2.2.1 安装Python

如果你还没有安装Python，请访问Python官网下载并安装最新版本（建议3.8或以上）。

安装时请勾选 **“Add Python to PATH”** 选项，这样可以在命令行中直接使用Python命令。

2.2.2 安装必要的库

打开命令行工具（Windows按 `Win+R` 输入 `cmd` ，Mac打开终端），执行以下命令：

```
# 安装openpyxl - 用于读写Excel 2010及以上版本（.xlsx格式）
pip install openpyxl

# 安装pandas - 强大的数据分析库
pip install pandas

# 安装matplotlib - 用于生成图表
pip install matplotlib

# 安装xlrd - 用于读取旧版Excel（.xls格式，可选）
pip install xlrd
```

2.2.3 验证安装

安装完成后，在命令行中输入 `python` 进入Python交互环境，然后输入：

```
import openpyxl
import pandas as pd
import matplotlib.pyplot as plt

print("openpyxl版本:", openpyxl.__version__)
print("pandas版本:", pd.__version__)
print("安装成功！")
```

如果看到版本号输出且没有报错，说明安装成功！按`Ctrl+Z`（Windows）或`Ctrl+D`（Mac）退出Python环境。

2.3 实战案例

2.3.1 案例一：批量读取多个Excel文件

场景：假设你是销售部门的数据分析师，每月需要汇总各区域的销售数据。每个区域的数据保存在单独的Excel文件中，你需要将它们合并成一个总表。

准备工作：

1. 创建一个文件夹`sales_data`
2. 在里面放入多个Excel文件，如：`华东区.xlsx`、`华南区.xlsx`、`华北区.xlsx`
3. 每个文件的格式相同，包含列：日期、产品名称、销售额、销售员

完整代码：

```
import pandas as pd
import os

def merge_excel_files(folder_path, output_file):
    """
    批量合并文件夹中的所有Excel文件

    参数:
        folder_path: Excel文件所在的文件夹路径
        output_file: 合并后的输出文件名
    """
    # 创建一个空列表，用于存储所有数据
    all_data = []

    # 遍历文件夹中的所有文件
    for filename in os.listdir(folder_path):
        # 只处理.xlsx和.xls格式的文件
        if filename.endswith(('.xlsx', '.xls')):
            # 构建完整的文件路径
            file_path = os.path.join(folder_path, filename)

            # 读取Excel文件
            # sheet_name=0 表示读取第一个工作表
            df = pd.read_excel(file_path, sheet_name=0)

            # 添加一列，记录数据来源（文件名）
            df['数据来源'] = filename.replace('.xlsx', '').replace('.xls', ''

            # 将数据添加到列表中
            all_data.append(df)

            print(f"✓ 已读取: {filename}, 包含 {len(df)} 行数据")

    # 如果读取到了数据，进行合并
    if all_data:
        # 使用concat函数合并所有数据
        merged_df = pd.concat(all_data, ignore_index=True)

        # 保存到新的Excel文件
```

```
        merged_df.to_excel(output_file, index=False, engine='openpyxl')

        print(f"\n✅ 合并完成！")
        print(f"总计读取 {len(all_data)} 个文件")
        print(f"合并后共有 {len(merged_df)} 行数据")
        print(f"结果已保存至: {output_file}")

        return merged_df
    else:
        print("❌ 未找到任何Excel文件")
        return None

# ========== 使用示例 ==========
if __name__ == "__main__":
    # 设置文件夹路径和输出文件名
    folder = "./sales_data"  # 存放源文件的文件夹
    output = "合并后的销售数据.xlsx"  # 输出文件名

    # 执行合并操作
    result = merge_excel_files(folder, output)

    # 显示前5行数据预览
    if result is not None:
        print("\n数据预览（前5行）：")
        print(result.head())
```

代码解析：

- `os.listdir()`：获取文件夹中的所有文件名
- `pd.read_excel()`：读取Excel文件为DataFrame（类似Excel表格的数据结构）
- `pd.concat()`：将多个DataFrame纵向拼接
- `to_excel()`：将DataFrame保存为Excel文件
- `ignore_index=True`：重新生成行索引，避免重复

◆ 2.3.2 案例二：数据清洗（去重、格式化）

场景：合并后的数据往往存在重复记录、格式不统一等问题，需要进行清洗才能用于分析。

完整代码：

```python
import pandas as pd
import numpy as np
from datetime import datetime

def clean_sales_data(input_file, output_file):
    """
    清洗销售数据：去重、格式化、处理缺失值

    参数:
        input_file: 原始数据文件路径
        output_file: 清洗后的输出文件路径
    """
    # 1. 读取原始数据
    print("📖 正在读取数据...")
    df = pd.read_excel(input_file)
    print(f"原始数据: {len(df)} 行")

    # 2. 查看数据基本信息
    print("\n📊 数据基本信息：")
    print(df.info())
    print("\n缺失值统计：")
    print(df.isnull().sum())

    # 3. 删除完全重复的行
    print("\n🧹 正在去除重复数据...")
    before_count = len(df)
    df = df.drop_duplicates()
    after_count = len(df)
    print(f"删除了 {before_count - after_count} 行重复数据")

    # 4. 处理缺失值
    print("\n🔧 正在处理缺失值...")
    # 删除关键字段为空的行（如产品名称、销售额为空）
    df = df.dropna(subset=['产品名称', '销售额'])

    # 销售员为空时，填充为"未知"
    if '销售员' in df.columns:
        df['销售员'] = df['销售员'].fillna('未知')
```

```
    # 5. 数据格式化
    print("\n✨ 正在格式化数据...")

    # 5.1 日期格式统一
    if '日期' in df.columns:
        # 将日期列转换为datetime类型
        df['日期'] = pd.to_datetime(df['日期'], errors='coerce')
        # 删除日期格式错误的行
        df = df.dropna(subset=['日期'])
        # 添加年月列，方便后续统计
        df['年份'] = df['日期'].dt.year
        df['月份'] = df['日期'].dt.month

    # 5.2 产品名称标准化（去除前后空格，统一大小写）
    if '产品名称' in df.columns:
        df['产品名称'] = df['产品名称'].str.strip()  # 去除空格
        df['产品名称'] = df['产品名称'].str.title()  # 首字母大写

    # 5.3 销售额格式处理
    if '销售额' in df.columns:
        # 确保销售额是数值类型
        df['销售额'] = pd.to_numeric(df['销售额'], errors='coerce')
        # 删除销售额为0或负数的异常数据
        df = df[df['销售额'] > 0]
        # 保留两位小数
        df['销售额'] = df['销售额'].round(2)

    # 5.4 销售员姓名统一
    if '销售员' in df.columns:
        df['销售员'] = df['销售员'].str.strip()

    # 6. 保存清洗后的数据
    df.to_excel(output_file, index=False, engine='openpyxl')

    print(f"\n✅ 数据清洗完成！")
    print(f"清洗后数据: {len(df)} 行")
    print(f"结果已保存至: {output_file}")

    return df
```

```
# ========== 使用示例 ==========
if __name__ == "__main__":
    input_file = "合并后的销售数据.xlsx"
    output_file = "清洗后的销售数据.xlsx"

    clean_df = clean_sales_data(input_file, output_file)

    # 显示清洗后的数据样本
    print("\n清洗后的数据预览：")
    print(clean_df.head(10))
```

代码解析：

- `drop_duplicates()`：删除完全相同的重复行
- `dropna()`：删除包含缺失值的行
- `fillna()`：用指定值填充缺失值
- `pd.to_datetime()`：统一日期格式
- `str.strip()` / `str.title()`：字符串格式化
- `pd.to_numeric()`：转换为数值类型

