

SI 506 Lecture 24

Topics

1. The Python module
2. `try` and `except` statements
3. Data caching
4. `create_*` entity helper functions

Vocabulary


- **cache:** Storage location that holds data in order to 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 The Python module

Recall that Python features *two* file execution modes. Code in a file can be executed as a script from the command line or the code can be *imported* into another Python file in order to access its definitions and statements.


If a Python file is executed from the command line the Python interpreter will run the file under the special name of `__main__` rather than the program's actual file name (e.g., `lecture_XX.py`). We refer to such a file as a *script* or a program.

A *module* is 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. If a Python file is imported as a *module* into another Python file it is known by its file name.

 Arguably, all Python files including scripts are modules since their definitions and statements can be imported into another Python file. But importing a Python script into another can result in unintended execution flow side effects so you need to think carefully about the purpose of each file you write as you modularize your code.

1.1 Importing modules

A module's definitions and statements can be *imported* into a Python program or script by referencing the module in an `import` statement.

 By convention `import` statements are located at the top of a Python file, although such placement is not required.


You've already learned that the [Python standard library](#) includes a number of "built-in" modules that must be explicitly imported in order to be used. Third-party libraries such as the `requests` package can also be

installed and imported as modules.

```
import csv
import json
import requests
```

Local Python files can also be imported as modules. The filename less the `.py` extension constitutes the module name.


```
lecture_24.py          <-- intended as a script
lecture_24_utils.py    <-- intended as a module
```

 Review the module `lecture_24_utils.py`. Note that it imports other modules including `json` and `requests`.

```
import lecture_24_utils
```

Aliasing imported modules is accomplished using the reserved word (also called a *keyword*) `as` and specifying a short alias that is easy to comprehend:

```
import lecture_24_utils as utl
```

 Aliasing modules is recommended practice whenever working with a module name that is longer than 4-5 characters.

You can also import a module's definitions and statements directly using the `from` keyword:

```
from lecture_24_utils import SWAPI_ENDPOINT, get_swapi_resource,
read_json, write_json
```

```
from lecture_24_utils import (
    convert_data, convert_to_float, convert_to_int, get_swapi_resource,
    SWAPI_ENDPOINT, SWAPI_CATEGORIES, SWAPI_FILMS, SWAPI_PEOPLE,
    SWAPI_PLANETS,
    SWAPI_SPECIES, SWAPI_STARSHIPS, SWAPI_VEHICLES, read_json, write_json
) # (...) permits import statement to be expressed across multiple
lines
```

You are even permitted to use a wildcard (*) in order to import all definitions and statements in a module (except those whose names begin with an underscore (_)):

```
from lecture_24_utils import *
```

! Although convenient, employing a wildcard to import a module's definitions and statements is *not recommended* because it introduces an unknown set of names that may overlap statements and defined earlier. It's opaqueness also undermines code readability.



You can access a module name by using dot notation to reference the "dunder" `__name__` value.

```
import lecture_24_utils as utl

module_name = utl.__name__
```

1.2 Accessing module definitions and statements

Employ dot notation (.) to access a module's definitions and statements.

In the example below dot notation is used in the importing script or module to access both the function `get_swapi_resource` and the constant `SWAPI_PLANETS` (a URL) contained in the module `lecture_24_utils`.

```
import lecture_24_utils

response =
lecture_24_utils.get_swapi_resource(lecture_24_utils.SWAPI_PLANETS,
{'search': 'hoth'})
```

Given the module name's length use of an alias is recommended instead:

```
import lecture_24_utils as utl

response = utl.get_swapi_resource(utl.SWAPI_PLANETS, {'search': 'hoth'})
```

As noted above, you can also import definitions and statements directly employing the `from` keyword. Note that this approach does *not* introduce the module name from which the names are drawn to the importing script or module.

```
from lecture_24_utils import get_swapi_resource, SWAPI_PLANETS

response = get_swapi_resource(SWAPI_PLANETS, {'search': 'hoth'})
```

You can also bind aliases to the imported names:

```
from lecture_24_utils import get_swapi_resource as get, SWAPI_PLANETS as url

response = get(url, {'search': 'hoth'})
```

1.3 Built-in `dir()` function

The built-in `dir()` function can be used to return a list containing a module's definitions and statement names.

```
utl_names = dir(utl)

print(f"\nutl module's names = {utl_names}")
```

2.0 `try` and `except` statements

SWAPI JSON objects contain values of two types: strings and arrays (i.e., lists). For example, the ice planet `Hoth` contains a number of strings that could be converted to other types (e.g., `int`, `float`, `list`, `None`) or expressed in a different way (`'gravity': {'unit_of_measure': 'standard': 'value': 1.1}`).

```
{
  "name": "Hoth",
  "rotation_period": "23",
  "orbital_period": "549",
  "diameter": "7200",
  "climate": "frozen",
  "gravity": "1.1 standard",
  "terrain": "tundra, ice caves, mountain ranges",
  "surface_water": "100",
  "population": "unknown",
  "residents": [],
  "films": ["https://swapi.py4e.com/api/films/2/"],
  "created": "2014-12-10T11:39:13.934000Z",
  "edited": "2014-12-20T20:58:18.423000Z",
  "url": "https://swapi.py4e.com/api/planets/4/"
}
```

We could target certain Hoth key-value pairs for conversion to a integer after reading the JSON into a dictionary:

```
int_keys = ('diameter', 'orbital_period', 'rotation_period',
            'surface_water')
```

```

hoth_v1 = utl.read_json('./fxt-hoth_v1.json')
for key, val in hoth_v1.items():
    if key in int_keys:
        hoth_v1[key] = int(val)

```

But what if certain of the targeted key-value pairs contained values that cannot be converted to an integer such as "n/a" or `None` as is the case in `fxt-hoth_v2.json`? If we ran the code above after reading in `hoth_v2` we would trigger a runtime `TypeError` when passing the "orbital_period" to the built-in function `int()`.

