PLEASE HAND IN

PLEASE HAND IN

UNIVERSITY OF TORONTO
Department of Computer and Mathematical Science

DECEMBER 2012 EXAMINATIONS

CSC A08 H3F
Instructor: Brian Harrington

Duration — 3 hours

Examination Aids: None

Student Number: |_|_|_|_|_|_|_|_|_|_|

Family Name(s): ______________________________

Given Name(s): ______________________________

---

*Do **not** turn this page until you have received the signal to start.*
In the meantime, please read the instructions below *carefully.*

# 1:	______/ 8
# 2:	______/ 4
# 3:	______/ 2
# 4:	______/ 4
# 5:	______/ 5
# 6:	______/10
# 7:	______/ 5
# 8:	______/ 9
# 9:	______/10
# 10:	______/10
# 11:	______/ 3
# 12:	______/ 2
TOTAL:	______/72

This final examination paper consists of 12 questions on 20 pages (including this one). *When you receive the signal to start, please make sure that your copy of the final examination is complete.*
Comments and docstrings are not required except where indicated, although they may help us mark your answers. They may also get you part marks if you can't figure out how to write the code.
You do not need to put `import` statements in your answers.
You may not use `break` or `continue` on this exam.
If you use any space for rough work, indicate clearly what you want marked.
Assume all input is valid unless otherwise indicated; there is no need to error-check.

Good Luck!

Question 1. [8 MARKS]

For each code fragment in the table below, indicate whether or not the code causes an error. If it runs without error, give the output. If it has an error, explain the cause of the error.

Code	**Error? (yes or no)**	**Output or Cause of Error**
`print(9 / 5)`		
`print(9 // 5)`		
`print(9 % 5)`		
`a = [1, 2, 3, 4, 5]` `print(a[3])`		
`b = ['hello', 'world']` `print(b[2])`		
`c = 'moogah'` `c[0] = 'b'` `print(c[0])`		
`d = {2:  ['sam', 'pat', 'carl'],` `0:  ['little', 'bo', 'peep'],` `1:  ['tina', 'jo']}` `print(d[0][-1][1:])`		
`e = ['A', 'B', 'C']` `e[0] = 8663` `print(e)`		

Question 2. [4 MARKS]

In the box below, write what is printed when the following program is executed.

```
def square(x):
    print('Squaring', x)
    return x * x

def add(y, z):
    print('Adding', y, z)
    return y + z

if add(-25, square(5)) > 0 and add(-4, square(10)) > 0:
    print('Woohoo!')
else:
    print('Rats', add(square(3), 0))
```

Question 3. [2 MARKS]

The following function does not always do what the docstring claims (although the type contract is correct).

```
def filter_positives(L):
    '''(list of int) -> list of int

    Return a new list containing the elements of L that are positive (greater
    than 0).
    '''

    acc = []
    for item in L:
        if item <= 0:
            return acc
        else:
            acc.append(item)

    return acc
```

In the table below, write the simplest possible test case that reveals the problem.

Test Case Description	L	Expected Return Value According to Docstring	Actual Return Value Based on Code

Question 4. [4 MARKS]

Consider the following code.

```
def my_function(x, y):
    x.append(5)
    y = y + [5]

x = [1, 2, 3]
x1 = x
x2 = x[:]

y = [1, 2, 3]
y1 = y
y2 = y[:]

my_function(x, y)
```

Once this code has been executed, what are the values that the variables now refer to?

Variable	**Value**
x	
x1	
x2	
y	
y1	
y2	

Question 5. [5 MARKS]

Some, but not all, of the lines containing a comment can be reached. All of these comment lines have this form, where `X` is a number: `# Condition X`

In the table at the bottom of this page, complete each row by specifying one set of inputs to the function that will cause the comment line to be reached. If there are no possible inputs that would allow a particular comment to be reached, indicate this with 'X's in the corresponding table row cells. If the value of a variable is not relevant, write 'ANY' in the corresponding table row cell. We have filled in the first two rows for you so that you can see what we want.

```
def mystery_code_paths(b1, b2, i, s):
    '''(bool, bool, int, str) -> NoneType'''

    if False:
        # Condition 0
    elif b1:
        # Condition 1

    if (b1 or b2) and i == 0:
        if i < 0:
            # Condition 2
        elif b1:
            # Condition 3
        elif not b1 and not b2:
            # Condition 4
        elif len(s) == i:
            # Condition 5
        else:
            # Condition 6
```

Condition	b1	b2	i	s
0	X	X	X	X
1	True	ANY	ANY	ANY
2				
3				
4				
5				
6				

Question 6. [10 MARKS]

Word ladder is a single-player game in which a player tries to link an initial word (e.g., *cold*) to a goal word (e.g., *warm*) by finding a chain of words to link the two. Words next to each other in the chain must differ by exactly one letter (adding or subtracting a letter does not count as a change). For example: *cold* → *cord* → *card* → *ward* → *warm*.