◆ 2.3.3 案例三：自动生成统计报表

场景：清洗后的数据需要按不同维度进行统计分析，生成各类报表供管理层决策。

完整代码：

```
import pandas as pd
from openpyxl import Workbook
from openpyxl.styles import Font, Alignment, PatternFill, Border, Side
from openpyxl.utils.dataframe import dataframe_to_rows

def generate_sales_report(data_file, output_file):
    """
    自动生成多维度销售统计报表

    参数:
        data_file: 清洗后的数据文件路径
        output_file: 报表输出文件路径
    """
    # 1. 读取清洗后的数据
    df = pd.read_excel(data_file)
    print(f"📖 已加载数据: {len(df)} 行")

    # 2. 创建Excel工作簿
    wb = Workbook()
    wb.remove(wb.active)  # 删除默认工作表

    # ========== 报表1: 总体概览 ==========
    ws1 = wb.create_sheet("总体概览")

    # 计算关键指标
    total_sales = df['销售额'].sum()
    total_orders = len(df)
    avg_sales = df['销售额'].mean()
    max_sales = df['销售额'].max()
    min_sales = df['销售额'].min()

    # 写入标题
    ws1['A1'] = "销售数据总体概览"
    ws1['A1'].font = Font(size=16, bold=True, color="FFFFFF")
    ws1['A1'].fill = PatternFill(start_color="4472C4", end_color="4472C4", fi
    ws1['A1'].alignment = Alignment(horizontal="center")
    ws1.merge_cells('A1:B1')

    # 写入指标
```

```
    overview_data = [
        ["指标", "数值"],
        ["总销售额", f"¥{total_sales:,.2f}"],
        ["总订单数", f"{total_orders:,} 笔"],
        ["平均订单金额", f"¥{avg_sales:,.2f}"],
        ["最大单笔金额", f"¥{max_sales:,.2f}"],
        ["最小单笔金额", f"¥{min_sales:,.2f}"],
    ]

    for row_idx, row_data in enumerate(overview_data, start=3):
        for col_idx, value in enumerate(row_data, start=1):
            cell = ws1.cell(row=row_idx, column=col_idx, value=value)
            if row_idx == 3:  # 表头行
                cell.font = Font(bold=True)
                cell.fill = PatternFill(start_color="D9E1F2", end_color="D9E1
            cell.alignment = Alignment(horizontal="center")

    ws1.column_dimensions['A'].width = 20
    ws1.column_dimensions['B'].width = 20

    # ========== 报表2：按产品统计 ==========
    ws2 = wb.create_sheet("产品统计")

    product_stats = df.groupby('产品名称').agg({
        '销售额': ['sum', 'count', 'mean']
    }).round(2)
    product_stats.columns = ['总销售额', '订单数', '平均订单额']
    product_stats = product_stats.sort_values('总销售额', ascending=False)
    product_stats = product_stats.reset_index()

    # 添加占比列
    product_stats['销售占比'] = (product_stats['总销售额'] / total_sales * 100
    product_stats['销售占比'] = product_stats['销售占比'].astype(str) + '%'

    # 写入标题
    ws2['A1'] = "产品销售统计报表"
    ws2['A1'].font = Font(size=14, bold=True)
    ws2.merge_cells('A1:E1')

    # 写入数据
```

```
    for r_idx, row in enumerate(dataframe_to_rows(product_stats, index=False,
        for c_idx, value in enumerate(row, start=1):
            cell = ws2.cell(row=r_idx, column=c_idx, value=value)
            if r_idx == 3:  # 表头
                cell.font = Font(bold=True, color="FFFFFF")
                cell.fill = PatternFill(start_color="4472C4", end_color="4472
            cell.alignment = Alignment(horizontal="center")

    # 设置列宽
    for col in ['A', 'B', 'C', 'D', 'E']:
        ws2.column_dimensions[col].width = 18

    # ========== 报表3：按月份统计 ==========
    ws3 = wb.create_sheet("月度统计")

    monthly_stats = df.groupby(['年份', '月份']).agg({
        '销售额': ['sum', 'count']
    }).round(2)
    monthly_stats.columns = ['销售额', '订单数']
    monthly_stats = monthly_stats.reset_index()
    monthly_stats['年月'] = monthly_stats['年份'].astype(str) + '-' + monthly_
    monthly_stats = monthly_stats[['年月', '销售额', '订单数']]

    # 计算环比增长率
    monthly_stats['环比增长'] = monthly_stats['销售额'].pct_change() * 100
    monthly_stats['环比增长'] = monthly_stats['环比增长'].round(2).astype(str)

    ws3['A1'] = "月度销售统计报表"
    ws3['A1'].font = Font(size=14, bold=True)
    ws3.merge_cells('A1:D1')

    for r_idx, row in enumerate(dataframe_to_rows(monthly_stats, index=False,
        for c_idx, value in enumerate(row, start=1):
            cell = ws3.cell(row=r_idx, column=c_idx, value=value)
            if r_idx == 3:
                cell.font = Font(bold=True, color="FFFFFF")
                cell.fill = PatternFill(start_color="4472C4", end_color="4472
            cell.alignment = Alignment(horizontal="center")

    for col in ['A', 'B', 'C', 'D']:
```

```
        ws3.column_dimensions[col].width = 18

    # ========== 报表4：按区域统计 ==========
    if '数据来源' in df.columns:
        ws4 = wb.create_sheet("区域统计")

        region_stats = df.groupby('数据来源').agg({
            '销售额': ['sum', 'count', 'mean']
        }).round(2)
        region_stats.columns = ['总销售额', '订单数', '平均订单额']
        region_stats = region_stats.sort_values('总销售额', ascending=False)
        region_stats = region_stats.reset_index()

        ws4['A1'] = "区域销售统计报表"
        ws4['A1'].font = Font(size=14, bold=True)
        ws4.merge_cells('A1:D1')

        for r_idx, row in enumerate(dataframe_to_rows(region_stats, index=Fal
            for c_idx, value in enumerate(row, start=1):
                cell = ws4.cell(row=r_idx, column=c_idx, value=value)
                if r_idx == 3:
                    cell.font = Font(bold=True, color="FFFFFF")
                    cell.fill = PatternFill(start_color="4472C4", end_color="
                cell.alignment = Alignment(horizontal="center")

        for col in ['A', 'B', 'C', 'D']:
            ws4.column_dimensions[col].width = 18

    # 保存报表
    wb.save(output_file)
    print(f"\n✅ 报表生成完成！")
    print(f"包含工作表：{wb.sheetnames}")
    print(f"结果已保存至：{output_file}")

# ========== 使用示例 ==========
if __name__ == "__main__":
    data_file = "清洗后的销售数据.xlsx"
    report_file = "销售统计报表.xlsx"
```

```
generate_sales_report(data_file, report_file)
```

代码解析：

- `groupby()`：按指定列分组统计
- `agg()`：聚合计算（sum、count、mean等）
- `pct_change()`：计算环比增长率
- `openpyxl`：创建带格式的Excel报表
- `Font`、`Alignment`、`PatternFill`：设置单元格样式

◆ 2.3.4 案例四：批量生成图表

场景：数据可视化能够更直观地展示分析结果，我们需要根据统计数据自动生成各类图表。

完整代码：

```
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib
from matplotlib import font_manager

# 设置中文字体（解决中文显示问题）
# Windows系统使用 SimHei，Mac使用 Arial Unicode MS
plt.rcParams['font.sans-serif'] = ['SimHei', 'Arial Unicode MS', 'DejaVu Sans
plt.rcParams['axes.unicode_minus'] = False  # 解决负号显示问题

def generate_sales_charts(data_file, output_folder):
    """
    批量生成销售数据可视化图表

    参数:
        data_file: 数据文件路径
        output_folder: 图表输出文件夹
    """
    # 读取数据
    df = pd.read_excel(data_file)
    print(f"📖 已加载数据: {len(df)} 行")

    # 创建输出文件夹
    import os
    os.makedirs(output_folder, exist_ok=True)

    # ========== 图表1: 销售额趋势图（折线图）==========
    plt.figure(figsize=(12, 6))

    monthly_sales = df.groupby(['年份', '月份'])['销售额'].sum().reset_index()
    monthly_sales['年月'] = monthly_sales['年份'].astype(str) + '-' + monthly_

    plt.plot(monthly_sales['年月'], monthly_sales['销售额'],
             marker='o', linewidth=2, markersize=8, color='#4472C4')
    plt.title('月度销售额趋势', fontsize=16, fontweight='bold', pad=20)
    plt.xlabel('月份', fontsize=12)
    plt.ylabel('销售额（元）', fontsize=12)
    plt.xticks(rotation=45)
    plt.grid(True, alpha=0.3, linestyle='--')
```

```
    # 添加数值标签
    for i, (x, y) in enumerate(zip(monthly_sales['年月'], monthly_sales['销售
        plt.annotate(f'{y:,.0f}', (x, y), textcoords="offset points",
                     xytext=(0,10), ha='center', fontsize=9)

    plt.tight_layout()
    plt.savefig(f'{output_folder}/01_销售额趋势图.png', dpi=300, bbox_inches='
    print("✓ 已生成: 01_销售额趋势图.png")
    plt.close()

    # ========== 图表2：产品销售额占比（饼图）==========
    plt.figure(figsize=(10, 8))

    product_sales = df.groupby('产品名称')['销售额'].sum().sort_values(ascendi

    # 如果产品太多，只显示前5个，其他归为"其他"
    if len(product_sales) > 5:
        top5 = product_sales.head(5)
        others = product_sales.iloc[5:].sum()
        product_sales = pd.concat([top5, pd.Series({'其他': others})])

    colors = ['#4472C4', '#ED7D31', '#A5A5A5', '#FFC000', '#5B9BD5', '#70AD47

    wedges, texts, autotexts = plt.pie(product_sales, labels=product_sales.in
                                       autopct='%1.1f%%', startangle=90,
                                       colors=colors, textprops={'fontsize':

    # 设置百分比文字样式
    for autotext in autotexts:
        autotext.set_color('white')
        autotext.set_fontweight('bold')

    plt.title('产品销售额占比', fontsize=16, fontweight='bold', pad=20)
    plt.axis('equal')
    plt.tight_layout()
    plt.savefig(f'{output_folder}/02_产品销售额占比.png', dpi=300, bbox_inches
    print("✓ 已生成: 02_产品销售额占比.png")
    plt.close()
```

```
# ========== 图表3：区域销售对比（柱状图）==========
if '数据来源' in df.columns:
    plt.figure(figsize=(10, 6))

    region_sales = df.groupby('数据来源')['销售额'].sum().sort_values(asce

    bars = plt.barh(region_sales.index, region_sales.values, color='#4472
    plt.title('各区域销售额对比', fontsize=16, fontweight='bold', pad=20)
    plt.xlabel('销售额（元）', fontsize=12)
    plt.ylabel('区域', fontsize=12)
    plt.grid(True, alpha=0.3, axis='x', linestyle='--')

    # 添加数值标签
    for bar in bars:
        width = bar.get_width()
        plt.text(width, bar.get_y() + bar.get_height()/2,
                 f'¥{width:,.0f}', ha='left', va='center', fontsize=10)

    plt.tight_layout()
    plt.savefig(f'{output_folder}/03_区域销售对比.png', dpi=300, bbox_inch
    print("✓ 已生成: 03_区域销售对比.png")
    plt.close()

# ========== 图表4：销售员业绩排名（横向柱状图）==========
if '销售员' in df.columns:
    plt.figure(figsize=(10, 8))

    salesperson_sales = df.groupby('销售员')['销售额'].sum().sort_values(a

    # 只显示前10名
    if len(salesperson_sales) > 10:
        salesperson_sales = salesperson_sales.tail(10)

    colors = plt.cm.Blues([0.3 + 0.7 * i / len(salesperson_sales)
                           for i in range(len(salesperson_sales))])

    bars = plt.barh(salesperson_sales.index, salesperson_sales.values, co
    plt.title('销售员业绩排名（Top 10）', fontsize=16, fontweight='bold',
    plt.xlabel('销售额（元）', fontsize=12)
    plt.ylabel('销售员', fontsize=12)
```

```
        plt.grid(True, alpha=0.3, axis='x', linestyle='--')

        # 添加数值标签
        for bar in bars:
            width = bar.get_width()
            plt.text(width, bar.get_y() + bar.get_height()/2,
                     f'¥{width:,.0f}', ha='left', va='center', fontsize=9)

        plt.tight_layout()
        plt.savefig(f'{output_folder}/04_销售员业绩排名.png', dpi=300, bbox_in
        print("✓ 已生成: 04_销售员业绩排名.png")
        plt.close()

    # ========== 图表5：月度订单量趋势（双轴图）==========
    fig, ax1 = plt.subplots(figsize=(12, 6))

    monthly_stats = df.groupby(['年份', '月份']).agg({
        '销售额': 'sum',
        '产品名称': 'count'  # 统计订单数
    }).reset_index()
    monthly_stats.columns = ['年份', '月份', '销售额', '订单数']
    monthly_stats['年月'] = monthly_stats['年份'].astype(str) + '-' + monthly_

    # 左轴：销售额
    color1 = '#4472C4'
    ax1.set_xlabel('月份', fontsize=12)
    ax1.set_ylabel('销售额（元）', color=color1, fontsize=12)
    line1 = ax1.plot(monthly_stats['年月'], monthly_stats['销售额'],
                     color=color1, marker='o', linewidth=2, label='销售额')
    ax1.tick_params(axis='y', labelcolor=color1)
    ax1.tick_params(axis='x', rotation=45)

    # 右轴：订单数
    ax2 = ax1.twinx()
    color2 = '#ED7D31'
    ax2.set_ylabel('订单数', color=color2, fontsize=12)
    line2 = ax2.plot(monthly_stats['年月'], monthly_stats['订单数'],
                     color=color2, marker='s', linewidth=2, label='订单数')
    ax2.tick_params(axis='y', labelcolor=color2)
```

```
    # 合并图例
    lines = line1 + line2
    labels = [l.get_label() for l in lines]
    ax1.legend(lines, labels, loc='upper left')

    plt.title('销售额与订单量趋势对比', fontsize=16, fontweight='bold', pad=20)
    plt.grid(True, alpha=0.3, linestyle='--')
    plt.tight_layout()
    plt.savefig(f'{output_folder}/05_销售趋势对比.png', dpi=300, bbox_inches='
    print("✓ 已生成: 05_销售趋势对比.png")
    plt.close()

    print(f"\n✅ 所有图表生成完成！")
    print(f"图表保存在: {output_folder}/")

# ========== 使用示例 ==========
if __name__ == "__main__":
    data_file = "清洗后的销售数据.xlsx"
    output_folder = "销售图表"

    generate_sales_charts(data_file, output_folder)
```

