PLEASE HAND IN

PLEASE HAND IN

UNIVERSITY OF TORONTO
Faculty of Arts and Science

DECEMBER 2016 EXAMINATIONS

CSC 108 H1F
Instructor(s): Smith, Gries, de Lara

Duration—3 hours

No Aids Allowed

You must earn at least 30 out of 75 marks (40%) on this final examination in order to pass the course. Otherwise, your final course grade will be no higher than 47%.

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

Last (Family) Name(s): _____

First (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*.

This Final Examination paper consists of 10 questions on 20 pages (including this one), printed on both sides of the paper. *When you receive the signal to start, please make sure that your copy of the paper is complete and fill in your name and student number above.*

- Comments and docstrings are not required except where indicated, although they may help us mark your answers.
- You do not need to put `import` statements in your answers.
- No error checking is required: assume all user input and all argument values are valid.
- If you use any space for rough work, indicate clearly what you want marked.
- Do not remove pages or take the exam apart.

Marking Guide

1: _____/ 5
2: _____/ 9
3: _____/10
4: _____/ 6
5: _____/10
6: _____/ 8
7: _____/ 6
8: _____/ 8
9: _____/ 5
10: _____/ 8

TOTAL: _____/75

Good Luck!

over. . .

Question 1. [5 MARKS]

Part (a) [4 MARKS]

Complete the function body below according to its docstring description.

```
def zigzagzip(s1, s2):
    """ (str, str) -> str

    Precondition: len(s1) == len(s2).

    Return a string made up of alternating letters from s1 and s2,
    starting with s1[0], then s2[1], s1[2], and so on.

    >>> zigzagzip('abc', '123')
    'a2c'
    >>> zigzagzip('abcd', '1234')
    'a2c4'
    """
```

Part (b) [1 MARK]

The docstring for `zigzagzip` includes a precondition that `s1` and `s2` will have the same length. Briefly explain the error that could occur and cause the code to crash if `zigzagzip` was called with arguments that do ***NOT*** meet the precondition.

Question 2. [9 MARKS]

Part (a) [5 MARKS] Fill in the boxes to complete the type contract and docstring examples for the function below. You do not need to write a description.

```
def func(words):
    """ (list of str, int) ->

    >>> v1 = ['panda', 'cat', 'ox', 'beluga', 'dogs']
    >>> func(v1)
    >>> v1

    >>> v2 =

    >>> func(v2)
    >>> v2
    ['pie', 'cake', 'coconut', 'waffle', 'pizza']
    """
    i = 0
    while i < (len(words) - 1) and len(words[i]) > len(words[i + 1]):
        temp = words[i]
        words[i] = words[i + 1]
        words[i + 1] = temp
        i = i + 1
```

Part (b) [1 MARK] Briefly explain what error could happen if we changed the loop condition in `func` from `i < (len(words) - 1)` to `i < len(words)`?

Part (c) [1 MARK]

Write a formula in terms of k, where k is `len(words)`, that describes the *maximum* number of comparisons made on a call to `func`.

Part (d) [1 MARK]

Write a formula in terms of k, where k is `len(words)`, that describes the *minimum* number of comparisons made on a call to `func`.

Part (e) [1 MARK] Which of the following best describes the *worst case* running time of `func`? Circle your answer.

constant linear quadratic something else

Question 3. [10 MARKS]

Part (a) [8 MARKS] Complete the body of this function according to its docstring description.

```
def get_positions(text):
    """ (str) -> dict of {str: list of int}

    Return a dictionary where the keys are the individual words in text, and
    the values are the positions in the text where the words occur (starting
    at 1). Include punctuation and numbers in words, and convert alphabetic
    characters to lowercase.

    >>> result = get_positions('cats Cats CATS CATS!!!')
    >>> result == {'cats': [1, 2, 3], 'cats!!!': [4]}
    True
    >>> result = get_positions('I think I like CSC108.')
    >>> result == {'i': [1, 3], 'think': [2], 'like': [4], 'csc108.': [5]}
    True
    """
```

For the next two parts, consider the following function that operates on the return value of `get_positions`. (Full docstring omitted for space.)

```
def has_single_word(d):
    """ (dict of {str: list of int}) -> bool """
    for key in d:
        if len(d[key]) == 1:
            return True
    return False
```

Part (b) [1 MARK]

Write a formula in terms of k, where k is `len(d)`, to express the *minimum* number of comparisons made when the `has_single_word` function is called.

