CN1

# Web Application Using Google Classroom API for ITS100 Management

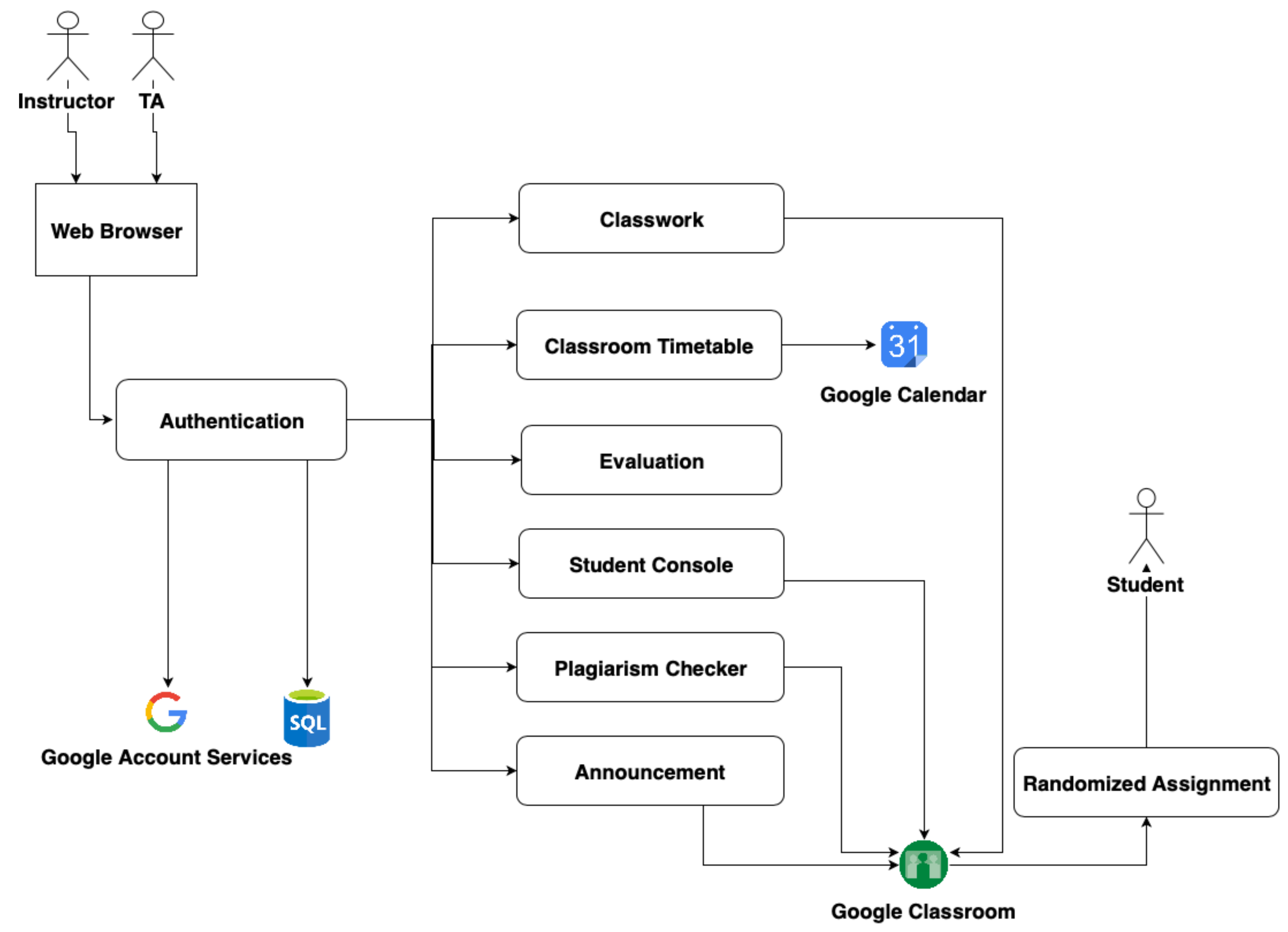
## Project Goal

Develop web application that consists of many Google APIs for instructors and teacher assistants to manage their student properly.

## Detailed Description

The aim of this project is to create a new user-friendly web application for ITS100 laboratory which facilitates instructor and teacher assistants to manage their students and classes. Moreover, we also aim to break the limitation of Google Classroom by creating the randomizing method which will random the set of classwork and distribute to students.

## System Architecture



## Technology and Tools

**Front End Framework:**
- Materialize framework

**Back End Framework :**
- Django framework

**Prototypes :**
- Adobe XD ,Draw.io

**Tools & APIs :**
- Pypi Similar, Google Cloud Platform Console (GCPC), Google Oauth 2.0, Google Classroom, Google Calendar , Google Drive, Google REST APIs

**Languages using** :
- Python, Javascript, CSS,HTML

**Databases :**
- SQL , SQLite3

## Benefits

- Reduce complication as users do not have to switch to each specific Google product.
- Web application satisfies the specific usage of ITS100 laboratory class.

