SI 506: Last assignment

1.0 Dates

- Available: Tuesday, 7 December 2021, 4:00 PM Eastern
- Due: Monday, 20 December 2021, on or before 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.

We recommend that at a minimum you bookmark the following w3schools Python pages and/or have them open in a set of browser tabs as you work on the last assignment:

- Python keywords
- Python operators
- Python built-in functions
- Python `dict` methods
- Python `list` methods
- Python `str` methods
- SWAPI documentation

3.0 Points

The last assignment is worth 1800 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 taking the exam. The last assignment code that you submit *must* be your own work. Likewise, you are prohibited from assisting any other student required to take this exam. This includes those taking the exam during the regular exam period, as well as those who may take the exam at another time and/or place due to scheduling conflicts or other issues.

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.

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.

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, and Wikipedia.

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

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

7.0 The built-in `print()` function is your friend

💡 as you work through the challenges make frequent use of the built-in `print()` function to check your variable assignments. Recall that you can call `print()` from inside a function, loop, or conditional statement. Use f-strings and the newline escape character `\n` to better identify the output that `print()` sends to the terminal as illustrated in the following example.

```
print(f"\nSOME_VAL = {some_val}")
```

8.0 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 Separtist 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.

Challenges 02-07 utilize a second *Clone Wars* data set that provides summary data about the first forty-four (44) episodes that comprise seasons one and two. 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 08-14 recreate the escape of the light freighter *Twilight* from the sabotaged and doomed Separtist 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 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 Senator Padmé Amidala and the protocol 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 classes, methods, and functions before instantiating objects and generating a JSON document that recreates the escape from the *Malevolence*.

May the Force be with You.

8.1 Challenge 01

Task: Utilize a small *Clone Wars* data set to demonstrate your indexing and slicing skills. Retrieve subsets of the data and individual values using indexing and slicing.

8.1.1 Get data

In `main` call the `read_csv` function and retrieve data contained in the file `clone_wars.csv`. The file provides general information about *The Clone Wars* animated series, seasons 1-7. Assign the return value to a variable named `clone_wars`.

8.1.2 indexing and slicing

❗ Using a `for` loop is neither required nor permitted. 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`.

8.2 Challenge 02

Task: Explore the the first two seasons of *Clone Wars* episodes. Implement a function that checks whether or not an episode possesses viewership information.

8.2.1 Get data

In `main` call the `read_csv_to_dicts` function and retrieve data contained in the file `clone_wars_episodes.csv`. The file provides information about the first forty-four (44) episodes of *The Clone Wars* (seasons 1-2). Assign the return value to a variable named `clone_wars_episodes`.

💡 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 `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.

8.2.2 `has_viewer_data` function

Implement the function named `has_viewer_data`. The function computes the truth value of the episode's "episode_us_viewers_mm" key-value pair, returning either `True` or `False`. Read the function's Docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

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" value.

💡 Do this by implementing a `for` loop and a conditional statement inside the loop block that tests the return value of the function `has_viewer_data` (returns either `True` or `False`). If `True` increment the count by 1.

❗ The number of episodes that possess a "episode_us_viewers_mm" viewership value equals twenty-four (`24`). 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.

8.3 Challenge 03

Task: Implement the functions `convert_to_int`, `convert_to_float`, and `convert_to_list`. All three functions attempt to convert a passed in value to a more appropriate type.

❗ Each function in this challenge *must* employ `try` and `except` statements in order to handle runtime exceptions whenever an invalid type conversion is attempted. 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. Simply return the passed in value unchanged.

8.3.1 `convert_to_int` function

Implement the function named `convert_to_int` located in the module `swapi_utils.py`. The function will attempt to convert a passed in value to an `int`. Read the function's Docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

8.3.2 `convert_to_float` function

Implement the function named `convert_to_float` located in the module `swapi_utils.py`. The function will attempt to convert a passed in value to a `float`. Read the function's Docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

8.3.3 `convert_to_list` function

Implement the function named `convert_to_list` located in the module `swapi_utils.py`. The function will attempt to convert a passed in value to a `list` using a provided `delimiter`. Read 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 an exception is encountered.

