Learning Objectives

- **Explain how caching speeds up performance**
- **Identify different ways to cache data in Django**
- **Add view caching to different views**
- **Adjust the time for which views are cached**
- **Vary caching based on cookies**
- **Cache only specific parts of a template**
- **Perform lower level caching actions**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

Introduction to Caching Part 1

Introduction to Caching Part 1

Caching is a way of speeding up programs by storing the results of computationally heavy operations, and referring directly to the result instead of calculating it again the next time it's needed.

A basic way of using a cache in Python looks a bit like this, where `cache` is an object that supports getting and setting items.

```
if key in cache:
    # get the value from the cache if it exists
    value = cache.get(key)_
else:
    # calculate the value and store it in the cache so it can be
        fetched next time
    value = calculate_value_for_key(key)
    cache.set(key, value)
```

The cache is actually (usually) a reference to another application running on your (or another) machine, specifically optimized for storing mappings between keys and values. These services also have support for setting a timeout on stored values. Some values we might want to keep in the cache for a long time, if they aren't expected to change often. Others will change more often, so we only want to store them for a little amount of time. The actual timeout values that are chosen will be dependent on the application.

For example:

```
# store the count of all the posts in the system for 300 seconds
        (5 minutes)
cache.set("post_count", post_count, 300)
# if displaying a post count on the site, we'd only need to
        fetch and count them once every 5 minutes
```

Sometimes you might choose to cache a value forever, and instead manually invalidate it when you know that it's changed. For example, we could fetch the content of a `Post` and cache it, then invalidate the cache it when the `Post` is saved, since we know that's the only time it will change.

```
class Post:
    def save(self, *args, **kwargs):
        super(Post, self).save(*args, **kwargs)

        # delete this post content from the cache on save
        cache.delete(f"post_content_{self.pk}")
```

We mentioned that a cache is an application running outside of Django, in most cases, but Django has different backends to choose from, some of which cache inside the Django process. Let's look at these different backends and discuss how they store cached data.

The backend and its related options are set with the `CACHES` setting. Some of the options for backends are:

Memcached

Memcached is an application/service that can run on your machine or another. It listens on a TCP port and multiple Django processes can share the same Memcached backend – this means that as soon as one process has cached a value the others can retrieve it from the cache.

A simple way of using Memcache as your backend is to with this setting:

```
CACHES = {
    "default": {
        "BACKEND":
        "django.core.cache.backends.memcached.PyMemcacheCache",
        "LOCATION"": "127.0.0.1:11211",
    }
}
```

This will try to connect to Memcached on port *11211* on localhost (*127.0.0.1*). Before it will work, though, we need the pymemcache library to support the `PyMemcacheCache` backend. It's easily installed with `pip` though.

Memcached is easy to set up, and fast. It's a well trusted cache used in many production environments.

▼ Installing Memcached

On your Codio environment, as well as with other Ubunutu or Debian systems, Memcached can be installed with a single command:

```
$ sudo apt install memcached
```

There are a couple of things to be aware of with Memcached. First, since Memcached stores everything in memory, if it crashes or is restarted it will lose all cached values. This isn't a critical problem as caches are, by design, meant to only store copies of persistent data for faster retrieval. If the cached data is lost it should be repopulated the next time requests for cacheable data are made.

Second, by default it can only store values of 1MB per key. You usually won't be storing anything this large in day-to-day Django usage, but it could be a possibility depending on your application. Be aware of this limit if your caches don't work as you expect.

Database caching

Django can cache data to your database. This can be fast, but probably not as fast as in-memory cache. It is persistent though and will survive crashes, and you don't need to install any other applications.

This cache backend is set up with this setting:

```
CACHES = {
    "default": {
        "BACKEND":
        "django.core.cache.backends.db.DatabaseCache",
        "LOCATION": "my_cache_table",
    }
}
```

The caches are stored in the table specified by the `LOCATION` key (in this case, `my_cache_table`). The cache table needs to be created before it can be used, with the manage command `createcachetable`:

```
$ python manage.py createcachetable
```

Multiple Django processes across multiple machines can share the same database cache.

