SI 506 Last assignment

1.0 Dates

- Available: Thursday, 13 April 2023, 4:00 PM Eastern
- Due: on or before Monday, 24 April 2023, 11:59 PM Eastern

❗ No late submissions will be accepted for scoring.

2.0 Overview

The last assignment is open network, open readings, and open notes. You may refer to code in previous lecture exercises, lab exercises, and problem sets for inspiration. See the `last_assignment_overview.pdf` document for more details regarding this assignment.

3.0 Points

The last assignment is worth **1800** points and you accumulate points by passing a series of autograder tests.

❗ Per section 5.4 of the Syllabus you *must* earn a minimum `4650` points *and* attempt all challenges comprising the last assignment to earn a final course grade of `A`.

4.0 Solo effort

Please abide by the following rules:

1. The last assignment that you submit **must constitute your own work**. You are **prohibited from soliciting assistance or accepting assistance** from any person or generative artificial intelligence (AI) platform, chatbot, or tool while working to complete the programming assignment. This includes but is not limited to individual classmates, study group members, tutors, ChatGPT or Github copilot.
    - ❗ If you have formed or participated in an SI 506 study group please **suspend all study group activities** for the duration of the midterm assignment.
    - ❗ If you work with a tutor please **suspend contact** with the tutor for the duration of the assignment.
2. Likewise, you are **prohibited from assisting any other student** who is required to complete this assignment. This includes students attempting the assignment during the regular exam period, as well as those who may attempt the assignment at another time and/or place due to scheduling conflicts or other issues.

5.0 Questions

Direct all **questions** regarding the assignment to the Slack SI 506 workspace `#last_assignment` channel. Do not post code snippets to the `#last_assignment` channel. Avoid sending private direct messages (DMs) to teaching team members about the midterm challenges. Doing so is inefficient since it limits the response to a single team member and increases the likelihood of a delayed response. Surfacing your question on the `#last_assignment` channel also helps reduce message duplication.

That said, if a personal issue arises during the assignment period please send a private DM to Anthony.

6.0 Files

In line with the weekly lab exercises and problem sets you will be provided with a number of files:

File	Purpose
`README.md`	Assignment instructions
`last_assignment.py`	Program/script including a `main()` function and other definitions and statements.
`five_oh_six.py`	Module containing utility functions and statements.
`data-clone_wars_episodes.csv`	Data file.
`data-nyt_star_wars_articles.json`	Data file.
`data-wookieepedia_droids.json`	Data file.
`data-wookieepedia_people.json`	Data file.
`data-wookieepedia_planets.csv`	Data file.
`data-wookieepedia_starships.csv`	Data file.
`fxt_*.json`	Collection of test fixture files that you must match with the files you produce.

Please download the assignment files from Canvas Files as soon as they are released. This is a timed event and delays in acquiring the assignment files will shorten the time available to engage with the challenges. The clock is not your friend.

❗ *DO NOT* modify or remove the scaffolded code that we provide in the Python script or module files unless instructed to do so.

6.1 Module imports

The template file `last_assignment.py` includes the following `import` statements:

```
import copy
import five_oh_six as utl
```

The utilities module `five_oh_six.py` includes the following `import` statements:

```
import csv
import json
import requests

from urllib.parse import quote, urlencode, urljoin
```

❗ **Do not** comment out or remove these `import` statements. That said, check your `import` statements periodically. If you discover that other `import` statements have been added to your Python files remove them. In such cases, VS Code is attempting to assist you by inserting additional `import` statements based on your keystrokes. Their presence can trigger `ModuleNotFoundError` runtime exceptions when you submit your code to Gradescope.

6.2 Caching

As discussed in class, this assignment utilizes a caching workflow that eliminates redundant HTTP GET requests made to SWAPI by storing the SWAPI responses locally. Caching is implemented *fully* and all you need do is call the function `get_swapi_resource()` whenever you need to retrieve a SWAPI representation of a person/droid, planet, species, or starship, either locally from the cache or remotely from SWAPI. The cache dictionary is serialized as JSON and written to `CACHE.json` every time you run `last_assignment.py`.

❗ *Do not* call the function named `utl.get_resource` directly. Doing so sidesteps the cache and undercuts the caching optimization strategy.

7.0 Data

The Star Wars saga has spawned films, animated series, books, music, artwork, toys, games, fandom websites, cosplayers, scientific names for new organisms (e.g., *Trigonopterus yoda*), and even a Darth

Vader *grotesque* attached to the northwest tower of the Washington National Cathedral. Leading US news sources such as the New York Times cover the Star Wars phenomenon on a regular basis.

The last assignment adds yet another Star Wars-inspired artifact to the list. The data used in this assignment is sourced from the Star Wars API (SWAPI), Wookieepedia, Wikipedia, and the New York Times.

8.0 Debugging

As you write your code take advantage of the built-in `print()` function, VS code's debugger, and VS Codes file comparison feature to check your work and debug your code. See the `last_assignment_overview.pdf` for additional details and instructions.

9.0 Gradescope submissions

You may submit your solution to Gradescope as many times as needed before the expiration of the assignment time. Your **final** submission will constitute your assignment submission.

❗ You *must* submit your solution file to *Gradescope* before the expiration of exam time. Solution files submitted to the teaching team after the expiration of the assginment time will receive a score of zero (0).

If you are unable to earn full points on the assignment the teaching team will grade your submission **manually**. Partial credit **may** be awarded for submissions that fail one or more autograder tests if the teaching team (at their sole discretion) deem a score adjustment warranted.

If you submit a partial solution, feel free to include comments (if you have time) that explain what you were attempting to accomplish in the area(s) of the program that are not working properly. We will review your comments when determining partial credit.

10.0 Challenges

A long time ago in a galaxy far, far away, there occured the Clone Wars (22-19 BBY), a major conflict that pitted the Galatic Republic against the breakaway Separatist Alliance. The Republic fielded genetically modified human clone troopers commanded by members of the Jedi order against Separatist battle droids. The struggle was waged across the galaxy and, in time, inspired an animated television series entitled *Star Wars: The Clone Wars* which debuted in October 2008 and ran for seven seasons (2008-2014, 2020).

The last assignment features four groups of challenges:

Challenge 01. Implement a number of `utl.to_*()` functions employing `try` and `except` blocks that will be employed in later challenges.

Challenges 02-05. Utilize a *Clone Wars* data set that provides summary data about the animated series. You will implement a number of functions that will simplify interacting with the data in order to surface basic information about the episodes and their directors, writers, and viewership.

Challenges 06-08. Work with New York Times article data that charts the creative, cultural, and economic impact of the *Star Wars* saga both within the US and elsewhere over the past forty-six years.

Challenges 09-20. Recreates the escape of the light freighter *Twilight* from the sabotaged and doomed Separatist heavy cruiser *Malevolence* which took place during the first year of the conflict (22 BBY). Your task is to reassemble the crew of the *Twilight* and take on passengers before disengaging from the *Malevolence* and heading into deep space. The Jedi generals Anakin Skywalker and Obi-Wan Kenobi together with the astromech droid (robot) R2-D2 had earlier boarded the *Malevolence* after maneuvering the much smaller *Twilight* up against the heavy cruiser and docking via an emergency air lock. Their mission was twofold:

1. Retrieve the Republican Senator Padmé Amidala and the protocol (communications) droid C-3PO whose ship had been seized after being caught in the *Malevolence's* tractor beam, and
2. Sabotage the warship.

In these challenges you will implement functions and follow a workflow that generates a JSON document that recreates the *Twilight's* escape from the *Malevolence*.

May the Force be with You.

10.1 Challenge 01 (150 points)

Task: Implement the functions `utl.to_none()`, `utl.to_int()`, `utl.to_float()`, and `utl.to_list()`. Each function attempts to convert a passed in `value` to a more appropriate type.

10.1.1 Implement `utl.to_none()`

Replace `pass` with a code block that attempts to convert the passed in `value` to `None`. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Requirements

1. Employ `try` and `except` statements in order to handle runtime exceptions whenever an invalid conversion is attempted.

   ! **Do not** place code outside the `try` and `except` code blocks.

