Plag Checker: Explained

Generated by Doxygen 1.9.0

Chapter 1

Namespace Index

1.1 Packages

Here are the packages with brief descriptions (if available):

rest.checker_core.backup_checker . . . . . ??
rest.checker_core.checker_cpp . . . . . ??
rest.checker_core.checker_java . . . . . ??
rest.checker_core.checker_py . . . . . ??
rest.checker_core.Final_Checker . . . . . ??

Chapter 2

Namespace Documentation

2.1 rest.checker_core.backup_checker Namespace Reference

Functions

- def backup_tokenize (filename)

2.1.1 Detailed Description

```
Python module pygments is used to tokenize the code files.
This module supports most of the popular languages
http://pygments.org/languages/
Hence this program can be used to clean up codes written in most languages
This program generates tokenized version of source code files
using pygments to identify the token type
This is a general checker with basic functionality for tokenization
It will be invoked in case files are of any type other than C++/Pyhton/JAVA
or the primary tokenizer for thses langugaes encounters an error
```

2.1.2 Function Documentation

2.1.2.1 backup_tokenize()

```
def rest.checker_core.backup_checker.backup_tokenize (
              filename )
```

```
This function takes filename as input and generates tokens based on the following rules -
1) 'funct' keyword is used for functions - Functions calls will be represented by this token.
2) 'class' keyword is used for classes - Instances of classes/objects will be
    replaced by this token.
3) 'v' is the token used for variable declarations
4) Keywords, operators, indentifiers, builtin methods, attributes and decorators are added
    as it is in string form
5) Whitespaces, comments, punctuation and literals are ignored
```

Definition at line 17 of file backup_checker.py.

```
17 def backup_tokenize(filename):
18
19     """
20     This function takes filename as input and generates tokens based on the following rules -
21     1) 'funct' keyword is used for functions - Functions calls will be represented by this token.
22     2) 'class' keyword is used for classes - Instances of classes/objects will be
23         replaced by this token.
24     3) 'v' is the token used for variable declarations
25     4) Keywords, operators, indentifiers, builtin methods, attributes and decorators are added
26         as it is in string form
27     5) Whitespaces, comments, punctuation and literals are ignored
28     """
29     file = open(filename, "r")
30
31     if os.path.exists("work"):
32         os.remove("work")
33
34     work = open('work', 'a')        #an auxillary file created to store the cource code tet with extra
       whitespace removed#
35
36     for l in file:
37         if l == '' or l.isspace():      #ignore whitespace#
38             pass
39         else:
40             work.write(l.rstrip())
41             work.write('\n')
42
43     file.close()
44     work.close()
45
46     file = open('work', 'r')    #read all text from auxillary file#
47     text = file.read()
48
49     lexer = pygments.lexers.guess_lexer_for_filename(filename, text)        #obtain lexer from pygmnets #
50     tokens = lexer.get_tokens(text)
51     tokens = list(tokens)
52     result = []
53     lenT = len(tokens)
54     file_tokens = []
55     class_list = []
56
57
58
59     for i in range(lenT):
60
61         if tokens[i][0] == pygments.token.Name.Function:
62             file_tokens.append('funct')
63
64         elif tokens[i][0] == pygments.token.Name.Class or str(tokens[i][1]) in class_list:      #identify
       instances of classes and update class_list#
65             class_list.append(str(tokens[i][1]))                                                #identify
       objects/ instances of user defined classes and assign 'class' token to it#
66             file_tokens.append('class')
67
68         elif (tokens[i][0] == pygments.token.Name or tokens[i][0] in pygments.token.Name) and not i ==
       lenT - 1 and not tokens[i + 1][1] == '(':
69
70             if str(tokens[i][1]) in class_list:                                                 #identify
       objects/ instances of user defined classes and assign 'class' token to it#
71                 file_tokens.append('class')
72
73             elif tokens[i][0] in pygments.token.Name.Namespace:                                 #identify
       namespaces and add them#
74                 file_tokens.extend(str(tokens[i][0]).split())
75
76             elif tokens[i][0] in pygments.token.Name.Builtin or tokens[i][0] in
       pygments.token.Name.Attribute or tokens[i][0] in pygments.token.Name.Decorator :
77                 file_tokens.append(str(tokens[i][1]))                       #identify builtin methods,
       decorators and attributes and add the token as string form to the list of file tokens#
78
79             else:
80                 file_tokens.append('v')     #if the token does not satisfy any of the condition above, it
       is a variable name. So 'v' token is assigned to it#
81
82         elif tokens[i][0] in pygments.token.Literal.String:
83             pass
84
85         elif tokens[i][0] == pygments.token.Text or tokens[i][0] in pygments.token.Comment or
       tokens[i][0] in pygments.token.Punctuation:
86             pass   #whitespaces and comments ignored
87
88         else:
89             file_tokens.append(str(tokens[i][1]))           #remaining tokens are identifiers, operators
       and keywords, so are appended to file_tokens#
90
91     if os.path.exists("work"):
```

```
92          os.remove("work")
93
94      return ''.join(file_tokens)
```

2.2 rest.checker_core.checker_cpp Namespace Reference

Functions

- def keywords ()
- def identifiers ()
- def operators ()
- def delimiters ()
- def add_func (token, func_tokens)
- def basicCheck (token, file_tokens, func_tokens, class_list)
- def funcCheck (token, func_tokens, func_list, class_list)
- def delimiterCorrection (line)
- def isWhiteSpace (word)
- def hasWhiteSpace (token)
- def class_n_func_tokens (class_all_list, func_all_list)
- def tokenize (path, file_tokens, func_tokens, class_all_list, func_all_list)
- def run (path)
- def tokenize_cpp (file)

Variables

- int **scope_depth** = 0
- int **is_function** = -1
- **generator_path**
- **generator_name**
- **xml_generator_config**
- **declarations**
- **global_namespace**
- **std**

2.2.1 Detailed Description

```
Python library pygccxml is used.
This package provides functionality to extract and inspect declarations from
C/C++ header files.
This is accomplished by invoking an external tool like CastXML or GCC-XML, which parses
a header file and dumps the declarations as a XML file.
This XML file is then read by pygccxml and the contents are made available
as appropriate Python objects.

To parse a set of C/C++ header files you use the parse function in the :mod:parser sub
package which returns a tree that contains all declarations found in the header files.
The root of the tree represents the main namespace :: and the children nodes represent
the namespace contents such as other namespaces, classes, functions, etc.
Each node in the tree is an object of a type derived from the declaration_t class.
An inner node is always either a namespace declarations.namespace_t or a class
declarations.class_t, which are both derived from declarations.scopedef_t class

Thus, we can obtain -
1) a list of function and their arguments
2) classes and their variables, constructors, operators and methods
3) global variables
```

2.2.2 Function Documentation

2.2.2.1 add_func()

```
def rest.checker_core.checker_cpp.add_func (
              token,
              func_tokens )
```

takes the dictionary of func_tokens and function name as argument and returns a
list of tokens corresponding to the function
Invoked whnever a function is called and token is name of the function

Definition at line 77 of file checker_cpp.py.

```
77 def add_func(token, func_tokens):
78     "'takes the dictionary of func_tokens and function name as argument and returns a
79     list of tokens corresponding to the function
80     Invoked whnever a function is called and token is name of the function"'
81     new_list = []
82     for t in func_tokens[token]:
83         if t != token:
84             if t in func_tokens:
85                 new_list = new_list + add_func(t, func_tokens)
86             else:
87                 new_list.append(t)
88
89     return new_list
90
91
92
```

