SI 506 Last assignment

1.0 Dates

- Available: Thursday, 14 April 2022, 4:00 PM Eastern
- Due: on or before Tuesday, 26 April 2022, 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 **1700** points and you accumulate points by passing a series of autograder tests.

4.0 Solo effort

❗ You are prohibited from soliciting assistance or accepting assistance from any person while completing the programming assignment. The code that you submit *must* be your own work. Likewise, you are prohibited from assisting any other student required to complete this assignment. This includes those 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.

❗ If you have formed or participate in an SI 506 study group please suspend all study group activities for the duration of the last assignment.

5.0 Files

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

1. `swapi.md`: assignment instructions
2. `swapi.py`: script including a `main()` function and other definitions and statements
3. `sw_utils.py`: module containing utility functions and constants
4. One or more `*.csv` and/or `*.json` files that contain assignment data
5. One or more `fxt_*.json` 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.

5.1 Module imports

The template file `swapi.py` includes a single `import` statement:

```
import sw_utils as utl
```

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

```
import copy
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.

6.0 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.

Besides retrieving data from SWAPI you will also access information locally from the following data files:

- `clone_wars.csv`
- `clone_wars_episodes.csv`
- `nyt_star_wars_articles.json`
- `wookieepedia_droids.json`
- `wookieepedia_people.json`
- `wookieepedia_planets.csv`
- `wookieepedia_starships.csv`

7.0 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.

8.0 Gradescope submissions

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

9.0 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).

Challenge 01 features a small *Clone Wars* data set that provides general information about each season. You will use it to demonstrate your indexing and slicing skills.

Challenge 02 implement a number of `utl.convert_to_*` functions employing `try` and `except` blocks that will be employed in later challenges.

Challenges 03-06 utilize a second *Clone Wars* data set that provides summary data about the first two seasons of 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 07-09 work with New York Times article data (1977-2022) that charts the creative, cultural, and economic impact of the *Star Wars* saga both within the US and elsewhere over the past forty-five years.

Challenges 10-18 recreate 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*.

Caching: This assignment utilizes a local "cache" dictionary located in the utilities module that eliminates redundant HTTP GET requests made to SWAPI by storing the SWAPI responses locally. The caching workflow is implemented *fully* and all you need do is call the function `utl.get_resource` whenever you need to retrieve a SWAPI representation of a person/droid, planet, species, or starship.

❗ *Do not* call `get_swapi_resource` directly. Doing so sidesteps the cache and undercuts the built-in caching optimization strategy.

The cache dictionary is written to a JSON file every time you run `swapi.py`:

```
# PERSIST CACHE (DO NOT COMMENT OUT)
utl.write_json(utl.CACHE_FILEPATH, utl.cache)
```

The cache JSON document will remain empty until you start working on Challenge 12. Thereafter the cache file will record resources retrieved from SWAPI.

May the Force be with You.

9.1 Challenge 01

Task: Refactor (e.g., modify) the function `utl.read_csv` to use a list comprehension and then call the function to read a small CSV file that contains general information about the first seven (`7`) seasons of the *The Clone Wars* animated series.

❗ Inspect `clone_wars.csv` visually in order to complete Challenge 01.

9.1.1 refactor `utl.read_csv`

Examine the commented out code in `utl.read_csv` function (**do not** uncomment). Reimplement the function by writing code in the `with` block that retrieves a `csv` reader object and employs a **list comprehension** to traverse the rows in the reader object and return a new list of row elements to the caller.

Requirements

1. You are limited to writing two (2) lines of code.
    1. Line 01 assigns the "reader" object returned by calling `csv.reader` to a variable named `reader`.
    2. Line 02 returns a new list of `reader` "row" 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 (i.e., `reader`, `row`).

9.1.2 Call function

After refactoring `utl.read_csv` return to `main`. Call the function and retrieve the data contained in the file `clone_wars.csv`. Assign the return value to a variable named `clone_wars`.

Call the built-in function `print()` and pass `clone_wars` to it. Confirm that the data is stored in a list of lists. Comment out `print()` once confirmed.

❗ 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` and get the function working so that you can continue with the assignment.

9.1.3 Indexing and slicing

❗ Using a `for` loop is neither required **nor permitted**. You *must* also **exclude** *The Clone Wars* "headers" list element when slicing the list.

1. In `main` employ slicing to access the subset of all *The Clone Wars* seasons that feature twenty-two (22) episodes. Assign the list to a variable named `clone_wars_22`.
2. In `main` employ slicing to access the subset of *The Clone Wars* seasons that either started *or* ended during the year 2012. Assign the list to a variable named `clone_wars_2012`.
3. In `main` employ indexing to access *The Clone Wars* season URL string that *does not* include the substring "_Season_" in it. Assign the string to a variable named `clone_wars_url`.
4. In `main` employ slicing to access all *The Clone Wars* even-numbered seasons. Assign the list to a variable named `clone_wars_even_num_seasons`.

9.2 Challenge 02

Task: Implement the functions `utl.convert_to_none`, `utl.convert_to_int`, `utl.convert_to_float`, and `utl.convert_to_list`. Each function attempts to convert a passed in `value` to a more appropriate type.

9.2.1 `utl.convert_to_none` function

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. The function *must* employ `try` and `except` statements in order to handle runtime exceptions whenever an invalid type conversion is attempted. **Do not** place code outside the `try`/`except` code blocks.
2. The function *must* perform a **case-insensitive** comparison between the passed in `value` and the items in the `utl.NONE_VALUES` tuple constant:

   ```
   NONE_VALUES = ('', 'n/a', 'none', 'unknown')
   ```

   If a match is obtained inside the `try` block the function will return `None` to the caller, otherwise, the value will be returned 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 `utl.NONE_VALUES` item.

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

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

9.2.2 `utl.convert_to_int` function

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. The function *must* employ `try` and `except` statements in order to handle runtime exceptions whenever an invalid type conversion is attempted. **Do not** place code outside the `try`/`except` code blocks.

2. The function *must* convert numbers masquerading as strings, incuding those with commas that represent a thousand separator:

   - "5" -> 5
   - "50,000" -> 50000
   - "5,000,000" -> 5000000

3. If a runtime exception is encountered the `except` block will "catch" the exception, pass the `value` to the function `convert_to_none`, and then return the value returned by `convert_to_none` to the caller.

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

9.2.3 `utl.convert_to_float` function

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. The function *must* employ `try` and `except` statements in order to handle runtime exceptions whenever an invalid type conversion is attempted. **Do not** place code outside the `try`/`except` code blocks.

2. If a runtime exception is encountered the `except` block will "catch" the exception, pass the `value` to the function `convert_to_none`, and then return the value returned by `convert_to_none` to the caller.

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

9.2.4 `utl.convert_to_list` function

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.

💡 Model `convert_to_list` on the other type conversion functions. This challenge involves adjusting your implementation per the hints below so that the function can handle converting strings to lists or return the passed in value *unchanged* if a runtime exception is encountered.

Requirements

1. The function *must* employ `try` and `except` statements in order to handle runtime exceptions whenever an invalid type conversion is attempted. **Do not** place code outside the `try`/`except` code blocks.

2. If the passed in `value` matches an item in the `utl.NONE_VALUES` tuple (**case-insensitive comparison**), the function *must* return `None`.

3. If a delimiter value is provided the function will use it to split the `value`; otherwise, the string will be split without specifying a delimiter value.

   💡 Note that the function's `delimiter` parameter defaults to `None`. You *must* check the truth value of `delimiter` in the function block. If `True` pass the delimiter value to the appropriate `str` method; otherwise rely on the `str` method's default behavior.

   ❗ 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 list.

4. If a runtime exception is encountered the `except` block will "catch" the exception and the `value` will be returned to the caller *unchanged*.

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

9.2.5 Call the functions

After implementing the four `utl.convert_to_*` functions return to `swapi.py`. In `main` test the functions by calling each 2-3 times. Pass a value that can be converted and returned as a new type and a couple of values that will trigger an exception and be returned unchanged. You can utilize the built-in function `print()` to output each value to the terminal as illustrated by the following example:

```
print(f"\nconvert_to_none -> None = {utl.convert_to_none(' N/A ')}")
print(f"\nconvert_to_none -> None = {utl.convert_to_none('')}")
print(f"\nconvert_to_none -> no change = {utl.convert_to_none('Yoda ')}")
print(f"\nconvert_to_none -> no change = {utl.convert_to_none(5.5)}")
print(f"\nconvert_to_none -> no change = {utl.convert_to_none((1, 2, 3))}")

print(f"\nconvert_to_int -> int = {utl.convert_to_int('506 ')}")
print(f"\nconvert_to_int -> None = {utl.convert_to_int(' unknown')}")
print(f"\nconvert_to_int -> no change = {utl.convert_to_int([506, 507])}")
```

```
# Devise additional tests yourself for convert_to_float and convert_to_list
```

9.3 Challenge 03

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 first two seasons of *Clone Wars* episodes. Then implement the function `has_viewer_data` that checks whether or not an episode possesses viewership information.

💡 This challenge has you working with a list of nested dictionaries. Use the built-in function `print()` to explore the nested dictionary 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.

9.3.1 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 in the `with` block that retrieves an instance of `csv.DictReader` and employs a list comprehension to traverse the lines in the reader object and return a new list of line elements to the caller.

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 (i.e., `reader`, `line`).

9.3.2 Call function

After refactoring `utl.read_csv_to_dicts` return to `main`. Call the function and retrieve the data contained in the file `clone_wars_episodes.csv`. Assign the return value to a variable named `clone_wars_episodes`.

❗ 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.

9.3.3 `has_viewer_data` function

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.

9.3.4 Call function

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

❗ The number of episodes that possess an "episode_us_viewers_mm" viewership value equals twenty-five (`25`). If your loop does not accumulate this value, recheck both your implementation of `has_viewer_data` and your `for` loop and loop block `if` statement.

9.4 Challenge 04

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

9.4.1 `convert_episode_values` function

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 and you *must* implement a nested loop to perform the value type conversions.

   - Outer loop: passed in `episodes` list of nested dictionaries
   - Inner loop: "episode" dictionary items

2. Implement the necessary conditional logic to convert the passed in dictionary values to the specified types, delegating to the `utl.convert_*` functions the task of converting strings to either `int`, `float`, or `list` per the conversion chart below.

Conversion	value(s)	Delegate to	Notes
`str` to `int`	'series_season_num', 'series_episode_num', 'season_episode_num'	sw_utils.convert_to_int()	Blank values are converted to `None` if `utl.convert_to_none` is called by `utl.convert_to_int`.

Conversion	value(s)	Delegate to	Notes
`str` to `float`	'episode_prod_code', 'episode_us_viewers_mm'	sw_utils.convert_to_float()	Blank values are converted to `None` if `utl.convert_to_none` is called by `utl.convert_to_float`.
`str` to `list`	'episode_writers'	sw_utils.convert_to_list()	

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

9.4.2 Call function

After implementing `convert_episode_values`, return to `main`. Call the function passing the `clone_wars_episodes` list as the argument. Assign the return value to `clone_wars_episodes`.

9.4.3 Write to file

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.

9.5 Challenge 05

Task: Implement functions to retrieve the most viewed episode(s) of the first two seasons of *The Clone Wars*.

9.5.1 `get_most_viewed_episode` function

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 with the highest recorded viewership. Includes 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. Ignores 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. You need to check if "episode_us_viewers_mm" has a value 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).

9.5.2 Call function

After implementing `get_most_viewed_episode` return to `main`. Call the function and pass `clone_wars_episodes` to it as the argument. Assign the return value to `most_viewed_episode`. 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': 2, 'series_episode_num': 45,
      'season_episode_num': 23,
      'episode_title': 'Test Record',
      'episode_director': 'Anthony Whyte',
      'episode_writers': ['Anthony Whyte', 'Chris Teplovs'],
      'episode_release_date': 'May 7, 2010',
      'episode_prod_code': 2.22,
      'episode_us_viewers_mm': 4.92
   }
]
```

9.6 Challenge 06

Task: Construct a dictionary of directors and a count of the number of episodes each directed during the first two seasons of *The Clone Wars*.

