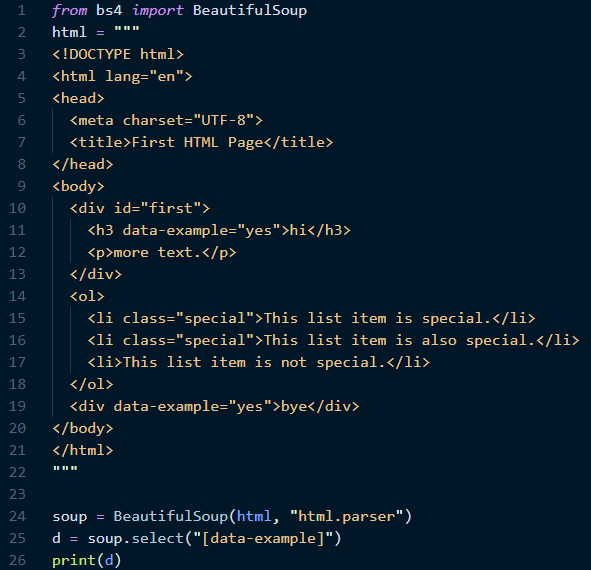
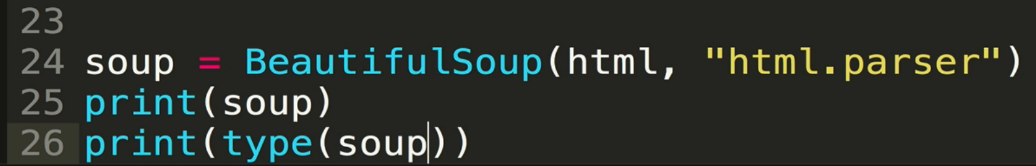
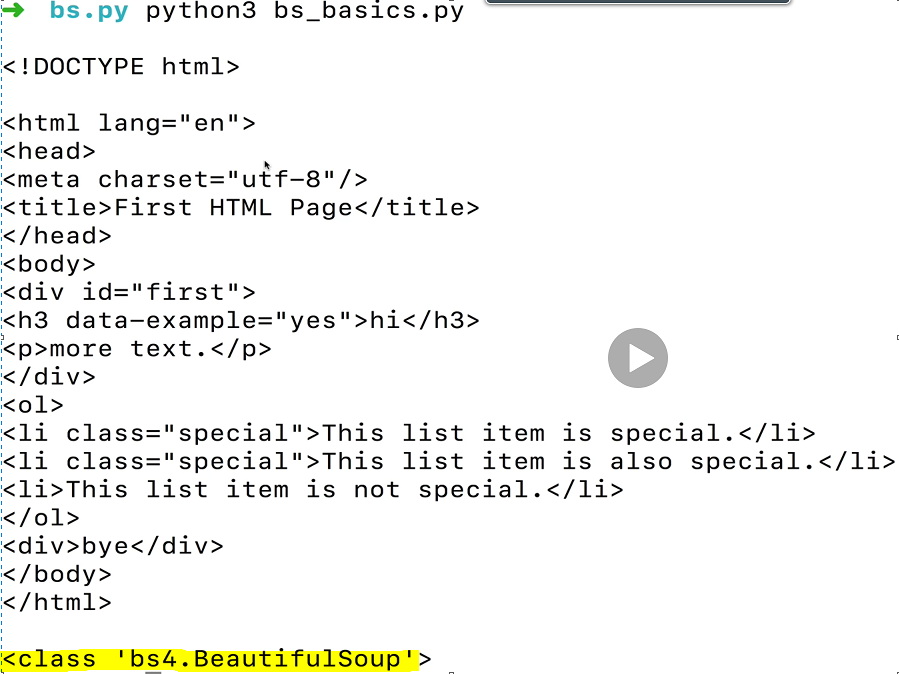
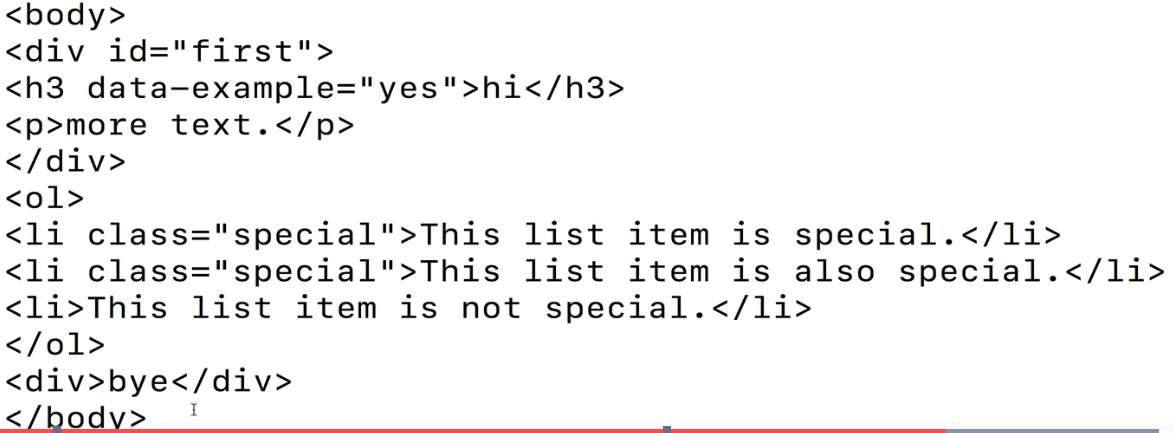
* Introduction to Web Scraping
  + Web scraping simply refers to programmatically grabbing data from a web page
    - This specifically refers to web pages in the form in HTML
  + Three steps to this
    - Download the data (HTML)
    - Parse through and extract the data
    - Then do something with the data: manipulate, aggregate, deposit into a database, etc
* Why web scrape?
  + Oftentimes used for when a website has data that you want to collect but does not provide it in an API
    - Example: If you wanted to collect rent data for listings on Craigslist
  + Can sometimes be messy when things are “buried” in the HTML
  + Another option is to manually copy and paste the data that you want, but who wants to do that?
* The ethics of web scraping – is it okay to do?
  + Many websites don’t want people to scrape them
  + The best practice is to consult the **robots.txt** file
    - This file is a website’s way of telling people which pages they want or do not want people to scrape
  + If you are making many requests to a website, time them out
  + If you are too aggressive with your web scraping, your IP address can be blocked
* **Beautiful Soup** is a Python library that is used for webscraping
  + The actual package name is *bs4*, and needs to be installed to your Python environment using *pip*
  + BS is used to extract data from HTML
  + BS lets us navigate through HTML with Python
  + However, BS does NOT download HTML for us. For this, we need to manually make a request using the *requests* module
  + Basic example with mock HTML:
    - Start by importing BeautifulSoup
      * from bs4 import BeautifulSoup (or BS)
    - Then parse the HTML using the html **parser**
      * Remember that an HTML request comes back as a giant string that needs to be passed
      * BS also supports parsing XML, so for HTML we need to specify HTML
    - After parsing, the data is returned as a BeautifulSoup object, NOT a string. This BS essentially contains numerous “sub-objects” for each of the different tags that was found in the HTML



