

## The W3

The next topic we'll cover in this class is how to manipulate files on the World Wide Web (WWW or W3).

The W3 is a **system of linked documents stored on servers** on the Intranet.

A **web browser** is typically used to:

1. View the web pages that may contain text, images, videos, and other multimedia
2. Navigate between web pages by using hyperlinks, by which pages are typically accessed

Some links about the history of the W3:

- <http://info.cern.ch/>
- <http://www.w3.org/History.html>
- <http://nethistory.info/History%20of%20the%20Internet/web.html>

The crucial people/institutions/technologies:

- People
  - Doug Englebart
  - Tim Berners Lee
  - Robert Cailliau
  - Marc Andreessen
- Institutions
  - CERN
  - Netscape

## Uniform Resource Locators (URLs)

In order to unambiguously identify and access particular resources on the W3, each resource must have a unique identifier.

A **uniform resource identifier** (URI) is a string of characters used to identify a resource on the Internet.

A URI can be classified as a **locator** (URL) or a **name** (URN) or both.

A URN functions like a person's name whereas a URL is like a person's street address. So the URN is the item's identity, and the URL is the method for finding it.

A URL is a URI that, in addition to identifying a resource, specifies the **means of acting upon or obtaining the representation**. It does this one of two ways (although the two can be combined):

1. Through a description of the primary access mechanism
2. Through a network "location"

From the W3C (<http://www.w3.org/Addressing/>):

Uniform Resource Identifiers (URIs aka URLs) are short strings that identify resources in the web: documents, images, downloadable files, services, electronic mailboxes, and other resources.

They make resources available under a variety of naming schemes and access methods such as HTTP, FTP, and Internet mail addressable in the same simple way. They reduce the tedium of "log into this server, then issue this magic command ..." down to a single click.

Here is a typical URL dissected:

---

<http://facweb.cdm.depaul.edu/azoko/csc242/helloworld.html>

\\_\_\_/ \\_\_\_\_\_/\\_\_\_\_\_/  
| | |  
protocol host path

---

Here is a more general (and fabricated) example:

<http://user:pass@example.com:992/animal/bird?species=seagull#wings>

\\_\_\_/\\_\_\_/\\_\_\_/\\_\_\_/\\_\_\_/\\_\_\_/\\_\_\_/  
| | | | | | |  
protocol login hosts port path query anchor/fragment

## Hypertext Markup Language (HTML)

From the W3C (<http://www.w3.org/TR/html401/intro/intro.html#h-2.2>):

To publish information for global distribution, one needs a universally understood language, a kind of publishing mother tongue that all computers may potentially understand.

The publishing language used by the World Wide Web is HTML (from HyperText Markup Language).

HTML gives authors the means to:

- Publish online documents with headings, text, tables, lists, photos, etc.
- Retrieve online information via hypertext links, at the click of a button
- Design forms for conducting transactions with remote services, for use in searching for information, making reservations, ordering products, etc.
- Include spreadsheets, video clips, sound clips, and other applications directly in their documents.

Some examples:

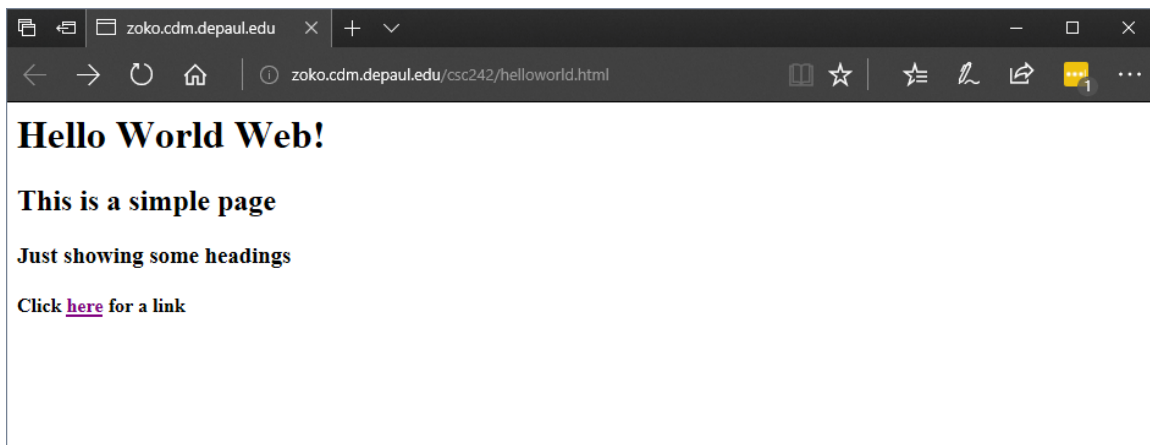
- <http://facweb.cdm.depaul.edu/azoko/csc242/helloworld.html>
- <http://facweb.cdm.depaul.edu/azoko/csc242/image.html>
- <http://facweb.cdm.depaul.edu/azoko/csc242/random.html>
- <http://cdm.depaul.edu>

