`Midterm 2, Fall 2022: Capturing Data Changes for Slowly Changing Dimensions`

Version 1.0.0

Version History

1.0.0

- Initial release.

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

Topics, number of exercises: This problem builds on your knowledge of `working with tabular data`. It has **9** exercises, numbered 0 to **8**. There are **19** available points. However, to earn 100% the threshold is **12** points. (Therefore, once you hit **12** 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 your 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: **3** point(s)
- Exercise 1: **2** point(s)
- Exercise 2: **1** point(s)
- Exercise 3: **2** point(s)
- Exercise 4: **3** point(s)
- Exercise 5: **3** point(s)
- Exercise 6: **1** point(s)
- Exercise 7: **2** point(s)
- Exercise 8: **2** point(s)

Final reminders:

- Submit after **every exercise**
- 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

Scenario (don't dwell on this)

You have just been hired by the hot new startup **Spot-i-flix-ify** (this is a fictional company which will offer video and audio streaming services) as a Data Scientist. This is a small startup so you have to "wear many different hats," so to speak. Your first task on the job is to set up their data warehousing so that they can capture a historical record of their operations for analysis later. The operational database (which someone else has already set up) only contains the current state of the operation to maintain maximum efficiency while performing tasks like adding new customers, changing services, applying promotions, etc. It will not contain any history and is not intended have complex queries run against it.

While this is a fictional company and simulation data, **there is a real-world use case** for the processes developed in this notebook.

Data (Don't dwell on this. The structures you are working with will be explained in each exercise.)

You are working with four tables:

- `customers` - Center of the "star" schema. Primary Key: `id`.
  - `customers.id` - a unique identifier for an individual customer.
  - `customers.paid` - ('True'|'False') - indicates whether a customer has paid their bill for their upcoming month of service.
- `prices` - The prices of the services offered by **Spot-i-flix-ify**. Primary Key: `service, tier, promo`
  - `prices.service` - Name of the sevice
  - `prices.tier` - Tier of the service. A service can be offered in several tiers. Higher tiers give customers more features.
  - `prices.promo` - Promotion which can be applied to a service/tier combination to offer a discount to customers.
  - `prices.price` - The price of a particular service/tier/promo combination.
- `services` - Services which each customer is subscribed. Primary Key: `cust_id, service`; Foreign Keys: `cust_id` references `customers.id`, (`service, tier`) references (`prices.service, prices.tier`)
  - `services.cust_id` - id of the customer associated with this subscription.
  - `services.service` - name of service associated with this subscription.
  - `services.tier` - tier of service associated with a subscription.
- `promos` - All promos which a customer has ever used for any service. This historical information is required to prevent customers from using the same promo twice.
  - `cust_id` - id of a customer associated with a particular record.
  - `service` - service which a customer used a particular promo on.
  - `promo` - name of the promo associated with a particular record.

- `time_left` - number of remaining months for which the promo price is applied to a service for the customer. If all promos associated with a `cust_id`/`service` pair have 0 months left. The "base" promo is applied to the customer for that service.

On data types

These tables are made available to you in a staging environment as Pandas `DataFrame` objects. **All columns in all of the DataFrames are strings (even the columns where you would expect other data types)**.

On SQL

We used Pandas exclusively in developing this exam, however some exercise are solvable using SQL. In the cell below we have included the function `dfs_to_conn` which can be used to create in-memory database connections. If you pass in a dictionary mapping table names to DataFrames, `dfs_to_conn` will return a sqlite 3 connection with all of the data in the DataFrames available under the names given as keys. You are also free to write to the in-memory database by creating tables, inserting/deleting/updating records, etc. Anything that SQLite allows should work!

Example: `my_df = pd.DataFrame({'A':[1,2,3], 'B': [4,5,6], 'C':['x', 'y', 'z']}) print(my_df) #   A B C # 0 1 4 x # 1 2 5 y # 2 3 6 z conn = dfs_to_conn({'my_table': my_df}) cur = conn.cursor() cur.execute('select A, B, C from my_table') result = cur.fetchall() conn.close() print(result) # list of tuples, each tuple is a row #[(1, 4, 'x'), (2, 5, 'y'), (3, 6, 'z')]`

```
### Global Imports
###
### AUTOGRADER TEST - DO NOT REMOVE
###
import pandas as pd
import time
overall_start = time.time()

def dfs_to_conn(conn_dfs, index=False):
    import sqlite3
    conn = sqlite3.connect(':memory:')
    for table_name, df in conn_dfs.items():
        df.to_sql(table_name, conn, if_exists='replace', index=index)
    return conn
```

Exercise 0 - (**3** Points):

Motivation (Don't dwell on this):
The business logic behind the database requires all promos which a customer has ever participated in be stored in the "live" business data in order to prevent a customer from using the same promotion twice. However, for keeping the historical record, the data consumers (i.e.

your bosses) are only interested in seeing which promotion is actually being applied to a customer's bill. We need to extract this information from the `promos` table.

Requirements:
Define `get_active_promos(promos)`. The input `promos` is a DataFrame with columns as described in the `promos` table above. These are the columns/descriptions:

- `cust_id` - id of a single customer.
- `service` - a service where the customer has participated in a promo.
- `promo` - name of a promo in which a customer has participated for the associated service.

- `time_left` - the time the customer has left on the promo.

Note: There may be many records in `promos` with the same `cust_id`/`service` combination. However, at most one such record will have a `time_left` value other than `'0'`.

Your function should return a new DataFrame, `active_promos` derived from `promos` with the schema outlined below. There should be exactly 1 record in `active_promos` for each unique combination of `cust_id`/`service` found in `promos`.

`active_promos` - the promotion which is actually applied to each customer for a particular service

- `'cust_id'` - identifies an individual customer.
- `'service'` - identifies a service for which the customer has an active promotion. **The customer may not actually be subscribed to the service!**
- `'promo'` - the active promo for the `cust_id`/`service` pair.
  - If `'time_left'` is '0' for all records associated with the `cust_id`/`service` pair in `promos`, this column should have a value of `'base'`.
  - If there is a record associated with a non-zero `'time_left'`, this column should have the `'promo'` from that record.

```
### Define demo inputs

demo_promos_ex0 = pd.DataFrame([
{'cust_id': '0', 'promo': 'promo_1', 'service': 'audio', 'time_left':
'5'},
{'cust_id': '0', 'promo': 'promo_3', 'service': 'audio', 'time_left':
'0'},
{'cust_id': '0', 'promo': 'base', 'service': 'audio', 'time_left':
'0'},
{'cust_id': '0', 'promo': 'promo_3', 'service': 'video', 'time_left':
'0'},
{'cust_id': '0', 'promo': 'promo_1', 'service': 'video', 'time_left':
'0'},
{'cust_id': '0', 'promo': 'base', 'service': 'video', 'time_left':
'0'},
{'cust_id': '1', 'promo': 'promo_3', 'service': 'audio', 'time_left':
'0'},
```

```
{'cust_id': '1', 'promo': 'promo_1', 'service': 'audio', 'time_left':
'0'},
{'cust_id': '1', 'promo': 'base', 'service': 'audio', 'time_left':
'0'},
{'cust_id': '1', 'promo': 'promo_3', 'service': 'video', 'time_left':
'0'},
{'cust_id': '1', 'promo': 'promo_1',  'service': 'video',
'time_left': '4'},
{'cust_id': '1', 'promo': 'base', 'service': 'video', 'time_left':
'0'}]
)
demo_promos_ex0
```

	cust_id	promo	service	time_left
0	0	promo_1	audio	5
1	0	promo_3	audio	0
2	0	base	audio	0
3	0	promo_3	video	0
4	0	promo_1	video	0
5	0	base	video	0
6	1	promo_3	audio	0
7	1	promo_1	audio	0
8	1	base	audio	0
9	1	promo_3	video	0
10	1	promo_1	video	4
11	1	base	video	0

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

	cust_id	promo	service
0	0	promo_1	audio
1	0	base	video
2	1	base	audio
3	1	promo_1	video

```
### Exercise 0 solution
def get_active_promos(promos):
    def active_helper(group):
        max_tl = group['time_left'].astype(int).max()
        idx = group['time_left'].astype(int).idxmax()
        row_dict = {
            'cust_id': group.loc[idx, 'cust_id'],
            'service': group.loc[idx, 'service'],
            'promo': group.loc[idx, 'promo'],
        }
        if max_tl == 0:
            row_dict['promo'] = 'base'
        return pd.Series(row_dict)
    return promos.groupby(['cust_id', 'service'],
```

```
as_index=False).apply(active_helper)

### demo function call
print(get_active_promos(demo_promos_ex0))
```

```
   cust_id service    promo
0        0   audio  promo_1
1        0   video     base
2        1   audio     base
3        1   video  promo_1
```