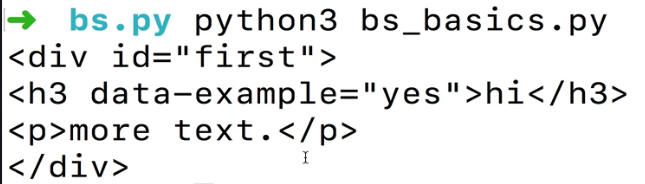




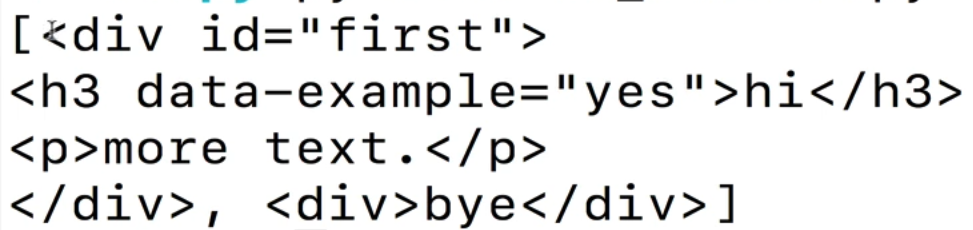
* + - Let’s take a look at the parsed body using *soup.body*



