CSS 390C - Autumn 2020 Final Exam

Introduction

This test is worth 30% of your final grade, one mark = 1%. There are 40 points available, with any excess to be applied to points lost in the assignments and/or midterm. It is usually most effective to tackle the easiest questions first. It's up to you to decide which ones are easier. You might want to skim through the exam before getting started.

All questions are to be answered using Python. You won't be penalized for missing semicolons and such, so don't panic if you don't remember precise syntax--as long as you're reasonable and clear. For the purposes of the exam, don't worry about cleaning up any temporary files you might create—unless otherwise stated.

Read each question carefully. A large portion of the grading is reading comprehension. Make sure you understand what is being asked. Avoid loss of points from answering the wrong thing.

You may use your notes and/or the internet. ***Cite anything nontrivial*** that you get from the webs; when in doubt, cite. Do not communicate with other students during the exam.

Take a deep breath and don't panic. You have (nominally) 2 hours starting now. Good luck!

1 [10 points]: The local sportsball league plays a round-robin tournament (i.e. every team plays every other team *exactly once* each). Game scores are tracked in a CSV text file of the form

team-1, goals-for-team-1, team-2, goals-for-team-2

For any game, the winning team is the one that scores more goals than the other team. Either team 1 or team 2 could be the winner of any particular game, or they could be tied (the teams score an equal number of goals). In the tournament standings, a team gets 2 points for each game win and 1 point per tie. The overall tournament winner is the team that gets the most points (total goals scored is irrelevant —only the victories).

1.1 [5 points]: Write a function called **`get_scores`** that takes an open file object and calculates points awarded to each team for games played. You may assume the file is well-formed (correct syntax), but you should verify that the tournament structure was met and throw a **`ValueError`** exception if the data lacks *sanity*. I.e. you are checking for logical errors in the data (hint: what invariant(s) should sane data satisfy?). You may do the Right Thing and use the **`csv`** standard library module for reading the file or do the fast-and-dirty thing to extract the fields.

1.2 [5 points]: Write a function **`print_winners`** that takes whatever **`get_scores`** returns and prints the top 3 places. If there is a tie, all teams with the tying rank are awarded. If there is a tie for first place, second place is not awarded. If, between first and second place, at least 3 awards have been given, no third place is awarded.

Hint 1: The **`sorted`** built-in function takes an optional keyword argument **`key`** that expects a callable of one argument that is used to extract the comparison key from each element.

Hint 2: Before coding your solution, think about the data structure(s) that might simplify your code.

Example:

	Input	*Output*
2014-2015	montreal, 3, ottawa, 2 montreal, 3, boston, 1 montreal, 0, toronto, 1 montreal, 4, vancouver, 2 ottawa, 6, boston, 4 ottawa, 1, toronto, 2 ottawa, 1, vancouver, 0 boston, 3, toronto, 4 boston, 3, vancouver, 2 toronto, 3, vancouver, 1	first place: toronto second place: montreal third place: ottawa
2015-2016	montreal, 3, ottawa, 2 montreal, 3, boston, 1 montreal, 0, toronto, 1 montreal, 4, vancouver, 2 ottawa, 6, boston, 4 ottawa, 1, toronto, 2 ottawa, 1, vancouver, 0 boston, 3, toronto, 4 boston, 3, vancouver, 2 toronto, 2, vancouver, 4	first place: toronto, montreal third place: ottawa
2016-2017	montreal, 2, ottawa, 2 montreal, 1, toronto, 0 ottawa, 1, toronto, 2 vancouver, 3, montreal, 3 ottawa, 4, boston, 5 boston, 2, vancouver, 3 vancouver, 2, toronto, 4 montreal, 3, boston, 0 ottawa, 3, vancouver, 3 boston, 2, toronto, 0	first place: montreal second place: toronto, boston, vancouver

2 [10 points]

2.1 [1 point]: Write a function **`sleeper(t)`** that takes a time **`t`** in seconds and returns a no-argument callable that calls **`time.sleep(t)`**.

2.2 [9 points]: Write a function **`max_tries`** that takes a **`count`** and an optional callable **`on_retry`**. **`max_tries`** should return a function that can be used as a decorator. The decorator will keep trying the decorated function until a value other than **`None`** is returned, an exception is thrown, or the count of tries is exceeded. If the count is exceeded, raise the **`MaxRetriesError`** exception. After each failed try except the last one (i.e. before every *re*try), the **`on_retry`** function should be called, if one was supplied.

Typically, the retry function would print a log message and/or have a delay to give the remote site you're trying to connect to time to reset itself.

Note: **`max_tries`** is not the decorator. The thing **`max_tries`** returns is the decorator. See the sample usage on the next page.

Example:

```
>>> thrice = max_tries(3, lambda:print("please answer Yes"))
>>>
>>> @thrice
... def ask():
...     print("query?")
...     l = sys.stdin.readline()
...     if l[0] == 'y' or l[0] == 'Y':
...         return True
...     return None
...
>>> ask()
query?
yes
True
>>> ask()
query?
no
please answer Yes
query?
no
please answer Yes
query?
yes
True
>>> ask()
query?
no
please answer Yes
query?
no
please answer Yes
query?
no
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/morris/uwb/css390/2018-q4/final/src/max_tries.py", line 24, in decorated
    raise MaxRetriesError("maximum number of retries exceeded")
max_tries.MaxRetriesError: maximum number of retries exceeded
>>>
```

3 [10 points]: A *vector* **A = (a_0, a_1, …, a_{n-1})** is a sequence of **n** *scalars* (numbers). A vector is considered to be *sparse* when the number **m** of non-zero elements is much, much smaller than the length, *and* the length is sufficiently large that it matters. To save memory, one way to represent a sparse vector is to maintain a list pairs **(a_i, i)** where **a_i** is a non-zero element. For efficient lookup (and other) operations, the list is sorted by the index **i**.

