CS 1410 Assignment 2: HTML

In this assignment, you will create a hierarchy of python classes that will allow you to easily create HTML web pages by writing python programs. You will store these classes in a python module named `html`.

The assignment is split into two parts. Each of the parts has a set of specific pass-off requirements and a due date.

In the first part, you will create generic python classes that have most of the functionality necessary to represent and display the text of HTML tags. The pass-off requirements for this part include the creation of some simple HTML documents using the classes in your html module.

In the second part, you will create python classes that inherit from the base classes of the first part to enhance the usability of your html module. These classes will each represent a single HTML tag. The pass-off requirement for this part includes the creation of a specific web page using your html module.

Getting started

Download the starter files here:

- html_part1_test.py

This program is used to do unit testing on your solution to part 1 of the assignment. Download it and save it in the same folder where you will be working on this assignment.

- html_part2_test.py

This program is used to do unit testing on your solution to part 2 of the assignment. Download it and save it in the same folder where you will be working on this assignment.

- html_starter.zip

These files are used for part 2 of the assignment. Unzip it and load `cs1410.html` in your browser to see what the final web page will look like.

HTML

For this assignment, you will need to know some of the basics of HTML. HTML is a markup language, most commonly used for describing what information belongs on a web page, and how to display that information in a web browser.

The most fundamental element of an HTML document is a tag. Here is an example tag:

`<img src="images/cit.gif" alt="CIT Logo" border="0" />`.

This is an `img` tag. Note the letters that follow the opening `<` character. This tag has 3 attributes, `src`, `alt`, and `border`. Notice that each attribute has a value. Finally, the tag closes with `/>`.

You will not need to know every tag and every possible attribute for HTML documents. You only need to understand the structure of tags, and be able to learn limited use of about a dozen tags, which we will show you in this assignment.

There are two special attributes that you will need to know about, `id` and `class`. Please don't confuse the HTML attribute `class` with the python classes you are creating.

The `id` attribute is an optional HTML attribute, which gives a tag an identifying name that is unique for all elements on a given page. This is mainly used by the stylesheet for dictating how the page should be displayed. For example, a page might have a list of the CS courses on the right hand side of the page. If this is contained in an HTML `<div>` element with the `right-menu` ID, it would be written as `<div id="right-menu">` in HTML.

The `class` attribute is an optional HTML attribute, which gives a tag zero or more `class` names. A class is not necessarily unique on the page, but identifies groups of elements that are related. For example, if each hyper-link entry in a page belongs to the class `citcss`, it would allow all links to be displayed with a similar design. This is written as `<a class="citcss">`.

HTML tags can also have zero or more `attributes`. An attribute has a name and a value. For example, in the tag

`<img src="images/cit.gif" alt="CIT Logo" border="0" />`,

there are 3 attributes, `src`, `alt`, and `border`, with the values `images/cit.gif`, `CIT Logo`, and `0`, respectively.

Some HTML tags can contain other tags. For example, this shows a `p` tag that contains an `a` tag:

`<p><a href="syllabus.php">Syllabus</a></p>`.

(`p` tags define paragraphs, and `a` tags define hyper-links.) Notice that the `p` tag is denoted with opening and closing brackets, followed by its contents (the `a` tag), then the `p` tag finally closes with `</p>`. This is different than the closing of the `img` tag above, because that tag did not contain any other tags.

If you want to read up more on HTML, try reading through a few of these introductions:

- http://www.w3schools.com/html/default.asp
- http://www.w3.org/MarkUp/Guide/
- http://en.wikipedia.org/wiki/HTML

Part I

Your first task is to create a hierarchy of base classes to represent HTML. In part II you will create many classes where each class represents a single HTML tag. The classes from part II will inherit from these four base classes.

All of your work in part I should be in a single file called `html.py`. These are the four classes to be created in this part. The details of their implementation will follow.

1. `Tag`: This is the base class for all of the types. It knows how to add and store attributes, set the ID of the tag, add classes, and convert a basic tag into its HTML representation.
2. `BlockTag`: This is a simple extension to the `Tag` class. A `BlockTag` is always rendered with a newline in front of it, so it starts on its own line.
3. `Container`: This is an extension of `Tag` that adds the ability to hold content. For example, a paragraph tag `<p>` normally has text inside it, e.g., `<p>Three blind mice.</p>`. The `Container` class tracks any number of contained items, and implements new methods to add items of various types.
4. `BlockContainer`: This is a simple extension of `Container` that renders itself with a leading newline, similar to `BlockTag`. In addition, it sometimes adds newline characters around the items it contains.