2. In the `try` block check if the passed in `value` can be found in `NONE_VALUES` (perform a **case insensitive** membership check). If a match is obtained return `None` to the caller; otherwise, return the `value` unchanged.

   ! Don't assume that `value` is "clean"; program defensively and remove leading/trailing spaces before checking if the "cleaned" version of the string matches a `NONE_VALUES` item.

3. If a runtime exception is encountered "catch" the exception in the `except` block and return the `value` to the caller **unchanged**.

💡 You do not need to specify specific exceptions in the `except` statement.

10.1.2 Test `utl.to_none()`

After implementing the function return to `main()`.

1. Uncomment the relevant `assert` statements and test the function.
2. If an `AssertionError` is raised, debug your code, and then retest. Repeat as necessary.

10.1.3 Implement `utl.to_float()`

Replace `pass` with a code block that attempts to convert the passed in `value` to a `float`. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Requirements

1. Employ `try` and `except` statements in order to handle runtime exceptions whenever an invalid conversion is attempted.

   ❗ **Do not** place code outside the `try` and `except` code blocks.

2. If a runtime exception is encountered "catch" the exception in the `except` block and return the `value` to the caller **unchanged**.

   💡 You do not need to specify specific exceptions in the `except` statement.

10.1.4 Test `utl.to_float()`

After implementing the function return to `main()`.

1. Uncomment the relevant `assert` statements and test the function.
2. If an `AssertionError` is raised, debug your code, and then retest. Repeat as necessary.

10.1.5 Implement `utl.to_int()`

Replace `pass` with a code block that attempts to convert the passed in `value` to an `int`. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Requirements

1. Employ `try` and `except` statements in order to handle runtime exceptions whenever an invalid conversion is attempted.

   ❗ **Do not** place code outside the `try` and `except` code blocks.

2. The function *must* convert numbers masquerading as strings, incuding those with commas that represent a thousand separator (e.g., '500,000,000') *and* those with a period that designates a fractional component (e.g., '500,000,000.9999'), to a whole number.

3. If a runtime exception is encountered "catch" the exception in the `except` block and return the `value` to the caller **unchanged**.

   💡 You do not need to specify specific exceptions in the `except` statement.

10.1.6 Test `utl.to_int()`

After implementing the function return to `main()`.

1. Uncomment the relevant `assert` statements and test the function.

2. If an `AssertionError` is raised, debug your code, and then retest. Repeat as necessary.

10.1.7 Implement `utl.to_list()`

Replace `pass` with a code block that attempts to convert the passed in `value` to a `list` using a `delimiter` if one is provided. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Requirements

1. Employ `try` and `except` statements in order to handle runtime exceptions whenever an invalid conversion is attempted.

   ❗ **Do not** place code outside the `try` and `except` code blocks.

2. If the caller provides a `delimiter` value the function *must* use it to split the `value`; otherwise, split the string without specifying a delimiter value.

   💡 Let the truth value of `delimiter` determine how you choose to split the string.

   ❗ Don't assume that `value` is "clean"; program defensively and remove leading/trailing spaces before attempting to convert the string to a list.

3. If a runtime exception is encountered "catch" the exception in the `except` block and return the `value` to the caller **unchanged**.

   💡 You do not need to specify specific exceptions in the `except` statement.

10.1.8 Test `utl.to_list()`

After implementing the function return to `main()`.

1. Uncomment the relevant `assert` statements and test the function.
2. If an `AssertionError` is raised, debug your code, and then retest. Repeat as necessary.

10.2 Challenge 02 (80 points)

Task: Refactor (e.g., modify) the function `utl.read_csv_to_dicts()` to use a **list comprehension** and then call the function to read a CSV file that contains information about the *Clone Wars* episodes. Then implement the function `has_viewer_data()` that checks whether or not an episode possesses viewership information.

💡 This challenge involves a list of nested dictionaries. Use the built-in function `print()` to explore one of nested dictionaries or call the function `utl.write_json()` in `main()`, encode the data as JSON, and write it to a "test" JSON file so that you can view the list of dictionaries more easily.

10.2.1 Refactor `utl.read_csv_to_dicts()`

Examine the commented out code in `utl.read_csv_to_dicts()` function (**do not** uncomment). Reimplement the function by writing code inside the `with` block that retrieves an instance of the `csv.DictReader` and then employs a list comprehension to traverse the lines in the reader object and return a new list of line elements to the caller.

❗ Review lecture notes and code solution files if you have forgotten how to write a list comprehension. If you are unsuccessful in your endeavors uncomment the code in `utl.read_csv_to_dicts()` and get the function working so that you can continue with the assignment.

Requirements

1. You are limited to writing two (2) lines of code.
    1. Line 01 assigns an instance of `csv.DictReader` to a variable named `reader`.
    2. Line 02 returns a new list of `reader` "line" elements to the caller using **a list comprehension**.
2. You *must* employ existing variable names that appear in the commented out code when writing your list comprehension.

10.2.2 Test `utl.read_csv_to_dicts()`

After refactoring `utl.read_csv_to_dicts()` return to `main()`.

1. Call the function and retrieve the data contained in the file `data-clone_wars_episodes.csv`.
2. Assign the return value to a variable named `clone_wars_episodes`.

10.2.3 Implement `has_viewer_data()`

Replace `pass` with a code block that checks whether or not an individual *Clone Wars* episode possesses viewership information. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Requirements

1. The function *must* compute the truth value of the passed in episode's "episode_us_viewers_mm" key-value pair, returning either `True` or `False` to the caller.

   💡 Recall that a function can include more than one `return` statement. That said, you can also employ Python's ternary operator to solve this challenge with a single line of code.

10.2.4 Call `has_viewer_data()`

After implementing the function return to `main()`.

1. Test your implementation of `has_viewer_data()` by counting the number of episodes in the `clone_wars_episodes` list that possess a "episode_us_viewers_mm" numeric value. Employ a loop, a conditional statement, and the accumulator pattern to accomplish the task. Whenever the return value of `has_viewer_data()` equals `True` increment your episode count by 1.

   💡 Recall that a function call is considered an expression and `if` statements are composed of one or more expressions.

2. The number of episodes that possess an "episode_us_viewers_mm" viewership value equals eighty-eight (`88`). If your loop does not accumulate this total, recheck both your implementation of `has_viewer_data()` and your `for` loop and loop block `if` statement.

10.3 Challenge 03 (100 points)

Task: Implement a function that converts *Clone Wars* episode string values to more appropriate types.

10.3.1 Implement `convert_episode_values()`

Replace `pass` with a code block that converts specifed string values to more appropriate types. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Requirements

1. The function accepts a list of nested "episode" dictionaries. You *must* implement a nested loop to perform the value type conversions.
    - Outer loop: passed in `episodes` list of nested dictionaries.
    - Inner loop: individual "episode" dictionary items.
2. Employ `if-elif-else` conditional statements to convert the values encountered to more appropriate types. Utilize the conditional statements to perform the following operations on each key and value encountered:
    1. Check if the value is a member of `none_values`. If `True` assign `None` to the converted value's associated key.
    2. Convert certain episode string values to more appropriate types by passing the value to the appropriate `utl.to_*()` function and assigning the return value to the converted value's associated key:

Current type	to	New type
series_season_num (`str`)	->	series_season_num (`int` | `None`)
series_episode_num (`str`)	->	series_episode_num (`int` | `None`)
season_episode_num (`str`)	->	season_episode_num (`int` | `None`)
episode_prod_code (`str`)	->	episode_prod_code (`float` | `None`)
episode_us_viewers_mm (`str`)	->	episode_us_viewers_mm (`float` | `None`)
episode_writers (`str`)	->	episode_writers (`list` | `None`)

3. After the outer loop terminates return the list of mutated dictionaries to the caller.

10.3.2 Call `convert_episode_values()`

After implementing the function, return to `main()`.

