Midterm 1, Fall 2023: Email centrality

Version history:

- 1.0.5: Fixed bug in notebook code (no effect on exercises)
- 1.0.4: Fixed another typo on Ex. 7 prompt
- 1.0.3: Fixed small bug in ex3 test
- 1.0.2: Fixed typo on Ex.7 prompt
- 1.0.1: Fixed issue with Ex. 7 cell being read-only
- 1.0: Initial release

All of the header information is important. Please read it.

Topics, number of exercises: This problem builds on your knowledge of the core Python data structures, strings, and ability to translate simple math into code. It has **8** exercises, numbered 0 to **7**. There are **13** available points. However, to earn 100% the threshold is **11** points. (Therefore, once you hit **11** points, you can stop. There is no extra credit for exceeding this threshold.)

Exercise ordering: Each exercise builds logically on previous exercises, but you may solve them in any order. That is, if you can't solve an exercise, you can still move on and try the next one. Use this to your advantage, as the exercises are **not** necessarily ordered in terms of difficulty. Higher point values generally indicate more difficult exercises.

Demo cells: Code cells starting with the comment `### define demo inputs` load results from prior exercises applied to the entire data set and use those to build demo inputs. These must be run for subsequent demos to work properly, but they do not affect the test cells. The data loaded in these cells may be rather large (at least in terms of human readability). You are free to print or otherwise use Python to explore them, but we did not print them in the starter code.

Debugging you code: Right before each exercise test cell, there is a block of text explaining the variables available to you for debugging. You may use these to test your code and can print/display them as needed (careful when printing large objects, you may want to print the head or chunks of rows at a time).

Exercise point breakdown:

- Exercise 0: **1** point
- Exercise 1: **2** points
- Exercise 2: **1** point — **FREE** *(no coding, but you must submit to get this point)*
- Exercise 3: **1** point
- Exercise 4: **1** point
- Exercise 5: **2** points
- Exercise 6: **2** points
- Exercise 7: **3** points

Final reminders:

- Submit after **every exercise** to maximize credit
- Review the generated grade report after you submit to see what errors were returned
- Stay calm, skip problems as needed, and take short breaks at your leisure

Overview: Email Centrality

The US Government is investigating possible fraud committed by a company. The company has a large email archive, so the investigators are asking for your analytics help.

Thankfully, a former CSE 6040 student created a neat analysis tool that, given a collection of objects and their relationships to one another, can rank the objects by the "importance" of their connections. But to use that tool, you need to take the raw email archive and convert it into the form that the tool expects.

Here is your overall workflow for this problem:

- **Part A**: Two warm-up exercises
- **Part B**: The data: Enron emails and "email objects"
- **Part C**: Data cleaning and preliminary analysis
- **Part D**: Interfacing with the analysis tool

Before beginning, run the following code, which will set up some of the environment and data you'll need.

```python
In [ ]: ### Global Imports
        ### BEGIN HIDDEN TESTS
        %load_ext autoreload
        %autoreload 2

        if False: # set to True to set up
            REGENERATE_OUTPUTS = True

            import dill
            import hashlib
            def hash_check(f1, f2, verbose=True):
                with open(f1, 'rb') as f:
                    h1 = hashlib.md5(f.read()).hexdigest()
                with open(f2, 'rb') as f:
                    h2 = hashlib.md5(f.read()).hexdigest()
                if verbose:
                    print(h1)
                    print(h2)
                assert h1 == h2, f'The file "{f1}" has been modified'
            with open('resource/asnlib/public/hash_check.pkl', 'wb') as f:
                dill.dump(hash_check, f)
            del hash_check
            with open('resource/asnlib/public/hash_check.pkl', 'rb') as f:
                hash_check = dill.load(f)
            for fname in ['testers.py', '__init__.py', 'test_utils.py']:
                hash_check(f'tester_fw/{fname}', f'resource/asnlib/public/
        {fname}')
            for fname in ['__init__.py', 'utils.py', 'ranker.py']:
                hash_check(f'cse6040/{fname}', f'resource/asnlib/public/cse
        6040/{fname}')
            del hash_check
        ### END HIDDEN TESTS

        # Some extra functions that this notebook needs:
        from pprint import pprint # Pretty-printer

        import cse6040
        from cse6040.utils import load_text_from_file, load_obj_from_file
        from cse6040.enron import load_emails
        from cse6040.ranker import rank_items
```

Part A: Warm-up Exercises

Here are two basic Python exercises to help you "warm up" and, hopefully, build your confidence.

Recall: Python's `filter` function

Recall the built-in Python function, `filter(fun, iterable)`, introduced in the Prymer from the first week of class: given an iterable sequence `iterable` and a function `fun`, `filter` calls `fun` on each element `e` of the sequence and returns a list of only those items where `fun(e)` is True. For example:

```
In [ ]:  def is_even(x): # Returns `True` only if `x` is an even integer
             return (x % 2) == 0

         # Filter a sequence, keeping just the even values:
         for val in filter(is_even, [3, 6, 8, 2, 7, 1, 4, 9, 5, 0]):
             print(val, end=' ')
```

Let's write a simple function to extend the functionality of `filter`.

Ex. 0 (1 pt): `enum_filter`

Suppose we want to filter a sequence but return **both** the items for which the function is `True` **and** the integer **position** of the item in the sequence, assuming the first item is at position 0. Complete the function,

```
def enum_filter(fun, iterable):
    ...
```

to perform this task.

Inputs:

- `fun`: A function that takes a single value and returns either `True` or `False`.
- `iterable`: An iterable sequence of values that can be indexed by an integer starting from 0, like a `str` or a `list`.

Output: Your function should return a Python list of (`position, value`) pairs where `fun(value)` is `True`. See the demo cell below for examples.