All tag types from part II will inherit from one of these base classes.

Tag

Start by implementing the main `Tag` class. This is the most complicated of the base classes; the others inherit from it and make smaller changes to it. As you develop this class, and the others that follow it, you should use the `html_part1_test.py` program to test your implementation. This is a unit test script that attempts to identify errors in individual classes and methods.

This class should store the following data members:

- The tag name. This is a string like `'img'`.
- The tag id. This is a string like `'right-menu'`. If no id is set, then the id should be the empty string.
- The tag classes. This is a list of strings like `[ 'citcss', 'red' ]`. If no classes are added for the tag, then it should be the empty list.
- The tag's other attributes. This is a list of 2-tuples like `[ ('src', 'images/cit.gif'), ('alt', 'CIT Logo') ]`. If no attributes are added for the tag, then it should be the empty list. Note that there is a method named `attrs`, so do not use that for the name of this list.

The `Tag` class must have the following methods:

- The constructor should take the tag name as its only argument (besides `self`, of course). For example, a paragraph tag would be initialized with `'p'` as its constructor parameter. The constructor should store this for later use, and also initialize the other fields of the object as needed.

Example usage to create the tag `<p />`:

```
Tag('p')
```

- `setId(self, id)`: this method sets the ID of the tag. This value should be stored for later use. If this method is called multiple times, it should overwrite the ID each time and keep the most recent one.

  Example usage to create the tag `<div id="maincolumn" />`:

  ```
  t = Tag('div')
  t.setId('maincolumn')
  ```

  To facilitate method chaining (explained later), this method should return `self`.

- `addClass(self, name)`: this method adds a class to the tag. The complete list of classes added should be stored for use when the tag is rendered.

  Example usage to create the tag `<span class="underline bold" />`:

  ```
  t = Tag('span')
  t.addClass('underline')
  t.addClass('bold')
  ```

  This method should also return `self`.

- `addAttr(self, key, value)`: this method adds an arbitrary attribute to the tag. The object should track a list of all attributes that have been added, and then add them to the HTML when the tag is rendered. Example usage to create the tag `<a href="/foo" />`:

  ```
  t = Tag('a')
  t.addAttr('href', '/foo')
  ```

  These can be stored using a list of pairs, to which you can append new attributes as they are added.

  This method should also return `self`.

- `attrs(self)`: this method renders the id, classes, and other attributes into a string suitable for building the complete tag. The string should be empty if there are no attributes.

  If the tag has an ID set, it should add a space, followed by `id="<x>"` to the result, where `<x>` is the ID value. Do not forget the space in front of `id`.

  If the tag has one or more classes, it should add a space, followed by `class="<x> <y> <z> ..."` to the result, where `<x>`, `<y>`, `<z>`, and `...` are the class names that were added to the tag. Note that the class names are seperated by spaces.

  Following the ID and classes (if any), additional attributes that were added to the tag should be appended. These should follow the same pattern: a leading space, the attribute name, and then `="<x>"` where `<x>` is the attribute value.

  The attribute values, id values, and class names should first have characters such as `<` and `>` escaped so that they do not create confusion and cause the page to render incorrectly. To escape them, `import` the `cgi` package, and call `value = cgi.escape(rawvalue, True)` for each attribute value. This will give you a `value` that will be suitable for inclusion in the tag, given a `rawvalue` that was provided to the `addAttr` method.

  Example usage to create the tag `<label id="first" class="right block" for="name" />`:

  ```
  t = Tag('label')
  t.setId('first')
  t.addClass('right')
  t.addClass('block')
  t.addAttr('for', 'name')
  t.attrs()


  # returns: ' id="first" class="right block" for="name"'
  ```

  Note that since `setId`, `addClass`, and `addAttr` all return `self`, we could also write this example using method

chaining as:

```
t = Tag('label')
t.setId('first').addClass('right').addClass('block')
t.addAttr('for', 'name').attrs()


# returns: ' id="first" class="right block" for="name"'
```

This method should return the result string.