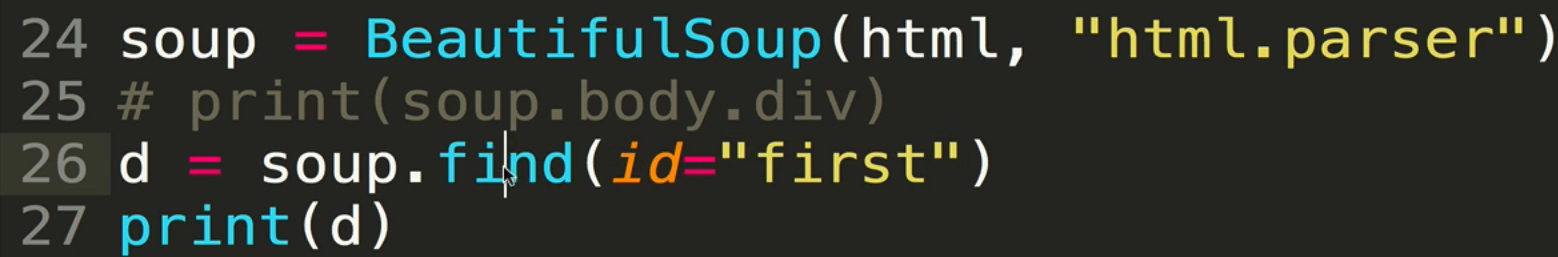
* + - Once parsed, there are several ways to navigate. One is by using the Tag name (<h1>, <a>, <p>, <div>, etc.)
      * Use the **find()** method to find the first matching tag
      * Use **find\_all()** to find all matching tags. Returns the matching tags as a list
      * How about finding the *divs* within the body? Let’s use *soup.body.div*. Notice that it only returns the first div in this HTML body (there are two of them total)
        + We could have also used *soup.find(“div”)* to do the same search
        + Note that this search

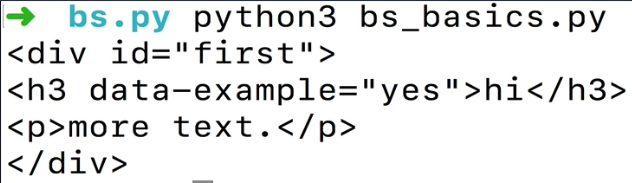


* + - * If we want to find all divs in the soup object, we can use *find\_all()*
        + *soup.find\_all(“div”)* on our example returns the following, a list containing each div in the soup

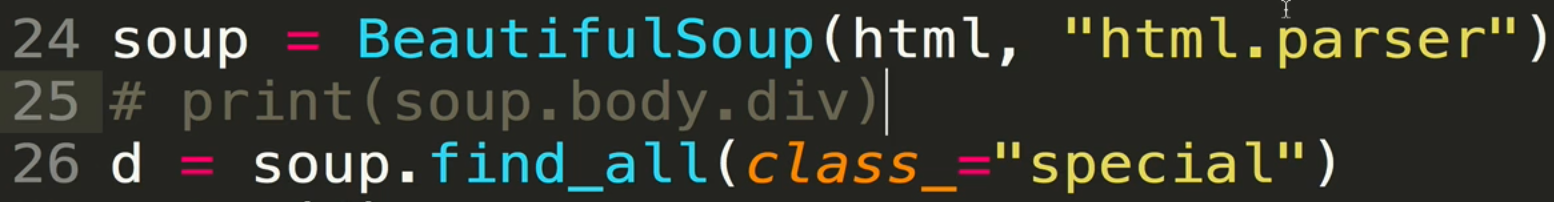


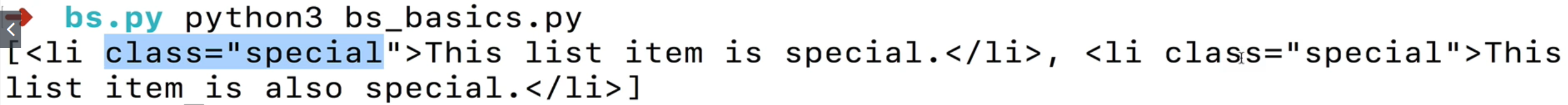
* + - We can also find using **tag IDs** (instead of tag name). In this example, the HTML element that has the id “first” will be returned
      * Remember that HTML id’s are only intended to be used once.