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 `str` method used to convert a string to a list; otherwise rely on the `str` method's default behavior (i.e., don't pass it a delimiter argument).

❗ Don't assume that the strings are "clean"; remove any leading/trailing spaces in the passed in string before attempting to convert the value to a list.

8.3.4 Test the `convert_to_*` functions

After implementing the three 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_int converted = {utl.convert_to_int('506')}")
print(f"\nconvert_to_int no change = {utl.convert_to_int('unknown')}")
print(f"\nconvert_to_int no change = {utl.convert_to_int([506, 507])}")
```

8.4 Challenge 04

Task: Implement a function that converts specified string values to more appropriate types.

8.4.1 `convert_episode_values` function

Implement the function named `convert_episode_values`. Read 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 accepts a list of nested "episode" dictionaries. In the function block implement a nested loop with the outer loop iterating over the passed in `espisodes` and the inner loop iterating over each episode's key-value pairs. Implement the necessary conditional logic to convert the passed in dictionary values to the specified types, delegating to the `convert_*` functions located in the module `swapi_utils` the task of converting strings to either `int`, `float`, or `list` per the conversion chart below.

Conversion	value(s)	Delegate to
`str` to `None`	All blank or empty values	handled locally

Conversion	value(s)	Delegate to
`str` to `int`	'series_season_num', 'series_episode_num', 'season_episode_num'	swapi_utils.convert_to_int()
`str` to `float`	'episode_prod_code', 'episode_us_viewers_mm'	swapi_utils.convert_to_float()
`str` to `list`	'episode_writers'	swapi_utils.convert_to_list()

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

8.4.2 Call function; write to file

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`.

Then call the function `write_json` and pass the filepath `stu-clone_wars-episodes_converted.json` and the converted list `clone_wars_episodes` to it as arguments. 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.

8.5 Challenge 05

Task: Implement functions to retrieve the most viewed / least viewed episodes of the first two seasons of *The Clone Wars*.

8.5.1 `get_most_viewed_episode` function

Implement the function named `get_most_viewed_episode`. The function returns the episode from the `episodes` list with the highest recorded viewership (ignores ties). Read the function's Docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

💡 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 compare the current "episode_us_viewers_mm" value to the previous value.

8.5.2 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`.

💡 The episode with the highest recorded viewership is listed below. If this dictionary is assigned to `most_viewed_episode` (confirm by passing it to `print()`) then proceed to the next step; otherwise recheck your work.

```
{
    '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
}
```

8.5.3 `get_least_viewed_episode` function

Implement the function named `get_least_viewed_episode`. The function returns the episode from the `episodes` list with the lowest recorded viewership (ignores ties). Read the function's Docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

💡 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 compare the current "episode_us_viewers_mm" value to the previous value.

8.5.4 Call function

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

💡 The episode with the lowest recorded viewership is listed below. If this dictionary is assigned to `least_viewed_episode` (confirm by passing it to `print()`) then proceed to the next challenge; otherwise recheck your work.

```
{
    'series_title': 'Star Wars: The Clone Wars',
    'series_season_num': 1,
    'series_episode_num': 9,
    'season_episode_num': 9,
    'episode_title': 'Cloak of Darkness',
    'episode_director': 'Dave Filoni',
    'episode_writers': ['Paul Dini'],
    'episode_release_date': 'December 5, 2008',
    'episode_prod_code': 1.1,
    'episode_us_viewers_mm': 1.95
}
```

8.6 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*.

8.6.1 `count_episodes_by_director` function

Implement the function named `count_episodes_by_director`. The function builds and 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 directed. Read the function's Docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

❗ 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 >,
   ...
}
```

8.6.2 Call function; write to file

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`.

Then call the function `write_json` and pass the filepath `stu-clone_wars-director_episode_counts.json` and `director_episode_counts` to it as arguments.

If your output matches the JSON object in `fxt-clone_wars-director_episode_counts.json` proceed to the next challenge. Otherwise, recheck your work.

