WebScour

Web Crawler Using Python

A Python-Based Automated Web Crawling System

For Information Discovery and Data Collection

Author: Kunal Kushwaha

Infosys Virtual Internship Program
Domain: Python / Web Technologies

Project Overview

WebScour is a Python-based web crawler developed as part of the Infosys Virtual Internship Program. The project focuses on automated information discovery, web page collection, and building a strong foundation for a search engine pipeline.

The internet contains a massive amount of information, making manual data collection inefficient and impractical. WebScour solves this problem by automatically discovering, crawling, and storing web pages starting from a given seed URL.

The crawler follows a FIFO (queue-based) approach, avoids duplicate visits, filters invalid links, restricts crawling to the same domain, and stores downloaded pages locally for further processing and searching.

What Problem Does WebScour Solve?

Challenges on the Web

- Huge amount of information available online
- Manual discovery and collection of pages is time-consuming
- Difficulty in organizing and searching collected data

WebScour Solution

WebScour automates this process by:

- Crawling selected websites
- Collecting and storing page content
- Preparing data for indexing and searching

Web Crawling Pipeline

WebScour is designed based on the standard search engine workflow:

Crawling → Collecting → Indexing → Searching

1. Crawling

- Starts from a seed URL
- Downloads the web page
- Extracts hyperlinks
- Discovers new pages recursively
- Avoids duplicate URLs
- Restricts crawling to the same domain

2. Collecting

- Stores downloaded HTML pages locally
- Creates a structured dataset for future processing

3. Indexing (Future Scope)

- Organizes collected content
- Builds keyword-based indexes for fast searching

4. Searching (Future Scope)

- Allows users to search indexed content
- Forms the final layer of a search engine

> **Current Implementation Status:**
>
> - ✓ Crawling

- ✓ Collecting
- ○ Indexing (future enhancement)
- ○ Searching (future enhancement)

Main Focus of the Web Crawler

The WebScour crawler primarily focuses on:

Information Discovery

Automatically finding new web pages through hyperlinks.

Automation

Removing manual effort by automatically fetching, filtering, and storing pages.

Search Enablement

Preparing clean and structured data that can later be indexed and searched.

System Architecture

The architecture of WebScour follows real-world search engine design:

Seed URL → Web Crawler → Page Storage → Indexer → Search Engine

Component	Description
Seed URL	The starting point of the crawl that defines the scope
Web Crawler	• Fetches web pages • Handles retries on failure • Extracts and filters links • Avoids duplicate crawling • Enforces same-domain restriction
Page Storage	• Saves HTML pages locally • Maintains crawled data for analysis
Indexer (Future Scope)	Converts pages into searchable data
Search Engine (Future Scope)	Provides keyword-based search functionality

Tech Stack – Python Focused

Programming Language

Python 3

Core Python Concepts Used

- Functions and modules
- Data structures:
  - list (queue)
  - set (visited URLs)
- File handling
- Exception handling
- Retry logic

Libraries and Tools

- **Requests** – HTTP requests
- **BeautifulSoup** – HTML parsing
- **OS module** – File and directory handling
- **Time module** – Delays and performance tracking
- **urllib.parse** – URL normalization and domain extraction

How the Crawler Works

1. Initialize a queue with a seed URL
2. Fetch the webpage content
3. Save the HTML page locally
4. Extract valid HTTP/HTTPS links
5. Filter invalid links (mailto, javascript, tel, #)
6. Restrict crawling to the same domain
7. Avoid duplicate URLs using a visited set
8. Retry failed requests (limited attempts)
9. Repeat until the maximum page limit is reached

Features Implemented

- FIFO queue-based crawling
- Same-domain crawling only
- Duplicate URL prevention
- Invalid link filtering
- Retry logic for failed URLs
- Politeness delay (0.5 seconds)
- Local storage of HTML pages
- Performance measurement (time & average speed)
- Logging of visited URLs

Project Structure

The WebScour project follows a simple and well-organized directory structure:

```
webscour/ ├── crawler.py # Main Python script ├── pages/ # Downloaded HTML pages | ├── page_1.html | ├── page_2.html | └── ... ├── visited.txt # Visited URLs log └── README.md # Documentation
```

Component	Description
crawler.py	Contains the complete crawler logic
pages/	Stores downloaded web pages
visited.txt	Maintains record of visited URLs
README.md	Explains the project

How to Run the Project

Step 1: Install Dependencies

```
pip install requests beautifulsoup4
```

Step 2: Run the Crawler

```
python crawler.py
```

Configuration

The behavior of the WebScour crawler can be customized by modifying the following variables in the crawler.py file:

Seed URL

Defines the starting point of the crawl.

```
seed_url = "https://en.wikipedia.org/wiki/Infosys"
```

Maximum Pages

Controls how many pages the crawler is allowed to download.

```
MAX_PAGES = 5
```

This limit helps prevent excessive crawling and avoids overloading target websites.

