Python Refresher and the Jupyter Notebook

Due 11:59pm Friday, September 6

Objective

A pre-requisite for this course is prior experience with Python at the level of 15-112 or 95-888. This assignment provides an opportunity to brush up on your Python knowledge.

- Start Jupyter notebook (or lab) with `% jupyter notebook` (where `%` is the prompt)
- Start a new notebook and call it `a1-python-review`. The suffix of `ipynb` will be automatically added
- Recall that an *expression* evaluates to a value and a *statement* does something. E.g., `3+4` evaluates to `7` whereas `print(8)` prints `8` on the screen. When you have a sequence of expressions in a cell and evaluate them you only see the value of the last expression. For example, if
  ```
  4+8
  2+5
  ```
  were in a cell and it was evaluated we would only see 7. If you want to see both you need to print the values:
  ```
  print(4+8)
  print(2+5)
  ```
- Use list comprehensions where possible
- Use destructuring assignments (aka tuple unpacking). Rarely should you have to index into a list. For this assignment, avoid doing `lst[i]`.
- You are welcome to use additional libraries (e.g., the `collections` module). Do NOT use `pandas` for this assignment. Pandas will be the focus of the next assignment.

Write the solutions to the following Python exercises in the Python notebook conforming to the following instructions:

1. Write your full name and andrewID in Markdown cell at the top of your notebook.
2. Precede each code cell with a Markdown cell which briefly describes the problem you are solving in the cell below. Just 1-2 lines will suffice. Cut and pasting the problem description from the assignment write-up will be acceptable.
3. For ease of grading, use the specified function names (e.g., `amicable_numbers` etc.)

This is an individual assignment. You are allowed to discuss high level ideas, but the final work needs to be your own. Software tools will be used to check submissions for authenticity. You are welcome (and expected) to use online resources (stackoverflow etc.) to seek answers to specific Python related questions. If you do use any online resources reference them in your markdown comments. **Do not use any AI assistant in solving this assignment.** The work needs to be your own.

Be sure that your submission conforms to the expectations in our Python style guide. Non-conformance will result in points off.

Note:

Please review the Python Style Guide carefully. Ensure you understand the difference between docstrings, markdown comments, and code comments. It is recommended that you write the docstrings (*specifications)* for all functions first before starting to write the code. In separate cells in Jupyter write the *function signature + docstring* for all functions of this assignment. Then come back and start writing the actual code.

The assignment provides some examples of expected behavior. You are required to include test cases of your own (2-3 test cases will suffice) demonstrating that your work meets the specifications. Retain your test cases in your submssion. Extra test cases provided by you contributes towards assignment credit. (You don't need to for formal unit testing etc with `pytest` or so.)

Ensure your final notebook runs without any errors. Rest your Jupyter kernel and run all cells in your notebook double checking that there is no runtime error. Notebooks that do not run will not receive any credit.

Time box your work. Personally I've got a lot of value out of the Pomodoro technique of time management. Start early!

95-885 Data Science and Big Data, Fall 2024

Exercises

1. **Time difference.** Write a function `time_diff` which takes a start time and an end time in military time notation returns the number of hours and minutes between the two times:

```
time_diff( '0900', '1730') -> (8, 30)   # 8 hours and 30 minutes between 9:00 am and 5:30 pm
time_diff( '1730', '0900') -> (15, 30)  # 15 hours and 30 minutes between 5:30 pm and 9:00 am
```

   Note that the first argument is always the start time and the second argument the end time chronologically after the start time except if the start and end time are the same, return (0, 0). Do not use any library functions to do time calculations. Write this function using basic Python.

2. **Amicable Numbers.** A perfect number is equal to the sum of its proper divisors. For example, 6 = 1 + 2 + 3 and 28 = 1 + 2 + 4 + 7 + 14. Amicable numbers are two different numbers so related that the sum of the proper divisors of each is equal to the other number. The smallest pair of amicable numbers are (220, 284). Write a function `amicable_numbers(n)` which will find all pairs of amicable numbers less than or equal to `n`. Do this in two parts:
   - **a.** Write the function `proper_divisors(n)` which returns a list of all the proper divisors of `n`. Use a list comprehension. Essentially it will be a 1 line function. We can now find perfect numbers with a simple list comprehension: `[n for n in range(1,100) if sum(proper_divisors(n)) == n]`
   - **b.** Now write the function `amicable_numbers(n)` which will find all the pairs of amicable numbers less than or equal to `n`. While there are special case formulas (see the Wikipedia article), you are expected to do a brute force process by examining all numbers from 1..n. `amicable_numbers(10000)` should give you `[(220, 284), (1184, 1210), (2620, 2924), (5020, 5564), (6232, 6368)]`. If you take a very simple approach you may land up with a very inefficient algorithm due to the same computation being repeated. Think of ways of avoiding repeating the same computation.

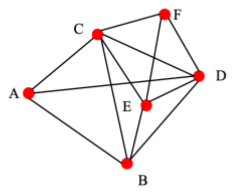
3. **Arabic to Roman numerals.** The number three can be denoted as 3 (arabic numeral) or III (roman numeral). Write a function to convert from arabic numerals to roman numerals:

```
arabic2roman( 3 ) -> 'III'
arabic2roman(48) -> 'XLVIII'
arabic2roman(2023) -> 'MMXXIII'
```

   The value of various symbols in Roman numerals is given below. Note that a `I` can be placed before a `V` (5) or a `X` (10) to make 4 or 9; a `X` can be placed before a `L` (50) or `C` (100) to make 40 or 90; a `C` can be placed before a `D` (500) or M (1000) to make 400 or 900.