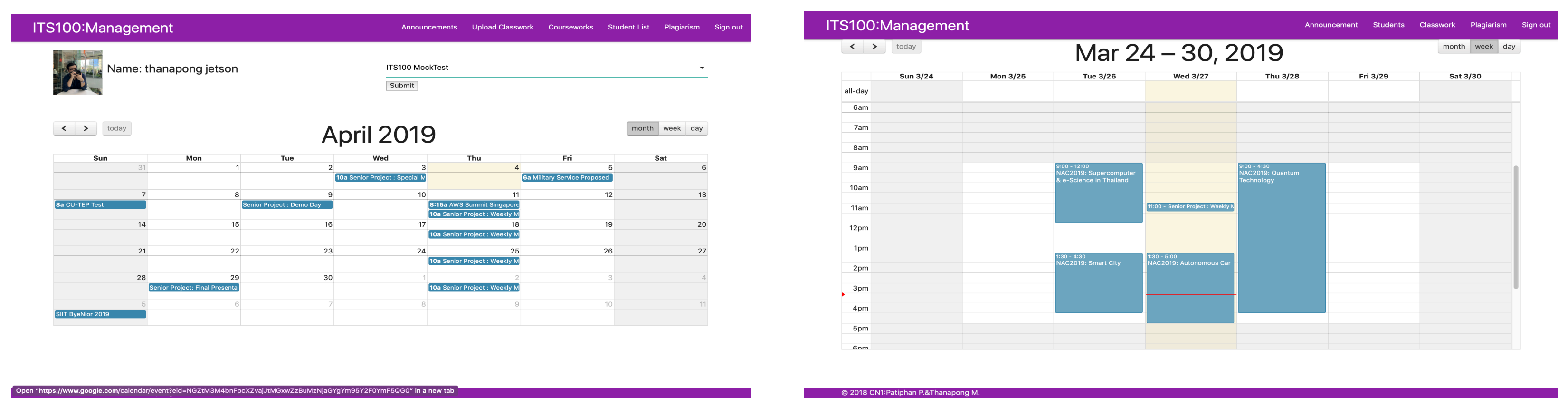
## Features and Functions

### 1. Authentication

Allow user to login to the system using their Gmail accounts to log in using Google Oauth2.0 in order to verify and allow permission to access user data.

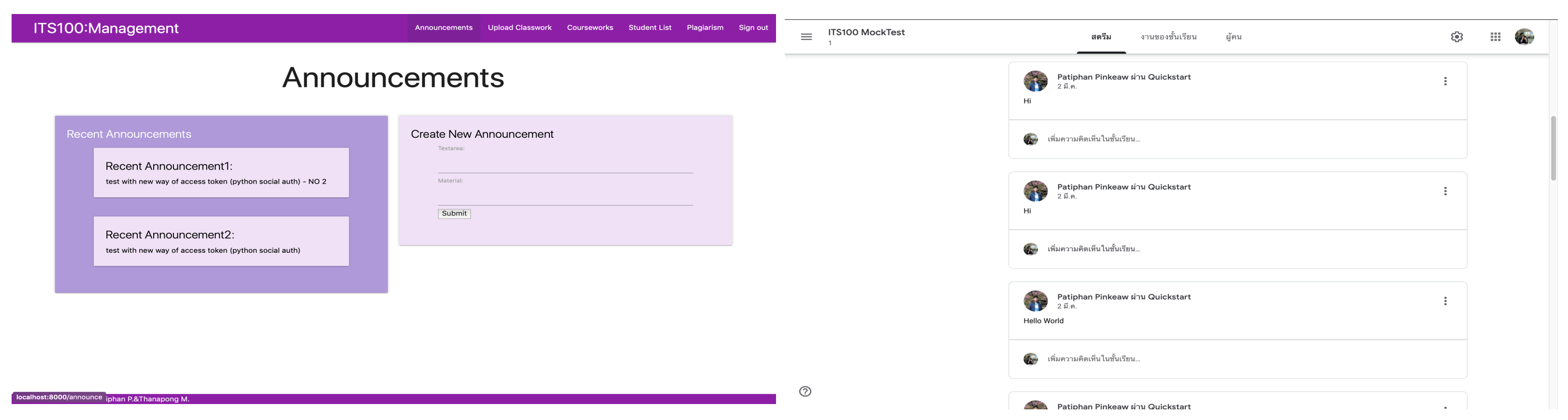
### 2. Classroom Timetable

The system displays the master classroom calendar form instructor in order to inform every staff in the classroom. To display we have to grant the access from Google Calendar.
In addition the user can change the calendar view in many aspects for example, monthly, weekly and daily.



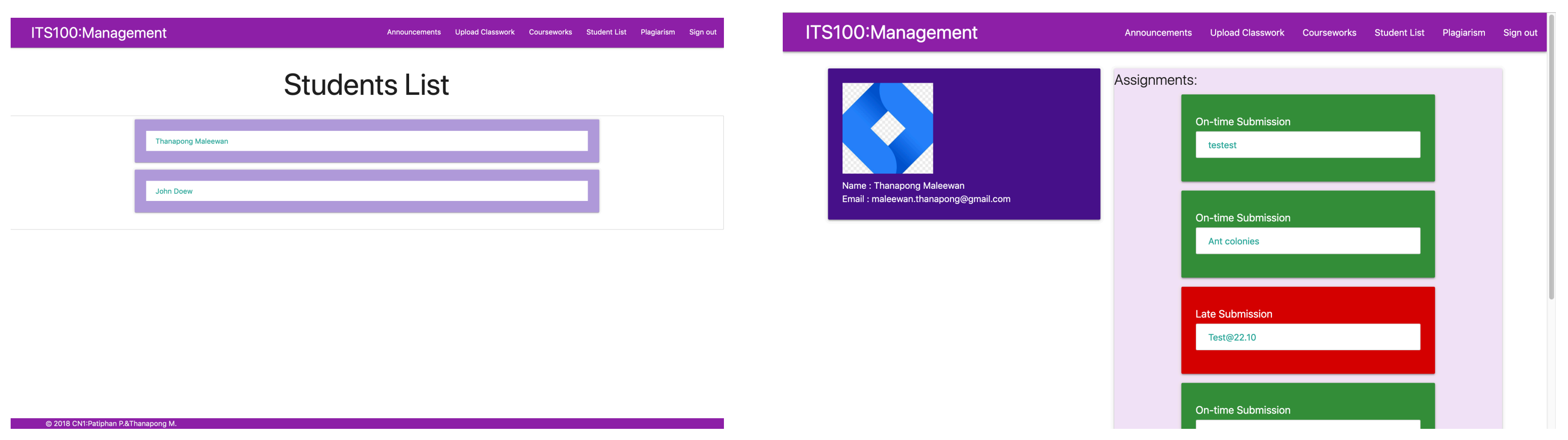
### 3. Announcements

Allow instructor to publish the announcement via the system after the announcement is published it also notify to all the member in the classroom. Using Google Classroom API and RESTful API to publish the announcement and display the recently 2 published announcements on the card below.



