Exercise 0 (1 Points):

After downloading and unzipping the file, you will find the data stored in a **JSON** format. You may or may not have seen this type of data format before, so we will take care of reading the data into our Python environment. The result of the code below is that `food_lod` will load the serialized contents of the json file into Python objects - in this case, a `list` of `dicts`.

Run the test cell below to load the data. We are treating this as a "test" cell, so you will get one point for just submitting. How generous!

In [ ]:
```
### BEGIN HIDDEN TESTS
if False:
    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', 'tester_6040.py', 'test_utils.py']:
        hash_check(f'tester_fw/{fname}', f'resource/asnlib/public/{fname}')
    del hash_check
### END HIDDEN TESTS

import json                                          # import the json module
path = './resource/asnlib/publicdata/food_data.json' # path to the data file
with open(path, 'r') as file:                        # open the path and keep
the file object as a context
    food_lod = json.load(file)                       # load the file into a va
riable in our Python environment
```

Exploring the data

Let's start by taking a look at some of the basic attributes about this data.

In [ ]:
```
{
    'type': type(food_lod),
    'length': len(food_lod),
    'value_types': {type(v) for v in food_lod}
}
```

Well... `food_lod` is a `list`, with 30,000 entries, and each of those are of type `dict`. Let's take a look at some of the keys in one of the `dicts`.

In [ ]:
```
food_lod[0].keys()
```

Exercise 1 (1 Points):

These look like some promising candidates for extracting information about individual foods. There appear to be some "category" related keys, which may be useful for grouping foods and comparing between groups as well. For further analysis, we want to know if the `dicts` are all of similar structure to the first one. A good start to analyzing this is determining which keys are common to all of them...

Given an input `lod`, which is a `list` of `dicts` complete the function `common_keys(lod)` to return a Python `set` of the keys which are common to all of the `dicts` in `lod`.

In [ ]:
```
### Define common_keys
def common_keys(lod):
    ### BEGIN SOLUTION
    my_keys = set(lod[0].keys())
    for d in lod:
        my_keys &= set(d.keys())
    return my_keys
    ### END SOLUTION
```

The demo cell below should display the following output:

```
{'bar', 'qux'}
```

In [ ]:
```
### define demo inputs
demo_key_list_ex0 = ['foo', 'bar', 'baz', 'qux', 'tav', 'wot']
demo_lod_ex0 = [
    {k: 1 for k in demo_key_list_ex0[1:]},
    {k: 1 for k in demo_key_list_ex0[:-1]},
    {k: 1 for k in demo_key_list_ex0[1::2]}
]
```

In [ ]:
```
### call demo funtion
common_keys(demo_lod_ex0)
```

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 [ ]:
```
### 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', 'tester_6040.py', 'test_utils.py']:
    hash_check(f'tester_fw/{fname}', f'resource/asnlib/public/{fname}')
del hash_check
del dill
del hashlib
### END HIDDEN TESTS

from tester_fw.testers import Tester_ex1
tester = Tester_ex1()
for _ in range(20):
    try:
        tester.run_test(common_keys)
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise

### BEGIN HIDDEN TESTS
tester = Tester_ex1(key=b'4gbCuwhFj77ZLr-nS9-6_7OKthpg6HpqGWs1vnuKkgk=', path
='resource/asnlib/publicdata/encrypted/')
for _ in range(20):
    try:
        tester.run_test(common_keys)
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise
### END HIDDEN TESTS
print('Passed! Please submit.')
```

Even if your solution was incorrect or you skipped this exercise, run this cell to see the expected output of a call to `common_keys(food_lod)`.

In [ ]:
```
### Loading results
import pickle
import os
path = './resource/asnlib/publicdata/ex1.pkl'
if not os.path.exists(path):
    with open(path, 'wb') as file:
        pickle.dump(common_keys(food_lod), file)
with open(path, 'rb') as file:
    food_keys = pickle.load(file)
```

Exercise 2 (4 Points):

For our analysis, we are interested in the nutritional content and ingredients contained in each food. Additionally we would like to group foods by the categories given. The keys of interest are 'description', 'ingredients', 'labelNutrients', and 'brandedFoodCategory'.