Politeness Delay

A delay of 0.5 seconds is applied between successive HTTP requests to reduce server load and promote ethical crawling. This value can be adjusted if required.

Output

After execution, the crawler generates the following outputs:

Downloaded Pages

- All successfully crawled HTML pages are stored in the `pages/` directory
- Each page is saved with a unique filename (e.g., page_1.html, page_2.html)

Visited URLs File

- All unique URLs visited during the crawl are stored in `visited.txt`
- This file helps verify duplicate prevention and crawl coverage

Console Output

The terminal displays important crawling statistics, including:

- Total number of pages crawled
- Number of duplicate links encountered
- Total crawling time
- Average time taken per page

Limitations

Although WebScour works efficiently for small to medium-sized websites, it has the following limitations:

- Crawls only static HTML pages
- Does not execute JavaScript
- Uses a single-threaded crawling approach
- Does not currently respect robots.txt
- Not suitable for very large-scale web crawling

Future Enhancements

The project can be extended with the following features:

- Implement multi-threaded crawling for better performance
- Add robots.txt compliance for ethical crawling
- Introduce depth-based crawling control
- Store crawled data in a database
- Implement indexing of collected pages
- Build a keyword-based search engine
- Create a web interface using Flask or similar frameworks

Learning Outcomes

Through this project, the following concepts were learned and applied:

- Web crawler architecture and workflow
- Search engine pipeline (crawling → collecting → indexing → searching)
- Use of Python data structures such as queues and sets
- Handling HTTP requests and responses
- Parsing HTML using BeautifulSoup
- Error handling and retry mechanisms
- File and directory management
- Writing clean, maintainable, and well-documented code

Internship Information

Field	Details
Program	Infosys Virtual Internship
Project Title	WebScour – Web Crawler Using Python
Domain	Python / Web Technologies
Author	Kunal Kushwaha

References

- Python Official Documentation
- Requests Library Documentation
- BeautifulSoup Documentation
- Wikipedia (used for testing purposes)

PROJECT DEVELOPMENT AND CODE IMPLEMENTATION DETAILS

(Web Crawler Using Python)

INTRODUCTION TO CODE DEVELOPMENT

The Web Crawler project was developed in a systematic and incremental manner. Instead of directly writing the complete code, the project followed a structured approach where the logic was first designed using pseudocode and then implemented step by step. Each component of the crawler was built individually and later enhanced through a series of well-defined tasks. This approach ensured clarity, correctness, and easy extensibility of the code.

PHASE 1: PSEUDOCODE AND INITIAL DESIGN

The development process began with writing pseudocode to clearly define the crawling workflow. The pseudocode described the following core steps:

- Initializing a queue with a seed URL
- Maintaining a visited set to avoid duplicate crawling
- Fetching web pages one by one using a FIFO approach
- Saving the downloaded HTML content
- Extracting links from each page
- Adding new links to the queue
- Repeating the process until a maximum page limit is reached

This pseudocode acted as a blueprint for the actual implementation and helped avoid logical errors during coding.

Pseudocode:

```
START

1. Initialize:
   queue = [seed_url]
   visited = empty_set
   page_id = 1

2. While queue is not empty AND pages crawled < MAX_PAGES:
   a. Take a URL from queue (FIFO)
   b. If URL already in visited:
      skip this URL
   c. Fetch HTML for the URL
      if fetching failed → skip
   d. Save the HTML content to a file (page_id.html)
   e. Extract all links from the HTML
   f. For each extracted link:
      if link NOT in visited AND NOT in queue:
         add link to queue
   g. Mark current URL as visited
   h. Increase page_id by 1
   i. Sleep for a short time (0.5 sec) to avoid server overload
```

```
3. END LOOP

4. Print total pages crawled

END
```

1. Start with one URL

- Put the seed URL into a queue.
- Create an empty visited set.
- Set page_id = 1.

2. Loop while you still have pages to crawl

- Take the first URL from the queue.
- If you already visited it → skip.

3. Fetch the page

- Download the HTML.
- If download fails → skip.

4. Save the HTML

- Store the HTML into a file like: page_1.html, page_2.html, etc.