Note that most of the values in the (conceptual) vector are not physically stored in the data structure and are simply the default value.

Write a class **SparseVector** that implements the following methods:

3.1 [2 points]: The constructor should take two arguments: a length **n**, and an iterable of *unsorted* pairs, which you must sort (note that you are sorting on the second component of the pair).

3.2 [3 points]: The multiply operator (*****) should efficiently (i.e **O(m)**) compute the dot product of two vectors. The dot product is a scalar value defined as $\vec{A} \cdot \vec{B} = \sum_{i=0}^{n-1} (a_i \cdot b_i)$.

Hint: if for some value **i**, either **a_i** or **b_i** is zero, the product **a_i*b_i** is zero.

3.3 [2 points]: **x.avg** is a property that takes a sparse vector **x** and efficiently computes the average of its components.

$$avg(\boldsymbol{x}) = \frac{\sum_{i=0}^{n-1} x_i}{n} \quad .$$

3.4 [3 points]: Using your solution to 3.2 and/or 3.3 or otherwise, write a function (not a method) **cov(x, y)** takes sparse vectors **x** and **y** and efficiently computes their covariance

$$cov(\boldsymbol{x}, \boldsymbol{y}) = \left(\frac{\sum_{i=0}^{n-1} (x_i \cdot y_i)}{n}\right) - \left(\frac{(\sum_{i=0}^{n-1} x_i) \cdot (\sum_{i=0}^{n-1} y_i)}{n^2}\right) \quad .$$

You may assume the two vectors have equal length **n**, but not necessarily the same number of non-zero elements **m_x** and **m_y**. *Do not assume the non-zero elements are aligned in any way*.

Hint: Consider the **sum** built-in function, generator expressions, iterators, etc.

Example:

```
>>> v1 = SparseVector(
...     200,
...     [(-0.946, 6),
...      (-0.541, 11),
...      ( 0.211, 61),
...      (-0.966, 62),
...      (-0.380, 91),
...      (-0.736, 101),
...      ( 0.743, 123),
...      (-0.426, 139),
...      (-0.038, 159),
...      ( 0.787, 195)])
>>>
>>> v2 = SparseVector(
...     200,
...     [(-0.849, 3),
...      ( 0.579, 6),
...      (-0.876, 57),
...      ( 0.933, 62),
...      (-0.323, 69),
...      ( 0.008, 101),
...      ( 0.953, 108),
...      ( 0.522, 116),
...      ( 0.100, 139),
...      ( 0.106, 143),
...      (-0.162, 159),
...      (-0.594, 161),
...      (-0.196, 167),
...      ( 0.379, 195)])
>>>
>>> print("averages:   {:9.6f}  {:9.6f}".format(v1.avg, v2.avg))
averages:   -0.011460   0.002900
>>> print("dot product:  {:9.6f}".format(v1 * v2))
dot product:  -1.193071
>>> print("magnitudes:  {:9.6f}  {:9.6f}".format(math.sqrt(v1 * v1), math.sqrt(v2 * v2)))
magnitudes:   2.050782   2.135735
>>> print("covariance:  {:9.6f}".format(cov(v1, v2)))
covariance:  -0.005932
```

4 [10 points]: Create a class **Emitter** that has a field `level` and methods `print` and `indent`. The print method takes the same arguments as the built-in print function but prints each line indented by 4 spaces times the indentation level. The indent method should return a context manager that increments the indent level and restores indentation at the end of the block.

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Usage example:

```
morris@yogi2:~/uwb/css390/2019-q4/final/src$ cat -n example.py
     1  #! /usr/bin/python3
     2
     3  from emitter import Emitter
     4
     5  e = Emitter()
     6
     7  def cases():
     8      for case in ["THIS", "THAT", "OTHER"]:
     9          e.print(f"case {case}:")
    10          with e.indent():
    11              e.print("// this is the code for", case)
    12              e.print("// with some boilerplate")
    13          e.print("break;")
    14
    15  def body():
    16      with e.indent():
    17          e.print("while (true) {")
    18          with e.indent():
    19              e.print("switch(state) {")
    20              with e.indent():
    21                  cases()
    22              e.print("}")
    23          e.print("}")
    24
    25
    26  def machine():
    27      e.print("void Machine(enum State initial_state)")
    28      e.print("{")
    29      body()
    30      e.print("}")
    31
    32
    33  if __name__ == "__main__":
    34      machine()
```

```
morris@yogi2:~/uwb/css390/2019-q4/final/src$ ./example.py
 void Machine(enum State initial_state)
 {
     while (true) {
         switch(state) {
             case THIS:
                 // this is the code for THIS
                 // with some boilerplate
             break;
             case THAT:
                 // this is the code for THAT
                 // with some boilerplate
             break;
             case OTHER:
                 // this is the code for OTHER
                 // with some boilerplate
             break;
         }
     }
 }
```