- `makeTag(self, contents)`: this method builds the complete string representation of the tag. A tag always starts with `<`, followed by the tag name. Then all attributes are added in (as rendered by `attrs`). The `contents` parameter is a string, which will be an empty string if there are no contents.

  We are using XHTML, so the tag should take one of two forms, depending on whether or not anything is inside the element (the `contents` argument). If there is nothing inside, the tag should be close with `/>`, i.e., a space followed by a slash and a closing angle bracket.

  If the element has contents, the tag should end with a `>` character, then the contents should be appended, and finally a closing tag should be added. A closing tag is `</` (an opening angle bracket followed by a slash), followed by the tag name, and ending with a `>` character.

  Example usage to create the tag `<p class="right">Stuff</p>`:

  ```
  t = Tag('p')
  t.addClass('right')
  t.makeTag('Stuff')


  # returns: '<p class="right">Stuff</p>'
  ```

  Example usage to create the tag `<img src="/foo.gif" />`:

  ```
  t = Tag('img')
  t.addAttr('src', '/foo.gif')
  t.makeTag('')


  # returns: '<img src="/foo.gif" />'
  ```

  Example usage to create the tag `<img src="/foo<d>.gif" />`:

  ```
  t = Tag('img')
  t.addAttr('src', '/foo<d>.gif')
  t.makeTag('')


  # returns: '<img src="/foo&lt;d&gt;.gif" />'
  ```

  Note that the `<` and `>` characters in the attribute value are escaped as `<` and `>`, respectively.

  This method should return the complete tag string.

- `__str__(self)`: this is a convenience method that just returns whatever `self.makeTag('')` returns. It is automatically called whenever python needs to convert the object into a string representation.

  Example usage to create the tag `<img src="/foo.gif" />` and display it:

  ```
  t = Tag('img')
  t.addAttr('src', '/foo.gif')
  t.makeTag('')
  print t


  # displays: '<img src="/foo.gif" />'
  ```

- `__repr__(self)`: this is a convenience method that just returns whatever `self.__str__()` returns. It is automatically called whenever python needs to display the object from the interactive prompt. After defining this method, an interactive session should look like this:

```
>>> from html import *
>>> t = Tag('a')
>>> t.setId('main')
<a id="main" />
>>> t.addClass('flashing')
<a id="main" class="flashing" />
>>> t.addClass('bold')
<a id="main" class="flashing bold" />
```

Here `>>>` is the python prompt. Note that the fully-rendered version of the tag is displayed after each line of input. This is because `setId` and `addClass` both return `self`, and python calls `__repr__` to decide how to present `self` to the interactive user.

BlockTag

The second class to implement is `BlockTag`. It inherits from `Tag`.

- The constructor should take the same arguments as the `Tag` constructor, and it should simply call the `Tag` constructor.
- `__str__(self)`: `BlockTag` overrides the `__str__` method. It should call `__str__` from its parent class, and then return the same result but with a newline character prepended (`'\n'`).

That is it. Classes do not need to make extensive customizations to justify extending a class. Sometimes a little tweak is all that is necessary.

Container

Most of the tags we are interested in have opening and closing tags with contents in the middle. The `Container` class should inherit from `Tag`, and add methods and data members to manage contained content.

This class should store the following additional data members:

- The contents of the Container. This is a list of tags and text that are contained in this tag. If there are no contents, the contents should be the empty list. Note that there is a method named `contents`, so do not use that for the name of this list.

The `Container` class must have the following methods:

- The constructor should take the tag name and pass it along to the `Tag` constructor. It should also do any additional initialization necessary for data members.

  Example usage:

  ```
  t = Container('div')
  print t


  # prints: '<div />'
  ```

- `addTag(self, child)`: this methods should store `child` as the next element contained inside this tag. The child element should be stored in a list, and any number of child elements should be stored.

  Example usage for the tag `<div><p /><a href="/foo" /></div>`:

  ```
  t = Container('div')
  t.addTag(Tag('p'))
  t.addTag(Tag('a').addAttr('href', '/foo'))


  # note: the contents (the 'a' tag) are not displayed
  #       until you implement the __str__ method below.
  ```

  This method should return `self`.