In [ ]:
```
keys_of_interest = {'description', 'ingredients', 'labelNutrients', 'brandedFo
odCategory'}
{k:v for k, v in food_lod[0].items() if k in keys_of_interest}
```

Define `extract_basic_data` to meet the following requirements. Given a `list` of `dicts`, `lod`, create a **new** `list` of `dicts` called `basic_data`. For each `dict` in `lod`, there should be a corresponding `dict` in `basic_data` with the following key/value pairs:

- `'description'` - `str` associated with `'description'` in the `lod` `dict`.
- `'list_of_ingredients'` - `list` of all the ingredients associated with `'ingredients'` in the `lod` `dict`. *See "Notes on ingredients" below*.
- `'raw_nutrients'` - `dict` mapping the nutrient name (`str`) to it's amount (`float`). *See "Notes on nutrients" below*.
- `'category'` - `str` associated with `'brandedFoodCategory'` in the `lod` `dict`.

Notes on ingredients

- For each `dict`, `d` in `lod` the **ingredients** are stored as a `str` associated with the `'ingredients'` key.
- Sometimes there is extra information wrapped in parentheses. We do not want to include this information in our analysis, so any text wrapped in `()` (and the parentheses themselves) should be left out of further processing. There may be **multiple** sets of parentheses in an `ingredients` string.
  - You can assume that there are not **nested** parentheses. For example strings of this form **will not** occur - `'item, item1 (level 1 (another, level)), item2.'`
  - The `re` module may be helpful here.
  - Note that there can be *anything* in between the parentheses and all of that text should be discarded. For example `'ingredient 1, ingredient 2 (ingredient 2.1, ingredient 2.2, [ingredient 2.2.1, ingredient 2.2.2]), ingredient 3.'` should result in just `['ingredient 1', 'ingredient 2', 'ingredient 3']` as it's associated `'list_of_ingredients'`.
- The ingredient string ends in `'.'`, which should also be left out of further processing.
- The individual ingredients are separated by `', '`.
- The ingredients in `basic_data[i]['list_of_ingredients']` should not have any leading or trailing whitespace.

Notes on nutrients

- For each `dict`, `d`, in `lod`, the nutrients are associated with the `'labelNutrients'` key.
- `d['labelNutrients']` is a dictionary of the form `{'protein': {'value': 10}, 'riboflavin': {'value': 2}}`, i.e. mapping the nutrient name to a dictionary with one key (`'value'`) which is mapped to the amount of that nutrient present in a particular food.

In [ ]:
```
### Define extract_basic_data
def extract_basic_data(lod):
    ### BEGIN SOLUTION
    return [extract_record_data(record) for record in lod]
def extract_record_data(record):
    ### Convert ingredients from `str` to `list`
    import re
    pattern = re.compile(r'\(.*?\)|\.') # matches "anything" wrapped in parent
hesis or the '.' character
    clean_str = re.sub(pattern, '', record['ingredients'])
    loi = [s.strip() for s in clean_str.split(', ')]
    ### Convert nutrients into proper form
    nut_dict = {k: float(v['value']) for k, v in record['labelNutrients'].item
s()}
    ### Build and return result
    return {
        'description': record['description']
        ,'list_of_ingredients': loi
        ,'raw_nutrients': nut_dict
        ,'category': record['brandedFoodCategory']
    }
    ### END SOLUTION
```