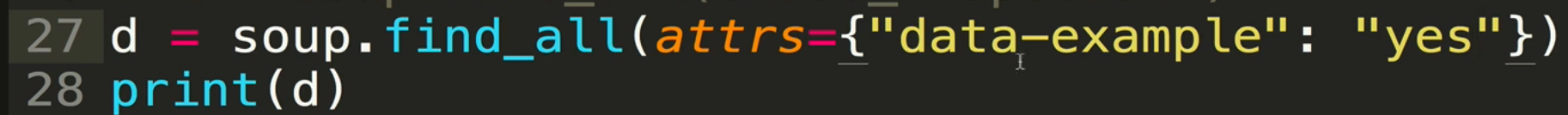


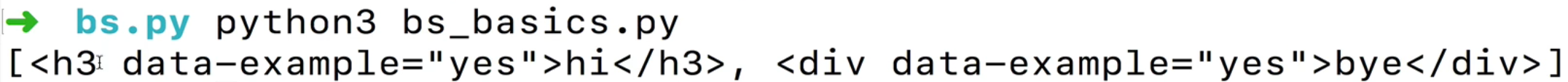
* + - We can also find using **class names** (instead of tag type). In this example, we use **find\_all()** to return all elements with that class name
      * Class names ARE intended to be used more than once, and so it makes sense to use find\_all() instead of find()
      * Remember that **class** is a reserved word in Python, so you need to use **class\_** instead





* + - Can we select based on an *HTML attribute* (instead of a tag name, class name, or ID? Sure we can! We do this using the **attrs** keyword, and we pass in a dictionary containing the attribute name and the value we want to looks for:

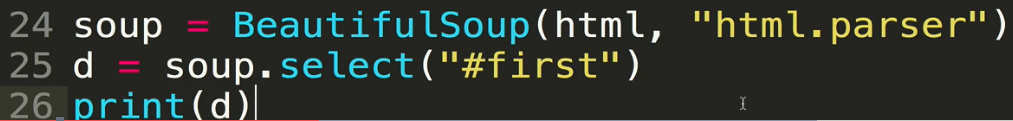


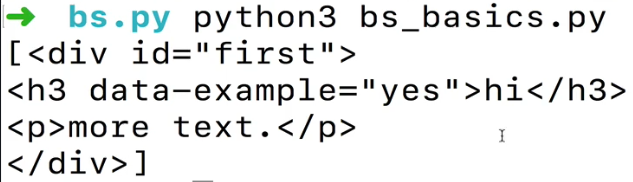


* + - Summary of selecting using find() or find\_all()

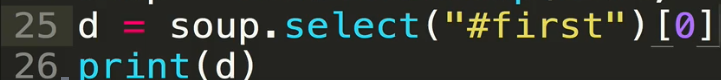
|  |  |
| --- | --- |
| **Approach** | **Example** |
| Tag Name | soup.find\_all(“h1”) |
| Tag ID | soup.find(id=”tag ID”) |
| Class Name | soup.find\_all(class\_=“class name”) |
| HTML attribute | soup.find\_all(attrs={“attribute\_name”:“attribute\_value”} |

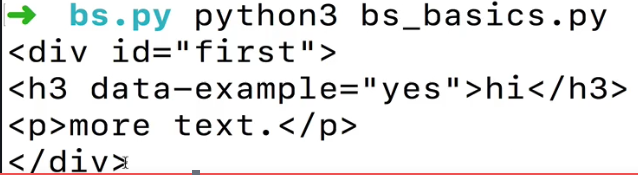
* + Selecting with CSS style selectors. This usually results in nicer, cleaner code, but may not be great for those who are unfamiliar with CSS
    - We use the **select()** method for this. Note that when you use select, a list is ALWAYS returned
      * This means that if you want the actual data, you need to select the indexed element
    - Selecting by **ID** using the octothorp (hashtag, #). In this example, we select the element with the ID “first”