 If the "surface_water" value was passed to `int()` a `ValueError` exception would be triggered.

```

{
  "name": "Hoth",
  "rotation_period": "23",
  "orbital_period": null,
  "diameter": "7200",
  ...
  "surface_water": "n/a",
  ...
  "url": "https://swapi.py4e.com/api/planets/4/"
}

```

```

hoth_v2 = utl.read_json('./fxt-hoth_v2.json')
for key, val in hoth_v2.items():
    if key in int_keys:
        hoth_v2[key] = int(val) # Triggers a runtime exception

```

```

...
TypeError: int() argument must be a string, a bytes-like object or a real
number, not 'NoneType'

```

Given a large planetary data set the presence of such problematic values is not so easily discerned. Luckily, Python provides `try` and `except` statements that allow for the handling of exceptions at runtime.

In the type conversion example above you can guard against exceptions by employing `try` and `except` statement blocks. The utility function `convert_to_int` illustrates their use.

```

def convert_to_int(value):
    """Attempts to convert a string or a number to an int. If unsuccessful
    returns the value unchanged. Note that this function will return True
    for
    boolean values, faux string boolean values (e.g., "true"), "NaN",
    exponential notation, etc.

```

```
Parameters:
    value (str|int): string or number to be converted

Returns:
    int: if value successfully converted else returns value unchanged
"""

try:
    return int(value)
except:
    return value
```

When a **value** is passed to **convert_to_int** the Python interpreter will attempt to execute the **try** clause. Should the **try** block result in an exception, the interpreter will proceed directly to the **except** clause and execute its statement block, thus avoiding the termination of the program's execution due to an exception (such as a **ValueError**).

An **except** clause may specify a specific exception type or multiple exceptions expressed as a parenthesized tuple:

```
except ValueError:
    ...
```

```
except (AttributeError, TypeError, ValueError):
    ...
```

Also a **try** statement may be accompanied by more than one **except** clause in order to specify different handlers for different exceptions. An **else** clause can be added after the **except** statement(s) in order to include code that *must* be executed if the **try** statement block does not raise an exception.

! If an exception occurs that does not match the specified exception(s) named in the **except** clause an *unhandled* exception is triggered and code execution will cease as a result.

3.0 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 revisit a page.

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.

Python provides a number of built-in caching options including the `cachetools` a Least Recently Used (LRU) memoization strategy implemented by the `functools.lru_cache` or the third-party `cachetools` package. However, today we will create a cache using a Python dictionary.

3.1 SWAPI 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](#).

```
[
  {
    "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.

You can create a simple cache using a dictionary. Then implement the following workflow:

1. Before every HTTP GET request, check the cache for the desired resource. This step requires generating a key based on the URL and any querystring parameters and checking whether or not the key is found in the cache dictionary's current set of keys.
2. If the resource is stored in the cache, retrieve it from the cache (do not issue the HTTP GET request).
3. If the resource is *not* stored in the cache, issue the HTTP GET request and retrieve the resource from the remote service.
4. Update the cache with a copy of the resource retrieved from the remote service.
5. Optionally, persist the cache by writing the cache dictionary to a JSON file.

Step 1 requires purpose-built "cache" keys that facilitate discovery and retrieval. 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 '+' or '%20'. 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.

```

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

    if params:
        querystring = f"?{urlencode(params)}" # prefix '?' separator
        return urljoin(url, querystring).lower() # space replaced with '+'
    else:
        return url.lower()

```


If you wanted to retrieve 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+skywalker'`.



The cache key is itself a valid URL.

Steps 2-4 can be implemented in a few lines of code. One could modify the function `get_swapi_resource` in order to accommodate the new workflow but that would overload the function with tasks that it should not perform. Instead, consider implementing a separate function that determines whether or not to retrieve data from the cache or issue an HTTP GET request to the remote endpoint by calling `get_swapi_resource`. The function `get_resource` illustrates the approach:

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

    key = create_cache_key(url, params)
    if key in cache.keys():
        return cache[key] # retrieve from cache
    else:
        resource = get_swapi_resource(url, params, timeout)
        cache[key] = resource # add to cache
        return resource
```

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 entire decoded document—a dictionary comprising four key-value pairs—is stored in the cache:

```
{
    'https://swapi.py4e.com/api/people/?search=anakin+skywalker': {
        '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:

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

Accessing cached resources is identical to accessing the "payload" in a SWAPI response:



The constant `utl.SWAPI_PEOPLE` is assigned the base URL "https://swapi.py4e.com/api/people/".

```
response = utl.get_resource(utl.SWAPI_PEOPLE, {'search': 'Anakin
Skywalker'})
anakin = response['results'][0]
```

or

```
anakin = utl.get_resource(utl.SWAPI_PEOPLE, {'search': 'Anakin Skywalker'})
['results'][0]
```

or

```
anakin = utl.get_resource(f"{utl.SWAPI_PEOPLE}11/")
```

4.0 `create_*` entity helper 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:

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

    if data.get('species'):
        species_data = utl.get_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')
    }
```