1. Call the function `convert_episode_values()` and pass it the following argument: `clone_wars_episodes` and the constant `NONE_VALUES`. Assign the return value to `clone_wars_episodes`.
2. Call the function `utl.write_json()` and write `clone_wars_episodes` to the file `stu-clone_wars-episodes_converted.json`. Compare your file to the test fixture file `fxt-clone_wars-episodes_converted.json`. Both files *must* match, line-for-line, and character-for-character.

10.4 Challenge 04 (90 points)

Task: Implement a function that retrieves the most viewed *Clone Wars* episode(s).

10.4.1 Implement `get_most_viewed_episode()`

Replace `pass` with a code block that finds the most viewed *Clone Wars* episode(s) in the data set. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Requirements

1. The function *must* return a list of one or more episodes from the passed in `episodes` list that feature the highest recorded viewership. Include in the list only those episodes that tie for the highest recorded viewership. If no ties exist only one episode will be returned in the list.

   ❗ **Ignore** episodes with no viewership value.

2. Delegate to `has_viewer_data()` the task of checking whether an episode contains a *truthy* "episode_us_viewers_mm" value. The function needs to check the truth value of "episode_us_viewers_mm" before you attempt to compare the current "episode_us_viewers_mm" value to the previous value.

   💡 Assign two local "accumulator" variables to the viewer count and the top episode(s).

10.4.2 Call `get_most_viewed_episode()`

After implementing the function return to `main()`.

1. Call the function and pass it the following argument: `clone_wars_episodes`.

2. Assign the return value to `most_viewed_episode`. Pass the variable to `print()` and review the terminal output. If the list contains the following elements proceed to the next challenge; if not, recheck your code.

```
[
   {
      'series_title': 'Star Wars: The Clone Wars',
      'series_season_num': 1,
      'series_episode_num': 2,
      'season_episode_num': 2,
      'episode_title': 'Rising Malevolence',
      'episode_director': 'Dave Filoni',
      'episode_writers': ['Steven Melching'],
      'episode_release_date': 'October 3, 2008',
      'episode_prod_code': 1.07,
      'episode_us_viewers_mm': 4.92
     },
   {
```

```
        'series_title': 'Star Wars: The Clone Wars',
        'series_season_num': 7,
        'series_episode_num': 134,
        'season_episode_num': 13,
        'episode_title': 'The Lecturers',
        'episode_director': 'Anthony Whyte',
        'episode_writers': ['Anthony Whyte', 'Chris Teplovs'],
        'episode_release_date': 'May 7, 2020',
        'episode_prod_code': 7.25,
        'episode_us_viewers_mm': 4.92
        }
]
```

10.5 Challenge 05 (90 points)

Task: Construct a dictionary of *Clone Wars* directors along with a count of the number of episodes each directed. Sort by episode count (descending) and the director name (ascending).

10.5.1 Implement `count_episodes_by_director()`

Replace `pass` with a code block that returns a dictionary of key-value pairs that associate each director in the `episodes` list with a count of the episodes that they are credited with directing. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Requirements

1. The function *must* accumulates episode counts for each director listed in the `episodes` list. Create an empty accumulator dictionary to hold the director-count key-value pairs (variable name your choice).

2. The director's name comprises the key and the associated value represents a count of the number of episodes that they directed.

   ```
   {
      < director_name_01 >: < episode_count >,
      < director_name_02 >: < episode_count >,
      ...
   }
   ```

3. Employ a nested loop to solve this challenge. As you loop over each episode you will need to extract the director(s) from a string into a list and then loop over each in turn. In most cases you will encounter a single director but exceptions exist. For example Season 7, episode 128 was directed by Saul Ruiz and Bosco Ng.

```
series_title,series_season_num,series_episode_num,season_episode_num,e
pisode_title,episode_director,episode_writers,episode_release_date,epi
sode_prod_code,episode_us_viewers_mm
...
Star Wars: The Clone Wars,7,128,7,Dangerous Debt,"Saul Ruiz, Bosco
Ng","Dave Filoni, Charles Murray","April 3, 2020",6.07,
...
```

4. Implement conditional logic in the inner loop block to ensure each director's episode counts are properly tabulated per the following rules:

    1. Check if the local accumulator dictionary possesses a key-value pair for the director. If `True`, increment the associated value; otherwise add a new key-value pair to the dictionary.

    2. When accumulating values, increment the director's episode count by `1.0` (a `float`) **if, and only if,** the director is the only person credited with directing the episode.

    3. Otherwise, if two or more directors are credited with directing the episode allocate a fraction of `1.0` to each director's episode count. This value is calculated by dividing `1.0` by the number of directors credited with directing the episode.

        For example, if two directors are credited with directing the episode, each director's count is incremented by `0.5`; if three directors are credited with directing the episode, each director's count is incremented by `0.33`.

        Employ simple division when incrementing the count. Consider how you can employ the list of directors in the equation that you write.

10.5.2 Call `count_episodes_by_director()`

After implementing the function return to `main()`.

1. Call the function and pass `clone_wars_episodes` to it as the argument.

2. Assign the return value to `director_episode_counts`.

3. **Uncomment** the provided dictionary comprehension that employs the built-in function `sorted()` and a `lambda` function to sort the episode counts by the count (descending) and the director's last name.

4. Call the function `utl.write_json()` and write the *sorted* `director_episode_counts` to the file `stu-clone_wars-director_episode_counts.json`. Compare your file to the test fixture file `fxt-clone_wars-director_episode_counts.json`. Both files *must* match, line-for-line, and character-for-character.

10.6 Challenge 06 (75 points)

Task: Implement the function `get_news_desks()`.

10.6.1 Implement `get_news_desks()`

Replace `pass` with a code block that returns a list of New York Times "news desks" sourced from the passed in `articles` list. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

💡 Each article dictionary contains a "news_desk" key-value pair.

Requirements

1. The list of news desk names returned by the function *must not* contain any duplicate elements. Accumulate the values carefully.
2. The function must delegate to the function `utl.to_none()` the task of converting "news_desk" values that equal "None" (a string) to `None`. Only news_desk values that are "truthy" (i.e., not None) are to be returned in the list.

   💡 There are a number of articles with a "news_desk" value of "None". Exclude this value from the list by passing each "news_desk" value to `utl.to_none()` and assigning the return value to a local variable. You can filter out the `None` values with a truth value test.
3. The function *must* return a new version of the accumulator list sorted alphanumerically.

10.6.2 Call `get_news_desks()`

After implementing the function return to `main()`.

1. Call the function `utl.read_json()` and retrieve the New York Times article data in the file `./data-nyt_star_wars_articles.json`. Assign the return value to `articles`.
2. Test your implementation of `get_news_desks()` by calling the function and passing to it the following arguments: `articles` and the constant `NONE_VALUES`. Assign the return value to the variable `news_desks`.
3. Call the function `utl.write_json()` and write `news_desks` to the file `stu-nyt_news_desks.json`. Compare your file to the test fixture file`fxt-nyt_news_desks.json`. The files *must* match line for line, indent for indent, and character for character.

10.7 Challenge 07 (100 points)

Task: Implement the function `group_articles_by_news_desk()`.

10.7.1 Implement `group_articles_by_news_desk()`

Replace `pass` with a code block that returns a dictionary of "news desk" key-value pairs that group the passed in `articles` by their parent news desk drawn from the `news_desks` list. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Requirements

1. Implement a nested loop. Review the `data-nyt_star_wars_articles.json` and `stu-nyt_news_desks.json` files and decide which list should be traversed by the outer loop and which list should be traversed by the inner loop.

   💡 The news desk name provides the link between the two lists.

2. Assign an empty list to a local variable. You will accumulate article dictionaries in this list and then assign the list to its "parent" news desk key. There are three locations in the function block where this initial variable assignment could be placed: outside the loops, inside the outer loop, or inside the inner loop. Choose wisely.

