Documentation for Capital One Assessment

By: Shikhar Nandi

INTRODUCTION

The purpose of this document is to give a detailed explanation of how the application works, the organisation of the code, and what the purpose of every class, function is and how they work in a detailed explanation.

PURPOSE OF THE APPLICATION

The main purpose of the application is to extract particular details from a programming file. Programming files are files containing various code and are commonly defined by an extension ex. java, .py, .js, .c… Just to name a few. The application extracts six various details from the programming file. These are defined below.

Total Lines: The total number of lines that the file contains, this also includes empty lines.
Single Line Comments: The total number of lines that contain single comments.
Block Line Comment: The total number of blocks that contain comments.
Comment Line within block comments: The total number of lines within the blocks.
TODO Comments: The total number of lines that contain a TODO comment. * A TODO comment is also counted as a single line comment.
Comment Lines: The total number of lines that contain comments. This is essentially Single Line Comments + Comment Line within blocks comments.

ASSUMPTIONS

There are a number of assumptions we make to better define more unique cases, as while certain lines of code clearly fit the descriptions lined above, others need to made clear in order to avoid assumptions by the user. There are a variety of scenarios that could happen and thus a number of assumptions are made and are outlined below, to better help understand why certain things are counted and others not.

1. Block Line Comments are only counted if they use the block comment syntax of a specific language. Single Line Comments grouped together are not counted as a block of comments, but rather multiple single line comments.
2. Block Line Comments exclude the start of the block comment and end of the block comment when counting the number of comment lines within a block comment.
3. Block Line Comments that start and end on the same line are considered to be one block line comment, but do not contribute towards the number of comment lines within a block comment.
4. Single Line Comments and TODO Comments defined within a block Line Comment are not considered as part of the Single Line Comments and TODO Comments respectively.
5. The use of any form of comment syntax within quotations marks or within parenthesis are not counted as any form of comment.

TWO VERSIONS:

For the purpose of trying out different ideas, two versions of the extractinfo.oy were created. The reason by this approach was considering what we want to try and add in the future. The two versions are listed as extractinfo.py and extractinfo_2.py. While both versions may use similar approaches, the main difference comes from the way information is extracted, hence the difference in the extractinfo file name. These would be extractinfo.py and extractinfo_2.py. extractinfo.py uses a character by character parsing method, whereas extractinfo_2.py uses a line by line parsing method.

extractinfo.py:

This version makes use of the method of parsing the file character by character. The primary version, is able to generate a more accurate result, as going character by character allows us to handle special cases more efficiently. This is important because while we may hope people follow certain conventions, this will not always be the case, thus the ability to handle unique cases well is crucial, especially since different developers who could be using this may have their own styles of coding. However, the downside with this approach is if future requirements require the need to extract information not related to comments it can prove a bit difficult to start adapting those requirements. Nonetheless, this is the version I think is better of the two.

extractinfo_2.py:

This version makes use of the method of parsing the file line by line. The secondary version is able to make it much easier to adapt more and more different requirements if future requirements demand it but loses accuracy as checking line by line makes it harder to check for more special cases, especially if developers are not following certain conventions when writing code.

APPLICATION:

BRIEF SUMMARY

The application was written using Python Version 3.7 or later. The reasoning behind this was that python offered a minimal use of external packages and provides a more friendly way of tackling parsing. *Note, both application versions require python 3.7 or a later version to run. The application contains 4 files that are required for it to run.

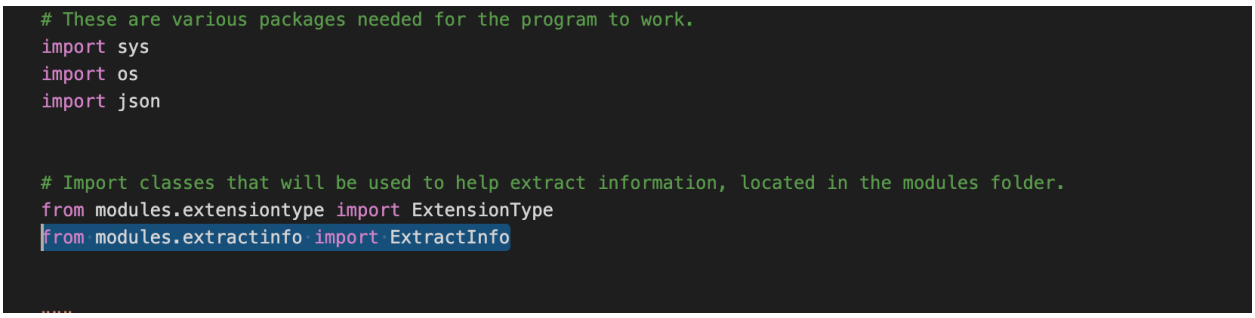
- main.py (file)
- modules (folder)
    - extractinfo.py (file, class)

    - extractinfo_2.py(file, alternate class)
    - extensiontype.py (file, class)
- syntaxes.py (file)

There are three packages both versions need in order run. These should already be installed in your computer, if you have Python 3.7 or later installed and therefore the code should run without any problem. They are listed below.