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.

```
### test_cell_ex0
exercise_start = time.time()
###
### AUTOGRADER TEST - DO NOT REMOVE
###
from tester_fw.testers import Tester

conf = {
    'case_file':'tc_0',
    'func': get_active_promos, # replace this with the function
defined above
    'inputs':{ # input config dict. keys are parameter names
        'promos':{
            'dtype':'df', # data type of param.
            'check_modified':True,
        }
    },
    'outputs':{
        'output_0':{
            'index':0,
            'dtype':'pd.DataFrame',
            'check_dtype': True,
            'check_col_dtypes': True, # Ignored if dtype is not df
            'check_col_order': False, # Ignored if dtype is not df
            'check_row_order': False, # 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'6IRWMcPsVIAZqzDJnPgv_MfUZsxqo4Utjm2Favidv-
A=', path='resource/asnlib/publicdata/')
for _ in range(70):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars,
true_output_vars) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars,
true_output_vars) = tester.get_test_vars()
        raise

###
### AUTOGRADER TEST - DO NOT REMOVE
###
exercise_end = time.time()
print(f"This test executed in {(pd.to_datetime(exercise_end, unit='s')
- pd.to_datetime(exercise_start, unit='s')).seconds} seconds")
print('Passed! Please submit.')
```

```
This test executed in 1 seconds
Passed! Please submit.
```

Exercise 1 - (**2** Points):

Motivation (Don't dwell on this):
To allow for faster updates, the business schema is somewhat normalized. To get the full picture of each service a customer is subscribed to, we have to use the relationships between the keys in each table to piece everything together. For our historical record, the requirement is a little different. We will only add records to the history. Recording the history of each normalized table is not desired as it will require more engineering to piece together. We will instead de-normalize the tables and then record the history of our de-normalized result.

Requirements:
Define the function `denormalize(customers, services, active_promos, prices)` which takes the DataFrame inputs which have the same structure as those in the introduction and from the result of exercise 0. See the data model diagram for the relationships between the 4 tables.

data_model

Your function should return a DataFrame `df` which contains the following columns:

- `id` - identifies a particular customer (from `customers`)
- `paid` - ('True'|'False') indicating whether the customer `id` has paid their bill (from `customers`)

- `service` - a service which a customer is subscribed. There should be one record for each unique `id`/`service` pair (from `services`)
- `tier` - tier of a service for the `id`/`service` pair. (from `services`)
- `promo` - promo being applied to the `id`/`service` pair. (from `active_promos`)
  - Remember that a record existing with a `cust_id`/`service` combination in `active_promos` *does not imply the customer is subscribed to that service*.
- `price` - price charged for the `id`/`service` pair (from `prices`)

You can accomplish this task using a series of "left" merges.

```
### Define demo inputs

demo_customers_ex1 = pd.DataFrame({'id': {0: '0', 1: '1'}, 'paid': {0:
'True', 1: 'False'}})
demo_active_promos_ex1 = pd.DataFrame({'cust_id': {0: '0', 1: '0', 2:
'1', 3: '1'},
 'service': {0: 'audio', 1: 'video', 2: 'audio', 3: 'video'},
 'promo': {0: 'intro', 1: 'base', 2: 'base', 3: 'intro'}})
demo_prices_ex1 = pd.DataFrame(
    {'service': {0: 'audio',  1: 'audio',  2: 'audio',  3: 'audio',
4: 'video',  5: 'video',  6: 'video',  7: 'video'},
    'tier': {0: '1', 1: '1', 2: '2', 3: '2', 4: '1', 5: '1', 6: '2',
7: '2'},
    'promo': {0: 'base', 1: 'intro', 2: 'base', 3: 'intro', 4: 'base',
5: 'intro', 6: 'base', 7: 'intro'},
    'price': {0: '8.99', 1: '5.99', 2: '12.99', 3: '9.99', 4: '10.99',
5: '8.99', 6: '15.99', 7: '11.99'}})
demo_services_ex1 = pd.DataFrame({'cust_id': {0: '0', 1: '1', 2: '1'},
 'service': {0: 'audio', 1: 'video', 2: 'audio'},
 'tier': {0: '1', 1: '1', 2: '2'}})

print('customers')
print(demo_customers_ex1)
print()
print('services')
print(demo_services_ex1)
print()
print('active_promos')
print(demo_active_promos_ex1)
print()
print('prices')
print(demo_prices_ex1)
```

```
customers
  id   paid
0  0   True
1  1  False

services
```

```
  cust_id service tier
0       0   audio    1
1       1   video    1
2       1   audio    2

active_promos
  cust_id service  promo
0       0   audio  intro
1       0   video   base
2       1   audio   base
3       1   video  intro

prices
  service tier  promo  price
0   audio    1   base   8.99
1   audio    1  intro   5.99
2   audio    2   base  12.99
3   audio    2  intro   9.99
4   video    1   base  10.99
5   video    1  intro   8.99
6   video    2   base  15.99
7   video    2  intro  11.99
```

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

```
  id   paid service tier  promo  price
0  0   True   audio    1  intro   5.99
1  1  False   video    1  intro   8.99
2  1  False   audio    2   base  12.99
```

```
### Exercise 1 solution
def denormalize(customers, services, active_promos, prices):
    return customers.merge(services, left_on='id', right_on='cust_id',
how='left')\
        .merge(active_promos, on=['cust_id', 'service'], how='left')\
        .merge(prices, on=['service', 'tier', 'promo'], how='left')\
        .drop('cust_id', axis=1)

demo_ex1_output = denormalize(demo_customers_ex1, demo_services_ex1,
demo_active_promos_ex1, demo_prices_ex1)
print(demo_ex1_output)
```

```
  id   paid service tier  promo  price
0  0   True   audio    1  intro   5.99
1  1  False   video    1  intro   8.99
2  1  False   audio    2   base  12.99
```

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.

```
### test_cell_ex1
exercise_start = time.time()
###
### AUTOGRADER TEST - DO NOT REMOVE
###
from tester_fw.testers import Tester

conf = {
    'case_file':'tc_1',
    'func': denormalize, # replace this with the function defined
above
    'inputs':{ # input config dict. keys are parameter names
        'customers':{
            'dtype':'pd.DataFrame', # data type of param.
            'check_modified':True,
        },
        'services':{
            'dtype':'pd.DataFrame', # data type of param.
            'check_modified':True,
        },
        'active_promos':{
            'dtype':'pd.DataFrame', # data type of param.
            'check_modified':True,
        },
        'prices':{
            'dtype':'pd.DataFrame', # data type of param.
            'check_modified':True,
        }
    },
    'outputs':{
        'output_0':{
            'index':0,
            'dtype':'',
            'check_dtype': True,
            'check_col_dtypes': True, # Ignored if dtype is not df
            'check_col_order': False, # Ignored if dtype is not df
            'check_row_order': False, # 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'6IRWMcPsVIAZqzDJnPgv_MfUZsxqo4Utjm2Favidv-
A=', path='resource/asnlib/publicdata/')
for _ in range(70):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars,
true_output_vars) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars,
true_output_vars) = tester.get_test_vars()
        raise

###
### AUTOGRADER TEST - DO NOT REMOVE
###
exercise_end = time.time()
print(f"This test executed in {(pd.to_datetime(exercise_end, unit='s')
- pd.to_datetime(exercise_start, unit='s')).seconds} seconds")
print('Passed! Please submit.')
```

```
This test executed in 5 seconds
Passed! Please submit.
```

Exercise 2 - (1 Points):

Motivation (Don't dwell on this):
The business in interested in determining the revenue generated from its customers (go figure!). After de-normalizing this is pretty easy to calculate. All we have to do is take the sum of the `price` column!

Requirements:

Define the function `get_revenue(df)`.

The input `df` is a DataFrame with the same structure as the result from exercise 1. Return the total of the `'price'` column. Recall from the intro that all of the data fields are strings, so you will have to explicitly cast to a `float` before computing the total. **Round the result to 2 decimal places**.

```
### Define demo inputs

demo_df_ex2 = pd.DataFrame({'id': {0: '0', 1: '2', 2: '2', 3: '3', 4:
'4'},
 'paid': {0: 'True', 1: 'True', 2: 'True', 3: 'True', 4: 'True'},
 'service': {0: 'audio', 1: 'video', 2: 'audio', 3: 'audio', 4:
'video'},
 'tier': {0: '1', 1: '2', 2: '2', 3: '1', 4: '1'},
 'promo': {0: 'base', 1: 'base', 2: 'base', 3: 'base', 4: 'base'},
```

```
 'price': {0: '8.99', 1: '15.99', 2: '12.99', 3: '8.99', 4: '10.99'}})

print(demo_df_ex2)
```

```
  id  paid service tier promo  price
0  0  True   audio    1  base   8.99
1  2  True   video    2  base  15.99
2  2  True   audio    2  base  12.99
3  3  True   audio    1  base   8.99
4  4  True   video    1  base  10.99
```

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

```
57.95

### Exercise 2 solution
def get_revenue(df):
    ###
    return round(df['price'].astype(float).sum(), 2)
    ###

print(get_revenue(demo_df_ex2))
```

```
57.95
```

The cell below will test your solution for Exercise 2. 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.

```
### test_cell_ex2
exercise_start = time.time()
###
### AUTOGRADER TEST - DO NOT REMOVE
###
from tester_fw.testers import Tester

conf = {
    'case_file':'tc_2',
    'func': get_revenue, # replace this with the function defined
above
    'inputs':{ # input config dict. keys are parameter names
        'df':{
```

```
            'dtype':'pd.DataFrame', # data type of param.
            'check_modified':True,
        }
    },
    'outputs':{
        'output_0':{
            'index':0,
            'dtype':'float',
            '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'6IRWMcPsVIAZqzDJnPgv_MfUZsxqo4Utjm2Favidv-
A=', path='resource/asnlib/publicdata/')
for _ in range(70):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars,
true_output_vars) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars,
true_output_vars) = tester.get_test_vars()
        raise

###
### AUTOGRADER TEST - DO NOT REMOVE
###
exercise_end = time.time()
print(f"This test executed in {(pd.to_datetime(exercise_end, unit='s')
- pd.to_datetime(exercise_start, unit='s')).seconds} seconds")
print('Passed! Please submit.')
```

```
This test executed in 1 seconds
Passed! Please submit.
```

Capturing Data Changes (feel free to skip reading this.)

