**Computer Science 231   
Assignment #3**

[John Aycock](mailto:aycock@ucalgary.ca)

Due: 4pm Friday, 24 March 2017

**Purpose**

To use dictionaries, lists, and functions; to perform more robust error checking; to build abstraction skills; to construct useful tools; to work with program specifications.

**Important Notes**

* This is an **individual** assignment. What you submit must be your own work, and you **must** write the code yourself, although you may discuss the problem in general terms with other people. You should definitely **not** be showing other people your code, and generally speaking, it is not a good idea to talk about the assignment when you're sitting in front of the computer.
* Sources of algorithms and code, if any, must be properly cited. Remember that plagiarism regulations apply to code too. You can put citations into comments in your Python code.
* If you have any questions about what you can and can't safely do, feel free to [email me](mailto:aycock@ucalgary.ca).
* Per course policy, late assignments are **not** accepted.

**Every Workplace has Someone Like Carl**

Congratulations! You've been hired as a summer intern in Professor X's research lab. Carl, a postdoc in the lab, had just started working on a new project when there was an unexpected “accident”. Details are sketchy, but it appears to have involved a drunken bet, open flame, and an enraged moose. Needless to say, Carl won't be returning to work for a while.

This means that Carl's project is now your project. Before the Incident, Carl managed to write up two test programs, which you can find [here](http://pages.cpsc.ucalgary.ca/~aycock/231/as3/tokentest.py) and [here](http://pages.cpsc.ucalgary.ca/~aycock/231/as3/difftest.py). Using code from those two test programs, write a program that meets the specifications below.

* Your program is a tool to analyze Python programs using one of four methods. The method and the filename(s) of the Python input file(s) are specified on the command line when your program is run.
* If no method is specified on the command line, or an unknown method is specified, then a usage message should be printed.
* If the wrong number of filenames (too many or too few) are specified for a particular method, an appropriate error message should be printed.
* If there is a problem reading in a Python input file, an appropriate error message should be printed.
* Copy the relevant functions from Carl's test programs into your program, using comments to identify where they came from.
* You should use the functions in Carl's test programs in your program to implement the specified methods. You should not be writing code to read the input files yourself.
* You may assume valid Python source code is present in the input files.
* You program may exit immediately after detecting an error and printing an error message. You may find sys.exit() helpful for this.
* The tdiff method takes two Python input files and computes and prints their similarity score, only using the token values. COMMENT, NL, and NEWLINE tokens should be skipped.
* The adiff method takes two Python input files and computes and prints their similarity score, only using the attribute values. COMMENT, NL, and NEWLINE tokens' attributes should be skipped.
* The imports method takes one Python input file and prints out the modules that that file imports. Each module name should only be printed once. You only need to support the three variants of importstatement that you have seen used in lectures and  the course notes.
* Running untrusted Python code can have unwanted effects. The danger method takes one Python input file and prints out the frequency with which the following features occur in the input Python code:
  + import
  + exec
  + eval
  + open
  + Names starting with '\_\_' and ending with '\_\_'

Features that do not occur (i.e., those that have a frequency of zero) should not be printed.

**Sample Run**

This sample run refers to three Python input files: [input1.py](http://pages.cpsc.ucalgary.ca/~aycock/231/as3/input1.py), [input2.py](http://pages.cpsc.ucalgary.ca/~aycock/231/as3/input2.py), and [hello.py](http://pages.cpsc.ucalgary.ca/~aycock/231/as3/hello.py). Blank lines have been added in between the program executions for readability.

% python3 as3.py

Usage: python3 as3.py danger file.py

python3 as3.py tdiff file1.py file2.py

python3 as3.py imports file.py

python3 as3.py adiff file1.py file2.py

% python3 as3.py unknownmethod

Usage: python3 as3.py danger file.py

python3 as3.py tdiff file1.py file2.py

python3 as3.py imports file.py

python3 as3.py adiff file1.py file2.py

% python3 as3.py imports

Error: wrong number of filenames for imports

% python3 as3.py imports foo bar

Error: wrong number of filenames for imports

% python3 as3.py imports input1.py

sys

math

turtle

% python3 as3.py danger input1.py

eval x 1

\_\_builtins\_\_ x 1

exec x 1

import x 5

% python3 as3.py tdiff input1.py input2.py hello.py

Error: wrong number of filenames for tdiff

% python3 as3.py tdiff input1.py

Error: wrong number of filenames for tdiff

% python3 as3.py tdiff input1.py input2.py

0.6842105263157895 => files are similar

% python3 as3.py tdiff input1.py hello.py

0.14634146341463414 => files are not similar

% python3 as3.py adiff input1.py input2.py

0.618421052631579 => files are similar

% python3 as3.py adiff input1.py hello.py

0.0975609756097561 => files are not similar

% python3 as3.py adiff hello.py hello.py

1.0 => files are similar

**Bonus (+1 mark)**

Make the imports method print the correct module names for *all* the variants of the [Python import statement](https://docs.python.org/3/reference/simple_stmts.html#the-import-statement).