The demo cell below should display the following output:

```
[{'description': 'KETTLE COOKED POTATO CHIPS, PINK HIMALAYAN SALT & RED WINE VINEGA
R',
  'list_of_ingredients': ['POTATOES',
   'VEGETABLE OIL',
   'MALTODEXTRIN',
   'HIMALAYAN SALT',
   'RED WINE VINEGAR',
   'CITRIC ACID',
   'SUGAR',
   'WHITE DISTILLED VINEGAR',
   'NATURAL FLAVOR'],
  'raw_nutrients': {'fat': 7.0,
   'saturatedFat': 0.501,
   'transFat': 0.0,
   'cholesterol': 0.0,
   'sodium': 140.0,
   'carbohydrates': 17.0,
   'fiber': 1.01,
   'sugars': 0.0,
   'protein': 2.0,
   'calcium': 0.0,
   'iron': 0.4,
   'potassium': 319.0,
   'addedSugar': 0.0,
   'calories': 140.0},
  'category': 'Chips, Pretzels & Snacks'},
 {'description': 'TOMATO BASIL PASTA SAUCE',
  'list_of_ingredients': ['TOMATO PUREE',
   'TOMATOES',
   'SUGAR',
   'SOYBEAN OIL',
   'SALT',
   'DRIED ONIONS',
   'DRIED GARLIC',
   'SPICES',
   'LEMON JUICE CONCENTRATE',
   'ROMANO CHEESE'],
  'raw_nutrients': {'fat': 2.0,
   'sodium': 580.0,
   'carbohydrates': 17.0,
   'fiber': 2.94,
   'sugars': 10.0,
   'protein': 3.0,
   'calcium': 29.4,
   'iron': 0.998,
   'potassium': 750.0,
   'addedSugar': 2.05,
```

```
 'calories': 89.6},
'category': 'Prepared Pasta & Pizza Sauces'}]
```

In [ ]:
```
### define demo inputs
keys_of_interest = {'description', 'ingredients', 'labelNutrients', 'brandedFo
odCategory'}
demo_lod_ex1 = [{k:v for k, v in d.items() if k in keys_of_interest} for d in
food_lod[:2]]
```

In [ ]:
```
### call demo funtion
extract_basic_data(demo_lod_ex1)
```

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.

In [ ]:
```
### test_cell_ex2

### 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', 'tester_6040.py', 'test_utils.py']:
    hash_check(f'tester_fw/{fname}', f'resource/asnlib/public/{fname}')
del hash_check
del dill
del hashlib
### END HIDDEN TESTS

from tester_fw.testers import Tester_ex2
tester = Tester_ex2()
for _ in range(20):
    try:
        tester.run_test(extract_basic_data)
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise

### BEGIN HIDDEN TESTS
tester = Tester_ex2(key=b'4gbCuwhFj77ZLr-nS9-6_7OKthpg6HpqGWs1vnuKkgk=', path
='resource/asnlib/publicdata/encrypted/')
for _ in range(20):
    try:
        tester.run_test(extract_basic_data)
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise
### END HIDDEN TESTS
print('Passed! Please submit.')
```

Even if your solution was incorrect or you skipped this exercise, run this cell to see the expected output of a call to `extract_basic_data(food_lod)`.

In [ ]:
```
### Loading results
import pickle
import os
path = './resource/asnlib/publicdata/ex2.pkl'
if not os.path.exists(path):
    with open(path, 'wb') as file:
        pickle.dump(extract_basic_data(food_lod), file)
with open(path, 'rb') as file:
    basic_data = pickle.load(file)
```

Exercise 3 (1 Points):

Our analysis requires that the foods have at least 3 listed ingredients and have amounts for nutrients: 'fat', 'protein', 'sodium', and 'carbohydrates'.

Given `data`, a `list` of `dicts`, define `filter_basic_data(data)` to filter out unwanted records.

- This function should return a **new** list containing the same `dicts` as `data` with the following exceptions.
- You can assume that each `dict` in `data` will have `'list_of_ingredients` and `'raw_nutrients'` as keys and that the respective values for those keys are of type `list` and `dict`.
- Any `dict` with fewer than 3 items in it's `'list_of_ingredients'` should not be included.
- Any `dict`, `d`, where `d['raw_nutrients']` does not have **all** of `{'fat', 'protein', 'sodium', 'carbohydrates'}` as keys should not be included.