We are going to store the history by using "type-2" journaling. This process involves scanning the business data periodically and keeping track of the first and last dates which a record existed in a particular form.

To do this we rely on the assumption that there is a "key" which identifies a particular record in the business data. The key will never change, and it can be either a single column or a combination of multiple columns. In this application it is the `id` and `service` columns. All non-key columns are subject to change. We need multiple versions of the each record, so in our

journal we add columns to track the "effective date" and the "expiration date" of each version. The effective date is when the record first existed in a particular form, and the expiration date is the last date when the record existed in a particular form. By convention, the expiration date for any records which currently exist in the business data (i.e. "active records") will be '9999-12-31' (the maximum date that can be represented in YYYY-MM-DD).

We update the journal as follows:

- Key is found in business data which does not exist in the journal (new record)
  - Add the record to the journal with the effective date as the current snapshot date and expiration date as '9999-12-31'.
- Non-key columns are changed in the business data for a key which already exists in the journal (changed record)
  - Set the expiration date for the active record in the journal to 1 day prior to the current snapshot date.
  - Add the current record in the business data to the journal with the effective date as the current snapshot date and expiration date as '9999-12-31'.
- A key which has an active record in the journal no longer exists in the business data (deleted record)
  - Set the expiration date for the active record in the journal to 1 day prior to the current snapshot date.

There will be exactly one record in the journal with a particular key and effective date, and there will be exactly one record with a particular key and expiration date. We can re-create a snapshot of the business data for a particular date in the past by filtering the journal to only include records where that date is inclusively between the effective and expiration dates.

The next several exercises will break down the process into digestable bits, so don't feel overwhelmed if you don't fully grasp this concept.

Exercise 3 - (**2** Points):

Motivation (don't dwell on this):

The first task in our journaling process is to identify which records in the existing journal are active and which records are not. We will do so by checking the `'exp_dt'` column. All records with `'9999-12-31'` as their expiration date are considered active. We will be rebuilding the entire journal, so we need to partition the existing journal into active and not-active records and return both parts. The active records will be compared with the business data, and the inactive records will be included in the updated journal without modification. Additionally, on the initial load, there will not be an existing journal, so we will need to create it based on the data being loaded and the desired audit columns.

Requirements:
Define `partition_journal(df, audit_cols, existing_journal=None)`.

- The input `df` is a DataFrame - we do not care about it's structure.
- The input `audit_cols` is a `list` of strings. These are the names of audit columns used to track history in the journal. `audit_cols` will always include the strings `'eff_dt'` and `'exp_dt'`.

- The optional input `existing_journal` is a DataFrame or `None`. If `existing_journal` is not `None` it will have all of the columns in `df` and all of the `audit_cols` as its columns.

Your function should do the following:

- If `existing_journal` is `None`, create an empty DataFrame which has all of the columns in `df` and all of the `audit_cols` as its columns. This empty DataFrame will be used in the subsequent operations.
- Create `historical_journal` which is a DataFrame containing all rows of `existing_journal` where `'exp_dt'` is something other than `'9999-12-31'`.
- Create `active_journal` which is a DataFrame containing all rows of `existing_journal` where `'exp_dt'` is `'9999-12-31'`.
- Return the tuple `(historical_journal, active_journal)` - If the `existing_journal` was newly created these will be two empty DataFrames with all columns present in `df` and all of the `audit_cols`.

```
### Define demo inputs

demo_df_ex3 = pd.DataFrame({'id': {0: '1', 1: '2', 2: '2'},
 'paid': {0: 'True', 1: 'True', 2: 'True'},
 'service': {0: 'audio', 1: 'video', 2: 'audio'},
 'tier': {0: '1', 1: '2', 2: '2'},
 'promo': {0: 'base', 1: 'base', 2: 'base'},
 'price': {0: '8.99', 1: '15.99', 2: '12.99'}})
demo_existing_journal_ex3 = pd.DataFrame(
    {'id': {667: '0', 668: '1', 669: '2', 670: '2', 671: '3', 672:
'3', 673: '4', 9: '3', 10: '3', 17: '0', 1881: '1', 1882: '2', 1883:
'2', 1884: '4'},
    'paid': {667: 'True', 668: 'True', 669: 'True', 670: 'True', 671:
'True', 672: 'True', 673: 'True', 9: 'False', 10: 'False', 17: 'True',

            1881: 'True', 1882: 'True', 1883: 'True', 1884: 'True'},
    'service': {667: 'video', 668: 'audio', 669: 'video', 670:
'audio', 671: 'video', 672: 'audio', 673: 'audio', 9: 'video', 10:
'audio', 17: 'video',
            1881: 'audio', 1882: 'video', 1883: 'audio', 1884:
'audio'},
    'tier': {667: '2', 668: '1', 669: '2', 670: '2', 671: '1', 672:
'1', 673: '1', 9: '1', 10: '1', 17: '2', 1881: '1', 1882: '2', 1883:
'2', 1884: '1'},
    'promo': {667: 'intro', 668: 'intro', 669: 'intro', 670: 'intro',
671: 'intro', 672: 'intro', 673: 'intro', 9: 'base', 10: 'base', 17:
'base',
            1881: 'base', 1882: 'base', 1883: 'base', 1884: 'base'},
    'price': {667: '11.99', 668: '5.99', 669: '11.99', 670: '9.99',
671: '8.99', 672: '5.99', 673: '5.99', 9: '10.99', 10: '8.99', 17:
'15.99',
            1881: '8.99', 1882: '15.99', 1883: '12.99', 1884: '8.99'},
    'eff_dt': {667: '2018-02-01', 668: '2018-02-01', 669: '2018-02-
```

```
01', 670: '2018-02-01', 671: '2018-02-01', 672: '2018-02-01', 673:
'2018-02-01',
            9: '2018-08-01', 10: '2018-08-01', 17: '2018-08-01', 1881:
'2018-08-01', 1882: '2018-08-01', 1883: '2018-08-01', 1884: '2018-08-
01'},
    'exp_dt': {667: '2018-07-31', 668: '2018-07-31', 669: '2018-07-
31', 670: '2018-07-31', 671: '2018-07-31', 672: '2018-07-31', 673:
'2018-07-31',
            9: '2018-08-31', 10: '2018-08-31', 17: '2019-02-28', 1881:
'9999-12-31', 1882: '9999-12-31', 1883: '9999-12-31', 1884: '9999-12-
31'}})
demo_audit_cols_ex3 = ['eff_dt', 'exp_dt']

print('df')
print(demo_df_ex3)
print()
print('audit_cols')
print(demo_audit_cols_ex3)
print()
print('existing_journal')
print(demo_existing_journal_ex3)
```

```
df
  id  paid service tier promo  price
0  1  True   audio    1  base   8.99
1  2  True   video    2  base  15.99
2  2  True   audio    2  base  12.99

audit_cols
['eff_dt', 'exp_dt']

existing_journal
     id   paid service tier  promo  price      eff_dt      exp_dt
667   0   True   video    2  intro  11.99  2018-02-01  2018-07-31
668   1   True   audio    1  intro   5.99  2018-02-01  2018-07-31
669   2   True   video    2  intro  11.99  2018-02-01  2018-07-31
670   2   True   audio    2  intro   9.99  2018-02-01  2018-07-31
671   3   True   video    1  intro   8.99  2018-02-01  2018-07-31
672   3   True   audio    1  intro   5.99  2018-02-01  2018-07-31
673   4   True   audio    1  intro   5.99  2018-02-01  2018-07-31
9     3  False   video    1   base  10.99  2018-08-01  2018-08-31
10    3  False   audio    1   base   8.99  2018-08-01  2018-08-31
17    0   True   video    2   base  15.99  2018-08-01  2019-02-28
1881  1   True   audio    1   base   8.99  2018-08-01  9999-12-31
1882  2   True   video    2   base  15.99  2018-08-01  9999-12-31
1883  2   True   audio    2   base  12.99  2018-08-01  9999-12-31
1884  4   True   audio    1   base   8.99  2018-08-01  9999-12-31
```