9.6.1 `count_episodes_by_director` function

Replace `pass` with a code block that returns a dictionary of key-value pairs that associate each director in the `episodes` list of nested dictionaries 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 `eposides` list.
2. The director's name comprises the key and the associated value a count of the number of episodes they directed. Implement conditional logic to ensure that each director is assigned a key and the episode counts are properly tabulated and assigned as the value.

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

9.6.2 Call function

After implementing `count_episodes_by_director` return to `main`. Call the function and pass `clone_wars_episodes` to it as the argument. Assign the return value to `director_episode_counts`.

9.6.2 Write to file

Call the function `utl.write_json` and write `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.

9.7 Challenge 07

Task: Implement the function `get_nyt_news_desks`.

9.7.1 `get_nyt_news_desks` function

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.convert_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 eight (8) articles with a "news_desk" value of "None". Exclude this value from the list by passing each "news_desk" value to `utl.convert_to_none` and assigning the return value to a local variable. You can filter out the `None` values with a truth value test.

9.7.2 Call function

After implementing `get_nyt_news_desks` return to `main`. Call the function `utl.read_json` and retrieve the New York Times article data in the file `./nyt_star_wars_articles.json`. Assign the return value to `articles`.

Test your implementation of `get_nyt_news_desks` by calling the function and passing to it the argument `articles`. Assign the return value to the variable `news_desks`.

9.7.3 Write to file

Check your work. 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 and character-for-character.

9.8 Challenge 08

Task: Implement the function `group_nyt_articles_by_news_desk`.

9.8.1 `group_nyt_articles_by_news_desk` function

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 `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.

9.8.2 Call function

After implementing `nyt_star_wars_articles.json` return to `main`. Call the function and pass it `news_desks` and `articles` as arguments. Assign the return value to the variable `news_desk_articles`.

9.8.3 Write to file

Check your work. 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 and character-for-character.

9.9 Challenge 09

Task Implement the function `calculate_articles_mean_word_count`

9.9.1 `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 of zero (0).
2. Accumulate the news desk article word counts and assign the running count to a local variable.
3. Maintain a local count of the number of news desk articles with a word count of zero (`0`). You will need to subtract this number from the total number of articles passed to the function to ensure that the divisor reflects the actual number of articles upon which to compute the mean. An `if-else` block is your friend here.
4. You *must* **round** the mean value to the second (2nd) decimal place before returning the value to the caller.

9.9.2 Call function

After implementing `calculate_articles_mean_word_count` return to `main`. Create an empty dictionary named `mean_word_counts`. You will use it to accumulate mean words counts.

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.

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": 823.14,
   ...
}
```

9.9.3 Write to file

Check your work. 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.

9.10 Challenge 10

Task: Implement the function `utl.convert_gravity_value`.

9.10.1 `utl.convert_gravity_value` function

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. The function *must* employ `try` and `except` statements in order to handle runtime exceptions whenever an invalid type conversion is attempted. **Do not** place code outside the `try`/`except` code blocks.

2. The function *must* **remove** the substring "standard" if it exists anywhere in the passed in `value` *irrespective of case*. In other words lowercase, mixed case, and uppercase versions of the substring must be removed.

3. If the substring exists in `value`, remove it and return a version of `value` that contains only the numeric portion of the string.

   💡 a handy `str` method exists for locating substrings in a string.

   ❗ 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.

4. The function *must* delegate to the function `utl.convert_to_float` the task of converting the "numeric" version of `value` to a float. The return value of `utl.convert_to_float` is then returned to the caller.

5. If a runtime exception is encountered the `except` block will "catch" the exception, pass the `value` to the function `utl.convert_to_none`, and then return the value returned by `utl.convert_to_none` to the caller.

9.10.2 Call function

After implementing `convert_gravity_value` return to `main`. Test your implementation by calling the function from inside `print()` and passing to it different test values such as "1 standard", "N/A", and the list `0.98`.