**“Got any tips?”**

Minor spoilers

*You are free to ignore these.*

* *Always* look at your data. Run the test programs to see what they do, and by extension, what the functions in each one do.
* Create your own Python input files as test cases for your program.
* You don't necessarily need to understand how the test programs are functioning. You can abstract that away and simply rely on their interface: what arguments you need to pass to the functions inside them, and what they return. In fact, the test programs use features that we not see in CPSC 231 until much later, and this is done on purpose, because it is not uncommon to have to work with code that works but that you may not (have time to) understand fully.
* If you find yourself writing similar code in two places, see if you can express that using a function, possibly with a parameter to distinguish the two cases if necessary. Doing this effectively can save you a *lot* of code.
* You may find some string functions that help you determine when a string starts with something or ends with something.
* Terminology: tokenization essentially is breaking Python programs into “words”. A *token* is a symbolic representation of a word's type, and its associated *attribute* is the actual text of the word. For example, all comments in Python code would have the token COMMENT, and the attribute of each will be the text of the comment.

**“I have no idea how to do this!”**

Major spoilers

*You are free to ignore these. There are many ways to do this assignment.*

* All minor spoilers still apply.
* You'll find the frequency code in the lecture notes.
* You'll find code in the lecture notes to ensure that strings are only printed once, no matter how many times they occur.
* Don't forget sys.argv. If you don't remember what it is and how to use it, type in this test program:
* import sys

print(sys.argv)

and run it in various ways with command-line arguments to Python to see how sys.argv changes. For example:

% python3 argvtest.py

% python3 argvtest.py foo

% python3 argvtest.py foo bar

* Each different method your program must support can be treated separately, so for the most part you can work on one at a time and abstract away the others in your head.

**“I've been programming since I was a fetus!”**

Making a better solution

What you're ideally creating here is a generic framework for analyzing Python code into which you can easily plug in new analyses to extend the framework. A good solution would allow you to incorporate new analyses with the minimum number of changes to the rest of the code.

* One way to accomplish this is to have a table (dictionary) that maps a method name to all the information associated with that method, including the number of arguments *and* the function that implements it. For the latter, if you use a function name in your code *without* the parentheses after it, it's a reference to that function. For example:
* def foo():
* print('foo')
* def bar():
* print('bar')
* baz = foo
* baz()
* baz = bar

baz()

If you get your table and the code using it set up correctly, adding a new analysis function will simply be a matter of adding the function plus adding exactly one line to your table. Everything else (including the usage information) draws its information from the table.

* Another way to accomplish this is to use reflection, which is mentioned in a recent participation and practice exercise. With this approach, there's no table, and adding a new analysis function is a matter of adding the function; that's it. You'll need to figure out how to extract/store the associated information that would have been in the table-driven version.
* You may find Python's extended call syntax, particularly the \* version, to be useful.

**General Hints**

* Make a directory to keep your files for this assignment in so you can keep them separate from things for other assignments and tutorials.
* Take notes as you go along, so you can remember what you've tried already and what did and didn't work.
* Break the task down into small pieces.
* Use an incremental approach, and test as often as possible!

**Assignment Submission**

To hand the assignment in, send *one* email to your TA containing:

* Your student ID.
* Your Python source code .py file. One only.

You are responsible for getting your assignment to your TA on time; keep in mind that email can take a few minutes to deliver! To be safe, you should CC yourself a copy of the email too.

**Evaluation**

You cannot be given a grade above zero if:

* You do not submit the required two items.
* There is no submission or the code doesn't run at all.
* You do not have a program in a .py file — you cannot turn in a solution that only uses the Python command line.
* Your code does not run on the CPSC machines using Python 3.
* You have different .py files for different methods – you must have one .py file handling *all* methods.
* You have different .py files for different input files – you must have one .py file handling *all* input files.

Your assignment will be given a grade as follows. Marks are only given if a feature is fully implemented as specified.

* 1/9 — tdiff method implemented correctly.
* 1/9 — adiff method implemented correctly.
* 1/9 — imports method implemented correctly.
* 1/9 — danger method implemented correctly.
* 1/9 — usage message printed for unknown or missing method name.
* 1/9 — error message printed if wrong number of filenames given.
* 1/9 — error message printed if tokenization failed for an input file.
* 2/9 — Code quality.

For the code quality mark, here's a rough, not necessarily all-inclusive guide to properties that would get the different marks.

* 0/2 — Appears to be one dense blob of code; no documentation; documentation does not agree with code; variable name choices don't reflect their usage; heavily overdocumented; documentation has extensive spelling/grammar errors; design of code is unclear; unnecessary duplicated code.
* 1/2 — Somewhere in between.
* 2/2 — Documentation well written and used where needed; variable name choices reflect usage; documentation and code agree; whitespace used effectively in code to convey logical groupings of code and make code aesthetically pleasing to read; design clear; no unnecessary duplicated code; functions used effectively where appropriate; could use as an example of excellence in lecture/tutorial.

*John Aycock, 08 March 2017*