目录

1 实验介绍

1.1 实验内容

本次实验包含 3 个练习:

1. 给定任意网页内容, 返回网页中所有超链接的 URL（不包括图片地址）, 并将结果打印至文件 res1.txt 中, 每一行为一个链接地址.

2. 给定任意网页内容, 返回网页中所有图片地址, 并将结果打印至文件 res2.txt 中, 每一行为一个图片地址.

3. 给定糗事百科有图有真相任意一页内容, 返回网页中图片和相应文本, 以及下一页的网址, 并将图片地址与相应文本以下述格式打印至文件 res3.txt 中, 每一行对应一个图片地址与相应文本, 格式为: 图片地址 \t 相应文本.

1.2 实验环境

操作系统: Ubuntu 18.04 LTS

Python 版本: 3.7

1.3 所需技能

本实验的编程部分主要使用 Python 语言, 利用爬虫技术获取网页内容, 进而按照一定规则进行解析, 提取出所需信息. 其中涉及到 Python 的 Requests 库 (用以模拟网络请求),BeautifulSoup 库 (用以解析网页), 以及其他的一些库. 当然, 对于 Linux 系统及其命令操作的基本了解, 以及网络相关的知识, 也有助于更好更快地完成本次实验.

2 实验过程

2.1 准备实验环境

首先需要说明, 我并非故意不遵照 PPT 的方案, 而是在初步尝试之后, 发现其有些过时 (从推荐使用 Ubuntu 14.04 可窥见一斑). 故为了避免遇到一些繁琐而无关实验实质的问题, 才另辟蹊径, 选择采用现在的方法.

首先, 相较于 PPT 中建议使用的 Virtual Box 虚拟机, VMWare 无疑是更加成熟的解决方案, 并且由于 VMware Academic Program, 其商业化的特点并不构成阻碍. 至于其具体优势, 在本实验中体现在可以" 一键" 安装好操作系统, 更加方便的设置共享文件夹等. 故, 我选择使用 VMWare.

其次, Ubuntu 18.04 是 Ubuntu 最新的 LTS 版本, 与发行于 2014 年的 14.04 版本相较, 我认为更加可靠. 毕竟, 使用新版本可以很大程度上避免潜在的 Bugs. 故, 我选择使用 Ubuntu 18.04.

而在 Python 版本的选择上, 由于我曾有过编写爬虫的经验, 经历过 Python 2 对于中文编码的各种不兼容, 加上如今使用 Python 3 的大势所趋, 自然, 我选择用最新的 Python 3.7, 而非建议的 Python 2.7.

在明确了以上选择之后, 至于如何安装 VMWare, 如何在虚拟机中安装 Ubuntu, 如何设置虚拟机, 如何在 Linux 环境下配置编程所需环境, 都不过是细枝末节, 依照所涉及软件的官方指引操作即可, 在此不表.

2.2 程序总体设计

分析本次实验的任务, 可以发现, 涉及到了爬虫系统的两大主要模块:Crawler 和 Parser. 那么, 很自然地, 运用以往所学的程序设计思想, 我计划将这两个模块分开而各自单独实现在一个 Python 文件中. 具体而言, 例如 Parser, 既要实现一个基本的解析器用以提取网页中的 URL 和图片, 又需要针对糗事百科实现一个专门的解析器, 那么, 面对对象的程序设计方法就有了用武之地: 先实现一个 BaseParser, 再以此派生出糗事百科的 Parser.

在实现这两个模块之后, 再着手于本次实验的具体任务, 只需编写一个主程序 (main.py), 在其中实例化所需的 Crawler 和 Parser, 执行相应的任务即可.