2.2.2.2 basicCheck()

```
def rest.checker_core.checker_cpp.basicCheck (
              token,
              file_tokens,
              func_tokens,
              class_list )
```

token is single token to be processed now
file_tokens is the list to which the token might be added
func_tokens is a dictionary with function names mapped to corresponding declarations
and is used to add tokens corresponding to a function call
class_list is a list of classes to identify object instances

This fucntion examines the given token and determines whether it needs to be appended.
Whitespaces, comments, delimiters/punctuation/literals are ignored.
Tokens which are keywords/ identifiers/ operators are added as strings to the file_tokens list
Variables names are assigned token with 'v' keyword
Numbers of any type(int/ float) are assigned token with 'no' keyword
Headers of any type are assigned token with 'he' keyword
Objects/ instances of a class are assigned token woth 'obj' keyword
For function calls, add_func is passes the function name and tokens corresponding to the
function are added to file_tokens

Definition at line 93 of file checker_cpp.py.

```
93 def basicCheck(token, file_tokens, func_tokens, class_list):
94
95     '''
96     token is single token to be processed now
97     file_tokens is the list to which the token might be added
98     func_tokens is a dictionary with function names mapped to corresponding declarations
99     and is used to add tokens corresponding to a function call
100     class_list is a list of classes to identify object instances
101
102     This fucntion examines the given token and determines whether it needs to be appended.
103     Whitespaces, comments, delimiters/punctuation/literals are ignored.
104     Tokens which are keywords/ identifiers/ operators are added as strings to the file_tokens list
105     Variables names are assigned token with 'v' keyword
106     Numbers of any type(int/ float) are assigned token with 'no' keyword
107     Headers of any type are assigned token with 'he' keyword
108     Objects/ instances of a class are assigned token woth 'obj' keyword
109     For function calls, add_func is passes the function name and tokens corresponding to the
110     function are added to file_tokens
111     '''
112     global scope_depth, is_function
113     varPtrn = re.compile(r"[a-zA-Z_][a-zA-Z0-9_]")  # variables
114     headerPtrn = re.compile(r"\w[a-zA-Z]+[.]h")  # header files
115     digitPtrn = re.compile(r'\d')   #digits
116     floatPtrn = re.compile(r'\d+[.]\d+')    #decimals
117
118     if token in mysrc.delimiters():
119         description = mysrc.delimiters()[token]
120         if description == 'LCBRACE':
121             scope_depth += 1
122
123         elif description == 'RCBRACE':
124             scope_depth -= 1
125             if is_function != -1 and scope_depth == 0:
126                 is_function = -1
127
128         else:
129             pass
130     elif token in mysrc.keywords():
131
132         if is_function != -1:
133             pass
134         else:
135             file_tokens.append(token)
136
137     elif token in mysrc.identifiers():
138
139         if is_function != -1:
140             pass
141         else:
142             file_tokens.append(token)
143
144     elif token in mysrc.operators().keys():
145
146         if is_function != -1:
147             pass
148         else:
149             file_tokens.append(token)
150
151     elif re.search(headerPtrn, token):
152
153         file_tokens.append('head')
154
155     elif token in func_tokens.keys() and token != 'main':
156         file_tokens.extend(add_func(token, func_tokens))
157
158
159     elif token in class_list:
160         file_tokens.append('obj' )
161
162     elif token == 'head':
163         file_tokens.append('he')
164
165     elif token == 'num':
166         file_tokens.append('no')
167
168     elif token == 'obj':
169         file_tokens.append('obj')
170
171     elif re.match(varPtrn, token) or "'" in token or '"' in token:
172         if is_function != -1:
173             pass
174         else:
175             file_tokens.append('v')
176     elif re.match(digitPtrn, token):
177
178         if is_function != -1:
```

```
179             pass
180         else:
181             file_tokens.append('no')
182 
183     return True
184 
```

2.2.2.3 class_n_func_tokens()

```
def rest.checker_core.checker_cpp.class_n_func_tokens (
              class_all_list,
              func_all_list )
```

```
Takes as input list of all classes and functions in the file as identified by
the pygccxml parser
Generates new lists - func_list and class_list of user defined functions and classes
For functions - the line number where function definition begins is identified
using regex matching and stored in func_start with the same order as func_list
For Classses - constructors, operators, variables and memeber functions corresponding
to each user defined class are obtained from the parser and added to dictionary
class_tokens mapped to class name
Returns the above generated lists/dictionaries
```

Definition at line 333 of file checker_cpp.py.

```
333 def class_n_func_tokens(class_all_list, func_all_list):
334 
335     '''
336     Takes as input list of all classes and functions in the file as identified by
337     the pygccxml parser
338     Generates new lists - func_list and class_list of user defined functions and classes
339     For functions - the line number where function definition begins is identified
340     using regex matching and stored in func_start with the same order as func_list
341     For Classses - constructors, operators, variables and memeber functions corresponding
342     to each user defined class are obtained from the parser and added to dictionary
343     class_tokens mapped to class name
344     Returns the above generated lists/dictionaries
345     '''
346 
347     func_start = []
348     func_list = []
349     func_tokens = []
350     class_tokens = {}
351     f = open('work', 'r')
352     txt = f.read()
353     for func in func_all_list:
354         pat = r"\s*"+str(func.name)+r"+\s*\("
355         res = re.findall(pat, txt)
356         if (len(res)>0):
357             func_list.append(str(func.name))
358             pat2 = r"\s*"+str(func.name)+r"+\s*\(([\w+\s+\w+])*\)\s*\{"
359             pos = re.search(pat2, txt)
360             if pos != None:
361                 line_no = len(re.findall('\n', txt[0:int(pos.start())]))
362                 func_start.append(line_no)
363 
364     class_list = []
365     for class_ in reversed(class_all_list):
366         pat = r"\s*"+str(class_.name)+r"+\s*\{"
367         res = re.findall(pat, txt)
368         if (len(res)>0):
369             class_tokens[class_.name] = []
370             for base in class_.bases:
371                 class_tokens[class_.name].extend(base.related_class.name.split())
372 
373             for derive in class_.derived:
374                 class_tokens[class_.name].extend(derive.related_class.name.spilt())
375 
376             for p in class_.constructors(allow_empty = True):
377                 if p is None:
378                     break
379                 for a in p.argument_types:
380                     class_tokens[class_.name].extend(str(a).split())
381 
```

```
382             for p in class_.operators(allow_empty = True):
383                 if p is not None:
384                     p = re.sub(r'operator', r'ope', str(p.name))
385                     class_tokens[class_.name].append(p)
386 
387             for p in class_.variables(allow_empty = True):
388                 class_tokens[class_.name].append(str(p.decl_type))
389 
390             for p in class_.member_functions(allow_empty = True):
391                 if p is None:
392                     break
393                 for a in p.argument_types:
394                     class_tokens[class_.name].append(str(a))
395 
396             class_list.append(str(class_.name))
397 
398     return func_list, func_start, class_list, class_tokens
399 
```

2.2.2.4 delimiterCorrection()

```
def rest.checker_core.checker_cpp.delimiterCorrection (
              line )
```

```
Takes a line as input and splits it into tokens using whitespace as separator
To ensure that delimiters are taken into account poistion of delimeters are identified
and replaced with padding of spaces around them for effective splitting
Returned is list of tokens generated from the line excluding whitespaces
The tokens generated now are just words in the source code file, they need to be
processed further
```