In [ ]:
```
### Define filter_basic_data
def filter_basic_data(data):
    ### BEGIN SOLUTION
    def keep_record(record):
        return len(record['list_of_ingredients']) >= 3 \
            and len({'fat', 'protein', 'carbohydrates', 'sodium'} & set(record
['raw_nutrients'].keys())) == 4

    return [record for record in data if keep_record(record)]
    ### END SOLUTION
```

The demo cell below should display the following output:

```
[{'list_of_ingredients': [1, 2, 'this ok', 'milk'],
  'raw_nutrients': {'fat': 22,
   'protein': 'caterpillar',
   'carbohydrates': 5,
   'sodium': 100,
   'awesome sauce': 'this one should be kept'}}]
```

In [ ]:
```
### define demo inputs
demo_data_ex3 = [
    {
        'list_of_ingredients': [1, 2, 'this ok', 'milk'],
        'raw_nutrients': {
            'fat':22,
            'protein': 'caterpillar',
            'carbohydrates': 5,
            'sodium': 100,
            'awesome sauce': 'this one should be kept'}
    },
    {
        'list_of_ingredients': ['catfish', 2, 'cse6040', 'bicycle'],
        'raw_nutrients': {
            'fat':12,
            'carbohydrates': 35,
            'sodium': 70,
            'awesome sauce': 'this one should be rejected - no protein'
        }
    },
    {
        'list_of_ingredients': ['marble', 2.5],
        'raw_nutrients': {
            'fat':12,
            'carbohydrates': 35,
            'protein': 7,
            'sodium': 70,
            'awesome sauce': 'this one should be rejected too - not enough ing
redients'
        }
    }
]
```

In [ ]:
```
### call demo funtion
filter_basic_data(demo_data_ex3)
```

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', 'tester_6040.py', 'test_utils.py']:
    hash_check(f'tester_fw/{fname}', f'resource/asnlib/public/{fname}')
del hash_check
del dill
del hashlib
### END HIDDEN TESTS

from tester_fw.testers import Tester_ex3
tester = Tester_ex3()
for _ in range(20):
    try:
        tester.run_test(filter_basic_data)
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise

### BEGIN HIDDEN TESTS
tester = Tester_ex3(key=b'4gbCuwhFj77ZLr-nS9-6_7OKthpg6HpqGWs1vnuKkgk=', path
='resource/asnlib/publicdata/encrypted/')
for _ in range(20):
    try:
        tester.run_test(filter_basic_data)
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise
### END HIDDEN TESTS
print('Passed! Please submit.')
```

Even if your solution was incorrect or you skipped this exercise, run this cell to see the expected output of a call to `filter_basic_data(basic_data)`.

In [ ]:
```
### Loading results
import pickle
import os
path = './resource/asnlib/publicdata/ex3.pkl'
if not os.path.exists(path):
    with open(path, 'wb') as file:
        pickle.dump(filter_basic_data(basic_data), file)
with open(path, 'rb') as file:
    filtered_data = pickle.load(file)
```

Exercise 4 (2 Points):

We want to compute summary statistics on the nutrients present in each food. While you might be able to find formulas for these statistics and implement them yourselves - there is no need to reinvent the wheel. Feel free to use the `statistics` module, but you will have to `import` it yourself.

Given a `list` of `dicts`, `data` structured as `basic_data`, define `make_summary(data, key)` to generate a dictionary of summary statistics.

- We will assume that each `dict` in `data` has a `'raw_nutrients'` key mapped to a dictionary which maps nutrients to amounts.
- Extract the amount of the nutrient given by `key` for each `dict`, `d` in `data` - we will call these the observations. I.e. `data[0]['raw_nutrients'][key]` is one observation.
- Note: each entry in `data` counts as an observation and for all `dicts` `d` in `data`, `d['raw_nutrients'][key]` is **not** guaranteed to exist. In such cases, we will interpret the observation as a 0.
- Compute statistics on the observations. Store the results in a dictionary with the following mapping:
  - `'mean'` - (`float`) mean of all observations
  - `'median'` - (`float`) median of all observations
  - `'stdev'` - (`float`) **population** standard deviation of all observations - check your stats notes and documentation to make sure you're computing this correctly
  - `'min'` - (`float`) minimum
  - `'max'` - (`float`) maximum

In [ ]:
```
### Define make_summary
def make_summary(data, key):
    ### BEGIN SOLUTION
    def stat_summary(nums):
        from statistics import mean, median, pstdev
        return {
            'mean': float(mean(nums))
            , 'median': float(median(nums))
            , 'stdev': float(pstdev(nums))
            , 'min': float(min(nums))
            , 'max': float(max(nums))
        }
    nums = [record['raw_nutrients'].get(key, 0) for record in data]
    return stat_summary(nums)
    ### END SOLUTION
```

The demo cell below should display the following output:

```
key: foo
{'mean': 18.714285714285715, 'median': 17.0, 'stdev': 12.75835060672349, 'min': 5.
0, 'max': 48.0}

key: bar
{'mean': 23.571428571428573, 'median': 33.0, 'stdev': 14.907880397936646, 'min': 0.
0, 'max': 33.0}

key: baz
{'mean': 100.0, 'median': 100.0, 'stdev': 0.0, 'min': 100.0, 'max': 100.0}
```

In [ ]:
```
### define demo inputs
demo_data_ex4 = [
    {'raw_nutrients': {'foo': 12,   'bar': 33,  'baz': 100}},
    {'raw_nutrients': {'foo': 48,   'bar': 33,  'baz': 100}},
    {'raw_nutrients': {'foo': 17,   'bar': 33,  'baz': 100}},
    {'raw_nutrients': {'foo': 5,                'baz': 100}},
    {'raw_nutrients': {'foo': 18,   'bar': 33,  'baz': 100}},
    {'raw_nutrients': {'foo': 12,   'bar': 33,  'baz': 100}},
    {'raw_nutrients': {'foo': 19,               'baz': 100}},
]
demo_keys_ex4 = ['foo', 'bar', 'baz']
```

In [ ]:
```
### call demo funtion
for k in demo_keys_ex4:
    print(f'key: {k}')
    print(make_summary(demo_data_ex4, k))
    print()
```

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', 'tester_6040.py', 'test_utils.py']:
    hash_check(f'tester_fw/{fname}', f'resource/asnlib/public/{fname}')
del hash_check
del dill
del hashlib
### END HIDDEN TESTS

from tester_fw.testers import Tester_ex4
tester = Tester_ex4()
for _ in range(20):
    try:
        tester.run_test(make_summary)
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise

### BEGIN HIDDEN TESTS
tester = Tester_ex4(key=b'4gbCuwhFj77ZLr-nS9-6_7OKthpg6HpqGWs1vnuKkgk=', path
='resource/asnlib/publicdata/encrypted/')
for _ in range(20):
    try:
        tester.run_test(make_summary)
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise
### END HIDDEN TESTS
print('Passed! Please submit.')
```

Even if your solution was incorrect or you skipped this exercise, run this cell. You would get the same result as `summary_dict` if you were to run the code below with a correct implementation of `make_summary`.

```
keys = ('fat', 'protein', 'carbohydrates', 'sodium')
{key:make_summary(filtered_data, key) for key in keys}
```

In [ ]:
```
### Loading results
import pickle
import os
path = './resource/asnlib/publicdata/ex4.pkl'
if not os.path.exists(path):
    with open(path, 'wb') as file:
        pickle.dump({key:make_summary(filtered_data, key) for key in ('fat',
'protein', 'carbohydrates', 'sodium')}, file)
with open(path, 'rb') as file:
    summary_dict = pickle.load(file)
```

Exercise 5 (3 Points):

We are interested in whether the amount of one particular nutrient in a food has any relationship with the amounts of other nutrients in the food. For this, we will compare the observations of multiple nutrients and compute the correlation between them.

Given `data`, a `list` of `dicts`, and `keys`, a `list` of `strings`, complete the function `create_cor_dict(data, key)` to find the correlation between each nutrient listed and all of the other nutrients listed. Return the result as a `dict` which maps each key to a dictionary mapping the other keys to the correlation between the parent key and the child key. For example, if `keys=['fat', 'protein', 'carbohydrates']` then the result would look something like this:

```
{'fat': {                                       # parent key is 'fat'
    'protein': 0.1854653535334078,              # parent key is 'fat' --> correlati
on between 'fat' and 'protein
    'carbohydrates': -0.6720362432582452        # correlation between 'fat' and 'ca
rbohydrates'
    },
 'protein': {                                   # 'protein'
     'fat': 0.1854653535334078,                 # 'protein' correlation w/ 'fat'
     'carbohydrates': -0.3814834566078096       # 'protein' correlation w/ 'carbohy
drates'
     },
 'carbohydrates': {                             # 'carbohydrates'
     'fat': -0.6720362432582452,                # 'carbohydrates' correlation w/ 'f
at'
     'protein': -0.3814834566078096}            # 'carbohydrates' correlation w/ 'p
rotein'
     }
```