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

4. **Genetic Algorithm Encoding.** Genetic algorithms are a evolutionary inspired optimization technique. A classic optimizat ion problem is the Traveling Salesperson Problem (TSP): visit all the cities on a map optimizing some criteria e.g., the total distance travelled. For example for the adjacent map a possible tour is B D A F C E. For reason s we may discuss later such tours are often encoded in the following way.
   - Setup a standard ordering of all the cities in a tour (you may assume all cities will be present exactly once):



```
A B C D E F
0 1 2 3 4 5
```

A tour like `BDAFCE` has the encoding code: 120200
The algorithm is note the position of a letter in the standard ordering and then remove it:

```
B is 1 in ABCDEF
D is 2 in ACDEF
A is 0 in ACEF
F is 2 in CEF
C is 0 in CE
E is 0 in E
```

Write two functions that will do this encoding and decoding.
`encode('BDAFCE')` will return `[1,2,0,2,0,0]` and
`decode([1,2,0,2,0,0])` will return `'BDAFCE'`

5. **Dense to Sparse matrix conversion.** Many big data applications work with large sparse matrices. Large in that they will have several thousand (sometimes even a million) rows and columns. Sparse in that most of the entries will be 0 (typically only 10% of all entries will be non-zero). Storing such a large sparse matrix in a traditional format would be a huge waste of space as we would be storing a whole bunch of zeros. There are several alternative representations for such sparse matrices which are space efficient. One representation is to explicitly list the row index, column index, and value as a list of triples. For example, consider the following dense (traditional) representation of a matrix:

```
[[1, 0, 2]
 [0, 0, 3]]
```

In this sparse representation we only store the non-zero values as a list of triples (i, j, value). Indexing of the rows and columns starts with 0.

```
[(0,0,1), (0,2,2), (1,2,3)]
```

Write a function that will take a dense matrix and return the equivalent sparse representation.

```
dense2sparse(
   [ [1, 0, 2],
     [0, 0, 3]
   ])
```

Returns `[(0,0,1), (0,2,2), (1,2,3)].` Note that an interesting property of this representation for sparse matrices is that the order of the triples doesn't matter.

The main learning objective of this question is (i) practice in list comprehensions and (ii) exposure to the python function `enumerate.` Hence your solution needs to be of the form:

```
def dense2spares(m):
    ans = < a nested list comprehension using enumerate twice >
    return ans
```

Of course, you will specify a doc string, as always. Read online about the `enumerate` function.

6. **Pascal's Triangle.** Is a triangular array of the binomial coefficients arising in probability theory, combinatorics, and algebra. Skim the Wikipedia article (just the intro will suffice) to learn more about the mathematical object. Note that the article gives a recurrence relation for defining the values of each row of the triangle. You are NOT expected to write a recursive solution to this problem. Rather you will generate each row of the triangle from the previous row and iteratively build the triangle. Here are sample outputs:

```
pascal_triangle( 0 ) ->  [ [1] ]
pascal_triangle( 1 ) ->  [ [1], [1,1] ]
pascal_triangle( 2 ) ->  [ [1], [1,1], [1,2,1] ]
pascal_triangle( 3 ) ->  [ [1], [1,1], [1,2,1], [1,3,3,1] ]
```

7. Closure algorithm. A crucial step of the Boyce-Codd normal form in relational database systems is determining the closure of a set of attributes. Write a function `closure` that will take a collection of functional dependencies (FDs) and a set of attributes and return the closure of that attribute set. The collection of FDs is specified by a multi-line string of individual functional dependencies. The set of attributes for which you are determining the closure also specified as a string. Following is an example:

```
FDs_1 = '''
A B -> C
A -> D
D -> E
A C -> B
'''
```

Following is sample output:

```
closure(FDs_1, 'A') -> {'A', 'D', 'E'}
closure(FDs_1, 'A B') -> {'A', 'B', 'C', 'D', 'E'}
closure(FDs_1, 'B') -> {'B'}
```

As a quick review, you form the closure of a set of attributes by repeatedly adding the right-hand side of applicable FDs. Following is a detailed work thru.

To determine the closure of `'A'` with reference to `FDs_1` we have:
`{'A'}` -- initialize with `'A'`
`{'A', 'D'}` -- since `A -> D`
`{'A', 'D', 'E'}` -- since `D-> E`
No more FDs apply and we are done.

A learning objective of this question is to gain experience working with the *set* data type of Python. The following are requirements of your solution.

i. Write a function `parse_fd` which takes a string representation of a functional dependency and will return a *tuple of sets* representing the left-hand-side and right-hand-side of the FD.

   ```
   parse_fd('A B -> C') -> ( {'A', 'B'}, {'C'})
   ```

ii. Write two helper functions `lhs` and `rhs` which will take a tuple representation of an FD and return the corresponding left-hand or right-hand side. These are essentially 1 line functions.

iii. The `closure` function will take the set of attributes, whose closure it is to determine, and convert that string representation to a set representation. Inside the body of `closure` you should NOT index into the tuple with `[0]` or `[1]` but rather use the helpers `lhs` and `rhs`. You will want to review the operations possible on the set data type in Python (e.g., `.issubset`) and write your code using the set operations.

Your solution needs to be independent of the order of the functional dependencies i.e., if you permute the order of the FDs you should get the same result.

What to submit:

A single Jupyter notebook `a1-python-review.ipynb`

(1) Please ensure your notebook runs without any errors. (2) Double check that you have submitted the correct file to Canvas. Notebooks with execution errors will not receive any credit. No resubmissions will be accepted beyond the submission deadline.