8.7 Challenge 07

Task: Construct a dictionary that groups the first forty-four (44) episodes of *The Clone Wars* by each contributing writer.

8.7.1 `group_episodes_by_writer` function

Implement the function named `group_episodes_by_writer`. The function builds and returns a dictionary of key-value pairs that associates each writer to a list of episodes in which they are credited with contributing to the screenplay. Read the function's Docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

❗ The writer's name comprises the key and the associated value a list of nested episode dictionaries in which they are credited as a writer. Implement conditional logic to ensure that each writer is assigned a key and the associated list value comprising one or more episode dictionaries is constructed properly.

```
{
   < writer_name_01 >: [{< episode_01 >}, {< episode_03 >}, ...],
   < writer_name_02 >: [{< episode_02 >}, {< episode_03 >}, ...],
   ...
}
```

💡 Consider implementing a nested loop in the function block in order to access all the writers associated with an individual episode.

8.7.2 Call function; write to file

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

Then call the function `write_json` and pass the filepath `stu-clone_wars-writer_episodes.json` and `writer_episodes` to it as arguments. Compare your file to the test fixture file `fxt-clone_wars-writer_episodes.json`. Both files *must* match, line for line, and character for character.

8.8 Challenge 08

Task: Implement the utility functions named `convert_to_none` and `convert_gravity_value`.

8.8.1 `convert_to_none` function

Implement the function named `convert_to_none` located in the module `swapi_utils.py`. The function will attempt to convert a passed in string value to `None` if the value equals any of the following strings:

- "n/a" or "N/A'
- "none" or "None"
- "unknown" or "Unknown"

The function *must* return the value unchanged if the passed in value does not match any of the strings above *or* an exception is encountered.

Read 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 *must* employ `try` and `except` statements in order to handle runtime exceptions whenever an invalid type conversion is attempted. If an exception is encountered the `except` block will "catch" the exception and the value will be returned to the caller unchanged.

8.8.2 Call function

After implementing `convert_to_none` return to `main`. Test your implementation by calling the function several times from inside `print()` and passing to it test values such as "Unknown", "Yoda", and the list `[1, 2, 3]`.

```
print(f"\nChallenge 08: convert_to_none('Unknown') =
{utl.convert_to_none('Unknown')}")
# returns None (converted)
print(f"\nChallenge 08: convert_to_none('Yoda') =
{utl.convert_to_none('Yoda')}")
# returns 'Yoda' (no change)
print(f"\nChallenge 08: convert_to_none([1, 2, 3]) =
{utl.convert_to_none([1, 2, 3])}")
# returns [1, 2, 3] (no change, exception caught)
```

8.8.3 `convert_gravity_value` function

Implement the function named `convert_gravity_value` located in the module `swapi_utils.py`. The function will attempt to convert a planet's "gravity" value to a float by first removing the "standard" unit of measure if it exists in the string. It then delegates to the function `convert_to_float` the task of casting the value to a `float`.

The function *must* return the value unchanged if the passed in value does not match any of the strings above *or* an exception is encountered.

Read 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 *must* employ `try` and `except` statements in order to handle runtime exceptions whenever an invalid type conversion is attempted. If an exception is encountered the `except` block will "catch" the exception and the value will be returned to the caller unchanged.

8.8.4 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", "1.56", and the list `[1, 2, 3]`.

```
print(f"\nChallenge 08: convert_gravity_value('1 standard') =
{utl.convert_gravity_value('1 standard')}")
# returns 1.0 (converted)
print(f"\nChallenge 08: convert_gravity_value('1.56') =
{utl.convert_gravity_value('1.56')}")
# returns 1.56 (converted)
print(f"\nChallenge 08: convert_gravity_value([1, 2, 3]) =
{utl.convert_gravity_value([1, 2, 3])}")
# returns [1, 2, 3] (no change; exception caught)
```

8.9 Challenge 09

Task: Implement the `Planet` class and the function `create_planets()`. Review the associated Docstrings for implementation details.

8.9.1 `Planet` class

`__init__`. Replace `pass` with the required and optional instance variable assignment statements. Assign `None` to all optional instance variables.

