SI 506 Lecture 25

Topics

1. Type checking (with `isinstance()`)
2. Challenges

Vocabulary

- **cache**: Storage location that holds data in order in increase the speed by which previously requested data can be retrieved again if required. A cache is usually designed to hold data temporarily; cached data is allowed to "expire" after a given interval and is removed from storage. Cache expiration policies help to reduce the chance that data held in the cache no longer matches the origin data.

- **Module**: a Python file that contains definitions and statements that are intended to be *imported* into a Python script (a.k.a a program), an interactive console session, or another module.

1.0 Type checking (with `isinstance()`)

There is a function in this week's problem set in which it is necessary to confirm whether or not the passed in dictionary values are strings. There are two built-in functions that can confirm a value's type. You are already familiar with the built-in `type()` function. You can check if a value is of type `str` (i.e., a string object) in an `if` statement:

```
>>> if type(value) is str:
...     print('Value is a string')
... else:
...    print('Value is not a string')
...
Value is a string
```

However, you can also confirm a value's type using the built-in function `isinstance()`:

```
>>> if isinstance(value, str):
...     print('Value is a string')
... else:
...     print('Value is not a string')
...
Value is a string
```

bulb: `isinstance()` can also check whether or not a value is a subtype of passed in supertype. In other words, the function is aware of the class hierarchy in which the value resides. For example, and `OrderedDict` (subtype) is a type of `dict` (supertype) as `isinstance()` confirms:

```
>>> from collections import OrderedDict
>>> value = OrderedDict({'a': 1, 'b': 2, 'c': 3})
>>> if isinstance(value, dict):
...     'Value is a dictionary'
... else:
...     'Value is not a dictionary'
...
'Value is a dictionary'
```

Challenge 01

Task: Implement a function that creates a "thinned" representation of a SWAPI starship.

1. Implement the function `create_starship`. Review the function's docstring to better understand its expected behavior but implement only what is specified below.

   Review the starship objects in the file `episode_iv_starships.json`. Then utilize a **dictionary literal** to return a dictionary comprising the following key-value pairs in the specified order. Access the values from the passed in `data` dictionary. No type conversions are required.

   - url
   - name
   - model
   - starship_class

2. After implementing the function, return to `main`. Call the `utl` module's `read_json` function and return a list of starship dictionaries stored in the file `episode_iv_starships.json`. Assign the list to a variable named `swapi_starships`.

3. Write a **list comprehension** that passes each starship in `swapi_starships` to the function `create_starship` in order to return a new representation of the starship. Assign the new list to a variable named `starships`.

4. Call the `utl` module's `write_json` function and write `starships` serialized as JSON to a file named `stu-starships.json`.

Challenge 02

Task: Implement a string to float conversion function that employs `try`/`except` blocks to catch and handle runtime exceptions if an illegal type conversion is attempted.

1. Implement the `utl` module's `convert_to_float` function. Review the function's docstring to better understand its expected behavior.

   ❗ Since any arbitrary value could be passed to the function employ `try` and `except` blocks to guard against illegal type conversions as is illustrated by the example below:

   ```
   >>> string = 'unknown'
   >>> num = float(string)
   ```

```
Traceback (most recent call last):
...
ValueError: could not convert string to float: 'unknown'
```

If an exception is triggered, "catch" it in the `except` block and return it to the caller unchanged.

2. Return to the function `create_starship`. Add two (2) additional key-value pairs sourced from `data` to the starship dictionary literal in positions five and six.

    - hyperdrive_rating
    - top_speed_mglt

    ❗ The TIE/LN starfighter does not possess a hyperdrive. The SWAPI representation of the TIE fighter excludes both the "hyper_drive" and "MGLT" key-value pairs. In other words, the data retrieved from `episode_iv_starships.json` is semi-structured (i.e., not all starships are structured the same in terms of their key-value pairs).

    When assigning values to the dictionary literal's "hyperdrive_rating" and "top_speed_mglt" key-value pairs utilize a dictionary method designed to guard against triggering a runtime `KeyError` if a key is not found in a dictionary. If you forget which method to call check the w3school's dictionary methods page.

3. Convert the string values of each by passing the value's to the `utl` module's `convert_to_float` function.

    ❗ Map (i.e., assign) the "MGLT" value to the new dictionary's "top_speed_mglt" key.

    💡 A long time ago in a galaxy far, Megalight (MGLT) was the standard unit of distance in space.

4. Run `lecture_25.py` and update the file `stu-starships.json`. Confirm that each starship's "hyper_drive" and "top_speed_mglt" values have been converted to a number with a fractional component (i.e., decimal value).

Challenge 03

Task: Refactor the function `convert_to_int` so that it can convert numeric strings that include a thousands separator comma to an integer (`int`).

