24.7. Caching Response Content

You haven't experienced it yet, but if you get complicated data back from a REST API, it may take you many tries to compose and debug code that *processes* that data in the way that you want. (See the Nested Data chapter.) It is a good practice, for many reasons, not to keep contacting a REST API to re-request the same data every time you run your program.

To avoid re-requesting the same data, we will use a programming pattern known as **caching**. It works like this:

1. Before doing some expensive operation (like calling `requests.get` to get data from a REST API), check whether you have already saved ("cached") the results that would be generated by making that request.
2. If so, return that same data.
3. If not, perform the expensive operation and save ("cache") the results (e.g. the complicated data) in your cache so you won't have to perform it again the next time.

If you go on to learn about web development, you'll find that you encounter caching all the time – if you've ever had the experience of seeing old data when you go to a website and thinking, "Huh, that's weird, it should really be different now… why am I still seeing that?" that happens because the browser has accessed a cached version of the site.

There are at least four reasons why caching is a good idea during your software development using REST APIs:

- It reduces load on the website that is providing you data. It is always nice to be courteous when using other people's resources. Moreover, some websites impose rate limits: for example, after 15 requests in a 15 minute period, the site may start sending error responses. That will be confusing and annoying for you.
- It will make your program run faster. Connections over the Internet can take a few seconds, or even tens of seconds, if you are requesting a lot of data. It might not seem like much, but debugging is a lot easier when you can make a change in your program, run it, and get an almost instant response.
- It is harder to debug the code that processes complicated data if the content that is coming back can change on each run of your code. It's amazing to be able to write programs that fetch real-time data like the available iTunes podcasts or the latest tweets from Twitter. But it can be hard to debug that code if you are having problems that only occur on certain Tweets (e.g. those in foreign languages). When you encounter problematic data, it's helpful if you save a copy and can debug your program working on that saved, static copy of the data.
- It is easier to run automated tests on code that retrieves data if the data can never change, for the same reasons it is helpful for debugging. In fact, we rely on use of cached data in the automated tests that check your code in exercises.

There are some downsides to caching data – for example, if you always want to find out when data has changed, and your default is to rely on already-cached data, then you have a problem. However, when you're working on developing code that will work, caching is worth the tradeoff.

24.7.1. The `requests_with_caching` module

In this book, we are providing a special module, called `request_with_caching`.

Here's how you'll use this module.

- Your code will include a statement to import the module, `import requests_with_caching`.
- Instead of invoking `requests.get()`, you'll invoke `requests_with_caching.get()`.

You'll get exactly the same Response object back that you would have gotten. But you'll also get a printout in the output window with one of the following three diagnostic messages:

- found in permanent cache
- found in page-specific cache
- new; adding to cache

The permanent cache is contained in a file that is built into the textbook. Your program can use its contents but can't add to it.

The page-specific cache is a new file that is created the first time you make a request for a url that wasn't in the permanent cache. Each subsequent request for a new url results in more data being written to the page-specific cache. After you run an activecode that adds something to the page-specific cache, you'll see a little window below it where you can inspect the contents of the page-specific cache. When you reload the webpage, that page-specific cache will be gone; hence the name.

There are a couple of other optional parameters for the function `requests_with_caching.get()`.

- `cache_file` – it's value should be a string specifying the name of the file containing the permanent cache. If you don't specify anything, the default value is "permanent_cache.txt". For the datamuse API, we've provide a cache in a file called datamuse_cache.txt. It just contains the saved response to the query for "https://api.datamuse.com/words?rel_rhy=funny".
- `private_keys_to_ignore` – its value should be a list of strings. These are keys from the parameters dictionary that should be ignored when deciding whether the current request matches a previous request. The main purpose of this is that it allows us to return a result from the cache for some REST APIs that would otherwise require you to provide an API key in order to make a request. By default, it is set to ["api_key"], which is a query parameter used with the flickr API. You should not need to set this optional parameter.

Save & Run | Original - 1 of 1 | Show in CodeLens

```
1 import requests_with_caching
2 # it's not found in the permanent cache
3 res = requests_with_caching.get("https://api.datamuse.com/words?rel_rhy=happy", per
4 print(res.text[:100])
5 # this time it will be found in the temporary cache
6 res = requests_with_caching.get("https://api.datamuse.com/words?rel_rhy=happy", per
7 # This one is in the permanent cache.
8 res = requests_with_caching.get("https://api.datamuse.com/words?rel_rhy=funny", per
9
```

```
new; adding to cache
[{"word":"nappy","score":703,"numSyllables":2},{"word":"snappy","score":698,"numSyllables":2},{"wo
rd
found in page-specific cache
found in permanent_cache
```

Activity: 1 -- ActiveCode (ac24_7_1)

Data file: `this_page_cache.txt`


```
{"https://api.datamuse.com/words?rel_rhy=happy":"
[{\"word\":\"nappy\",\"score\":703,\"numSyllables\":2},
{\"word\":\"snappy\",\"score\":698,\"numSyllables\":2},
{\"word\":\"scrappy\",\"score\":697,\"numSyllables\":2},
{\"word\":\"sappy\",\"score\":619,\"numSyllables\":2},
{\"word\":\"chappy\",\"score\":537,\"numSyllables\":2},
{\"word\":\"unhappy\",\"score\":497,\"numSyllables\":3},
{\"word\":\"zappy\",\"score\":397,\"numSyllables\":2},
{\"word\":\"yappy\",\"score\":384,\"numSyllables\":2},
{\"word\":\"flappy\",\"score\":364,\"numSyllables\":2},
```


24.7.2. Implementation of the `requests_with_caching` module

You may find it useful to understand how this module works. The source code is not very complicated; we've reproduced it below. You can use it as a template for implementing code for your own caching pattern in other settings.