You can assume that if `d` is a `dict` in `data`, then `d` will have `'raw_nutrients'` as a key which is mapped to a `dict` which itself maps strings to integers. For example:

```
[
    {'raw_nutrients': {'foo': 17,   'bar': 33,  'baz': 150}},
    {'raw_nutrients': {'foo': 5,                'baz': 35}},
    {'raw_nutrients': {'foo': 18,   'bar': 33,  'baz': 200}}
]
```

Each dictionary in `data` should be treated as a single observation. You can compute the correlation with the following formulas.

- n is the number of observations.
- $\bar{x}, \bar{y}$ - Means nutrient x, and nutrient y.
- $\bar{xy} = \frac{1}{n}\sum_{i=0}^{n-1} x_i y_i$
- $\sigma_x =$ **population** standard deviaiton - check your stats notes and documentation to make sure that you are calculating this correctly
- Correlation:

$$c = \frac{\bar{xy} - (\bar{x})(\bar{y})}{\sigma_x \sigma_y}$$

In [ ]:
```
### Define make_correlations
def make_correlations(data, keys):
    ### BEGIN SOLUTION
    def correlation(x, y):
        from statistics import mean, pstdev
        xy = [x_*y_ for x_, y_ in zip(x, y)]
        return (mean(xy) - mean(x)*mean(y)) / pstdev(x) / pstdev(y)

    def key2list(key):
        return [record['raw_nutrients'].get(key, 0.0) for record in data]

    return {k1:{k2: correlation(key2list(k1), key2list(k2)) for k2 in keys if
k2 != k1} for k1 in keys}
    ### END SOLUTION
```

The demo cell below should display the following output:

```
{'foo': {'bar': 0.3328398218980465, 'baz': 0.983194888209125},
 'bar': {'foo': 0.33283982189804656, 'baz': 0.31688680340974},
 'baz': {'foo': 0.983194888209125, 'bar': 0.3168868034097400 7}}
```

In [ ]:
```
### define demo inputs
### use naming convention demo_varname_ex_* to name demo variables
demo_data_ex5 = [
    {'raw_nutrients': {'foo': 12,   'bar': 33,  'baz': 100}},
    {'raw_nutrients': {'foo': 48,   'bar': 33,  'baz': 400}},
    {'raw_nutrients': {'foo': 17,   'bar': 33,  'baz': 150}},
    {'raw_nutrients': {'foo': 5,                'baz': 35}},
    {'raw_nutrients': {'foo': 18,   'bar': 33,  'baz': 200}},
    {'raw_nutrients': {'foo': 12,   'bar': 33,  'baz': 105}},
    {'raw_nutrients': {'foo': 19,               'baz': 195}},
]
demo_keys_ex5 = ['foo', 'bar', 'baz']
```

In [ ]:
```
### call demo funtion
make_correlations(demo_data_ex5, demo_keys_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', 'tester_6040.py', 'test_utils.py']:
    hash_check(f'tester_fw/{fname}', f'resource/asnlib/public/{fname}')
del hash_check
del dill
del hashlib
### END HIDDEN TESTS

from tester_fw.testers import Tester_ex5
tester = Tester_ex5()
for _ in range(20):
    try:
        tester.run_test(make_correlations)
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise

### BEGIN HIDDEN TESTS
tester = Tester_ex5(key=b'4gbCuwhFj77ZLr-nS9-6_7OKthpg6HpqGWs1vnuKkgk=', path
='resource/asnlib/publicdata/encrypted/')
for _ in range(20):
    try:
        tester.run_test(make_correlations)
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
    except:
        (input_vars, original_input_vars, returned_output_vars, true_output_va
rs) = tester.get_test_vars()
        raise
### END HIDDEN TESTS
print('Passed! Please submit.')
```