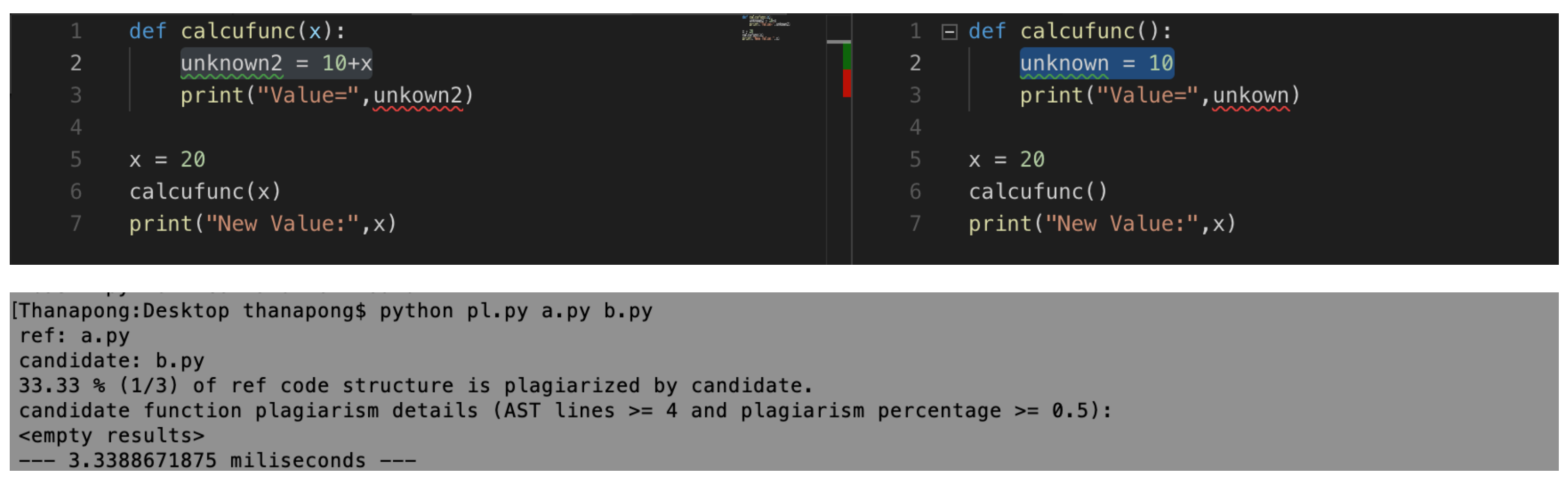
### 4. Student Console

Allow instructors and teacher assistants to see the list students in the classroom and also allow to see their assignment submissions by using Google Classroom API and RESTful API.



### 5. Plagiarism Checker

Allow instructors and teacher assistants to check the similarity between students assignments using Pycode-similar from PyPi to check and it will return the result as the percentages of thesimilarity and the method that check the similarity is normalize python AST representation and use difflib to get the modification from referenced code to candidate code.



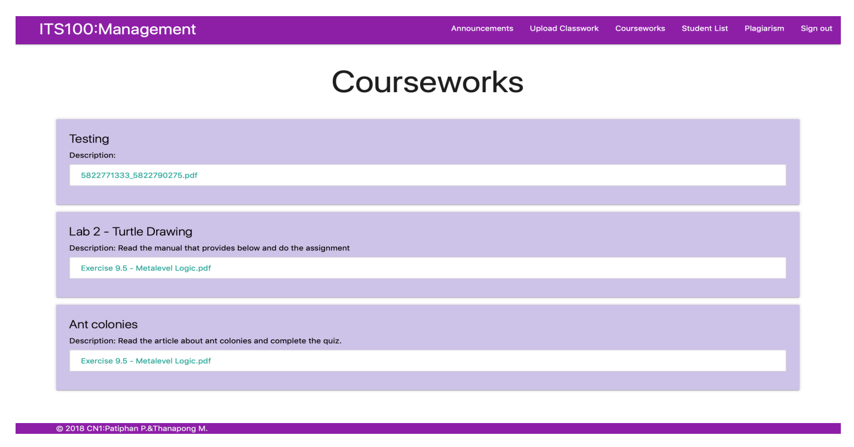
```python
def calcufunc(x):
    unknown2 = 10+x
    print("Value=",unkown2)

x = 20
calcufunc(x)
print("New Value:",x)
```

```python
def calcufunc():
    unknown = 10
    print("Value=",unkown)

x = 20
calcufunc()
print("New Value:",x)
```

```
[Thanapong:Desktop thanapong$ python pl.py a.py b.py
 ref: a.py
 candidate: b.py
 33.33 % (1/3) of ref code structure is plagiarized by candidate.
 candidate function plagiarism details (AST lines >= 4 and plagiarism percentage >= 0.5):
 <empty results>
 --- 3.3388671875 miliseconds ---
```

### 6. Classwork

Allow instructors to directly upload the weekly assignments on to the system.



### 7. Assignments

Random assignments and distribute to students.