代码解析:

- `plt.figure()`：创建新图表，设置尺寸
- `plt.plot()`：绘制折线图
- `plt.pie()`：绘制饼图
- `plt.barh()`：绘制横向柱状图
- `twinx()`：创建双Y轴图表
- `plt.savefig()`：保存图表为图片
- `dpi=300`：设置图片清晰度

2.4 本章小结

通过本章的学习，你已经掌握了使用Python进行Excel自动化的核心技能：

1. **批量读取**：使用 `pandas` 和 `os` 模块，轻松合并多个Excel文件
2. **数据清洗**：运用 `drop_duplicates()` 、 `fillna()` 等方法，快速处理脏数据
3. **统计报表**：利用 `groupby()` 和 `openpyxl` ，生成专业的多维度报表
4. **可视化**：借助 `matplotlib` ，批量生成各类图表

◆ 效率对比

任务	手工操作	Python自动化	效率提升
合并30个Excel文件	2小时	3分钟	40倍
清洗10万行数据	4小时	30秒	480倍
生成统计报表	3小时	1分钟	180倍
制作5张图表	2小时	30秒	240倍

◆ 下一步学习建议

- 尝试修改案例代码，适应你自己的业务场景
- 学习 `pandas` 更多高级功能，如数据透视表 `pivot_table()`
- 探索 `seaborn` 库，创建更美观的统计图表
- 了解 `schedule` 库，实现定时自动执行脚本

练习作业：

1. 准备3-5个Excel文件，练习批量合并功能
2. 在数据清洗案例中添加一个新功能：检测并标记异常值（如销售额超过平均值3倍的数据）
3. 在报表中添加一个"销售员业绩排名"工作表
4. 尝试修改图表样式，使用你喜欢的配色方案

> 💡 ***小贴士***：*将本章代码保存为 `.py` 文件后，可以设置Windows任务计划程序或Mac的cron定时执行，实现真正的"无人值守"自动化办公！*

第3章 PDF自动化处理

在日常办公中，PDF（Portable Document Format）是最常用的文档格式之一。它具有跨平台、格式固定、不易篡改等优点，但也因此给自动化处理带来了挑战。本章将详细介绍如何使用Python实现PDF的自动化处理，包括合并、拆分、格式转换和表单填写等常见场景。

3.1 PDF处理的常见场景

在实际工作中，我们经常遇到以下PDF处理需求：

场景类型	具体需求	典型应用
文档整合	将多个PDF文件合并成一个	合同归档、报告汇总、发票整理
文档拆分	将多页PDF拆分成单页或多个文件	按章节拆分电子书、提取特定页面
格式转换	PDF转Word/Excel/图片	编辑PDF内容、数据提取分析
表单处理	自动填写PDF表单	批量生成合同、填写申请表
信息提取	提取PDF中的文字、表格	财务数据分析、文档检索

掌握这些自动化技能，可以大幅提升办公效率，减少重复性手工操作。

3.2 环境搭建

3.2.1 安装必要的库

Python生态中有多个优秀的PDF处理库，我们将主要使用以下几个：

- **PyPDF2**：用于PDF的合并、拆分、旋转等基础操作
- **pdfplumber**：用于从PDF中提取文字和表格
- **pdf2docx**：用于将PDF转换为Word文档
- **PyMuPDF (fitz)**：用于PDF转图片和高级操作
- **pdfrw**：用于处理PDF表单

打开命令行，执行以下安装命令：

```
# 基础PDF操作库
pip install PyPDF2

# PDF内容提取库
pip install pdfplumber

# PDF转Word
pip install pdf2docx

# PDF转图片和高级操作
pip install PyMuPDF

# PDF表单处理
pip install pdfrw

# 用于处理Excel（PDF转Excel时需要）
pip install openpyxl
```

3.2.2 验证安装

安装完成后，创建测试脚本验证环境是否正常：

```
# 验证PDF库安装
import PyPDF2
import pdfplumber
from pdf2docx import Converter
import fitz  # PyMuPDF
from pdfrw import PdfReader

print("✓ PyPDF2 版本:", PyPDF2.__version__)
print("✓ pdfplumber 安装成功")
print("✓ pdf2docx 安装成功")
print("✓ PyMuPDF (fitz) 安装成功")
print("✓ pdfrw 安装成功")
print("\n所有PDF处理库安装完成！")
```

运行后如果显示所有库版本信息，说明环境搭建成功。

3.3 实战案例

3.3.1 案例一：批量合并多个PDF

场景描述：将指定文件夹中的所有PDF文件按文件名排序后合并成一个PDF文档。

完整代码：

```
import os
from PyPDF2 import PdfMerger

def merge_pdfs(input_folder, output_file):
    """
    批量合并PDF文件

    参数:
        input_folder: 存放PDF文件的文件夹路径
        output_file: 合并后的输出文件名
    """
    # 创建PDF合并器对象
    merger = PdfMerger()

    # 获取文件夹中所有PDF文件，并按文件名排序
    pdf_files = [f for f in os.listdir(input_folder) if f.endswith('.pdf')]
    pdf_files.sort()  # 按文件名排序，确保合并顺序

    print(f"发现 {len(pdf_files)} 个PDF文件:")

    # 逐个添加PDF文件
    for pdf_file in pdf_files:
        pdf_path = os.path.join(input_folder, pdf_file)
        print(f"  正在合并: {pdf_file}")

        # 将PDF添加到合并器
        merger.append(pdf_path)

    # 写入合并后的文件
    merger.write(output_file)
    merger.close()

    print(f"\n✓ 合并完成！输出文件: {output_file}")
    print(f"✓ 总页数: {len(merger.pages) if hasattr(merger, 'pages') else '已

# ========== 使用示例 ==========
if __name__ == "__main__":
    # 设置输入文件夹和输出文件名
    INPUT_FOLDER = "./pdf_files"      # 存放源PDF的文件夹
```

```
OUTPUT_FILE = "./merged_output.pdf"  # 合并后的文件名

# 检查输入文件夹是否存在
if not os.path.exists(INPUT_FOLDER):
    print(f"错误：文件夹 '{INPUT_FOLDER}' 不存在")
    print("请创建该文件夹并放入PDF文件")
else:
    merge_pdfs(INPUT_FOLDER, OUTPUT_FILE)
```

代码说明：

- `PdfMerger()`：创建合并器对象
- `merger.append()`：添加PDF文件到合并队列
- `merger.write()`：将合并结果写入文件
- `pdf_files.sort()`：确保按文件名顺序合并