Additional notes:

- The tester only uses `iterable` objects that can be indexed with integers, like `str` and `list`.
- The elements or values of the sequence may be strings, integers, or floats.

```
In [ ]:  ### Define demo inputs ###

         # The function: Returns `True` if `e` is an all-lowercase string
         demo_fun_ex0 = lambda e: e == e.lower()

         # Three test cases: A string, a list of characters, and a set of ch
         aracters
         demo_iterable_ex0A = 'The QuiCk broWN fOX jumPEd ovEr thE lAzY DOg'
         demo_iterable_ex0B = list('The QuiCk broWN fOX jumPEd ovEr thE lAzY
         DOg')
```

The demo included in the solution cell below should display the following output:

```
=== Test case A ===
[(1, 'h'), (2, 'e'), (3, ' '), (5, 'u'), (6, 'i'), (8, 'k'), (9, ' '), (1
0, 'b'), (11, 'r'), (12, 'o'), (15, ' '), (16, 'f'), (19, ' '), (20,
'j'), (21, 'u'), (22, 'm'), (25, 'd'), (26, ' '), (27, 'o'), (28, 'v'),
(30, 'r'), (31, ' '), (32, 't'), (33, 'h'), (35, ' '), (36, 'l'), (38,
'z'), (40, ' '), (43, 'g')]

=== Test case B ===
[(1, 'h'), (2, 'e'), (3, ' '), (5, 'u'), (6, 'i'), (8, 'k'), (9, ' '), (1
0, 'b'), (11, 'r'), (12, 'o'), (15, ' '), (16, 'f'), (19, ' '), (20,
'j'), (21, 'u'), (22, 'm'), (25, 'd'), (26, ' '), (27, 'o'), (28, 'v'),
(30, 'r'), (31, ' '), (32, 't'), (33, 'h'), (35, ' '), (36, 'l'), (38,
'z'), (40, ' '), (43, 'g')]
```

```python
In [ ]:  ### Exercise 0 solution
         def enum_filter(fun, iterable):
             ### BEGIN SOLUTION
             return list(filter(lambda e: fun(e[1]), enumerate(iterable)))
             ### END SOLUTION

         ### demo function call
         print("=== Test case A ===", enum_filter(demo_fun_ex0, demo_iterabl
         e_ex0A), '', sep="\n")
         print("=== Test case B ===", enum_filter(demo_fun_ex0, demo_iterabl
         e_ex0B), '', sep="\n")
```

The cell below will test your solution for Exercise 0. The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [ ]:
```python
### test_cell_ex0
### BEGIN HIDDEN TESTS
import dill
import hashlib
with open('resource/asnlib/public/hash_check.pkl', 'rb') as f:
    hash_check = dill.load(f)
for fname in ['testers.py', '__init__.py', 'test_utils.py']:
    hash_check(f'tester_fw/{fname}', f'resource/asnlib/public/{fnam
e}')
del hash_check
del dill
del hashlib
### END HIDDEN TESTS
from tester_fw.testers import Tester

conf = {
    'case_file':'tc_0',
    'func': enum_filter, # replace this with the function defined a
bove
    'inputs':{ # input config dict. keys are parameter names
        'fun':{
            'dtype':'function', # data type of param.
            'check_modified': False,
        },
        'iterable':{
            'dtype':'', # data type of param.
            'check_modified': True,
        }
    },
    'outputs':{
        'output_0':{
            'index':0,
            'dtype':'list',
            'check_dtype': True,
            'check_col_dtypes': True, # Ignored if dtype is not df
            'check_col_order': True, # Ignored if dtype is not df
            'check_row_order': True, # Ignored if dtype is not df
            'check_column_type': True, # Ignored if dtype is not df
            'float_tolerance': 10 ** (-6)
        }
    }
}
tester = Tester(conf, key=b'66lGoH4BeY6guYRngj0DTLf5PgS5u60_N0vVqGy
XN7w=', path='resource/asnlib/publicdata/')
for _ in range(70):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, tru
e_output_vars) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, tru
e_output_vars) = tester.get_test_vars()
        raise

### BEGIN HIDDEN TESTS
tester = Tester(conf, key=b'1VrZu8ZlwHw20ESTvKgqVXujnQVImp9nhaWtlHV
uprc=', path='resource/asnlib/publicdata/encrypted/')
for _ in range(20):
    try:
        tester.run_test()
```

```
            (input_vars, original_input_vars, returned_output_vars, tru
e_output_vars) = tester.get_test_vars()
        except:
            (input_vars, original_input_vars, returned_output_vars, tru
e_output_vars) = tester.get_test_vars()
            raise
### END HIDDEN TESTS
print('Passed! Please submit.')
```

Ex. 1 (2 pt): `invert_dict`

Later in this notebook, we will need to **invert** a Python dictionary. That means we want to construct a new dictionary where keys and values of the original dictionary have been swapped.

Complete the function,

```
def invert_dict(dictionary):
    ...
```

to accomplish this task, as specified below.

Input: The input `dictionary` is a Python dictionary.

Your task: Build and return the "inverse" of `dictionary`. That is, suppose `(key, value)` is a key-value pair, meaning `dictionary[key] == value`. Then the `inverse_dictionary` will have the pair `inverse_dictionary[value] == key`.

Outputs: A new Python dictionary that is the inverse of the input `dictionary`. **However**, if the input `dictionary` has any values that **repeat**, then your function should raise a `ValueError` exception.

Additional notes:

1. Similar to Notebook 5.1, there is a wrapper function, `eif_wrapper`, that can be used to detect exceptions without halting this notebook.
2. The keys and values of the dictionary may be integers or strings.

```
In [ ]:   # ValueError wrapper
          def eif_wrapper(s, func):
              """
              Returns a (bool, function return) pair where the first element
          is True when a ValueError is raised
              and False if a Value Error is not raised. The second output is
          the return value from calling `func(s)`.
              """
              raised_value_error = False
              result = None
              try:
                  result = func(s)
              except ValueError:
                  raised_value_error = True
              finally:
                  return (raised_value_error, result)
```

```
In [ ]:  ### Define demo inputs ###

         demo_dictionary_ex1_okay = {1: 7, 'cat': 9, 3: False}
         demo_dictionary_ex1_bad = {1: 7, 'cat': 9, 3: 7}
```

The demo cell calls your function via `eif_wrapper` and prints its returned value, which is a pair. You will see `(True, None)` if your function raised an exception, and otherwise `(False, ...)`, where `...` is the returned value. For the demo inputs, a correction solution will print the following:

```
(False, {7: 1, 9: 'cat', False: 3})
(True, None)
```

```
In [ ]:  ### Exercise 1 solution
         def invert_dict(dictionary):
             ### BEGIN SOLUTION
             inverse = {value: key for key, value in dictionary.items()}
             if len(inverse) != len(dictionary):
                 raise ValueError("This dictionary is non-invertible.")
             return inverse

         ### demo function call
         print(eif_wrapper(demo_dictionary_ex1_okay, invert_dict))
         print(eif_wrapper(demo_dictionary_ex1_bad, invert_dict))
```