Introduction to Caching Part 2

Introduction to Caching Part 2

Filesystem Caching

Django can cache values to the filesystem. This can be fast, depending on the speed of your drive, but probably not as fast as in memory caching. It is persistent though, and data survives crashes. It's set up like this:

```
CACHES = {
    "default": {
        "BACKEND":
        "django.core.cache.backends.filebased.FileBasedCache",
        "LOCATION": "/var/tmp/django_cache",
    }
}
```

The `LOCATION` setting is an absolute path of a directory, which must exist (Django won't create it). It must also be writable by the Django process. Cached values are stored as individual files inside the cache directory. Multiple Django processes on a single machine can share the cache, or it could be shared across the multiple machines if using a network filesystem.

Local-memory Caching

Django can cache data in memory, inside its own Python process. This can give you the speed advantages of in-memory caching without having to set up Memcached. However, since the cache is inside the Python process' memory, it can't be shared across multiple Django instances.

```
CACHES = {
    "default": {
        "BACKEND":
        "django.core.cache.backends.locmem.LocMemCache",
        "LOCATION": "unique-snowflake",
    }
}
```

The `LOCATION` setting is not required, unless you have multiple local-memory caches. In which case, it should be unique for each one.

This is actually the default backend that Django uses, and it's good for development as you can test caching without setting up any external services.

Dummy Caching

Finally there's the Dummy Cache, which has a cache interface but doesn't actually cache anything. This is only for development purposes, as it means you can "cache" data to check that your caching code is being called correctly, but you don't have to actually wait for the cache to expire as you make changes to your code.

It's set up like this:

```
CACHES = {
    "default": {
        "BACKEND":
        "django.core.cache.backends.dummy.DummyCache",
    }
}
```

You might end up switching between Local-memory Caching and Dummy Caching when developing, depending on if you're making changes to most parts of your code or template, or if you specifically want to test how your cache is behaving.

Custom Cache Backends

Django will also let you write your own cache backend, if none of the built in ones are quite right. You can read the custom cache backend documentation for more information.

Other Cache Arguments

There are also a number of arguments that can be provided when setting up the cache backend, the full list is at the cache arguments documentation. Some examples of what you can set are: `TIMEOUT`, the default timeout in seconds if no timeout is specified when setting a value; and `KEY_PREFIX`, a string that's prepended to each key you use.

Now we know about all the cache backends, for development we're going to stick with the `LocMemCache`, which means we don't need to change any settings. Let's look at some ways of using the cache in Django.

View Caching

View Caching

Probably the simplest way to add caching anywhere in Django is to use a view caching. This means caching the output (response) from a view function, based on its arguments (i.e. the first request for a URL will call the view function and generate a response, subsequent requests will use the cached version).

This method of caching can be set up with the `django.views.decorators.cache.cache_page` function, used as a decorator. It takes one required argument, the number of seconds that the response should be cached for.

Here's how we could use it to cache the response of our index view. First, in `blog/views.py`, import the `cache_page` function:

```
from django.views.decorators.cache import cache_page
```

Then, decorate the `index` view, in this case we want to cache the response for 300 seconds (5 minutes):

```
@cache_page(300)
def index(request):
    # existing view code
```

Now the list of `Post` objects will only be queried once every five minutes, and likewise the template will only be rendered once every five minutes, and all other responses will come from the cache.

This decorator needs to be used carefully though. It assumes that the response from the view is only generated based on the URL, and no other parts of the request. One major security flaw you could introduce is if you decided to cache any page whose content depends on the logged-in user.

Imagine that you have a page that showed a user's profile, at the path `/profile/`. Inside the view, we would use the `request.user` attribute to fetch the correct profile for the current user. Any user who wanted to view their own profile would visit the same URL.