Definition at line 263 of file checker_cpp.py.

```
263 def delimiterCorrection(line):
264 
265     '''Takes a line as input and splits it into tokens using whitespace as separator
266     To ensure that delimiters are taken into account poistion of delimeters are identified
267     and replaced with padding of spaces around them for effective splitting
268     Returned is list of tokens generated from the line excluding whitespaces
269     The tokens generated now are just words in the source code file, they need to be
270     processed further'''
271 
272     for delim in mysrc.delimiters().keys():
273         if delim in line:
274             line = line.replace(delim, ' '+delim+' ')
275 
276     tokens = line.split(" ")
277     for delimiter in mysrc.delimiters().keys():
278         for token in tokens:
279 
280             if token == delimiter:
281                 pass
282             elif delimiter in token:
283 
284                 pos = token.find(delimiter)
285                 tokens.remove(token)
286                 token = token.replace(delimiter, " ")
287                 extra = token[:pos]
288                 token = token[pos + 1 :]
289                 tokens.append(delimiter)
290                 tokens.append(extra)
291                 tokens.append(token)
292             else:
293                 pass
294     for token in tokens:
295         if isWhiteSpace(token):
296             tokens.remove(token)
297         elif ' ' in token:
298             tokens.remove(token)
299             token = token.split(' ')
300             for d in token:
301                 tokens.append(d)
302     return tokens
303 
```

2.2.2.5 delimiters()

```
def rest.checker_core.checker_cpp.delimiters ( )
```

a dictionary of cpp delimiters

Definition at line 69 of file checker_cpp.py.

```
69 def delimiters():
70     "'a dictionary of cpp delimiters"'
71     delimiters = {
72     "\t": "TAB", "\n": "NEWLINE","(": "LPAR", ")": "RPAR", "[": "LBRACE", "]": "RBRACE", "{": "LCBRACE",
       "}": "RCBRACE", "=": "ASSIGN",":": "COLON", ",": "COMMA", ";": "SEMICOL", "«": "OUT", "»": "IN",
73     }
74     #print(len(delimiters)) = 14
75     return delimiters
76
```

2.2.2.6 funcCheck()

```
def rest.checker_core.checker_cpp.funcCheck (
              token,
              func_tokens,
              func_list,
              class_list )
```

token is single token to be processed now
file_tokens is the list to which the token might be added
func_tokens is a dictionary with function names mapped to corresponding declarations and
is used to add tokens corresponding to a function call
class_list is a list of classes to identify object instances

Similar to basicChecker but works on processing tokens of a particular function.
The name of current function being processed is stored in is_function var.
It appends the generated token value to the dictionary func_tokens mapped to corresponding name

Definition at line 185 of file checker_cpp.py.

```
185 def funcCheck(token, func_tokens, func_list, class_list):
186
187     "'
188     token is single token to be processed now
189     file_tokens is the list to which the token might be added
190     func_tokens is a dictionary with function names mapped to corresponding declarations and
191     is used to add tokens corresponding to a function call
192     class_list is a list of classes to identify object instances
193
194     Similar to basicChecker but works on processing tokens of a particular function.
195     The name of current function being processed is stored in is_function var.
196     It appends the generated token value to the dictionary func_tokens mapped to corresponding name
197     "'
198
199     global scope_depth, is_function
200     varPtrn = re.compile(r"[a-zA-Z_][a-zA-Z0-9_]")  # variables
201     headerPtrn = re.compile(r"\w[a-zA-Z]+[.]h")  # header files
202     digitPtrn = re.compile(r'\d') #digits
203     floatPtrn = re.compile(r'\d+[.]\d+') #decimals
204
205     if token in mysrc.delimiters():
206         description = mysrc.delimiters()[token]
207         if description == 'LCBRACE':
208             scope_depth += 1
209
210         elif description == 'RCBRACE':
211             scope_depth -= 1
212             if is_function != -1 and scope_depth == 0:
213                 is_function = -1
214
215         else:
```

```
216             pass
217     elif token in mysrc.keywords():
218 
219         if is_function != -1:
220             func_tokens[is_function].append(token)
221         else:
222             pass
223     elif token in mysrc.identifiers():
224 
225         if is_function != -1:
226             func_tokens[is_function].append(token)
227         else:
228             pass
229 
230     elif token in mysrc.operators().keys():
231 
232         if is_function != -1:
233             func_tokens[is_function].append(token)
234         else:
235             pass
236     elif token in func_list and token != is_function and is_function!= -1:
237         func_tokens[is_function].append(token)
238 
239     elif token in class_list and is_function!= -1:
240         func_tokens[is_function].append('obj' )
241 
242     elif re.search(headerPtrn, token):
243 
244         pass
245     elif re.match(varPtrn, token) or "'" in token or '"' in token:
246 
247         if is_function != -1:
248             func_tokens[is_function].append('v')
249         else:
250             pass
251 
252 
253     elif re.match(digitPtrn, token):
254 
255         if is_function != -1:
256             func_tokens[is_function].append('no')
257             pass
258         else:
259             pass
260 
261     return True
262 
```

2.2.2.7 hasWhiteSpace()

```
def rest.checker_core.checker_cpp.hasWhiteSpace (
            token )
```

```
Checks if a token has a whitespace in it
If it is present, it is interpreted as aliteral and returned with single quotes added
Else return false
```

Definition at line 315 of file checker_cpp.py.

```
315 def hasWhiteSpace(token):
316 
317     '''
318     Checks if a token has a whitespace in it
319     If it is present, it is interpreted as aliteral and returned with single quotes added
320     Else return false
321     '''
322     ptrn = ['\t', '\n']
323     if isWhiteSpace(token) == False:
324         for item in ptrn:
325             if item in token:
326                 result = "'" + item + "'"
327                 return result
328             else:
329                 pass
330     return False
331 
332 
```

2.2.2.8 identifiers()

```
def rest.checker_core.checker_cpp.identifiers ( )
```

a list of cpp identifiers

Definition at line 55 of file checker_cpp.py.

```
55 def identifiers():
56     "'a list of cpp identifiers"'
57     identifiers = [
58     "auto", "bool", "char", "double", "enum", "float", "int", "long", "short", "string" ]
59     #print(len(identifiers)) = 10
60     return identifiers
61
```

2.2.2.9 isWhiteSpace()

```
def rest.checker_core.checker_cpp.isWhiteSpace (
              word )
```

takes token as input and return true if it comes under whitespace else false

Definition at line 304 of file checker_cpp.py.

```
304 def isWhiteSpace(word):
305     "'
306     takes token as input and return true if it comes under whitespace else false
307     "'
308     ptrn = [ " ", "\t", "\n"]
309     for item in ptrn:
310         if word == item:
311             return True
312         else:
313             return False
314
```

2.2.2.10 keywords()

```
def rest.checker_core.checker_cpp.keywords ( )
```

a list of cpp keywords

Definition at line 46 of file checker_cpp.py.

```
46 def keywords():
47     "'a list of cpp keywords"'
48     keywords = [
49     "break", "case", "catch", "word", "class", "const", "continue", "delete", "do", "else", "false",
       "for", "goto", "if",
50     "namespace", "not", "or", "private", "protected", "public", "return", "signed", "sizeof", "static",
       "struct", "switch", "true", "try", "unsigned", "void", "while",
51     "endl", "cout", "cin", 'queue', 'stack', 'vector', 'array', 'list', 'forward_list', 'set' ,'map',
       'deque', 'priority_queue', 'multiset', 'multimap',
52     ]
53     return keywords
54
```

2.2.2.11 operators()

```
def rest.checker_core.checker_cpp.operators ( )
```

a dictionary of cpp operators

Definition at line 62 of file checker_cpp.py.

```
62 def operators():
63     '''a dictionary of cpp operators'''
64     operators = {
65     "+": "PLUS", "-": "MINUS", "*": "MUL", "/": "DIV", "%": "MOD", "+=": "PLUSEQ", "-=": "MINUSEQ", "*=":
       "MULEQ", "/=": "DIVEQ", "++": "INC", "--": "DEC", "|": "OR", "&&": "AND", "&": "REF",
66     }
67     return operators
68
```