- import sys
- Import os
- Import json

In order to run the alternate approach to extractinfo.py, simply change the line highlighted below in main.py from 'from modules.extractinfo import ExtractInfo' to 'from modules.extractinfo_2 import ExtractInfo'



HOW TO RUN

The application is a terminal run application, which means it must be executed from the command line. The output is presented on the command line. Instructions to run are listed below.

1. Open your command line
2. cd into the assignment folder
3. To run write: python main.py [path-to-file-that-wants-to-be-parsed]
4. The result will show on your command line.

FILES:

The descriptions below pertain to each individual file that is needed to run the program and a general summary of what the file and what the functions in the file do. For more specific detail on the code look at comments in the code. Both versions used all the same identical files, with the only difference being extractinfo.py and extractinfo_2.py.

syntaxes.json

syntaxes.json is a file that stores all the syntax related to specific languages. The file is structured as follows. A key is an extension of the file ex. .java. Each key currently contains three parameters. If new languages need to be added they must follow the convention as shown in the picture below.

1. firstComment: The syntax for a single comment.
2. blockComment: The syntax for a block comment. (Block comments are commonly represented by an open and closed syntax, hence a need for an array, where the first element represents the open part and the second the corresponding closed.
3. todo: The syntax for a todo comment

The reason why JSON was used as a means of storage is that provides great flexibility, if one wants the program to work with additional languages. Provided the user follows the structure explained above any language can be added and nothing in the program would need to be changed. That would only happen if a new type of comment is introduced.

main.py:

Summary:

The main.py is the file that executes the application. It consists of a variety of functions as listed below. The listing showcases the functions used in this file and what they do.

Functions:

1. If __name__ == “__main__”

This executes the program and also checks whether a file was actually given for parsing. It then passed on the file for parsing as the variable **f** into main(f)

2. main(file)

This is where the main calls to helpers and objects are created to make parsing easier.

    a. The execution order is as follows.

        i. openJSON()
        ii. checkExtension(fileName, syntaxes)
        iii. ExtensionType(extension,syntaxes)
        iv. ExtractInfo(file,languageSyntax)
        v. openJSON()

3. openJSON()

This loads up the syntaxes.json file and returns it in a variable called **syntaxes.**

4. checkExtension(fileName,data)

Using the name **fileName**, this will check whether or not this programming file's type exists in **syntaxes.** If it does it proceeds to return only the extension which will be return and be stored in a variable called **extension**.

5. ExtensionType(extension, syntaxes)

Using the **extension** variable and **syntaxes**, it is passed onto a class that will extract the specific comment syntax details related to a specific **extension** from the **syntaxes** parameter. Further detail about the class is discussed later in this document. The resulting specificities are stored in a variable called **languageSyntax.**

6. ExtractInfo(file,languageSyntax)

Using the **file** and the **languageSyntax** (extensionType object) this function extracts all the necessary information from the file.

extensiontype.py:

Summary

ExtensionType is a class that is designed to retrieve a specific programming language's particular comment syntax, which can be used later when conducting the analysis of the file.

The object takes in an **extension** and the **syntaxes** and generates three attributes. These are listed below.

- SingleComments: The syntax used to define a single comment in a given extension.
- BlockComments: The syntax used to define a block comment for a given extension, this will be an array as block comments are commonly defined by an opening and closing mark.
- TODO: The syntax used to define a todo comment for a given extension.

Functions:

1. getSingleCommentSyntax(extension,syntaxes)

The **extension** acts as a key used to retrieve the value of a single comment syntax from **syntaxes**. A string representing the syntax is returned and stored in the SingleComments attribute of the object.

2. getBlockCommentSyntax(extension,syntaxes)

The **extension** acts as a key used to retrieve the value of a block comment syntax from the **syntaxes**. Because comments have an open and close syntax, an array is returned where the first element is the open syntax and the second element is the closed syntax. This is stored in the BlockComments attribute of the object.

3. getTodoSyntax(extension,syntaxes)

The **extension** acts as a key used to retrieve the value of a TODO comment syntax from the **syntaxes**. A string representing the syntax is returned and stored in the todo attribute of the object.

extractinfo.py (Version 1):

Summary

extractinfo is a class that is designed to retrieve all the necessary information from the file, this is where the actual parsing occurs.

The object takes in the **file** and extensiontype object called **languageSyntax** and generates six attributes.

- totalLines: The total number of lines that the file contains.
- commentLines: The total number of lines that are considered comments, this is essentially singleLineComments + commentLinesWithinBlocks.
- singleLineComments: The total number of lines that contain a single comment, this includes todo comments.

- commentLineWithinBlocks: The total number of lines within every block Comment.
- blockLineComments: The total number of block comments.
- Todo: The total number of lines that contain a todo comment.