进阶功能：带书签的合并

```
import os
from PyPDF2 import PdfMerger

def merge_pdfs_with_bookmarks(input_folder, output_file):
    """
    合并PDF并添加书签（目录导航）
    """
    merger = PdfMerger()
    pdf_files = sorted([f for f in os.listdir(input_folder) if f.endswith('.p

    for pdf_file in pdf_files:
        pdf_path = os.path.join(input_folder, pdf_file)
        # 使用文件名（不含扩展名）作为书签
        bookmark_name = os.path.splitext(pdf_file)[0]
        merger.append(pdf_path, outline_item=bookmark_name)
        print(f"已添加书签: {bookmark_name}")

    merger.write(output_file)
    merger.close()
    print(f"\n✓ 带书签的PDF已保存: {output_file}")

# 使用示例
# merge_pdfs_with_bookmarks("./pdf_files", "./merged_with_bookmarks.pdf")
```

◆ 3.3.2 案例二：批量拆分PDF

场景描述：将一个多页PDF按页码范围拆分成多个文件，或提取指定页面。

完整代码：

```python
import os
from PyPDF2 import PdfReader, PdfWriter

def split_pdf_by_pages(input_file, output_folder, pages_per_file=1):
    """
    按页数拆分PDF

    参数:
        input_file: 源PDF文件路径
        output_folder: 输出文件夹
        pages_per_file: 每个输出文件包含的页数（默认1页=逐页拆分）
    """
    # 确保输出文件夹存在
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    # 读取源PDF
    reader = PdfReader(input_file)
    total_pages = len(reader.pages)

    print(f"源文件总页数: {total_pages}")
    print(f"每文件页数: {pages_per_file}")

    # 计算需要生成多少个文件
    num_files = (total_pages + pages_per_file - 1) // pages_per_file

    file_count = 0
    for i in range(0, total_pages, pages_per_file):
        writer = PdfWriter()

        # 计算当前文件的页码范围
        start_page = i
        end_page = min(i + pages_per_file, total_pages)

        # 添加页面到写入器
        for page_num in range(start_page, end_page):
            writer.add_page(reader.pages[page_num])

        # 生成输出文件名
```

```
            file_count += 1
            output_filename = f"page_{start_page+1:03d}_to_{end_page:03d}.pdf"
            output_path = os.path.join(output_folder, output_filename)

            # 保存文件
            with open(output_path, 'wb') as output_pdf:
                writer.write(output_pdf)

            print(f"  已生成: {output_filename} (第{start_page+1}-{end_page}页)")

    print(f"\n✓ 拆分完成！共生成 {file_count} 个文件")
    print(f"✓ 保存位置: {output_folder}")

def extract_specific_pages(input_file, output_file, page_numbers):
    """
    提取指定页码的页面

    参数:
        input_file: 源PDF文件路径
        output_file: 输出文件路径
        page_numbers: 要提取的页码列表（从1开始计数）
    """
    reader = PdfReader(input_file)
    writer = PdfWriter()

    print(f"正在提取页面: {page_numbers}")

    for page_num in page_numbers:
        # 转换为0-based索引
        index = page_num - 1
        if 0 <= index < len(reader.pages):
            writer.add_page(reader.pages[index])
            print(f"  ✓ 已添加第 {page_num} 页")
        else:
            print(f"  ✗ 第 {page_num} 页不存在（超出范围）")

    with open(output_file, 'wb') as output_pdf:
        writer.write(output_pdf)

    print(f"\n✓ 提取完成！保存为: {output_file}")
```

```
# ========== 使用示例 ==========
if __name__ == "__main__":
    INPUT_FILE = "./source.pdf"       # 源PDF文件
    OUTPUT_FOLDER = "./split_output"  # 输出文件夹

    # 示例1：逐页拆分
    # split_pdf_by_pages(INPUT_FILE, OUTPUT_FOLDER, pages_per_file=1)

    # 示例2：每5页拆分成一个文件
    # split_pdf_by_pages(INPUT_FILE, OUTPUT_FOLDER, pages_per_file=5)

    # 示例3：提取指定页面（如第1、3、5页）
    # extract_specific_pages(INPUT_FILE, "./extracted.pdf", [1, 3, 5])
```

代码说明：

- `PdfReader()`：读取PDF文件
- `PdfWriter()`：创建新的PDF写入器
- `writer.add_page()`：添加页面到输出文件
- 页码使用1-based（用户习惯），内部使用0-based（Python习惯）

◆ 3.3.3 案例三：PDF转Word/Excel

▸ 3.3.3.1 PDF转Word

场景描述：将PDF文档转换为可编辑的Word文档，保留原始格式。

完整代码：

```
import os
from pdf2docx import Converter

def pdf_to_word(input_pdf, output_docx=None):
    """
    将PDF转换为Word文档

    参数:
        input_pdf: 输入PDF文件路径
        output_docx: 输出Word文件路径（默认与PDF同名）
    """
    # 如果未指定输出文件名，自动生成
    if output_docx is None:
        output_docx = os.path.splitext(input_pdf)[0] + ".docx"

    print(f"开始转换: {input_pdf}")
    print(f"输出文件: {output_docx}")

    try:
        # 创建转换器对象
        cv = Converter(input_pdf)

        # 执行转换
        cv.convert(output_docx, start=0, end=None)
        cv.close()

        print(f"✓ 转换成功! ")

        # 显示文件大小
        pdf_size = os.path.getsize(input_pdf) / 1024
        docx_size = os.path.getsize(output_docx) / 1024
        print(f"  PDF大小: {pdf_size:.1f} KB")
        print(f"  Word大小: {docx_size:.1f} KB")

    except Exception as e:
        print(f"✗ 转换失败: {str(e)}")

def batch_pdf_to_word(input_folder, output_folder=None):
    """
```

```
    批量将文件夹中的PDF转换为Word

    参数:
        input_folder: 输入文件夹路径
        output_folder: 输出文件夹路径（默认与输入相同）
    """
    if output_folder is None:
        output_folder = input_folder

    # 确保输出文件夹存在
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    # 获取所有PDF文件
    pdf_files = [f for f in os.listdir(input_folder) if f.endswith('.pdf')]

    print(f"发现 {len(pdf_files)} 个PDF文件")
    print("=" * 40)

    for pdf_file in pdf_files:
        input_path = os.path.join(input_folder, pdf_file)
        output_path = os.path.join(output_folder,
                                   os.path.splitext(pdf_file)[0] + ".docx")

        print(f"\n处理: {pdf_file}")

        try:
            cv = Converter(input_path)
            cv.convert(output_path)
            cv.close()
            print(f"  ✓ 成功")
        except Exception as e:
            print(f"  ✗ 失败: {str(e)}")

    print("\n" + "=" * 40)
    print("批量转换完成! ")

# ========== 使用示例 ==========
if __name__ == "__main__":
    # 单个文件转换
```

```
    # pdf_to_word("./document.pdf")
    # pdf_to_word("./document.pdf", "./output.docx")

    # 批量转换
    # batch_pdf_to_word("./pdf_files", "./word_files")
```

3.3.3.2 PDF转Excel（提取表格）

场景描述：从PDF中提取表格数据并保存为Excel文件。

完整代码：

```
import pdfplumber
import pandas as pd
import os

def extract_tables_from_pdf(input_pdf, output_excel=None):
    """
    从PDF中提取表格并保存为Excel

    参数:
        input_pdf: 输入PDF文件路径
        output_excel: 输出Excel文件路径
    """
    if output_excel is None:
        output_excel = os.path.splitext(input_pdf)[0] + ".xlsx"

    print(f"正在分析PDF: {input_pdf}")

    all_tables = []

    # 打开PDF文件
    with pdfplumber.open(input_pdf) as pdf:
        print(f"总页数: {len(pdf.pages)}")

        for page_num, page in enumerate(pdf.pages, 1):
            print(f"\n正在处理第 {page_num} 页...")

            # 提取当前页的表格
            tables = page.extract_tables()

            if tables:
                print(f"  发现 {len(tables)} 个表格")

                for table_index, table in enumerate(tables):
                    # 将表格转换为DataFrame
                    if table and len(table) > 0:
                        # 第一行作为表头
                        df = pd.DataFrame(table[1:], columns=table[0])
                        df['来源页码'] = page_num
                        df['表格序号'] = table_index + 1
```

```
                        all_tables.append(df)
                        print(f"    表格 {table_index+1}: {len(df)} 行 x {len
            else:
                print(f"  该页无表格")

    # 保存到Excel
    if all_tables:
        with pd.ExcelWriter(output_excel, engine='openpyxl') as writer:
            # 每个表格保存为一个工作表
            for idx, df in enumerate(all_tables):
                sheet_name = f"表格_{idx+1}"
                df.to_excel(writer, sheet_name=sheet_name, index=False)

            # 同时创建一个汇总表
            combined_df = pd.concat(all_tables, ignore_index=True)
            combined_df.to_excel(writer, sheet_name='全部数据', index=False)

        print(f"\n✓ 提取完成！")
        print(f"  共提取 {len(all_tables)} 个表格")
        print(f"  保存位置: {output_excel}")
    else:
        print("\n✗ 未在PDF中发现表格")

def extract_text_to_excel(input_pdf, output_excel=None):
    """
    将PDF中的文字按页提取到Excel

    参数:
        input_pdf: 输入PDF文件路径
        output_excel: 输出Excel文件路径
    """
    if output_excel is None:
        output_excel = os.path.splitext(input_pdf)[0] + "_text.xlsx"

    data = []

    with pdfplumber.open(input_pdf) as pdf:
        for page_num, page in enumerate(pdf.pages, 1):
            text = page.extract_text()
            data.append({
```

```
                '页码': page_num,
                '内容': text if text else ''
            })

    # 保存到Excel
    df = pd.DataFrame(data)
    df.to_excel(output_excel, index=False)

    print(f"✓ 文字提取完成！")
    print(f"  共 {len(data)} 页")
    print(f"  保存位置: {output_excel}")

# ========== 使用示例 ==========
if __name__ == "__main__":
    # 提取表格
    # extract_tables_from_pdf("./report.pdf")
    # extract_tables_from_pdf("./report.pdf", "./output.xlsx")

    # 提取纯文本
    # extract_text_to_excel("./document.pdf")
```