Even if your solution was incorrect or you skipped this exercise, run this cell. You would get the same result as `corr_dict` if you were to run the code below with a correct implementation of `make_correlations`.

```
make_correlations(filtered_data, ('fat', 'carbohydrates', 'protein'))
```

In [ ]:
```
### Loading results
import pickle
import os
path = './resource/asnlib/publicdata/ex5.pkl'
if not os.path.exists(path):
    with open(path, 'wb') as file:
        pickle.dump(make_correlations(filtered_data, ('fat', 'carbohydrates',
'protein')), file)
with open(path, 'rb') as file:
    corr_dict = pickle.load(file)
corr_dict
```

Exercise 6 (2 Points):

We are interested in the most common ingredients listed for foods. Instead of gathering this information on the whole data set, we want it on a category level. A good strategy for drilling down could be useful for generating category level summaries and correlations as well. The function below will transform our `basic_data` structure (a list of dictionaries) into a dictionary mapping each category to a list of dictionaries which have that category. Each of these lists will have the same structure as `basic_data`. You may (or may not) find it useful in completing exercise 6.

In [ ]:
```
def group_by_category(data):
    from collections import defaultdict
    g = defaultdict(list)
    for d in data:
        c = d['category']
        g[c].append(d)
    return dict(g)
```

Complete the function `top_ingredients` to accomplish the following:

- Parameters
  - `data` - `list` of `dicts`. You can assume that `d['list_of_ingredients']` is a `list` of strings, and `d['category']` is a string - for any `d` in `data`. Each of these `dicts` contains data on a single food.
  - `n` - `int` - number of ingredients to list
- We will say that an ingredient's "strength" within a category is given by the following:

$$x_i = \text{number of times ingredient x has been listed in position i}$$

$$\text{Strength}_x = 3x_0 + 2x_1 + x_2$$

- For each unique category (value of `d['category']`) compute the strength of all ingredients present in that category.
- Return a dictionary mapping each category to a list containing the top `n` ingredients in that category, ranked by strength in descending order. Only include ingredients which have strength greater than 0.
- In the instance of ties (two ingredients having the same strength in a category), break the tie by ranking ingredients alphabetically.
- If there are fewer than `n` ingredients - all of the ingredients should be included. There should always be `n` or fewer ingredients listed for each category.

In [ ]:
```
### Define top_ingredients
def top_ingredients(data, n=3):
    ### BEGIN SOLUTION
    from collections import defaultdict
    strength = defaultdict(lambda: defaultdict(int))
    for item in data:
        for rank, ingredient in enumerate(item['list_of_ingredients'][:min([3,
len(item['list_of_ingredients'])])]):
            strength[item['category']][ingredient] += 3-rank
    return {k:sorted(v, key=lambda x: (-v.get(x), x))[:min([len(v), n])] for
k, v in strength.items()}
    ### END SOLUTION
```

The demo cell below should display the following output:

```
{'cat0': ['bar', 'foo', 'baz', 'tux'],
 'cat1': ['bax', 'rak', 'foo'],
 'cat2': ['rah']}
```

In [ ]:
```
### define demo inputs
demo_data_ex6 = [
    {'category': 'cat0', 'list_of_ingredients':['foo', 'bar', 'baz', 'tux', 'r
ak']},
    {'category': 'cat0', 'list_of_ingredients':['bar', 'foo', 'baz', 'tux', 'b
az']},
    {'category': 'cat0', 'list_of_ingredients':['bar', 'foo', 'tux']},
    {'category': 'cat0', 'list_of_ingredients':['bar', 'baz', 'tux',]},

    {'category': 'cat1', 'list_of_ingredients':['rak', 'foo', 'bax']},
    {'category': 'cat1', 'list_of_ingredients':['rak', 'bax']},
    {'category': 'cat1', 'list_of_ingredients':['bax', 'rak', 'foo']},
    {'category': 'cat1', 'list_of_ingredients':['bax', 'foo', 'rak']},

    {'category': 'cat2', 'list_of_ingredients':['rah']},
    {'category': 'cat2', 'list_of_ingredients':['rah']},
    {'category': 'cat2', 'list_of_ingredients':['rah']},
]
```

In [ ]:
```
### call demo funtion
top_ingredients(demo_data_ex6, n=5)
```