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):

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     """
30     file = open(filename, "r")
31
32     if os.path.exists("work"):
33         os.remove("work")
34
35     work = open('work', 'a')        #an auxillary file created to store the cource code tet with extra
       whitespace removed#
36
37     for l in file:
38         if l == '' or l.isspace():      #ignore whitespace#
39             pass
40         else:
41             work.write(l.rstrip())
42             work.write('\n')
43
44     file.close()
45     work.close()
46
47     file = open('work', 'r')    #read all text from auxillary file#
48     text = file.read()
49
50     lexer = pygments.lexers.guess_lexer_for_filename(filename, text)        #obtain lexer from pygmnets #
51     tokens = lexer.get_tokens(text)
52     tokens = list(tokens)
53     result = []
54     lenT = len(tokens)
55     file_tokens = []
56     class_list = []
57
58
59
60     for i in range(lenT):
61
62         if tokens[i][0] == pygments.token.Name.Function:
63             file_tokens.append('funct')
64
65         elif tokens[i][0] == pygments.token.Name.Class or str(tokens[i][1]) in class_list:      #identify
       instances of classes and update class_list#
66             class_list.append(str(tokens[i][1]))                                                #identify
       objects/ instances of user defined classes and assign 'class' token to it#
67             file_tokens.append('class')
68
69         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] == '(':
70
71             if str(tokens[i][1]) in class_list:                                                 #identify
       objects/ instances of user defined classes and assign 'class' token to it#
72                 file_tokens.append('class')
73
74             elif tokens[i][0] in pygments.token.Name.Namespace:                                 #identify
       namespaces and add them#
75                 file_tokens.extend(str(tokens[i][0]).split())
76
77             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 :
78                 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#
79
80             else:
81                 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#
82
83         elif tokens[i][0] in pygments.token.Literal.String:
84             pass
85
86         elif tokens[i][0] == pygments.token.Text or tokens[i][0] in pygments.token.Comment or
       tokens[i][0] in pygments.token.Punctuation:
87             pass   #whitespaces and comments ignored
88
89         else:
90             file_tokens.append(str(tokens[i][1]))           #remaining tokens are identifiers, operators
       and keywords, so are appended to file_tokens#
91
```

```
92     if os.path.exists("work"):
93         os.remove("work")
94
95     return ''.join(file_tokens)
96
```

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 81 of file checker_cpp.py.

```
81 def add_func(token, func_tokens):
82     "'takes the dictionary of func_tokens and function name as argument and returns a
83         list of tokens corresponding to the function
84     Invoked whnever a function is called and token is name of the function"'
85     new_list = []
86     for t in func_tokens[token]:
87         if t != token:
88             if t in func_tokens:
89                 new_list = new_list + add_func(t, func_tokens)
90             else:
91                 new_list.append(t)
92
93     return new_list
94
95
96
```

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 97 of file checker_cpp.py.

```
97 def basicCheck(token, file_tokens, func_tokens, class_list):
98
99     '''
100     token is single token to be processed now
101     file_tokens is the list to which the token might be added
102     func_tokens is a dictionary with function names mapped to corresponding declarations and is
103         used to add tokens corresponding to a function call
104     class_list is a list of classes to identify object instances
105
106     This fucntion examines the given token and determines whether it needs to be appended.
107         Whitespaces, comments, delimiters/punctuation/literals are ignored.
108     Tokens which are keywords/ identifiers/ operators are added as strings to the file_tokens list
109     Variables names are assigned token with 'v' keyword
110     Numbers of any type(int/ float) are assigned token with 'no' keyword
111     Headers of any type are assigned token with 'he' keyword
112     Objects/ instances of a class are assigned token woth 'obj' keyword
113     For function calls, add_func is passes the function name and tokens corresponding to the function
114         are added to file_tokens
115     '''
116     global scope_depth, is_function
117     varPtrn = re.compile(r"[a-zA-Z_][a-zA-Z0-9_]")  # variables
118     headerPtrn = re.compile(r"\w[a-zA-Z]+[.]h")  # header files
119     digitPtrn = re.compile(r'\d')   #digits
120     floatPtrn = re.compile(r'\d+[.]\d+')    #decimals
121
122     if token in mysrc.delimiters():
123         description = mysrc.delimiters()[token]
124         if description == 'LCBRACE':
125             scope_depth += 1
126
127         elif description == 'RCBRACE':
128             scope_depth -= 1
129             if is_function != -1 and scope_depth == 0:
130                 is_function = -1
131
132         else:
133             pass
134     elif token in mysrc.keywords():
135
136         if is_function != -1:
137             pass
138         else:
139             file_tokens.append(token)
140
141     elif token in mysrc.identifiers():
142
143         if is_function != -1:
144             pass
145         else:
146             file_tokens.append(token)
147
148     elif token in mysrc.operators().keys():
149
150         if is_function != -1:
151             pass
152         else:
153             file_tokens.append(token)
154
155     elif re.search(headerPtrn, token):
156
157         file_tokens.append('head')
158
159     elif token in func_tokens.keys() and token != 'main':
160         file_tokens.extend(add_func(token, func_tokens))
161
162
163     elif token in class_list:
164         file_tokens.append('obj' )
165
166     elif token == 'head':
167         file_tokens.append('he')
168
169     elif token == 'num':
170         file_tokens.append('no')
171
172     elif token == 'obj':
173         file_tokens.append('obj')
174
175     elif re.match(varPtrn, token) or "'" in token or '"' in token:
176         if is_function != -1:
177             pass
178         else:
179             file_tokens.append('v')
180     elif re.match(digitPtrn, token):
181
182         if is_function != -1:
```

