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 [12 points]

A construction company schedules projects using a set of task records containing tab-separated fields consisting of task-id, start-date, end-date, and zero or more required resources. Dates are represented in ISO-8601 date format: yyyy-mm-dd. Task and resource ids are alphanumeric identifiers.

The company recently had a massive (and expensive) SNAFU due to inadvertently double-booking their excavator. They've fixed the bug in their scheduling system, but you have been assigned the task of looking for other double-bookings in their current project schedules.

Write a script to read task records from standard input. For each resource, collect all the tasks that require that resource to determine whether any tasks have overlapping requirements for that resource. Your report should list the resource and the pair of overlapping tasks on a single line. A resource may be listed more than once if it has more than one pair of overlapping tasks.

Hint 1: the **`datetime`** library module has a **`date`** class which is constructed with numeric arguments for year, month, and day (in that order). The class implements the comparison operators so you may use **`date`** objects as the key field to sort task objects. The **`sorted`** built-in function takes the **`key`** keyword argument which lets you specify a function that returns the comparison key. String-to-int conversion may be done using the **`int`** built-in function.

Hint 2: two tasks are concurrent if the one with the later start date begins before the other one ends. Consider writing a predicate function or method **`overlaps(T1, T2)`** which returns **`True`** if tasks T1 and T2 overlap.

2 [8 points]

2.1 [3 points] In calculus, the derivative of a function f(x) is defined as f'(x) = (f(x + dx) - f(x)) / dx where dx becomes arbitrarily small. We can get a useful approximation of the derivative by using a really small value for dx (relative to x).

Write a function **`make_derivative`** that takes as arguments a callable **`f`** and a numeric value **`dx`**. **`f`** should take a single numeric argument **`x`** and returns a numeric result.

`make_deriviative` should return a callable that computes the derivative f'(x) = (f(x + dx) - f(x)) / dx.

Example:

Analytically, the derivative of $x^2 + 3x + 4$ is $2x + 3$.

```
>>> def f(x):
...     return x ** 2 + 3 * x + 4
...
>>> fprime = make_derivative(f, 0.0001)
>>> for i in range(6):
...     x = 0.5 * i
...     print "%6.2f %6.2f %6.2f" % (x, f(x), fprime(x))
...
  0.00   4.00   3.00
  0.50   5.75   4.00
  1.00   8.00   5.00
  1.50  10.75   6.00
  2.00  14.00   7.00
  2.50  17.75   8.00
```

2.2 [5 points] There is a numerical-analysis technique for finding the roots of an equation, known as the Newton-Raphson method, which involves iterative refinement of an initial guess. It works quite well provided the equation and initial guess satisfy certain conditions (beyond the scope of this exam).

Write a function or class **`root_iterator`** that takes arguments a numeric function **`f`**, an initial guess **`x0`**, a delta value (for approximating the derivative) **`dx`**, and a threshold **`epsilon`**. **`root_iterator`** should create an iterator that computes the sequence $x_{n+1} = x_n - f(x_n)/f'(x_n)$. The iteration should terminate after $|f(x_n)| < \varepsilon$.

Example:

```
>>> def f(x):
...     return (x - 3) * (x + 6)
...
>>> for x in root_iterator(f, -4, 0.0001, 0.01):
...      print "%6.2f  %8.4f" % (x, f(x))
...
 -6.80    7.8406
 -6.06    0.5470
 -6.00    0.0036
>>> for x in root_iterator(f, 12, 0.0001, 0.01):
...      print "%6.2f  %8.4f" % (x, f(x))
...
  6.00   36.0003
  3.60    5.7602
  3.04    0.3190
  3.00    0.0012
```

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;
         }
     }
 }
```