2.2.2.12 run()

```
def rest.checker_core.checker_cpp.run (
              path )
```

Takes the path of file name as input
The parser and xml generator use the file to generate a list of declarations
The global namespace is obtained and the list of declarations in the global namespace
is examined for functions and classes which are identified by the parser
This list slong woth the file path id passed to the tokenize function to produce
tokens for the source code

Definition at line 474 of file checker_cpp.py.

```
474 def run(path):
475
476     '''
477     Takes the path of file name as input
478     The parser and xml generator use the file to generate a list of declarations
479     The global namespace is obtained and the list of declarations in the global namespace
480     is examined for functions and classes which are identified by the parser
481     This list slong woth the file path id passed to the tokenize function to produce
482     tokens for the source code
483     '''
484
485     declarations = parser.parse([path], xml_generator_config)
486     global_namespace = declarations.get_global_namespace(declarations)
487     std = global_namespace.namespace("std")
488
489     func_all_list = []
490     class_all_list = []
491
492     for d in global_namespace.declarations:
493         if isinstance(d, declarations.class_declaration_t):
494             pass
495
496         if isinstance(d, declarations.class_t) and d.parent == global_namespace:
497             class_all_list.append(d)
498
499         if isinstance(d, declarations.free_function_t):
500             func_all_list.append(d)
501
502     file_tokens = []
503     func_tokens = {}
504
505     '''
506     file_tokens is the list which will store all the tokens generated from the file
507     func_tokens is a dictionary with function names mapped to corresponding declarations
508     and is used to add tokens corresponding to a function call
509     '''
510     tokenize(path, file_tokens, func_tokens, class_all_list, func_all_list)
511     return file_tokens, func_tokens
512
513
```

2.2.2.13 tokenize()

```
def rest.checker_core.checker_cpp.tokenize (
              path,
              file_tokens,
              func_tokens,
              class_all_list,
              func_all_list )
```

```
path is the path of file to be processed
file_tokens is the list which will store all the tokens generated from the file
func_tokens is a dictionary with function names mapped to corresponding declarations
and is used to add tokens corresponding to a function call
class_all_list and func_all_list are lists of all classes and functions in the file
as identified by the pygccxml parser

This function first invokes class_n_func_tokens to generate information about functions
and tokens corresponding to classes
It then invokes funcCheck to generate tokens corresponding to a fucntion and
store in Func_tokens
Subsequently, basicCheck is called to tokenize the entire file and store tokens in file_tokens
```

Definition at line 400 of file checker_cpp.py.

```
400 def tokenize(path, file_tokens, func_tokens, class_all_list, func_all_list):
401
402     '''
403     path is the path of file to be processed
404     file_tokens is the list which will store all the tokens generated from the file
405     func_tokens is a dictionary with function names mapped to corresponding declarations
406     and is used to add tokens corresponding to a function call
407     class_all_list and func_all_list are lists of all classes and functions in the file
408     as identified by the pygccxml parser
409
410     This function first invokes class_n_func_tokens to generate information about functions
411     and tokens corresponding to classes
412     It then invokes funcCheck to generate tokens corresponding to a fucntion and
413     store in Func_tokens
414     Subsequently, basicCheck is called to tokenize the entire file and store tokens in file_tokens
415     '''
416
417     global is_function
418     var_list = []
419     try:
420         file = open(path)
421         f = file.read()
422
423
424         lines = f.split("\n")
425         file.close()
426
427
428         # check if file exists
429         if os.path.exists("work"):
430             os.remove("work")
431         file = open('work', 'a')
432
433         for line in lines:
434             line = line.strip()
435             if line is not None and line is not '':
436                 file.write(line)
437                 file.write('\n')
438         file.close()
439
440         func_list, func_start, class_list, class_tokens = class_n_func_tokens(class_all_list,
      func_all_list)
441
442         count = -1
443         for line in lines:
444             line = line.strip()
445             if line is not None and line is not '':
446                 count +=1
447                 if count in func_start:
448
449                     is_function = func_list[func_start.index(count)]
450
451                     func_tokens[is_function] = []
452                 tokens = delimiterCorrection(line)
```

```
453
454                 for token in tokens:
455                     funcCheck(token, func_tokens, func_list, class_list)
456
457         for token in func_tokens['main']:
458             basicCheck(token, file_tokens, func_tokens, class_list)
459
460         for c in class_tokens.keys():
461             for token in class_tokens[str(c)]:
462                 token = str(token)
463
464                 if (token[0:3] == 'ope'):
465                     file_tokens.append(token)
466                 else:
467                     basicCheck(token, file_tokens, func_tokens, class_list)
468
469         return True
470     except FileNotFoundError:
471         print("\nInvald Path. Retry")
472         run()
473
```

2.2.2.14 tokenize_cpp()

```
def rest.checker_core.checker_cpp.tokenize_cpp (
              file )
```

```
Takes file as argument and invokes run function to obtain the list of tokens
generated from the file
It returns a single string of all tokens joined together
```

Definition at line 514 of file checker_cpp.py.

```
514 def tokenize_cpp(file):
515     '''
516     Takes file as argument and invokes run function to obtain the list of tokens
517     generated from the file
518     It returns a single string of all tokens joined together
519     '''
520     t1a, t1f = run(file)
521     return ''.join(t1a)
```

2.2.3 Variable Documentation

2.2.3.1 xml_generator_config

```
rest.checker_core.checker_cpp.xml_generator_config
```

Initial value:

```
1 =  parser.xml_generator_configuration_t(
2     xml_generator_path=generator_path,
3     xml_generator=generator_name)
```

Definition at line 39 of file checker_cpp.py.

2.3 rest.checker_core.checker_java Namespace Reference

Functions

- def tokenize_jav (filename)

2.3.1 Detailed Description

```
Python module pygments is used to tokenize the code files. This module
supports most of the popular languages
http://pygments.org/languages/
Hence this program can be used to clean up source code
This program generates tokenized version of java source code files using
pygments to identify the token type
```

2.3.2 Function Documentation

2.3.2.1 tokenize_jav()

```
def rest.checker_core.checker_java.tokenize_jav (
              filename )
```

```
This function takes filename as input and returns the tokenized version of
source code as string.
It first removes all extra whitespaces. Then it identifies classes and functions
and prepares a list of them
Subsequently the remaining files is tokenized and list of tokens stored in file_tokens
Whenever a fucntion call/ class instance/ variable name is encountered, specific
keywords are used as tokens ('function'/ 'class'/ 'var')
Comments, punctutation, literals are ignored
```