The cell below will test your solution for Exercise 1. The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [ ]:
```python
### test_cell_ex1
### BEGIN HIDDEN TESTS
import dill
import hashlib
with open('resource/asnlib/public/hash_check.pkl', 'rb') as f:
    hash_check = dill.load(f)
for fname in ['testers.py', '__init__.py', 'test_utils.py']:
    hash_check(f'tester_fw/{fname}', f'resource/asnlib/public/{fnam
e}')
del hash_check
del dill
del hashlib
### END HIDDEN TESTS
from tester_fw.testers import Tester

conf = {
    'case_file':'tc_1',
    'func': lambda dictionary: eif_wrapper(dictionary, invert_dic
t),
    'inputs':{ # input config dict. keys are parameter names
        'dictionary':{
            'dtype':'dict', # data type of param.
            'check_modified':True,
        }
    },
    'outputs':{
        'output_0':{
            'index':0,
            'dtype':'bool',
            'check_dtype': True,
            'check_col_dtypes': True, # Ignored if dtype is not df
            'check_col_order': True, # Ignored if dtype is not df
            'check_row_order': True, # Ignored if dtype is not df
            'check_column_type': True, # Ignored if dtype is not df
            'float_tolerance': 10 ** (-6)
        }
    }
}
tester = Tester(conf, key=b'66lGoH4BeY6guYRngj0DTLf5PgS5u60_N0vVqGy
XN7w=', path='resource/asnlib/publicdata/')
for _ in range(70):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, tru
e_output_vars) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, tru
e_output_vars) = tester.get_test_vars()
        raise

### BEGIN HIDDEN TESTS
tester = Tester(conf, key=b'1VrZu8ZlwHw20ESTvKgqVXujnQVImp9nhaWtlHV
uprc=', path='resource/asnlib/publicdata/encrypted/')
for _ in range(20):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, tru
e_output_vars) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, tru
```

```
        e_output_vars) = tester.get_test_vars()
                raise
        ### END HIDDEN TESTS
        print('Passed! Please submit.')
```

Part B: Enron Data and `email` Objects

The company under investigation is Enron Corporation. The following code will load a sample of Enron's email archives into a list of `email` objects, which is a special object for representing email messages.

Let's explore the data *and* learn about `email` objects with a **FREE** 1-point exercise!

Ex. 2 (1 pt - FREE): `email` Object Tutorial

This next "exercise" requires no coding. All you have to do is read the tutorial on email objects, below. You'll need these concepts in subsequent exercises.

The following cell loads the email archive into a list, with one element per email message:

```
In [ ]:  emails = load_emails()
         type(emails), len(emails)
```

Let's display one of these messages, first by inspecting its type:

```
In [ ]:  email_example = emails[5]
         type(email_example)
```

Next, let's ask for a visual representation of the email as a string:

```
In [ ]:  print(email_example.as_string())
```

These `email` message objects allow simple queries. For example, the next few cells extract the contents of the `'From'` and `'Subject'` fields:

```
In [ ]:  email_example.get('From')
```

```
In [ ]:  email_example.get('Subject')
```

Neat! Of course, it's going to turn out that we'll still need to do a little cleaning to complete our task. For instance, take a look at the `'To'` field of this example:

```
In [ ]:  email_example.get('To')
```

You can see that it is a string, but one where the different addresses are separated by commas and whitespace, including newlines. That fact is more clear when you use `print`:

```
In [ ]: print(email_example.get('To'))
```

Okay, that is enough information to get you started. Run the test cell below to get your **FREE** point!

```
In [ ]: ### test_cell_ex2

print('Passed! Please submit.')
```

Ex. 3 (1 pt): `gather_addresses_from_field`

Let's write a function to collect the email *addresses* from a specified field of the email object. By "field" we mean things like `'From'` and `'To'` from the previous "exercise."

Complete the function,

```
def gather_addresses_from_field(email, field):
    ...
```

so that it takes an email object, `email`, and a target field, `field`, and returns the set of emails stored in that field.

Inputs:

- `email`: An email object
- `field`: A string naming the field to search, such as `'From'` and `'To'`.

Your task: Extract all email addresses from the field.

Outputs: Return a Python `set` of the email addresses.

Additional notes (IMPORTANT!):

1. To help you out, the function below includes a predefined regular expression that *robustly* matches a valid email. It is based on the RFC5322 standard, and you can [this demo on regex101 (https://regex101.com/r/dJ9rWc/1)](https://regex101.com/r/dJ9rWc/1).
2. A field **might be empty**! In that case, querying the field might return an empty string **or** a `None` object. In these instances, **you should return an empty set**.

```
In [ ]: ### Define demo inputs ###

demo_email_ex3 = email_example
```

The demo included in the solution cell checks four fields: `'From'`, `'To'`, `'Cc'`, and `'Bcc'`. A correct solution will display the following output:

```
'From' ==> {'im-urlaub@t-online.de'}

'To' ==> {'onvacation@pdq.net', 'ddarter@webtv.net', 'kimberly.s.olinger@
enron.com', 'scott.hendrickson@enron.com', 'bdarter@coral-energy.com', 'v
daniels@imsday.com', 'cnmfree@hrb.de'}

'Cc' ==> set()

'Bcc' ==> set()
```

```
In [ ]: ### Exercise 3 solution
        def gather_addresses_from_field(email, field):
            pattern = r'''((?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%
        &'*+/=?^_`{|}~-]+)*|\"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x
        5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])*\")@(?:(?:[a-z0-9](?:[a-z0
        -9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:25[0-5]|
        2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-
        9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a
        \x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\]))'''

            ### BEGIN SOLUTION
            from re import findall
            field_value = email.get(field) or ''
            return set(findall(pattern, field_value))
            ### END SOLUTION

        ### demo function call ###
        for demo_field_ex3 in ['From', 'To', 'Cc', 'Bcc']:
            demo_result_ex3 = gather_addresses_from_field(demo_email_ex3, d
        emo_field_ex3)
            print(f"'{demo_field_ex3}' ==> {demo_result_ex3}\n")
```