💡 The values required to instantiate a class instance are specified in the `__init__` method's parameter list.

`jsonable`. Replace `pass` and return a JSON-friendly dictionary comprising all instance variables and their values expressed as key-value pairs in the order specified by the Docstring. This is best accomplished by

returning a dictionary literal.

8.9.2 `create_planet` function

Replace `pass` with a code block that instantiates an instance of `Planet`, assigns values to optional instance variables sourced from the passed in `data` dictionary, and returns the new `Planet` instance to the caller. Read the function's Docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Optional instance variables require special handling and are subject to the following type conversion rules:

1. Convert all `data` string values to `None` that equal "n/a" or "N/A", "none" or "None" or "unknown" or "Unknown".
2. Convert other `data` values to `int`, `float` or `list` as specified in the table below.

`data`	`Droid`	**Notes**
region (`str`)	region (`str`)	No change
sector (`str`)	sector (`str`)	No change
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`)	

💡 Leverage the `convert_*` functions in the `swapi-utils` module for effecting the type conversions.

8.9.3 Create planet Tatooine instance

After implementing `create_planet` return to `main`. Call the function `read_csv_to_dicts` and retrieve the supplementary Wookieepedia planet data in the file `wookieepedia_planets.csv`. Assign the return value to `wookiee_planets`.

Call the function `get_swapi_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 `tatooine_data`.

💡 The `swapi_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.

Next, access the "Tatooine" dictionary in `wookiee_planets` and update the `tatooine_data` dictionary with the Wookieepedia Tatooine key-value pairs.

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

8.9.4 Write to file

Check your work. Call the function `write_json` and pass the filepath `stu-tatooine.json` and a JSON-friendly representation of `tatooine` to it as the arguments. Compare your file to the test fixture file `fxt-tatooine.json`. Both files *must* match line for line and character for character.

8.10 Challenge 10

Task: Implement the `Droid` class and the function `create_droids()`. Review the associated Docstrings for implementation details.

💡 Challenge 09's workflow illustrates the general creational pattern applied to each planet, person, droid, and starship encountered in Challenges 10-14.

- retrieve SWAPI data
- combine with Wookieepedia data
- create class instance
- assign (converted) values to object's optional instance variables
- 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.

8.10.1 `Droid` class

`__init__`. Replace `pass` with the required and optional instance variable assignment statements. Assign `None` to all optional instance variables.

💡 The values required to instantiate a class instance are specified in the `__init__` method's parameter list.

`jsonable`. Replace `pass` and return a JSON-friendly dictionary comprising all instance variables and their values expressed as key-value pairs in the order specified by the Docstring. This is best accomplished by returning a dictionary literal.

8.10.2 `create_droid` function

Replace `pass` with a code block that instantiates an instance of `Droid`, assigns values to optional instance variables sourced from the passed in `data` dictionary, and returns the new `Droid` instance to the caller. Read the function's Docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Optional instance variables require special handling and are subject to the following type conversion rules:

1. Convert all `data` string values to `None` that equal "n/a" or "N/A", "none" or "None" or "unknown" or "Unknown".
2. Convert other `data` values to `int`, `float` or `list` as specified in the table below.

`data`	`Droid`	**Notes**
manufacturer (`str`)	manufacturer (`str`)	No change
create-year (`str`)	create-year (`str`)	No change
height (`str`)	height_m (`float`)	
mass (`str`)	mass_kg (`float`)	
equipment (`str`)	equipment (`list`)	Check delimiter in `wookieepedia_droids.json`

💡 Leverage the `convert_*` functions in the `swapi-utils` module for effecting the type conversions.

8.10.3 Create R2-D2 `Droid` instance

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

Call the function `get_swapi_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 `r2_d2_data`.

💡 The `swapi_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.

Access the "R2-D2" dictionary in `wookiee_droids` and update the `r2_d2_data` dictionary with the Wookieepedia "R2-D2" key-value pairs.

Call the function `create_droid()` and pass the updated `r2_d2_data` to it as the argument. Assign the return value (a `Droid` instance) to a variable named `r2_d2`.