```
183             pass
184         else:
185             file_tokens.append('no')
186 
187     return True
188 
```

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 336 of file checker_cpp.py.

```
336 def class_n_func_tokens(class_all_list, func_all_list):
337 
338     '''
339     Takes as input list of all classes and functions in the file as identified by the pygccxml parser
340     Generates new lists - func_list and class_list of user defined functions and classes
341     For functions - the line number where function definition begins is identified using regex
342         matching and stored in func_start with the same order as func_list
343     For Classses - constructors, operators, variables and memeber functions corresponding to each user
344     defined class are obtained from the parser and added to dictionary class_tokens mapped to class name
345     Returns the above generated lists/dictionaries
346     '''
347 
348     func_start = []
349     func_list = []
350     func_tokens = []
351     class_tokens = {}
352     f = open('work', 'r')
353     txt = f.read()
354     for func in func_all_list:
355         pat = r"\s*"+str(func.name)+r"+\s*\("
356         res = re.findall(pat, txt)
357         if (len(res)>0):
358             func_list.append(str(func.name))
359             pat2 = r"\s*"+str(func.name)+r"+\s*\(([\w+\s+\w+])*\)\s*\{"
360             pos = re.search(pat2, txt)
361             if pos != None:
362                 line_no = len(re.findall('\n', txt[0:int(pos.start())]))
363                 func_start.append(line_no)
364 
365     class_list = []
366     for class_ in reversed(class_all_list):
367         pat = r"\s*"+str(class_.name)+r"+\s*\{"
368         res = re.findall(pat, txt)
369         if (len(res)>0):
370             class_tokens[class_.name] = []
371             for base in class_.bases:
372                 class_tokens[class_.name].extend(base.related_class.name.split())
373 
374             for derive in class_.derived:
375                 class_tokens[class_.name].extend(derive.related_class.name.spilt())
376 
377             for p in class_.constructors(allow_empty = True):
378                 if p is None:
379                     break
380                 for a in p.argument_types:
381                     class_tokens[class_.name].extend(str(a).split())
382 
383             for p in class_.operators(allow_empty = True):
384                 if p is not None:
385                     p = re.sub(r'operator', r'ope', str(p.name))
386                     class_tokens[class_.name].append(p)
387 
```

```
388             for p in class_.variables(allow_empty = True):
389                 class_tokens[class_.name].append(str(p.decl_type))
390 
391             for p in class_.member_functions(allow_empty = True):
392                 if p is None:
393                     break
394                 for a in p.argument_types:
395                     class_tokens[class_.name].append(str(a))
396 
397             class_list.append(str(class_.name))
398 
399     return func_list, func_start, class_list, class_tokens
400 
```

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 267 of file checker_cpp.py.

```
267 def delimiterCorrection(line):
268 
269     "'Takes a line as input and splits it into tokens using whitespace as separator
270     To ensure that delimiters are taken into account poistion of delimeters are identified and
271         replaced with padding of spaces around them for effective splitting
272     Returned is list of tokens generated from the line excluding whitespaces
273     The tokens generated now are just words in the source code file, they need to be processed further"'
274 
275     for delim in mysrc.delimiters().keys():
276         if delim in line:
277             line = line.replace(delim, ' '+delim+' ')
278 
279     tokens = line.split(" ")
280     for delimiter in mysrc.delimiters().keys():
281         for token in tokens:
282 
283             if token == delimiter:
284                 pass
285             elif delimiter in token:
286 
287                 pos = token.find(delimiter)
288                 tokens.remove(token)
289                 token = token.replace(delimiter, " ")
290                 extra = token[:pos]
291                 token = token[pos + 1 :]
292                 tokens.append(delimiter)
293                 tokens.append(extra)
294                 tokens.append(token)
295             else:
296                 pass
297     for token in tokens:
298         if isWhiteSpace(token):
299             tokens.remove(token)
300         elif ' ' in token:
301             tokens.remove(token)
302             token = token.split(' ')
303             for d in token:
304                 tokens.append(d)
305     return tokens
306 
```

2.2.2.5 delimiters()

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

a dictionary of cpp delimiters

Definition at line 73 of file checker_cpp.py.