```
print(f"\nconvert_gravity_value -> float = {utl.convert_gravity_value('1
standard')}")
print(f"\nconvert_gravity_value -> None =
{utl.convert_gravity_value('N/A')}")
print(f"\nconvert_gravity_value -> float =
```

```
{utl.convert_gravity_value('0.98')}")
```

9.11 Challenge 11

Task: Implement the function `get_wookieepedia_data`.

9.11.1 `get_wookieepedia_data` function

Replace `pass` with a code block that utilizes a `filter` string to return a nested dictionary from the passed in `wookiee_data` list if the dictionary's "name" value matches the `filter` value. 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 can be 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:

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

Requirements

1. The function must perform a **case insenstitive** comparison between the passed in `filter` value and each nested dictionary's "name" value. If a match is obtained it returns the nested dictionary to the caller.

2. If no match is obtained the function returns `None` to the caller.

9.11.2 Call function

In `main` call the `utl.read_csv_to_dicts` function and retrieve the supplementary Wookieepedia planet data in the file `wookieepedia_planets.csv`. Assign the return value to `wookiee_planets`.

Call the function `get_wookieepedia_data` and pass to it as arguments `wookiee_planets` and the *lowercase* string "dagobah". Assign the return value to the variable `wookiee_dagobah`.

Call the function a second time and pass to it as arguments `wookiee_planets` and the *uppercase* string "HARUUN KAL". Assign the return value to the variable `wookiee_haruun_kal`.

9.11.3 Write to file

Check your work. Call the function `utl.write_json` and write `wookiee_dagobah` to the file `stu-wookiee_dagobah.json`. Call `utl.write_json` a second time and write `wookiee_haruun_kal` to the file `stu-wookiee_haruun_kal.json`. Compare your file to the test fixture files `fxt-wookiee_dagobah.json` and `fxt-wookiee_haruun_kal.json`. The files *must* match line-for-line and character-for-character.

9.12 Challenge 12

Task: Implement the function `create_planet`.

💡 This challenge's workflow illustrates the general creational pattern applied to each droid, person, planet, species, and starship encountered in later challenges.

- retrieve SWAPI data
- combine with Wookieepedia data
- create new dictionary instance that retains a subset of the passed in key-value pairs, occasionally substituting in new keys and converting certain values to more appropriate types (e.g., `str` to `int`).
- write to file (i.e., check your work)

❗ The SWAPI data 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.

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

9.12.1 `create_planet` function

Replace `pass` with a code block that returns a new planet 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

Certain `data` values require special handling and are subject to the following type conversion rules:

1. Convert all `data` values to `None` that match any of the `utl.NONE_VALUES` items (case-insensitive match of strings stripped of leading/trailing spaces). This can be accomplished by judicious use of the `utl.convert_to_*` functions.

2. Convert other `data` values to `int`, `float` or `list` as specified in the table below.

   ❗ The new dictionary may contain keys that differ from the passed in `data` dictionary keys.

`data`	Convert to	Notes
suns (`str`)	suns (`int`)	
moons (`str`)	moons (`int`)	
orbital_period (`str`)	orbital_period_days (`float`)	
diameter (`str`)	diameter_km (`int`)	
gravity (`str`)	gravity_std (`float`)	
climate (`str`)	climate (`list`)	
terrain (`str`)	terrain (`list`)	
population (`str`)	population (`int`)	

9.12.2 Create the planet Tatooine

After implementing `create_planet` return to `main`. Call the function `get_resource` and retrieve a SWAPI representation of the planet Tatooine. Access the "Tatooine" dictionary which is stored in the response object and assign the value to `swapi_tatooine`.

💡 The `sw_utils` module includes a SWAPI "planets" URL constant that you can pass as the `url` argument. If you need help constructing the `params` argument review the lecture notes and code.

Call `get_wookieepedia_data` passing it the appropriate arguments and retrieve the "Tatooine" dictionary in `wookiee_planets`. Assign the return value to `wookiee_tatooine`. Check the truth value of `wookiee_tatooine`. If "truthy" update `swapi_tatooine` with `wookiee_tatooine`.

Call the function `create_planet()` and pass the updated `swapi_tatooine` as the argument. Assign the return value to a variable named `tatooine`.

9.12.3 Write to file

Check your work. Call the function `utl.write_json` and write `tatooine` to the file `stu-tatooine.json`. Compare your file to the test fixture file `fxt-tatooine.json`. Both files *must* match line-for-line and character-for-character.

9.13 Challenge 13

Task: Implement the function `create_droids`.

9.13.1 `create_droid` function

Replace `pass` with a code block that returns a new droid 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

Certain `data` values require special handling and are subject to the following type conversion rules:

1. Convert all `data` values to `None` that match any of the `utl.NONE_VALUES` items (case-insensitive match of strings stripped of leading/trailing spaces). This can be accomplished by judicious use of the `utl.convert_to_*` functions.