3. Each article dictionary added to its parent news desk list represents a "thinned" version of the original. The keys to employ and their order is illustrated by the example below:

   ```
   {
      "web_url":
   "https://www.nytimes.com/2016/10/20/business/media/lucasfilm-sues-
   jedi-classes.html",
      "headline_main": "Classes for Jedis Run Afoul of the Lucasfilm
   Empire",
      "news_desk": "Business",
      "byline_original": "By Erin McCann",
      "document_type": "article",
      "material_type": "News",
      "abstract": "A man whose businesses offers private lessons and
   certifications for fine-tuning lightsaber skills is operating without
   the permission of the “Star Wars” owner.",
      "word_count": 865,
      "pub_date": "2016-10-19T13:26:21+0000"
   }
   ```

   ❗ Certain keys such as "headline_main", "byline_original", and "material_type" are not found in the original New York Times dictionaries. Hopefully, the names provide a sufficient hint about which values to map (i.e., assign) to each key.

10.7.2 Call `group_articles_by_news_desk()`

After implementing the function return to `main()`.

1. Call the function and pass it `news_desks` and `articles` as arguments. Assign the return value to the variable `news_desk_articles`.

2. Call the function `utl.write_json()` and write `news_desk_articles` to the file `stu-nyt_news_desk_articles.json`. Compare your file to the test fixture file `fxt-nyt_news_desk_articles.json`. The files *must* match line for line, indent for indent, and character for character.

10.8 Challenge 08 (90 points)

Task Implement the function `calculate_articles_mean_word_count()`.

10.8.1 `calculate_articles_mean_word_count()` function

Replace `pass` with a code block that returns the mean (e.g., average) word count of the passed in list of `articles` less any articles with a word count of zero (0). Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

mean: central value of a set of values that is determined by calculating *the sum of the values divided by the number of values*.

Requirements

1. The function *must* calculate the mean word count of the passed in articles **excluding** from the calculation all articles with a "word_count" value of zero (`0`) or `None`.

2. The function *must* maintain a count of the number of articles evaluated and a count of the total words accumulated from each article's "word_count" value. Assign the running counts to two local "accumulator" variables.

3. The function *must* check the truth value of each article's "word_count" before attempting to increment the article count and total words count. If the truth vallue of the "word_count" is `False` the article is excluded from the count.

4. The function *must* **round** the mean value to the second (2nd) decimal place before returning the value to the caller.

10.8.2 Call `calculate_articles_mean_word_count()`

After implementing the function return to `main()`.

1. Create an empty dictionary named `mean_word_counts`. You will use it to accumulate mean words counts.

2. Loop over the `news_desk_articles` key-value pairs. Write a conditional statement inside the loop block that checks if the current key is a member of the `ignore` news desks tuple. If the key is **not** a

member call the function `calculate_articles_mean_word_count()` and pass it the list of articles mapped (i.e., assigned) to the key.

3. Inside the loop add a new key-value pair to `mean_word_counts` consisting of the current key and the return value of the call to `calculate_articles_mean_word_count()`. Below is one of the key-value pairs added to `mean_word_counts` that your code *must* produce:

```
{
    ...
    "Obits": 876.62,
    ...
}
```

4. Call the function `utl.write_json()` and write `mean_word_counts` to the file `stu-nyt_news_desk_mean_word_counts.json`. Compare your file to the test fixture file `fxt-nyt_news_desk_mean_word_counts.json`. The files *must* match line-for-line and character-for-character.

10.9 Challenge 09 (60 points)

Task: Implement the utility function `utl.get_nested_dict()`.

10.9.1 Implement `utl.get_nested_dict()`

Replace `pass` with a code block that utilizes a `filter` string to return a nested dictionary from the passed in `data` list. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

The function is employed to traverse lists of nested dictionaries sourced from the following files in search of a particular dictionary representation of a Star Wars droid, person, planet, or starship:

- `data-wookieepedia_droids.json`
- `data-wookieepedia_people.json`
- `data-wookieepedia_planets.csv`
- `data-wookieepedia_starships.csv`

💡 Familiarize yourself with the Wookieepedia-sourced data files before commencing the remaining challenges.

Requirements

1. Loop over the nested dictionaries in the passed in `data` list.

2. Inside the loop block utilize the passed in `key` name to identify the key-value pair in the nested dictionary to evaluate. The value mapped to the `key` *must* be compared to the passed in `filter`

value. If an **exact match** is obtained the nested dictionary is returned to the caller; otherwise `None` is returned.

10.9.2 Call `utl.get_nested_dict()`

After implementing the function return to `main()`.

1. Call the function `utl.read_csv_to_dicts()` and retrieve the supplementary Wookieepedia planet data in the file `data-wookieepedia_planets.csv`. Assign the return value to `wookiee_planets`.
2. Call the function `utl.get_nested_dict()` and pass to it the following arguments: `wookiee_planets`, the key "name", and the *lowercase* string "Dagobah".
3. Assign the return value to the variable `wookiee_dagobah`.
4. Call the function `utl.write_json()` and write `wookiee_dagobah` to the file `stu-wookiee_dagobah.json`. Compare your file to the test fixture file `fxt-wookiee_dagobah.json` The files *must* match line for line, indent for indent, and character for character.
5. Call `utl.get_nested_dict()` a second time and pass to it the following arguments: `wookiee_planets`, the key "system", and the string "Al'Har system".
6. Assign the return value to the variable `wookiee_haruun_kal`.
7. Call the function `utl.write_json()` and write `wookiee_haruun_kal` to the file `stu-wookiee_haruun_kal.json`. Compare your file to the test fixture file `fxt-wookiee_haruun_kal.json`. The files *must* match line for line, indent for indent, and character for character.

10.10 Challenge 10 (115 points)

Task: Implement the functions `utl.to_gravity_value()` and `create_planet()`.

10.10.1 Implement `utl.to_gravity_value()`

Replace `pass` with a code block that attempts to convert a planet's "gravity" value to a float by first removing the "standard" unit of measure substring (if it exists) before converting the remaining number to a float. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

💡 Note that "gravity" values vary from planet to planet. The following examples illustrate the challenge:

```
{
    'name': Tatooine,
    ...
    'gravity': '1 standard',
    ...
```

```
}

{
   'name': Dagobah,
    ...,
    'gravity': 'N/A',
    ...
}

{
    'name': Haruun Kal,
    ...
    'gravity': '0.98',
    ...
}
```

Requirements

1. Employ `try` and `except` statements in order to handle runtime exceptions whenever an invalid conversion is attempted.

   exclamation: **Do not** place code outside the `try` and `except` code blocks.

2. In the `try` block evaluate whether or not the substring "standard" is part of `value` (perform a *case insensitive* check). If found, remove the substring and pass the new (truncated) string to the function `utl.to_float()` in order to convert the numeric part of the `value` to a float. If not found pass `value` directly to `utl.to_float()`.

   ❗ Don't assume that `value` is "clean"; program defensively and remove leading/trailing spaces before attempting to convert the "cleaned" version of the string to a float.

3. If a runtime exception is encountered "catch" the exception in the `except` block and return the `value` to the caller **unchanged**.

   💡 You do not need to specify specific exceptions in the `except` statement.

10.10.2 Test `utl.to_gravity_value()`

After implementing the function return to `main()`.

1. Uncomment the relevant `assert` statements and test the function.
2. If an `AssertionError` is raised, debug your code, and then retest. Repeat as necessary.

10.10.3 Implement `create_planet()`

Replace `pass` with a code block that returns a dictionary representation of a planet based on the passed in `swapi_data` dictionary. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

💡 The workflow outlined below illustrates the general creational pattern applied to each droid, person, planet, species, and starship encountered in the following challenges.

Workflow

1. Retrieve a SWAPI representation of the entity if one exists.
2. Update the SWAPI entity with Wookieepedia data if provided.
3. Return a "thinned" version of the dictionary that retains a subset of the original SWAPI/Wookieepedia key-value pairs, substituting in new keys when required and converting certain values to more appropriate types, including replacing certain string values with `None`.
4. Serialize the new dictionary as JSON and write the object to a file (i.e., check your work).

❗ The SWAPI entity will serve as the default representation of the entities that feature in the assignment. The Wookieepedia data will be used to enrich the SWAPI data with new and updated key-value pairs.