代码说明：

- `pdfplumber.open()`：打开PDF文件
- `page.extract_tables()`：提取页面中的所有表格
- `page.extract_text()`：提取页面中的文字
- `pd.ExcelWriter()`：创建多工作表的Excel文件

◆ 3.3.4 案例四：自动填写PDF表单

场景描述：自动填写PDF表单字段，支持批量生成填好内容的PDF文件。

完整代码：

```python
from pdfrw import PdfReader, PdfWriter, PdfDict
import os
import json

def fill_pdf_form(template_pdf, output_pdf, data):
    """
    填写PDF表单

    参数:
        template_pdf: 表单模板PDF路径
        output_pdf: 输出PDF路径
        data: 表单数据字典，key为字段名，value为字段值
    """
    # 读取模板PDF
    template = PdfReader(template_pdf)

    # 获取第一个页面的表单字段
    if template.Root.AcroForm:
        fields = template.Root.AcroForm.Fields

        print(f"发现 {len(fields)} 个表单字段")

        # 遍历所有字段并填充数据
        for field in fields:
            field_name = field.T[1:-1] if field.T else "未知"  # 去掉括号

            if field_name in data:
                # 设置字段值
                field.update(PdfDict(V=data[field_name]))
                print(f"  ✓ 填写字段 '{field_name}': {data[field_name]}")
            else:
                print(f"  - 跳过字段 '{field_name}'（无对应数据）")
    else:
        print("警告：PDF中没有发现表单字段")

    # 保存填写后的PDF
    writer = PdfWriter()
    writer.write(output_pdf, template)
```

```
    print(f"\n✓ 表单填写完成！")
    print(f"  输出文件: {output_pdf}")

def list_form_fields(template_pdf):
    """
    列出PDF表单中的所有字段名

    参数:
        template_pdf: PDF表单模板路径
    """
    template = PdfReader(template_pdf)

    print(f"PDF表单字段列表:")
    print("=" * 40)

    if template.Root.AcroForm and template.Root.AcroForm.Fields:
        fields = template.Root.AcroForm.Fields
        for i, field in enumerate(fields, 1):
            field_name = field.T[1:-1] if field.T else "未知"
            field_type = field.FT[1:-1] if field.FT else "文本"
            print(f"{i}. {field_name} (类型: {field_type})")
    else:
        print("该PDF中没有表单字段")

    print("=" * 40)

def batch_fill_forms(template_pdf, data_list, output_folder):
    """
    批量填写PDF表单

    参数:
        template_pdf: 表单模板路径
        data_list: 数据列表，每个元素是一个字典
        output_folder: 输出文件夹
    """
    # 确保输出文件夹存在
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    print(f"开始批量填写，共 {len(data_list)} 份")
```

```
    print("=" * 40)

    for idx, data in enumerate(data_list, 1):
        output_pdf = os.path.join(output_folder, f"form_filled_{idx:03d}.pdf"
        print(f"\n[{idx}/{len(data_list)}] 处理中...")
        fill_pdf_form(template_pdf, output_pdf, data)

    print("\n" + "=" * 40)
    print(f"✓ 批量填写完成！文件保存在: {output_folder}")

# ========== 使用示例 ==========
if __name__ == "__main__":
    TEMPLATE_PDF = "./form_template.pdf"  # 表单模板文件

    # 示例1：列出所有表单字段
    # list_form_fields(TEMPLATE_PDF)

    # 示例2：填写单个表单
    # form_data = {
    #     "姓名": "张三",
    #     "电话": "13800138000",
    #     "邮箱": "zhangsan@example.com",
    #     "部门": "技术部",
    #     "日期": "2024-01-15"
    # }
    # fill_pdf_form(TEMPLATE_PDF, "./filled_form.pdf", form_data)

    # 示例3：批量填写表单
    # batch_data = [
    #     {"姓名": "张三", "电话": "13800138001", "部门": "技术部"},
    #     {"姓名": "李四", "电话": "13800138002", "部门": "市场部"},
    #     {"姓名": "王五", "电话": "13800138003", "部门": "财务部"},
    # ]
    # batch_fill_forms(TEMPLATE_PDF, batch_data, "./filled_forms")
```

代码说明：

- `PdfReader()`：读取PDF模板
- `template.Root.AcroForm.Fields`：获取表单字段列表
- `field.update(PdfDict(V=value))`：设置字段值

- `PdfWriter().write()`：保存填写后的PDF

使用提示：

1. 首先需要使用Adobe Acrobat或其他工具创建带表单字段的PDF模板
2. 使用`list_form_fields()`函数查看PDF中的所有字段名
3. 确保数据字典的key与PDF中的字段名完全匹配

3.4 常见问题与解决方案

问题1：合并后的PDF文件损坏

原因：某些PDF使用了特殊编码或加密 **解决**：使用PyPDF2的`strict=False`参数，或先用其他工具修复PDF

问题2：提取的表格数据错乱

原因：PDF中的表格可能是图片形式，或格式不规范 **解决**：结合OCR技术识别图片表格，或使用更复杂的表格识别算法

问题3：PDF转Word后格式丢失

原因：PDF的复杂布局难以完全保留 **解决**：转换后手动调整，或使用专业转换软件作为备选

◆ 问题4：表单字段无法识别

原因：PDF中的"表单"可能只是图形，不是真正的AcroForm **解决**：使用Adobe Acrobat检查表单类型，或使用OCR+坐标定位的方式填写

✦ 3.5 本章小结

本章介绍了Python处理PDF的核心技术：

1. **环境搭建**：安装了PyPDF2、pdfplumber、pdf2docx等核心库
2. **PDF合并**：使用PdfMerger实现多文件合并，支持书签功能
3. **PDF拆分**：使用PdfReader和PdfWriter实现按页拆分和指定页面提取
4. **格式转换**：使用pdf2docx转Word，使用pdfplumber提取表格转Excel
5. **表单填写**：使用pdfrw自动填写PDF表单字段

掌握这些技能后，你可以轻松应对日常办公中的PDF处理需求，大幅提升工作效率。建议将本章代码保存为工具脚本，需要时直接调用即可。

练习建议：

1. 准备几个测试PDF文件，运行本章所有案例代码
2. 尝试修改代码参数，如每5页拆分、提取特定页面范围等
3. 结合实际工作需求，封装自己的PDF处理工具函数

第4章 邮件自动化

4.1 用Python发送邮件的优势

在日常办公中，邮件是最常用的沟通工具之一。无论是发送工作报告、通知公告，还是与客户保持联系，邮件都扮演着重要角色。然而，当需要发送大量邮件时，手动操作不仅耗时耗力，还容易出错。Python邮件自动化正是解决这一痛点的利器。

为什么选择Python进行邮件自动化?

1. 高效批量处理

传统方式发送100封邮件可能需要数小时，而Python脚本可以在几分钟内完成。通过循环和模板技术，轻松实现批量邮件发送。

2. 内容个性化

Python支持变量替换和动态内容生成，可以为每封邮件定制收件人姓名、公司信息等个性化内容，提升邮件的专业度和打开率。

3. 附件自动化

批量添加附件、自动匹配文件与收件人，这些繁琐的操作在Python中只需几行代码即可实现。

4. 定时与触发机制

结合操作系统的定时任务或事件触发，可以实现邮件的自动发送，如每日报表、生日祝福等场景。

5. 邮件读取与处理

不仅能发送邮件，Python还能自动读取收件箱，实现邮件分类、自动回复、信息提取等高级功能。

6. 零成本

Python标准库自带邮件处理模块，无需额外购买软件，降低办公成本。

✦ 4.2 环境搭建：smtplib/email安装

好消息是，Python进行邮件自动化所需的核心模块都是标准库的一部分，无需额外安装！

◆ 核心模块介绍

模块	用途
`smtplib`	负责与邮件服务器通信，发送邮件
`email`	构建邮件内容，支持文本、HTML、附件等
`imaplib`	读取和管理收件箱邮件

◆ 验证环境

打开终端或命令提示符，运行以下代码验证环境：

```
import smtplib
import email
import imaplib

print("✅ smtplib 版本:", smtplib.__version__ if hasattr(smtplib, '__version_
print("✅ email 模块加载成功")
print("✅ imaplib 模块加载成功")
```

如果没有任何报错，说明环境已经准备就绪！

◆ 邮件服务器配置

发送邮件需要SMTP服务器信息，常见邮箱配置如下：

邮箱服务商	SMTP服务器	端口	安全协议
QQ邮箱	smtp.qq.com	465/587	SSL/TLS
163邮箱	smtp.163.com	465	SSL
Gmail	smtp.gmail.com	587	TLS
Outlook	smtp.office365.com	587	TLS
企业微信邮箱	smtp.exmail.qq.com	465	SSL

◆ 获取授权码

大多数邮箱需要使用**授权码**而非登录密码：

1. **QQ邮箱**：设置 → 账户 → 开启SMTP服务 → 获取授权码
2. **163邮箱**：设置 → POP3/SMTP/IMAP → 开启服务 → 获取授权码
3. **Gmail**：需要开启两步验证后生成应用专用密码