2.3 编程实现

2.3.1 Crawler

在本次实验中, 仅仅涉及到基本的 GET 请求以及添加请求的 Headers 信息, 故实现一个 BaseCrawler 即可. 不过, 在具体的实现上, 我使用了 Requests 库而非 Urllib 库, 原因在于 Requests 库具有更简洁的操作方式, 同时支持 Session(以便于处理某些需要登录的站点, 尽管本次未涉及). 关键代码 (位于 crawlers.py 文件) 如下:

```
import requests
class BaseCrawler(object):

    def __init__(self, headers=None):
        self.new_session()
        self.headers = headers
```

```
    def get_html(self, url, headers=None):
        assert self.session != None
        headers = headers or self.headers  # if headers are not specified for this single request, use the global headers
        content = self.session.get(url, headers=headers).content
        decoded_content = content.decode("utf8")  # maybe not utf-8 for some sites?
        return decoded_content

    def new_session(self):
        self.session = requests.Session()
```

在我的实现中, 若在单次请求时未指定 Headers, 则沿用对象创建时设定的 Headers. 至于网页编码, 尽管 UTF-8 编码可以应对多数网页, 但在实际测试中我确实发现了一些网页并非如此, 所以, 在后续的实验中, 可能需要对此进行一些修改.

2.3.2 Parser

如前所言, 主要有 BaseParser 和 QSBKParser. 在 BaseParser 中, 利用 BeautiSoup 以及正则表达式实现了提取网页中的 URL 和图片地址的功能. 但对于实际的网页, `<a>` 标签的 `href` 属性和 `<img>` 标签的 `src` 属性中的内容并不一定总是标准的 URL, 很有可能是相对路径, 甚至根本不是链接 (如某些链接实际是调用 Javascript 脚本).

故, 需要对提取到的可能为链接的内容进行处理. 具体而言, 即去除非法链接, 将相对路径补全为标准链接 (此处用到了 urllib.parser 中的 urljoin). 而对于图片, 还需判断其后缀是否为常见的图片格式. 具体实现如下:

```
from bs4 import BeautifulSoup
from urllib.parse import urljoin
import re
class BaseParser(object):

    def __init__(self):
        self.standard_url_pattern = re.compile(r"^(https?:)?//[^\s]*$")  # https:// or http:// or //
        self.relative_path_pattern = re.compile(r"^\.{0,2}/[^\s]*$")  # / or ./ or ../
        self.common_img_formats = ("bmp", "jpg", "jpeg", "png", "gif")

    def parse_url(self, html_content, current_url=None):
        soup = BeautifulSoup(html_content, features="html.parser")  # specify features to avoid potential warnings
        url_set = set()
        for a in soup.findAll("a"):
```

```
            maybe_url = a.get("href", "")
            final_url = self.handle_url(maybe_url, current_url)
            if final_url:
                url_set.add(final_url)
        return url_set

    def parse_img(self, html_content, current_url=None):
        soup = BeautifulSoup(html_content, features="html.parser")
        img_set = set()
        for img in soup.findAll("img"):
            maybe_img = img.get("src", "")
            final_img = self.handle_img(maybe_img, current_url)
            if final_img:
                img_set.add(final_img)
        return img_set

    def handle_url(self, maybe_url, current_url=None):
        maybe_url = maybe_url.strip()
        if self.standard_url_pattern.match(maybe_url) or
        ↪  self.relative_path_pattern.match(maybe_url):
            if current_url:
                return urljoin(current_url, maybe_url)
            else:
                return maybe_url  # without the current url, I can't turn a
                ↪  relative path into a standard url
        else:
            return None

    def handle_img(self, maybe_img, current_url=None):
        final_url = self.handle_url(maybe_img, current_url)
        if not final_url:  # make sure that it's a valid url
            return None
        last = final_url.split(".")[-1]
        if last in self.common_img_formats:
            return final_url
        else:
            return None
```

需要注意的是, 将相对路径转换为绝对路径需要已知当前路径, 即代码中的`current_url`. 在调用时需要指定值, 否则无法处理相对路径, 只能将其不做处理原样返回.

以及另一个细节, 在使用 BeautifulSoup 时, 若不指定`features="html.parser"`, 则会产生 Warnings, 虽不影响程序运行, 但产生了大量无关输出.