5. Extract links

- Get all <a href="..."> links from the HTML.

6. Add new links to the queue

- Only add links that:
    - are not visited
    - not already in the queue

7. Mark visited

- Add the current URL to visited.

8. Increase counters

- page_id = page_id + 1
- Sleep 0.5 seconds so you don't overload websites.

9. After loop ends

- Print how many pages you crawled.

Summary - Key Things to Implement in Code:

- **Queue:** Use a list or collections.deque to store URLs to be crawled

- **Visited Set:** Use a set to track already crawled URLs and prevent duplicates
- **Page Counter:** Maintain a page_id variable to name saved files uniquely
- **MAX_PAGES:** Define a limit to stop crawling after a certain number of pages
- **FIFO Processing:** Always take URLs from the front of the queue (First In, First Out)
- **HTML Fetching:** Use requests library to download web pages with error handling
- **File Saving:** Save each downloaded HTML to a file with sequential naming
- **Link Extraction:** Use BeautifulSoup to parse HTML and extract all hyperlinks
- **Duplicate Prevention:** Check if URL is already visited or in queue before adding
- **Rate Limiting:** Add sleep delay between requests to avoid overloading servers
- **Statistics:** Track and display total pages crawled at the end

PHASE 2: FUNCTION-WISE IMPLEMENTATION

After finalizing the pseudocode, the crawler was implemented in a modular way using separate functions.

fetch_page Function

The fetch_page function is responsible for downloading the HTML content of a given URL.

Purpose:

- To safely fetch web pages using HTTP requests
- To handle network-related errors

Key Functionality:

- Sends HTTP GET requests using the requests library
- Uses a custom User-Agent to identify the crawler
- Handles timeouts and connection errors
- Returns HTML content when the request is successful
- Returns None if the request fails

Code Implementation:

```
def fetch_page(url):
    try:
        response = requests.get(
            url,
            timeout=5,
            headers={"User-Agent": "WebScourCrawler/1.0"}
        )
        if response.status_code == 200:
            return response.text
        else:
            print(f"[ERROR] Could not load page ({response.status_code}): {url}")
            return None
    except Exception as e:
        print(f"[ERROR] Problem while fetching {url}: {e}")
```

```
        return None
```

Later, retry logic was added to this function as part of Task 6 to improve reliability.

extract_links Function

The extract_links function is responsible for extracting hyperlinks from the downloaded HTML content.

Purpose:

- To discover new web pages from the current page

Key Functionality:

- Parses HTML using BeautifulSoup
- Extracts all anchor tags containing href attributes
- Converts relative URLs into absolute URLs
- Removes URL fragments (such as #section)
- Filters out invalid and non-useful links

Code Implementation:

```
def extract_links(html, base_url):
    soup = BeautifulSoup(html, "html.parser")
    links = set()

    # get all <a href=""> links
    for tag in soup.find_all("a", href=True):
        link = requests.compat.urljoin(base_url, tag['href'])
        links.add(link)

    return links
```

This function ensures that only meaningful and valid links are passed to the crawler.

main Function

The main function controls the entire crawling process.

Purpose:

- To coordinate fetching, link extraction, storage, and task execution

Key Functionality:

- Initializes queue, visited set, counters, and timers
- Implements FIFO crawling using a queue
- Calls fetch_page and extract_links functions
- Saves crawled pages locally
- Applies task-based restrictions and enhancements

- Tracks crawling statistics and performance

Code Implementation:

```
def main():
    seed_url = "https://example.com"
    MAX_PAGES = 3

    queue = [seed_url]
    visited = set()
    page_id = 1
    pages_crawled = 0

    while queue and pages_crawled < MAX_PAGES:
        # a. Take URL from queue
        url = queue.pop(0)

        # b. Skip if already visited
        if url in visited:
            continue

        # c. Fetch page
        html = fetch_page(url)
        if html is None:
            continue

        # d. Save page to file
        filename = f"page_{page_id}.html"
        with open(filename, "w", encoding="utf-8") as f:
            f.write(html)
        print(f"[SAVED] {url} --> {filename}")

        # e. Extract links
        links = extract_links(html, url)

        # f. Add new links to queue
        for link in links:
            if link not in visited and link not in queue:
                queue.append(link)

        # g. Mark as visited
        visited.add(url)

        # h. Increase page_id
        page_id += 1
        pages_crawled += 1

        # i. Sleep
        time.sleep(0.5)

    print("Total pages crawled:", pages_crawled)
```

The main function acts as the core controller of the web crawler.