4.3 实战案例

案例1：批量发送邮件

场景：向50位客户发送产品推广邮件

```python
import smtplib
from email.mime.text import MIMEText
from email.header import Header

# ==================== 配置区域 ====================
SMTP_SERVER = "smtp.qq.com"      # SMTP服务器地址
SMTP_PORT = 465                  # SMTP端口（SSL）
SENDER_EMAIL = "your_email@qq.com"  # 发件人邮箱
AUTH_CODE = "your_auth_code"      # 邮箱授权码（非登录密码）
# ================================================

def send_email(receiver, subject, content):
    """
    发送单封邮件

    参数:
        receiver: 收件人邮箱地址
        subject: 邮件主题
        content: 邮件正文（纯文本）
    """
    # 创建邮件对象
    msg = MIMEText(content, 'plain', 'utf-8')
    msg['From'] = Header(SENDER_EMAIL)
    msg['To'] = Header(receiver)
    msg['Subject'] = Header(subject, 'utf-8')

    try:
        # 连接SMTP服务器（SSL加密）
        server = smtplib.SMTP_SSL(SMTP_SERVER, SMTP_PORT)
        server.login(SENDER_EMAIL, AUTH_CODE)

        # 发送邮件
        server.sendmail(SENDER_EMAIL, [receiver], msg.as_string())
        server.quit()

        print(f"✅ 邮件发送成功: {receiver}")
        return True

    except Exception as e:
```

```
        print(f"❌ 发送失败 {receiver}: {str(e)}")
        return False


def batch_send_emails():
    """批量发送邮件主函数"""

    # 收件人列表
    recipients = [
        "client1@example.com",
        "client2@example.com",
        "client3@example.com",
        # ... 更多收件人
    ]

    # 邮件内容
    subject = "【新品上市】智能办公助手限时优惠"
    content = """尊敬的客户：

您好！感谢您一直以来对我们的支持与信任。

我们很高兴地通知您，全新智能办公助手已正式上线！

🎁 限时优惠：前100名购买用户享受8折优惠
🗓️ 活动时间：即日起至本月底

如需了解更多详情，请回复此邮件或致电：400-XXX-XXXX

祝商祺！
智能办公团队
"""

    # 批量发送
    success_count = 0
    for email_addr in recipients:
        if send_email(email_addr, subject, content):
            success_count += 1

    print(f"\n📊 发送完成：成功 {success_count}/{len(recipients)}")
```

```
# 执行批量发送
if __name__ == "__main__":
    batch_send_emails()
```

代码要点解析：

- `MIMEText` 用于创建纯文本邮件，支持指定编码避免中文乱码
- `SMTP_SSL` 建立加密连接，保护邮件内容安全
- 批量发送时建议添加延时（`time.sleep(1)`），避免触发反垃圾机制

◆ 案例2：邮件内容个性化（变量替换）

场景：给客户发送个性化促销邮件，包含客户姓名和专属优惠码

```
import smtplib
from email.mime.text import MIMEText
from email.header import Header
import time

# ==================== 配置区域 ====================
SMTP_SERVER = "smtp.qq.com"
SMTP_PORT = 465
SENDER_EMAIL = "your_email@qq.com"
AUTH_CODE = "your_auth_code"
# ================================================

def generate_personalized_content(template, variables):
    """
    使用变量替换生成个性化邮件内容

    参数:
        template: 邮件模板，使用 {变量名} 作为占位符
        variables: 字典，包含变量名和对应值

    返回:
        替换后的邮件内容
    """
    try:
        content = template.format(**variables)
        return content
    except KeyError as e:
        print(f"❌ 模板变量缺失: {e}")
        return None


def send_personalized_email(recipient_info):
    """
    发送个性化邮件

    参数:
        recipient_info: 字典，包含收件人信息
            - email: 邮箱地址
            - name: 客户姓名
```

```
            - company: 公司名称
            - discount_code: 专属优惠码
            - discount_rate: 折扣力度
    """
    # 邮件模板
    template = """尊敬的 {name} 先生/女士：

您好！感谢您和 {company} 一直以来对我们的支持。

🎉 专属福利来袭！

作为我们的重要合作伙伴，我们特别为您准备了专属优惠：

━━━━━━━━━━━━━━━━━━━
🎁 您的专属优惠码：{discount_code}
💰 优惠力度：{discount_rate}
⏰ 有效期：30天
━━━━━━━━━━━━━━━━━━━

此优惠码仅限您本人使用，请勿分享给他人。

如有任何问题，请随时联系您的专属客服。

祝生意兴隆！

智能办公团队
客服热线：400-XXX-XXXX
"""

    # 生成个性化内容
    content = generate_personalized_content(template, recipient_info)
    if not content:
        return False

    # 个性化主题
    subject = f"【专属优惠】{recipient_info['name']}，您的{recipient_info['dis

    # 创建邮件对象
    msg = MIMEText(content, 'plain', 'utf-8')
    msg['From'] = Header(f"智能办公团队 <{SENDER_EMAIL}>", 'utf-8')
```

```
    msg['To'] = Header(recipient_info['email'])
    msg['Subject'] = Header(subject, 'utf-8')

    try:
        server = smtplib.SMTP_SSL(SMTP_SERVER, SMTP_PORT)
        server.login(SENDER_EMAIL, AUTH_CODE)
        server.sendmail(SENDER_EMAIL, [recipient_info['email']], msg.as_strin
        server.quit()

        print(f"✅ 已发送给: {recipient_info['name']} ({recipient_info['email
        return True

    except Exception as e:
        print(f"❌ 发送失败: {recipient_info['email']} - {str(e)}")
        return False


def batch_personalized_emails():
    """批量发送个性化邮件"""

    # 客户数据（实际应用中可从Excel/数据库读取）
    customers = [
        {
            "email": "zhangsan@company-a.com",
            "name": "张三",
            "company": "A科技有限公司",
            "discount_code": "VIP-A001",
            "discount_rate": "8折"
        },
        {
            "email": "lisi@company-b.com",
            "name": "李四",
            "company": "B贸易集团",
            "discount_code": "VIP-B002",
            "discount_rate": "7.5折"
        },
        {
            "email": "wangwu@company-c.com",
            "name": "王五",
            "company": "C咨询公司",
```

```
            "discount_code": "VIP-C003",
            "discount_rate": "8.5折"
        }
    ]

    # 批量发送，每封间隔2秒
    for customer in customers:
        send_personalized_email(customer)
        time.sleep(2)  # 延时避免触发反垃圾机制


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

进阶技巧：

- 使用 `string.Template` 提供更安全的变量替换
- 从CSV/Excel文件读取客户数据，实现真正的批量处理
- 添加HTML模板支持，发送更美观的富文本邮件

◆ 案例3：批量添加附件

场景：向各部门发送月度报表，每个部门收到对应的PDF文件

```python
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from email.mime.base import MIMEBase
from email.header import Header
from email import encoders
import os

# ==================== 配置区域 ====================
SMTP_SERVER = "smtp.qq.com"
SMTP_PORT = 465
SENDER_EMAIL = "your_email@qq.com"
AUTH_CODE = "your_auth_code"
# ================================================

def attach_file(msg, file_path):
    """
    为邮件添加附件

    参数:
        msg: MIMEMultipart邮件对象
        file_path: 附件文件的完整路径
    """
    if not os.path.exists(file_path):
        print(f"⚠️ 文件不存在: {file_path}")
        return False

    # 获取文件名
    filename = os.path.basename(file_path)

    # 读取文件内容
    with open(file_path, 'rb') as f:
        attachment = MIMEBase('application', 'octet-stream')
        attachment.set_payload(f.read())

    # Base64编码
    encoders.encode_base64(attachment)

    # 设置附件头信息（支持中文文件名）
```

```
        attachment.add_header(
            'Content-Disposition',
            f'attachment; filename="{filename}"'
        )

        # 添加到邮件
        msg.attach(attachment)
        print(f"📎 已添加附件: {filename}")
        return True


def send_email_with_attachments(receiver, subject, content, attachments):
    """
    发送带附件的邮件

    参数:
        receiver: 收件人邮箱
        subject: 邮件主题
        content: 邮件正文
        attachments: 附件路径列表
    """
    # 创建多部分邮件对象
    msg = MIMEMultipart()
    msg['From'] = Header(SENDER_EMAIL)
    msg['To'] = Header(receiver)
    msg['Subject'] = Header(subject, 'utf-8')

    # 添加邮件正文
    msg.attach(MIMEText(content, 'plain', 'utf-8'))

    # 添加附件
    attached_count = 0
    for file_path in attachments:
        if attach_file(msg, file_path):
            attached_count += 1

    try:
        server = smtplib.SMTP_SSL(SMTP_SERVER, SMTP_PORT)
        server.login(SENDER_EMAIL, AUTH_CODE)
        server.sendmail(SENDER_EMAIL, [receiver], msg.as_string())
```

```
        server.quit()

        print(f"✅ 邮件发送成功 ({attached_count}个附件): {receiver}")
        return True

    except Exception as e:
        print(f"❌ 发送失败: {receiver} - {str(e)}")
        return False


def batch_send_reports():
    """批量发送部门报表"""

    # 部门报表配置
    reports_config = [
        {
            "department": "销售部",
            "email": "sales@company.com",
            "files": ["reports/销售部_月度报表.pdf", "reports/销售数据_图表.xls
        },
        {
            "department": "市场部",
            "email": "marketing@company.com",
            "files": ["reports/市场部_月度报表.pdf", "reports/推广活动总结.pptx
        },
        {
            "department": "技术部",
            "email": "tech@company.com",
            "files": ["reports/技术部_月度报表.pdf"]
        }
    ]