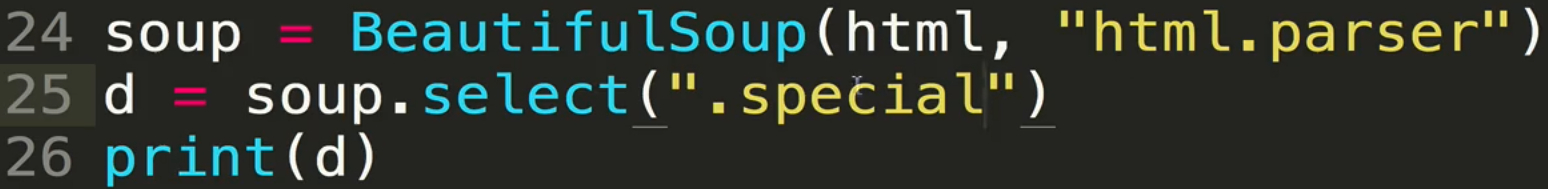


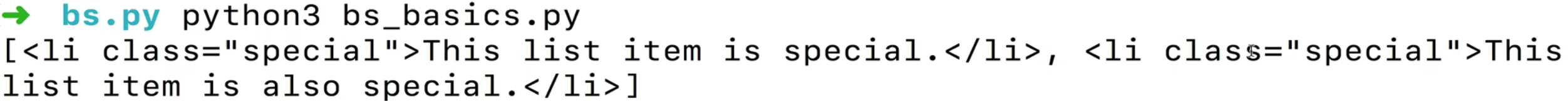
* + - * Now access the actual data within the list. Notice that the data is no longer in a list:



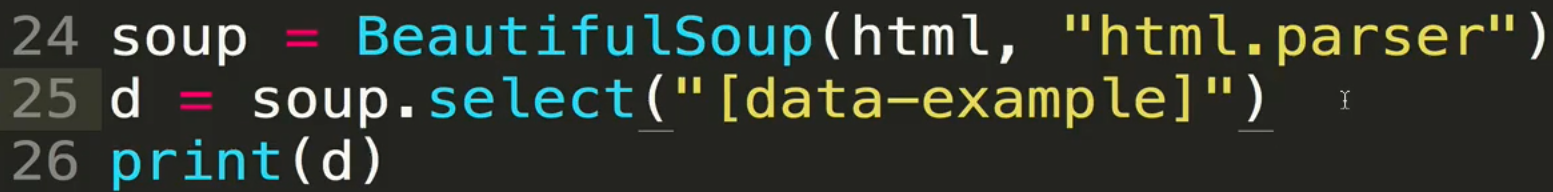


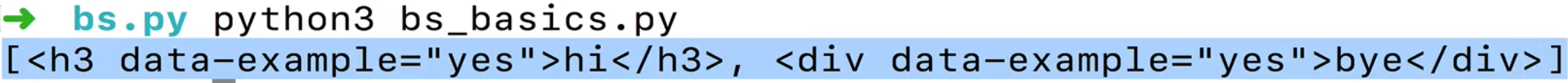
* + - Selecting by **class**. We use the **select(“.class\_name”)** syntax for this. Again, it returns a list of all elements with that class name, and to get at the data you need to select it by index.





* + - Selection by **attribute**. We use **the select(“[attribute-name]”)** syntax





* + - We can also select the attribute AND value:

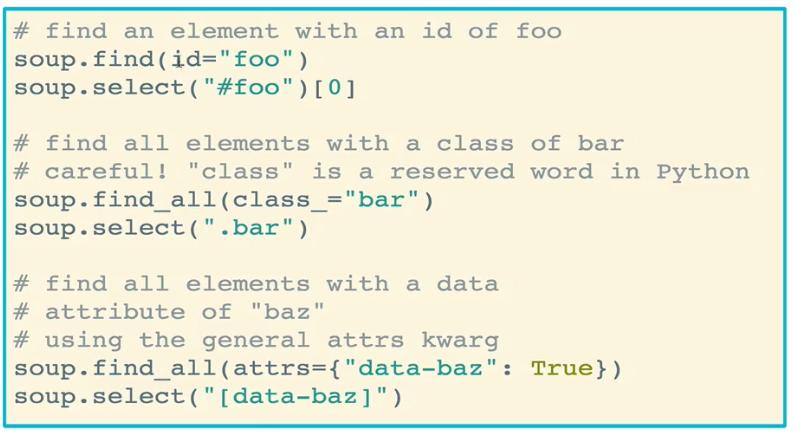




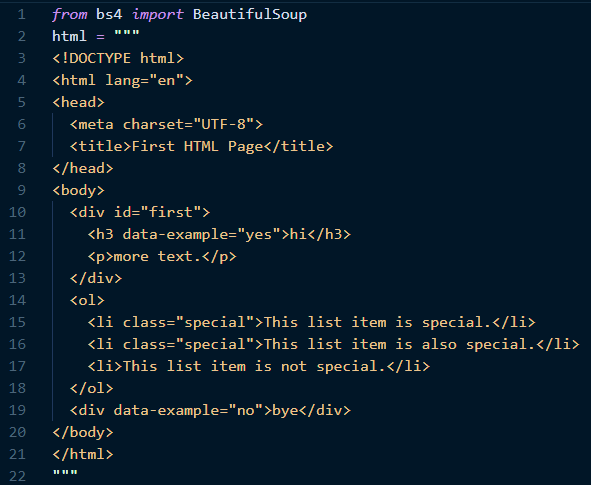
* + - Summary of selection using **select()**

|  |  |
| --- | --- |
| **Approach** | **Example** |
| id | soup.select(“#id-name”) |
| Class name | soup.select(“.class\_name”) |
| HTML attribute | soup.select(“[attribute\_name]”) |
| soup.select(“[attribute\_name = ‘attribute\_value’]”) |

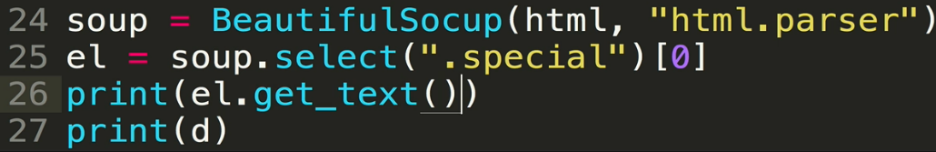
* + Direct comparison between selection using *find* and selection using *select*

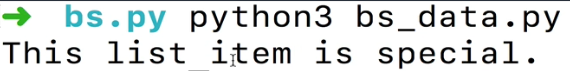


* We’ve parsed our HTML and selected our data in BS. Now how do we read it out?
  + Several approaches (not an exhaustive list)
    - **get\_text()** – access the inner text in an element. Note that this is a method
    - **name** – get the tag name of a given element
    - **attrs** – dictionary of attributes of the lement
      * Can also access attributes using brackets
  + Let’s use the same HTML block as above for our example.

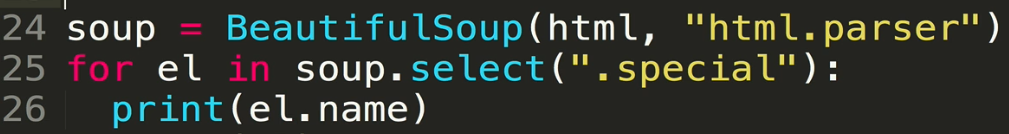


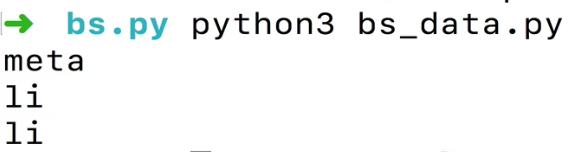
* + - Start by selecting. In this case we’ll use the first element belonging to the class “special”. Remember that a list is returned, so we need to access the first element through indexing
      * Next, we use the **element.get\_text()** method to get the actual text
      * Note that the element itself is an object