8.10.4 Write to file

Check your work. Call the function `write_json` and pass the filepath `stu-r2_d2.json` and a JSON-friendly representation of `r2_d2` to it as the arguments. Compare your file to the test fixture file `fxt-r2_d2.json`. Both files *must* match line for line and character for character.

8.11 Challenge 11

Task: Implement the `Person` class and the function `create_person()`. Review the associated Docstrings for implementation details.

8.11.1 `Person` class

`__init__`. Replace `pass` with the required and optional instance variable assignment statements. Assign `None` to all optional instance variables.

💡 The values required to instantiate a class instance are specified in the `__init__` method's parameter list.

`jsonable`. Replace `pass` and return a JSON-friendly dictionary comprising all instance variables and their values expressed as key-value pairs in the order specified by the Docstring. This is best accomplished by returning a dictionary literal.

❗ When implementing `Person.jsonable` recall that a `Person` instance's `self.homeworld` valure requires special handling in order to ensure that the method returns a JSON-friendly representation of both the person and their home planet.

8.11.2 `create_person` function

Replace `pass` with a code block that instantiates an instance of `Person`, assigns values to optional instance variables sourced from the passed in `data` dictionary, and returns the new `Person` instance to the caller. Read the function's Docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Optional instance variables require special handling and are subject to the following type conversion rules:

1. Convert all `data` string values to `None` that equal "n/a" or "N/A", "none" or "None" or "unknown" or "Unknown".
2. Convert other `data` values to `int`, `float` or `list` as specified in the table below.

`data`	`Person`	**Notes**
height (`str`)	height_m (`float`)	
mass (`str`)	mass_kg (`float`)	
homeworld (`str`)	homeworld (`Planet`)	Enriched with passed in Wookieepedia data
force_sensitive ('bool')	force_sensitive ('bool')	No change

   💡 Leverage the `convert_*` functions in the `swapi-utils` module for effecting the type conversions.

3. ❗ Converting the `Planet` instance's `homeworld` attribute to an instance of `Planet` requires implementing the same "creational" workflow applied to other entities:
   - retrieve SWAPI planet data
   - combine with Wookieepedia data (sourced from passed in `planets` if argument is not `None`)

- call `create_planet` and return class instance with converted instance variable values
- assign new `Planet` instance to person object's `homeworld` instance variable

8.11.3 Create Anakin Skywalker `Person` instance

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

Call the function `get_swapi_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 `anakin_data`.

💡 The `swapi_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.

Access the "Anakin" dictionary in `wookiee_people` and update the `anakin_data` dictionary with the Wookieepedia "Anakin" key-value pairs.

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

8.11.4 Write to file

Check your work. Call the function `write_json` and pass the filepath `stu-anakin_skywalker.json` and a JSON-friendly representation of `anakin` to it as the arguments. Compare your file to the test fixture file `fxt-anakin_skywalker.json`. Both files *must* match line for line and character for character.

8.12 Challenge 12

Task: Implement the `Starship` class and the function `create_starship()`. Review the associated Docstrings for implementation details.

8.12.1 `Starship` class

`__init__`. Replace `pass` with the required and optional instance variable assignment statements. Assign `None` to all optional instance variables.

💡 The values required to instantiate a class instance are specified in the `__init__` method's parameter list. Note that neither `self.crew_members` nor `self.passengers_on_board` need to be included in `__init__`. You will implement other methods tasked with adding crew members and passengers.

`jsonable`. Replace `pass` and return a JSON-friendly dictionary comprising all instance variables and their values expressed as key-value pairs in the order specified by the Docstring. This is best accomplished by returning a dictionary literal.

❗ A `Starship` instance can be assigned `Crew` and `Passengers` instances by calling the `Starship.assign_crew_members` and `Starship.add_passengers` methods. As we've discussed previously such objects require special handling when building and returning a JSON-friendly representation of the `Starship`.