Unfortunately, as the URL is shared, the first person to visit it would have it generate a response and cache it. The next person to visit would get the same response, that is, the first user's profile!

We can see this problem in action, and then look at how to fix it. You'll only be able to test this out if you have more than one browser installed on your computer. Alternatively you might be able to get this test to work by using a browser window in Incognito/Private mode and another window not in Incognito/Private mode.

Look at the `blango/views.py` file. We'll make a temporary change to have it return the current username that's logged in, or *AnonymousUser* if no one is logged in.

Update the `index` function to add two lines directly after the function definition (**Note,** there is not caching for this function):

```
def index(request):
    from django.http import HttpResponse
    return HttpResponse(str(request.user).encode("ascii"))
    posts =
        Post.objects.filter(published_at__lte=timezone.now())
    logger.debug("Got %d posts", len(posts))
    return render(request, "blog/index.html", {"posts": posts})
```

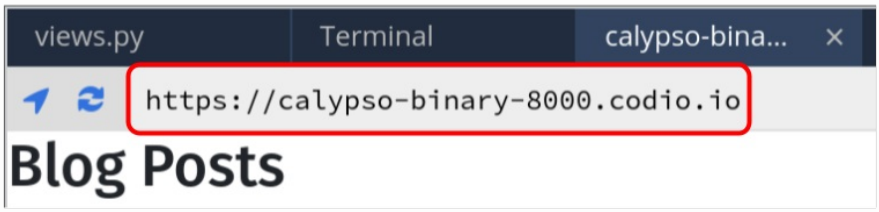
You can leave the rest of the function intact, it will be skipped since we're just making an early return.

Now, open a two browsers windows, or one browser with both a private and non-private browsing window. In each of them go to the index page of Blango. The browser you've been using should show your logged-in username, and the new one should show *AnonymousUser*. If they both show *AnonymousUser*, then in one of them, visit the Django admin login screen, log in, and return to the index page. It should now show your user.

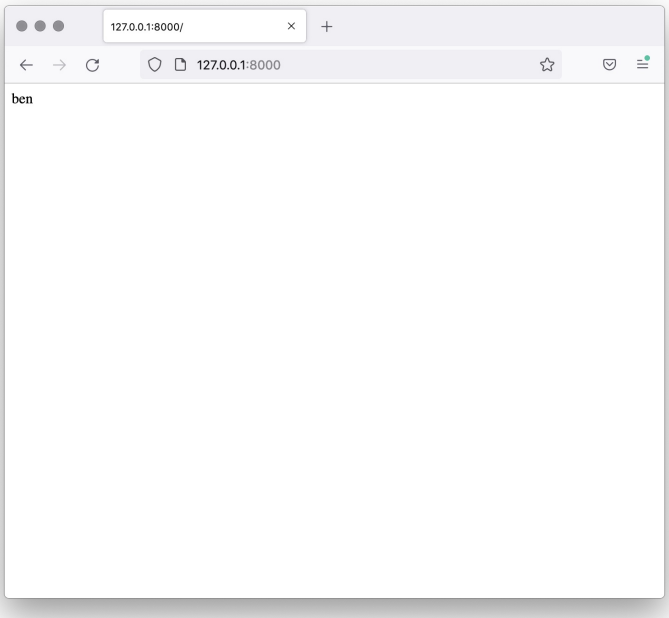
View Blog

▼ **Loading Blango in Another Browser**

If you want to load Blango in another browser, start the dev server and open the blog in Codio. Copy the URL from your Codio browser (see image below) and paste it into another browser.