Note

This module is not available outside this textbook; in a full python environment you won't be able to install a requests_with_caching module. But you can copy the code and make it work outside the textbook environment.

Note

We have optimized this code for conceptual simplicity, so that it is useful as a teaching tool. It is not very efficient, because it always stores cached contents in a file, rather than saving it in memory. If you are ever implementing the caching pattern just for the duration of a program's run, you might want to save cached content in a python dictionary in memory rather than writing it to a file.

The basic idea in the code is to maintain the cache as a dictionary with keys representing API requests that have been made, and values representing the text that was retrieved. In order to make our cache live beyond one program execution, we store it in a file. Hence, there are helper functions `_write_to_file` and `read_to_file` that write a cache dictionary to and read it from a file.

In order for the textbook to provide a cache file that can't be overwritten, we distinguish between the permanent file, which is provided as part of the online textbook, and a temporary cache file that will live only until the page is reloaded.

```
import requests
import json

PERMANENT_CACHE_FNAME = "permanent_cache.txt"
TEMP_CACHE_FNAME = "this_page_cache.txt"

def _write_to_file(cache, fname):
    with open(fname, 'w') as outfile:
        outfile.write(json.dumps(cache, indent=2))

def _read_from_file(fname):
    try:
        with open(fname, 'r') as infile:
            res = infile.read()
            return json.loads(res)
    except:
        return {}

def add_to_cache(cache_file, cache_key, cache_value):
    temp_cache = _read_from_file(cache_file)
    temp_cache[cache_key] = cache_value
    _write_to_file(temp_cache, cache_file)

def clear_cache(cache_file=TEMP_CACHE_FNAME):
    _write_to_file({}, cache_file)

def make_cache_key(baseurl, params_d, private_keys=["api_key"]):
    """Makes a long string representing the query.
    Alphabetize the keys from the params dictionary so we get the same order each time.
    Omit keys with private info."""
    alphabetized_keys = sorted(params_d.keys())
    res = []
    for k in alphabetized_keys:
        if k not in private_keys:
            res.append("{}-{}".format(k, params_d[k]))
    return baseurl + "_".join(res)

def get(baseurl, params={}, private_keys_to_ignore=["api_key"], permanent_cache_file=PERMA
NENT_CACHE_FNAME, temp_cache_file=TEMP_CACHE_FNAME):
    full_url = requests.requestURL(baseurl, params)
    cache_key = make_cache_key(baseurl, params, private_keys_to_ignore)
    # Load the permanent and page-specific caches from files
    permanent_cache = _read_from_file(permanent_cache_file)
    temp_cache = _read_from_file(temp_cache_file)
    if cache_key in temp_cache:
        print("found in temp_cache")
        # make a Response object containing text from the change, and the full_url that wo
uld have been fetched
        return requests.Response(temp_cache[cache_key], full_url)
    elif cache_key in permanent_cache:
        print("found in permanent_cache")
        # make a Response object containing text from the change, and the full_url that wo
uld have been fetched
        return requests.Response(permanent_cache[cache_key], full_url)
    else:
```

```
        print("new; adding to cache")
        # actually request it
        resp = requests.get(baseurl, params)
        # save it
        add_to_cache(temp_cache_file, cache_key, resp.text)
        return resp
```

Check your understanding

requests-7-1: Why is it important to use a function like make_cache_key in this caching pattern rather than just uring the full url as the key?

- (•) A. Because when requests.get encodes URL parameters, the keys in the params dictionary might be in any order, which would make it hard to compare one URL to another later on, and you could cache the same request multiple times.
- ( ) B. Because otherwise, it's too much data in the same function, and the program will not run.
- ( ) C. You don't, actually. This function is just a fancy way of calling requests.get.
- ( ) D. Because the make_cache_key function as written here is what saves the cache data file so you have it later!

Check me | Compare me

✔ Comparing the strings "rowling&harry+potter" and "harry+potter&rowling", they are different as far as Python is concerned, but they are the same as far as meaning to a REST API is concerned! That's why we need to manipulate these strings carefully to always get the same, canonical key for the cache dictionary.

Activity: 2 -- Multiple Choice (restapis_caching_1)

Data file: `datamuse_cache.txt`

```
{
  "https://api.datamuse.com/words?rel_rhy=funny": "[{\"word\":\"money\",\"score\":4423,\"num
}
```

You have attempted 3 of 2 activities on this page

24.6. Fetching a page">
tching a page">
24.8. Figuring Out How to Use a REST API">
✔ Completed. We
24.8. Figuring Out How to Use a REST API">Next Section - 24.8. Figuring Out How to Use a REST API

| Back to top