    # 邮件模板
    template = """{department} 同事:

您好！附件是{department}的月度工作报告，请查收。

报告包含以下内容:
- 本月工作总结
- 数据分析与图表
```

```
- 下月工作计划

如有疑问，请联系行政部。

行政部
"""

    # 批量发送
    for config in reports_config:
        content = template.format(department=config["department"])
        subject = f"【月度报表】{config['department']} - {os.path.basename(cor

        send_email_with_attachments(
            config["email"],
            subject,
            content,
            config["files"]
        )


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

附件处理技巧：

- 使用 `MIMEMultipart` 支持多部分内容（正文+附件）
- 大文件建议压缩后再发送
- 可使用 `mimetypes` 模块自动识别文件类型

◆ 案例4：自动读取回复邮件

场景：自动读取收件箱，提取客户反馈邮件并分类处理

```
import imaplib
import email
from email.header import decode_header
import re

# ==================== 配置区域 ====================
IMAP_SERVER = "imap.qq.com"       # IMAP服务器地址
IMAP_PORT = 993                    # IMAP端口（SSL）
EMAIL_ACCOUNT = "your_email@qq.com"
AUTH_CODE = "your_auth_code"
# ================================================

def decode_email_header(header_value):
    """
    解码邮件头（处理中文乱码）

    参数:
        header_value: 邮件头字符串
    返回:
        解码后的字符串
    """
    if not header_value:
        return ""

    decoded_parts = decode_header(header_value)
    result = []

    for part, charset in decoded_parts:
        if isinstance(part, bytes):
            result.append(part.decode(charset or 'utf-8', errors='ignore'))
        else:
            result.append(part)

    return ''.join(result)


def get_email_body(msg):
    """
    提取邮件正文内容
```

```
    参数:
        msg: email.message.Message对象
    返回:
        邮件正文文本
    """
    body = ""

    if msg.is_multipart():
        # 遍历所有部分
        for part in msg.walk():
            content_type = part.get_content_type()
            content_disposition = str(part.get("Content-Disposition", ""))

            # 跳过附件
            if "attachment" in content_disposition:
                continue

            # 提取文本内容
            if content_type == "text/plain":
                try:
                    payload = part.get_payload(decode=True)
                    charset = part.get_content_charset() or 'utf-8'
                    body = payload.decode(charset, errors='ignore')
                    break
                except:
                    continue
            elif content_type == "text/html" and not body:
                try:
                    payload = part.get_payload(decode=True)
                    charset = part.get_content_charset() or 'utf-8'
                    body = payload.decode(charset, errors='ignore')
                except:
                    continue
    else:
        # 单部分邮件
        try:
            payload = msg.get_payload(decode=True)
            charset = msg.get_content_charset() or 'utf-8'
            body = payload.decode(charset, errors='ignore')
```

```
        except:
            body = str(msg.get_payload())

    return body


def classify_email(subject, body, sender):
    """
    根据内容对邮件进行分类

    参数:
        subject: 邮件主题
        body: 邮件正文
        sender: 发件人
    返回:
        分类标签
    """
    text = (subject + " " + body).lower()

    # 关键词匹配分类
    if any(kw in text for kw in ['投诉', '不满意', '退款', '差评', '问题']):
        return "投诉反馈"
    elif any(kw in text for kw in ['咨询', '询问', '怎么', '如何', '多少钱']):
        return "产品咨询"
    elif any(kw in text for kw in ['合作', '代理', '加盟', '商务']):
        return "商务合作"
    elif any(kw in text for kw in ['简历', '应聘', '求职', '招聘']):
        return "招聘相关"
    elif any(kw in text for kw in ['感谢', '好评', '满意', '不错']):
        return "正面反馈"
    else:
        return "其他"


def fetch_and_process_emails(limit=10):
    """
    读取并处理邮件

    参数:
        limit: 最多读取的邮件数量
```

```
    """
    try:
        # 连接IMAP服务器
        mail = imaplib.IMAP4_SSL(IMAP_SERVER, IMAP_PORT)
        mail.login(EMAIL_ACCOUNT, AUTH_CODE)

        # 选择收件箱
        mail.select("inbox")

        # 搜索未读邮件（UNSEEN）或所有邮件（ALL）
        status, messages = mail.search(None, "UNSEEN")

        if status != "OK":
            print("❌ 搜索邮件失败")
            return

        email_ids = messages[0].split()

        if not email_ids:
            print("📭 没有新邮件")
            return

        print(f"📬 发现 {len(email_ids)} 封新邮件\n")

        # 处理每封邮件
        for i, email_id in enumerate(email_ids[:limit]):
            status, msg_data = mail.fetch(email_id, "(RFC822)")

            if status != "OK":
                continue

            # 解析邮件
            raw_email = msg_data[0][1]
            msg = email.message_from_bytes(raw_email)

            # 提取信息
            subject = decode_email_header(msg["Subject"])
            sender = decode_email_header(msg["From"])
            date = msg["Date"]
            body = get_email_body(msg)
```

```
            # 分类
            category = classify_email(subject, body, sender)

            # 输出结果
            print(f"{'='*50}")
            print(f"📧 邮件 {i+1}")
            print(f"{'='*50}")
            print(f"📌 分类: {category}")
            print(f"📨 发件人: {sender}")
            print(f"📋 主题: {subject}")
            print(f"📅 日期: {date}")
            print(f"📝 内容预览: {body[:200]}...")
            print()

            # 根据分类执行不同操作
            if category == "投诉反馈":
                print("⚠️  标记为紧急 - 需要立即处理")
                # 可以在这里添加自动回复或转发给客服
            elif category == "商务合作":
                print("💼 转发给商务部门")

        # 关闭连接
        mail.close()
        mail.logout()

    except Exception as e:
        print(f"❌ 错误: {str(e)}")


if __name__ == "__main__":
    fetch_and_process_emails(limit=5)
```

邮件读取进阶功能:

- 标记已读/未读状态
- 移动邮件到指定文件夹
- 下载附件保存到本地
- 结合NLP进行更智能的内容分析

第5章 文件管理自动化

5.1 文件管理的痛点

在日常工作中，文件管理是许多人面临的难题：

常见痛点：

1. **文件命名混乱** - `新建文本文档(1).txt`、`新建文本文档(2).txt` 等无意义命名
2. **文件散落各处** - 文档、图片、下载文件混杂在同一文件夹
3. **重复文件占用空间** - 同一份文件多次下载，占用大量存储空间
4. **重要文件丢失风险** - 缺乏备份机制，误删或硬盘故障导致数据丢失
5. **查找困难** - 需要某个文件时，在众多文件夹中翻找耗时

Python解决方案：

痛点	Python解决方案	实现难度
命名混乱	批量重命名	⭐
文件散乱	按类型自动分类	⭐⭐
重复文件	哈希值比对去重	⭐⭐⭐
数据丢失	定时自动备份	⭐⭐

5.2 实战案例

案例1：批量重命名文件

场景：将下载的图片从 `IMG_20240101_001.jpg` 重命名为 `产品展示_001.jpg`

```
import os
import re
from pathlib import Path

def batch_rename(directory, pattern=None, prefix="", suffix="", start_num=1):
    """
    批量重命名文件

    参数:
        directory: 目标文件夹路径
        pattern: 正则表达式，匹配需要重命名的文件（None表示全部）
        prefix: 新文件名前缀
        suffix: 新文件名后缀（保留原扩展名）
        start_num: 起始编号

    返回:
        重命名成功的文件数量
    """
    if not os.path.exists(directory):
        print(f"❌ 目录不存在: {directory}")
        return 0

    # 获取所有文件
    files = [f for f in os.listdir(directory) if os.path.isfile(os.path.join(

    # 如果指定了匹配模式，进行过滤
    if pattern:
        regex = re.compile(pattern)
        files = [f for f in files if regex.match(f)]

    if not files:
        print("⚠️ 没有匹配的文件")
        return 0

    print(f"📁 找到 {len(files)} 个文件待重命名\n")

    renamed_count = 0

    for i, filename in enumerate(files, start=start_num):
```

```
        old_path = os.path.join(directory, filename)

        # 获取文件扩展名
        _, ext = os.path.splitext(filename)

        # 生成新文件名
        new_filename = f"{prefix}{i:03d}{suffix}{ext}"
        new_path = os.path.join(directory, new_filename)

        # 避免重名
        counter = 1
        while os.path.exists(new_path):
            new_filename = f"{prefix}{i:03d}_{counter}{suffix}{ext}"
            new_path = os.path.join(directory, new_filename)
            counter += 1

        try:
            os.rename(old_path, new_path)
            print(f"✅ {filename} → {new_filename}")
            renamed_count += 1
        except Exception as e:
            print(f"❌ 重命名失败 {filename}: {str(e)}")

    print(f"\n📊 完成: {renamed_count}/{len(files)} 个文件已重命名")
    return renamed_count


def rename_by_pattern(directory, old_pattern, new_pattern):
    """
    根据正则表达式替换文件名中的特定模式

    参数:
        directory: 目标文件夹
        old_pattern: 要替换的正则模式
        new_pattern: 替换后的字符串（可使用\1, \2等引用分组）
    """
    if not os.path.exists(directory):
        print(f"❌ 目录不存在: {directory}")
        return
```

```
    regex = re.compile(old_pattern)
    files = [f for f in os.listdir(directory) if os.path.isfile(os.path.join(

    renamed_count = 0

    for filename in files:
        if regex.search(filename):
            new_filename = regex.sub(new_pattern, filename)
            old_path = os.path.join(directory, filename)
            new_path = os.path.join(directory, new_filename)

            try:
                os.rename(old_path, new_path)
                print(f"✅ {filename} → {new_filename}")
                renamed_count += 1
            except Exception as e:
                print(f"❌ 失败: {filename} - {str(e)}")

    print(f"\n📊 完成: {renamed_count} 个文件已重命名")


# ==================== 使用示例 ====================

if __name__ == "__main__":
    # 示例1: 简单批量重命名
    # 将 download 文件夹中的图片重命名为 产品展示_001.jpg, 产品展示_002.jpg...
    batch_rename(
        directory="./download",
        pattern=r".*\.(jpg|jpeg|png|gif){{CONTENT}}quot;,  # 只匹配图片文件
        prefix="产品展示_",
        start_num=1
    )

    # 示例2: 根据正则替换文件名
    # 将 "IMG_20240101_001.jpg" 改为 "2024-01-01_001.jpg"
    # rename_by_pattern(
    #     directory="./photos",
    #     old_pattern=r"IMG_(\d{4})(\d{2})(\d{2})_",
```

```
    #     new_pattern=r"\1-\2-\3_"
    # )
```