```
73 def delimiters():
74     "'a dictionary of cpp delimiters"'
75     delimiters = {
76     "\t": "TAB", "\n": "NEWLINE","(": "LPAR", ")": "RPAR", "[": "LBRACE", "]": "RBRACE", "{": "LCBRACE",
       "}": "RCBRACE", "=": "ASSIGN",":": "COLON", ",": "COMMA", ";": "SEMICOL", "«": "OUT", "»": "IN",
77     }
78     #print(len(delimiters)) = 14
79     return delimiters
80
```

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 189 of file checker_cpp.py.

```
189 def funcCheck(token, func_tokens, func_list, class_list):
190
191     "'
192     token is single token to be processed now
193     file_tokens is the list to which the token might be added
194     func_tokens is a dictionary with function names mapped to corresponding declarations and is used
195         to add tokens corresponding to a function call
196     class_list is a list of classes to identify object instances
197
198     Similar to basicChecker but works on processing tokens of a particular function. The name of
199         current function being processed is stored in is_function var.
200     It appends the generated token value to the dictionary func_tokens mapped to corresponding name
201     "'
202
203     global scope_depth, is_function
204     varPtrn = re.compile(r"[a-zA-Z_][a-zA-Z0-9_]")  # variables
205     headerPtrn = re.compile(r"\w[a-zA-Z]+[.]h")  # header files
206     digitPtrn = re.compile(r'\d') #digits
207     floatPtrn = re.compile(r'\d+[.]\d+') #decimals
208
209     if token in mysrc.delimiters():
210         description = mysrc.delimiters()[token]
211         if description == 'LCBRACE':
212             scope_depth += 1
213
214         elif description == 'RCBRACE':
215             scope_depth -= 1
216             if is_function != -1 and scope_depth == 0:
217                 is_function = -1
218
219         else:
```

```
220             pass
221     elif token in mysrc.keywords():
222 
223         if is_function != -1:
224             func_tokens[is_function].append(token)
225         else:
226             pass
227     elif token in mysrc.identifiers():
228 
229         if is_function != -1:
230             func_tokens[is_function].append(token)
231         else:
232             pass
233 
234     elif token in mysrc.operators().keys():
235 
236         if is_function != -1:
237             func_tokens[is_function].append(token)
238         else:
239             pass
240     elif token in func_list and token != is_function and is_function!= -1:
241         func_tokens[is_function].append(token)
242 
243     elif token in class_list and is_function!= -1:
244         func_tokens[is_function].append('obj' )
245 
246     elif re.search(headerPtrn, token):
247 
248         pass
249     elif re.match(varPtrn, token) or "'" in token or '"' in token:
250 
251         if is_function != -1:
252             func_tokens[is_function].append('v')
253         else:
254             pass
255 
256 
257     elif re.match(digitPtrn, token):
258 
259         if is_function != -1:
260             func_tokens[is_function].append('no')
261             pass
262         else:
263             pass
264 
265     return True
266 
```

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 318 of file checker_cpp.py.

```
318 def hasWhiteSpace(token):
319 
320     '''
321     Checks if a token has a whitespace in it
322     If it is present, it is interpreted as aliteral and returned with single quotes added
323     Else return false
324     '''
325     ptrn = ['\t', '\n']
326     if isWhiteSpace(token) == False:
327         for item in ptrn:
328             if item in token:
329                 result = "'" + item + "'"
330                 return result
331             else:
332                 pass
333     return False
334 
335 
```

2.2.2.8 identifiers()

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

a list of cpp identifiers

Definition at line 59 of file checker_cpp.py.

```
59 def identifiers():
60     "'a list of cpp identifiers"'
61     identifiers = [
62     "auto", "bool", "char", "double", "enum", "float", "int", "long", "short", "string" ]
63     #print(len(identifiers)) = 10
64     return identifiers
65
```

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 307 of file checker_cpp.py.

```
307 def isWhiteSpace(word):
308     "'
309     takes token as input and return true if it comes under whitespace else false
310     "'
311     ptrn = [ " ", "\t", "\n"]
312     for item in ptrn:
313         if word == item:
314             return True
315         else:
316             return False
317
```

2.2.2.10 keywords()

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

a list of cpp keywords

Definition at line 50 of file checker_cpp.py.

