Learning Outcomes: Throughout this HW, you will work to:

- Understand the distinction between an ADT and its interface, versus the data structure used in the underlying implementation.
- Understand one good use of a stack.
- Learn how HTML — and more generally, XML — is structured.
- Continue use of command-line arguments in Python.

Assignment: All code should go in a file named `dcs229_hw5.py`.

1. Write a new function named `readFile` that accepts a single string argument corresponding to a filename. Include code that will open and read a file whose name is provided as the argument. For opening and reading the file, use Python's "`with open() as`" pattern (see `https://realpython.com/lessons/with-open-pattern/`) and `read()` to read the HTML file. Return the contents of the files as a single string. In your docstring, include a "Raises" block indicating that your function may raise a `FileNotFoundException` (via `open`).

2. In `main`, pass `sys.argv[1]` (see the "Refresher" on command-line arguments at the end of this document) as the first argument to `readFile` so that the user can provide the filename as a command-line argument. In the example execution below:

   ```
   % python dcs229_hw5.py little_boat.html
   ```

   the user will be asking the program to read and process the file named `little_boat.html`.

   Include the above logic within a try-except block, looking for two different possible exceptions:

   (a) An `IndexError` exception that will allow you to handle the case when the user has not provided an appropriate number of command-line arguments. Print a useful error message (detailing how to use your program), and call `sys.exit(2)` — see `https://docs.python.org/3/library/sys.html#sys.exit`.

   (b) A `FileNotFoundError` exception — which `readFile` may raise – that will allow you to handle the case when the user provides the name of a file that does not exist. Print a useful error message, and call `sys.exit(1)` — again, see `https://docs.python.org/3/library/sys.html#sys.exit`.

3. Write a new function named `parseHTML` having a single `str` parameter corresponding to the entire HTML text read and returned by `readFile`. Your function must return `True` if the HTML consists entirely of properly-matched tags, or `False` if there is any improperly-matched tag. If the HTML is not properly matched, you must also print a description of what went wrong. Below are two different examples of ways a file could fail, and appropriate printed messages:

   ```
   mismatched <html> to </i>

   unmatched tags: <head>,<html>
   ```

   You must use the (provided) `Stack` class in your solution. (You are not allowed to use `BeautifulSoup` or similar libraries to parse the HTML — rather, use native-Python string methods.)

   Your solution must consider so-called "empty tags" (e.g., `<br/>`) which do not have separate opening and closing tags. A list of such tags can be found here. Note that empty tags are sometimes found with a forward slash, sometimes without (i.e., either `<br/>` or `<br>`) — make sure that your solution works in either case.

   Include appropriate type hinting and a complete docstring.

4. Test your solution carefully on a variety of HTML files. On Lyceum, I have provided the HTML file `little_boat.html` having properly matched tags. Copy and modify this file to test your implementation on different valid and invalid tag pairings.

Reflection: In the online Lyceum text, provide answers to each of the following:

1. Discuss one additional application of where use of a stack would apply — be specific in how you would use `push` and `pop` in the application.
2. What was challenging on this assignment, if anything?
3. What do you still need work on, in terms of concepts covered in this assignment?
4. List all online-help resources you used in completing this assignment (which you should have also included as comments in your program's top block comment). You do not have to cite Python's documentation.

Submitting: When done, upload your `dcs229_hw5.py` solution to Lyceum.

Refresher: Python Command-Line Arguments

- When you run your program at the command line, you can specify so-called command-line arguments to your program. (This was used in your Lab 3 assignment as well.) To do so, you will need to `import sys` and then leverage the included `sys.argv` list.
- For example, given the simple program below:

  ```
  import sys

  def main() -> None:
      print(f"sys.argv = {sys.argv}")
      print(f"name of program = {sys.argv[0]}")
      for i in range(1, len(sys.argv)):
          print(f"arg {i} = {sys.argv[i]} \t type = {type(sys.argv[i])}")

  if __name__ == "__main__":
      main()
  ```

  when you execute the program, you will see how command-line arguments work in Python.
- Try the above simple program. Run the program with a varying number of arguments to your program, as in:

  ```
  % python cla.py
  sys.argv = ['cla.py']
  name of program = cla.py

  % python cla.py 2 blue
  sys.argv = ['cla.py', '2', 'blue']
  name of program = cla.py
  arg 1 = 2       type = <class 'str'>
  arg 2 = blue    type = <class 'str'>
  ```