The cell below will test your solution for Exercise 3. The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

```
In [ ]:   ### test_cell_ex3
          ### BEGIN HIDDEN TESTS
          import dill
          import hashlib
          with open('resource/asnlib/public/hash_check.pkl', 'rb') as f:
              hash_check = dill.load(f)
          for fname in ['testers.py', '__init__.py', 'test_utils.py']:
              hash_check(f'tester_fw/{fname}', f'resource/asnlib/public/{fnam
          e}')
          del hash_check
          del dill
          del hashlib
          ### END HIDDEN TESTS
          from tester_fw.testers import Tester

          conf = {
              'case_file':'tc_3',
              'func': gather_addresses_from_field, # replace this with the fu
          nction defined above
              'inputs':{ # input config dict. keys are parameter names
                  'email':{
                      'dtype':'', # data type of param.
                      'check_modified':False,
                  },
                  'field':{
                      'dtype':'str',
                      'check_modified':False
                  }
              },
              'outputs':{
                  'output_0':{
                      'index':0,
                      'dtype':'set',
                      'check_dtype': True,
                      'check_col_dtypes': True, # Ignored if dtype is not df
                      'check_col_order': True, # Ignored if dtype is not df
                      'check_row_order': True, # Ignored if dtype is not df
                      'check_column_type': True, # Ignored if dtype is not df
                      'float_tolerance': 10 ** (-6)
                  }
              }
          }
          tester = Tester(conf, key=b'66lGoH4BeY6guYRngj0DTLf5PgS5u60_N0vVqGy
          XN7w=', path='resource/asnlib/publicdata/')
          for _ in range(70):
              try:
                  tester.run_test()
                  (input_vars, original_input_vars, returned_output_vars, tru
          e_output_vars) = tester.get_test_vars()
              except:
                  (input_vars, original_input_vars, returned_output_vars, tru
          e_output_vars) = tester.get_test_vars()
                  raise

          ### BEGIN HIDDEN TESTS
          tester = Tester(conf, key=b'1VrZu8ZlwHw20ESTvKgqVXujnQVImp9nhaWtlHV
          uprc=', path='resource/asnlib/publicdata/encrypted/')
          for _ in range(20):
              try:
                  tester.run_test()
```

```
        (input_vars, original_input_vars, returned_output_vars, tru
e_output_vars) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, tru
e_output_vars) = tester.get_test_vars()
        raise
### END HIDDEN TESTS
print('Passed! Please submit.')
```

RUN ME: `all_addresses`

Using a correct implementation of `gather_addresses_from_field`, we can sweep the entire email archive and gather all email addresses. The cell loads the results of such a sweep. **Run this cell whether or not your implementation is correct.**

```
In [ ]:  all_addresses = load_obj_from_file('ex3_all_addresses.dill')

         print(f"There are {len(all_addresses):,} unique email addresses in
         the archive, considering the `'From'`, `'To'`, `'Cc'`, and `'Bcc'`
         fields.")
```

Ex. 4 (1 pt): `get_enron_addresses`

The government wants us to focus on Enron employees. Complete the function,

```
    def get_enron_addresses(addresses):
        ...
```

to find all of the Enron email addresses.

Inputs: `addresses` is a set of string email addresses.

Outputs: Return the subset of addresses belonging to Enron employees. These are the ones ending in `'@enron.com'`.

Additional notes: The input `addresses` set could be the empty set. In this case, your function should also return an empty set.

```
In [ ]:  ### Define demo inputs ###

         demo_addresses_ex4 = {
             'grampus@sunbeach.net',
             'petersm@energystore.net',
             'knowak@wpo.org',
             'lhgas@hoegh.no',
             'llightfoot@coral-energy.com',
             'sjohnsto@wutc.wa.gov',
             'amitava.dhar@enron.com',
             'jennifer.blay@enron.com',
             'lburns@hotmail.com',
             'steve.schneider@enron.com',
             'amy.fitzpatrick@enron.com'
         }
```

The demo included in the solution cell below should display the following output:

```
{'amitava.dhar@enron.com',
 'jennifer.blay@enron.com',
 'steve.schneider@enron.com',
 'amy.fitzpatrick@enron.com'}
```

> Since the return value is a set, it's possible the order when printed will differ from what you see above. However, the autograder will use set-comparison, so order should not matter.

```
In [ ]:   ### Exercise 4 solution
          def get_enron_addresses(addresses):
              ### BEGIN SOLUTION
              return {a for a in addresses if a[-10:] == '@enron.com'}
              ### END SOLUTION

          ### demo function call ###
          get_enron_addresses(demo_addresses_ex4)
```

The cell below will test your solution for Exercise 4. The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

```
In [ ]: ### test_cell_ex4
        ### BEGIN HIDDEN TESTS
        import dill
        import hashlib
        with open('resource/asnlib/public/hash_check.pkl', 'rb') as f:
            hash_check = dill.load(f)
        for fname in ['testers.py', '__init__.py', 'test_utils.py']:
            hash_check(f'tester_fw/{fname}', f'resource/asnlib/public/{fnam
        e}')
        del hash_check
        del dill
        del hashlib
        ### END HIDDEN TESTS
        from tester_fw.testers import Tester

        conf = {
            'case_file':'tc_4',
            'func': get_enron_addresses, # replace this with the function d
        efined above
            'inputs':{ # input config dict. keys are parameter names
                'addresses':{
                    'dtype':'set', # data type of param.
                    'check_modified':True,
                }
            },
            'outputs':{
                'output_0':{
                    'index':0,
                    'dtype':'set',
                    'check_dtype': True,
                    'check_col_dtypes': True, # Ignored if dtype is not df
                    'check_col_order': True, # Ignored if dtype is not df
                    'check_row_order': True, # Ignored if dtype is not df
                    'check_column_type': True, # Ignored if dtype is not df
                    'float_tolerance': 10 ** (-6)
                }
            }
        }
        tester = Tester(conf, key=b'66lGoH4BeY6guYRngj0DTLf5PgS5u60_N0vVqGy
        XN7w=', path='resource/asnlib/publicdata/')
        for _ in range(70):
            try:
                tester.run_test()
                (input_vars, original_input_vars, returned_output_vars, tru
        e_output_vars) = tester.get_test_vars()
            except:
                (input_vars, original_input_vars, returned_output_vars, tru
        e_output_vars) = tester.get_test_vars()
                raise

        ### BEGIN HIDDEN TESTS
        tester = Tester(conf, key=b'1VrZu8ZlwHw20ESTvKgqVXujnQVImp9nhaWtlHV
        uprc=', path='resource/asnlib/publicdata/encrypted/')
        for _ in range(20):
            try:
                tester.run_test()
                (input_vars, original_input_vars, returned_output_vars, tru
        e_output_vars) = tester.get_test_vars()
            except:
                (input_vars, original_input_vars, returned_output_vars, tru
```

```
    e_output_vars) = tester.get_test_vars()
            raise
### END HIDDEN TESTS
print('Passed! Please submit.')
```