Requirements

1. Assign an empty dictionary to a local variable (name your choice). This "planet" dictionary will be employed to accumulate new key-value pairs sourced from `swapi_data`.
2. If an optional `wookiee_data` dictionary is provided by the caller, **update** `swapi_data` with the `wookiee_data` key-value pairs **prior** to creating the new dictionary representation of the planet.
3. The `keys` dictionary includes droid, person, planet, species, and starship dictionaries. Each of these nested dictionaries specify the following new dictionary attributes:
    - the `swapi_data` key-value pairs to be mapped to the new dictionary.
    - the order in which the `swapi_data` key-value pairs are mapped to the new dictionary.
    - the key names to be used in the new dictionary.

    For example, each key in the `keys` "planet" dictionary corresponds to a key in `swapi_data`. Each value in the `keys` "planet" dictionary represents the (new) key name to be used in the new planet dictionary.

```
keys = {
      ...
      "planet": {
          "url": "url", # old key: new key
          ...,
          "orbital_period": "orbital_period_days", # old key: new key
          ...
      },
```

```
    ...
}
```

4. Access the `keys` "planet" dictionary and loop over its keys and values. Inside the loop block employ `if-elif-else` conditional statements to populate the new planet dictionary with key-value pairs sourced from `swapi_data`.

   Each key in the `keys` "planet" dictionary maps to a `swapi_data` key-value pair. The associated "planet" value provides the key to utilize in the new dictionary.

   Convert `swapi_data` values to more appropriate types as outlined below in the mappings table. Strings found in `none_values` must be converted to `None` **irrespective of case**. Delegate type conversions to the various `utl.to_*()` functions. Then map the `swapi_data` value to the new key when assigning the key-value pair to the new dictionary.

   For example, if the `keys` "planet" key equals "diameter" perform the following actions:

   1. access the corresponding `swapi_data` "diameter" value, converting it as required to a more appropriate type or `None`.
   2. assign the (converted) value to the new dictionary using the "diameter_km" value as the new key.

```
    < some condition >
       new_dict[< value >] = swapi_data[< key >] # pass expression to
utl.to_*() if required
   < some other condition >
       ...
   ...
```

💡 Review the mappings carefully. Opportunities exist to reduce the number of `elif` statements by leveraging the commonalities across the keys.

Planet mappings

`swapi_data` dictionary	**to**	**"planet" dictionary**
url (`str`)	->	url (`str`)
name (`str`)	->	name (`str` | `None`)
region (`str`)	->	region (`str` | `None`)
sector (`str`)	->	sector (`str` | `None`)
suns (`str`)	->	suns (`int` | `None`)
moons (`str`)	->	moons (`int` | `None`)
orbital_period (`str`)	->	orbital_period_days (`float` | `None`)

`swapi_data` dictionary	to	"planet" dictionary
diameter (`str`)	->	diameter_km (`int` | `None`)
gravity (`str`)	->	gravity_std (`float` | `None`)
climate (`str`)	->	climate (`list` | `None`)
terrain (`str`)	->	terrain (`list` | `None`)
population (`str`)	->	population (`int` | `None`)

10.10.4 Test `create_planet()`

After implementing `create_planet()` return to `main()`.

1. Call the function `get_swapi_resource()` and retrieve a SWAPI representation of the planet Tatooine. Make use of the appropriate constant to simplify construction of the URL. Access the "Tatooine" dictionary which is stored in the response object and assign the value to `swapi_tatooine`.
2. Call the function `create_planet()` and pass to it the following arguments: `keys`, and `swapi_tatooine`. Assign the return value to a variable named `tatooine`.
3. Call the function `utl.write_json()` and write `tatooine` to the file `stu-tatooine-v1p0.json`. Compare your file to the test fixture file `fxt-tatooine-v1p0.json`. Both files *must* match line for line, indent for indent, and character for character.
4. Test the function `create_planet()` a second time. First, call the function `utl.get_nested_dict()` and pass to it the following arguments: `wookiee_planets`, the key "name", and the `swapi_tatooine` *name* value. Assign the return value to `wookiee_tatooine`.
5. Call the function `create_planet()` and pass to it the following arguments: `keys`, `swapi_tatooine`, and `wookiee_tatooine`. Assign the return value to the variable `tatooine`.
6. Call the function `utl.write_json()` and write `tatooine` to the file `stu-tatooine-v1p1.json`. Compare your file to the test fixture file `fxt-tatooine-v1p1.json`. Both files *must* match line for line, indent for indent, and character for character.

10.11 Challenge 11 (120 points)

Task: Implement the functions `utl.to_year_era()` and `create_droid()`.

10.11.1 Implement `utl.to_year_era()`

Replace `pass` with a code block that attempts to separate the Galactic standard calendar year and era (e.g., 896BBY, 24ABY) value into a dictionary comprising year and era key-value pairs.

Requirements

1. Employ `try` and `except` statements in order to handle runtime exceptions whenever an invalid conversion is attempted. **Do not** place code outside the `try` and `except` code blocks.
2. In the `try` block use slicing to access the year and era segments of the string.

   💡 Note that while the year segment's length varies (e.g., 896, 19, 0) the era segment of the Galactic calender date string comprises three characters: "BBY" or "ABY". Keep this in mind as you design your slicing expressions.
3. Before mapping the sliced segments to a dictionary, you *must* first check if the "year" segment of the `value` is a number by employing the appropriate `str` method. If the substring **is numeric**, return a dictionary literal that maps the necessary slicing expressions to "year" and "era" keys as values. Structure the dictionary as follows:

   ```
   {'year': < year > (int), 'era': < era > (`str`)}
   ```
4. Otherwise, if the "year" segment is not considered numeric return the `value` to the caller **unchanged**.
5. If the year segment is numeric, convert the "year" segment to an integer by passing it as the argument to the function `to_int()`. Call the function from within the dictionary literal.
6. If a runtime exception is encountered "catch" the exception in the `except` block and return the `value` to the caller **unchanged**.

   💡 You do not need to specify specific exceptions in the `except` statement.

10.11.2 Test `utl.to_year_era()`

After implementing the function return to `main()`.

1. Uncomment the relevant `assert` statements and test the function.
2. If an `AssertionError` is raised, debug your code, and then retest. Repeat as necessary.

10.11.3 Implement `create_droid()`

Replace `pass` with a code block that returns a dictionary representation of a driod (e.g., a sentient robot) based on the passed in `swapi_data` dictionary. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Requirements

1. Assign an empty dictionary to a local variable (name your choice). This "droid" dictionary will be employed to accumulate new key-value pairs sourced from `swapi_data`.

2. If an optional `wookiee_data` dictionary is provided by the caller, **update** `swapi_data` with the `wookiee_data` key-value pairs **prior** to creating the new dictionary representation of the droid.

3. The `keys` dictionary includes droid, person, planet, species, and starship dictionaries. Access the `keys` "droid" dictionary and loop over its keys and values. Inside the loop block employ `if-elif-else` conditional statements to populate the new droid dictionary with key-value pairs sourced from `swapi_data`.

   Each key in the `keys` "droid" dictionary maps to a `swapi_data` key-value pair. The associated "droid" value provides the key to utilize in the new dictionary.

   Convert `swapi_data` values to more appropriate types as outlined below in the mappings table. Strings found in `none_values` must be converted to `None` **irrespective of case**. Delegate type conversions to the various `utl.to_*()` functions. Then map the `swapi_data` value to the new key when assigning the key-value pair to the new dictionary.

    Review the mappings carefully. Opportunities exist to reduce the number of `elif` statements by leveraging the commonalities across the keys.

   #### Droid mappings

`swapi_data` dictionary	to	"droid" dictionary
url (`str`)	->	url (`str`)
name (`str`)	->	name (`str` | `None`)
model (`str`)	->	model (`str` | `None`)
manufacturer (`str`)	->	manufacturer (`str` | `None`)
create_year (`str`)	->	create_date (`dict` | `None`)
height (`str`)	->	height_cm (`float` | `None`)
mass (`str`)	->	mass_kg (`float` | `None`)
equipment (`str`)	->	equipment (`list` | `None`)
instructions (`str`)	->	instructions (`list` | `None`)

10.11.4 Test `create_droid()`

After implementing `create_droid()` return to `main()`.