Part (c) [1 MARK]

Write a formula in terms of k, where k is `len(d)`, to express the *maximum* number of comparisons made when the `has_single_word` function is called.

Question 4. [6 MARKS]

Complete the function body below according to its docstring description.

Hint: the `int` function and `%` (mod) operator will be helpful here.

```
>>> int(2.9)
2
>>> 2.5 % 1
0.5
```

```
def truncate_and_accumulate(values):
    """ (list of number) -> float

    Modify values so that each number is rounded down to the nearest integer
    value, and return the accumulated lost amount.

    >>> values = [1, 2.5, 3.3, 4.01]
    >>> truncate_and_accumulate(values)
    0.81
    >>> values
    [1, 2, 3, 4]
    >>> values = [0.25, 0.5, 0, 1, 33]
    >>> truncate_and_accumulate(values)
    0.75
    >>> values
    [0, 0, 0, 1, 33]
    >>> values = [10, 15, 20]
    >>> truncate_and_accumulate(values)
    0.0
    >>> values
    [10, 15, 20]
    """
```

Question 5. [10 MARKS]

Part (a) [5 MARKS] Recall the algorithms insertion sort, selection sort, and bubble sort.

Consider the following lists. For each list, and each algorithm, consider whether it is possible that two passes of the algorithm could have been completed on the list. Check the box corresponding to the algorithm if the list is a possible result of two passes of the algorithm.

	Insertion?	**Selection?**	**Bubble?**
`[1, 2, 3, 3, 7, 8]`			
`[2, 1, 3, 4, 5, 6]`			
`[3, 4, 7, 1, 2, 8]`			
`[1, 2, 4, 5, 3, 6]`			
`[1, 3, 5, 4, 5, 6]`			

Part (b) [5 MARKS]

Fill in the blanks to complete the function below according to its docstring description.

```
def bubble_sort(words):
    """ (dict of {str: list of int}) -> list of str

    Precondition: key.isalpha() == True for every key in words

    Return a list containing the keys in words sorted in ascending order by the
    length of their associated lists. In a tie, sort the keys in alphabetical order.

    >>> d = {'apple': [4, 9, 10], 'banana': [7], 'melon': [1, 3, 5, 6], \
    'kiwi': [2, 8]}
    >>> bubble_sort(d)
    ['banana', 'kiwi', 'apple', 'melon']
    >>> d = {'cat': [2, 5], 'dog': [1], 'ant': [3, 4]}
    >>> bubble_sort(d)
    ['dog', 'ant', 'cat']
    """

    L = list(words.keys())
    end = [                ]

    while end != 0:
        for i in range(end):
            if len(words[L[i]]) [    ] len(words[L[i + 1]]):
                L[i], L[i + 1] = L[i + 1], L[i]
            elif [                                ] and [                                ]:
                L[i], L[i + 1] = L[i + 1], L[i]
        end = [                ]
    return L
```

Question 6. [8 MARKS]

Complete the body of the function below according to its docstring.

```
def find_population(continent_info):
    """ (dict of {str: dict of {str: dict of {str: int}}}) -> dict of {str: int}

    Precondition: continent_info has keys representing continents, and the
    values are dictionaries where the keys represent countries on that
    continent and the values are dictionaries where the keys represent cities
    in that country and the values represent city populations.

    Return a dictionary where the keys are continents from continent_info
    and the values are the total population of all cities on that continent.

    >>> data = {'Europe': {'France': {'Paris': 100, 'Nice': 50, 'Lyon': 4}, \
    'Bulgaria': {'Sofia': 3000}}}
    >>> result = find_population(data)
    >>> result == {'Europe': 3154}
    True
    >>> data = { \
    'North America': {'Canada': {'Toronto': 5000, 'Ottawa': 200}, \
        'USA': {'Portland': 400, 'New York': 5000, 'Chicago': 1000}, \
        'Mexico': {'Mexico City': 10000}}, \
    'Asia': {'Thailand': {'Bangkok': 456}, \
        'Japan': {'Tokyo': 10000, 'Osaka': 5000}}, \
    'Antarctica': {}}
    >>> result = find_population(data)
    >>> result == {'North America': 21600, 'Asia': 15456, 'Antarctica': 0}
    True
    """
```