This question has you write a word ladder game. We first have you write two helper functions, and in the last part you will write a main program.

Part (a) [3 MARKS] Write the body for function `differ_by_one`.

```
def differ_by_one(word1, word2):
    ''' (str, str) -> bool

    Return True iff word2 can be formed from word1 by changing exactly one letter.

    >>> differ_by_one('cat', 'cot')
    True
    >>> differ_by_one('abc', 'aBc')
    True
    >>> differ_by_one('abc', 'abc')
    False
    >>> differ_by_one('abc', 'abcd')
    False
    '''
```

Part (b) [2 MARKS] Write the body for function `get_word_list`.

```
def get_word_list(word_file):
    ''' (io.TextIOWrapper) -> list of str

    word_file contains one word per line.  Return a list that contains
    each word from word_file with leading and trailing whitespace removed.
    '''
```

Part (c) [5 MARKS]

Complete the main program on the next page. Avoid duplicate code by calling function `differ_by_one` as a helper. You should do this even if your `differ_by_one` is incomplete/incorrect and the program will be marked as though it works correctly.

To play the game, the player repeatedly guesses the next word in the chain. If the word entered is not valid (it does not differ from the current word by exactly one letter or it does not appear in the file of words), the player is prompted to re-enter a word until a valid word is entered. Once a valid word is entered, if it is the goal word, the game ends. If it is not the goal word, the player is prompted to make another move using the word they just formed as the current word. The player continues to make moves until the goal word is formed. The object of the game is to form the goal word in as few moves as possible, so the number of moves is tracked (only valid moves are counted).

```
print("Welcome to word ladder!")

initial = 'cold'
current = initial
goal = 'warm'

# one possible solution is cold -> cord -> card -> ward -> warm

num_moves = 0
word_file = open('words.txt')
word_list = get_word_list(word_file)
word_file.close()

while current != goal:

    print("Current: ", current, ", Goal: ", goal)
    next_move = input("Please enter the next word in the chain: ")
```

```
print("You solved the word ladder in", num_moves, "moves!")
```

Question 7. [5 MARKS]

Consider a dictionary where each key is a 3-element tuple containing the name of a sports team, its city, and its sport, and each value is the number of wins for that team. For example:

```
d = {('Raptors', 'Toronto', 'basketball'): 10,
     ('Blues', 'Toronto', 'basketball'): 21,
     ('Senators', 'Ottawa', 'hockey'): 45,
     ('Leafs', 'Toronto', 'hockey'): 3}
```

The header for function `get_wins_by_category` appears below. It has two parameters:

- the first parameter, `team_to_wins`, is a dictionary of the form described above, and
- the second parameter, `category`, is a string: one of `'name'`, `'city'`, or `'sport'`.

`get_wins_by_category` returns a new dictionary based on `team_to_wins` in which the keys are either the names, cities or sports (as specified by `category`) and the values are the total number of wins associated with each key. Here is a sample shell interaction that uses variable `d`, initialized above.

```
>>> get_wins_by_category(d, 'city')
{'Toronto': 34, 'Ottawa': 45}
```

Complete function `get_wins_by_category`. Use good style: avoid duplicate code.

```
def get_wins_by_category(team_to_wins, category):
    '''(dict of {tuple of (str, str, str): int}, str) -> dict of {str: int}'''
```

Question 8. [9 MARKS]

Function `remove_dup_values` takes a dictionary `d` of `{int: int}` as input, removes from that dictionary all the entries whose values are not unique and returns the number of entries that were removed. In other words, whenever two (or more) entries in `d` have the same value, then all those entries are removed from `d`.

Part (a) [4 MARKS]

In the table below, we have outlined one test case for `remove_dup_values`. Add four more test cases chosen to test the function as thoroughly as possible.

Test Case Description	Dictionary Before Call	Dictionary After Call	Return Value
empty dictionary	`{}`	`{}`	`0`
single item dictionary	`{1: 4}`	`{1: 4}`	`0`

Part (b) [5 MARKS] Write the function body.

```
def remove_dup_values(d):
    ''' (dict of {int: int}) -> int
    Remove all entries of d whose values are not unique in d. Return the
    number of entries that were removed.
    '''
```

Question 9. [10 MARKS]

The *median* value of a list of numbers is a number where half the values from the list are larger and half are smaller. If the list were sorted, the median value is:

1. The middle value (if there are an odd number of values), or
2. The average of the two middle values (if there are an even number of values).