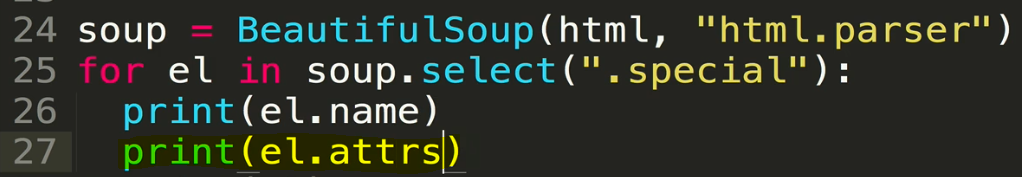


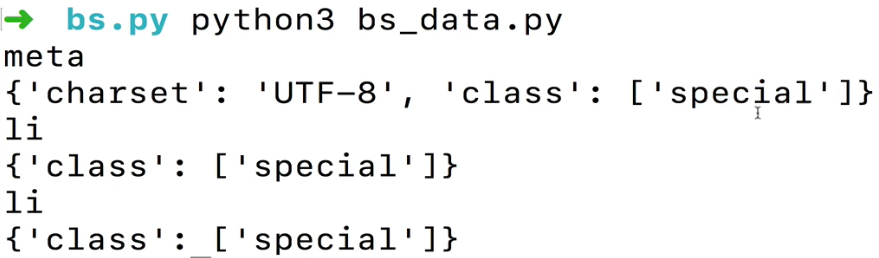
* + - Let’s now access the name of the element, using **element.name**, which refers the name of the tag associated with that element, such as “li” or “div”. In our modified example, there are three elements that are “special”: one “meta” and two “li”s
      * Also in this example, since select() returns a list, we can iterate through that list and access the tag name of each element in that list. Iterating is fairly common, but more often than not we are more interested in the data (e.g. text) than the tag name.



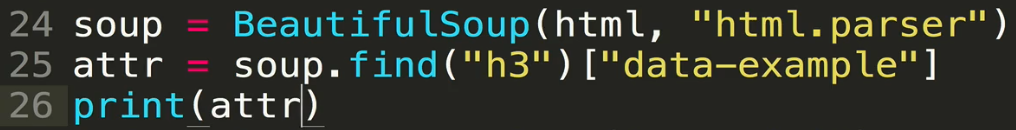


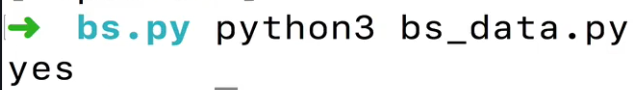
* + - Let’s access the key-value attribute pairs for the selected items using **element.attrs**. For every element selected, this will return a dictionary of the key-value pairs for that element, which you can then access at your will
      * Note that the **class** key has a list as a value since an element can have more than one class





* + - A cleaner way to access attributes: use bracket notation! Here, we want to access the “data-example” attribute for the first <h3> element. Spoiler: its value is “yes” in this example

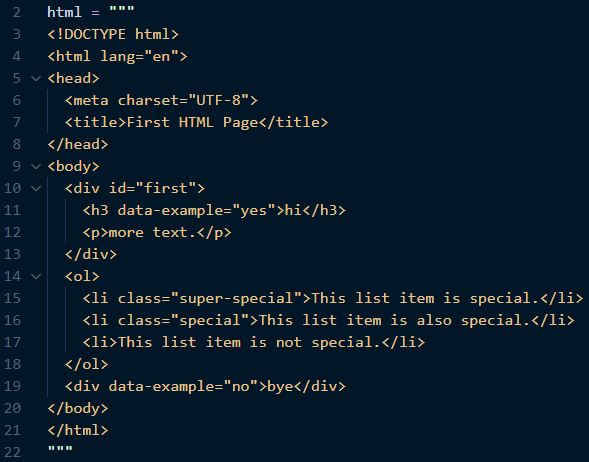




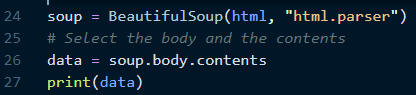
* + Summary for accessing data in elements that have been selected as *element*

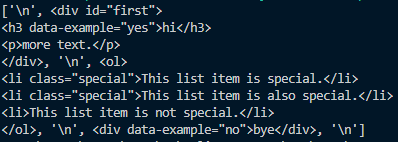
|  |  |
| --- | --- |
| **Approach** | **Example** |
| Get text data | element.get\_text() |
| Get the tag name of the element | element.name |
| Get the attribute(s) of the element | element.attrs – returns a dictionary of all attributes |
| element[“attribute\_name”] – returns the value of that one attribute |