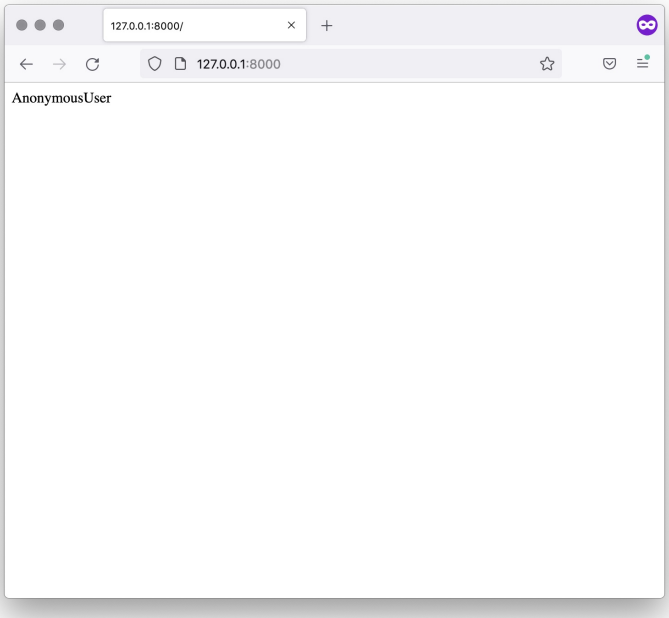


Here's how it looks in Firefox with the logged-in user:



Normal Browser

And the logged-out user:



Private Browser

Now, let's add caching, so we can see the problem. Return to `views.py` and import the `cache_page` function:

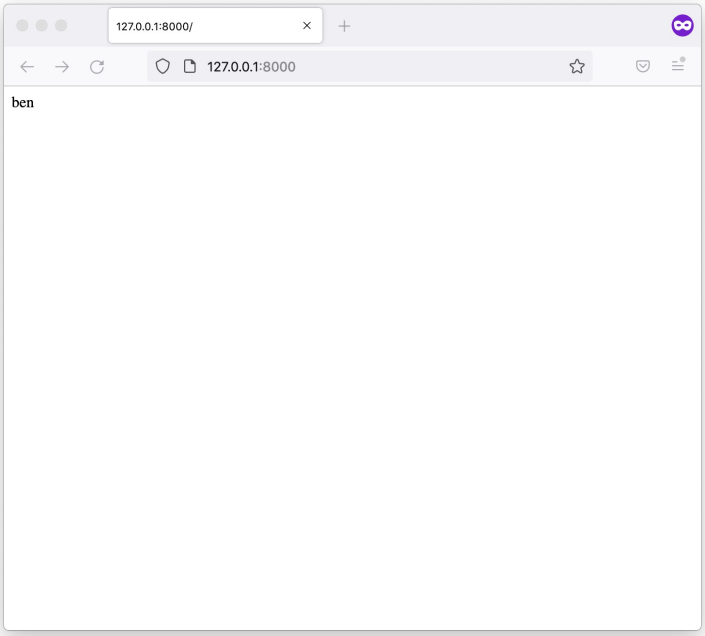
Open views.py

```
from django.views.decorators.cache import cache_page
```

Then decorate the `index` function with it:

```
@cache_page(300)
def index(request):
    from django.http import HttpResponse
    return HttpResponse(str(request.user).encode("ascii"))
    posts =
        Post.objects.filter(published_at__lte=timezone.now())
    logger.debug("Got %d posts", len(posts))
    return render(request, "blog/index.html", {"posts": posts})
```

Return to each browser and refresh each one. You should see that the second one you refreshed has the same username show as the first, whether it's a logged in user or `AnonymousUser` as well. The image below shows this effect, the logged out user sees the logged in user's page:



Private browser

Varying on Headers

Varying On Headers

To fix the problem we can give Django a hint on how the view function will vary its response. `request.user` is populated by data stored in the request session, which in turn is stored in a cookie, a value in the `Cookie` HTTP header. Therefore, Django needs to know that if the value in the `Cookie` header changes, then the response from the view will change.