```
50 def keywords():
51     "'a list of cpp keywords"'
52     keywords = [
53     "break", "case", "catch", "word", "class", "const", "continue", "delete", "do", "else", "false",
       "for", "goto", "if",
54     "namespace", "not", "or", "private", "protected", "public", "return", "signed", "sizeof", "static",
       "struct", "switch", "true", "try", "unsigned", "void", "while",
55     "endl", "cout", "cin", 'queue', 'stack', 'vector', 'array', 'list', 'forward_list', 'set' ,'map',
       'deque', 'priority_queue', 'multiset', 'multimap',
56     ]
57     return keywords
58
```

2.2.2.11 operators()

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

a dictionary of cpp operators

Definition at line 66 of file checker_cpp.py.

```
66 def operators():
67     "'a dictionary of cpp operators"'
68     operators = {
69     "+": "PLUS", "-": "MINUS", "*": "MUL", "/": "DIV", "%": "MOD", "+=": "PLUSEQ", "-=": "MINUSEQ", "*=":
       "MULEQ", "/=": "DIVEQ", "++": "INC", "--": "DEC", "|": "OR", "&&": "AND", "&": "REF",
70     }
71     return operators
72
```

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 is examined
480         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 tokens
482         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 and is
508         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 401 of file checker_cpp.py.