- `addText(self, text)`: Similar to `addTag`, but this method stores text to be rendered instead of a tag. HTML text should

be escaped before it is displayed to prevent `<` and `>` characters (among others) from confusing the browser. To do this, use:

```
safetext = cgi.escape(text)
```

Store the result like you would a child tag element.

Example usage for the tag `<div>A <div> element</div>`:

```
t = Container('div')
t.addText('A <div> element')


# note: the contents (the text 'A <div> element') are not displayed
#       until you implement the __str__ method below.
```

This method should return `self`.

- `contents(self)`: This method constructs a string from the list of contained tags and text and returns it.

  It should convert each child element (tag and text) to a string, and concatenate the resulting strings to form the contents of the tag. To convert any element into a string, use:

  ```
  s = str(element)
  ```

  Then it should return the resulting string.

  Example:

  ```
  t = Container('p')
  t.addTag(Tag('br'))
  t.addText('plain')
  print t.contents()


  # prints: '<br />plain'
  ```

- `__str__(self)`: this method should override the method provided by the superclass. It should convert the contents list to a string using the `contents` method. Then it should pass the result to `self.makeTag` and return what it returns.

  Example:

  ```
  t = Container('p')
  t.addTag(Tag('br'))
  t.addText('plain')
  print t


  # prints: '<p><br />plain</p>'
  ```

  In addition, try each of the examples above any make sure you get the expected result.

BlockContainer

Many tags with contents are easier to read (for humans) with a few extra newlines. The `BlockContainer` class inherits from `Container` and implements the following methods:

- The constructor should take the same parameters as its superclass, and should just pass them along to the superclass constructor.
- `__str__(self)`: this method should be similar to the method in the superclass, with the following changes:
  - The contents of the child elements are first gathered into a single string (just like in the superclass). If this string starts with a newline character (`\n`) and it does not end with a newline, then a newline should be added to the end of this string.

    Use indexing into the string to examine the first and last characters of the string. Does an empty string have a first or last character? How would you check for this case? What should you do in this case?

- The final tag string, created with `self.makeTag`, should be returned with a newline prepended onto it.

The resulting class should work just like `Container`, except that it will always be rendered with a newline preceeding the tag, and if the first element of the tag starts on its own line, the closing tag will as well.

Testing

As you implement these classes, you should be running the example lines to be sure that your classes are working correctly. Also use `html_part1_test.py` to help check your code.

Part I Pass-off

Demonstrate to the lab assistant that your `html.py` causes no errors when tested with `html_part1_test.py`. Submit your file for part 1 of this assignment.

Part II

As you develop these classes, you should use the `html_part2_test.py` program to test your implementation. This is a unit test script that attempts to identify errors in individual classes and methods.

Simple tags

In this part you will now implement several classes that will represent HTML tags. This first set are simple tags that will inherit from `Tag` or `BlockTag`.

- `Img` - Start with the `<img>` tag. In HTML, this tag includes an image to be displayed on the page. The `Img` class should inherit from `Tag`, and should require two arguments to its constructor (in addition to `self`). The first is the `src` attribute (the URL identifying where the image can be found), and the `alt` attribute (text to be displayed if the browser cannot display the image for some reason). The constructor should add these as attributes. Do not forget to call the superclass constructor and supply it with the tag name. No other methods are required.

  Example:

  ```
  t = Img('/kitten.jpg', 'A cute, fuzzy kitten')
  print t


  # prints: '<img src="/kitten.jpg" alt="A cute, fuzzy kitten" />'
  ```

- `Meta` - Next, implement the `<meta>` tag. This is used as part of the page header. We will only use it in a specific way, so we will implement that specific functionality and ignore other possibilities.

  `Meta` should extend `BlockTag`. It should call its superclass constructor with the tag name 'meta'. In addition, it should add two attributes from within the constructor:

  ```
  self.addAttr('http-equiv', 'content-type')
  self.addAttr('content', 'text/html; charset=utf-8')
  ```

  No other methods are required.