### 8. Evaluating

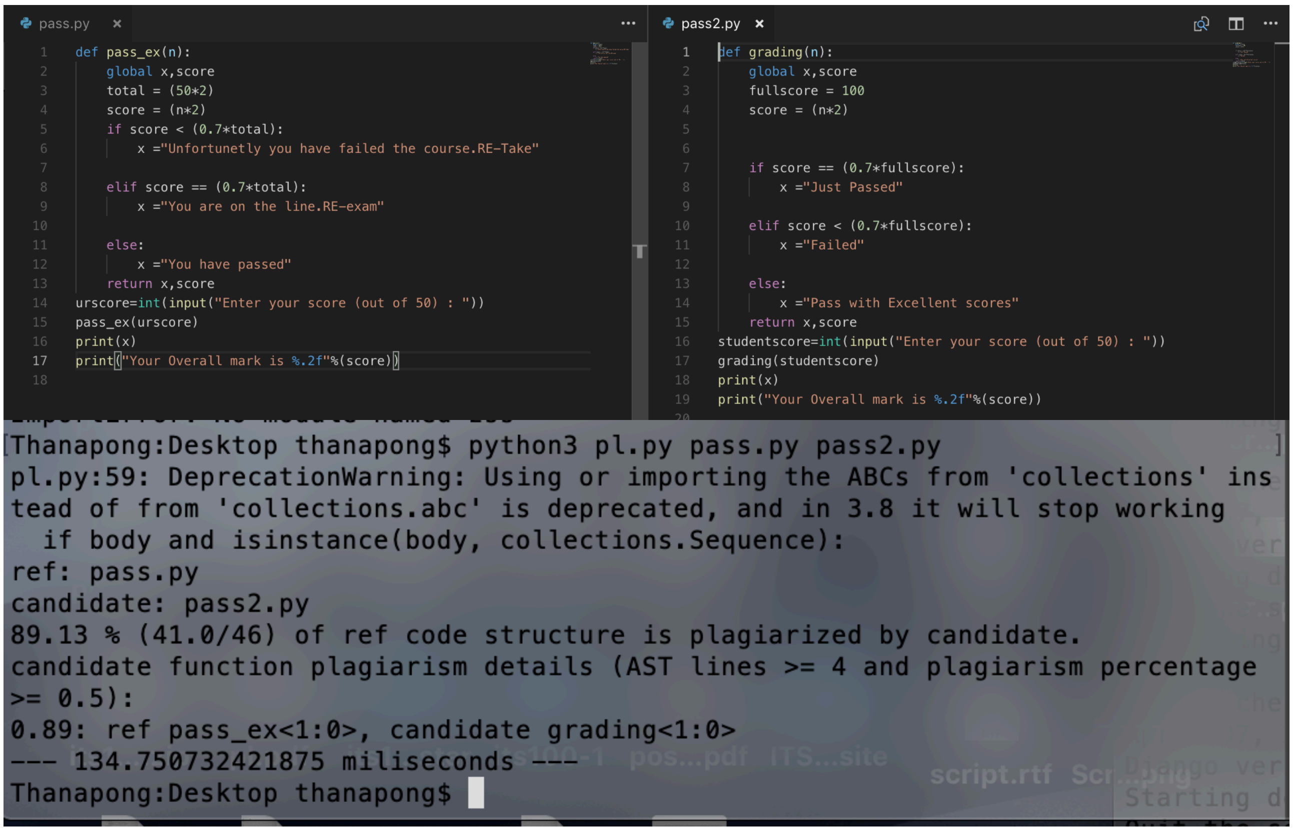
Allow instructors and teacher assistants to grade student assignment.