Question 7. [6 MARKS]

The code below implementing the `find_occurrences` function has some bugs in it. For this question, you will write two corrected versions of the `find_occurrences` function.

In each version, make whatever changes are necessary to the original body of the function `find_occurrences` so that it correctly matches its docstring. Write any lines that need to be modified in the **Modified Function Body** table. If a specific line requires no modification, you can leave its corresponding row blank in the **Modified Function Body** table. Do not add extra lines to the function body.

Part (a) [3 MARKS] In this part, correct the bugs in the body of the `find_occurrences` so that it considers overlapping occurrences of the substring.

```
def find_occurrences(msg, sub):
    """ (str, str) -> list of int

    Return a list containing all of the indicies in msg where sub
    appears.

    >>> find_occurrences('Paul eats lollipops', 'lol')
    [10]
    >>> find_occurrences('lololol', 'lol')
    [0, 2, 4]
    """
```

Line	Original Function Body
1.	`    result = ''`
2.	`    occurrence = msg.find(sub)`
3.	`    while occurrence != -1:`
4.	`        result = result.append(occurrence)`
5.	`        occurrence = msg.find(sub)`
6.	`    return result`

Line	Modified Function Body - Overlapping
1.	
2.	
3.	
4.	
5.	
6.	

Part (b) [3 MARKS]

In this part, correct the bugs in the body of the `find_occurrences` so that it does NOT consider overlapping occurrences of the substring.

```
def find_occurrences(msg, sub):
    """ (str, str) -> list of int

    Return a list containing all of the indicies at in msg
    where sub appears.

    >>> find_occurrences('Paul eats lollipops', 'lol')
    [10]
    >>> find_occurrences('lololol', 'lol')
    [0, 4]
    """
```

Line	Original Function Body
1.	`    result = ''`
2.	`    occurrence = msg.find(sub)`
3.	`    while occurrence != -1:`
4.	`        result = result.append(occurrence)`
5.	`        occurrence = msg.find(sub)`
6.	`    return result`

Line	Modified Function Body - Non-overlapping
1.	
2.	
3.	
4.	
5.	
6.	

Question 8. [8 MARKS]

In a play, the lines each character speaks have the following format:

```
[CHARACTER_NAME] first line of dialog
    additional lines of dialog (0 or more)
```

A file representing a play will contain character dialog formatted as above.

Here is an example from Shakespeare's *Antony and Cleopatra*:

```
[CLEOPATRA] I am sick and sullen.
[MARK ANTONY] I am sorry
    to give breathing
    to my purpose,--
[CLEOPATRA] Help me away,
    dear Charmian; I shall fall:
    It cannot be thus long,
    the sides of nature
    Will not sustain it.
[MARK ANTONY] Now,
    my dearest queen,--
[CLEOPATRA] Pray you,
    stand further from me.
[MARK ANTONY] What's the matter?
```

In this question, you will write a function that reads a file containing lines from a play in this format. Some things you can assume:

- Square brackets are never used except to mark a `CHARACTER_NAME`.
- `CHARACTER_NAME`s may be any case, but will have consistent case throughout the file.
- There are no blank lines.
- There will always be a non-empty first line of dialog.

On the following page, complete the body of the function according to its docstring. Your solution should be general, and work for any play file that has the format described above. You must use the existing starter code in your solution.

Assume the `play.txt` file referred to in the docstring example is the example file above.

