**Team e + π**

**Introduction:**

Ezget is a tool that allows the user to download web content from a command line. First it will prompt the user for a URL address and check to see if it’s valid. If not valid, it will tell the user what is wrong with the address given. If the URL is valid, Ezget will ask for a depth value. This depth will act as the number of layers the program will crawl through. The retrieved web pages will be stored and displayed on the screen once finished. They will be in order from the earliest layers, to the later layers.

**System Overview:**

* At the beginning of planning the architecture of this code, we tried both C as a language and Python. Because Python has many modules that can be imported, it made it much easier to finish this task efficiently and to not re-invent the wheel. Programmers building off one another brings society as a whole that much more advanced with its technology due to not wasting valuable time doing some task that has already been solved. For this reason, using Python as the chosen language was a no brainer.
* In order to run this program, bs4 will need to be manually installed in a Python3 compiler. Without either of these, you will be unable to run this software. If not known how to pip install a module, there are online compilers like Repl that do everything for you.
* One major problem that arose was adding the ability to terminate threads. One of the objectives was to set a 60 second timer that will end the program is there is no user input. The issue maybe specifically in Python or in general is it requires creating processes and sub-processes as a way to have multiple things run at the same time. This requires setting the timer in one thread, and the input in another separate thread. The issue was not the ability to create and start threads, it was the ability to turn off the threads once completed.
  + Daemon threads are supposed to do this automatically but due to the input prompt acting as a sort of loop, it was preventing a daemon thread to stop because it is not completed.
  + The closest way of getting it to work involved using a combination async, await and yield as it asynchronously accepts an input then uses an exception call to theoretically stop the thread. The idea is the same as a web call service as it waits for a server to respond from a request, then times out.
  + My theory is the declarations and process variables were out of scope which prevented the ability to control one another.
* Possible improvements to this program are adding functionality such as a GUI, artificially predicting and auto correcting errors in the URLs, and the ability to be able to pick if you want to download the webpages for example in HTML format or PDF.
* Throughout programming this code, there are many references to two movies specifically, Shrek and The Matrix. Each adding their own pungent but fulfilling satire that is intended to make a previously monotonous looking program a little more idiosyncratically interesting. As this captivating paradigm of coding will not help or meet code requirements in business, it will instead resuscitate the coders soul.

**System Architecture:**

First there is a greeting that explains and detail what this program is in the if \_\_name\_\_ == '\_\_main\_\_'. Next it will tell the user what to type in giving an example of the format the URL needs to be. The following while loops and if statements are put in place to elevate any errors from the given URL. If any of these if statements find errors, it will print what is wrong for the user to fix it in the next iteration.

* First check is to see if the beginning of the string has “<https://www>.” as this is required for the other module.
* Next checks if the end of the string has “.com/”, again as this is required to elevate any errors.
* Following is to check if the string given is not empty.
* Once both the beginning and end of the local string is checked, next is to check if the website is valid or up and running by using URLparse. The string will be checked and sent to the method called String\_checker that will attempt to parse the URI object on whether the port is active as well as parsing the string format, returning in first level domain format. This parsing using .scheme, .netloc and .path as a way to index and search the queries from the object given. This acts as checking the internet part of the URL given. If there is a response from the parser, it returns True to main method.
* The next check is see if the HTML request is valid. The string was checked to be in the correct format, but now it checks if the URL is a valid address from an internet standpoint. Using a .get command, it will use the request and urllib module to check the status of the URL, returning true if up and running and valid domain name, as well as checking if for example a firewall or some other connection error arises. If any errors, it throws an exception.

Next prompt is to ask the user for the level of depth they want to crawl through. This is straight forward, basically checking if the string of the depth is a positive integer, and less than 1000. Reasoning is to create a reasonable number of layers to download, also preventing any errors that can arise from being out of scope.

* First checks if the depth given is empty as we need a number for the depth simply by putting if (not string).
* Next to see if the string is a digit using .isDigit(). Because its checking if a string is an integer, it needs to be casted as an int.
* Next finds whether is less then 1000 due to many other factors such as computing speed, internet bandwidth and realistic amount of web pages, 1000 is the limit.
* The following is just to add a little sarcasm to the code. If the inputs go above 3, it brings the sass.

As a final catch for any possible errors that slipped through the cracks, it confirms urllib.request and urlparse is valid and true. Since the URL provided is valid, it stores that first in the web\_array. Once confirmed that url address and depth is valid, it calls the crawlin method that will actually download the web pages and store them in an array. It uses the module called BeautifulSoup from bs4, (how elegant of a name), to do this.

* The method crawlin receives the web address and depth as parameters and creates an array to store the URLs.
* Using the depth provided, using a for loop to crawl through however many iterations is set by user.
* In the for loop, if web address is not in the list, it appends or adds new URL to list. Using the urllib2.urlopen module, it tries to open the URL storing it as site. Otherwise prints that the found URL is not able to be opened.
* Storing the opened url as site, next uses the BeautifulSoup module to read and parse the HTML address in the web page. If the depth is more than 1, it will crawl through or parse more sub-links to store as a Tree list.
* Setting this Tree as a for loop, if the depth is greater than 1, it will scan the text in the read file searching for the anchor tag, ‘href’, (storing it as dict paraphrase). By doing this it is intended to parse the base url with other anchored webpages, fetching the last string.
* Once anchored webpages are saved as a Tree, it will then ‘split’ or return the string, ‘#’ getting the filename of the URL, while also keeping its parameters by setting it as [0].
* After splitting the anchored webpage, it continues to clean up and separate the real URLs from the garbage HTML code in the webpage by parsing the first 4 chars of http.
* It then appends or adds the crawled through url to the array list.

Once all the iterations are done which are set by the user’s depth, it points the url\_void array list to the parameter pages, returning it back. The returned array list of crawled through webpages will then be printed in the terminal.

**Roles and Responsibilities**

* Alex – Programmer and Designer
* Reese – Programmer and Designer