Samples can be found at <http://facweb.cdm.depaul.edu/azoko>

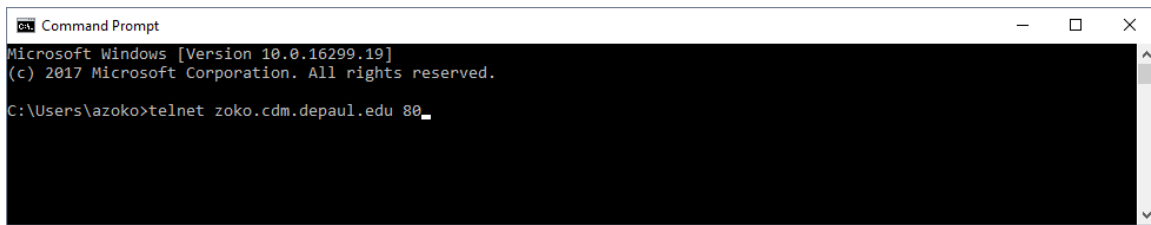
## HTTP

Resources, such as HTML documents, are retrieved using the HTTP protocol. HTTP is a text based protocol that makes it easy to retrieve content from a remote web server. There are two actors at play, a client and a server. A client is typically (but not always) a browser that speaks HTTP to request resources. A server is a machine that runs software that listens for HTTP *requests*, processes them and returns as a *response* the content requested by the client.

Here is an example of a browser making a request to server  
facweb.cdm.depaul.edu/azoko on port 80 for file /CSC242/helloworld.html



Here is an example of a console window making the same request. The command used was telnet facweb.cdm.depaul.edu/azoko 80



```
Command Prompt
Microsoft Windows [Version 10.0.16299.19]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\azoko>telnet zoko.cdm.depaul.edu 80
```

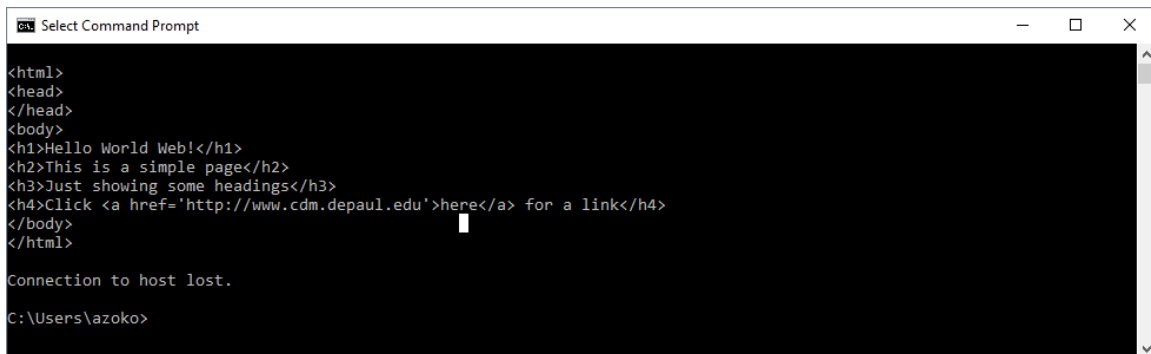
After pressing enter, a blank screen appears.



```
Telnet zoko.cdm.depaul.edu


```

If you type GET /CSC242/HelloWorld.html in the screen (you won't see it appear because the text is being sent to the server!) you will get a response with the HTML.



```
Select Command Prompt

<html>
<head>
</head>
<body>
<h1>Hello World Web!</h1>
<h2>This is a simple page</h2>
<h3>Just showing some headings</h3>
<h4>Click <a href='http://www.cdm.depaul.edu'>here</a> for a link</h4>
</body>
</html>

Connection to host lost.

C:\Users\azoko>
```

## An HTML primer

A file containing an HTML document is typically named with an .html or .htm extension. Any such document placed on a web server will appear on the W3.