在 BaseParser 的基础上, 以其为基类, 派生出 QSBKParser. 因此可以复用基类的方法, 只需为其单独实现针对糗事百科页面的解析功能. 代码如下:

```
class QSBKParser(BaseParser):

    def __init__(self):
        super(QSBKParser, self).__init__()

    def parse_page(self, html_content, current_url=None):
        soup = BeautifulSoup(html_content, features="html.parser")
        docs = dict()
        for single_post in soup.findAll("div", {"id":
        ↪  re.compile(r"qiushi_tag_\d+")}):
            qiushi_tag = single_post["id"].split("_")[-1]
            content = single_post.find("div", {"class":
            ↪  "content"}).span.text.strip()
            maybe_img = single_post.find("div", {"class":
            ↪  "thumb"}).a.img["src"]
            img_url = self.handle_img(maybe_img, current_url)
            docs[qiushi_tag] = {
                "content": content,
                "img_url": img_url,
            }
        maybe_next_page = soup.find("span", {"class": "next"}).parent["href"]
        next_page = self.handle_url(maybe_next_page, current_url)
        return docs, next_page
```

2.3.3 Main

由于本次实验要求解析命令行参数, 如通过命令行设定要解析的网址, 我使用了 sys 库以及 getopt 库. 前者用以获取运行程序的参数, 后者用以解析.

考虑到 3 个练习具有共性, 为减少冗余代码, 我并未为每个练习单独编写程序, 而是在调用时传入 `--task(-t)` 参数来选择. 除此之外, 还支持 `--url(-u)` 来设定网址, `--output(-o)` 来设定输出文件, 以及 `--help(-h)` 来输出帮助信息.

则 main.py 的主体代码 (略去具体函数实现) 如下:

```
def main(argv):
    url = "https://www.baidu.com"
    task = 0
    output_file = None
    try:
        opts, _ = getopt.getopt(argv, "u:t:o:h", ["url=", "task=", "output=",
        ↪  "help"])
```

```
    except getopt.GetoptError:
        print("Invalid Arguments!")
        opts = (("-h", None),)
    for opt, arg in opts:
        if opt in ("-h", "--help"):
            show_help()
            exit()
        elif opt in ("-u", "--url"):
            url = arg
        elif opt in ("-t", "--task"):
            task = int(arg)
        elif opt in ("-o", "--output"):
            output_file = arg
    if task == 1:
        get_urls(url, output_file or "res1.txt")
    elif task == 2:
        get_imgs(url, output_file or "res2.txt")
    elif task == 3:
        get_jokes(url, output_file or "res3.txt")

if __name__ == '__main__':
    main(sys.argv[1:])
```

在运行时, 可采取如下方式:

```
python main.py -u www.baidu.com -o baidu.txt -t 1
```

2.3.4 Unit Test

由于采用了模块化编程, 为了减少 Bug, 抑或更方便地发现 Bug, 对单个模块进行测试是必要的. 故我在 crawlers.py 和 parsers.py 中, 均编写了用以简单测试的代码. 例如, 在 crawlers.py 中:

```
def unit_test():
    print("Unit Test Begins.")
    test_crawler = BaseCrawler()
    print("Try to get the content of https://keithnull.top/")
    html = test_crawler.get_html("https://keithnull.top/")
    assert html.find("Keith") != -1
    print("Success")

if __name__ == '__main__':
    unit_test()
```

3 总结与分析

在这次实验之前, 我其实已有过编写爬虫的经验 (高中时期, 用 Python 2 编写了一个爬取校内信息并存储在数据库中的爬虫), 正是如此, 我意识到编写一个爬虫不仅仅是模拟请求然后解析数据, 而是需要对代码有合理的组织, 否则只会把自己困缚, 难以调试和维护. 所以, 本次实验我并没有完全依照 PPT 的建议, 而是自行改变了很多. 或许这些" 自作主张" 正朝着更合理的方向, 又或许毫无意义, 但都是我思考之后的决定.

这次实验给了我一个从零开始编写爬虫的机会, 有些时候我愿意按着自己的想法进行, 无谓对错, 我相信, 思考与实践的过程即是收获.