RUN ME: `enron_addresses`

Using a correct implementation of `get_enron_addresses`, we can sweep the full address list and identify just those associated with Enron. The cell below loads the results of such a sweep. **Run this cell whether or not your implementation is correct.**

The object `enron_addresses` is a set of emails associated with Enron only. Each element is a pair `(msg_id, addr)`,

```
In [ ]: enron_addresses = load_obj_from_file('ex4_enron_addresses.dill')

        print(f"There are {len(enron_addresses):,} unique `@enron.com` emai
        l addresses in the archive, considering the `'From'`, `'To'`, `'C
        c'`, and `'Bcc'` fields.")
```

Part C: Data Cleaning and Preliminary Analysis

Let's clean the data and do a simple analysis before trying your colleague's tool.

Address maps

Using the functions developed above, suppose we convert the archive into the following abstract representation, called an **address map**.

- Each email message is assigned a unique integer ID, starting from 0.
- For each message, we record the unique Enron email addresses that appear in the `'From'`, `'To'`, `'Cc'`, and `'Bcc'` fields.

The following data structure, called the `address_map`, captures this representation:

```
In [ ]: address_map = load_obj_from_file('ex4_messages_to_address_fields.di
        ll')

        print(type(address_map))
        address_map[:5]
```

Observe that `address_map` is a list of dictionaries of string-set pairs. (What a mouthful!)

Specifically, each element of the `address_map` list represents an email message that is stored as a dictionary:

- each key of this dictionary is a field, which is one of the four strings `'From'`, `'To'`, `'Cc'`, and `'Bcc'`;
- the corresponding value is a Python set of email addresses.

Only addresses at `enron.com` are included in this representation.

Duplicates

Many of the email messages have some redundancy across their address fields. For example, consider the message at position 121 of the list:

```
In [ ]:  address_map[121]
```

Notice that `john.lavorato@enron.com` appears in the `'From'` field and the `'To'` field. Furthermore, everyone in the `'Cc'` field appears *again* in the `'Bcc'` field. In the next exercise, let's remove these duplicates.

Ex 5 (2 pt): `remove_duplicates`

Complete the function,

```python
def remove_duplicates(address_map):
    ...
```

so that it removes duplicates according to the following precedence rules:

1. If an address appears in `'Cc'`, then it should *not* appear in `'Bcc'`.
2. If an address appears in `'To'`, then it should *not* appear in `'Cc'` *nor* in `'Bcc'`.
3. If an address appears in `'From'`, then it should *not* appear in any of `'To'`, `'Cc'`, and `'Bcc'`.

Inputs: The input `address_map` is a list of dictionaries, where each dictionary maps a field name (`'From'`, `'To'`, `'Cc'`, or `'Bcc'`) to a set of email addresses.

Your task: Create a new list of dictionaries. It should have the same structure as the input, but all redundant addresses should be removed per the rules given above.

Outputs: Return a **new** list of dictionaries of string-set pairs with duplicates removed.

Additional notes and hints (IMPORTANT!):

1. You may assume every dictionary has all four fields. However, a field might be "empty," meaning its value is an empty set.
2. As with all exercises, be especially careful **NOT** to modify any of the input. You should return a new data structure. If you think an in-place solution is the most natural one to write, consider using `copy.deepcopy()` [(https://docs.python.org/3/library/copy.html)](https://docs.python.org/3/library/copy.html) as a first step.

```python
In [ ]:  ### Define demo inputs ###

         demo_address_map_ex5 = [
             {'From': {'jeffery.fawcett@enron.com'},
              'To': {'drew.fossum@enron.com'},
              'Cc': set(),
              'Bcc': set()},
             {'From': {'lloyd.will@enron.com'},
              'To': {'smith.day@enron.com'},
              'Cc': {'clint.dean@enron.com',
                     'jeffrey.miller@enron.com',
                     'jim.homco@enron.com',
                     'smith.day@enron.com',
                     'tom.may@enron.com'},
              'Bcc': {'clint.dean@enron.com',
                      'jeffrey.miller@enron.com',
                      'jim.homco@enron.com',
                      'smith.day@enron.com',
                      'tom.may@enron.com'}}
         ]
```

The demo included in the solution cell below should display the following output:

```
[{'Bcc': set(),
  'Cc': set(),
  'From': {'jeffery.fawcett@enron.com'},
  'To': {'drew.fossum@enron.com'}},
 {'Bcc': set(),
  'Cc': {'clint.dean@enron.com',
         'jeffrey.miller@enron.com',
         'jim.homco@enron.com',
         'tom.may@enron.com'},
  'From': {'lloyd.will@enron.com'},
  'To': {'smith.day@enron.com'}}]
```

```
In [ ]:   ### Exercise 5 solution
          def remove_duplicates(address_map):
              ### BEGIN SOLUTION
              from copy import deepcopy
              from itertools import combinations
              cleaned = deepcopy(address_map)
              for fields in cleaned:
                  for A, B in combinations(['Bcc', 'Cc', 'To', 'From'], 2):
                      fields[A] -= fields[B]
              return cleaned
              ### END SOLUTION


          ### demo function call ###
          demo_your_solution_ex5 = remove_duplicates(demo_address_map_ex5)
          pprint(demo_your_solution_ex5)
```