Question 8 continued...

```
def read_lines(play, character):
    """ (file open for reading, str) -> list of str

    Return the list of dialogs (with all newlines removed) made by character in play.

    >>> file = open('play.txt')
    >>> actual = read_lines(file, 'MARK ANTONY')
    >>> expected = \
    ['I am sorry to give breathing to my purpose,--', \
    'Now, my dearest queen,--', \
    "What's the matter?"]
    >>> actual == expected
    True
    >>> file.close()
    """

    result = []

    # read the first line, which includes a character's name and first line of dialog
    line = play.readline().strip()

    while line:
        # Your answer goes here
```

Question 9. [5 MARKS]

Part (a) [4 MARKS]

The following function does not behave the way the docstring claims (although the type contract is correct).

```
def are_teenagers(ages):
    """ (list of int) -> list of bool

    Precondition: len(ages) > 0 and all the ints in ages are positive.

    Returns a list of boolean values where the element at index i of the list
    is True iff ages[i] is a teenager between 13 and 19 inclusive.
    """

    res = []
    for i in range(len(ages)):
        if ages[i] > 12:
            if ages[i] < 19:
                res.append(True)
            else:
                res.append(False)

    return res
```

In the table below, write the simplest two possible test cases that reveal two different bugs in the code.

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

Part (b) [1 MARK]

Give a formula in terms of k, where k is `len(ages)`, that describes how many times the loop iterates when the `are_teenagers` function (as defined above) is called.

Question 10. [8 MARKS]

In this question, you will develop two classes to represent enzymes and DNA strands.

Here is the header and docstring for class `Enzyme`.

```
class Enzyme:
    """ Information about a particular enzyme. """
```

Part (a) [2 MARKS] Here is the header and docstring for method `__init__` in class `Enzyme`. Complete the body of this method.

```
    def __init__(self, enzyme_name, recognition_sequence):
        """ (Enzyme, str, str) -> NoneType

        Initialize a new enzyme with name enzyme_name and sequence
        recognition_sequence.

        >>> enzyme1 = Enzyme('Sau3A', 'GATC')
        >>> user1.name
        'Sau3A'
        >>> user1.sequence
        'GATC'
        """
```

Part (b) [2 MARKS] Here is the header and docstring for method `__str__` in class `Enzyme`. Complete the body of this method.

```
    def __str__(self):
        """ (Enzyme) -> str

        Return a string representation of this enzyme.

        >>> enzyme1 = Enzyme('Sau3A', 'GATC')
        >>> print(enzyme1)
        The enzyme Sau3A has a recognition sequence of length 4
        >>> enzyme2 = Enzyme('HgaI', 'GACGC')
        >>> print(enzyme2)
        The enzyme HgaI has a recognition sequence of length 5
        """
```

Here is the header and docstring for class `DNAStrand`.

```
class DNAStrand:
    """ Information about a DNA strand. """
```

Part (c) [1 MARK] Here is the header and docstring for method `__init__` in class `DNAStrand`.

Complete the body of this method.

```
    def __init__(self, new_strand):
        """ (DNAStrand, str) -> NoneType

        Initialize a new DNA strand that has strand new_strand and an empty list
        of enzymes.

        >>> strand1 = DNAStrand('AGGCCT')
        >>> strand1.strand
        'AGGCCT'
        >>> strand1.enzymes
        []
        """
```

Part (d) [1 MARK] Here is the header and partial docstring for method `is_cutter` in class `DNAStrand`. Complete the type contract for the method.

```
    def is_cutter(self, enzyme):

        """ [                                                              ]

        Return True iff enzyme's sequences appears one or more times
        in this DNA strand's strand.

        >>> enzyme1 = Enzyme('HaeIII', 'GGCC')
        >>> strand1 = DNAStrand('AGGCCT')
        >>> strand1.is_cutter(enzyme1)
        True
        >>> enzyme2 = Enzyme('Sau3A', 'GATC')
        >>> strand1.is_cutter(enzyme2)
        False
        """

        return enzyme.sequence in self.strand
```

Part (e) [2 MARKS] Here is the header and docstring for method `add_enzyme` in class `DNAStrand`. You must call existing methods in your solution, and not write any duplicate code.