Definition at line 12 of file checker_java.py.

```
12 def tokenize_jav(filename):
13
14     """
15     This function takes filename as input and returns the tokenized version of
16     source code as string.
17     It first removes all extra whitespaces. Then it identifies classes and functions
18     and prepares a list of them
19     Subsequently the remaining files is tokenized and list of tokens stored in file_tokens
20     Whenever a fucntion call/ class instance/ variable name is encountered, specific
21     keywords are used as tokens ('function'/ 'class'/ 'var')
22     Comments, punctutation, literals are ignored
23     """
24
25     file = open(filename, "r")
26     if os.path.exists("work"):
27         os.remove("work")
28
29     work = open('work', 'a')        #an auxillary file created to store the cource code tet with extra
       whitespace removed#
30     in_func = -1
31
32     for l in file:
33         if l == '' or l.isspace():      #ignore whitespace#
34             pass
35         else:
36             work.write(l.rstrip())      #remove trailing space and write to auxillary file#
37             work.write('\n')
38
39     file.close()
40     work.close()
41
42     file = open('work', 'r')
43     text = file.read()      #read all text from auxillary file#
44
45     lexer = pygments.lexers.guess_lexer_for_filename(filename, text)
46     tokens = lexer.get_tokens(text)
47     tokens = list(tokens)
48     func_list = []      #list to store all the function names#
49     lenT = len(tokens)
50     file_tokens = []        #list to store the tokens corresponding to the entire source code file#
51     class_list = []     #list to store all the user defined classes#
52
```

```
53
54      #key_names stores dictionary of keywords which are not identified by pygments. They are assigned a
        shorter value as code to keep track of their weightage in tokenized string#
55      key_names = {'String' : 'str', 'ArrayList' : 'array', 'List': 'list', 'LinkedList': 'linked',
        'HashMap': 'hashma', 'HashSet':' hashse', 'BufferedReader': 'buffer',
56      'ArithmeticException' : 'arithmex', 'ArrayIndexOutOfBoundsException' : 'arrinoex', 'Iterator':
        'iterat', 'Pattern': 'pater', 'Matcher': 'match', 'PatternSyntaxException': 'patsynex',
57      'ClassNotFoundException': 'clasnoex', 'FileNotFoundException': 'filenoex', 'IOException': 'inpoutex',
        'InterruptedException': 'intexex', 'NoSuchFieldException': 'nofileex',
58      'NoSuchMethodException': 'nomethex', 'NullPointerException': 'nulponex', 'NumberFormatException':
        'numforex', 'RuntimeException': 'runtimex',
59      'StringIndexOutOfBoundsException': 'strioex', 'LocalDate': 'locdat', 'LocalTime': 'loctim',
        'LocalDateTime': 'dattim', 'DateTimeFormatter': 'dtform',
60      'Thread': 'thread', 'Main': 'main', 'Runnable': 'runble', 'Consumer': 'consum', 'private': 'scp',
        'public': 'scp', 'protected': 'scp',
61      'FileReader': 'filred', 'FileInputStream': 'fileinpstr', 'FileWriter': 'filewrit', 'BufferedWriter':
        'bufwrit', 'FileOutputStream': 'filoutstr',
62      'abstract': 'abstract', 'implements': 'implement', 'enum': 'enum', 'interface': 'interface', 'final':
        'final', 'extends': 'extends', 'forEach': 'forEa'}
63
64      #list of java's inbuilt methods for files#
65      file_methods = ['File', 'canRead', 'canWrite', 'createNewFile', 'delete', 'exists', 'getName',
        'length', 'list', 'mkdir', 'getAbsolutePath', 'FileWriter', 'write', 'close']
66
67
68      #list of java's inbuilt methods for strings#
69      string_methods = ['charAt', 'codePointAt', 'codePointBefore', 'codePointCount', 'compareTo',
        'compareToIgnoreCase', 'concat', 'contains', 'contentEquals', 'copyValueOf', 'endsWith', 'equals',
70          'equalsIgnoreCase', 'format', 'getBytes', 'getChars', 'hashCode', 'indexOf', 'intern', 'isEmpty',
        'lastIndexOf', 'length', 'matches', 'offsetByCodePoints', 'regionMatches', 'replace', 'replaceFirst',
71          'substring', 'toCharArray', 'toLowerCase', 'toString', 'toUpperCase', 'trim', 'valueOf']
72
73
74      #list of java's inbuilt methods for mathematical operations#
75      math_methods = ['abs', 'acos', 'asin', 'atan', 'atan2', 'cbrt', 'ceil', 'copySign', 'cos', 'cosh',
        'exp', 'expm1', 'floor', 'getExponent', 'hypot', 'log', 'log10', 'log1p', 'max',
76      'min', 'nextAfter', 'nextUp', 'pow', 'random', 'round', 'rint', 'signum', 'sin', 'sinh', 'sqrt',
        'tan', 'tanh', 'toDegrees', 'toRadians', 'ulp']
77
78      for i in range(lenT):
79          if tokens[i][0] == pygments.token.Name.Function:    #identify functions and update func_list#
80              func_list.append(str(tokens[i][1]))
81
82          elif tokens[i][0] == pygments.token.Name.Class:     #identify classes and update class_list#
83              class_list.append(str(tokens[i][1]))
84
85
86      for i in range(lenT):
87          if tokens[i][0] in pygments.token.Punctuation :  #punctuautions (,.[](){} etc) are ignored#
88              pass
89
90          elif str(tokens[i][1]) in func_list or tokens[i][0] == pygments.token.Name.Function:    #assign
        keyword 'function' to function calls and declarations and add to file_tokens#
91              file_tokens.append('function')
92
93          elif tokens[i][0] in class_list or tokens[i][0] == pygments.token.Name.Class:       #assign
        keyword 'class' to class declarations and its object instances#
94              file_tokens.append('class')
95
96
97          elif (tokens[i][0] == pygments.token.Name or tokens[i][0] in pygments.token.Name) and not i ==
        lenT - 1 and not tokens[i + 1][1] == '(':    #the token is of type name#
98
99              t = str(tokens[i][1])
100
101              if tokens[i][0] == pygments.token.Name.Class or str(tokens[i][1]) in class_list:
        #identify objects/ instances of user defined classes and assign 'class' token to it#
102                  file_tokens.append('class')
103
104              elif tokens[i][0] in pygments.token.Name.Namespace:     #identify imports and obtain a short
        keyword for their type#
105                  toks = t.split('.')[-1]
106                  if toks in key_names.keys():
107                      file_tokens.append(key_names[toks])
108
109              elif tokens[i][0] in pygments.token.Name.Builtin or tokens[i][0] in
        pygments.token.Name.Decorator :
110                  file_tokens.append(t)       #identify builtin methods of java and decorators and add the
        token as string form to the list of file tokens#
111
112              elif tokens[i][0] in pygments.token.Name.Attribute:
113                  file_tokens.append('fun')
114
115              else:
116                  #check if the token is included in our defined vocabulary#
117
118                  if t in key_names.keys():
```

```
119                     file_tokens.append(key_names[t])
120
121                 elif t in file_methods:
122                     file_tokens.append(t)
123
124                 elif t in string_methods:
125                     file_tokens.append(t)
126
127                 elif t in math_methods:
128                     file_tokens.append(t)
129                 else:
130                     file_tokens.append('var')   #if the token does not satisfy any of the condition
    above, it is a variable name. So 'var' token is assigned#
131
132         elif tokens[i][0] == pygments.token.Name.Class:     #assign keyword 'class' to class
    declarations and its object instances#
133             file_tokens.append('class')
134
135         elif tokens[i][0] == pygments.token.Name or tokens[i][0] in pygments.token.Name:
136             file_tokens.append('var')
137
138         elif tokens[i][0] in pygments.token.Literal.String:
139             pass      #ignore values of string type#
140
141         elif tokens[i][0] == pygments.token.Text or tokens[i][0] in pygments.token.Comment:
142             pass   #ignore tabs, comments and other unnecessary text#
143
144         elif str(tokens[i][1]) == 'import':     #import keyword is ignored#
145             pass
146
147         else:
148             t = str(tokens[i][1])       #remaining tokens are identifiers, operators and keywords, so
    are appended to file_tokens#
149             if t in key_names.keys():
150                 file_tokens.append(key_names[t])
151             else:
152                 file_tokens.append(t)
153
154     if os.path.exists("work"):
155         os.remove("work")
156     return ''.join(file_tokens)
157
```