The cell below will test your solution for Exercise 5. The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

```
In [ ]: ### test_cell_ex5
        ### BEGIN HIDDEN TESTS
        import dill
        import hashlib
        with open('resource/asnlib/public/hash_check.pkl', 'rb') as f:
            hash_check = dill.load(f)
        for fname in ['testers.py', '__init__.py', 'test_utils.py']:
            hash_check(f'tester_fw/{fname}', f'resource/asnlib/public/{fnam
        e}')
        del hash_check
        del dill
        del hashlib
        ### END HIDDEN TESTS
        from tester_fw.testers import Tester

        conf = {
            'case_file':'tc_5',
            'func': remove_duplicates, # replace this with the function def
        ined above
            'inputs':{ # input config dict. keys are parameter names
                'address_map':{
                    'dtype':'list', # data type of param.
                    'check_modified':True,
                }
            },
            'outputs':{
                'output_0':{
                    'index':0,
                    'dtype':'list',
                    'check_dtype': True,
                    'check_col_dtypes': True, # Ignored if dtype is not df
                    'check_col_order': True, # Ignored if dtype is not df
                    'check_row_order': True, # Ignored if dtype is not df
                    'check_column_type': True, # Ignored if dtype is not df
                    'float_tolerance': 10 ** (-6)
                }
            }
        }
        tester = Tester(conf, key=b'66lGoH4BeY6guYRngj0DTLf5PgS5u60_N0vVqGy
        XN7w=', path='resource/asnlib/publicdata/')
        for _ in range(70):
            try:
                tester.run_test()
                (input_vars, original_input_vars, returned_output_vars, tru
        e_output_vars) = tester.get_test_vars()
            except:
                (input_vars, original_input_vars, returned_output_vars, tru
        e_output_vars) = tester.get_test_vars()
                raise

        ### BEGIN HIDDEN TESTS
        tester = Tester(conf, key=b'1VrZu8ZlwHw20ESTvKgqVXujnQVImp9nhaWtlHV
        uprc=', path='resource/asnlib/publicdata/encrypted/')
        for _ in range(20):
            try:
                tester.run_test()
                (input_vars, original_input_vars, returned_output_vars, tru
        e_output_vars) = tester.get_test_vars()
            except:
                (input_vars, original_input_vars, returned_output_vars, tru
```

```
        e_output_vars) = tester.get_test_vars()
                raise
### END HIDDEN TESTS
print('Passed! Please submit.')
```

RUN ME: `clean_address_map`

Using a correct implementation of `remove_duplicates`, we can sweep the address map and remove all duplicate addresses for each message. The cell below loads the results of such a sweep. **Run this cell whether or not your implementation is correct.**

The object `clean_address_map` is the cleaned (de-duplicated) version of `address_map`.

```
In [ ]: clean_address_map = load_obj_from_file('ex5_clean_address_map.dil
        l')

        print("* Recall the 'dirty' `address_map[121]`:\n")
        pprint(address_map[121])

        print("\n* Here is `clean_address_map[121]`:\n")
        pprint(clean_address_map[121])
```

Ex. 6 (2 pt): `count_addresses`

Suppose you are given an `address_map`, which is already cleaned (duplicates are removed). Complete the function,

```
def count_addresses(address_map):
    ...
```

so that it counts how many times each unique address appears in an email message.

Inputs: An `address_map`, i.e.,

- it is a *list* ...
- where each element `address_map[k]` is a *dictionary* ...
- where each key is a field, one of `'From'`, `'To'`, `'Cc'`, and `'Bcc'`, ...
- and each value is a *set* of email addresses (as strings).

Your task: For each unique email address, count how many messages it appears in.

Outputs: Construct a dictionary. The key should be one of the unique email addresses, and the value should be an integer count of the number of times that address appeared in any field of any message.

```
In [ ]:  ### Define demo inputs ###

         demo_address_map_ex6 = [
             # Message 0:
             {'From': {'a@enron.com'}, 'To': {'b@enron.com', 'c@enron.com'},
         'Cc': set(), 'Bcc': {'d@enron.com'}},
             # Message 1:
             {'From': {'b@enron.com'}, 'To': set(), 'Cc': set(), 'Bcc': {'d@
         enron.com'}},
             # Message 2:
             {'From': {'d@enron.com'}, 'To': {'a@enron.com'}, 'Cc': set(),
         'Bcc': {'e@enron.com'}},
         ]
```

The demo included in the solution cell below should display the following output (the comments explain the counts and are *not* part of your output):

```
{'a@enron.com': 2,      # Messages 0 and 2
 'b@enron.com': 2,      # Messages 0 and 1
 'c@enron.com': 1,      # Message 0
 'd@enron.com': 3,      # Messages 0, 1, and 2
 'e@enron.com': 1}      # Message 2
```

```
In [ ]:  ### Exercise 6 solution
         def count_addresses(address_map):
             ### BEGIN SOLUTION
             from collections import Counter
             counts = sum((Counter(addresses) for fields in address_map
                                               for addresses in fields.values
         ()),
                          start=Counter())
             return dict(counts)
             ### END SOLUTION


         ### demo function call ###
         count_addresses(demo_address_map_ex6)
```

The cell below will test your solution for Exercise 6. The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

In [ ]:
```python
### test_cell_ex6
### BEGIN HIDDEN TESTS
import dill
import hashlib
with open('resource/asnlib/public/hash_check.pkl', 'rb') as f:
    hash_check = dill.load(f)
for fname in ['testers.py', '__init__.py', 'test_utils.py']:
    hash_check(f'tester_fw/{fname}', f'resource/asnlib/public/{fnam
e}')
del hash_check
del dill
del hashlib
### END HIDDEN TESTS
from tester_fw.testers import Tester

conf = {
    'case_file':'tc_6',
    'func': count_addresses, # replace this with the function defin
ed above
    'inputs':{ # input config dict. keys are parameter names
        'address_map':{
            'dtype':'list', # data type of param.
            'check_modified':True,
        }
    },
    'outputs':{
        'output_0':{
            'index':0,
            'dtype':'dict',
            'check_dtype': True,
            'check_col_dtypes': True, # Ignored if dtype is not df
            'check_col_order': True, # Ignored if dtype is not df
            'check_row_order': True, # Ignored if dtype is not df
            'check_column_type': True, # Ignored if dtype is not df
            'float_tolerance': 10 ** (-6)
        }
    }
}
tester = Tester(conf, key=b'66lGoH4BeY6guYRngj0DTLf5PgS5u60_N0vVqGy
XN7w=', path='resource/asnlib/publicdata/')
for _ in range(70):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, tru
e_output_vars) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, tru
e_output_vars) = tester.get_test_vars()
        raise

### BEGIN HIDDEN TESTS
tester = Tester(conf, key=b'1VrZu8ZlwHw20ESTvKgqVXujnQVImp9nhaWtlHV
uprc=', path='resource/asnlib/publicdata/encrypted/')
for _ in range(20):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, tru
e_output_vars) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, tru
```

```
e_output_vars) = tester.get_test_vars()
        raise
### END HIDDEN TESTS
print('Passed! Please submit.')
```