The following is our first example, also found in the file **HelloWorld.html**:

```
<html>
<head>
  <title>The title of the document</title>
</head>
<body>
  <h1>Hello world web!</h1>
  <h2>This is a simple page</h2>
  <h3>Just showing some headings</h3>
  <h4>Click <a
href='http://www.cdm.depaul.edu'>here</a> for a link</h4>
</body>
</html>
```

Note that every declaration has an opening tag (<html>) and a closing tag (</html>).

The same format applies for most of the tags in HTML.

## HTML document structure

Every HTML document consists of two basic parts: the “head” and the “body”.

The head starts with the <head> tag and ends with the closing </head> tag.

The body starts with the <body> tag and ends with the closing </body> tag.

Both the head and the body must be enclosed in an html tag:  
<html>...</html>

Thus a typical HTML document has the following basic structure:

```
<html>
<head>
...
</head>
<body>
...
</body>
</html>
```

The above example illustrates an important feature of HTML tags, called nesting.

If you open tag A and then open tag B, you must close tag B before closing tag A.

## Head

The **head** is where you place the title of your document and possibly other features (like references to style sheets, event handlers for dynamic content, etc.).

The title tag usually contains the site's name which is displayed in the browser's title bar and tabs.

```
<title>The title of the document</title>
```

## Body

The **body** of the HTML document contains all content that is displayed in a browser including text, images, lists, tables, and more.

Text in HTML documents is usually enclosed in paragraph tags, and new lines break using the break tag.

```
<h1>Hello world web!</h1>
```

```
<h2>This is a simple page</h2>
```

```
<h3>Just showing some headings</h3>
```

```
<h4>Click <a href='http://www.cdm.depaul.edu'>here</a> for a  
link</h4>
```

## Headlines

There are **six types of headline tags**, or headings.

Headings provide a shortcut for creating larger text, as well as providing a logical (semantic) hierarchy for ordering content.

While you can simulate the headline effect by playing with the text's weight and size, but correct HTML is also about semantically ordering text. Using headings provides a way to do that.

The headings are in decreasing size and weight, meaning that h1 is larger than h6.

See **HelloWorld.html** for an example.

## Lists

Often a page has to include bulleted or numbered lists of various things.

In HTML there are multiple ways to construct ordered and unordered lists of items.

## Ordered lists

The tags for **ordered lists** in HTML are `<ol>` and `</ol>`. To create **list items** within an ordered list you use the tags `<li>` and `</li>`.

The default for ordered lists is to use **Arabic numerals starting at 1**.

For example, the following:

```
<ol>
<li>Item 1</li>
<li>Item 2</li>
</ol>
```

Would be rendered as:

1. Item 1
2. Item 2

You can change the default numbering using the **type attribute**.

For example, the following:

```
<ol type="A">
<li>Item 1</li>
<li>Item 2</li>
<li>Item 3</li>
</ol>
```

Would be rendered as:

- A. Item 1
- B. Item 2
- C. Item 3

## Unordered lists

To create an unordered (bulleted) list, you need to use the `<ul>` and `</ul>` tags.

For example, the following:

```
<ul>
<li>Item 1</li>
<li>Item 2</li>
</ul>
```

Would render as follows:

- Item 1
- Item 2

You can also use the **type attribute** to change the bullets that display.

For example, the following:

```
<ul type = "square">
<li>Item 1</li>
<li>Item 2</li>
</ul>
```

Would render as follows:

- Item 1
- Item 2
- 

See <http://facweb.cdm.depaul.edu/azoko/csc242/lists.html>

## Images

To insert an image into an HTML page, you need to use the `<img>` tag.

In the `<img>` tag the **src attribute** is where you specify the location and file name of the image that you want to display.

For example, the following would place a file called `cookie.jpg` located in the same directory as the HTML file into the page.

```

```

we can also reference a remote image:

```

```

See the **image.html** example for the full page.

## Hyperlinks

HTML uses the anchor (`<a>` and `</a>`) tag to create a link to another document.

An anchor can point to any resource on the Web, including an HTML page, an image, a sound file, a movie, etc.

The syntax for creating an anchor looks like:

```
<a href =
"http://facweb.cdm.depaul.edu/azoko/csc242/helloworld.html
">First Page</a>
```

This particular anchor uses what is called an **absolute URL**. This is the URL that you could type into a browser window in order to move to the page. (Note that it must include the `http://` although some browsers don't require that information).

