Goals:

- Implement the MyPL lexical analyzer;
- Practice working with unit tests.

Instructions:

1. Use the GitHub Classroom link (posted in Piazza) to copy the starter code into your own repository. Clone the repository in the directory where you will be working on the assignment.
2. Complete the `next_token()` function in `mypl_lexer.py`.
3. Ensure your code passes the unit tests provided in `hw1_tests.py`. (Note you will want to do steps 2 and 3 iteratively.)
4. Ensure your lexer implementation correctly handles the example files within the `examples` subdirectory.
5. Create an additional example test file called `hw1_example.mypl` for this assignment and ensure your program works correctly over the file. (Be sure to save this file in the main source code directory to receive credit, not within the `examples` subdirectory.)
6. Create a short write up as a **pdf file** named `hw1-writeup.pdf`. For this assignment, your write up should provide a short description of any challenges and/or issues you faced in finishing the assignment and how you addressed them along with a brief description of the tests you created in the example file.
7. Submit your program by ensuring all of your code, test file, and writeup is pushed to your GitHub repo. You can verify that your work has been submitted via the GitHub page for your repo.

Additional Requirements: Note that in addition to items listed below, details will also be discussed in class and in lecture notes.

1. It is fine to implement the `next_token()` function without breaking it into separate helper functions (i.e., you can have it be one large function). However, if you would like to "modularize" it, you are welcome to. If you do break it out into helper functions, you must explain how you did this in your writeup.
2. You must implement your `next_token()` function by reading one character at a time via the `read()` and `peek()` helper functions provided in the `Lexer` class. In addition, to report errors, you must use the `error()` helper function provided by the Lexer class. The `eof()` file is also provided to help you check for an EOF (end of file) character.

3. Python provides some useful helper functions for checking for specific types of characters. In particular, I used the `isspace()` (check for whitespace, which includes newlines, tabs, and so on), `isdecimal()` (to check for a digit in our case), and `isalpha()` (to check for a letter in our case) functions. Note these are called on a string. Since we are reading one character at a time, where the next character is held in the `ch` variable, you would write `ch.isspace()`, e.g., to call the helper function.

4. The full set of token types for MyPL are provided in the `mypl_token.py` file. Note that your `next_token()` function is creating and returning `Token` objects with these listed types.

5. Note that `next_token()` uses an iterator, i.e., stream-based, model. This means that one call to `next_token()` returns only the next token in the input. The `Lexer` object maintains state, including where it is in the current input (to be able to return the next token in the input, and so on).

6. The `Lexer` class maintains `line` and `column` member variables. These variables are to keep track of the current line and column for building tokens. Your `next_token()` function will need to update these member variables and also use them to build up new token objects.

7. Each token should have a non-empty lexeme. For tokens with "unimportant" lexemes, you can just use their corresponding symbol. For example, the lexeme for + should be `"+"` and the lexeme for `int` should be `"int"`.

8. The `hello.out` and `tokens.out` files within the `example` subdirectory give examples of what your results for running `./mypl --lex` on the files should be. Your program must output the exact same information as what is in these files to be considered correct. Note that additional example input files are also included, which will also be used as tests of your program.

9. A non-comprehensive set of unit tests are provided in the file `hw1_tests.py`. To run these tests, simply type `pytest hw1_tests.py` at the command line. (Note that many IDEs, including VS Code, provide integrated support for pytest, but using this is not requried.) Your implementation will need to pass all of the unit tests from `hw1_tests` to be considered correct. Additional options for `pytest` that are useful include `pytest -v`, which expands the description of each test, and `-x` to exit instantly on the first error or failed test (which reduces output as you are trying to get tests to pass). Many more options are available and can be seen using the `-h` option.

Hints and Tips:

1. The basic layout for `next_token()` that I used in my implementation is, in order: (1) read all whitespace (checking for EOF); (2) check for EOF; (3) check for single character tokens (e.g., arithmetic operators, punctuation, etc.); (4) check for the trickier symbols that can involve or require two characters (e.g., < vs <=, `!=`, and so on); (5) check for string values; (6) check for integer and double values; (7) check for reserved words; and then (8) identifiers. Again, it is much easier to do this incrementally as opposed to all at once and then try to debug.

2. You are encouraged (but not required) to create your own unit tests. Note that the unit tests provided are not guaranteed to be comprehensive. As mentioned in the `hw1_tests.py` file, just because your program passes the unit tests *does not mean your code is correct!*

Homework Submission and Grading. Your homework will be graded using the files you have pushed to your GitHub repository. Thus, you must ensure that all of the files needed to compile and run your code have been successfully pushed to your GitHub repo for the assignment. Note that this also includes your homework writeup. This homework assignment is worth a total of 40 points. The points will be allocated according to the following.

1. **Correct and Complete (30 points).** Your homework will be evaluated using a variety of different tests (for most assignments, via unit tests as well as test runs using specific input files). Each failed test will result in a loss of 2 points. If 15 or more tests fail, but some tests pass, 6 points (out of the 30) will be awarded as partial credit. Note that all 30 points may be deducted if your code does not run, large portions of work are missing or incomplete (e.g., stubbed out), and/or the specified techniques, design, or instructions were not followed.

2. **Evidence and Quality of Testing (5 points).** For each assignment, you must provide additional tests that you used to ensure your program works correctly. Note that for most assignments, a specific set of tests will be requested. A score of 0 is given if no additional tests are provided, 1–4 points if the tests are only partially completed (e.g., missing tests) or the tests provided are of low quality, and 5 if the minimum number of tests are provided and are of sufficient quality.

3. **Clean Code (2 points).** In this class, "clean code" refers to consistent and proper code formatting (indentation, white space, new lines), use of appropriate comments throughout the code, no debugging output, no commented out code, meaningful variable names and helper functions (if allowed), and overall well-organized, efficient, and straightforward code that uses standard coding techniques. A score of 0 is given if there are major issues, 1 if there are minor issues, and 2 if the "cleanliness" of the code submitted is satisfactory for the assignment.

4. **Writeup (3 points).** Each assignment will require you to provide a small writeup addressing challenges you faced and how you addressed them as well as an explanation of the tests you developed. Additional items may also be requested depending on the assignment. Homework writeups do not need to be long, and instead, should be clear and concise. A score of 0 is given if no writeup is provided, 1 if parts are missing, and 2 if the writeup is satisfactory.