  Example:

  ```
  t = Meta()
  print t


  # prints: '\n<meta http-equiv="content-type"
             content="text/html; charset=utf-8" />'
  # note: We added a line break to make it fit on the page
  ```

- `Stylesheet` - The page will need to link to its CSS stylesheet. This uses a `<link>` tag like this:

```
<link rel="stylesheet" href="blog.css" type="text/css" />
```

where `blog.css` is the name of the stylesheet file. Create a `Stylesheet` class that extends `BlockTag` to create a tag of this form. It should produce the output given above when used as follows:

```
t = Stylesheet('blog.css')
print t
```

- `Br` - The page will need to use some HTML line breaks. They are created with the `<br />` tag like this:

```
<br />
```

Create a `Br` class that extends `Tag` to create a tag of this form. It should produce the output given above when used as follows:

```
t = Br()
print t
```

- `Hr` - The page will need to use some horizontal rules. They are created with the `<hr />` tag like this:

```
<hr />
```

Create a `Hr` class that extends `Tag` to create a tag of this form. It should produce the output given above when used as follows:

```
t = Hr()
print t
```

Container tags

Each of the remaining classes that you must define inherits from either `Container` or `BlockContainer`. For each one, examples of how it will be used are provided, and you must infer the constructor parameters and functionality for each one. Except where otherwise noted, each one needs a new constructor and nothing else. Do not forget to call the superclass constructor in each case.

Inheriting from `Container`

- `A`: The anchor class, used for creating links:

```
t = A('/target/url')
# t renders as: '<a href="/target/url" />'


t.addText('target text')
# t renders as: '<a href="/target/url">target text</a>'
```

Inheriting from `BlockContainer`

- `Title`: The title of the page (for display in the browser's title bar):

```
t = Title()
# t renders as: '\n<title />'


t.addText('My blog')
# t renders as: '\n<title>My blog</title>'
```

- `Div`: A generic block-level container used to organize the page:

```
t = Div()
# t renders as: '\n<div />'
```

- `Strong`: A block container used to cause text to be drawn in a stronger font.

```
t = Strong().addText('Bold Text')
# t renders as: '\n<strong>Bold Text</strong>'
```

- `Em`: A block container used to cause text to be drawn in an italic font.

```
t = Em().addText('Emphasized Text')
# t renders as: '\n<em>Emphasized Text</em>'
```

- `P`: A paragraph of text:

```
t = P()
# t renders as: '\n<p />'
```

- `H`: A heading, requiring a parameter that is a value from one to six:

```
t = H(1)
# t renders as: '\n<h1 />'


t = H(2)
# t renders as: '\n<h2 />'
```

  If the value is less than one, use one. If the value is greater than six, use six. Be sure to convert the value into an integer.

  Examples:

```
t = H(0)
# t renders as: '\n<h1 />'


t = H(7)
# t renders as: '\n<h6 />'


t = H("4")
# t renders as: '\n<h4 />'
```

- `Ul`: An unordered list (usually displayed with bullets):

```
t = Ul()
# t renders as: '\n<ul />'
```

- `Ol`: An ordered list (usually displayed with numbers):

```
t = Ol()
# t renders as: '\n<ol />'
```

- `Li`: A list item for an ordered or unordered list:

```
t = Li()
# t renders as '\n<li />'
```

- `Head`: The head of the HTML document. It is normally filled with metadata, including the title of the document and a link to the stylesheet.

```
t = Head()
# t renders as '\n<head />'
```

  Nothing in this tag is rendered directly on the page, though the title (which is usually set inside here) is usually displayed in the title bar of the browser.

- `Body`: The body of the HTML document (the head contains metadata, the body the actual contents to be displayed):

```
t = Body()
# t renders as '\n<body />'
```

- `Html`: The tag that contains the entire page. To conform to the XHTML 1.0 Transitional standard, it should add a

`xmlns` attribute to itself, and should prepend the tag itself with a special `DOCTYPE` line (which is always exactly the same). Notice, that you will need to add a `__str__` method to facilitate this additional text.

`Html` should inherit from `BlockContainer`. Example:

```
t = Html()
t.addTag(Body())
print t


# prints:
# <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
#    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
# <html xmlns="http://www.w3.org/1999/xhtml">
# <body />
# </html>
```

Testing

Once you have implemented all of these classes, you can run `create_cs1410.py`, from `html_starter.zip` and it should create a file named `student-cs1410.html` that exactly matches the contents of `cs1410.html`.

Part II Pass-off

Demonstrate to the lab assistant that your `student-cs1410.html` is created by `create_cs1410.py` with your `html.py` and is identical to `cs1410.html`. Submit your `html.py` file for part 2 of this assignment.