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

```
historical_journal WITH NO existing journal
Empty DataFrame
Columns: [id, paid, service, tier, promo, price, eff_dt, exp_dt]
Index: []

active_journal WITH NO existing journal
Empty DataFrame
Columns: [id, paid, service, tier, promo, price, eff_dt, exp_dt]
Index: []

historical_journal WITH existing journal
    id   paid service tier  promo  price      eff_dt      exp_dt
667  0   True   video    2  intro  11.99  2018-02-01  2018-07-31
668  1   True   audio    1  intro   5.99  2018-02-01  2018-07-31
669  2   True   video    2  intro  11.99  2018-02-01  2018-07-31
670  2   True   audio    2  intro   9.99  2018-02-01  2018-07-31
671  3   True   video    1  intro   8.99  2018-02-01  2018-07-31
672  3   True   audio    1  intro   5.99  2018-02-01  2018-07-31
673  4   True   audio    1  intro   5.99  2018-02-01  2018-07-31
9    3  False   video    1   base  10.99  2018-08-01  2018-08-31
10   3  False   audio    1   base   8.99  2018-08-01  2018-08-31
17   0   True   video    2   base  15.99  2018-08-01  2019-02-28

active_journal WITH existing journal
     id  paid service tier promo  price      eff_dt      exp_dt
1881  1  True   audio    1  base   8.99  2018-08-01  9999-12-31
1882  2  True   video    2  base  15.99  2018-08-01  9999-12-31
1883  2  True   audio    2  base  12.99  2018-08-01  9999-12-31
1884  4  True   audio    1  base   8.99  2018-08-01  9999-12-31
```

Note - This demo runs your solution two times. The first two DataFrames are the expected result when `exixting_journal` is `None`, and the second two DataFrames are the expected result for the `existing_journal` defined in the cell above.

```
### Exercise 3 solution
def partition_journal(df, audit_cols, existing_journal=None):
    if existing_journal is None:
        cols = list(df.columns) + audit_cols
        existing_journal = pd.DataFrame(columns=cols)

    active_mask = existing_journal['exp_dt'] == '9999-12-31'

    historical_journal = existing_journal.loc[~active_mask, :]
    active_journal = existing_journal.loc[active_mask, :]

    return historical_journal, active_journal

### demo function call
new_hist, new_active = partition_journal(demo_df_ex3,
```

```
demo_audit_cols_ex3)
hist, active = partition_journal(demo_df_ex3, demo_audit_cols_ex3,
demo_existing_journal_ex3)
print('historical_journal WITH NO existing journal')
print(new_hist)
print()
print('active_journal WITH NO existing journal')
print(new_active)
print()
print('historical_journal WITH existing journal')
print(hist)
print()
print('active_journal WITH existing journal')
print(active)
```

```
historical_journal WITH NO existing journal
Empty DataFrame
Columns: [id, paid, service, tier, promo, price, eff_dt, exp_dt]
Index: []

active_journal WITH NO existing journal
Empty DataFrame
Columns: [id, paid, service, tier, promo, price, eff_dt, exp_dt]
Index: []

historical_journal WITH existing journal
     id   paid service tier  promo  price      eff_dt      exp_dt
667  0    True   video    2  intro  11.99  2018-02-01  2018-07-31
668  1    True   audio    1  intro   5.99  2018-02-01  2018-07-31
669  2    True   video    2  intro  11.99  2018-02-01  2018-07-31
670  2    True   audio    2  intro   9.99  2018-02-01  2018-07-31
671  3    True   video    1  intro   8.99  2018-02-01  2018-07-31
672  3    True   audio    1  intro   5.99  2018-02-01  2018-07-31
673  4    True   audio    1  intro   5.99  2018-02-01  2018-07-31
9    3   False   video    1   base  10.99  2018-08-01  2018-08-31
10   3   False   audio    1   base   8.99  2018-08-01  2018-08-31
17   0    True   video    2   base  15.99  2018-08-01  2019-02-28

active_journal WITH existing journal
      id  paid service tier promo  price      eff_dt      exp_dt
1881  1   True   audio    1  base   8.99  2018-08-01  9999-12-31
1882  2   True   video    2  base  15.99  2018-08-01  9999-12-31
1883  2   True   audio    2  base  12.99  2018-08-01  9999-12-31
1884  4   True   audio    1  base   8.99  2018-08-01  9999-12-31
```

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.

```
### test_cell_ex3
exercise_start = time.time()
###
### AUTOGRADER TEST - DO NOT REMOVE
###
from tester_fw.testers import Tester

conf = {
    'case_file':'tc_3',
    'func': partition_journal, # replace this with the function
defined above
    'inputs':{ # input config dict. keys are parameter names
        'df':{
            'dtype':'pd.DataFrame', # data type of param.
            'check_modified':True,
        },
        'audit_cols':{
            'dtype':'list', # data type of param.
            'check_modified':True,
        },
        'existing_journal':{
            'dtype':'pd.DataFrame', # data type of param.
            'check_modified':True,
        }
    },
    'outputs':{
        'historical_journal':{
            'index':0,
            'dtype':'pd.DataFrame',
            'check_dtype': True,
            'check_col_dtypes': True, # Ignored if dtype is not df
            'check_col_order': False, # 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)
        },
        'active_journal':{
            'index':1,
            'dtype':'pd.DataFrame',
            'check_dtype': True,
            'check_col_dtypes': True, # Ignored if dtype is not df
            'check_col_order': False, # 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'6IRWMcPsVIAZqzDJnPgv_MfUZsxqo4Utjm2Favidv-A=', path='resource/asnlib/publicdata/')
for _ in range(70):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars, true_output_vars) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_vars) = tester.get_test_vars()
        raise

###
### AUTOGRADER TEST - DO NOT REMOVE
###
exercise_end = time.time()
print(f"This test executed in {(pd.to_datetime(exercise_end, unit='s') - pd.to_datetime(exercise_start, unit='s')).seconds} seconds")
print('Passed! Please submit.')
```

```
This test executed in 9 seconds
Passed! Please submit.
```

Helper function `drop_rename_sort`

Parameters

- `df` - any pandas DataFrame
- `drop_pattern` - regular expression pattern
- `rename_pattern` - regular expression pattern
- `key_cols` - list of strings (all of these strings must be column names of `df`)

Functionality

- drop any columns in `df` which match `drop_pattern`
- rename any of the remaining columns in `df` to names with the `rename_pattern` removed
- sort the rows in the result by the `key_cols` in descending order
- re-index the result

```
def drop_rename_sort(df, drop_pattern, rename_pattern, key_cols):
    import re
```

```
    return df\
        .drop(columns=[c for c in df.columns if
re.search(drop_pattern, c) is not None])\
        .rename(columns={c: re.sub(rename_pattern, '', c) for c in
df.columns})\
        .sort_values(key_cols)\
        .reset_index(drop=True)

df = pd.DataFrame([
    {'col_0': 'val_0_0', 'col_1_x':'val_1_0x', 'col_1_y':'val_1_0y',
'key_col_0':1, 'key_col_1':1},
    {'col_0': 'val_0_1', 'col_1_x':'val_1_1x', 'col_1_y':'val_1_1y',
'key_col_0':0, 'key_col_1':2},
    {'col_0': 'val_0_2', 'col_1_x':'val_1_2x', 'col_1_y':'val_1_2y',
'key_col_0':2, 'key_col_1':4},
    {'col_0': 'val_0_3', 'col_1_x':'val_1_3x', 'col_1_y':'val_1_3y',
'key_col_0':2, 'key_col_1':3}
])
print('df')
print(df)
drop_x = drop_rename_sort(df, '_x$', '_y$', ['key_col_0',
'key_col_1'])
print()
print("result - drop ")
print(drop_x)
```

```
df
     col_0   col_1_x   col_1_y  key_col_0  key_col_1
0  val_0_0  val_1_0x  val_1_0y          1          1
1  val_0_1  val_1_1x  val_1_1y          0          2
2  val_0_2  val_1_2x  val_1_2y          2          4
3  val_0_3  val_1_3x  val_1_3y          2          3

result - drop
     col_0     col_1  key_col_0  key_col_1
0  val_0_1  val_1_1y          0          2
1  val_0_0  val_1_0y          1          1
2  val_0_3  val_1_3y          2          3
3  val_0_2  val_1_2y          2          4
```

Exercise 4 - (**3** Points):

Motivation (don't dwell on this):
The next task is to determine which keys exist in both the active partition of the journal and the business data as well as which keys exist in only one or the other. Then we need to partition the business data into two parts (records with keys already in the journal and records without keys in the journal). We also need to partition the active journal data into two parts (records with keys existing in the business data and records without keys existing in the business data).

Requirements:
Define the function `compare_to_journal(df, key_cols, active_journal)`.

The inputs are as follows:

- `df` - a DataFrame.
- `active_journal` - another DataFrame. It will have all of the columns which are in `df`, but it may have additional columns. This input may be an empty DataFrame having 0 records.
- `key_cols` - a list of strings denoting some columns in `df` and `active_journal`. We can uniquely identify one record in either `df` or `active_journal` by a combination of these columns.