```
401 def tokenize(path, file_tokens, func_tokens, class_all_list, func_all_list):
402
403     '''
404     path is the path of file to be processed
405     file_tokens is the list which will store all the tokens generated from the file
406     func_tokens is a dictionary with function names mapped to corresponding declarations and is used
407         to add tokens corresponding to a function call
408     class_all_list and func_all_list are lists of all classes and functions in the file as identified
409         by the pygccxml parser
410
411     This function first invokes class_n_func_tokens to generate information about functions and tokens
412         corresponding to classes
413     It then invokes funcCheck to generate tokens corresponding to a fucntion and 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 43 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 their list
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 15 of file checker_java.py.

```
15 def tokenize_jav(filename):
16
17     """
18     This function takes filename as input and returns the tokenized version of source code as string.
19     It first removes all extra whitespaces. Then it identifies classes and functions
20         and prepares their list
21     Subsequently the remaining files is tokenized and list of tokens stored in file_tokens
22     Whenever a fucntion call/ class instance/ variable name is encountered, specific keywords are
23         used as tokens ('function'/ 'class'/ 'var')
24     Comments, punctutation, literals are ignored
25     """
26
27     file = open(filename, "r")
28     if os.path.exists("work"):
29         os.remove("work")
30
31     work = open('work', 'a')        #an auxillary file created to store the cource code tet with extra
       whitespace removed#
32     in_func = -1
33
34     for l in file:
35         if l == '' or l.isspace():      #ignore whitespace#
36             pass
37         else:
38             work.write(l.rstrip())      #remove trailing space and write to auxillary file#
39             work.write('\n')
40
41     file.close()
42     work.close()
43
44     file = open('work', 'r')
45     text = file.read()      #read all text from auxillary file#
46
47     lexer = pygments.lexers.guess_lexer_for_filename(filename, text)
48     tokens = lexer.get_tokens(text)
49     tokens = list(tokens)
50     func_list = []      #list to store all the function names#
51     lenT = len(tokens)
52     file_tokens = []       #list to store the tokens corresponding to the entire source code file#
53     class_list = []     #list to store all the user defined classes#
54
55
```

```
56      #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#
57      key_names = {'String' : 'str', 'ArrayList' : 'array', 'List': 'list', 'LinkedList': 'linked',
        'HashMap': 'hashma', 'HashSet':' hashse', 'BufferedReader': 'buffer',
58      'ArithmeticException' : 'arithmex', 'ArrayIndexOutOfBoundsException' : 'arrinoex', 'Iterator':
        'iterat', 'Pattern': 'pater', 'Matcher': 'match', 'PatternSyntaxException': 'patsynex',
59      'ClassNotFoundException': 'clasnoex', 'FileNotFoundException': 'filenoex', 'IOException': 'inpoutex',
        'InterruptedException': 'intexex', 'NoSuchFieldException': 'nofileex',
60      'NoSuchMethodException': 'nomethex', 'NullPointerException': 'nulponex', 'NumberFormatException':
        'numforex', 'RuntimeException': 'runtimex',
61      'StringIndexOutOfBoundsException': 'strioex', 'LocalDate': 'locdat', 'LocalTime': 'loctim',
        'LocalDateTime': 'dattim', 'DateTimeFormatter': 'dtform',
62      'Thread': 'thread', 'Main': 'main', 'Runnable': 'runble', 'Consumer': 'consum', 'private': 'scp',
        'public': 'scp', 'protected': 'scp',
63      'FileReader': 'filred', 'FileInputStream': 'fileinpstr', 'FileWriter': 'filewrit', 'BufferedWriter':
        'bufwrit', 'FileOutputStream': 'filoutstr',
64      'abstract': 'abstract', 'implements': 'implement', 'enum': 'enum', 'interface': 'interface', 'final':
        'final', 'extends': 'extends', 'forEach': 'forEa'}
65
66      #list of java's inbuilt methods for files#
67      file_methods = ['File', 'canRead', 'canWrite', 'createNewFile', 'delete', 'exists', 'getName',
        'length', 'list', 'mkdir', 'getAbsolutePath', 'FileWriter', 'write', 'close']
68
69
70      #list of java's inbuilt methods for strings#
71      string_methods = ['charAt', 'codePointAt', 'codePointBefore', 'codePointCount', 'compareTo',
        'compareToIgnoreCase', 'concat', 'contains', 'contentEquals', 'copyValueOf', 'endsWith', 'equals',
72          'equalsIgnoreCase', 'format', 'getBytes', 'getChars', 'hashCode', 'indexOf', 'intern', 'isEmpty',
        'lastIndexOf', 'length', 'matches', 'offsetByCodePoints', 'regionMatches', 'replace', 'replaceFirst',
73          'substring', 'toCharArray', 'toLowerCase', 'toString', 'toUpperCase', 'trim', 'valueOf']
74
75
76      #list of java's inbuilt methods for mathematical operations#
77      math_methods = ['abs', 'acos', 'asin', 'atan', 'atan2', 'cbrt', 'ceil', 'copySign', 'cos', 'cosh',
        'exp', 'expm1', 'floor', 'getExponent', 'hypot', 'log', 'log10', 'log1p', 'max',
78      'min', 'nextAfter', 'nextUp', 'pow', 'random', 'round', 'rint', 'signum', 'sin', 'sinh', 'sqrt',
        'tan', 'tanh', 'toDegrees', 'toRadians', 'ulp']
79
80      for i in range(lenT):
81          if tokens[i][0] == pygments.token.Name.Function:    #identify functions and update func_list#
82              func_list.append(str(tokens[i][1]))
83
84          elif tokens[i][0] == pygments.token.Name.Class:     #identify classes and update class_list#
85              class_list.append(str(tokens[i][1]))
86
87
88      for i in range(lenT):
89          if tokens[i][0] in pygments.token.Punctuation :  #punctuautions (,.[](){} etc) are ignored#
90              pass
91
92          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#
93              file_tokens.append('function')
94
95          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#
96              file_tokens.append('class')
97
98
99          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#
100
101             t = str(tokens[i][1])
102
103             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#
104                 file_tokens.append('class')
105
106             elif tokens[i][0] in pygments.token.Name.Namespace:     #identify imports and obtain a short
        keyword for their type#
107                 toks = t.split('.')[-1]
108                 if toks in key_names.keys():
109                     file_tokens.append(key_names[toks])
110
111             elif tokens[i][0] in pygments.token.Name.Builtin or tokens[i][0] in
        pygments.token.Name.Decorator :
112                 file_tokens.append(t)       #identify builtin methods of java and decorators and add the
        token as string form to the list of file tokens#
113
114             elif tokens[i][0] in pygments.token.Name.Attribute:
115                 file_tokens.append('fun')
116
117             else:
118                 #check if the token is included in our defined vocabulary#
119
120                 if t in key_names.keys():
121                     file_tokens.append(key_names[t])
```

```
122
123                 elif t in file_methods:
124                     file_tokens.append(t)
125
126                 elif t in string_methods:
127                     file_tokens.append(t)
128
129                 elif t in math_methods:
130                     file_tokens.append(t)
131                 else:
132                     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#
133
134         elif tokens[i][0] == pygments.token.Name.Class:     #assign keyword 'class' to class
    declarations and its object instances#
135             file_tokens.append('class')
136
137         elif tokens[i][0] == pygments.token.Name or tokens[i][0] in pygments.token.Name:
138             file_tokens.append('var')
139
140         elif tokens[i][0] in pygments.token.Literal.String:
141             pass      #ignore values of string type#
142
143         elif tokens[i][0] == pygments.token.Text or tokens[i][0] in pygments.token.Comment:
144             pass   #ignore tabs, comments and other unnecessary text#
145
146         elif str(tokens[i][1]) == 'import':     #import keyword is ignored#
147             pass
148
149         else:
150             t = str(tokens[i][1])       #remaining tokens are identifiers, operators and keywords, so
    are appended to file_tokens#
151             if t in key_names.keys():
152                 file_tokens.append(key_names[t])
153             else:
154                 file_tokens.append(t)
155
156      if os.path.exists("work"):
157          os.remove("work")
158      return ''.join(file_tokens)
159
```

