SI 506 Lecture 25

Topics

1. Data caching
    1. `five_oh_six` cache
    2. Deep copies of mutable objects
2. Type checking with `isinstance()`
3. `create_< entity >()` functions
4. Challenges
    1. Challenge 01
    2. Challenge 02
    3. Challenge 03
    4. Challenge 04
    5. Challenge 05
    6. Challenge 06
    7. Challenge 07

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.
- **Deep copying**. Constructs a new compound object from a given mutable object (e.g., `dict`, `list`), recursively copying into it objects found in the original.
- **Higher order function**: a function that acts on or returns other functions. Built-in functions such as `map()`, `filter()`, and `sorted()` are consider higher-order functions as are the method `list.sort()` and the function `functools.reduce()`.
- **Key function**: A key function or collation function is a callable that returns a value used for sorting or ordering. Both `list.sort()` and the built-in function `sorted()` can be passed a key function such as a lambda as an optional argument in order to specify how a sequence is to be ordered.
- **Lambda**: An anonymous inline function consisting of a single expression which is evaluated when the function is called.
- **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 Data caching

Data caching is an optimization strategy designed to reduce duplicate data requests by storing the data retrieved locally. If the data is again required it can be fetched from the local cache, which typically results in a performance boost.

💡 Web browsers cache text, images, CSS styles, scripts, and media files in order to reduce load times whenever you revist a page.

A cache is usually designed to hold data temporarily; cached data is allowed to "expire" after a given interval and is either refreshed or removed from storage. Cache controls and expiration policies help to reduce the chance that data held in the cache no longer matches the origin data.

Python provides a number of built-in caching options including a "Least Recently Used" (LRU) memoization strategy implemented by the `functools.lru_cache` or the third-party `cachetools` package. However, the "five_oh_six" caching infrastructure will employ a Python dictionary.

1.1 `five_oh_six` cache

When retrieving a SWAPI representation of a person, the opportunity exists to retrieve additional relevant information since certain values stored in the JSON document are in the form of URLs that can be used to retrieve representations of SWAPI planets, species, starships, vehicles, and films. Each URL is an identifier that represents an object that can be retrieved from a particular location (e.g.,`https://swapi.py4e.com`).

SWAPI data structures like that of the Corellian smuggler Han Solo and his partner the Wookiee Chewbacca are inspired by linked data design principles that undergird the semantic web.

Han Solo's and Chewbacca's SWAPI JSON documents contains a number of links that associate the smugglers with other Star Wars entities:

```
[
   {
    "name": "Han Solo",
    ...
    "homeworld": "https://swapi.py4e.com/api/planets/22/",
    "films": [
        "https://swapi.py4e.com/api/films/1/",
        "https://swapi.py4e.com/api/films/2/",
        "https://swapi.py4e.com/api/films/3/",
        "https://swapi.py4e.com/api/films/7/"
    ],
    "species": [
        "https://swapi.py4e.com/api/species/1/"
    ],
    ...
    "starships": [
        "https://swapi.py4e.com/api/starships/10/",
        "https://swapi.py4e.com/api/starships/22/"
    ],
    ...
    "url": "https://swapi.py4e.com/api/people/14/"
  },
  {
    "name": "Chewbacca",
```

```
    ...
    "homeworld": "https://swapi.py4e.com/api/planets/14/",
    "films": [
      "https://swapi.py4e.com/api/films/1/",
      "https://swapi.py4e.com/api/films/2/",
      "https://swapi.py4e.com/api/films/3/",
      "https://swapi.py4e.com/api/films/6/",
      "https://swapi.py4e.com/api/films/7/"
    ],
    "species": [
          "https://swapi.py4e.com/api/species/3/"
      ],
    ...
    "starships": [
      "https://swapi.py4e.com/api/starships/10/",
      "https://swapi.py4e.com/api/starships/22/"
    ],
    ...
    "url": "https://swapi.py4e.com/api/people/13/"
  }
]
```

Both data structures share links to three films and two starships. If you were to retrieve Han's and Chewie's linked data from SWAPI, five of the HTTP GET requests would duplicate previous calls. Five unnecessary API requests is something to avoid. Luckily, we can implement a simple cache in order to optimize our interactions with SWAPI.

The `five_oh_six` cache stores data in a dictionary named `cache`. Below is the caching workflow:

1. Check if `cache` contents have been saved to the file system; if true seed the in-memory cache with content stored previously; otherwise return an empty `cache` to the caller.

2. Before every HTTP GET request, check `cache` for the desired resource. This step requires generating a key based on the URL and any querystring parameters before checking whether or not the key is found in the `cache` dictionary's current set of keys.

3. If the desired resource is mapped to a key stored in the `cache`, retrieve it from the `cache` (do not issue the HTTP GET request).