We can tell Django which headers will cause the response to vary by decorating the view function with the `django.views.decorators.vary.vary_on_headers` function. This function's parameters are names of the headers that will cause the response to vary. In our case, it's just the `Cookie` header, so we can decorate the view like this:

```
@cache_page(300)
@vary_on_headers("Cookie")
def index(request):
    # existing view code
```

Conveniently, since varying on cookies is so common, Django provides a shortcut function for this: `django.views.decorators.vary.vary_on_cookie`. Using this shortcut function should be preferred as it would prevent a security vulnerability if you made a typo like `@vary_on_headers("Cookies")`.

challenge

Try it out:

Modify the `index` function so that it varies on cookies:

- Fix up the view caching by importing `vary_on_cookie` from `django.views.decorators.vary`.
- Decorate your view with it like in the example above. Try refreshing each browser window showing the index page. You should see that each now shows a different user.
- Verify that caching is still active by adding a basic log message before returning the response.

▼ **Solution**

Your imports and `index` function should look something like this:

```
from django.shortcuts import render,
        get_object_or_404,redirect
from django.utils import timezone
from blog.models import Post
from blog.forms import CommentForm
import logging
from django.views.decorators.cache import cache_page
from django.views.decorators.vary import vary_on_cookie

logger = logging.getLogger(__name__)

# Create your views here.
@cache_page(300)
@vary_on_cookie
def index(request):
    from django.http import HttpResponse
    logger.debug("Index function is called!")
    return HttpResponse(str(request.user).encode("ascii"))
    posts =
        Post.objects.filter(published_at__lte=timezone.now()
        )
    logger.debug("Got %d posts", len(posts))
    return render(request, "blog/index.html", {"posts":
        posts})
```

Try refreshing your browser(s), you should see the log message only on the first request for a user. Subsequent requests don't execute the view function, and thus won't output the message. If you want five minutes, or

restart the Django dev server (since we're using local-memory caching) then you should see the message another time.

View Blog

We're finished experimenting with the Django view cache, so we can actually take out all our caching code.

Your `index` function should return to looking like this (no decorators):

```
def index(request):
    posts =
        Post.objects.filter(published_at__lte=timezone.now())
    logger.debug("Got %d posts", len(posts))
    return render(request, "blog/index.html", {"posts": posts})
```

And you can remove the cache helper function imports as we're not using them any more.

We don't have to cache the entire response, for more fine-grained caching Django gives us the ability to cache template fragments. We'll look at that next.

Template Fragment Caching

Template Fragment Caching

Instead of caching the entire response, we can just cache parts of templates that won't change. For example, in our Blango post detail view, the list of recent posts won't change that often, so we could cache it. However comments on the post might change more frequently so we could choose not to cache them, instead preferring to have a live view.

Caching template fragments is easy, there are only two steps:

1. Load the `cache` template tags by adding a `{% load cache %}` near the top of your template.
2. Wrap any code you want cached in `{% cache %}` and `{% endcache %}` tags.

Cache takes two required positional arguments: the timeout, in seconds, and the cache name.

Let's add caching to our `post-detail.html` template, but just around the `recent_posts` template tag. At the top of the file where we're loading `blog_extras`, also load `cache`:

```
{% load blog_extras cache %}
```

Then wrap the `recent_posts` tag. We will cache for 3600 seconds (one hour), and name the cache fragment `recent_posts`:

```
{% cache 3600 recent_posts %}
    {% recent_posts post %}
{% endcache %}
```

That's all that we need to do. Try loading a Post detail page, it should still look the same.

View Blog

How do we know it's working? Let's add some logging to our `recent_posts` template tag function. The log message should only be printed the first time the fragment is rendered. Start by opening the `blog/templatetags/blog_extras.py` file. First we need to import the `logging` module at the top of the file:

Open blog_extras.py

```
import logging
```

Next, create a `logger` variable, add this line after the line that creates the `register` variable:

```
logger = logging.getLogger(__name__)
```

Finally let's add the log call inside the `recent_posts` function, add this on a line after retrieving the posts:

```
logger.debug("Loaded %d recent posts for post %d", len(posts),
        post.pk)
```

Now refresh a post detail page, you'll see the debug message in the console, but subsequent refreshes will not show it, since it's fetching the template fragment from the cache instead.