RUN ME: `enron_address_counts`

Using a correct implementation of `count_addresses`, we can sweep the (cleaned) address map and tally the number of messages containing each email address. The cell below loads the results of such a sweep. **Run this cell whether or not your implementation is correct.**

The object `enron_address_counts` is the cleaned version of `address_map`.

```
In [ ]:  enron_address_counts = load_obj_from_file('ex6_address_counts.dil
         l')

         print("* A sample of counts:\n")
         pprint(dict(list(enron_address_counts.items())[:5]))
```

Part D: Interfacing with the Analysis Tool

From the address counts of Exercise 6, here are the most frequently occurring email addresses:

```
In [ ]:  print("Here are the Enron emails that occurred most commonly across
         messages:")
         sorted(enron_address_counts.items(), key=lambda ac: ac[1], reverse=
         True)[:10]
```

```
In [ ]:  enum_filter(lambda e: e[0] == 'kenneth.lay@enron.com', list(enron_a
         ddress_counts.items()))
```

If you know anything about the Enron case, you'll might notice these email addresses do not necessarily correspond with any notion of "importance" we might have.

For example, `enron.announcements@enron.com` is tied for second place, having appeared 16 times; however, it does not seem like it would be an important email address. And two people who later turned out to be major figures in the Enron controversy—`jeff.skilling@enron.com` and `kenneth.lay@enron.com`—appear nowhere near the top 10 (they are around 400). So mining for important figures based on email counts, at least in this sample, does not seem especially meaningful.

Can we do better? It's time to try your colleague's tool.

Preliminaries: A "coordinate" representation

To use this tool, you need to convert your data into a particular "coordinate" representation:

1. Start with the email message-to-address representation, such as `clean_address_map` (Ex. 5)
2. Assign each message an integer ID, starting from 0. Since the address map is a list, a natural ID for a message is simply its index-position in the list.
3. Assign each email address an integer ID, starting from 0. We will do this for you.
4. For each (message ID, address ID) pair, assign a "score." You will do that in the next (and last) exercise, Ex. 7.

For address IDs (item 3 just above), the dictionary `enron_addr_to_id` maps every Enron email address to an integer ID. The dictionary `enron_id_to_addr` is the inverse computed from a correct Ex. 1: it maps the integer ID back to an address.

```
In [ ]: enron_addr_to_id = cse6040.utils.load_obj_from_file('ex4_enron_addr
        _to_id.dill')
        enron_id_to_addr = cse6040.utils.load_obj_from_file('ex4_enron_id_t
        o_addr.dill')


        print(enron_addr_to_id['mike.coleman@enron.com'], "<==>", enron_id_
        to_addr[987])
        print(enron_addr_to_id['jeff.skilling@enron.com'], "<==>", enron_id
        _to_addr[865])
```

Lastly, each (message ID, address ID) pair is called a "coordinate," and its score is its "weight." To plug these into the analysis tool, we need to store these coordinates as a dictionary keyed on `(message ID, address ID)` pairs mapping to the score (weight).

Ex. 7 (3 pts): `gen_coords`

Complete the function,

```
def gen_coords(address_map, address_to_id, address_counts):
    ...
```

so that it generates and returns the coordinate representation.

Inputs:

- `address_map`: A (clean) address map, stored as a list of dictionaries whose keys are strings (e.g., `'From'`, `'To'`) and values are sets of email addresses. (See Ex. 5)
- `address_to_id`: A dictionary whose key is an email address and whose value is its unique integer ID.
- `address_counts`: A dictionary that maps each address (key) to a count (the number of messages in which it appears).

Your task:

- Start with an empty output dictionary, which will hold the coordinate representation.
- Visit each message of the address map, and for each message, visit each address.
- Assign the address a score, computed as described below.
- Add the coordinate and score, (`message ID, address ID, score`), to the output dictionary using (`message ID, address ID`) as a key and `score` as the value.

To compute the score for an address, call it a, use the following scheme:

- Let M be the total number of messages.
- Let n_a be the total number of messages in which address a appears, obtainable from `address_counts`.
- Let β be a field-specific weight. If address a is in the `'From'` field, use a weight of $\beta = 1.0$. For the `'To'` field, use $\beta = 0.5$. For the `'Cc'` field, use $\beta = 0.25$. And for the `'Bcc'` field, use $\beta = 0.75$.
- If $n_a > 0$, then calculate the score as $\dfrac{1}{\log\left(\dfrac{M+1}{\beta \cdot n_a}\right)}$.

Outputs: Your function will return a list of triples, constructed as sketched above.