Members: Patiphan Pinkaew 582280185 Thanapong Maleewan 5822790275

```python
# -*- coding: utf-8 -*-
__author__ = 'fyrestone@outlook.com'
__version__ = '1.2'

import sys
import ast
import difflib
import operator
import argparse
import itertools
import collections
import time


class FuncNodeCollector(ast.NodeTransformer):
    """
    Clean node attributes, delete the attributes that are not helpful for
recognition repetition.
    Then collect all function nodes.
    """

    def __init__(self):
        super(FuncNodeCollector, self).__init__()
        self._curr_class_names = []
        self._func_nodes = []
        self._last_node_lineno = -1
        self._node_count = 0

    @staticmethod
    def _mark_docstring_sub_nodes(node):
        """
        Inspired by ast.get_docstring, mark all docstring sub nodes.

        Case1:
        regular docstring of function/class/module

        Case2:
        def foo(self):
            '''pure string expression'''
            for x in self.contents:
                '''pure string expression'''
                print x
            if self.abc:
                '''pure string expression'''
                pass

        Case3:
        def foo(self):
            if self.abc:
                print('ok')
            else:
                '''pure string expression'''
                pass

        :param node: every ast node
        :return:
        """

        def _mark_docstring_nodes(body):
            if body and isinstance(body, collections.Sequence):
                for n in body:
                    if isinstance(n, ast.Expr) and isinstance(n.value, ast.Str):
                        n.is_docstring = True

        node_body = getattr(node, 'body', None)
        _mark_docstring_nodes(node_body)
        node_orelse = getattr(node, 'orelse', None)
        _mark_docstring_nodes(node_orelse)

    @staticmethod
    def _is_docstring(node):
        return getattr(node, 'is_docstring', False)

    def generic_visit(self, node):
        self._node_count = self._node_count + 1
        self._last_node_lineno = max(
            getattr(node, 'lineno', -1), self._last_node_lineno)
        self._mark_docstring_sub_nodes(node)
        return super(FuncNodeCollector, self).generic_visit(node)

    def visit_Str(self, node):
        del node.s
        self.generic_visit(node)
        return node

    def visit_Expr(self, node):
        if not self._is_docstring(node):
            self.generic_visit(node)
            if hasattr(node, 'value'):
                return node

    def visit_arg(self, node):
        """
        remove arg name & annotation for python3
        :param node: ast.arg
        :return:
        """
        del node.arg
        del node.annotation
        self.generic_visit(node)
        return node

    def visit_Name(self, node):
        del node.id
        del node.ctx
        self.generic_visit(node)
        return node

    def visit_Attribute(self, node):
        del node.attr
        del node.ctx
        self.generic_visit(node)
        return node

    def visit_Call(self, node):
        func = getattr(node, 'func', None)
        if func and isinstance(func, ast.Name) and func.id == 'print':
            return  # remove print call and its sub nodes for python3
        return node

    def visit_ClassDef(self, node):
        self._curr_class_names.append(node.name)
        self.generic_visit(node)
        self._curr_class_names.pop()
        return node

    def visit_FunctionDef(self, node):
        node.name = '.'.join(itertools.chain(
            self._curr_class_names, [node.name]))
        self._func_nodes.append(node)
        count = self._node_count
        self.generic_visit(node)
        node.endlineno = self._last_node_lineno
        node.nsubnodes = self._node_count - count
        return node

    def visit_Compare(self, node):

        def _simple_nomalize(*ops_type_names):
            if node.ops and len(node.ops) == 1 and type(node.ops[0]).__name__ in
ops_type_names:
                if node.left and node.comparators and len(node.comparators) == 1:
                    left, right = node.left, node.comparators[0]
                    if type(left).__name__ > type(right).__name__:
                        left, right = right, left
                        node.left = left
                        node.comparators = [right]
                        return True
            return False

        if _simple_nomalize('Eq'):
            pass

        if _simple_nomalize('Gt', 'Lt'):
            node.ops = [{ast.Lt: ast.Gt, ast.Gt: ast.Lt}[type(node.ops[0])]()]

        if _simple_nomalize('GtE', 'LtE'):
            node.ops = [{ast.LtE: ast.GtE, ast.GtE: ast.LtE}
                        [type(node.ops[0])]()]

        self.generic_visit(node)
        return node

    def visit_Print(self, node):
        # remove print expr for python2
        pass

    def clear(self):
        self._func_nodes = []

    def get_function_nodes(self):
        return self._func_nodes


class FuncInfo(object):
    """
    Part of the astor library for Python AST manipulation.

    License: 3-clause BSD

    Copyright 2012 (c) Patrick Maupin
    Copyright 2013 (c) Berker Peksag

    """

    class NonExistent(object):
        pass

    def __init__(self, func_node, code_lines):
        assert isinstance(func_node, ast.FunctionDef)
        self._func_node = func_node
        self._code_lines = code_lines
        self._func_name = func_node.__dict__.pop('name', '')
        self._func_code = None
        self._func_code_lines = None
        self._func_ast = None
        self._func_ast_lines = None

    def __str__(self):
        return '<' + type(self).__name__ + ': ' + self.func_name + '>'

    @property
    def func_name(self):
        return self._func_name

    @property
    def func_node(self):
        return self._func_node

    @property
    def func_code(self):
        if self._func_code is None:
            self._func_code = ''.join(self.func_code_lines)
        return self._func_code

    @property
    def func_code_lines(self):
        if self._func_code_lines is None:
            self._func_code_lines = self._retrieve_func_code_lines(
                self._func_node, self._code_lines)
        return self._func_code_lines

    @property
    def func_ast(self):
        if self._func_ast is None:
            self._func_ast = self._dump(self._func_node)
        return self._func_ast

    @property
    def func_ast_lines(self):
        if self._func_ast_lines is None:
            self._func_ast_lines = self.func_ast.splitlines(True)
        return self._func_ast_lines

    @staticmethod
    def _retrieve_func_code_lines(func_node, code_lines):
        if not isinstance(func_node, ast.FunctionDef):
            return []
        if not isinstance(code_lines, collections.Sequence) or
isinstance(code_lines, basestring):
            return []
        if getattr(func_node, 'endlineno', -1) < getattr(func_node, 'lineno', 0):
            return []
        lines = code_lines[func_node.lineno - 1: func_node.endlineno]
        if lines:
            padding = lines[0][:-len(lines[0].lstrip())]
            stripped_lines = []
            for l in lines:
                if l.startswith(padding):
                    stripped_lines.append(l[len(padding):])
                else:
                    stripped_lines = []
                    break
            if stripped_lines:
                return stripped_lines
        return lines

    @staticmethod
    def _iter_node(node, name='', missing=NonExistent):
        """Iterates over an object:

           - If the object has a _fields attribute,
             it gets attributes in the order of this
             and returns name, value pairs.

           - Otherwise, if the object is a list instance,
             it returns name, value pairs for each item
             in the list, where the name is passed into
             this function (defaults to blank).

        """
        fields = getattr(node, '_fields', None)
        if fields is not None:
            for name in fields:
                value = getattr(node, name, missing)
                if value is not missing:
                    yield value, name
        elif isinstance(node, list):
            for value in node:
                yield value, name

    @staticmethod
    def _dump(node, name=None, initial_indent='', indentation='    ',
              maxline=120, maxmerged=80, special=ast.AST):
        """Dumps an AST or similar structure:

           - Pretty-prints with indentation
           - Doesn't print line/column/ctx info

        """

        def _inner_dump(node, name=None, indent=''):
            level = indent + indentation
            name = name and name + '=' or ''
            values = list(FuncInfo._iter_node(node))
            if isinstance(node, list):
                prefix, suffix = '%s[' % name, ']'
            elif values:
                prefix, suffix = '%s%s(' % (name, type(node).__name__), ')'
            elif isinstance(node, special):
                prefix, suffix = name + type(node).__name__, ''
            else:
                return '%s%s' % (name, repr(node))
            node = [_inner_dump(a, b, level) for a, b in values if b != 'ctx']