Be sure to include conditional logic in the `Starship.jsonable` method that checks if the instance has the following attributes:

- `self.crew_members`
- `self.passengers_on_board`

If `True`, also check if the instance variable has been assigned a "truthy" value (e.g., not `None`). If both conditions resolve to `True` then it should be safe to call the value's `jsonable` method in order to return a dictionary representation of the object and assign it to a local variable.

8.12.2 `create_starship` function

Replace `pass` with a code block that instantiates an instance of `Starship`, assigns values to optional instance variables sourced from the passed in `data` dictionary, and returns the new `Starship` instance to the caller. Read the function's Docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Optional instance variables require special handling and are subject to the following type conversion rules:

1. Convert all `data` string values to `None` that equal "n/a" or "N/A", "none" or "None" or "unknown" or "Unknown".
2. Convert other `data` values to `int`, `float` or `list` as specified in the table below.

`data`	`Starship`	**Notes**
manufacturer (`str`)	manufacturer (`str`)	No change
length (`str`)	length_m (`float`)	
max_atmosphering_speed (`str`)	max_atmosphering_speed (`int`)	
hyperdrive_rating (`str`)	hyperdrive_rating (`float`)	
MGLT (`str`)	MGLT (`int`)	
armament (`str`)	armament (`list`)	check delimiter in `wookieepedia_starships.csv`
cargo_capacity (`str`)	cargo_capacity_kg (`int`)	
consumables (`str`)	consumables (`str`)	No change

 Leverage the `convert_*` functions in the `swapi-utils` module for effecting the type conversions.

8.12.3 Create the light freighter Twilight `Starship` instance

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

Access the light freigter named *Twilight* in `wookiee_starships`. Assign the return value to a variable named `twilight_data`.

💡 There is no SWAPI representation of the light freighter *Twilight*.

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

8.12.4 Write to file

Check your work. Call the function `write_json` and pass the filepath `stu-twilight.json` and a JSON-friendly representation of `twilight` to it as the arguments. Compare your file to the test fixture file `fxt-twilight.json`. Both files *must* match line for line and character for character.

8.13 Challenge 13

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

Task: Implement the `Starship.assign_crew_members` method. Assign Anakin Skywalker and Obi-Wan Kenobi as the crew of the *Twilight*.

8.13.1 `Starship.assign_crew_members` method

Implement the `Starship.assign_crew_members` method. Read the function's Docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Add conditional logic to the method in order to ensure that the following restrictions apply:

1. The passed in `crew` argument *must* be an instance of the `Crew` class.

If the condition is statisfied assign the passed in value to `self.crew_members`. If the condition is violated *do not* perform the variable assignment.

8.13.2 Assign a `Crew` instance to the *Twilight*

After implementing `Starship.assign_crew_members` return to `main`. Create a `Person` instance that represents the Jedi General Obi-Wan Kenobi. Utilize the same "creational" workflow employed to create the Anakin Skywalker `Person` instance. Consider using the following variable names to represent Obi-Wan.

- `obi_wan_data` (SWAPI and Wookieepedia dictionary data)
- `obi_wan` (`Person` instance)

Next, create a `Crew` instance assigning `anakin` as "pilot" and `obi_wan` as "copilot". Assign the instance to a variable named `crew`.

❗ The `Crew` class is fully implemented. Your task is to instantiate a new instance and provide the `__init__` method with the required "crew members" object. Review the `Crew` class's Docstrings as well as the lecture notes and solution files for more information and hints.

Call the appropriate `Starship` method for adding crew members and assign the Anakin and Obi-Wan `Crew` instance to the *Twilight*.

8.14 Challenge 14

Task: Implement the `Passengers` class. Assign Senator Padmé Amidala, the protocol droid C-3PO, and the astromech droid R2-D2 as passengers aboard the *Twilight*.

8.14.1 `Passengers` class

Implement the `Passengers` class by modeling its methods on the `Crew` class. The class implementations are nearly identical. The key difference involves the type of object used to instantiate an instance of each class.

`__init__`. accepts a **list** of `Person` and/or `Droid` instances (the `Crew` class accepts a dictionary). Note the difference and adjust the handling of the incoming object accordingly.