1. Call the function `get_swapi_resource()` and retrieve a SWAPI representation of the astromech droid R2-D2. Make use of the appropriate constant to simplify construction of the URL. Access the "R2-D2" dictionary which is stored in the response object and assign the value to `swapi_r2_d2`.

2. Call the function `utl.read_json()` and retrieve the supplementary Wookieepedia droid data in the file `data-wookieepedia_droids.json`. Assign the return value to `wookiee_droids`.

3. Call the function `utl.get_nested_dict()` and pass to it the arguments required to retrieve the Wookieepedia dictionary representation of R2-D2. Assign the return value to `wookiee_r2_d2`.

4. Call the function `create_droid()` and pass to it the following arguments: `keys`, `swapi_r2_d2`, and `wookiee_r2_d2`. Assign the return value to the variable `r2_d2`.

5. Call the function `utl.write_json()` and write `r2_d2` to the file `stu-r2_d2.json`. Compare your file to the test fixture file `fxt-r2_d2.json`. Both files *must* match line for line, indent for indent, and character for character.

10.12 Challenge 12 (60 points)

Task: Implement the function `create_species()`.

10.12.1 Implement `create_species()`

Replace `pass` with a code block that returns a dictionary representation of a species based on the passed in `swapi_data` dictionary. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Requirements

1. Assign an empty dictionary to a local variable (name your choice). This "species" dictionary will be employed to accumulate new key-value pairs sourced from `swapi_data`.

2. If an optional `wookiee_data` dictionary is provided by the caller, **update** `swapi_data` with the `wookiee_data` key-value pairs **prior** to creating the new dictionary representation of the species.

3. The `keys` dictionary includes droid, person, planet, species, and starship dictionaries. Access the `keys` "species" dictionary and loop over its keys and values. Inside the loop block employ `if-elif-else` conditional statements to populate the new species dictionary with key-value pairs sourced from `swapi_data`.

   Each key in the `keys` "species" dictionary maps to a `swapi_data` key-value pair. The associated "species" value provides the key to utilize in the new dictionary.

   Convert `swapi_data` values to more appropriate types as outlined below in the mappings table. Strings found in `none_values` must be converted to `None` **irrespective of case**. Delegate type conversions to the various `utl.to_*()` functions. Then map the `swapi_data` value to the new key when assigning the key-value pair to the new dictionary.

    Review the mappings carefully. Opportunities exist to reduce the number of `elif` statements by leveraging the commonalities across the keys.

Species mappings

`swapi_data` dictionary	to	"species" dictionary
url (`str`)	->	url (`str`)
name (`str`)	->	name (`str` | `None`)
classification (`str`)	->	classification (`str` | `None`)
designation (`str`)	->	designation (`str` | `None`)
average_lifespan (`str`)	->	average_lifespan_yrs (`int` | `None`)
average_height (`str`)	->	average_height_cm (`float` | `None`)
language (`str`)	->	language (`str`)

10.12.2 Call `create_species()`

After implementing `create_species()` return to `main()`.

1. Call the function `get_swapi_resource()` and retrieve a SWAPI representation of the human species. Make use of the appropriate constant to simplify construction of the URL. Access the "human" species dictionary which is stored in the response object and assign the value to `swapi_human_species`.
2. Call the function `create_species()` and pass to it the following arguments: `keys` and `swapi_human_species`. Assign the return value to the variable `human_species`.
3. Call the function `utl.write_json()` and write `human_species` to the file `stu-human_species.json`. Compare your file to the test fixture file `fxt-human_species.json`. Both files *must* match line for line, indent for indent, and character for character.

10.13 Challenge 13 (65 points)

Task: Implement the function `get_homeworld()` The function will be utilized by the function `create_person()` to enrich a person's home planet information.

10.13.1 Implement `get_homeworld()`

Replace `pass` with a code block that attempts to return a dictionary representation of a person's home planet. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Requirements

1. Retrieve a SWAPI representation of a species using the provided `swapi_url`. Assign the return value to a variable (name your choice).
2. Create a local variable (name your choice) to which the Wookieepedia-sourced dictionary, if retrieved, can be assigned. Assign it the default value `None`.
3. If an optional `planets` list is provided by the caller, delegate to the function `utl.get_nested_dict()` the task of retrieving a Wookieepedia-sourced representation of the planet. Assign the return value to the local variable.
4. Call the function `create_planet()` and pass to it all required *and* optional arguments. Return the function's return value to the caller.

10.13.2 Test `get_homeworld()`

After implementing the function return to `main()`.

1. Retrieve a SWAPI representation of the Jedi knight Anakin Skywalker. Pass the Jedi knight's name as the search `params` value. Make use of the appropriate constant to simplify construction of the URL.
2. Access the "Anakin" dictionary from the response object and assign the value to `swapi_anakin`.
3. Call the function `get_homeworld()` and pass to it the following arguments: `keys`, the `swapi_anakin` dictionary's "homeworld" value, and `wookiee_planets`. Assign the return value to a variable named `swapi_anakin_homeworld`.
4. Call the function `utl.write_json()` and write `swapi_anakin_homeworld` to the file `stu-anakin_homeworld.json`. Compare your file to the test fixture file `fxt-anakin_homeworld.json`. Both files *must* match line for line, indent for indent, and character for character.

10.14 Challenge 14 (65 points)

Task: Implement the function `get_species()` The function will be utilized by the function `create_person()` to enrich a person's species information.

10.14.1 Implement `get_species()`

Replace `pass` with a code block that attempts to return a dictionary representation of a person's species. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Requirements

1. Retrieve a SWAPI representation of a species using the provided `swapi_url`. Assign the return value to a variable (name your choice).

2. Create a local variable (name your choice) to which the Wookieepedia-sourced dictionary, if retrieved, can be assigned. Assign it the default value `None`.
3. If an optional `species` list is provided by the caller, delegate to the function `utl.get_nested_dict()` the task of retrieving a Wookieepedia-sourced representation of the species. Assign the return value to the local variable.
4. Call the function `create_species()` and pass to it the required *and* optional arguments. Return the function's return value to the caller.

10.14.2 Test `get_species()`

After implementing the function return to `main()`.

1. Call the function `get_species()` and pass to it the following arguments: `keys` and the first element in the `swapi_anakin` dictionary's "species" list. Assign the return value to a variable named `swapi_anakin_species`.
2. Call the function `utl.write_json()` and write `swapi_anakin_species` to the file `stu-anakin_species.json`. Compare your file to the test fixture file `fxt-anakin_species.json`. Both files *must* match line for line, indent for indent, and character for character.

10.15 Challenge 15 (100 points)

Task: Implement the function `create_person()`.

10.15.1 Implement `create_person()`

Replace `pass` with a code block that returns a new person dictionary based on the passed in `swapi_data` dictionary. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Requirements

1. Assign an empty dictionary to a local variable (name your choice). This "person" dictionary will be employed to accumulate new key-value pairs sourced from `swapi_data`.
2. If an optional `wookiee_data` dictionary is provided by the caller, **update** `swapi_data` with the `wookiee_data` key-value pairs **prior** to creating the new dictionary representation of the person.
3. The `keys` dictionary includes droid, person, planet, species, and starship dictionaries. Access the `keys` "person" dictionary and loop over its keys and values. Inside the loop block employ `if-elif-else` conditional statements to populate the new person dictionary with key-value pairs sourced from `swapi_data`.

Each key in the `keys` "person" dictionary maps to a `swapi_data` key-value pair. The associated "person" value provides the key to utilize in the new dictionary.

Convert `swapi_data` values to more appropriate types as outlined below in the mappings table. Strings found in `none_values` must be converted to `None` **irrespective of case**. Delegate type conversions to the various `utl.to_*()` functions. Then map the `swapi_data` value to the new key when assigning the key-value pair to the new dictionary.

Retrieving a dictionary representation of the person's home planet is delegated to the function `get_homeworld()`. Retrieving a dictionary representation of the person's species is delegated to the function `get_species()`. The `planets` and `planet_key` values are passed directly to `get_homeworld()` while the `species` and `species_key` values are passed directly to `get_species()`.