Your function should do the following:

- "Outer merge" `df` and `active_journal` on the `key_cols`. Take a look at the `indicator` and `suffixes` parameters in the docs. Let's call the result `merged`.
- Partition the rows in `merged` into these 3 partitions. The `indicator` parameter of `merge` will add an extra column to the result which is useful for this task. If you add it, it will need to be removed from the partitions.:
  - all rows in `merged` with keys existing only in `df`. Let's call this partition `df_only`.
  - all rows in `merged` with keys existing only in `active_journal`. Let's call this partition `aj_only`.
  - all rows in `merged` with keys existing in both `df` and `active_journal`. Let's call this partition `both`.
- Make **copies** of slices taken from the partitions as follows. The `suffixes` parameter of `merge` adds suffixes to duplicate column names to indicate where each came from. The provided helper function `drop_rename_sort` can be used to perform the heavy lifting here. These DataFrames are what should be returned.
  - `new_df` - all columns from `df_only` which **are not** duplicate columns originating from `active_journal`.
  - `expired_df` - all columns from `aj_only` which **are not** duplicate columns originating from `df`.
  - `compare_new_df` - all columns from `both` which **are not** duplicate columns originating from `active_journal`.
  - `compare_old_df` - all columns from `both` which **are not** duplicate columns originating from `df`.

The newly created DataFrames should be returned as a tuple, i.e. `return (new_df, expired_df, compare_new_df, compare_old_df)`.

All newly created DataFrames should be sorted lexographically based on `key_cols`.

Any suffixes or indicator columns added in the `merge` should not be included in the returned results. In other words, all 4 returned DataFrames should have the same column *names* as `active_journal`.

```
### Define demo inputs
```

```
demo_df_ex4 = pd.DataFrame([
    {'some_data_col':'some_new_value', 'some_key_col': 'new_key'},
    {'some_data_col':'some_changed_value', 'some_key_col':
'existing_key'},
    {'some_data_col':'other_changed_value', 'some_key_col':
'other_existing_key'}
])

demo_active_journal_ex4 = pd.DataFrame([
    {'some_data_col':'expiring_value', 'some_key_col': 'expiring_key',
'eff_dt': '0001-01-01', 'exp_dt':'9999-12-31'},
    {'some_data_col':'other_previous_value', 'some_key_col':
'other_existing_key', 'eff_dt': '0001-01-01', 'exp_dt':'9999-12-31'},
    {'some_data_col':'some_previous_value', 'some_key_col':
'existing_key', 'eff_dt': '6040-01-01', 'exp_dt':'9999-12-31'}
])

demo_key_cols_ex4 = ['some_key_col']

print('df')
print(demo_df_ex4)
print()
print('active_journal')
print(demo_active_journal_ex4)
print()
print('key_cols')
print(demo_key_cols_ex4)
```

```
df
         some_data_col        some_key_col
0       some_new_value             new_key
1   some_changed_value        existing_key
2  other_changed_value  other_existing_key

active_journal
          some_data_col        some_key_col      eff_dt      exp_dt
0        expiring_value        expiring_key  0001-01-01  9999-12-31
1  other_previous_value  other_existing_key  0001-01-01  9999-12-31
2   some_previous_value        existing_key  6040-01-01  9999-12-31

key_cols
['some_key_col']
```

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

```
new_df
    some_data_col some_key_col eff_dt exp_dt
0  some_new_value      new_key    NaN    NaN

expired_df
```

```
    some_data_col  some_key_col      eff_dt      exp_dt
0  expiring_value  expiring_key  0001-01-01  9999-12-31

compare_new_df
         some_data_col        some_key_col      eff_dt      exp_dt
0   some_changed_value        existing_key  6040-01-01  9999-12-31
1  other_changed_value  other_existing_key  0001-01-01  9999-12-31

compare_old_df
          some_data_col        some_key_col      eff_dt      exp_dt
0   some_previous_value        existing_key  6040-01-01  9999-12-31
1  other_previous_value  other_existing_key  0001-01-01  9999-12-31
```

Note: The `key_cols` and non-key columns may be something different than what are used in this demo. The demo names were chosen to make it clear which columns are keys and which columns are non-keys along with the effective date and expiration date.

The intermediate values from the demo should be the following if you set the `indicator` to add the extra column and `suffixes` to add `'_df'` and `'_aj'` suffixes to duplicate column names from `df` and `active_journal` respectively.

```
merged
      some_data_col_df        some_key_col      some_data_col_aj
eff_dt      exp_dt      _merge
0       some_new_value            new_key                   NaN
NaN         NaN   left_only
1   some_changed_value        existing_key   some_previous_value
6040-01-01  9999-12-31        both
2  other_changed_value  other_existing_key  other_previous_value
0001-01-01  9999-12-31        both
3                  NaN        expiring_key        expiring_value
0001-01-01  9999-12-31  right_only

df_only
  some_data_col_df some_key_col some_data_col_aj eff_dt exp_dt
0   some_new_value      new_key              NaN    NaN    NaN

aj_only
  some_data_col_df  some_key_col some_data_col_aj      eff_dt
exp_dt
3              NaN  expiring_key   expiring_value  0001-01-01  9999-
12-31

both
      some_data_col_df        some_key_col      some_data_col_aj
eff_dt      exp_dt
1   some_changed_value        existing_key   some_previous_value
6040-01-01  9999-12-31
```

```
2  other_changed_value  other_existing_key  other_previous_value
0001-01-01  9999-12-31

### Exercise 4 solution
def compare_to_journal(df, key_cols, active_journal):
    # Outer merge with suffixes and indicator set
    merged = df.merge(active_journal,
                      on=key_cols,
                      suffixes=['_df', '_aj'], # _df indicates data
comes from `df`, _aj ... from `active_journal`
                      indicator=True, # adds '_merge' column to
indicate which DataFrame the keys were found in
                      how='outer')

    # Partition `merged` based on '_merge' column, then drop it
    df_only = merged.loc[merged['_merge'] ==
'left_only'].drop('_merge', axis=1)
    aj_only = merged.loc[merged['_merge'] ==
'right_only'].drop('_merge', axis=1)
    both = merged.loc[merged['_merge'] == 'both'].drop('_merge',
axis=1)

    # Pull data originating from the appropriate source
    new_df = drop_rename_sort(df_only, '_aj$', '_df$', key_cols)
    expired_df = drop_rename_sort(aj_only, '_df$', '_aj$', key_cols)
    compare_new_df = drop_rename_sort(both, '_aj$', '_df$', key_cols)
    compare_old_df = drop_rename_sort(both, '_df$', '_aj$', key_cols)

    # Return the results
    return (new_df, expired_df, compare_new_df, compare_old_df)

### demo function call
demo_new_df, demo_expired_df, _demo_compare_new_df,
demo_compare_old_df = compare_to_journal(demo_df_ex4,
demo_key_cols_ex4, demo_active_journal_ex4)
print('new_df')
print(demo_new_df)
print()
print('expired_df')
print(demo_expired_df)
print()
print('compare_new_df')
print(_demo_compare_new_df)
print()
print('compare_old_df')
print(demo_compare_old_df)

new_df
    some_data_col some_key_col eff_dt exp_dt
0  some_new_value      new_key    NaN    NaN
```

```
expired_df
   some_key_col   some_data_col      eff_dt      exp_dt
0  expiring_key  expiring_value  0001-01-01  9999-12-31

compare_new_df
         some_data_col        some_key_col      eff_dt      exp_dt
0   some_changed_value        existing_key  6040-01-01  9999-12-31
1  other_changed_value  other_existing_key  0001-01-01  9999-12-31

compare_old_df
         some_key_col         some_data_col      eff_dt      exp_dt
0        existing_key   some_previous_value  6040-01-01  9999-12-31
1  other_existing_key  other_previous_value  0001-01-01  9999-12-31
```

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.

```
### test_cell_ex4
exercise_start = time.time()
###
### AUTOGRADER TEST - DO NOT REMOVE
###
from tester_fw.testers import Tester

conf = {
    'case_file':'tc_4',
    'func': compare_to_journal, # replace this with the function
defined above
    'inputs':{ # input config dict. keys are parameter names
        'df':{
            'dtype':'pd.DataFrame', # data type of param.
            'check_modified':True,
        },
        'key_cols':{
            'dtype':'list', # data type of param.
            'check_modified':True,
        },
        'active_journal':{
            'dtype':'pd.DataFrame', # data type of param.
```

```
            'check_modified':True,
        }
    },
    'outputs':{
        'new_df':{
            'index':0,
            'dtype':'pd.DataFrame',
            'check_dtype': True,
            'check_col_dtypes': True, # Ignored if dtype is not df
            'check_col_order': False, # 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)
        },
        'expired_df':{
            'index':1,
            'dtype':'pd.DataFrame',
            'check_dtype': True,
            'check_col_dtypes': True, # Ignored if dtype is not df
            'check_col_order': False, # 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)
        },
        'compare_new_df':{
            'index':2,
            'dtype':'pd.DataFrame',
            'check_dtype': True,
            'check_col_dtypes': True, # Ignored if dtype is not df
            'check_col_order': False, # 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)
        },
        'compare_old_df':{
            'index':3,
            'dtype':'pd.DataFrame',
            'check_dtype': True,
            'check_col_dtypes': True, # Ignored if dtype is not df
            'check_col_order': False, # 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'6IRWMcPsVIAZqzDJnPgv_MfUZsxqo4Utjm2Favidv-
A=', path='resource/asnlib/publicdata/')
for _ in range(70):
```

```
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars,
true_output_vars) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars,
true_output_vars) = tester.get_test_vars()
        raise

###
### AUTOGRADER TEST - DO NOT REMOVE
###
exercise_end = time.time()
print(f"This test executed in {(pd.to_datetime(exercise_end, unit='s')
- pd.to_datetime(exercise_start, unit='s')).seconds} seconds")
print('Passed! Please submit.')
```

```
This test executed in 10 seconds
Passed! Please submit.
```

Exercise 5 - (**3** Points):

Motivation (don't dwell on this):
Our next task is to identify records which have changed and which records are unchanged. These are a subset of records having keys in both the business data and the active journal. We need to partition both the business data and journal data into two parts based on whether the data has changed.