Complete the body of this method.

```
def add_enzyme(self, potential_enzyme_name, potential_enzyme_sequence):
    """ (DNAStrand, str, str) -> NoneType

    Modify this DNA strand's enzyme list to add the potential enzyme with
    name potential_enzyme_name and sequence potential_enzyme_sequence if
    and only if the potential enzyme is a cutter of the strand (as defined
    by the is_cutter method), and potential_enzyme_sequence is not a
    sequence of any enzyme this DNA strand already has. If this DNA strand
    already has an enzyme with that sequence, do not modify the enzyme
    list.

    >>> strand1 = DNAStrand('AGGCCT')
    >>> enzyme1 = Enzyme('HaeIII', 'GGCC')
    >>> strand1.enzymes.append(enzyme1)
    >>> len(strand1.enzymes)
    1
    >>> strand1.add_enzyme('XYZ', 'GGCC')
    >>> len(strand1.enzymes)
    1
    >>> strand1.add_enzyme('StuI', 'AGGCCT')
    >>> len(strand1.enzymes)
    2
    >>> strand1.add_enzyme('Sau3A', 'GATC')
    >>> len(strand1.enzymes)
    2
    """
```

Use the space on this "blank" page for scratch work, or for any answer that did not fit elsewhere.
Clearly label each such answer with the appropriate question and part number, and refer to this answer on the original question page.

Use the space on this "blank" page for scratch work, or for any answer that did not fit elsewhere.
Clearly label each such answer with the appropriate question and part number, and refer to this answer on the original question page.

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.
  abs(x) -> number
    Return the absolute value of x.
  chr(i) -> Unicode character
    Return a Unicode string of one character with ordinal i; 0 <= i <= 0x10ffff.
  int(x) -> int
    Convert x to an integer, if possible.  A floating point argument will be truncated
    towards zero.
  len(x) -> int
    Return the length of the list, tuple, dict, or string x.
  list(iterable) -> list
    Return a new list initialized from iterable's items
  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.
  open(name[, mode]) -> file open for reading, writing, or appending
    Open a file.  Legal modes are "r" (read), "w" (write), and "a" (append).
  ord(c) -> integer
    Return the integer ordinal of a one-character string.
  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.
  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
    Produce the value associated with the key k in D.
  del D[k]
    Remove D[k] from D.
  k in d --> bool
    Produce 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.
```

```
file open for reading:
  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 any newline.
    Return an empty string at EOF (End Of File).
  F.readlines() -> list of str
    Return a list of the lines from the file.  Each string retains any newline.

file open for writing:
  F.close() -> NoneType
    Close the file.
  F.write(x) -> int
    Write the string x to file F and return the number of characters written.

list:
  x in L --> bool
    Produce True if x is in L and False otherwise.
  L.append(x) -> NoneType
    Append x to the end of the list L.
  L.extend(iterable) -> NoneType
    Extend list L by appending elements from the iterable. Strings and lists are
    iterables whose elements are characters and list items respectively.
  L.index(value) -> int
    Return the lowest index of value in L.
  L.insert(index, x) -> NoneType
    Insert x at position index.
  L.pop([index]) -> object
    Remove and return item at index (default last).
  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 *IN PLACE*.

str:
  x in s --> bool
    Produce True if and only if x is in s.
  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.endswith(S2) -> bool
    Return True if and only if S ends with S2.
  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.isalpha() -> bool
  Return True if and only if all characters in S are alphabetic
  and there is at least one character in S.
S.isdigit() -> bool
  Return True if all characters in S are digits
  and there is at least one character in S, and False otherwise.
S.islower() -> bool
  Return True if and only if all cased characters in S are lowercase
  and there is at least one cased character in S.
S.isupper() -> bool
  Return True if and only if all cased characters in S are uppercase
  and there is at least one cased character in S.
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.startswith(S2) -> bool
  Return True if and only if S starts with S2.
S.strip([chars]) -> str
  Return a copy of S with leading and trailing whitespace removed.
  If chars is given and not None, remove characters in chars instead.
S.upper() -> str
  Return a copy of the string S converted to uppercase.
```