4. If the resource is *not* stored in the `cache`, issue the HTTP GET request and retrieve the resource from the remote service.

5. Generate a key for the resource and update the `cache` with a deep copy of the resource retrieved from the remote service.

6. Persist the mutated `cache` by writing the cache dictionary to a JSON file.

Vital to this caching strategy is the generation of purpose-built "cache" keys that facilitate discovery and retrieval of cached content. The HTTP GET request URL and parameters (if any) can be used to construct a unique key to which the associated SWAPI resource can be mapped.

Creating such keys can be done using the Python standard Library's `urllib.parse` module and two of its functions: `urlencode` and `urljoin`. You can call the `urlencode` function to convert the `params` dictionary passed to the function `get_swapi_resource` into a URL encoding querystring.

URL encoding involves "encoding" or replacing certain characters such as a space with a special character sequence such as '%20' or '+'. After encoding the querystring, the `urljoin` function is called to combine the SWAPI base URL (consisting of a scheme, host, and path) and the querystring (preceded by the `?` separator) for use as a key.

💡 The cache key is itself a valid URL.

```
def create_cache_key(url, params=None):
    """..."""

    if params:
        return urljoin(url, f"?{urlencode(params,
quote_via=quote)}").lower()
    else:
        return url.lower()
```

If you need to retrieve the SWAPI representation of Anakin Skywalker and store the response in the cache, passing `'https://swapi.py4e.com/api/people/'` and `{'search': 'Anakin Skywalker'}` to `create_cache_key` will return the "key" `'https://swapi.py4e.com/api/people/?search=anakin%20skywalker'`.

The remaining steps can be implemented in a few lines of code. The function `get_swapi_resource` is refactored to permit the necessary change in workflow required by the addition of an in-memory cache. Its new task is to check the cache for the desired resource and, if located, a "deep" copy of the resource will be returned to the caller. If the resource is not stored in the cache, the function will delegate to the utility function `get_resource` the task of issuing an HTTP GET request to retrieve the resource from SWAPI. Once the resource is retrieved a deep copy of the resource is mapped to the `key` and deposited in the cache before being returned to the caller.

Script

```
def get_swapi_resource(url, params=None, timeout=10):
    """..."""

    key = utl.create_cache_key(url, params)
    if key in cache.keys():
        return copy.deepcopy(cache[key]) # recursive copy of objects
    else:
        resource = utl.get_resource(url, params, timeout)
        cache[key] = copy.deepcopy(resource) # recursive copy of objects
        utl.write_json('./stu-cache.json', cache) # persist mutated cache

        return resource
```

Utilities module

```python
def get_resource(url, params=None, timeout=10):
    """..."""

    if params:
        return requests.get(url, params, timeout=timeout).json()
    else:
        return requests.get(url, timeout=timeout).json()
```

Data stored in the cache dictionary replicates the decoded JSON documents returned by SWAPI. Recall that SWAPI "searches" return a JSON document comprising four key-value pairs. The decoded document—a dictionary comprising four key-value pairs— is stored in the cache:

```python
{
      'https://swapi.py4e.com/api/people/?search=anakin%20skywalker': {
          'count': 1,
          'next': None,
          'previous': None,
          'results': [
              {
                  'name': 'Anakin Skywalker',
                  ...
              }
          ]
      }
}
```

If a SWAPI representation of Anakin was retrieved employing `https://swapi.py4e.com/api/people/11/` the decoded document would be stored in the cache as follows:

```python
{
    'https://swapi.py4e.com/api/people/11/': {
        'name': 'Anakin Skywalker',
         ...
    }
}
```

From the caller's perspective interacting with the function `get_swapi_resource` remains unchanged. The function's "signature", the mix of required and optional parameters defined for it, has not changed. Implementing the new workflow involves code changes that are all "under the hood."

1.2 Deep copies of mutable objects

When working with mutable objects such as lists one must approach copy operations carefully, especially when working with nested lists. Recall that variables are nothing more than pointers to objects. Objects can hold references to other objects and if the various object types are *mutable* working with (faux) object "copies" can result in unintended mutations of the original object.

You may have noticed when reviewing the refactored function `get_swapi_resource` that a resource *assigned to* the cache or *returned from* the cache are "deep" copies of the resource created using the standard library's `copy` module. The module provides a `copy.deepcopy()` function that returns a new compound object that contains copies of (rather than references to) objects contained in the original.