 Review the mappings carefully. Opportunities exist to reduce the number of `elif` statements by leveraging the commonalities across the keys.

Person mappings

`swapi_data` dictionary	to	"person" dictionary
url (`str`)	->	url (`str`)
name (`str`)	->	name (`str` | `None`)
birth_year (`str`)	->	birth_date (`dict` | `None`)
height (`str`)	->	height_cm (`float` | `None`)
mass (`str`)	->	mass_kg (`float` | `None`)
homeworld (`str`)	->	homeworld (`dict` | `None`)
species (`list`)	->	species (`dict` | `None`)
force_sensitive (`str`)	->	force_sensitive (`str`)

10.15.2 Call create_person()

After implementing the function return to `main()`.

1. Call the function `utl.read_json()` and retrieve the supplementary Wookieepedia person data in the file `data-wookieepedia_people.json`. Assign the return value to a variable named `wookiee_people`.
2. Call the function `utl.get_nested_dict()` and pass to it the arguments required to retrieve the "Anakin Skywalker" dictionary in `wookiee_people`. Assign the return value to a variable named `wookiee_anakin`.
3. Call the function `create_person()` and pass to it the following arguments: `keys`, `swapi_anakin`, `wookiee_anakin`, and `wookiee_planets`. Assign the return value to the variable `anakin`.

4. Call the function `utl.write_json()` and write `anakin` to the file `stu-anakin_skywalker.json`. Compare your file to the test fixture file `fxt-anakin_skywalker.json`. Both files *must* match line for line, indent for indent, and character for character.

5. Next, create an "enriched" dictionary representation of the Jedi master and general Obi-Wan Kenobi.

   Utilize the same "creational" workflow employed earlier to create Anakin Skywalker:

   1. Retrieve a SWAPI dictionary representation of Obi-Wan Kenobi (search on "obi-wan kenobi"). We recommend assigning the return value to a variable named `swapi_obi_wan`.

   2. Retrieve the Wookieepedia representation of Obi-Wan. Consider adopting the same variable naming format adopted for Anakin (e.g., `wookiee_obi_wan`).

   3. Call the function `create_person()` and pass to it the arguments it needs to create a "thinned" dictionary representation of Obi-Wan. Consider assigning the return value to a variable named `obi_wan`.

6. Call the function `utl.write_json()` and write `obi_wan` to the file `stu-obi_wan_kenobi.json`. Compare your file to the test fixture file `fxt-obi_wan_kenobi.json`. Both files *must* match line for line, indent for indent, and character for character.

10.16 Challenge 16 (80 points)

Task: Implement the function `create_starship()`.

10.16.1 Implement `create_starship()`

Replace `pass` with a code block that returns a new starship dictionary based on the passed in `data` dictionary. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Requirements

1. Assign an empty dictionary to a local variable (name your choice). This "starship" dictionary will be employed to accumulate new key-value pairs sourced from `swapi_data`.

2. If an optional `wookiee_data` dictionary is provided by the caller, **update** `swapi_data` with the `wookiee_data` key-value pairs **prior** to creating the new dictionary representation of the starship.

3. The `keys` dictionary includes droid, person, planet, species, and starship dictionaries. Access the `keys` "starship" dictionary and loop over its keys and values. Inside the loop block employ `if-elif-else` conditional statements to populate the new starship dictionary with key-value pairs sourced from `swapi_data`.

   Each key in the `keys` "starship" dictionary maps to a `swapi_data` key-value pair. The associated "starship" value provides the key to utilize in the new dictionary.

Convert `swapi_data` values to more appropriate types as outlined below in the mappings table. Strings found in `none_values` must be converted to `None` **irrespective of case**. Delegate type conversions to the various `utl.to_*()` functions. Then map the `swapi_data` value to the new key when assigning the key-value pair to the new dictionary.

💡 Review the mappings carefully. Opportunities exist to reduce the number of `elif` statements by leveraging the commonalities across the keys.

Starship mappings

`swapi_data` dictionary	to	"starship" dictionary
url (`str`)	->	url (`str`)
name (`str`)	->	name (`str` | `None`)
model (`str`)	->	model (`str` | `None`)
starship_class (`str`)	->	starship_class (`str` | `None`)
manufacturer (`str`)	->	manufacturer (`str` | `None`)
length (`str`)	->	length_m (`float` | `None`)
hyperdrive_rating (`str`)	->	hyperdrive_rating (`float` | `None`)
MGLT (`str`)	->	max_megalight_hr (`int` | `None`)
max_atmosphering_speed (`str`)	->	max_atmosphering_speed_kph (`int` | `None`)
crew (`str`)	->	crew_size (`int` | `None`)
crew_members (`None`)	->	crew_members (`list` | `None`)
passengers (`str`)	->	max_passengers (`int` | `None`)
passengers_on_board (`None`)	->	passengers_on_board (`list` | `None`)
cargo_capacity (`str`)	->	cargo_capacity_kg (`int` | `None`)
consumables (`str`)	->	consumables (`str`)
armament (`str`)	->	armament (`list` | `None`)

10.16.2 Call `create_starship()`

❗ The starship *Twilight* is sourced from Wookieepedia only. No SWAPI representation of the light freighter exists.

After implementing `create_starship()` return to `main()`.

1. Call the `utl.read_csv_to_dicts()` function and retrieve the supplementary Wookieepedia starship data in the file `data-wookieepedia_starships.csv`. Assign the return value to `wookiee_starships`.

2. Call `utl.get_nested_dict()` and pass to it the arguments required to retrieve the light freighter named *Twilight* in `wookiee_starships`. Assign the return value to a variable named `wookiee_twilight`.

3. Call the function `create_starship()` and pass the following arguments: `keys` and `wookiee_twilight` (pass the starship dictionary as the second (`2nd`) argument). Assign the return value to a variable named `twilight`.

4. Call the function `utl.write_json()` and write `twilight` to the file `stu-twilight.json`. Compare your file to the test fixture file `fxt-twilight.json`. Both files *must* match line for line, indent for indent, and character for character.

10.17 Challenge 17 (60 points)

> R2 are you quite certain that the ship is in this direction? This way looks potentially dangerous. *C-3PO*

Task: Implement the function `board_passengers()`. Get Senator Padmé Amidala, the protocol droid C-3PO, and the astromech droid R2-D2 aboard the *Twilight* as passengers.

10.17.1 Implement `board_passengers()`

Replace `pass` with a code block that assigns a limited number of passengers to a list. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Requirements

1. The passengers *must* be passed in a `list` to the `board_passengers()` function.

2. The number of passengers permitted to board a starship or other vehicle is limited by the provided `max_passengers` value. If the number of passengers attempting to board exceeds `max_passengers` only the first `n` passengers (where `n` = `max_passengers`) are permitted to board the vessel.

   For example, if a starship's `max_passengers` value equals `10` and `20` passengers attempt to board the starship, only the first `10` passengers are permitted aboard the vessel.

10.17.2 Call `board_passengers()`

After implementing `board_passengers()` return to `main()`.

1. Create a dictionary representation of the Galactic senator Padmé Amidala. Pass the senator's name as a `params` value. Utilize the same "creational" workflow employed to create the dictionary

representations of `anakin` and `obi_wan`. Use the following variable names to represent Padmé Amidala.

- `swapi_padme` (assigned to the SWAPI dictionary)
- `wookiee_padme` (assigned to the Wookieepedia dictionary)
- `padme` (assigned to the `create_person()` return value)

2. Call the function `utl.write_json()` and write `padme` to the file `stu-padme_amidala.json`. Compare your file to the test fixture file `fxt-padme_amidala.json`. Both files *must* match line for line, indent for indent, and character for character.

3. Create a dictionary representation of the protocol droid named C-3PO. Pass the droid's name as a `params` value. Utilize the same "creational" workflow employed to create `r2_d2`. Consider using the following variable names to represent C-3PO.

   - `swapi_c_3po` (assigned to the SWAPI dictionary)
   - `wookiee_c_3po` (assigned to the Wookieepedia dictionary)
   - `c_3po` (assigned to the `create_droid()` return value)

4. Call the function `utl.write_json()` and write `c_3po` to the file `stu-c_3po.json`. Compare your file to the test fixture file `fxt-c_3po.json`. Both files *must* match line for line, indent for indent, and character for character.