Requirements:
Define `compare_changes(compare_new_df, compare_old_df, audit_cols)`.

The inputs are as follows:

- `compare_new_df` - a DataFrame
- `compare_old_df` - another DataFrame with the same columns/shape/indexing as `compare_new_df`
- `audit_cols` - a list of column names which should not be used for comparison.

You can assume that the rows `compare_new_df` and `compare_old_df` are sorted and indexed such that they can be compared directly.

- Identify the columns in `compare_new_df` which are not in `audit_cols`. Let's call this `cols`.
- Compare the values in `compare_new_df[cols]` with the values in `compare_old_df[cols]`.
- Return these 3 new DataFrames:
  - `unchanged` - All of the rows in `compare_new_df` where **all** values are the same in the comparison.

- `old_changed` - All of the rows in `compare_old_df` where there are **any** differences in the comparison.
- `new_changed` - All of the rows in `compare_new_df` where there are **any** differences in the comparison.

It is possible that `compare_new_df` and `compare_old_df` are **both** empty DataFrames. If this is the case all 3 returned DataFrames would also be empty.

```
### Define demo inputs

demo_compare_new_df_ex5 = pd.DataFrame([
    {'some_column': 'new_val_0', 'key_column': 'changed_0',
'audit_column_1': None, 'audit_column_2': None},
    {'some_column': 'new_val_1', 'key_column': 'changed_1',
'audit_column_1': None, 'audit_column_2': None},
    {'some_column': 'same_val_0', 'key_column': 'unchanged_0',
'audit_column_1': None, 'audit_column_2': None},
    {'some_column': 'same_val_1', 'key_column': 'unchanged_1',
'audit_column_1': None, 'audit_column_2': None},
    {'some_column': 'new_val_2', 'key_column': 'changed_2',
'audit_column_1': None, 'audit_column_2': None},
    {'some_column': 'same_val_2', 'key_column': 'unchanged_2',
'audit_column_1': None, 'audit_column_2': None}
])

demo_compare_old_df_ex5 = pd.DataFrame([
    {'some_column': 'old_val_0', 'key_column': 'changed_0',
'audit_column_1': 'foo', 'audit_column_2': 'bar'},
    {'some_column': 'old_val_1', 'key_column': 'changed_1',
'audit_column_1': 'foo', 'audit_column_2': 'bar'},
    {'some_column': 'same_val_0', 'key_column': 'unchanged_0',
'audit_column_1': 'foo', 'audit_column_2': 'bar'},
    {'some_column': 'same_val_1', 'key_column': 'unchanged_1',
'audit_column_1': 'foo', 'audit_column_2': 'bar'},
    {'some_column': 'old_val_2', 'key_column': 'changed_2',
'audit_column_1': 'foo', 'audit_column_2': 'bar'},
    {'some_column': 'same_val_2', 'key_column': 'unchanged_2',
'audit_column_1': 'foo', 'audit_column_2': 'bar'}
])

demo_audit_cols_ex5 = ['audit_column_1', 'audit_column_2']

print('compare_new_df')
print(demo_compare_new_df_ex5)
print()
print('compare_old_df')
print(demo_compare_old_df_ex5)
print()
print('audit_cols')
```

```
print(demo_audit_cols_ex5)


compare_new_df
  some_column   key_column audit_column_1 audit_column_2
0   new_val_0    changed_0           None           None
1   new_val_1    changed_1           None           None
2  same_val_0  unchanged_0           None           None
3  same_val_1  unchanged_1           None           None
4   new_val_2    changed_2           None           None
5  same_val_2  unchanged_2           None           None

compare_old_df
  some_column   key_column audit_column_1 audit_column_2
0   old_val_0    changed_0            foo            bar
1   old_val_1    changed_1            foo            bar
2  same_val_0  unchanged_0            foo            bar
3  same_val_1  unchanged_1            foo            bar
4   old_val_2    changed_2            foo            bar
5  same_val_2  unchanged_2            foo            bar

audit_cols
['audit_column_1', 'audit_column_2']
```

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

```
unchanged
  some_column   key_column audit_column_1 audit_column_2
2  same_val_0  unchanged_0           None           None
3  same_val_1  unchanged_1           None           None
5  same_val_2  unchanged_2           None           None

old_changed
  some_column key_column audit_column_1 audit_column_2
0   old_val_0  changed_0            foo            bar
1   old_val_1  changed_1            foo            bar
4   old_val_2  changed_2            foo            bar

new_changed
  some_column key_column audit_column_1 audit_column_2
0   new_val_0  changed_0           None           None
1   new_val_1  changed_1           None           None
4   new_val_2  changed_2           None           None

### Exercise 5 solution
def compare_changes(compare_new_df, compare_old_df, audit_cols):
    # Handle the case of empty DataFrame inputs
    if compare_new_df.shape[0] == 0:
        compare_new_df.copy(), compare_new_df.copy(),
compare_new_df.copy() # 3 empty DataFrames with proper colunms
```

```
    # Identify all columns which are not `audit_cols`
    cols = [c for c in compare_new_df.columns if c not in audit_cols]

    # Create boolean mask - True when there is any difference between
the two frames, ignoring `audit_cols`
    different = (compare_new_df[cols] !=
compare_old_df[cols]).any(axis=1)

    # Use the mask to partition the DataFrames and return result
    unchanged = compare_new_df.loc[~different, :]
    old_changed = compare_old_df.loc[different, :]
    new_changed = compare_new_df.loc[different, :]
    return (unchanged,
           old_changed,
           new_changed)

# Run demo of function
(demo_unchanged_ex5,
demo_old_changed_ex5,
demo_new_changed_ex5) = compare_changes(demo_compare_new_df_ex5,
demo_compare_old_df_ex5, demo_audit_cols_ex5)
print('unchanged')
print(demo_unchanged_ex5)
print()
print('old_changed')
print(demo_old_changed_ex5)
print()
print('new_changed')
print(demo_new_changed_ex5)
```

```
unchanged
  some_column   key_column audit_column_1 audit_column_2
2  same_val_0  unchanged_0           None           None
3  same_val_1  unchanged_1           None           None
5  same_val_2  unchanged_2           None           None

old_changed
  some_column key_column audit_column_1 audit_column_2
0   old_val_0  changed_0            foo            bar
1   old_val_1  changed_1            foo            bar
4   old_val_2  changed_2            foo            bar

new_changed
  some_column key_column audit_column_1 audit_column_2
0   new_val_0  changed_0           None           None
1   new_val_1  changed_1           None           None
4   new_val_2  changed_2           None           None
```

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.

```
### test_cell_ex5
exercise_start = time.time()
###
### AUTOGRADER TEST - DO NOT REMOVE
###
from tester_fw.testers import Tester

conf = {
    'case_file':'tc_5',
    'func': compare_changes, # replace this with the function defined
above
    'inputs':{ # input config dict. keys are parameter names
        'compare_new_df':{
            'dtype':'pd.DataFrame', # data type of param.
            'check_modified':True,
        },
        'compare_old_df':{
            'dtype':'pd.DataFrame', # data type of param.
            'check_modified':True,
        },
        'audit_cols':{
            'dtype':'list', # data type of param.
            'check_modified':True,
        }
    },
    'outputs':{
        'unchanged':{
            'index':0,
            'dtype':'pd.DataFrame',
            'check_dtype': True,
            'check_col_dtypes': False, # Ignored if dtype is not df
            'check_col_order': False, # Ignored if dtype is not df
            'check_row_order': False, # Ignored if dtype is not df
            'check_column_type': True, # Ignored if dtype is not df
            'float_tolerance': 10 ** (-6)
        },
        'old_changed':{
            'index':1,
```

```
            'dtype':'pd.DataFrame',
            'check_dtype': True,
            'check_col_dtypes': False, # Ignored if dtype is not df
            'check_col_order': False, # Ignored if dtype is not df
            'check_row_order': False, # Ignored if dtype is not df
            'check_column_type': True, # Ignored if dtype is not df
            'float_tolerance': 10 ** (-6)
        },
        'new_changed':{
            'index':2,
            'dtype':'pd.DataFrame',
            'check_dtype': True,
            'check_col_dtypes': False, # Ignored if dtype is not df
            'check_col_order': False, # Ignored if dtype is not df
            'check_row_order': False, # 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'6IRWMcPsVIAZqzDJnPgv_MfUZsxqo4Utjm2Favidv-
A=', path='resource/asnlib/publicdata/')
for _ in range(70):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars,
true_output_vars) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars,
true_output_vars) = tester.get_test_vars()
        raise

###
### AUTOGRADER TEST - DO NOT REMOVE
###
exercise_end = time.time()
print(f"This test executed in {(pd.to_datetime(exercise_end, unit='s')
- pd.to_datetime(exercise_start, unit='s')).seconds} seconds")
print('Passed! Please submit.')
```

```
This test executed in 8 seconds
Passed! Please submit.
```

Exercise 6 - (**1** Points):

Motivation (Don't dwell on this): So far we have sliced and diced the business data and journal data into several partitions. Some of these partitions will need to be added to the journal by setting the effective and expiration dates. Here we will write a generic function to set the

effective date to the "data date" (date when the snapshot was taken) and set the expiration date to '9999-12-31' for an arbitrary partition.