You can also create anchors using a different kind of reference:



```
<a href="image.html">Go to the image sample!</a>
```

This is what is called a **relative URL**. See the full example in **test2.html**.

It gives a URL relative to the current page. So the absolute URL for this would include `http://`, the server on which the HTML page in which it's embedded is located, and the path on the server to reach the directory in which the HTML file is located.

The anchor tag is also used to create links that can **send a mail message**, provided that the browser accessing the web page is connected to software that allows for the creation of e-mail messages:

```
<a href="mailto:azoko@cdm.depaul.edu"> Click here to e-mail  
Mr. Zoko</a>
```

See the full example in **random.html**.

## Web clients and the `urllib` module

Users use browsers to access web pages on the W3. Any program may act as a client and access and download web pages.

To do this in Python, you would use the **`urllib.request`** module.

It contains functions and classes that are used to open URLs in a way similar to how files are opened.

A Python program can thus be an HTTP client and request and receive resources on the web.

In particular, the **`urllib.request`** method **`urlopen()`** function is similar to the built-in function `open()`, but accepts universal resource locators (URLs) instead of file names:

```
>>> from urllib.request import urlopen  
>>> response = urlopen('http://www.google.com')  
>>> type(response)  
<class 'http.client.HTTPResponse'>
```

A file-like object is returned. It supports the methods **`read()`**, **`getheaders()`**, **`geturl()`** and many others.

For example:

```
>>> response.getheaders()  
[('Date', 'Mon, 20 May 2013 21:51:04 GMT'), ('Expires', '-1'),  
(('Cache-Control', 'private, max-age=0', 'SAMEORIGIN'), ('Connection', 'close'))]
```

```
>>> html = response.read()
>>> type(html)
<class 'bytes'>
>>> response.geturl()
'http://www.google.com'
>>> html = html.decode()
>>> type(html)
<class 'str'>
>>> html.count('google')
92
```

Another useful method in the **urllib.request** module is **urlretrieve(url, filename)**.

It copies a network object denoted by a URL to a local file filename.

```
>>> from urllib.request import urlretrieve
>>> urlretrieve('http://www.google.com',
'google.txt')
('google.txt', <http.client.HTTPMessage object at
0x02478830>)
```

In the file **url.py** is a program that opens the URL for the DePaul CDM main page, reads it and decodes it, prints the HTML, and counts the number of times the word DePaul appears on the page.

Let's step through what that program is doing.

## Parsing HTML files

The goal of this portion of the class is to learn to mine web pages for information.

To do that we need tools to automatically parse HTML web pages. The Python module **html.parser** provides classes that make it easy to parse HTML files.

The class **HTMLParser** from this module parses HTML files as follows:

1. The **HTMLParser** class is instantiated without arguments.
2. An **HTMLParser** instance is fed HTML data and calls handler functions when tags begin and end.

For example:

```
>>> url =
'http://facweb.cdm.depaul.edu/azoko/csc242/helloworld
.html'
>>> from urllib.request import urlopen
>>> content = urlopen(url).read().decode()
>>> parser = HTMLParser()
>>> parser.feed(content)
>>>
```

Why did nothing happen?

The handler methods of the HTMLParser class are meant to be overridden by the user to provide a desired behavior.

Until we write definitions for the methods, those methods won't have any behaviors.

Some of the **handler methods** we can override include:

- **handle\_starttag(tag, attrs)**: Start tag handler
- **handle\_endtag(tag)**: End tag handler
- **handle\_data(data)**: Arbitrary text data handler

For example, **handle\_starttag()** would be invoked when an opening tag (i.e. something of the form **<tag attrs>** is encountered.

Note that the attributes are contained in a list (attrs) where each attribute in the list is represented by a tuple storing the name and value of the attribute.

**handle\_endtag()** would be invoked when a closing tag (i.e. **</tag>**) is encountered.

For an example see the **htmlParser.py** file.

In that file a subclass of the HTMLParser class is created that **overrides the handle\_starttag() and handle\_endtag() methods** so that they print information about the fact that the tag was encountered.