PHASE 3: TASK-WISE ENHANCEMENTS AND MODIFICATIONS

After the basic crawler was working, multiple tasks were introduced to improve functionality, organization, and robustness.

TASK 1: LIMITED PAGE CRAWLING

Objective:

To prevent infinite crawling and control the size of the crawl.

Implementation:

- Introduced a variable MAX_PAGES
- The crawler stops automatically once the page limit is reached

Code:

```
MAX_PAGES = 5

pages_crawled = 0

while queue and pages_crawled < MAX_PAGES:

  url = queue.pop(0)

  ...

  pages_crawled += 1
```

Benefit:

- Prevents overloading websites
- Ensures predictable execution time
- Makes the crawler safe and controlled

TASK 2: SAVE PAGES INSIDE A DEDICATED FOLDER

Objective:

To organize all crawled HTML files in a structured manner.

Implementation:

- Created a folder named "pages"
- All downloaded HTML pages are saved inside this folder
- Each page is stored with a unique filename such as page_1.html, page_2.html, etc.

Code:

```
import os

folder_name = "pages"

if not os.path.exists(folder_name):

  os.mkdir(folder_name)
```

filename = os.path.join(folder_name, f"page_{page_id}.html")

with open(filename, "w", encoding="utf-8") as f:

f.write(html)

Benefit:

- Keeps the project directory clean
- Makes data management easier
- Helps in offline analysis and future indexing

TASK 3: FILTER USELESS LINKS AND KEEP ONLY HTTP/HTTPS URLS

Objective:

To ensure the crawler processes only valid and meaningful web pages.

Implementation:

Filtered out links such as:

- mailto:
- javascript:
- anchor links (#section)
- Removed URL fragments
- Allowed only URLs with http and https schemes
- tel

Code:

from urllib.parse import urljoin, urlparse, urldefrag

href = tag["href"].strip()

if not href:

continue

if href.startswith("#"):

continue

if href.startswith("mailto:"):

continue

if href.startswith("javascript:"):

continue

if href.startswith("tel:"):

continue

absolute_url = urljoin(base_url, href)

absolute_url, _ = urldefrag(absolute_url)

```
parsed = urlparse(absolute_url)

if parsed.scheme not in ("http", "https"):

    continue
```

Benefit:

- Avoids crawling non-web resources
- Improves crawler accuracy
- Prevents unnecessary processing

TASK 4: SAME DOMAIN CRAWLING ONLY

Objective:

To restrict crawling to the same domain as the seed URL.

Implementation:

- Extracted the domain from the seed URL
- Compared the domain of each extracted link with the seed domain
- Ignored links belonging to external domains

Code:

```
from urllib.parse import urlparse


seed_domain = urlparse(seed_url).netloc


link_domain = urlparse(link).netloc
if link_domain != seed_domain:
    continue
```

Benefit:

- Keeps the crawl focused
- Prevents drifting to unrelated websites
- Makes the crawler suitable for domain-specific analysis

TASK 5: SAVE VISITED URLS TO A FILE

Objective:

To maintain a record of all crawled URLs.

Implementation:

- Stored all visited URLs in a text file named visited.txt
- Each URL is written on a separate line

Code:

```
with open("visited.txt", "w", encoding="utf-8") as file:
    for url in visited:
        file.write(url + "\n")
```

Benefit:

- Helps verify duplicate prevention
- Useful for analysis and reporting
- Provides transparency of crawl coverage

TASK 6: RETRY LOGIC FOR FAILED URLS

Objective:

To handle temporary network failures gracefully.

Implementation:

- Added retry logic inside the fetch_page function
- Retries fetching a page a limited number of times
- Stops retrying after reaching the maximum retry limit

Code:

```
import requests
import time


def fetch_page(url, max_retries=3):
    for attempt in range(max_retries):
        try:
            response = requests.get(url, timeout=5)
            if response.status_code == 200:
                return response.text
        except Exception:
            time.sleep(1)
    return None
```

Benefit:

- Improves crawler reliability
- Handles transient network issues
- Reduces crawl failure rate

PHASE 4: PERFORMANCE MEASUREMENT

To evaluate crawler efficiency, performance metrics were added.