2. Convert other `data` values to `int`, `float` or `list` as specified in the table below.

`data`	`Droid`	**Notes**
height (`str`)	height_cm (`float`)	
mass (`str`)	mass_kg (`float`)	
equipment (`str`)	equipment (`list`)	Check delimiter in `wookieepedia_droids.json`

9.13.2 Create the droid R2-D2

After implementing `create_droid` return to `main`. Call the `utl.read_json` function and retrieve the supplementary Wookieepedia droid data in the file `wookieepedia_droids.json`. Assign the return value to `wookiee_droids`.

Call the function `get_resource` and retrieve a SWAPI representation of the astromech droid R2-D2. Access the "R2-D2" dictionary which is stored in the response object and assign the value to `swapi_r2_d2`.

💡 The `sw_utils` module includes a SWAPI "people" URL constant that you can pass as the `url` argument (Droids are considered people in SWAPI). If you need help constructing the `params` argument review the lecture notes and code.

Call `get_wookieepedia_data` passing it the appropriate arguments and retrieve the "R2-D2" dictionary in `wookiee_droids`. Assign the return value to `wookiee_r2_d2`. Check the truth value of `wookiee_r2_d2`. If "truthy" update `swapi_r2_d2` with `wookiee_r2_d2`.

Call the function `create_droid()` and pass the updated `swapi_r2_d2` as the argument. Assign the return value to a variable named `r2_d2`.

9.13.3 Write to file

Check your work. 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 and character-for-character.

9.14 Challenge 14

Task: Implement the function `create_species`.

9.14.1 `create_species` function

Replace `pass` with a code block that returns a new species 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

Certain `data` values require special handling and are subject to the following type conversion rules:

1. Convert all `data` values to `None` that match any of the `utl.NONE_VALUES` items (case-insensitive match of strings stripped of leading/trailing spaces). This can be accomplished by judicious use of the `utl.convert_to_*` functions.

2. Convert other `data` values to `int`, `float` or `list` as specified in the table below.

`data`	`Droid`	Notes
average_lifespan (`str`)	average_lifespan (`int`)	
average_height(`str`)	average_height_cm (`float`)	

9.14.2 Create the species human

After implementing `create_species` return to `main`. Call the function `get_resource` and retrieve a SWAPI representation of the human species. Assign the return value to `swapi_human_species`.

💡 The `sw_utils` module includes a SWAPI "species" URL constant that you can pass as the `url` argument. If you need help constructing the `params` argument review the lecture notes and code.

Call the function `create_species()` and pass `swapi_human_species` as the argument. Assign the return value to a variable named `human_species`.

9.14.3 Write to file

Check your work. 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 and character-for-character.

9.15 Challenge 15

Task: Implement the function `create_person`.

9.15.1 `create_person` function

Replace `pass` with a code block that returns a new person 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

Certain `data` values require special handling and are subject to the following type conversion rules:

1. Convert all `data` values to `None` that match any of the `utl.NONE_VALUES` items (case-insensitive match of strings stripped of leading/trailing spaces). This can be accomplished by judicious use of the `utl.convert_to_*` functions.

2. Convert other `data` values to `int`, `float`, `dict`, or `list` as specified in the table below.

`data`	`Person`	Notes
height (`str`)	height_cm (`float`)	
mass (`str`)	mass_kg (`float`)	
homeworld (`str`)	homeworld (`dict`)	Retrieve from the cache or from SWAPI if the data is not
available locally; update values with the passed in `planets` data if not `None`.		
species (`str`)	species (`dict`)	Retrieve from the cache or from SWAPI if the data is not
available locally.		

3. ❗ The person's "homeworld" value *must* be converted to a dictionary representation of the home planet. Implement the following steps in your code to produce the required "homeworld" key-value pairs:

    1. Retrieve the planet dictionary from the cache or from SWAPI if the data is not available locally.

    2. If an optional Wookieepedia-sourced `planets` list is provided call `get_wookieepedia_data` and attempt to retrieve additional data that can be used to update the SWAPI homeworld dictionary.

    3. Call `create_planet` and pass it the (updated) homeworld dictionary.

    4. Assign the return value to the person's `homeworld` key.

4. ❗ The person's "species" value *must* also be converted to a dictionary representation of the species. Retrieve the species dictionary from the cache or from SWAPI if the data is not available locally. Once retrieved call `create_species` and pass the SWAPI species dictionary to it and assign the return value to the person's "species" key.

9.15.2 Create Anakin Skywalker

After implementing `create_person` return to `main`. Call the `utl.read_json` function and retrieve the supplementary Wookieepedia person data in the file `wookieepedia_people.json`. Assign the return value to `wookiee_people`.

Call the function `get_resource` and retrieve a SWAPI representation of the Jedi knight Anakin Skywalker. Access the "Anakin" dictionary which is stored in the response object and assign the value to `swapi_anakin`.