Unlike the `crew_members` dictionary in which each key is mapped to an instance variable, the `passengers` list of `Person` and/or `Droid` instances requires employing dot notation to access each `Person` or `Droid` instance's "name" value for use as a key. The two class's `jsonable` methods also vary slightly. Read the relevant Docstrings for hints.

```
crew = Crew({'some_role': < Person | Droid >, ...})

passengers = Passengers([< Person | Droid >, < Person | Droid >, ...])
```

❗ When creating new `Passengers` instance variable names you must reformat the person and droid name contained in the passed in list of passengers before passing the value to the built-in function `setattr()`. Three rules apply:

1. Change name to lowercase
2. Replace space (' ') with underscore ('_')
3. Replace dash ('-') with underscore ('_')

`jsonable`. Replace `pass` and return a JSON-friendly dictionary comprising all instance variables and their values expressed as key-value pairs. Mimic the `Crew` class's `jsonable` method implementation but note that the method *must* return a JSON-friendly **list** of `Person` and/or `Droid` instances. Call the appropriate `dict` method when looping over the `__dict__` attribute to access each `Person` and/or `Droid` instance.

8.14.2 `Starship.add_passengers` method

Implement the `Starship.add_passengers` method. Read the function's Docstring to better understand the task it is to perform, the parameters it defines, and the return value it computes.

Add conditional logic to the method in order to ensure that the following restriction applies:

1. The passed in `passengers` argument *must* be an instance of the `Passengers` class.

If the condition is statisfied assign the passed in value to `self.passengers_on_board`. If the condition is violated *do not* perform the variable assignment.

8.14.3 Assign a `Passengers` instance to the *Twilight*

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

After implementing `Starship.assign_crew_members` return to `main`. Create a `Person` instance that represents the Galactic senator Padmé Amidala. Utilize the same "creational" workflow employed to create the other `Person` instances. Consider using the following variable names to represent Padmé.

- `padme_data` (SWAPI and Wookieepedia dictionary data)
- `padme` (`Person` instance)

💡 The accented `é` (e-acute) character in Padmé's name is replaced with `e` when the Wookieepedia data is applied.

Create a `Droid` instance that represents the protocol droid named C-3PO. Utilize the same "creational" workflow employed to create the R2-D2 `Droid` instances. Consider using the following variable names to represent C-3PO.

- `c_3po_data` (SWAPI and Wookieepedia dictionary data)
- `c_3po` (`Droid` instance)

Next, create a `Passengers` instance assigning `padme`, `c_3po`, and `r2_d2` (in that order) as passengers. Assign the instance to a variable named `passengers`.

💡 Review the `Passengers` class `__init__` method to ensure that you pass the correct object to the constructor.

Call `twilight.add_passengers` and assign the Padmé, C-3PO, and R2-D2 `Passengers` instance to the *Twilight*.

8.14.4 Write to file

Call the function `write_json` and pass the filepath `stu-twilight_departs.json` and a JSON-friendly representation of `twilight` to it as the arguments. Compare your file to the test fixture file `fxt-twilight_departs.json`. Both files *must* match line for line and character for character.

9.0 Finis

> R2 release the docking clamp. *Anakin Skywalker*

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.

Your job is done. Never mind that Separtist starfighters are in hot pursuit of the *Twilight*. Declare victory anyways.

Congratulations on completing SI 506.

10.0 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.

11.0 Auto grader / manual scoring

The autograder runs a number of tests against the Python file you submit, which the autograder imports as a module so that it can gain access to and inspect the functions and other objects defined in your code. The functional tests are of two types:

1. The first type will call a function passing in known argument values and then perform an equality comparison between the return value and the expected return value. If the function's return value does not equal the expected return value the test will fail.

2. The second type of test involves checking variable assignments in `main()` or expressions in other functions. This type of test evaluates the code you write, character for character, against an expected line of code using a regular expression to account for permitted variations in the statements that you write. The test searches `main()` for the expected line of code. If the code is not located the test will fail.

If the auto grader is unable to grade your submission successfully with a score of 1800 points 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.