2.4 rest.checker_core.checker_py Namespace Reference

Functions

- def add_function_tokens (filename, name, func_text, func_tokens, class_list)
- def tokenize_py (filename)

2.4.1 Detailed Description

```
Python module pygments is used to tokenize the code files. This module
supports most of the popular languages
http://pygments.org/languages/
Hence this program can be used to clean up source code
This program generates tokenized version of python source code files using
pygments to identify the token type
```

2.4.2 Function Documentation

2.4.2.1 add_function_tokens()

```
def rest.checker_core.checker_py.add_function_tokens (
                filename,
                name,
                func_text,
                func_tokens,
                class_list )
```

```
Tokens of a function are removed and stored separately in a dictionary func_tokens.
Whenever a function call is encountered as a token this function is called and tokens corresponding
    to the function are inserted in the list of file_tokens.
*filename is used while obtaining the lexer for python from pygments module
*name is the name of the function for which tokens returned by this function
* func_text is a dictionary which maps the name of the function to the entire body of the function in
  textual form extracted from the source code file with unnecessary whitespaces and comments removed.
In case the function is encountered for the first time and its tokens have yet not been generated,
  func_text will be used to generate tokens and add to func_tokens dictionary
* func_tokens dictionary maps name of the function to tokens generated from the function
* class_list stores the list of class names and will be helpful while generating tokens to
  identify objects/ instances of classes defined by the user
```

Definition at line 12 of file checker_py.py.

```
12 def add_function_tokens(filename, name, func_text, func_tokens, class_list):
13     """
14     Tokens of a function are removed and stored separately in a dictionary func_tokens.
15     Whenever a function call is encountered as a token this function is called and tokens corresponding
16     to the function are inserted in the list of file_tokens.
17     *filename is used while obtaining the lexer for python from pygments module
18     *name is the name of the function for which tokens returned by this function
19     * func_text is a dictionary which maps the name of the function to the entire body of the function in

20       textual form extracted from the source code file with unnecessary whitespaces and comments removed.
21     In case the function is encountered for the first time and its tokens have yet not been generated,
22       func_text will be used to generate tokens and add to func_tokens dictionary
23     * func_tokens dictionary maps name of the function to tokens generated from the function
24     * class_list stores the list of class names and will be helpful while generating tokens to
25       identify objects/ instances of classes defined by the user
26     """
27     text = func_text[name] #extract text of the required function#
28     lexer = pygments.lexers.guess_lexer_for_filename(filename, text) #obtain lexer from pygmnets #
29     tokens = lexer.get_tokens(text) #generate tokens from the code#
30     tokens = list(tokens)
31     lenT = len(tokens) #length of tokens#
32     file_tokens = [] #list to store the tokens corresponding to fucntion if yet to be generated#
33
34     for i in range(lenT):
35         if tokens[i][0] == pygments.token.Name.Class and str(tokens[i][1]) not in class_list:  #identify
       instances of classes and update class_list#
36             class_list.append(str(tokens[i][1]))
37
38     for i in range(lenT):
39
40         if (tokens[i][0] == pygments.token.Name or tokens[i][0] in pygments.token.Name) and not i == lenT
       - 1 and not tokens[i + 1][1] == '(': #if the token is a name type#
41
42             if tokens[i][0] == pygments.token.Name.Class or str(tokens[i][1]) in class_list:  #identify
       objects/ instances of user defined classes and assign 'class' token to it#
43                 file_tokens.append('class')
44
45             elif tokens[i][0] in pygments.token.Name.Builtin or tokens[i][0] in
       pygments.token.Name.Function \
46                     or tokens[i][0] in pygments.token.Name.Attribute or tokens[i][0] in
       pygments.token.Name.Decorator \
47                     or tokens[i][0] in pygments.token.Name.Namespace:  #identify builtin methods of
       python, decorators and namespaces and add the token as string form#
48                 file_tokens.append(str(tokens[i][1]))
49
50             else:
51                 file_tokens.append('v') #if the token does not satisfy any of the condition above, it is
       a variable name. So 'v' token is assigned to it#
52
53
54         elif tokens[i][0] == pygments.token.Name.Class: #identify objects/ instances of builtin/other
       classes and assign 'class' token to it#
55             class_list.append(str(tokens[i][1]))
56             file_tokens.append('class')
```

```
57
58          elif tokens[i][0] in pygments.token.Literal.String: #ignore values of string type#
59              pass
60
61          elif str(tokens[i][1]) in func_tokens.keys():       #if function call is encountered, check if
        tokens have already been generated corresponding to it and add them to file_tokens#
62              file_tokens.extend(func_tokens[str(tokens[i][1])])
63
64          elif str(tokens[i][1]) in func_text.keys():     #if function call is encountered and tokens
        corresponding to it have yet not been generated #
65
66              if str(tokens[i][1]) != name:    #check if recursive call#
67                  func_tokens[str(tokens[i][1])] = add_function_tokens(filename, str(tokens[i][1]),
        func_text, func_tokens, class_list)
68                  file_tokens.extend(func_tokens[str(tokens[i][1])]) #generate tokens from text and add to
        the func_tokens dictionary#
69
70              else:
71                  pass
72          elif tokens[i][0] == pygments.token.Text or tokens[i][0] in pygments.token.Comment or
        tokens[i][0] in pygments.token.Punctuation:
73              pass   #ignore tabs, comments, punctuautions (,.[](){} etc) and other unnecessary text#
74
75          else:
76              file_tokens.append(str(tokens[i][1]))  #remaining tokens are identifiers and keywords, so are
        appended to file_tokens#
77      return file_tokens
78
79
80
```

2.4.2.2 tokenize_py()

```
def rest.checker_core.checker_py.tokenize_py (
                filename )
```

```
This function takes filename as input and returns the tokenized version
of source code as string.
It first removes all extra whitespaces.Then all functions are identified and their
text is code is removed and stored in a separate dictionary func_text
Tokens corresponding to the functions are generated and their list is mapped to the
function name in another dictionary func_tokens
Subsequently the remaining files is tokenized and list of tokens stored in file_tokens
Whenever a fucntion call is encountered, tokens corresponding to the fucntion are appended
```