💡 The `sw_utils` module includes a SWAPI "people" URL constant that you can pass as the `url` argument. If you need help constructing the `params` argument review the lecture notes and code.

Call `get_wookieepedia_data` passing it the appropriate arguments and retrieve the "Anakin Skywalker" dictionary in `wookiee_people`. Assign the return value to the `wookiee_anakin`. Check the truth value of `wookiee_anakin`. If "truthy" update `swapi_anakin` with `wookiee_anakin`.

Call the function `create_person()` and pass the updated `swapi_anakin` *and* `wookiee_planets` as the arguments. Assign the return value to a variable named `anakin`.

9.15.3 Write to file

Check your work. 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 and character-for-character.

9.16 Challenge 16

Task: Implement the function `create_starship`.

9.16.1 `create_starship` function

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

Certain `data` values require special handling and are subject to the following type conversion rules:

1. Convert all `data` values to `None` that match any of the `utl.NONE_VALUES` items (case-insensitive match of strings stripped of leading/trailing spaces). This can be accomplished by judicious use of the `utl.convert_to_*` functions.

2. Convert other `data` values to `int`, `float` or `list` as specified in the table below.

`data`	`Starship`	Notes
length (`str`)	length_m (`float`)	
max_atmosphering_speed (`str`)	max_atmosphering_speed (`int`)	
hyperdrive_rating (`str`)	hyperdrive_rating (`float`)	
MGLT (`str`)	top_speed_mglt (`int`)	A megalight, the standard unit of distance in space.
armament (`str`)	armament (`list`)	check delimiter in `wookieepedia_starships.csv`
cargo_capacity (`str`)	cargo_capacity_kg (`int`)	

9.16.2 Create the light freighter *Twilight*

After implementing `create_starship` return to `main`. Call the `utl.read_csv_to_dicts` function and retrieve the supplementary Wookieepedia starship data in the file `wookieepedia_starships.csv`. Assign the return value to `wookiee_starships`.

Call `get_wookieepedia_data` passing the appropriate arguments and retrieve the light freigter named *Twilight* in `wookiee_starships`. Assign the return value to a variable named `wookiee_twilight`.

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

Call the function `create_starship()` and pass `wookiee_twilight` to it as the argument. Assign the return value to a variable named `twilight`.

9.16.3 Write to file

Check your work. 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 and

character-for-character.

9.17 Challenge 17

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.

9.17.1 `board_passengers` function

Replace `pass` with a code block that assigns passengers to a starship. 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 is limited by the starship's "max_passengers" value. If the number of passengers attempting to board exceeds the starship's "max_passengers" value only the first `n` passengers (where `n` = "max_passengers") are permitted to board the vessel. This limitation *must* be imposed by the `board_passengers` function and will be subject to auto grader testing.

   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 to board the vessel.
3. After boarding the passengers return the starship to the caller.

9.17.2 Get passengers aboard the *Twilight*

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

After implementing `board_passengers` return to `main`. Create a dictionary representation of the Galactic senator Padmé Amidala. Utilize the same "creational" workflow employed to create the dictionary representation of Anakin Skywalker. Consider using the following variable names to represent Padmé.

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

Create a dictionary representation of the protocol droid named C-3PO. 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 data)
- `wookiee_c_3po` (assigned to the Wookieepedia dictionary data)
- `c_3po` (assigned to the `create_droid` return value)

Call the function `board_passengers` and pass the following arguments to it:

- `twilight`
- a list of passengers comprising `padme`, `c_3po`, and `r2_d2` (in that order).

Assign the return value to the variable `twilight`.

💡 Test your function by passing additional passengers to it in excess of the permitted "max_passengers" value. Consider creating dictionary representations of the Jedi masters Mace Windo, Plo Koon, Shaak Ti, and Yoda and attempt to add them as extra passengers. You can retrieve both SWAPI and Wookieepedia dictionary representations of each to use for testing.

9.18 Challenge 18

> 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.

9.18.1 `assign_crew_members` function

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 "conventional" `for` loop that adds the "crew_member" key-value pairs to an accumulation dictionary named `crew_members`. Get it working and then convert it to a dictionary comprehension.

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

3. The number of crew members that can be assigned to the 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 by the `assign_crew_members` function and will be subject to auto grader testing.

   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 and personnel are permitted to be added as key-value pairs to the crew members dictionary.

4. Both the passed in `crew_positions` and `personnel` lists should 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] >, ...}
   ```