1. Parse(file,languageSyntax):

The parse function is what goes through the **file** and extracts the necessary information to fill the attributes of the object. It goes through the file character by character and checks if the current character till the length of a given syntax (this will be explained further in the checkifValidComments() function below) meets a particular **languageSyntax** attribute and then increments the respective **extractinfo** attribute. Additionally, it also keeps track of whether or not we've hit syntax such as quotation marks or parenthesis, as anything inside those are not considered part of any comments as mentioned in the assumptions. The function makes use of flags to help keep track of when we see a certain **languageSyntax** attribute or symbols in order to help skip evaluations at following iterations until we reach a character that indicates the end of that **languageSyntax** or symbol.

The function checks for a variety of cases, each case is shown explained below.

- New Line Case: The first if statement checks if the character is "\n". This signifies a new line, which can let us help is evaluate the end of a single comment/todo comment and a line within a block comment.
- All flags = 0 Case: The second if statement helps us know this current iteration character is a part or within a comment or symbol. If successful we check for several things.
    - Quotations or parenthesis case: Checks if current character is an open quotation or parenthesis and if so ensures that the next iterations are ignored until we reach the corresponding closing quotation or parenthesis.
    - languageSyntax.todo case: Checks if the current character is a todo comment and if so ensures that the next evaluations are ignored until we reach the new line, which is done by changing the singleFlag to 1.
    - languageSyntax.singleComment case: Checks if the current character is a single comment and if so ensures that the next evaluations are ignored until we reach the new line, which is done by changing the singleFlag to 1.
    - languageSyntax.blockComment case: Checks if the current character is the opening part of a block comment and if so ensures that the next evaluations are ignored until we reach the closing part of the block comment, which is done by changing the blockFlag to 1.
- openFlag = 1 Case: This checks if we are currently within a quotations or parenthesis and if so checks if the current character is the corresponding closing symbol.
- blockFlag = 1 Case: This checks if we are currently within a block comment and if so, check if so checks if the current character is the corresponding closing block comment.

2. checkIfValidComment(commentSyntax,str,i):

The checkIfValidComment is called to check if we have a given **commentSyntax** in the current character of the string **str. i** is the index of the current character. We build a substring from the current character till the length of the **commentSyntax** from str. We then compare this newly built substring to **commentSyntax** to determine if it is the given **commentSyntax** and if so return a result that would determine what course of action the parse() function should do.

3. getTotalLines(file):

Simply just counts the total number of lines in the file.

4. getString(file):

Converts a given file to a string to allow the parse() function to iterate character by character.

5. getSymbols():

Returns data of the symbols that are the special cases, where anything in between these symbols in the string, will not be considered comments.

6. print():

This prints all the object attributes to the terminal, will be called in the main file after the object is constructed.

Extractinfo_2.py (Version 2):

Summary

extractinfo is a class that is designed to retrieve all the necessary information from the file, this is where the actual parsing occurs.

The object takes in the **file** and extensiontype object called **languageSyntax** and generates six attributes.

- totalLines: The total number of lines that the file contains.
- commentLines: The total number of lines that are considered comments, this is essentially singleLineComments + commentLinesWithinBlocks.
- singleLineComments: The total number of lines that contain a single comment, this includes todo comments.
- commentLineWithinBlocks: The total number of lines within every block Comment.
- blockLineComments: The total number of block comments.
- Todo: The total number of lines that contain a todo comment.

1. Parse(file,languageSyntax):

The parse function is what goes through the **file** and extracts the necessary information to fill the attributes of the object. It's approach towards parsing is different compared to version 1, as instead of parsing character by character it goes line by line. It works by checking whether a given line has a particular **languageSyntax** attribute, by using respective helper functions that each evaluate for a different **languageSyntax** attribute. The function only makes use of one flag to help keep track if the proceeding iterations are part of a block comment or not.

The function checks for a variety of cases, each case is shown explained below.

- languageSyntax.todo case: Checks whether the given line contains a todo comment.
- languageSyntax.singleComment case: Checks whether the given line contains a single line comment.
- languageSyntax.blockComment case: Checks whether the given line contains the opening part of a block comment and that the next evaluations are ignored until we reach the closing part of the block comment.

2. checkTodo(line,todoSyntax):

This helper function is called in the parse function to check if **todoSyntax** exists in the line. It first checks if the **todoSyntax** exists in the beginning of the sentence or if it exists in the line, checks if it is within a string or not.

3. checkSingleComment(line,singleCommentSyntax):

This helper function is called in the parse function to check if **singleCommentSyntax** exists in the **line**. It first checks if the **singleCommentSyntax** exists in the beginning of the current **line** or if it exists in the current **line**, checks if it is within a string or not.

4. checkBlockComment(line,blockCommentSyntax,blockFlag):

This helper function is called in the parse function to check if **blockCommentSytnax** exists in the line. If the **blockFlag** is 0, we check if an open syntax of a block comment exists in the **line**, else if it is 1, we check if the close syntax of a block comment exists in order to reset the flag and count the lines within a block.

5. checkIfString(line,commentSyntax):

This helper function is called in the checkFunctions above in order to check if a **commentSyntax** is within a string or not if it was previously identified in the line.

6. print()

This prints all the object attributes to the terminal, will be called in the main file after the object is constructed.