1. Refactor (i.e., restructure, revise) the `utl` module's `convert_to_int` function so that it can convert the following numeric string variations to an integer.

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

    Currently, the function can only handle numeric strings that do not include a comma (thousands separator).

2. After refactoring `convert_to_int` return to return to the function `create_starship`. Add another key-value pair sourced from `data` to the starship dictionary literal in position seven.

    - crew_size

   ❗ Map (i.e., assign) the "crew" value to the new dictionary's "crew_size" key.

3. Convert the "crew" string value by passing it to the `utl` module's `convert_to_int` function.

4. Run `lecture_25.py` and update the file `stu-starships.json`. Confirm that each starship's "crew_size" value has been converted to a number.

Challenge 04

Task: Combine SWAPI data with data sourced from Wookieepedia.

1. In the function `create_starships` add another key-value pair to the starship dictionary literal in the *last* position. No type conversion is required for this challenge.

    - armament

   💡 the armament value will be sourced from `wookieepedia_starships.csv`.

2. In `main` immediately below the `swapi_starships` variable assignment, call the `utl` module's `read_csv_to_dicts` function and return a list of starship dictionaries stored in the file `wookieepedia_starships.csv`. Assign the list to a variable named `wookiee_starships`.

3. Comment out the list comprehension used to create the `starships` list.

4. Create an empty accumulator list named `starships`. Implement a nested `for` loop that loops over `swapi_starships` (outer) and `wookiee_starships` (inner). Inside the inner loop check perform a **case insensitive** comparison of the SWAPI starship's "model" name and the Wookieepedia starship's "model" name. If the strings match *update* the SWAPI starship dictionary with the Wookieepedia starship dictionary. Then exit the inner loop. Otherwise, continue iterating over each Wookieepedia starship in pursuit of a match.

5. After the inner loop has terminated call the function `create_starship` and pass to it the SWAPI starship. Assign the return value to a variable and *append* the new starship dictionary to the `starships` list.

   💡 You can also pass `create_starship` and its argument to the `starships` append method.

6. Run `lecture_25.py` and update the file `stu-starships.json`. Confirm that starship's that were updated with Wookieepedia data include an "armament" key-value pair.

Challenge 05

Task: Implement a string to list conversion function that employs `try`/`except` blocks to catch and handle runtime exceptions if an illegal type conversion is attempted.

1. Implement the `utl` module's function named `convert_to_list`. Review the function's docstring to better understand its expected behavior.

❗ The function *must* be able to accommodate both the default delimiter value (a space) and other delimiter values intended to override the default string splitting behavior.

2. After implementing `convert_to_list` return to the function `create_starship`. Convert the "armament" string value by passing it to the `utl` module's `convert_to_list` function along with the appropriate delimiter value.

3. Run `lecture_25.py` and update the file `stu-starships.json`. Confirm that each starship's "aramament" value has been converted to a list.

```
"armament": [
   "2 x dorsal and ventral dual turbolaser turrets",
   "4 x Turbolaser cannons"
 ]
```

Challenge 06

Task: Add film credits to each starship.

1. Implement the function `create_film`. Review the function's docstring to better understand its expected behavior. Return a dictionary literal composed of key-value pairs specified in the docstring. No type conversions are required.

2. Refactor the function `create_starship`. Add code above the dictionary literal that performs the following tasks:

    1. Write an `if-else` statement that checks whether or not the passed in `data` possesses a "films" key-value pair (perform a truth value test). If the condition evaluates to `True` proceed to step 2; otherwise assign `None` to a variable named `films`.

    ```
    if < expression >:
        films = []
        # TODO Implement
    else:
        films = None
    ```

    2. If `data` includes the key-value pair create an empty accumulator list named `films`.

    3. Loop over the `data` "films" value and for each film URL encountered call the function `get_resource` and pass it the URL. Assign the return value to a local variable named `film`.

    4. Then call the function `create_film` and pass it `film`. Assign the return value to `film`.

    5. Append `film` to `films`.

    💡 Steps 2-5 can be implemented in one line of code if you employ a list comprehension.

    6. After adding the `if-else` blocks add another key-value pair to the starship dictionary literal in position eight.

        - film_credits

    7. Assign `films` to the "film_credits" key.

3. Return to `main` and call the `utl` module's `write_json` function and write `utl.cache` serialized as JSON to a file named `stu-cache.json`.

4. Run `lecture_25.py` and update the file `stu-starships.json`. Confirm that each starship's "film_credits" value has been converted to a list of film dictionaries.

```
{
   ...
   "film_credits": [
     {
       "url": "https://swapi.py4e.com/api/films/1/",
       "title": "A New Hope",
       "director": "George Lucas",
       "release_date": "1977-05-25",
       "opening_crawl": "It is a period of civil war...."
     },
     ...
   ]
 }
```