Definition at line 81 of file checker_py.py.

```
81 def tokenize_py(filename):
82
83     """
84     This function takes filename as input and returns the tokenized version
85     of source code as string.
86     It first removes all extra whitespaces.Then all functions are identified and their
87     text is code is removed and stored in a separate dictionary func_text
88     Tokens corresponding to the functions are generated and their list is mapped to the
89     function name in another dictionary func_tokens
90     Subsequently the remaining files is tokenized and list of tokens stored in file_tokens
91     Whenever a fucntion call is encountered, tokens corresponding to the fucntion are appended
92     """
93
94     file = open(filename, "r")
95     if os.path.exists("work"):
96         os.remove("work")
97
98     work = open('work', 'a') #an auxillary file created to store the cource code tet with extra
       whitespace removed#
99     func_text = {}
100     pat = r'^def +(\w)*\(.*?\):' #matches with python function declaration#
101     line_no = 0
102     func_pos = []
103     in_func = -1
104
105     for l in file:
106         if l == '' or l.isspace():  #ignore whitespace#
107             pass
```

```
108         elif l[0] == '\t' and in_func != -1:
109             func_text[name] += l   #if inside function, add code to the func_text dictionary
    corresponding to the function name#
110 
111         else:
112             match = re.search(pat, l)  #check if line has function declaration#
113             in_func = -1
114 
115             if match is not None:
116                 name = match.string.split()[1]
117                 name = name.split('(')[0]
118                 func_pos.append(line_no)
119                 in_func = name
120                 func_text[name] = ''  #create a new value in dictionary func_text if new fucntion found#
121 
122             else:
123                 work.write(l.rstrip()) #remove trailing space and write to auxillary file#
124                 work.write('\n')
125         line_no += 1
126 
127     file.close()
128     work.close()
129     file = open('work', 'r')
130     text = file.read() #read all text from auxillary file#
131 
132     lexer = pygments.lexers.guess_lexer_for_filename(filename, text) #obtain lexer from pygmnets #
133     tokens = lexer.get_tokens(text)
134     tokens = list(tokens)
135     lenT = len(tokens)
136     file_tokens = []    #list to store the tokens corresponding to the entire source code file#
137     func_tokens = {}
138     class_list = []     #list to store all the user defined classes#
139 
140     for i in range(lenT):
141         if tokens[i][0] == pygments.token.Name.Class:    #identify instances of classes and update
    class_list#
142             class_list.append(str(tokens[i][1]))
143 
144     for i in range(lenT):
145         if tokens[i][0] == pygments.token.Name.Class:
146             class_list.append(str(tokens[i][1]))
147             file_tokens.append('class')
148 
149         elif (tokens[i][0] == pygments.token.Name or tokens[i][0] in pygments.token.Name) and not i ==
    lenT - 1 and not tokens[i + 1][1] == '(':  #the token is of type name#
150 
151             if tokens[i][0] == pygments.token.Name.Class or str(tokens[i][1]) in class_list:
    #identify objects/ instances of user defined classes and assign 'class' token to it#
152                 file_tokens.append('class')
153 
154             elif tokens[i][0] in pygments.token.Name.Builtin or tokens[i][0] in
    pygments.token.Name.Function \
155                     or tokens[i][0] in pygments.token.Name.Attribute or tokens[i][0] in
    pygments.token.Name.Decorator \
156                     or tokens[i][0] in pygments.token.Name.Namespace:      #identify builtin methods of
    python, decorators and namespaces and add the token as string form to the list of file tokens#
157                 file_tokens.append(str(tokens[i][1]))
158 
159             else:
160                 file_tokens.append('v')     #if the token does not satisfy any of the condition above,
    it is a variable name. So 'v' token is assigned to it#
161 
162         elif tokens[i][0] in pygments.token.Literal.String:     #ignore values of string type#
163             pass
164 
165         elif tokens[i][0] == pygments.token.Name.Class: #identify objects/ instances of builtin/other
    classes and assign 'class' token to it#
166             class_list.append(str(tokens[i][1]))
167             file_tokens.append('class')
168 
169         elif str(tokens[i][1]) in func_tokens.keys():    #if function call is encountered, check if
    tokens have already been generated corresponding to it and add them to file_tokens#
170             file_tokens.extend(func_tokens[str(tokens[i][1])])
171 
172         elif str(tokens[i][1]) in func_text.keys():     #if function call is encountered and tokens
    corresponding to it have yet not been generated #
173 
174             func_tokens[str(tokens[i][1])] = add_function_tokens(filename, str(tokens[i][1]), func_text,
    func_tokens, class_list)
175             file_tokens.extend(func_tokens[str(tokens[i][1])])  #generate tokens from text and add to
    the func_tokens dictionary#
176 
177         elif tokens[i][0] == pygments.token.Text or tokens[i][0] in pygments.token.Comment or
    tokens[i][0] in pygments.token.Punctuation:
178             pass   #ignore tabs, comments, punctuautions (,.[](){} etc) and other unnecessary text#
179 
180         else:
```

```
181             file_tokens.append(str(tokens[i][1]))      #remaining tokens are identifiers, operators and
        keywords, so are appended to file_tokens#
182
183     if os.path.exists("work"):
184         os.remove("work")   #remove auxillary file#
185
186     print(str(' '.join(file_tokens)))
187     print('\n')
188     return ' '.join(file_tokens)    #return all tokens concatenated as a single string#
```

2.5 rest.checker_core.Final_Checker Namespace Reference

Functions

- def plagCheck (fp1, fp2, boilfp=None)
- def folder_compare (dir_path, boil_path=None)
- def saveres (inpath, outpath, boilpath=None)
- def extract_files (infile)
- def RunCheck (infile, boilfile=None)

2.5.1 Detailed Description

```
This code takes in a path to compressed file containing source code files, invokes tokenizers
depending  the langauge of the input file, and passes the tokenized code to the winnow()
function of the 'winnowing' module, to generate document fingerprints, which are matched
to produce a percentage similarity for every pair of source codes.
It saves the results in the form of a csv file and a pictorial representation of
the similarity matrix
```

2.5.2 Function Documentation

2.5.2.1 extract_files()

```
def rest.checker_core.Final_Checker.extract_files (
              infile )
```

```
infile is path to compressed file, this function extract files to
    'comparisons/input_files'
    folder, into the same base directory as input file
```

Definition at line 188 of file Final_Checker.py.

```
188 def extract_files(infile):
189     """infile is path to compressed file, this function extract files to
190     'comparisons/input_files'
191     folder, into the same base directory as input file"""
192     if infile.endswith(".zip"):
193         filename= os.path.splitext(os.path.basename(infile))[0]
194     if infile.endswith(".tar"):
195         filename= os.path.splitext(os.path.basename(infile))[0]
196     if infile.endswith(".tar.gz"):
197         filename= os.path.splitext(os.path.splitext(os.path.basename(infile))[0])[0]
198
199     dirname1= os.path.dirname(infile)
200     out_dir= os.path.join(dirname1, 'comparisons')
201
202     if os.path.exists(out_dir) and os.path.isdir(out_dir):
```

```
203         shutil.rmtree(out_dir, ignore_errors = False)
204
205     if infile.endswith(".zip"):
206         with zipfile.ZipFile(infile, 'r') as zip_ref:
207             zip_ref.extractall(os.path.join(out_dir, 'input_files'))
208
209     if tarfile.is_tarfile(infile):
210         tf=tarfile.open(infile)
211         tf.extractall(os.path.join(out_dir, 'input_files'))
212     temp=os.listdir(out_dir)
213     temp_dir= temp[0]
214     return out_dir, os.path.join(out_dir,temp_dir)
215
216
217
```

2.5.2.2 folder_compare()

```
def rest.checker_core.Final_Checker.folder_compare (
              dir_path,
              boil_path = None )
```

```
dir_path is the path of the directory containing all the code files to be compared,
    and boil_path is the path to boilerplate code file
    This function invokes tokenizers on various code files and generates the tokenized code
    which it passes to the wiinow() function, along with the 'kval'
    which is actually the size of the kgram used to genrate hash values of the tokenized code.
    Now, these fingerprints are compared pair wise, along with the boilerplate
    fingerprint(if exists), by passing to plagCheck() fucntion
    It returns a simialrity matrix alongwith a list of filenames as an output.
```