* **Navigating** with Beautiful Soup refers to finding elements relative to an element that you have selected
  + For example, if you have selected an <li> tag, you can ask BS to find the parent of that tag, or the parent of the parent of that tag, OR the next sibling/descendent of that <li>
  + One way to do this is through **tags**:
    - parent/parents
    - contents
    - next\_sibling/next\_sublings
    - previous\_sibling/previous\_siblings
  + Another way is through **Searching**, consisting of methods
    - find\_parent / find\_parents
    - find\_next\_sibling / find\_next\_siblings
    - find\_previous\_sibling / find\_previous\_siblings
* Navigation Example – refer to our sample HTML code again



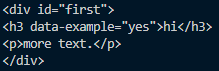
* The common approach here is to find an element to “hook” into that is close to the data you are trying to scrape, and then navigate the contents from there by selecting parents and children
  + First, attempt to print out the contents of the body by using soup.body.**contents**. Notice that we get a list of all contents of the body, including newline (“\n”) characters



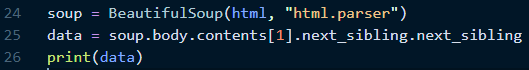


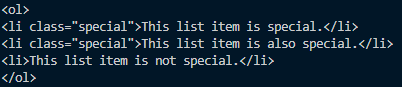
* + Next, to access the first “child” of the body that we care about, in this case the div, we use soup.body.contents[1], which returns the contents of the div. You can also call .contents on that selection to get all of the contents as a list



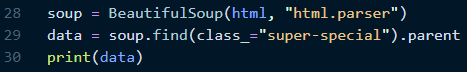


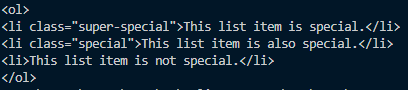
* + What happens if we have selected an element, and now want to select the next sibling element? We can do this using next\_sibling, which will select the next sibling tag. Note that in this example, since the next sibling is a newline, we’ll need to use next\_sibling again to actually get the next useful element



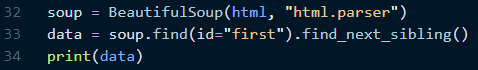


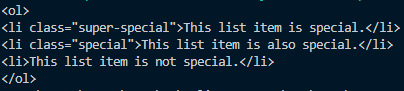
* + We can use a similar approach to select parents. In this example, we select an <li> of the class “super-special”, and then call the .parent command to select the parent <ol>



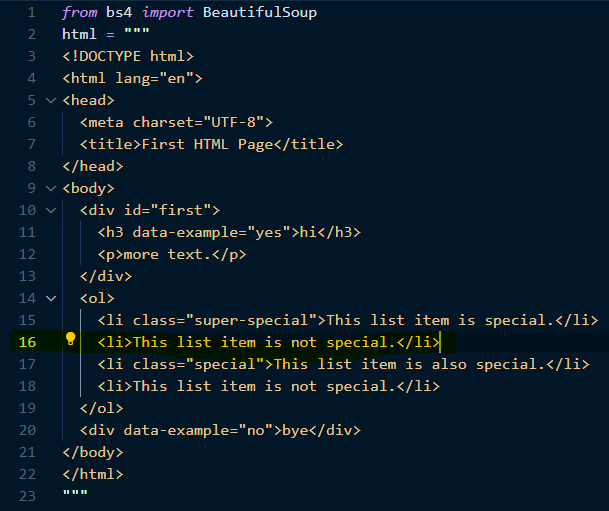


* Navigating with search methods
  + You can select an element and then call the **find\_next\_sibling()** method to get the next sibling element (same hierarchical level). In this case, the div with the id “first” is within a body tag and its next sibling that is not a newline is an <ol>
    - This method is set up to give us the next sibling that is NOT a newline**.** This is an advantage of using find\_next\_sibling





* + You can also pass in arguments to the search methods. This is useful when, for example, you want to find the next sibling of a particular class. Let’s start by modifying our HTML a bit to introduce a new non-special line item



* + - Next, let’s use find\_next\_sibling(), but pass in class\_ = “special”. Note that the search skips over the next “non-special” sibling and selects the next “special” one!