An example of using it (assuming you have a testParser method that creates the HTML parser and uses feed on the HTML from the page) looks like:

```
>>>
testParser('http://facweb.cdm.depaul.edu/azoko/
csc242/helloworld.html')

Encountered a html start tag
Encountered a head start tag
Encountered a title start tag
Encountered a title end tag
Encountered a head end tag
Encountered a body start tag
Encountered a h1 start tag
Encountered a h1 end tag
Encountered a h2 start tag
Encountered a h2 end tag
Encountered a h3 start tag
Encountered a h3 end tag
Encountered a h4 start tag
Encountered a a start tag
Encountered a a end tag
```

```
Encountered a h4 end tag
Encountered a body end tag
Encountered a html end tag >>>
```

Recall what the **helloworld.html** file looks like.

## Exercises

To understand how to override the functions of the HTMLParser class, let's work through a series of exercises.

**Exercise 1:** Write a parser class **LinksParser** that prints to the screen the links found in the web page provided to it.

For example, it would produce this:

```
>>>
testParser('http://facweb.cdm.depaul.edu/azoko/csc242
/random.html')
mailto:azoko@cdm.depaul.edu
image.html
>>>
testParser('http://facweb.cdm.depaul.edu/azoko/csc242
/helloworld.html')
http://www.cdm.depaul.edu
```

**Exercise 2:** Write a parser class **PrettyParser** that prints the names of the start and end tags in the order that they appear in the document, and with an indentation that is proportional to the element's depth in the tree structure of the document. The class should ignore HTML elements that do not require an end tag, such as p and br.

For example, it would work as follows:

```
>>>
testParser('http://facweb.cdm.depaul.edu/azoko/csc242/helloworld.html')
html start
  head start
    title start
    title end
  head end
  body start
    h1 start
    h1 end
    h2 start
    h2 end
    h3 start
    h3 end
    h4 start
      a start
      a end
    h4 end
  body end
html end
```

## Web crawler

We conclude this chapter by considering a basic web crawler, which is a program that systematically visits web pages by following hyperlinks.

### Version 1

The first approach to the web crawler will use two methods to crawl through web pages:

1. **crawl()** takes a URL as a parameter. It analyzes the web page at the specified URL to get the URLs of all linked pages and then recursively calls itself on the URLs of neighboring pages
2. **analyze()** takes a URL as a parameter. It opens the web page at that URL and uses a Collector object to collect the URLs of the pages linked to it.

Consider the code in the file **firstcrawler.py**. Visit <http://facweb.cdm.depaul.edu/azoko/CSC242/crawl/One.html>

Note that **crawl()** does not have a base case. This means that the function will continue to **crawl through web pages indefinitely**, which may be completely appropriate depending on what we want to do.

The exercises in the book consider limiting this in various ways.

Let's **test the first version of the crawler** on a set of web pages.

The web pages we will use were constructed solely for this purpose, and look like the following:

1. Page One → Page Two, Page Three
2. Page Two → Page Five, Page Six
3. Page Three → Page Four
4. Page Four → Page Five
5. Page Five → Page One, Page Two, Page Six

What happens when we run the crawler? Why?

### Version 2

A fix to the problem is to make sure that we don't visit a web page more than once.

**Exercise, part (a):** To do that we need a variable that keeps track of the web pages we have already visited.

**Exercise, part (b):** The **crawl()** method needs to be changed so that it only visits a web page if the link has not been visited before.

See the solution in **crawler.py**.

When we test this version of the crawler on the sample pages we don't have the same issues we did before:

```
>>> c=Crawler()
>>> c.crawl('http://facweb.cdm.depaul.edu/azoko/CSC242/crawl/One.html')
visiting
http://facweb.cdm.depaul.edu/azoko/CSC242/crawl/One.html
visiting
http://facweb.cdm.depaul.edu/azoko/CSC242/crawl/three.html
visiting
http://facweb.cdm.depaul.edu/azoko/CSC242/crawl/four.html
visiting
http://facweb.cdm.depaul.edu/azoko/CSC242/crawl/Five.html
visiting
http://facweb.cdm.depaul.edu/azoko/CSC242/crawl/Two.html
visiting
http://facweb.cdm.depaul.edu/azoko/CSC242/crawl/five.html
visiting
http://facweb.cdm.depaul.edu/azoko/CSC242/crawl/six.html
visiting
http://facweb.cdm.depaul.edu/azoko/CSC242/crawl/six.html
visiting
http://facweb.cdm.depaul.edu/azoko/CSC242/crawl/two.html
```

why did we visit page six twice?