            oneline = '%s%s%s' % (prefix, ', '.join(node), suffix)
            if len(oneline) + len(indent) < maxline:
                return '%s' % oneline
            if node and len(prefix) + len(node[0]) < maxmerged:
                prefix = '%s%s,' % (prefix, node.pop(0))
            node = (',\n%s' % level).join(node).lstrip()
            return '%s\n%s%s%s' % (prefix, level, node, suffix)

        return _inner_dump(node, name, initial_indent)


class ArgParser(argparse.ArgumentParser):
    """
    A simple ArgumentParser to print help when got error.
    """

    def error(self, message):
        self.print_help()
        from gettext import gettext as _

        self.exit(2, _('\n%s: error: %s\n') % (self.prog, message))


class FuncDiffInfo(object):
    """
    An object stores the result of candidate python code compared to referenced
python code.
    """

    info_ref = None
    info_candidate = None
    plagiarism_count = 0
    total_count = 0

    @property
    def plagiarism_percent(self):
        return 0 if self.total_count == 0 else (self.plagiarism_count /
float(self.total_count))

    def __str__(self):
        if isinstance(self.info_ref, FuncInfo) and isinstance(self.info_candidate,
FuncInfo):
            return '{:<4.2}: ref {}, candidate {}'.format(self.plagiarism_percent,
                                                          self.info_ref.func_name +
'<' + str(

self.info_ref.func_node.lineno) + ':' + str(

self.info_ref.func_node.col_offset) + '>',
self.info_candidate.func_name + '<' + str(

self.info_candidate.func_node.lineno) + ':' + str(

self.info_candidate.func_node.col_offset) + '>')
        return '{:<4.2}: ref {}, candidate {}'.format(0, None, None)


class UnifiedDiff(object):
    """
    Line diff algorithm to formatted AST string lines, naive but efficiency, result
is good enough.
    """

    @staticmethod
    def diff(a, b):
        """
        Simpler and faster implementation of difflib.unified_diff.
        """
        assert a is not None
        assert b is not None
        a = a.func_ast_lines
        b = b.func_ast_lines

        def _gen():
            for group in difflib.SequenceMatcher(None, a,
b).get_grouped_opcodes(0):
                for tag, i1, i2, j1, j2 in group:
                    if tag == 'equal':
                        for line in a[i1:i2]:
                            yield ''
                        continue
                    if tag in ('replace', 'delete'):
                        for line in a[i1:i2]:
                            yield '-'
                    if tag in ('replace', 'insert'):
                        for line in b[j1:j2]:
                            yield '+'

        return collections.Counter(_gen())['-']

    @staticmethod
    def total(a, b):
        assert a is not None  # b may be None
        return len(a.func_ast_lines)


class TreeDiff(object):
    """
    Tree edit distance algorithm to AST, very slow and the result is not good for
small functions.
    """

    @staticmethod
    def diff(a, b):
        assert a is not None
        assert b is not None

        def _str_dist(i, j):
            return 0 if i == j else 1

        def _get_label(n):
            return type(n).__name__

        def _get_children(n):
            if not hasattr(n, 'children'):
                n.children = list(ast.iter_child_nodes(n))
            return n.children

        import zss
        res = zss.distance(a.func_node, b.func_node, _get_children,
                           lambda node: 0,  # insert cost
                           lambda node: _str_dist(
                               _get_label(node), ''),  # remove cost
                           lambda _a, _b: _str_dist(_get_label(_a),
_get_label(_b)), )  # update cost
        return res

    @staticmethod
    def total(a, b):
        #  The count of AST nodes in referenced function
        assert a is not None  # b may be None
        return a.func_node.nsubnodes


class NoFuncException(Exception):
    def __init__(self, source):
        super(NoFuncException, self).__init__(
            'Can not find any functions from code, index = {}'.format(source))
        self.source = source


def detect(pycode_string_list, diff_method=UnifiedDiff):
    if len(pycode_string_list) < 2:
        return []

    func_info_list = []
    for index, code_str in enumerate(pycode_string_list):
        root_node = ast.parse(code_str)
        collector = FuncNodeCollector()
        collector.visit(root_node)
        code_utf8_lines = code_str.splitlines(True)
        func_info = [FuncInfo(n, code_utf8_lines)
                     for n in collector.get_function_nodes()]
        func_info_list.append((index, func_info))

    ast_diff_result = []
    index_ref, func_info_ref = func_info_list[0]
    if len(func_info_ref) == 0:
        raise NoFuncException(index_ref)

    for index_candidate, func_info_candidate in func_info_list[1:]:
        func_ast_diff_list = []
        for fi1 in func_info_ref:
            min_diff_value = int((1 << 31) - 1)
            min_diff_func_info = None
            for fi2 in func_info_candidate:
                dv = diff_method.diff(fi1, fi2)
                if dv < min_diff_value:
                    min_diff_value = dv
                    min_diff_func_info = fi2
                if dv == 0:  # entire function structure is plagiarized by
candidate
                    break

            func_diff_info = FuncDiffInfo()
            func_diff_info.info_ref = fi1
            func_diff_info.info_candidate = min_diff_func_info
            func_diff_info.total_count = diff_method.total(
                fi1, min_diff_func_info)
            func_diff_info.plagiarism_count = func_diff_info.total_count - \
                min_diff_value if min_diff_func_info else 0
            func_ast_diff_list.append(func_diff_info)
        func_ast_diff_list.sort(key=operator.attrgetter(
            'plagiarism_percent'), reverse=True)
        ast_diff_result.append((index_candidate, func_ast_diff_list))

    return ast_diff_result


def _profile(fn):
    """
    A simple profile decorator
    :param fn: target function to be profiled
    :return: The wrapper function
    """
    import functools
    import cProfile

    @functools.wraps(fn)
    def _wrapper(*args, **kwargs):
        pr = cProfile.Profile()
        pr.enable()
        res = fn(*args, **kwargs)
        pr.disable()
        pr.print_stats('cumulative')
        return res

    return _wrapper


# @_profile
def main():
    """
    The console_scripts Entry Point in setup.py
    """

    def check_line_limit(value):
        ivalue = int(value)
        if ivalue < 0:
            raise argparse.ArgumentTypeError(
                "%s is an invalid line limit" % value)
        return ivalue

    def check_percentage_limit(value):
        ivalue = float(value)
        if ivalue < 0:
            raise argparse.ArgumentTypeError(
                "%s is an invalid percentage limit" % value)
        return ivalue

    def get_file(value):
        return open(value, 'rb')

    parser = ArgParser(
        description='A simple plagiarism detection tool for python code')
    parser.add_argument('files', type=get_file, nargs=2,
                        help='the input files')
    parser.add_argument('-l', type=check_line_limit, default=4,
                        help='if AST line of the function >= value then output
detail (default: 4)')
    parser.add_argument('-p', type=check_percentage_limit, default=0.5,
                        help='if plagiarism percentage of the function >= value
then output detail (default: 0.5)')
    args = parser.parse_args()
    pycode_list = [(f.name, f.read()) for f in args.files]
    try:
        results = detect([c[1] for c in pycode_list])
    except NoFuncException as ex:
        print('error: can not find functions from {}.'.format(
            pycode_list[ex.source][0]))
        return

    for index, func_ast_diff_list in results:
        print('ref: {}'.format(pycode_list[0][0]))
        print('candidate: {}'.format(pycode_list[index][0]))
        sum_total_count = sum(
            func_diff_info.total_count for func_diff_info in func_ast_diff_list)
        sum_plagiarism_count = sum(
            func_diff_info.plagiarism_count for func_diff_info in
func_ast_diff_list)
        print('{:.2f} % ({}/{}) of ref code structure is plagiarized by
candidate.'.format(
            sum_plagiarism_count / float(sum_total_count) * 100,
            sum_plagiarism_count,
            sum_total_count))
        print('candidate function plagiarism details (AST lines >= {} and
plagiarism percentage >= {}):'.format(
            args.l,
            args.p,
        ))
        output_count = 0
        for func_diff_info in func_ast_diff_list:
            if len(func_diff_info.info_ref.func_ast_lines) >= args.l and
func_diff_info.plagiarism_percent >= args.p:
                output_count = output_count + 1
                print(func_diff_info)
        if output_count == 0:
            print('<empty results>')


if __name__ == '__main__':
    start_time = time.time()*1000
    main()
    print("--- %s miliseconds ---" % (time.time()*1000 - start_time))
```