Part (a) [5 MARKS]

Complete the following function. You *must* use this algorithm: call `sort` on the list and then return the middle value (or the average of the two middle values if the list has even length).

```
def median1(L):
    '''(list of number) -> number

    Return the median of the numbers in L.

    Precondition: len(L) >= 1

    >>> median1([4, 2, 1, 5, 7])
    4
    >>> median1([4, 2, 1, 5])
    3.5
    '''
```

Part (b) [5 MARKS]

Complete the following function. You *must* use this algorithm: continually remove the largest and smallest values from the list and return either the last value (if there is only one left) or the average of the two last values (if there are two left).

Do not sort the list, and you must not use any `for` loops.

Functions `max` and `min` may be helpful, as well as one or more `list` methods.

```
def median2(L):
    '''(list of number) -> number

    Return the median of the numbers in L.

    Precondition: len(L) >= 1

    >>> median2([4, 3, 1, 5, 7])
    4
    >>> median2([4, 3, 1, 5])
    3.5
    '''
```

Question 10. [10 MARKS]

Some transit riders pay for their trips with pre-loaded electronic cards that they tap as they enter the subway. They can reload their cards by specifying an amount of money to be added. Each card is also programmed with a default dollar amount that gets reloaded when the rider does not specify an amount.

Complete the `Card` class on this page and the next to match the comments. You may not change any of the provided code and your code must work correctly with it. Complete the main block, using the methods from the `Card` class. (Do not access the instance variables directly.) You must provide completed DocStrings for any methods you write.
Read all of starter code and comments on this page and the next before writing any code.

```
class Card(object):
    ''' A pre-paid subway card. '''
    # Write the constructor (__init__) method.
```

```
    def __str__(self):
        ''' (Card) -> str
        Return a string representation of this Card.
        '''
        return "Owner: {0} \tBalance: {1}".format(self.owner, self.balance)

    def reload_default(self):
        ''' (Card) -> float
        Add the default dollar reload amount to this card's balance and
        return the new balance.
        '''
        return self.reload(self.default_reload)

    # Write the reload method.
    #  Write the take_trip method, which has a parameter indicating the trip fare.
    #  If the card has enough money to cover the trip, the fare is deducted.
    #  If the card does not have enough money to cover the trip, the balance is
    #  unchanged. Return True iff the card had enough money to cover the trip.
```

```
if __name__ == '__main__':
    ## Create a card for Karen Stintz with an initial balance of $100 and a
    ## default reload value of $50.
    card_for_karen = Card("Karen Stintz", 100, 50)
    ## Have Karen Stintz take 25 regular trips at a fare of $2.50 each.




    ## Create a card for Rob Ford with an initial balance of $50 and a
    ## default reload value of $20.


    ## Have Rob Ford take one rush-hour trip at a fare of $3.


    ## Reload Karen Stintz's card with her default reload value.


    ## Have Rob Ford put an additional $5.75 on his card.


    ## Print out the owner and balance of both cards.
```

Question 11. [3 MARKS]

A *matrix* is a rectangular array of numbers. A *square matrix* is a matrix where the number of rows and columns is the same. The *diagonal* of a square matrix goes from the upper left element to the bottom right element. Here is an example of a 4 by 4 square matrix where the elements on the diagonal are in bold:

$$\begin{array}{cccc} \mathbf{1} & 2 & 7 & 4 \\ 3 & \mathbf{9} & 1 & 0 \\ 4 & 3 & \mathbf{8} & 7 \\ 2 & 4 & 0 & \mathbf{5} \end{array}$$

Relative to the number of rows of parameter `matrix`, this function has **quadratic** ($O(n^2)$) running time:

```
def diagonal(matrix):
    '''(list of list of int) -> (list of int)

    Return a list of the diagonal elements of matrix.

    Precondition: The matrix is square: number of rows and columns is the same.

    >>> diagonal([[1, 2, 7, 4], [3, 9, 1, 0], [4, 3, 8, 7], [2, 4, 0, 5]])
    [1, 9, 8, 5]
    '''

    size = len(matrix)
    result = []
    for row in range(size):
        for col in range(size):
            if row == col:
                result.append(matrix[row][col])

    return result
```

Rewrite the body of the function so that it has the same functionality, but **linear** ($O(n)$) running time. Do not recopy the header or the docstring.

Question 12. [2 MARKS]

Consider this code:

```
def g(L, y):
    ''' (list, object) -> int
    '''
    for i in range(len(L)):
        if L[i] == y:
            return i
    return -1
```

For each question below, a correct answer earns 1 mark, a blank answer earns 0 marks, and an incorrect answer earns -0.5 marks. **Do not guess.**

Part (a) [1 MARK] Let n be the length of the list `L`. In the **best case**, which of the following most accurately describes how the runtime of function `g` grows as n grows? Circle one.