Additional notes:

- You may assume the address map is "clean," meaning an address can only appear in one field of a given message.
- The $\log$ function is the natural logarithm.

```
In [ ]:  ### Define demo inputs ###

         demo_address_map_ex7 = [
             {'From': set(), 'To': {'judy.hernandez@enron.com'}, 'Cc': set
         (), 'Bcc': set()},
             {'From': {'juan.hernandez@enron.com'}, 'To': {'rudy.acevedo@enr
         on.com'}, 'Cc': set(), 'Bcc': set()},
             {'From': {'pete.davis@enron.com'}, 'To': set(), 'Cc': {'bert.me
         yers@enron.com',
                                                                     'bill.wi
         lliams.iii@enron.com',
                                                                     'craig.d
         ean@enron.com',
                                                                     'dporter
         3@enron.com',
                                                                     'eric.li
         nder@enron.com',
                                                                     'geir.so
         lberg@enron.com',
                                                                     'jbryson
         @enron.com',
                                                                     'leaf.ha
         rasin@enron.com',
                                                                     'mark.gu
         zman@enron.com',
                                                                     'monika.
         causholli@enron.com',
                                                                     'ryan.sl
         inger@enron.com',
                                                                     'steven.
         merris@enron.com'}, 'Bcc': set()}
         ]
         demo_address_to_id_ex7 = enron_addr_to_id.copy()
         demo_address_counts_ex7 = enron_address_counts.copy()
```

The demo included in the solution cell below should display the following output:

```
{(0, 53): 2.127643145234443,
 (1, 828): -2.4663034623764313,
 (1, 548): 2.127643145234443,
 (2, 238): -2.4663034623764313,
 (2, 602): 1.0195454478232662,
 (2, 549): 1.0195454478232662,
 (2, 399): 1.0195454478232662,
 (2, 1364): 1.0195454478232662,
 (2, 362): 0.8597337435263985,
 (2, 1075): 1.0195454478232662,
 (2, 1050): 1.2096599965665937,
 (2, 956): 1.0195454478232662,
 (2, 810): 1.4426950408889634,
 (2, 689): 1.0195454478232662,
 (2, 1065): 1.0195454478232662,
 (2, 1053): 1.0195454478232662}
```

```
In [ ]: ### Exercise 7 solution
        def gen_coords(address_map, address_to_id, address_counts):
            from math import log
            ### BEGIN SOLUTION
            FIELD_MULTIPLIER = {'From': 1.0, 'To': 0.5, 'Cc': 0.25, 'Bcc':
        0.75}
            n = len(address_map)
            coords = {}
            for eid, fields in enumerate(address_map):
                for field, addresses in fields.items():
                    xf = FIELD_MULTIPLIER[field]
                    for address in addresses:
                        aid = address_to_id[address]
                        ni = address_counts[address]
                        score = 0 if ni == 0 else 1.0 / log((n+1) / (ni*x
        f))
                        coords[(eid, aid)] = score
            return coords
            ### END SOLUTION

        ### demo function call ###
        gen_coords(demo_address_map_ex7, demo_address_to_id_ex7, demo_addre
        ss_counts_ex7)
```

The cell below will test your solution for Exercise 7. The testing variables will be available for debugging under the following names in a dictionary format.

- `input_vars` - Input variables for your solution.
- `original_input_vars` - Copy of input variables from prior to running your solution. These *should* be the same as `input_vars` - otherwise the inputs were modified by your solution.
- `returned_output_vars` - Outputs returned by your solution.
- `true_output_vars` - The expected output. This *should* "match" `returned_output_vars` based on the question requirements - otherwise, your solution is not returning the correct output.

```
In [ ]:  ### test_cell_ex7
         ### BEGIN HIDDEN TESTS
         import dill
         import hashlib
         with open('resource/asnlib/public/hash_check.pkl', 'rb') as f:
             hash_check = dill.load(f)
         for fname in ['testers.py', '__init__.py', 'test_utils.py']:
             hash_check(f'tester_fw/{fname}', f'resource/asnlib/public/{fnam
         e}')
         del hash_check
         del dill
         del hashlib
         ### END HIDDEN TESTS
         from tester_fw.testers import Tester

         conf = {
             'case_file':'tc_7',
             'func': gen_coords, # replace this with the function defined ab
         ove
             'inputs':{ # input config dict. keys are parameter names
                 'address_map':{
                     'dtype':'list', # data type of param.
                     'check_modified':True,
                 },
                 'address_to_id':{
                     'dtype':'list',
                     'check_modified': True
                 },
                 'address_counts':{
                     'dtype':'list',
                     'check_modified': True
                 }
             },
             'outputs':{
                 'output_0':{
                     'index':0,
                     'dtype':'dict',
                     'check_dtype': True,
                     'check_col_dtypes': True, # Ignored if dtype is not df
                     'check_col_order': True, # Ignored if dtype is not df
                     'check_row_order': True, # Ignored if dtype is not df
                     'check_column_type': True, # Ignored if dtype is not df
                     'float_tolerance': 10 ** (-6)
                 }
             }
         }
         tester = Tester(conf, key=b'66lGoH4BeY6guYRngj0DTLf5PgS5u60_N0vVqGy
         XN7w=', path='resource/asnlib/publicdata/')
         for _ in range(70):
             try:
                 tester.run_test()
                 (input_vars, original_input_vars, returned_output_vars, tru
         e_output_vars) = tester.get_test_vars()
             except:
                 (input_vars, original_input_vars, returned_output_vars, tru
         e_output_vars) = tester.get_test_vars()
                 raise

         ### BEGIN HIDDEN TESTS
         tester = Tester(conf, key=b'1VrZu8ZlwHw20ESTvKgqVXujnQVImp9nhaWtlHV
```

```
uprc=', path='resource/asnlib/publicdata/encrypted/')
for _ in range(20):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, tru
e_output_vars) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, tru
e_output_vars) = tester.get_test_vars()
        raise
### END HIDDEN TESTS
print('Passed! Please submit.')
```

Fin! *(End of Exam)*

Congratulations, you have reached the end of this exam. **Do not forget to submit your work!**

Indeed, if all your exericses are working, then you've successfully translated the raw input data into a form that your colleague's tool can analyze!

Epilogue. Indeed, if all your exericses are working, then you've successfully translated the raw input data into a form that your colleague's tool can analyze!

While we will discuss the method later in the semester, as a preview, observe plugging a correct coordinate represenation into this tool produces the following "top e-mail addresses" shown in descending order of some kind of abstract score:

```
In [ ]:  enron_coords = load_obj_from_file('ex7_enron_coords.dill')
         rows = [i for i, _ in enron_coords.keys()]
         cols = [j for _, j in enron_coords.keys()]
         vals = list(enron_coords.values())
         #viz_coords(rows, cols, vals, title='Address ID', ylabel='Message I
         D')

         _, aidrs = rank_items(rows, cols, vals, m=len(clean_address_map), n
         =len(enron_addr_to_id))
         [(score, aid, enron_id_to_addr[aid], enron_address_counts[enron_id_
         to_addr[aid]]) for aid, score in aidrs[:10]]
```