进阶功能扩展：

- 添加日期时间戳到文件名
- 根据EXIF信息重命名照片
- 支持递归处理子文件夹

◆ 案例2：按类型自动分类

场景：将下载文件夹中的文件按类型（文档、图片、视频等）自动分类到不同文件夹

```
import os
import shutil
from pathlib import Path
from collections import defaultdict

# 文件类型分类规则
FILE_CATEGORIES = {
    "文档": [".doc", ".docx", ".pdf", ".txt", ".rtf", ".odt", ".xls", ".xlsx"
    "图片": [".jpg", ".jpeg", ".png", ".gif", ".bmp", ".svg", ".webp", ".ico"
    "视频": [".mp4", ".avi", ".mkv", ".mov", ".wmv", ".flv", ".webm", ".m4v"]
    "音频": [".mp3", ".wav", ".flac", ".aac", ".ogg", ".m4a", ".wma"],
    "压缩包": [".zip", ".rar", ".7z", ".tar", ".gz", ".bz2"],
    "程序": [".exe", ".msi", ".dmg", ".pkg", ".deb", ".rpm"],
    "代码": [".py", ".js", ".html", ".css", ".java", ".cpp", ".c", ".h", ".ph
    "其他": []
}


def get_file_category(filename):
    """
    根据扩展名判断文件类别

    参数:
        filename: 文件名
    返回:
        类别名称
    """
    ext = os.path.splitext(filename)[1].lower()

    for category, extensions in FILE_CATEGORIES.items():
        if ext in extensions:
            return category

    return "其他"


def organize_files(source_dir, target_dir=None, copy=False):
    """
    按文件类型自动分类整理
```

```
    参数:
        source_dir: 源文件夹路径
        target_dir: 目标文件夹路径（默认与源文件夹相同）
        copy: True表示复制，False表示移动

    返回:
        分类统计字典
    """
    if not os.path.exists(source_dir):
        print(f"❌ 源目录不存在: {source_dir}")
        return {}

    if target_dir is None:
        target_dir = source_dir

    # 确保目标目录存在
    os.makedirs(target_dir, exist_ok=True)

    # 统计信息
    stats = defaultdict(int)
    operation = "复制" if copy else "移动"

    # 遍历源目录
    for filename in os.listdir(source_dir):
        file_path = os.path.join(source_dir, filename)

        # 跳过文件夹
        if os.path.isdir(file_path):
            continue

        # 获取文件类别
        category = get_file_category(filename)

        # 创建类别文件夹
        category_folder = os.path.join(target_dir, category)
        os.makedirs(category_folder, exist_ok=True)

        # 目标路径
        target_path = os.path.join(category_folder, filename)
```

```
        # 处理重名
        counter = 1
        original_target = target_path
        while os.path.exists(target_path):
            name, ext = os.path.splitext(filename)
            target_path = os.path.join(category_folder, f"{name}_{counter}{ex
            counter += 1

        try:
            if copy:
                shutil.copy2(file_path, target_path)
            else:
                shutil.move(file_path, target_path)

            stats[category] += 1
            print(f"✅ [{category}] {filename}")

        except Exception as e:
            print(f"❌ 失败: {filename} - {str(e)}")

    # 输出统计
    print(f"\n{'='*40}")
    print(f"📊 文件整理完成 ({operation}模式)")
    print(f"{'='*40}")
    total = sum(stats.values())
    for category, count in sorted(stats.items(), key=lambda x: -x[1]):
        print(f"  {category}: {count} 个文件")
    print(f"{'='*40}")
    print(f"  总计: {total} 个文件")

    return dict(stats)


# ==================== 使用示例 ====================

if __name__ == "__main__":
    # 整理下载文件夹
    organize_files(
        source_dir="./Downloads",
```

```
        target_dir="./Downloads/已整理",
        copy=False  # 移动文件
    )
```

扩展功能：

- 支持按日期分类（年/月）
- 支持自定义分类规则
- 支持递归处理子文件夹

◆ 案例3：自动清理重复文件

场景：扫描文件夹，找出并删除重复文件，释放存储空间

```
import os
import hashlib
from collections import defaultdict

def calculate_file_hash(file_path, algorithm='md5', block_size=65536):
    """
    计算文件哈希值

    参数:
        file_path: 文件路径
        algorithm: 哈希算法（md5, sha1, sha256）
        block_size: 读取块大小

    返回:
        文件哈希值字符串
    """
    hash_obj = hashlib.new(algorithm)

    try:
        with open(file_path, 'EOF

---
```

第6章 AI增强办公

6.1 Python + AI = 办公自动化2.0

在AI时代，不会编程不再是障碍。有了Claude Code、ChatGPT等AI助手，你只需要描述需求，AI就能帮你写出专业的Python代码。

6.2 如何用AI生成Python代码

6.2.1 向AI描述需求

清晰描述你的办公自动化需求：

- 要处理什么文件？
- 要完成什么操作？
- 输出什么结果？

示例提示词：

```
请帮我写一个Python脚本，实现以下功能：
1. 读取文件夹中的所有Excel文件
2. 提取每个文件的"销售额"列
3. 汇总到一个新的Excel文件中
4. 生成柱状图展示各文件销售额对比
```

6.2.2 AI生成代码

AI会根据你的描述生成完整代码，包含：

- 必要的库导入
- 详细的注释说明
- 错误处理机制
- 使用示例

6.2.3 运行和调试

1. 复制AI生成的代码
2. 保存为.py文件
3. 根据实际情况修改文件路径等参数
4. 运行测试

6.3 实战案例

案例一：AI辅助写Excel处理脚本

需求描述：

> 帮我写一个Python脚本，批量处理Excel文件：
> - 读取指定文件夹中的所有.xlsx文件
> - 删除空行和重复数据
> - 添加一列"处理日期"
> - 保存到新的文件夹

AI生成的核心代码：

```python
import pandas as pd
import os
from datetime import datetime

# 输入输出文件夹
input_folder = '原始数据'
output_folder = '处理后数据'

# 确保输出文件夹存在
if not os.path.exists(output_folder):
    os.makedirs(output_folder)

# 批量处理
for filename in os.listdir(input_folder):
    if filename.endswith('.xlsx'):
        # 读取文件
        df = pd.read_excel(os.path.join(input_folder, filename))

        # 删除空行
        df = df.dropna(how='all')

        # 删除重复行
        df = df.drop_duplicates()

        # 添加处理日期列
        df['处理日期'] = datetime.now().strftime('%Y-%m-%d')

        # 保存
        output_path = os.path.join(output_folder, filename)
        df.to_excel(output_path, index=False)
        print(f'已处理: {filename}')
```

◆ 案例二：全自动日报生成系统

完整流程：

1. 自动读取数据源（Excel/数据库）

2. AI分析数据生成文字总结
3. 自动生成图表
4. 自动发送邮件给领导

核心代码框架：

```
import pandas as pd
import matplotlib.pyplot as plt
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.base import MIMEBase
from email import encoders

# 1. 读取数据
df = pd.read_excel('销售数据.xlsx')

# 2. 数据分析
summary = f"""
今日销售日报：
- 总销售额：{df['销售额'].sum():,.0f}元
- 订单数量：{len(df)}单
- 平均客单价：{df['销售额'].mean():,.0f}元
- 最佳销售产品：{df.groupby('产品')['销售额'].sum().idxmax()}
"""

# 3. 生成图表
plt.figure(figsize=(10, 6))
df.groupby('产品')['销售额'].sum().plot(kind='bar')
plt.title('各产品销售额对比')
plt.savefig('日报图表.png')

# 4. 发送邮件（代码见第4章）
```

6.4 未来趋势

6.4.1 AI Agent在办公场景的应用

未来的办公自动化将更加智能：

- **自主决策**：AI Agent可以自主判断如何处理不同类型的文件
- **持续学习**：根据用户的反馈不断优化处理流程
- **多工具协作**：自动调用Excel、PDF、邮件等多种工具完成任务

6.4.2 低代码/无代码平台

- 可视化流程设计
- 拖拽式组件组装
- 自然语言描述需求

6.5 本章小结

核心要点：

1. AI可以帮你写Python代码，降低编程门槛
2. 清晰描述需求是获得好代码的关键
3. 从简单任务开始，逐步构建复杂自动化流程
4. 未来是AI + 人类协作的时代

下一步行动：

- 选择本章一个案例，用AI生成代码并运行
- 根据自己的工作场景，向AI描述需求
- 持续优化你的自动化脚本

结语：Python + AI = 办公自动化2.0

在这个AI时代，不会编程不再是障碍。有了Claude Code、ChatGPT等AI助手，你只需要描述需求，AI就能帮你写出专业的Python代码。

从今天开始，让Python和AI成为你的办公助手，把重复性工作交给程序，把创造力留给自己!