5. Next, call the function `board_passengers()` and pass the `twilight` starship's "max_passengers" value and a list of passengers comprising `padme`, `c_3po`, and `r2_d2` (in that order) as arguments.

6. Map (i.e., assign) the return value to the `twilight` dictionary's "passengers_on_board" key.

   Consider testing your function by passing additional passengers to it in excess of the permitted "max_passengers" value. Consider creating dictionary representations of the following Jedi masters:

   - Mace Windo (https://swapi.py4e.com/api/people/51/)
   - Plo Koon (https://swapi.py4e.com/api/people/58/)
   - Shaak Ti (https://swapi.py4e.com/api/people/78/)
   - Yoda (https://swapi.py4e.com/api/people/20/)

   If the function `board_passengers()` is implemented correctly only `padme`, `c_3po`, `r2_d2`, and three of the Jedi masters should be able to board the `twilight`. You can retrieve both SWAPI and Wookieepedia dictionary representations of each to use for testing. Do this on separate lines and remember to comment out these lines before submitting your solution to Gradescope.

   JSON fixture files of these Jedi have also been included in the file dump.

10.18 Challenge 18 (100 points)

> Let's get back to the ship. Power up the engines R2. *Anakin Skywalker*

Task: Implement the function `assign_crew_members()`. Assign Anakin Skywalker and Obi-Wan Kenobi to the *Twilight* as crew members.

10.18.1 Implement `assign_crew_members()`

Replace `pass` with a code block that assigns personnel by position (e.g., pilot, copilot) to a starship using a dictionary comprehension. Review the function's docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Requirements

1. To earn full credit you *must* create the "crew_members" dictionary by writing a **dictionary comprehension** on a **single line**.

   ❗ If necessary write a `for` loop that adds the position/crew member key-value pairs to an accumulation dictionary named `crew_members`. Get it working first and then convert it to a dictionary comprehension.

   💡 Avoid looping over the passed in lists. Instead **loop over a sequence of numbers** and think carefully about the appropriate stop value to employ in order to limit the number of loop iterations. Utilize the sequence of numbers to pair `crew_position` and `personnel` elements by their matching index position.

2. The crew positions (e.g., 'pilot', 'copilot') and personnel (e.g., Anakin Skywalker, Obi-Wan Kenobie) must be passed in separate `crew_positions` and `personnel` lists to the function `assign_crew_members()`.

3. Assume that each ship requires a full compliment of personnel to crew the vessel. In other words, if a starship's "crew_size" equals three (`3`) you *must* pass to `assign_crew_members()` three crew positions and three personnel.

   Assume too that the maximum number of crew members that can be assigned to a starship is limited by the starship's "crew_size" value. No additional crew members are permitted to be assigned to the starship even if included in the `crew_positions` and `personnel` lists. This limitation *must* be imposed from within the dictionary comprehension.

   For example, if a starship's "crew_size" value equals `3` but `4` crew positions/personnel are passed to the function only the first `3` crew positions/personnel are permitted to be added as key-value pairs to the crew members dictionary.

4. The passed in `crew_positions` and `personnel` lists *must* contain the same number of elements. The individual `crew_positions` and `personnel` elements are then paired by index position and stored in a dictionary structured as follows:

   ```
   {< crew_position[0] >: < personnel[0] >, < crew_position[1] >: <
   personnel[1] >, ...}
   ```

10.18.2 Call `assign_crew_members()`

After implementing `assign_crew_members()` return to `main()`.

1. Call the function `assign_crew_members()` and pass the Twilight's "crew_size" value, a crew positions list comprising the following string elements: "pilot" and "copilot", and a personnel list comprising `anakin` and `obi_wan`.

2. Map (i.e., assign) the return value to the `twilight` dictionary's "crew_members" key.

   💡 Consider testing your function by passing to it an additional crew position (e.g., navigator) and crew member (e.g, Mace Windo in excess of the permitted "crew_size". Do this on a separate line and remember to comment out the line before submitting your solution to Gradescope.

10.18.3 Issue instructions to R2-D2

Create a list containing Anakin's "Power up the engines" order (a string) and map (i.e., assign) the list to the droid `r2_d2's` "instructions" key.

10.19 Challenge 19 (150 points)

Task: Sort `wookiee_planets` and then issue commands to R2-D2 to chart a course to the planet Naboo. Also demonstrate that you can sort a list of dictionaries using a `lambda` function.

10.19.1 Sort `wookiee_planets`

1. Write a **single-line list comprehension** that transforms each Wookieepedia-sourced planet dictionary in the `wookiee_planets` list by passing each planet referenced in the comprehension to the function `create_planet()`. Assign the new list to a variable named `planets`.

   ❗ if your list comprehension triggers a `KeyError` exception, check your implementation `create_planet()`. The function is likely attempting to access a key in the planet dictionary that does not exist. Recall that there is a friendly `dict` method for dealing with such issues; refactor (i.e., revise) your function block accordingly.

2. Perform an **in_place** sort of the `planets` list passing to it as the `key` function a `lambda` function that sorts the planets **by name**. **Reverse** the sort so that the planets are sorted by name in **descending order**.

3. Call the function `utl.write_json()` and write `planets` to the file `stu-planets_sorted_name.json`. Compare your file to the test fixture file `fxt-planets_sorted_name.json`. Both files *must* match line for line, indent for indent, and character for character.

10.19.2 Issue instructions to R2-D2

1. Call the function `utl.get_nested_dict()` and pass it the following arguments: `planets`, `diameter_km`, and the integer `12120`. Assign the return value to a variable named `naboo`.
2. Access the `naboo` dictionary's "region" and "sector" values and include the names in a formatted string literal (f-string) structured as follows:

   ```
   "Plot course for Naboo, < region >, < sector >"
   ```

3. Add the f-string to `r2_d2's` "instructions" list so that *Twilight* can chart a course to the planet Naboo, Padmé Amidala's home world.

10.19.3 Sort planets by diameter and name

💡 If you get stuck on sorting `planets` by "diameter_km" and "name", pause your sorting work and proceed to the final task ("Escape from the Malevolvence") and complete it. Then return and restart this standalone task.

1. Employ the built-in function `sorted()` and a `lambda` function to sort `planets` by the following attributes:

Key	Value	Order
diameter_km	`int` | `None`	descending
name	ascending	`str`

2. You *must* write your `lambda` expression using the **ternary operator** when sorting on "diameter_km" because several planets lack a known diameter and in consequence `None` has been mapped (i.e., assigned) to their "diameter_km" key.
3. Assign the return value of `sorted()` to a variable named `planets_diameter_km`.

   ❗ Write the entire statement on a single line to facilitate auto grader testing:

   ```
   planets_diameter_km = < expression >
   ```

4. Call the function `utl.write_json()` and write `planets_diameter_km` to the file `stu-planets_sorted_diameter.json`. Compare your file to the test fixture file `fxt-planets_sorted_diameter.json`. Both files *must* match line for line, indent for indent, and character for character.

10.20 Challenge 20 (50 points)

> R2 release the docking clamp. *Anakin Skywalker*

Task: Instruct the astromech droid R2-D2 to release the docking clamp and detach the *Twilight* from the *Malevolence*.

10.20.1 Escape from the Malevolence

With our heroes on board the *Twilight* and the engines fired, the light freighter detaches itself from the stricken heavy cruiser *Malevolence* and departs to rejoin the Republican fleet before heading to Naboo.

1. Add Anakin's order "Release the docking clamp" to `r2_d2's` "instructions" key-value pair.
2. Call the function `utl.write_json()` and write `twilight` to the file `stu-twilight_departs.json`. Compare your file to the test fixture file `fxt-twilight_departs.json`. Both files *must* match line for line, indent for indent, and character for character.

If the files match your job is done. Never mind that Separatist starfighters are in hot pursuit of the *Twilight*—declare victory!

Congratulations on completing SI 506.

FINIS