Measured Metrics:

- Total crawling time
- Average time taken per page

- Number of duplicate URLs encountered

Benefit:

- Helps understand crawler speed
- Useful for performance optimization
- Provides quantitative evaluation of the crawler

Final code after completion of milestone 1

```
import requests
from bs4 import BeautifulSoup
import time
import os
from urllib.parse import urljoin, urlparse, urldefrag


def fetch_page(url, max_retries=3): # takes url and max retries as input(this is additional parameter to
control retries)
    """
    Tries to download the webpage at the given URL.
    Retries a few times if there is a temporary error (network, timeout, etc.).
    Returns the HTML text if successful, otherwise returns None.
    """
    for attempt in range(1, max_retries + 1):
        try:
            print(f"[INFO] Fetching (attempt {attempt}/{max_retries}):{url}") # logging the attempt number
            response = requests.get(
                url,
                timeout=5,
                headers={"User-Agent": "WebScourCrawler/1.0"}
            )

            if response.status_code == 200:
                return response.text
            else:
                # For non-200 codes we don't retry (here we just report)
                print(f"[ERROR] Status {response.status_code} for {url}") # As these are client/server
errors, no point in retrying
                return None

        except Exception as error:
            print(f"[ERROR] Problem while fetching {url}: {error}") # passing the error message before
retrying , if possible

            # If this was the last attempt, give up
            if attempt == max_retries:
                print(f"[ERROR] Giving up on {url} after {max_retries} attempts.")
                return None

            # Small delay before next retry
            time.sleep(1)
```

```python
def extract_links(html, base_url):
    """
    Extract only useful HTTP/HTTPS links from the page.
    Skips: mailto:, javascript:, tel:, #section, etc.
    """
    soup = BeautifulSoup(html, "html.parser")
    links = set()

    for tag in soup.find_all("a", href=True):
        href = tag["href"].strip()

        # 1. Skip obviously useless hrefs
        if not href:
            continue
        if href.startswith("#"):
            continue
        if href.startswith("mailto:"):
            continue
        if href.startswith("javascript:"):
            continue
        if href.startswith("tel:"):
            continue

        # 2. Make absolute URL
        absolute = urljoin(base_url, href)

        # 3. Remove fragment (#section)
        absolute, _ = urldefrag(absolute)

        # 4. Keep only http/https   ---> url have scheme , domain, path
        parsed = urlparse(absolute)
        if parsed.scheme not in ("http", "https"): # skips ftp://, file://, data:// etc.
            continue

        links.add(absolute)

    return links


def main():
    seed_url = "https://en.wikipedia.org/wiki/Infosys"
    MAX_PAGES = 50


    # This basically gets the domain from the seed URL for reference
    # e.g., for "https://en.wikipedia.org/xyz", it gets "en.wikipedia.org"
    # We can use this to restrict crawling to the same domain if needed.
    seed_domain = urlparse(seed_url).netloc
    print(f"[INFO] Seed domain: {seed_domain}")

    folder_name = "pages"
    if not os.path.exists(folder_name):
        os.mkdir(folder_name)

    queue = [seed_url]
    visited = set()
    page_id = 1
    pages_crawled = 0

    duplicate_count = 0   # SIMPLE duplicate counter
    start_time = time.time()

    while queue and pages_crawled < MAX_PAGES:
```

```
        url = queue.pop(0)

        # b. Check for duplicates
        if url in visited:
            duplicate_count += 1   #COUNT DUPLICATE HERE
            continue

        # c. Fetch page
        html = fetch_page(url)
        if html is None:
            continue

        # d. Save HTML
        filename = os.path.join(folder_name, f"page_{page_id}.html")
        with open(filename, "w", encoding="utf-8") as f:
            f.write(html)
        print(f"[SAVED] {url} --> {filename}")

        # e. Extract links
        links = extract_links(html, url)

        # f. Add new links to queue
        for link in links:
            link_domain = urlparse(link).netloc # This gets the domain of the link
            if link_domain != seed_domain: # Restrict to same domain
                continue
            if link not in visited and link not in queue:
                queue.append(link)

        # g. Mark visited
        visited.add(url)

        # h. Update counters
        page_id += 1
        pages_crawled += 1

        time.sleep(0.5)

    end_time = time.time()
    total_time = end_time - start_time
    avg_time = total_time / pages_crawled if pages_crawled > 0 else 0

    print("\n--- SUMMARY ---")
    print("Total pages crawled:", pages_crawled)
    print("Duplicate links encountered:", duplicate_count)
    print(f"Total time: {total_time:.2f} sec")
    print(f"Average time per page: {avg_time:.2f} sec")

    # Save visited URLs to a file
    visited_file = "visited.txt"
    with open(visited_file, "w", encoding="utf-8") as f:
        for v in sorted(visited):
            f.write(v + "\n")
    print(f"[INFO] Saved visited URLs to {visited_file}")


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