Requirements: Define `add_records(df, data_date)`

The input `df` is a DataFrame which can be assumed to have columns `'eff_dt'` and `'exp_dt'`. The input `data_date` is a Pandas Timestamp object. The function should return a *new* DataFrame having the same data as `df` with the following exceptions:

- The `'eff_dt'` field should be set to the `data_date` as a string in 'YYYY-MM-DD' format for all records. See the docs for more information on performing this transformation.
- The `'exp_dt'` field should be set to `'9999-12-31'` for all records.

Note `df` may be empty!

```
### Define demo inputs

demo_df_ex6 = pd.DataFrame([
    {'eff_dt':None, 'exp_dt':None, 'col0':'val_00', 'col1':'val01'},
    {'eff_dt':None, 'exp_dt':None, 'col0':'val_10', 'col1':'val11'},

    {'eff_dt':None, 'exp_dt':None, 'col0':'val_20', 'col1':'val21'},

])
print('df')
print(demo_df_ex6)
print()
demo_data_date_ex6 = pd.to_datetime('2020-10-15')
print('data_date')
print(type(demo_data_date_ex6))
print(demo_data_date_ex6)
```

```
df
  eff_dt exp_dt    col0   col1
0   None   None  val_00  val01
1   None   None  val_10  val11
2   None   None  val_20  val21

data_date
<class 'pandas._libs.tslibs.timestamps.Timestamp'>
2020-10-15 00:00:00
```

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

```
       eff_dt      exp_dt    col0   col1
0  2020-10-15  9999-12-31  val_00  val01
1  2020-10-15  9999-12-31  val_10  val11
2  2020-10-15  9999-12-31  val_20  val21
```

```
### Exercise 6 solution
def add_records(df, data_date):
    ###
    df = df.copy()
    df['exp_dt'] = '9999-12-31'
    df['eff_dt'] = data_date.strftime('%Y-%m-%d')
    return df
    ###

### demo function call
print(add_records(demo_df_ex6, demo_data_date_ex6))
```

```
       eff_dt      exp_dt    col0   col1
0  2020-10-15  9999-12-31  val_00  val01
1  2020-10-15  9999-12-31  val_10  val11
2  2020-10-15  9999-12-31  val_20  val21
```

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.

```
### test_cell_ex6
exercise_start = time.time()
###
### AUTOGRADER TEST - DO NOT REMOVE
###
from tester_fw.testers import Tester

conf = {
    'case_file':'tc_6',
    'func': add_records, # replace this with the function defined
above
    'inputs':{ # input config dict. keys are parameter names
        'df':{
            'dtype':'pd.DataFrame', # data type of param.
            'check_modified':True,
        },
        'data_date':{
            'dtype':'Timestamp',
            'check_modified':False
        }
```

```
    },
    'outputs':{
        'output_0':{
            'index':0,
            'dtype':'pd.DataFrame',
            'check_dtype': True,
            'check_col_dtypes': True, # Ignored if dtype is not df
            'check_col_order': False, # Ignored if dtype is not df
            'check_row_order': False, # 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'6IRWMcPsVIAZqzDJnPgv_MfUZsxqo4Utjm2Favidv-
A=', path='resource/asnlib/publicdata/')
for _ in range(70):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars,
true_output_vars) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars,
true_output_vars) = tester.get_test_vars()
        raise

###
### AUTOGRADER TEST - DO NOT REMOVE
###
exercise_end = time.time()
print(f"This test executed in {(pd.to_datetime(exercise_end, unit='s')
- pd.to_datetime(exercise_start, unit='s')).seconds} seconds")
print('Passed! Please submit.')
```

```
This test executed in 1 seconds
Passed! Please submit.
```

Exercise 7 - (**2** Points):

Motivation (Don't dwell on this): So far we have sliced and diced the business data and journal data into several partitions. Some of these partitions will need to be expired in the journal by updating the expiration dates. Here we will write a generic function to set the expiration date to one day prior to the "data date" (date when the snapshot was taken).

Requirements: Define `expire_records(df, data_date)`

The input `df` is a DataFrame which can be assumed to have columns `'eff_dt'` and `'exp_dt'`. The input `data_date` is a Pandas Timestamp object. The function should return a *new* DataFrame having the same data as `df` with the following exceptions:

- The `'exp_dt'` field should be set to **one day prior to the** `data_date` as a string in 'YYYY-MM-DD' format for all records. See the stackoverflow or pandas docs for more information on math with Timestamps and strftime docs for more information on extracting the string.

You will want to use a module that accounts for changes in months, years, and leap-days for calculating the `exp_dt`.

Note - `df` may be empty!

```
### Define demo inputs

demo_df_ex7 = pd.DataFrame([
    {'eff_dt':'0001-01-01', 'exp_dt':None, 'col0':'val_00',
'col1':'val01'},
    {'eff_dt':'0001-01-01', 'exp_dt':None, 'col0':'val_10',
'col1':'val11'},
    {'eff_dt':'0001-01-01', 'exp_dt':None, 'col0':'val_20',
'col1':'val21'},
])
print('df')
print(demo_df_ex7)
print()
demo_data_date_ex7 = pd.to_datetime('2020-03-01')
print('data_date')
print(type(demo_data_date_ex7))
print(demo_data_date_ex7)
```

```
df
       eff_dt exp_dt    col0   col1
0  0001-01-01   None  val_00  val01
1  0001-01-01   None  val_10  val11
2  0001-01-01   None  val_20  val21

data_date
<class 'pandas._libs.tslibs.timestamps.Timestamp'>
2020-03-01 00:00:00
```

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

```
       eff_dt      exp_dt    col0   col1
0  0001-01-01  2020-02-29  val_00  val01
1  0001-01-01  2020-02-29  val_10  val11
2  0001-01-01  2020-02-29  val_20  val21
```

```
### Exercise 7 solution
def expire_records(df, data_date):
    df = df.copy()
    df['exp_dt'] = (data_date - pd.Timedelta('1 day')).strftime('%Y-
%m-%d')
```

```
    return df

### demo function call
print(expire_records(demo_df_ex7, demo_data_date_ex7))
```

```
       eff_dt      exp_dt    col0   col1
0  0001-01-01  2020-02-29  val_00  val01
1  0001-01-01  2020-02-29  val_10  val11
2  0001-01-01  2020-02-29  val_20  val21
```

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.

```
### test_cell_ex7
exercise_start = time.time()
###
### AUTOGRADER TEST - DO NOT REMOVE
###
from tester_fw.testers import Tester

conf = {
    'case_file':'tc_7',
    'func': expire_records, # replace this with the function defined
above
    'inputs':{ # input config dict. keys are parameter names
        'df':{
            'dtype':'pd.DataFrame', # data type of param.
            'check_modified':True,
        },
        'data_date':{
            'dtype':'Timestamp',
            'check_modified':False
        }
    },
    'outputs':{
        'output_0':{
            'index':0,
            'dtype':'pd.DataFrame',
            'check_dtype': True,
            'check_col_dtypes': True, # Ignored if dtype is not df
```

```
            'check_col_order': False, # Ignored if dtype is not df
            'check_row_order': False, # 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'6IRWMcPsVIAZqzDJnPgv_MfUZsxqo4Utjm2Favidv-
A=', path='resource/asnlib/publicdata/')
for _ in range(70):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars,
true_output_vars) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars,
true_output_vars) = tester.get_test_vars()
        raise

###
### AUTOGRADER TEST - DO NOT REMOVE
###
exercise_end = time.time()
print(f"This test executed in {(pd.to_datetime(exercise_end, unit='s')
- pd.to_datetime(exercise_start, unit='s')).seconds} seconds")
print('Passed! Please submit.')
```

```
This test executed in 1 seconds
Passed! Please submit.
```

Putting it all together

(Don't dwell on this)
With the functions we wrote in the past few exercises, we have everything we need to perform a type 2 journaling operation. Here's how it looks all put together:

```
def journal(df, key_cols, data_date, existing_journal=None):
    audit_cols = ['eff_dt', 'exp_dt']
    historical_journal, active_journal = partition_jrnl(df,
audit_cols, existing_journal)
    new_df, expired_df, compare_new_df, compare_old_df =
compare_to_journal(df, key_cols, active_journal)
    unchanged, old_changed, new_changed =
compare_changes(compare_new_df, compare_old_df, audit_cols)

    new_records = add_records(new_df, data_date)
    new_changed_records = add_records(new_changed, data_date)
    expired_records = expire_records(expired_df, data_date)
    old_changed_records = expire_records(old_changed, data_date)
```

```
    return pd.concat([
        historical_journal,
        new_records,
        new_changed_records,
        expired_records,
        old_changed_records,
        unchanged
    ])
```

Exercise 8 - (**2** Points):

Motivation (don't dwell on this):
Here's where all of the tedious work we did partitioning these DataFrames pays off. We can reconstruct a snapshot of any particular date from the journal! To do so we just filter the journal to keep only records whose effective date is on or before that date and whose expiration date is on or after that date.

