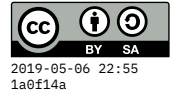


# Milestone 4: webgc part 1



## Contents

This will be the first part of implementation of `webgc`, a tool to garbage-collect unreferenced assets from static web sites. I would like development on this tool to have *its own* repository, so I created a new one at `gitlab.liu.edu/cs120s19/webgc`<sup>1</sup>. You should **fork** this into your own gitlab account, and then **clone** it to your local computer or VM. Place it *outside* of the previous `cs164` or `cs164pub` folders.



<sup>1</sup>`gitlab.liu.edu/cs120s19/webgc`

## Unit tests

On the command-line, you should be able to run:

```
python -m unittest discover
```

(If you get an error about `html.parser`, replace `python` with `python3` in that command.)

If it worked as it should, it will report 20 (or so) test failures. See the particular tests in `tests/test_extract.py`. We are starting with one of the most fundamental pieces of functionality the tool will need: reading HTML and CSS content, and extracting the links.

There are several kinds of links to external files or sites in HTML:

```
<html>
  <head>
    <link rel="stylesheet" href="style.css">
    <script src="bootstrap.js"></script>
  </head>
  <body>
    <a href="about.html">My page</a>
    
  </body>
</html>
```

The above HTML code contains four links: a stylesheet, a script, a linked web page, and an image. There are other HTML tags that also reference external files, but the relevant *attributes* tend to be named either `href` or `src`.

Style sheets can also contain references to external files, such as images. The syntax there is to use `url()`, as in this example:

```
.topbanner {  
    background: url('topbanner.png') #00D no-repeat fixed;  
}
```

Furthermore, CSS can be *embedded* within HTML, so we also have to look for `url()` within `<style>` segments of HTML files!

```
<head>  
    <style>  
        body { background: url(background.png); }  
    </style>  
</head>  
<p>Here I'm just mentioning url(uniform resource locator),  
    but it shouldn't register as a link!</p>
```

## Module code

The code being tested is in `webgc/extract.py` – roughly, these three functions:

```
def extract_html_links(content):  
    pass  
  
def extract_css_links(content):  
    pass  
  
def extract_links_from_file(pathname):  
    pass
```

In Python, `pass` is just a placeholder. You would replace it with your own code. In the first two functions, we expect the parameter `content` to be a string. In the third function, we expect a filename, possibly including its path in the filesystem hierarchy.

Each function is expected to return a Python `set` type. A set is a collection of elements, but unlike an array or list, the ordering is insignificant and duplicates are not allowed. You can convert any list (or iterable) into a set using `set()`, and you can add new elements with `.add()`. An example in the Python REPL:

```
>>> s1 = set([19,3])  
>>> s1  
{3, 19}  
>>> s1.add(40)  
>>> s1  
{3, 40, 19}  
>>> s1.add(19)  
>>> s1  
{3, 40, 19}
```

```
>>> s1 == set([19,40,3])
True
```

The subsections below cover some tips and specifications for implementing these functions.

### Extract from CSS

My tip here is to use regular expressions and the `findall` method of the Python `re` module<sup>2</sup>. It's tricky to construct regular expressions that do what you need (and not too much more). Here is a candidate I developed that may work fairly well:

```
CSS_URL_REGEX = \
    re.compile(r"""(?:url|@import(?: +url)?) *""" +
               r"""[\('"]*([^\']*|"[^"]*\])*""")
```

I can explain it more thoroughly in class, but essentially this will look for `url()` or `@import` (or even `@import url()`) and then grab the bit that follows, with or without quotes.

There is probably some tricky-but-valid CSS that will trick it or break it. When we find such an example, we would add it to `test_extract.py` as a new test case, and then get to work fixing the bug. Here's how to test out the regular expression on small examples within the Python REPL:

```
>>> import re
>>> CSS_URL_REGEX = \
...     re.compile(r"""(?:url|@import(?: +url)?) *""" +
...                 r"""[\('"]*([^\']*|"[^"]*\])*""")
>>> CSS_URL_REGEX.findall("background: url('tile.png')")
['tile.png']
```

### Extract from HTML

This is a little trickier because HTML is a more complex language than CSS. In particular, due to its nested and contextual structure, regular expressions may not be appropriate.

Fortunately, Python has a built-in HTML parser that will do much of the heavy lifting – see the `html.parser` module<sup>3</sup>.

As shown on that page, you can use a method `handle_starttag` that will give you access to the `attrs` (attributes) in each HTML tag. Some experimentation shows that `attrs` is a list of key/value pairs:

```
[("href", "about.html"), ("title", "Click me")]
```

So you can iterate through that and add anything



<sup>2</sup>[docs.python.org/3.6/library/re.html](https://docs.python.org/3.6/library/re.html)



<sup>3</sup>[docs.python.org/3.6/library/html.parser.html](https://docs.python.org/3.6/library/html.parser.html)

The set of links could be kept as an instance variable within the parser class, initially it's the empty set, so the class will need a constructor:

```
def __init__(self):
    super(HtmlLinkExtractor, self).__init__()
    self.links = set()
```

### Extract from a file

Although this function isn't currently exercised by the unit tests, the point of it is to read from a file (rather than) and depending on the filename (whether it ends with .html or .css or something else), delegate to the correct function. If the file is something *other than* HTML or CSS, then the set of links it returns can just be empty.

The main program within `extract.py` allows us to exercise these functions on larger files by specifying them on the command line, like this:

```
% python -m webgc.extract angular.html benchmarks.html badge_only.css
angular.html:../src/angular-sprintf.js
angular.html:../src/sprintf.js
angular.html:https://ajax.googleapis.com/ajax/libs/angularjs/1.3.0-rc.3/angu
benchmarks.html:../demo/functiontrace.html
benchmarks.html:../demo/parse.html
benchmarks.html:../assets/style.css
benchmarks.html:../demo/index.html
[...]
badge_only.css:../fonts/fontawesome-webfont.woff
badge_only.css:../fonts/fontawesome-webfont.ttf
badge_only.css:../fonts/fontawesome-webfont.eot
badge_only.css:../fonts/fontawesome-webfont.svg#FontAwesome
```

(These were just some HTML and CSS files I found lying around.)