Python permits both "shallow" and "deep" copying. The official documentation distinguishes between the two operations as follows:

> A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
>
> A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

The following example using the Python shell illustrates the concern and remedy nicely.

```
>>> import copy

>>> nums = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> nums
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

>>> nums2 = nums # not a copy

>>> nums2.append([10, 11, 12])
>>> nums2
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

>>> nums
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

>>> nums3 = nums.copy() # shallow copy

>>> nums3.append([13, 14, 15])
>>> nums3
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15]]

>>> nums
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

>>> nums3[0][0] = 1000
>>> nums3
[[1000, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15]]

>>> nums
[[1000, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```

```
>>> nums4 = copy.deepcopy(nums)
>>> nums4
[[1000, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

>>> nums4[0][1] = 2000
>>> nums4
[[1000, 2000, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

>>> nums
[[1000, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```

Observations

1. `nums2` is *not* a copy of `nums`. `nums2` is nothing more than a second pointer/label/name to the `list` (also) known as `nums`.
2. Mutating `nums2` mutates the `list` named `nums`.
3. `nums3` is a *shallow copy* of `nums`. This means that while you can add *new* elements to `nums3` without mutating `nums`, you cannot mutate *nested* list elements derived from `nums` without also mutating `nums`. Shallow copies contain *references* to objects inserted into it *not* copies.
4. `nums4` is a *deep copy* of nums returned by passing `nums` to the `copy` module's `deepcopy` function. `nums4` is a true copy of `nums` without residual references to the original list. Mutating `nums4` element values does not mutate the values contained in the original list.

2.0 Type checking with `isinstance()`

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 a value's type using the built-in function `isinstance()`.

Signature: `isinstance(< object >, < type >)`

In the example below `isinstance()` is employed to check the type assigned to the variable `jedi`. Assigning a different type to `jedi` results in a different operation being performed.

❗ Avoid using `isinstance()` to check if an object is `None`. You will trigger a runtime exception if you attempt the following:

```
isinstance(obj, None) # Triggers a TypeError
isinstance(obj, NoneType ) # triggers a NameError
```

You can pass the built-in function `type(None)` to `isinstance()` successfully but the preferred way to check if a value is `None` is to either check the object's truth value (negated) of use the identity operator `is`:

```python
isinstance(obj, type(None))

if not obj:
   ...

if obj is None:
   ...
```

START HERE

```python
jedi = []
# jedi = {}
# jedi = ''
# jedi = None
if isinstance(jedi, list):
   jedi.append(obi_wan)
   jedi.append(anakin)
elif isinstance(jedi, dict):
   jedi['obi_wan'] = obi_wan
   jedi['anakin'] = anakin
elif isinstance(jedi, str):
   jedi = f"{anakin['name']}, {obi_wan['name']}"
elif not jedi:
   jedi = f"{anakin['url']}, {obi_wan['url']}"
# elif jedi is None:
#     jedi = f"{anakin['url']}, {obi_wan['url']}"
```

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

3.0 `create_< entity >()` functions

SWAPI entity representations contain data that may prove superfluous to your needs. Thinning the data and/or converting values to different types is best handled by implementing "helper" functions to handling the reshaping of the data. Both `create_person()` and `create_species()` illustrate the technique:

```python
def create_person(data):
    """..."""

    if data.get('species'):
        species_data = get_swapi_resource(data['species'][0]) # checks
cache
        species = create_species(species_data) # trim
    else:
        species = None

    return {
        'url': data.get('url'),
        'name': data.get('name'),
        'birth_year': data.get('birth_year'),
        'species': species
        }


def create_species(data):
    """..."""

    return {
        'url': data.get('url'),
        'name': data.get('name'),
        'classification': data.get('classification'),
        'designation': data.get('designation'),
        'average_lifespan':
utl.convert_to_int(data.get('average_lifespan')),
        'language': data.get('language')
    }
```

As the function `create_species()` illustrates the `create_*()` functions work in tandem with utility functions such as `utl.convert_to_int()` to both clean and manipulate data.

4.0 Challenges

4.1 Challenge 01

Task: Refactor (i.e., revise) the utility functions `convert_to_float()` and `convert_to_int()` so that each function can convert a number masquerading as string that includes one or more commas employed as a thousands separator (e.g., "5,000,000").

1. Review the `convert_to_float()` and `convert_to_int()` docstrings and then refactor each function so that a string such as "506,000,000" can be converted successfully to either a float or an integer respectively.

2. After refactoring the functions return to your program's `main` function, uncomment the Challenge 01 `assert` statements, and run the program. If you raise an `AssertionError` return to the utility module and revise your code.

   ❗ Do not assume that the following `str` methods can identify a number typed as a string that includes thousand separator commas or decimal separator periods:

   ```
   >>> num = "506,000,000.9999"
   >>> num.isnumeric()
   False
   >>> num.isdigit()
   False
   >>> num.isdecimal()
   False
   ```

4.2 Challenge 02

Task: Refactor (i.e., revise) the utility function `convert_to_int()` a second time so that the function can convert a float masquerading as string (e.g., "500,000,000.9999") to an integer.