# PyCode Similar Result Samples

With variables and conditions modified.



pass.py

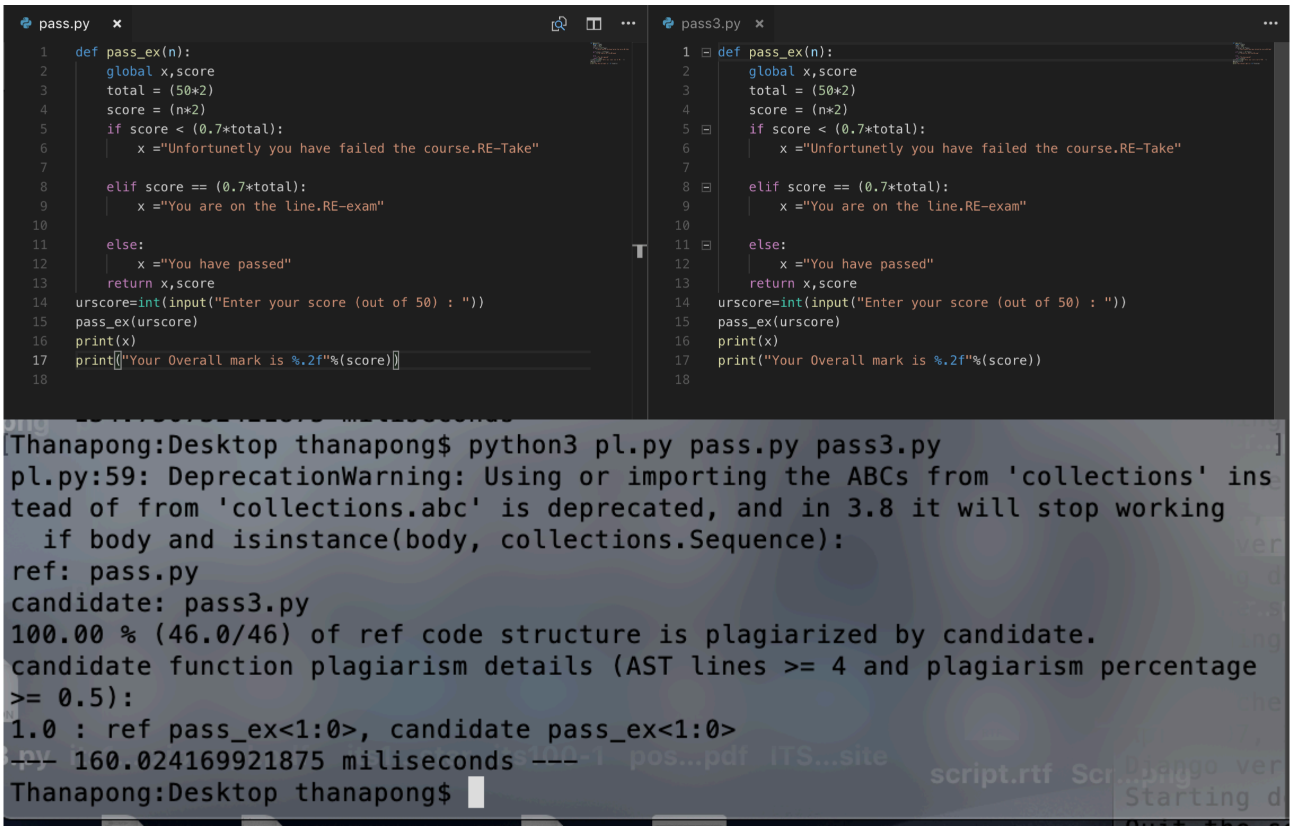
```python
def pass_ex(n):
    global x,score
    total = (50*2)
    score = (n*2)
    if score < (0.7*total):
        x ="Unfortunetly you have failed the course.RE-Take"

    elif score == (0.7*total):
        x ="You are on the line.RE-exam"

    else:
        x ="You have passed"
    return x,score
urscore=int(input("Enter your score (out of 50) : "))
pass_ex(urscore)
print(x)
print("Your Overall mark is %.2f"%(score))
```

pass2.py

```python
def grading(n):
    global x,score
    fullscore = 100
    score = (n*2)


    if score == (0.7*fullscore):
        x ="Just Passed"

    elif score < (0.7*fullscore):
        x ="Failed"

    else:
        x ="Pass with Excellent scores"
    return x,score
studentscore=int(input("Enter your score (out of 50) : "))
grading(studentscore)
print(x)
print("Your Overall mark is %.2f"%(score))
```

```
Thanapong:Desktop thanapong$ python3 pl.py pass.py pass2.py
pl.py:59: DeprecationWarning: Using or importing the ABCs from 'collections' ins
tead of from 'collections.abc' is deprecated, and in 3.8 it will stop working
  if body and isinstance(body, collections.Sequence):
ref: pass.py
candidate: pass2.py
89.13 % (41.0/46) of ref code structure is plagiarized by candidate.
candidate function plagiarism details (AST lines >= 4 and plagiarism percentage
>= 0.5):
0.89: ref pass_ex<1:0>, candidate grading<1:0>
--- 134.750732421875 miliseconds ---
Thanapong:Desktop thanapong$
```

No modification.