Requirements: Define the function `time_travel(journal, data_date)` as follows:

The input `journal` is a DataFrame with columns `'eff_dt'` and `'exp_dt'` in addition other arbitrary columns. The input `data_date` is a string representing a date in 'YYYY-MM-DD' format. The function should return a *new* DataFrame containing all records where `'eff_dt'` ≤ `'data_date'` ≤ `'exp_dt'`.

The `'eff_dt'` and `'exp_dt'` fields should not be included in the result.

Note: One convenient fact about storing dates as strings in 'YYYY-MM-DD' format is that you can compare the strings directly with `<`, `<=`, `!=`, `==`, `>=`, `>` without converting to a more complicated data type!

```
### Define demo inputs

demo_journal_ex8 = pd.DataFrame(
    {'id': {674: '10', 675: '10', 2057: '10', 2058: '10', 1307: '998',
1308: '998', 1003: '998', 1004: '998', 1163: '10', 1164: '10', 10:
'998', 11: '998'},
    'paid': {674: 'True', 675: 'True', 2057: 'True', 2058: 'True',
1307: 'True', 1308: 'True', 1003: 'True', 1004: 'True', 1163: 'True',
1164: 'True',
            10: 'False', 11: 'False'},
    'service': {674: 'video', 675: 'audio', 2057: 'video', 2058:
'audio', 1307: 'video', 1308: 'audio', 1003: 'video', 1004: 'audio',
1163: 'video',
            1164: 'audio', 10: 'video', 11: 'audio'},
    'tier': {674: '1', 675: '1', 2057: '1', 2058: '1', 1307: '1',
1308: '1', 1003: '1', 1004: '1', 1163: '2', 1164: '2', 10: '1', 11:
'1'},
    'promo': {674: 'intro', 675: 'intro', 2057: 'base', 2058: 'base',
```

```
1307: 'intro', 1308: 'intro', 1003: 'base', 1004: 'base', 1163:
'base',
            1164: 'base', 10: 'base', 11: 'base'},
    'price': {674: '8.99', 675: '5.99', 2057: '10.99', 2058: '8.99',
1307: '8.99', 1308: '5.99', 1003: '10.99', 1004: '8.99', 1163:
'15.99',
        1164: '12.99', 10: '10.99', 11: '8.99'},
    'eff_dt': {674: '2018-02-01', 675: '2018-02-01', 2057: '2018-08-
01', 2058: '2018-08-01', 1307: '2018-12-01', 1308: '2018-12-01',
            1003: '2019-06-01', 1004: '2019-06-01', 1163: '2019-08-
01', 1164: '2019-08-01', 10: '2019-10-01', 11: '2019-10-01'},
    'exp_dt': {674: '2018-07-31', 675: '2018-07-31', 2057: '2019-07-
31', 2058: '2019-07-31', 1307: '2019-05-31', 1308: '2019-05-31',
            1003: '2019-09-30', 1004: '2019-09-30', 1163: '9999-12-
31', 1164: '9999-12-31', 10: '2019-10-31', 11: '2019-10-31'}})
print('journal')
print(demo_journal_ex8)
```

```
journal
       id   paid service tier  promo  price      eff_dt      exp_dt
674    10   True   video    1  intro   8.99  2018-02-01  2018-07-31
675    10   True   audio    1  intro   5.99  2018-02-01  2018-07-31
2057   10   True   video    1   base  10.99  2018-08-01  2019-07-31
2058   10   True   audio    1   base   8.99  2018-08-01  2019-07-31
1307  998   True   video    1  intro   8.99  2018-12-01  2019-05-31
1308  998   True   audio    1  intro   5.99  2018-12-01  2019-05-31
1003  998   True   video    1   base  10.99  2019-06-01  2019-09-30
1004  998   True   audio    1   base   8.99  2019-06-01  2019-09-30
1163   10   True   video    2   base  15.99  2019-08-01  9999-12-31
1164   10   True   audio    2   base  12.99  2019-08-01  9999-12-31
10    998  False   video    1   base  10.99  2019-10-01  2019-10-31
11    998  False   audio    1   base   8.99  2019-10-01  2019-10-31
```

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

```
data_date: 2018-02-02
     id  paid service tier  promo price
674  10  True   video    1  intro  8.99
675  10  True   audio    1  intro  5.99

data_date: 2018-08-01
      id  paid service tier promo  price
2057  10  True   video    1  base  10.99
2058  10  True   audio    1  base   8.99

data_date: 2019-01-01
       id  paid service tier  promo  price
2057   10  True   video    1   base  10.99
2058   10  True   audio    1   base   8.99
```

```
1307  998  True   video    1  intro   8.99
1308  998  True   audio    1  intro   5.99

data_date: 2019-06-15
       id  paid service tier promo  price
2057   10  True   video    1  base  10.99
2058   10  True   audio    1  base   8.99
1003  998  True   video    1  base  10.99
1004  998  True   audio    1  base   8.99
```

Note - this demo runs your solution for several different `data_date` values. Each of the DataFrames displayed is from a single run.

```
### Exercise 8 solution
def time_travel(journal, data_date):
    eff_mask = journal['eff_dt'] <= data_date
    exp_mask = journal['exp_dt'] >= data_date
    mask = eff_mask & exp_mask
    return journal.loc[mask, :].drop(columns=['eff_dt', 'exp_dt'])

### demo function call
for demo_data_date_ex8 in ['2018-02-02', '2018-08-01', '2019-01-01',
'2019-06-15']:
    print(f'data_date: {demo_data_date_ex8}')
    print(time_travel(demo_journal_ex8, demo_data_date_ex8))
    print()
```

```
data_date: 2018-02-02
     id  paid service tier  promo price
674  10  True   video    1  intro  8.99
675  10  True   audio    1  intro  5.99

data_date: 2018-08-01
      id  paid service tier promo  price
2057  10  True   video    1  base  10.99
2058  10  True   audio    1  base   8.99

data_date: 2019-01-01
       id  paid service tier  promo  price
2057   10  True   video    1   base  10.99
2058   10  True   audio    1   base   8.99
1307  998  True   video    1  intro   8.99
1308  998  True   audio    1  intro   5.99

data_date: 2019-06-15
       id  paid service tier promo  price
2057   10  True   video    1  base  10.99
2058   10  True   audio    1  base   8.99
1003  998  True   video    1  base  10.99
```

```
1004  998  True   audio    1  base   8.99
```

The cell below will test your solution for Exercise 8. 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.

```
### test_cell_ex8
exercise_start = time.time()
###
### AUTOGRADER TEST - DO NOT REMOVE
###
from tester_fw.testers import Tester

conf = {
    'case_file':'tc_8',
    'func': time_travel, # replace this with the function defined
above
    'inputs':{ # input config dict. keys are parameter names
        'journal':{
            'dtype':'pd.DataFrame', # data type of param.
            'check_modified':True,
        },
        'data_date':{
            'dtype':'str',
            'check_modified':False
        }
    },
    'outputs':{
        'output_0':{
            'index':0,
            'dtype':'pd.DataFrame',
            'check_dtype': True,
            'check_col_dtypes': True, # Ignored if dtype is not df
            'check_col_order': False, # Ignored if dtype is not df
            'check_row_order': False, # 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'6IRWMcPsVIAZqzDJnPgv_MfUZsxqo4Utjm2Favidv-
A=', path='resource/asnlib/publicdata/')
for _ in range(70):
    try:
        tester.run_test()
        (input_vars, original_input_vars, returned_output_vars,
true_output_vars) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars,
true_output_vars) = tester.get_test_vars()
        raise

###
### AUTOGRADER TEST - DO NOT REMOVE
###
print('Passed! Please submit.')
exercise_end = time.time()
print(f"This test executed in {(pd.to_datetime(exercise_end, unit='s')
- pd.to_datetime(exercise_start, unit='s')).seconds} seconds")
overall_end = exercise_end
print(f"The exam executed in {(pd.to_datetime(overall_end, unit='s') -
pd.to_datetime(overall_start, unit='s')).seconds} seconds")
```

```
Passed! Please submit.
This test executed in 6 seconds
The exam executed in 1440 seconds
```

Fin. If you have made it this far, congratulations on completing the exam. **Don't forget to submit!**