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
18     *filename is used while obtaining the lexer for python from pygments module
19
20     *name is the name of the function for which tokens returned by this function
21
22     * func_text is a dictionary which maps the name of the function to the entire body of the function in

23         textual form extracted from the source code file with unnecessary whitespaces and comments
       removed.
24     In case the function is encountered for the first time and its tokens have yet not been generated,
25         func_text will be used to generate tokens and add to func_tokens dictionary
26
27     * func_tokens dictionary maps name of the function to tokens generated from the function
28
29     * class_list stores the list of class names and will be helpful while generating tokens to
30         identify objects/ instances of classes defined by the user
31     """
32     text = func_text[name] #extract text of the required function#
33     lexer = pygments.lexers.guess_lexer_for_filename(filename, text) #obtain lexer from pygmnets #
34     tokens = lexer.get_tokens(text) #generate tokens from the code#
35     tokens = list(tokens)
36     lenT = len(tokens) #length of tokens#
37     file_tokens = [] #list to store the tokens corresponding to fucntion if yet to be generated#
38
39     for i in range(lenT):
40         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#
41             class_list.append(str(tokens[i][1]))
42
43     for i in range(lenT):
44
45         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#
46
47             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#
48                 file_tokens.append('class')
49
50             elif tokens[i][0] in pygments.token.Name.Builtin or tokens[i][0] in
       pygments.token.Name.Function \
51                     or tokens[i][0] in pygments.token.Name.Attribute or tokens[i][0] in
       pygments.token.Name.Decorator \
```

```
52                  or tokens[i][0] in pygments.token.Name.Namespace:  #identify builtin methods of
    python, decorators and namespaces and add the token as string form#
53              file_tokens.append(str(tokens[i][1]))
54
55          else:
56              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#
57
58
59      elif tokens[i][0] == pygments.token.Name.Class: #identify objects/ instances of builtin/other
    classes and assign 'class' token to it#
60          class_list.append(str(tokens[i][1]))
61          file_tokens.append('class')
62
63      elif tokens[i][0] in pygments.token.Literal.String: #ignore values of string type#
64          pass
65
66      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#
67          file_tokens.extend(func_tokens[str(tokens[i][1])])
68
69      elif str(tokens[i][1]) in func_text.keys():     #if function call is encountered and tokens
    corresponding to it have yet not been generated #
70
71          if str(tokens[i][1]) != name:    #check if recursive call#
72              func_tokens[str(tokens[i][1])] = add_function_tokens(filename, str(tokens[i][1]),
    func_text, func_tokens, class_list)
73              file_tokens.extend(func_tokens[str(tokens[i][1])]) #generate tokens from text and add to
    the func_tokens dictionary#
74
75          else:
76              pass
77      elif tokens[i][0] == pygments.token.Text or tokens[i][0] in pygments.token.Comment or
    tokens[i][0] in pygments.token.Punctuation:
78          pass   #ignore tabs, comments, punctuautions (,.[](){} etc) and other unnecessary text#
79
80      else:
81          file_tokens.append(str(tokens[i][1]))  #remaining tokens are identifiers and keywords, so are
    appended to file_tokens#
82      return file_tokens
83
84
85
```

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 86 of file checker_py.py.

```
86 def tokenize_py(filename):
87
88     """
89     This function takes filename as input and returns the tokenized version of source code as string.
90     It first removes all extra whitespaces.Then all functions are identified and their text is code
91          is removed and stored in a separate dictionary func_text
92     Tokens corresponding to the functions are generated and their list is mapped to the function
93         name in another dictionary func_tokens
94     Subsequently the remaining files is tokenized and list of tokens stored in file_tokens
95     Whenever a fucntion call is encountered, tokens corresponding to the fucntion are appended
96     """
97
98     file = open(filename, "r")
99     if os.path.exists("work"):
100         os.remove("work")
101
```