Definition at line 60 of file Final_Checker.py.

```
60 def folder_compare(dir_path, boil_path=None):
61
62     """dir_path is the path of the directory containing all the code files to be compared,
63     and boil_path is the path to boilerplate code file
64     This function invokes tokenizers on various code files and generates the tokenized code
65     which it passes to the wiinow() function, along with the 'kval'
66     which is actually the size of the kgram used to genrate hash values of the tokenized code.
67     Now, these fingerprints are compared pair wise, along with the boilerplate
68     fingerprint(if exists), by passing to plagCheck() fucntion
69     It returns a simialrity matrix alongwith a list of filenames as an output.
70
71     """
72     kval=10
73
       file_formats=(".cc",".cxx",".c++",".ii",".ixx",".ipp",".i++",".inl",".idl",".ddl",".odl",".hh",".hxx",".hpp",".h++",".c
74
       ".phtml",".inc",".m",".markdown",".md",".mm",".dox",".pyw",".f90",".f95",".f03",".f08",".f18",".f",".for",".vhd",".vhdl
75     cppfiles=[]
76     filenames=[]
77     sim_mat=[]
78     files_fpr=[]
79     boil_fpr=[]
80     for path, subdirs, files in os.walk(dir_path):
81         for file in files:
82             if file.endswith((".cpp", ".py", ".c", ".h" , ".java")) and not file.startswith('.'):
83                 cppfiles.append(os.path.join(path, file))
84                 filenames.append(file)
85             elif file.endswith(file_formats) and not file.startswith('.'):
86                 cppfiles.append(os.path.join(path, file))
87                 filenames.append(file)
88
89
90     for file in cppfiles:
91         try:
92             if file.endswith((".cpp", ".h", ".c")):
93                 kval = 15
94                 data1 = tokenize_cpp(file)
95
96             if file.endswith(".py"):
97                 kval = 10
```

```
98                 data1 = tokenize_py(file)
99             if file.endswith(".java"):
100                 kval = 15
101                 data1= tokenize_jav(file)
102             if file.endswith(file_formats):
103                 kval = 10
104                 data1 = backup_tokenize(file)
105 
106         except:
107             data1 = backup_tokenize(file)
108 
109 
110         fpr_wpos=[]
111         for fprs in winnow(data1, kval):
112             fpr_wpos.append(fprs[1])
113         files_fpr.append(fpr_wpos)
114 
115     if boil_path != None:
116         try:
117             if boil_path.endswith(".cpp"):
118                 data_b = tokenize_cpp(boil_path)
119             if boil_path.endswith(".py"):
120                 data_b = tokenize_py(boil_path)
121             if boil_path.endswith(".java"):
122                 data_b = tokenize_py(boil_path)
123         except:
124             data_b = backup_tokenize(boil_path)
125 
126         for fprs in winnow(data1, kval):
127             boil_fpr.append(fprs[1])
128     if boil_fpr:
129         for fpr1 in files_fpr:
130             temp=[]
131             for fpr2 in files_fpr:
132                 temp.append(plagCheck(fpr1,fpr2, boil_fpr))
133             sim_mat.append(temp)
134     else:
135         for fpr1 in files_fpr:
136             temp=[]
137             for fpr2 in files_fpr:
138                 temp.append(plagCheck(fpr1,fpr2))
139             sim_mat.append(temp)
140 
141     res_mat = np.array(sim_mat)
142     return res_mat, filenames
143 
144 
145 
146 
```

2.5.2.3 plagCheck()

```
def rest.checker_core.Final_Checker.plagCheck (
                fp1,
                fp2,
                boilfp = None )
```

fp1 and fp2 are the fingerprints of the two files to be compared. These fingerprints have been generated from winnowing, the method is explained below.
boilfp is the fingerprint of boilerplate code.
This function finds the common fingerprints of the two files and returns the ratio of matched fingerprints and total fingerprints.
If boilerplate is given by the user, it removes all the common fingerprints for the two files with boilerplate

Definition at line 27 of file Final_Checker.py.

```
27 def plagCheck(fp1,fp2, boilfp=None):
28 
29     """
30     fp1 and fp2 are the fingerprints of the two files to be compared. These fingerprints
31     have been generated from winnowing, the method is explained below.
32     boilfp is the fingerprint of boilerplate code.
33     This function finds the common fingerprints of the two files and returns the ratio of
```

```
34      matched fingerprints and total fingerprints.
35      If boilerplate is given by the user, it removes all the common fingerprints for the
36      two files with boilerplate
37          """
38
39      if boilfp != None:
40          tempfp1=set(fp1).difference(boilfp)
41          tempfp2 = set(fp2).difference(boilfp)
42      else:
43          tempfp1 = set(fp1)
44          tempfp2 = set(fp2)
45      """A list of common fingerprints"""
46      comfpr=list(tempfp1 & tempfp2)
47
48
49      deno = min(len(tempfp1),len(tempfp2))
50
51      if deno ==0:
52          ratio =0.0
53      else:
54          ratio= len(comfpr)/deno
55
56      """returns the ratio of matches and toal fingerprints, we have used minimum of the number of
        fingerprints in the denominator i.e., for comparisons,
57      this is a fair assumption, based on tested results(makes it more sensitive to even small chunks of
        plagiarized snippets of codes"""
58      return ratio
59
```

2.5.2.4 RunCheck()

```
def rest.checker_core.Final_Checker.RunCheck (
              infile,
              boilfile = None )
```

This function takes in the path to input compressed files, and boilerplate code and invokes extract_files() function to extract the files and then savres() function to generate and save the results in the 'comparisons/results' folder

Definition at line 218 of file Final_Checker.py.

```
218 def RunCheck(infile, boilfile=None):
219
220     """This function takes in the path to input compressed files, and boilerplate code
221     and invokes extract_files() function to extract the files and then savres()
222     function to generate and save the results in the 'comparisons/results' folder """
223
224     formats=(".tar", ".tar.gz", ".zip")
225     if infile.endswith(formats):
226         try:
227             out_dir , files_dir = extract_files(infile)
228             res_dir= os.path.join(out_dir, 'results')
229             os.mkdir(res_dir)
230             if boilfile==None:
231                 saveres(files_dir, res_dir)
232             else:
233                 saveres(files_dir, res_dir, boilfile)
234             """returns 'success' and path to directory having generated results"""
235             return 'success' , res_dir
236             """ returns 'fail' in all other scenarios"""
237         except:
238             return 'fail' , ''
239
240     return 'fail', ''
241
242
243
```

2.5.2.5 saveres()

```
def rest.checker_core.Final_Checker.saveres (
              inpath,
              outpath,
              boilpath = None )
```

```
    inpath is path to directory containing code and boilpath is path to boilerplate code file.
    This function basically calls folder_compare() function on the input directory and
    saves the result in the form of csv to the output path(outpath),
    It also generates a graphical respresentation of the result and savs it in the
    outpath folder.
```

Definition at line 147 of file Final_Checker.py.

```
147 def saveres(inpath, outpath, boilpath=None):
148     """
149     inpath is path to directory containing code and boilpath is path to boilerplate code file.
150     This function basically calls folder_compare() function on the input directory and
151     saves the result in the form of csv to the output path(outpath),
152     It also generates a graphical respresentation of the result and savs it in the
153     outpath folder.
154     """
155
156     if boilpath==None:
157         matres, filenames=folder_compare(inpath)
158     else:
159         matres, filenames=folder_compare(inpath, boilpath)
160
161     extentt=np.arange(len(filenames)) + 0.5
162
163     """using pandas to generate dataframe and save it as csv from the similarity matrix"""
164
165     df = pd.DataFrame(matres, index= filenames, columns=filenames)
166
167     df.to_csv(os.path.join(outpath, 'results.csv'))
168
169
170     """using matplotlib to generate an image shwoing degree of plagiarism in a pair of file"""
171
172     fig, ax = plt.subplots(1,1)
173
174     img = ax.imshow(matres,cmap='Reds', vmin=0, vmax=1, extent=[0, len(filenames), 0, len(filenames)] )
175
176     ax.set_xticks(extentt)
177     ax.set_yticks(extentt)
178
179     ax.set_xticklabels(filenames, rotation= 60)
180     ax.set_yticklabels(filenames[::-1])
181
182     fig.colorbar(img)
183     plt.tight_layout()
184     plt.savefig(os.path.join(outpath, 'results.png'))
185
186
187
```