1. Tweak `convert_to_int()` so that a string such as "500,000,000.9999" can be converted successfully to an integer.

2. After refactoring the function return to your program's `main` function, uncomment the Challenge 02 `assert` statement, and run the program. If you raise an `AssertionError` return to the utility module and revise your code.

   ❗ Some countries utilize a comma as a decimal separator. In such cases, you must first set a new *locale* that change the locale to reflect use of a different separator:

   ```
   >>> import locale
   >>> locale.setlocale(locale.LC_ALL, 'de_DE.UTF-8')
   'de_DE.UTF-8' # Germany
   >>> num = '506,5'
   >>> locale.atof(num)
   506.5
   ```

   This is not an issue that you will confront in SI 506 but you might encounter it at some point later in life so take heed.

4.3 Challenge 03

Task: Fix the utility function `convert_to_list` so that it can split a string successfully with or without a provided `delimiter` value.

1. Review the `convert_to_list()` docstring and then refactor the `try` block so that a string `value` passed to the function is returned to the caller as a list.

   Requirements

   1. Remove leading/trailing spaces from the passed in `value`.

   2. Split the `value` into a list using either the passed in `delimiter` value or the `str.split()` method's default delimiter.

2. After refactoring the function return to your program's `main` function, uncomment the Challenge 03 `assert` statements, and run the program. If you raise an `AssertionError` return to the utility module and revise your code.

4.4 Challenge 04

Task: Combine SWAPI data with data sourced from Wookieepedia.

1. In `main()` uncomment code that read the files `episode_iv_starships.json` and `wookieepedia_starships.csv` and assigns the return values to `swapi_starships` and `wookiee_starships` respectively.

2. Write a nested `for` loop that loops over `swapi_starships` (outer loop) and `wookiee_starships` (inner loop). Position an `if` statement inside the appropriate loop that performs a *case insensitive* comparison of the SWAPI and Wookieepedia starships's "model" key-value pair. If the starships' "model" value matches update the current SWAPI starship dictionary with the Wookiee starship dictionary. Then break out of the inner loop to avoid unnecessary inner looping and proceed to the next outer loop iteration.

3. After implementing the nested loop and updating the `swapi_starships` dictionaries call the `utl` module's `write_json` function and write the `swapi_starships` list serialized as JSON to a file named `stu-starships_v1p0.json`.

4.5 Challenge 05

Task: Refactor the function `create_starship` so that certain values are converted to more appropriate types.

1. Review the `create_starship()` docstring and then refactor the function so that each of the following values are converted to a more appropriate type:

Value	Type	Map to	Convert to	Notes
hyperdrive_rating	`str`	hyperdrive_rating	`float`	

Value	Type	Map to	Convert to	Notes
MGLT	`str`	top_speed_mglt	`int`	
crew	`str`	crew_size	`int`	
armament	`str`	armament	`list`	Check `stu-starships_v1p0.json` for the appropriate delimiter/separator to pass as an argument.

2. 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`.

3. Call the `utl` module's `write_json` function and write `starships` serialized as JSON to a file named `stu-starships_v1p1.json`.

4.6 Challenge 06

Task Sort `starships` *in-place* using a `lambda` function as the key function.

1. Perform an *in-place* sort of the `starship` list. Pass to the appropriate list method as the "key" function a `lambda` expression that sorts the list of starships by their "name" value in *descending* order.

2. Call the `utl` module's `write_json` function and write `starships` serialized as JSON to a file named `stu-starships_v1p2.json`.

4.7 Challenge 07

Task Sort `starships` using the built-in function `sorted()` and a `lambda` function that applies multiple conditions to the sort.

1. Sort the `starships` list using `sorted()`. Sort on the starship's hyperdrive rating and then by its name, both in ascending order.

   💡 A starship's hyperdrive permitted high speed galatic travel in the alternate dimension known as [hyperspace](). The lower the hyperdrive rating the faster the starship.

2. Pass `starships` and a `lambda` expression as arguments. The `lambda` expression *must* apply the two sorting conditions ordered as follows:

   1. Sort by the "hyperdrive_rating" (ascending order)

      ❗ The hyperdrive rating for the TIE/LN starfighter is **blank** (all other starship hyperdrive rating values are integers). Employ the _*ternary operator* in your `lambda` expression to handle this exception and choose any number greater than 4 to use as the `else` value.

```
[value when True] if [expression] else [value when False]
```

2. Sort by "name" (ascending order)

Assign the return value of `sorted()` to a variable named `hyperdrive_ratings`.

3. Call the `utl` module's `write_json` function and write `hyperdrive_ratings` serialized as JSON to a file named `stu-starships_v1p3.json`.