```
102     work = open('work', 'a') #an auxillary file created to store the cource code tet with extra
        whitespace removed#
103     func_text = {}
104     pat = r'^def +(\w)*\(.*?\):' #matches with python function declaration#
105     line_no = 0
106     func_pos = []
107     in_func = -1
108
109     for l in file:
110         if l == '' or l.isspace():  #ignore whitespace#
111             pass
112         elif l[0] == '\t' and in_func != -1:
113             func_text[name] += l   #if inside function, add code to the func_text dictionary
        corresponding to the function name#
114
115         else:
116             match = re.search(pat, l)  #check if line has function declaration#
117             in_func = -1
118
119             if match is not None:
120                 name = match.string.split()[1]
121                 name = name.split('(')[0]
122                 func_pos.append(line_no)
123                 in_func = name
124                 func_text[name] = ''  #create a new value in dictionary func_text if new fucntion found#
125
126             else:
127                 work.write(l.rstrip()) #remove trailing space and write to auxillary file#
128                 work.write('\n')
129         line_no += 1
130
131     file.close()
132     work.close()
133     file = open('work', 'r')
134     text = file.read() #read all text from auxillary file#
135
136     lexer = pygments.lexers.guess_lexer_for_filename(filename, text) #obtain lexer from pygmnets #
137     tokens = lexer.get_tokens(text)
138     tokens = list(tokens)
139     lenT = len(tokens)
140     file_tokens = []    #list to store the tokens corresponding to the entire source code file#
141     func_tokens = {}
142     class_list = []     #list to store all the user defined classes#
143
144     for i in range(lenT):
145         if tokens[i][0] == pygments.token.Name.Class:    #identify instances of classes and update
        class_list#
146             class_list.append(str(tokens[i][1]))
147
148     for i in range(lenT):
149         if tokens[i][0] == pygments.token.Name.Class:
150             class_list.append(str(tokens[i][1]))
151             file_tokens.append('class')
152
153         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#
154
155             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#
156                 file_tokens.append('class')
157
158             elif tokens[i][0] in pygments.token.Name.Builtin or tokens[i][0] in
        pygments.token.Name.Function \
159                     or tokens[i][0] in pygments.token.Name.Attribute or tokens[i][0] in
        pygments.token.Name.Decorator \
160                     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#
161                 file_tokens.append(str(tokens[i][1]))
162
163             else:
164                 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#
165
166         elif tokens[i][0] in pygments.token.Literal.String:     #ignore values of string type#
167             pass
168
169         elif tokens[i][0] == pygments.token.Name.Class: #identify objects/ instances of builtin/other
        classes and assign 'class' token to it#
170             class_list.append(str(tokens[i][1]))
171             file_tokens.append('class')
172
173         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#
174             file_tokens.extend(func_tokens[str(tokens[i][1])])
175
176         elif str(tokens[i][1]) in func_text.keys():     #if function call is encountered and tokens
        corresponding to it have yet not been generated #
```

```
177
178             func_tokens[str(tokens[i][1])] = add_function_tokens(filename, str(tokens[i][1]), func_text,
    func_tokens, class_list)
179             file_tokens.extend(func_tokens[str(tokens[i][1])])  #generate tokens from text and add to
    the func_tokens dictionary#
180
181         elif tokens[i][0] == pygments.token.Text or tokens[i][0] in pygments.token.Comment or
    tokens[i][0] in pygments.token.Punctuation:
182             pass   #ignore tabs, comments, punctuautions (,.[](){} etc) and other unnecessary text#
183
184         else:
185             file_tokens.append(str(tokens[i][1]))      #remaining tokens are identifiers, operators and
    keywords, so are appended to file_tokens#
186
187     if os.path.exists("work"):
188         os.remove("work")   #remove auxillary file#
189
190     print(str(' '.join(file_tokens)))
191     print('\n')
192     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 176 of file Final_Checker.py.

```
176 def extract_files(infile):
177     """infile is path to compressed file, this function extract files to 'comparisons/input_files'
178         folder, into the same base directory as input file"""
179     if infile.endswith(".zip"):
180         filename= os.path.splitext(os.path.basename(infile))[0]
181     if infile.endswith(".tar"):
182         filename= os.path.splitext(os.path.basename(infile))[0]
183     if infile.endswith(".tar.gz"):
184         filename= os.path.splitext(os.path.splitext(os.path.basename(infile))[0])[0]
185 
186     dirname1= os.path.dirname(infile)
187     out_dir= os.path.join(dirname1, 'comparisons')
188 
189     if os.path.exists(out_dir) and os.path.isdir(out_dir):
190         shutil.rmtree(out_dir, ignore_errors = False)
191 
192     if infile.endswith(".zip"):
193         with zipfile.ZipFile(infile, 'r') as zip_ref:
194             zip_ref.extractall(os.path.join(out_dir, 'input_files'))
195 
196     if tarfile.is_tarfile(infile):
197         tf=tarfile.open(infile)
198         tf.extractall(os.path.join(out_dir, 'input_files'))
199     temp=os.listdir(out_dir)
200     temp_dir= temp[0]
201     return out_dir, os.path.join(out_dir,temp_dir)
202 
203 
204 
```