5. Map (i.e., assign) the new dictionary to a new starship key named "crew_members" before returning the crewed starship to the caller.

> 💡 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.

9.18.2 Assign crew members to the *Twilight*

After implementing `assign_crew_members` return to `main`. Create a dictionary representation of the the Jedi General Obi-Wan Kenobi. Utilize the same "creational" workflow employed to create the other people. Consider using the following variable names to represent Obi-Wan.

- `swapi_obi_wan` (assigned to the SWAPI dictionary data)
- `wookiee_obi_wan` (assigned to the Wookieepedia dictionary data)
- `obi_wan` (assigned to the `create_person` return value)

Next, call the function `assign_crew_members` and pass the following arguments to it:

- `twilight`
- a crew positions list comprising the following string elements: "pilot" and "copilot"
- a personnel list comprising `anakin` and `obi_wan`

Assign the return value to the variable `twilight`.

💡 Test your function by passing additional crew positions and personnel to it in excess of the permitted "crew_size" value. Consider creating dictionary representations of the Jedi masters Mace Windo, Plo Koon, Shaak Ti, or Yoda and attempt to assign one or more as extra crew members. You can retrieve both SWAPI and Wookieepedia dictionary representations of each to use for testing.

9.18.3 Issue instructions to R2-D2

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

10.0 Finis

> R2 release the docking clamp. *Anakin Skywalker*

10.0.1 Issue instructions to R2-D2

Add Anakin's order "Release the docking clamp" to `r2_d2's` "instructions" key-value pair.

10.0.2 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.

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 and character-for-character.

Your job is done. Never mind that Separatist starfighters are in hot pursuit of the *Twilight*; declare victory.

Congratulations on completing SI 506.