(a) It grows linearly, like n does. (b) It grows quadratically, like n^2 does.

(c) It grows more than quadratically. (d) It does not grow with respect to n.

Part (b) [1 MARK] Let n be the length of the list `L`. In the **worst case**, which of the following most accurately describes how the runtime of function `g` grows as n grows? Circle one.

(a) It grows linearly, like n does. (b) It grows quadratically, like n^2 does.

(c) It grows more than quadratically. (d) It does not grow.

Total Marks = 72

[Use the space below for rough work. This page will ***not*** *be marked, unless you clearly indicate the part of your work that you want us to mark.]*

Short Python function/method descriptions:

```
__builtins__:
  input([prompt]) -> str
    Read a string from standard input. The trailing newline is stripped. The prompt string,
    if given, is printed without a trailing newline before reading.
  len(x) -> int
    Return the length of the list, tuple, dict, or string x.
  max(iterable) -> object
  max(a, b, c, ...) -> object
    With a single iterable argument, return its largest item.
    With two or more arguments, return the largest argument.
  min(iterable) -> object
  min(a, b, c, ...) -> object
      With a single iterable argument, return its smallest item.
      With two or more arguments, return the smallest argument.
  print(value, ..., sep=' ', end='\n') -> NoneType
    Prints the values. Optional keyword arguments:
    sep:  string inserted between values, default a space.
    end:  string appended after the last value, default a newline.
  open(name[, mode]) -> io.TextIOWrapper
    Open a file.  Legal modes are "r" (read), "w" (write), and "a" (append).
  range([start], stop, [step]) -> list-like-object of int
    Return the integers starting with start and ending with
    stop - 1 with step specifying the amount to increment (or decrement).
    If start is not specified, the list starts at 0.  If step is not specified,
    the values are incremented by 1.

dict:
  D[k] --> object
    Return the value associated with the key k in D.
  del D[k]
    Remove D[k] from D.
  k in d --> bool
    Return True if k is a key in D and False otherwise.
  D.get(k) -> object
    Return D[k] if k in D, otherwise return None.
  D.keys() -> list-like-object of object
    Return the keys of D.
  D.values() -> list-like-object of object
    Return the values associated with the keys of D.
  D.items() -> list-like-object of tuple of (object, object)
    Return the (key, value) pairs of D, as 2-tuples.

io.TextIOWrapper:
  F.close() -> NoneType
    Close the file.
  F.read() -> str
    Read until EOF (End Of File) is reached, and return as a string.
  F.readline() -> str
    Read and return the next line from the file, as a string. Retain newline.
    Return an empty string at EOF (End Of File).
  F.readlines() -> list of str
    Call readline() repeatedly and return a list of the lines so read until
    EOF (End Of File) is reached.
```

```
list:
  x in L --> bool
    Return True if x is in L and False otherwise.
  L.append(x) -> NoneType
    Append x to the end of the list L.
  L.index(value) -> int
    Return the lowest index of value in L.
  L.insert(index, x) -> NoneType
    Insert x at position index.
  L.pop() -> object
    Remove and return the last item from L.
  L.remove(value) -> NoneType
    Remove the first occurrence of value from L.
  L.reverse() -> NoneType
    Reverse *IN PLACE*.
  L.sort() -> NoneType
    Sort the list in ascending order.

str:
  x in s --> bool
    Return True if x is in s and False otherwise.
  str(x) -> str
    Convert an object into its string representation, if possible.
  S.count(sub[, start[, end]]) -> int
    Return the number of non-overlapping occurrences of substring sub in
    string S[start:end].  Optional arguments start and end are interpreted
    as in slice notation.
  S.find(sub[, i]) -> int
    Return the lowest index in S (starting at S[i], if i is given) where the
    string sub is found or -1 if sub does not occur in S.
  S.index(sub) -> int
    Like find but raises an exception if sub does not occur in S.
  S.isdigit() -> bool
    Return True if all characters in S are digits and False otherwise.
  S.lower() -> str
    Return a copy of the string S converted to lowercase.
  S.lstrip([chars]) -> str
    Return a copy of the string S with leading whitespace removed.
    If chars is given and not None, remove characters in chars instead.
  S.replace(old, new) -> str
    Return a copy of string S with all occurrences of the string old replaced
    with the string new.
  S.rstrip([chars]) -> str
    Return a copy of the string S with trailing whitespace removed.
    If chars is given and not None, remove characters in chars instead.
  S.split([sep]) -> list of str
    Return a list of the words in S, using string sep as the separator and
    any whitespace string if sep is not specified.
  S.strip() -> str
    Return a copy of S with leading and trailing whitespace removed.
  S.upper() -> str
    Return a copy of the string S converted to uppercase.
```