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     cppfiles=[]
74     filenames=[]
75     sim_mat=[]
76     files_fpr=[]
77     boil_fpr=[]
78     for path, subdirs, files in os.walk(dir_path):
79         for file in files:
80             if file.endswith((".cpp", ".py", ".java")) and not file.startswith('.'):
81                 cppfiles.append(os.path.join(path, file))
82                 filenames.append(file)
83 
84     for file in cppfiles:
85         try:
```

```
86              if file.endswith(".cpp"):
87                  kval = 15
88                  data1 = tokenize_cpp(file)
89              if file.endswith(".py"):
90                  kval = 10
91                  data1 = tokenize_py(file)
92              if file.endswith(".java"):
93                  kval = 15
94                  data1= tokenize_jav(file)
95          except:
96              data1 = backup_tokenize(file)
97
98
99          fpr_wpos=[]
100          for fprs in winnow(data1, kval):
101              fpr_wpos.append(fprs[1])
102          files_fpr.append(fpr_wpos)
103
104      if boil_path != None:
105          try:
106              if boil_path.endswith(".cpp"):
107                  data_b = tokenize_cpp(boil_path)
108              if boil_path.endswith(".py"):
109                  data_b = tokenize_py(boil_path)
110              if boil_path.endswith(".java"):
111                  data_b = tokenize_py(boil_path)
112          except:
113              data_b = backup_tokenize(boil_path)
114
115          for fprs in winnow(data1, kval):
116              boil_fpr.append(fprs[1])
117      if boil_fpr:
118          for fpr1 in files_fpr:
119              temp=[]
120              for fpr2 in files_fpr:
121                  temp.append(plagCheck(fpr1,fpr2, boil_fpr))
122              sim_mat.append(temp)
123      else:
124          for fpr1 in files_fpr:
125              temp=[]
126              for fpr2 in files_fpr:
127                  temp.append(plagCheck(fpr1,fpr2))
128              sim_mat.append(temp)
129
130      res_mat = np.array(sim_mat)
131      return res_mat, filenames
132
133
134
135
```

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 205 of file Final_Checker.py.

```
205 def RunCheck(infile, boilfile=None):
206
207     """This function takes in the path to input compressed files, and boilerplate code and invokes
208         extract_files() function to extract the files and then savres()
209     function to generate and save the results in the 'comparisons/results' folder """
210
211     formats=(".tar", ".tar.gz", ".zip")
212     if infile.endswith(formats):
213         try:
214             out_dir , files_dir = extract_files(infile)
215             res_dir= os.path.join(out_dir, 'results')
216             os.mkdir(res_dir)
217             if boilfile==None:
218                 saveres(files_dir, res_dir)
219             else:
220                 saveres(files_dir, res_dir, boilfile)
221             """returns 'success' and path to directory having generated results"""
222             return 'success' , res_dir
223         """ returns 'fail' in all other scenarios"""
224         except:
225             return 'fail' , ''
226
227     return 'fail', ''
228
229
230
231
232
233
```

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 136 of file Final_Checker.py.

```
136 def saveres(inpath, outpath, boilpath=None):
137     """inpath is path to directory containing code and boilpath is path to boilerplate code file.
138     This function basically calls folder_compare() function on the input directory and
139         saves the result in the form of csv to the output path(outpath),
140     It also generates a graphical respresentation of the result and savs it in the
141         outpath folder.
142     """
143
144     if boilpath==None:
145         matres, filenames=folder_compare(inpath)
146     else:
147         matres, filenames=folder_compare(inpath, boilpath)
148
149     extentt=np.arange(len(filenames)) + 0.5
150
151     """using pandas to generate dataframe and save it as csv from the similarity matrix"""
152
153     df = pd.DataFrame(matres, index= filenames, columns=filenames)
154
155     df.to_csv(os.path.join(outpath, 'results.csv'))
156
157
158     """using matplotlib to generate an image shwoing degree of plagiarism in a pair of file"""
159
160     fig, ax = plt.subplots(1,1)
161
162     img = ax.imshow(matres,cmap='Reds', vmin=0, vmax=1, extent=[0, len(filenames), 0, len(filenames)] )
163
164     ax.set_xticks(extentt)
165     ax.set_yticks(extentt)
166
167     ax.set_xticklabels(filenames, rotation= 60)
168     ax.set_yticklabels(filenames[::-1])
169
170     fig.colorbar(img)
171     plt.tight_layout()
172     plt.savefig(os.path.join(outpath, 'results.png'))
173
174
175
```