pass.py

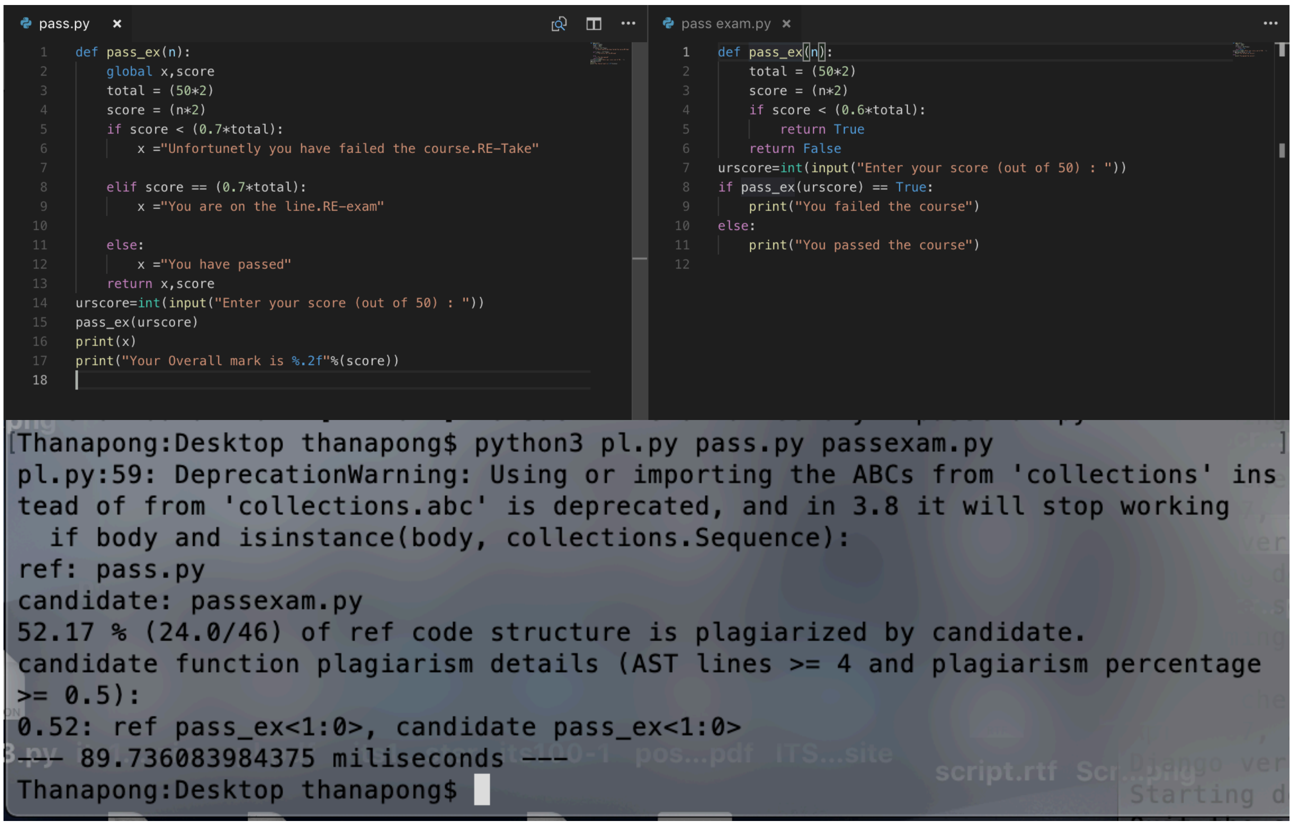
```python
def pass_ex(n):
    global x,score
    total = (50*2)
    score = (n*2)
    if score < (0.7*total):
        x ="Unfortunetly you have failed the course.RE-Take"

    elif score == (0.7*total):
        x ="You are on the line.RE-exam"

    else:
        x ="You have passed"
    return x,score
urscore=int(input("Enter your score (out of 50) : "))
pass_ex(urscore)
print(x)
print("Your Overall mark is %.2f"%(score))
```

pass3.py

```python
def pass_ex(n):
    global x,score
    total = (50*2)
    score = (n*2)
    if score < (0.7*total):
        x ="Unfortunetly you have failed the course.RE-Take"

    elif score == (0.7*total):
        x ="You are on the line.RE-exam"

    else:
        x ="You have passed"
    return x,score
urscore=int(input("Enter your score (out of 50) : "))
pass_ex(urscore)
print(x)
print("Your Overall mark is %.2f"%(score))
```

```
Thanapong:Desktop thanapong$ python3 pl.py pass.py pass3.py
pl.py:59: DeprecationWarning: Using or importing the ABCs from 'collections' ins
tead of from 'collections.abc' is deprecated, and in 3.8 it will stop working
  if body and isinstance(body, collections.Sequence):
ref: pass.py
candidate: pass3.py
100.00 % (46.0/46) of ref code structure is plagiarized by candidate.
candidate function plagiarism details (AST lines >= 4 and plagiarism percentage
>= 0.5):
1.0 : ref pass_ex<1:0>, candidate pass_ex<1:0>
--- 160.024169921875 miliseconds ---
Thanapong:Desktop thanapong$
```

With different code.



```python
def pass_ex(n):
    global x,score
    total = (50*2)
    score = (n*2)
    if score < (0.7*total):
        x ="Unfortunetly you have failed the course.RE-Take"

    elif score == (0.7*total):
        x ="You are on the line.RE-exam"

    else:
        x ="You have passed"
    return x,score
urscore=int(input("Enter your score (out of 50) : "))
pass_ex(urscore)
print(x)
print("Your Overall mark is %.2f"%(score))
```

```python
def pass_ex(n):
    total = (50*2)
    score = (n*2)
    if score < (0.6*total):
        return True
    return False
urscore=int(input("Enter your score (out of 50) : "))
if pass_ex(urscore) == True:
    print("You failed the course")
else:
    print("You passed the course")
```

```
Thanapong:Desktop thanapong$ python3 pl.py pass.py passexam.py
pl.py:59: DeprecationWarning: Using or importing the ABCs from 'collections' ins
tead of from 'collections.abc' is deprecated, and in 3.8 it will stop working
  if body and isinstance(body, collections.Sequence):
ref: pass.py
candidate: passexam.py
52.17 % (24.0/46) of ref code structure is plagiarized by candidate.
candidate function plagiarism details (AST lines >= 4 and plagiarism percentage
>= 0.5):
0.52: ref pass_ex<1:0>, candidate pass_ex<1:0>
--- 89.736083984375 miliseconds ---
Thanapong:Desktop thanapong$
```