You may have noticed that there's a problem with how we've cached the fragment: it's the same for every post. This might mean seeing the current post we're viewing in its own related posts link.

We can add extra arguments to the `cache` template tag to make it more specific. In our case, we want to add the post we're viewing as an argument, so that the recent posts list is specific to that post. It's as simple as adding `post` (the current `Post` object in the template) to that tag.

Like this:

```
{% cache 3600 recent_posts post %}
```

Now, the `Post` object will be included the cache key so the list of recent posts will be specific to the post being viewed. Refresh the blog and view some different posts, you should notice the bug is fixed, and a post is never its own related post.

This should cover most use cases for template caching, but the official documentation goes into more detail. For example, you can see how to cache the template fragment to a different cache backend, or see how to retrieve the fragment from the cache in Python code.

If you need even more control over how you cache data, you can use the lower level caching API. We'll look at that now.

Lower Level Cache API Part 1

Lower Level Cache API Part 1

At the start of this module we discussed how a cache can be accessed kind of like a dictionary, we can set and get items for keys. The main difference is that when an item is set we provide a timeout, in seconds. If we try to fetch the item after that timeout has elapsed we will get `None` back.

The first step in accessing the cached items is to get an instance of the cache. Django provides a dictionary containing all cache instances, at `django.core.cache.caches`. The keys to this dictionary match the keys of the `CACHES` setting. Django supports multiple caches with different aliases, but we're just using one: `default`. We can therefore access this cache in our Python code like this:

```
from django.core.cache import caches
default_cache = caches["default"]
```

However, since we're just using the default cache, Django provides a shortcut for us to access it: `django.core.cache.cache`. Or to show it in use:

```
from django.core.cache import cache
# cache is the equivalent of caches["default"]/our default_cache
        variable
```

Going forward, we'll access the default cache using the `cache` variable.

Setting and getting values is easy. The `set()` method takes four arguments:

- `key`: the cache key in which to store the value. Note that depending on the `CACHES` settings the key might have a prefix added before being set in the actual cache.
- `value`: the value to store.
- `timeout` (optional): the timeout of the cache key, in seconds. If omitted this defaults to a `TIMEOUT` value specified in the `CACHES` setting, then to the default timeout for the cache backend – usually 300 seconds.
- `version` (optional): a version number can be added to cache keys so that if you change the format of data this is stored when you deploy a new version of your application, any old values in the cache will still work for older versions that might still be running.

To retrieve a value, use `get()`. It takes three arguments:

- `key`: the cache key from which to retrieve the value.
- `default` (optional): the value to return if the cache key doesn't exist. Defaults to `None`.
- `version` (optional): the version parameter as per the `set()` method.

We won't be implementing low level cache access in Blango, so to demonstrate it we'll just use the Django Python shell. You can follow along with the examples by starting your own Django Python shell with the management command `shell`:

```
python3 manage.py shell
```

You are in the Python shell if you see the following output:

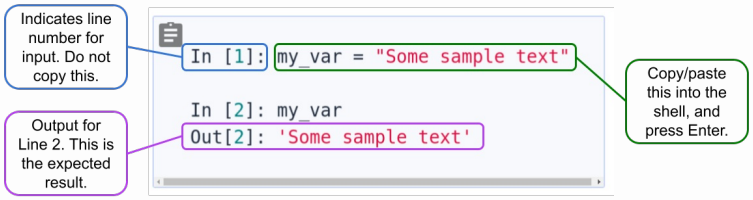
```
DEBUG 2021-08-18 16:47:24,415 selector_events 600
        140543462569792 Using selector: EpollSelector
Python 3.6.9 (default, Apr 18 2020, 01:56:04)
Type "copyright", "credits" or "license" for more information.

IPython 5.5.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra
        details.

In [1]:
```

Entering Shell Commands

When copying/pasting shell commands into the terminal, it is important to not copy the entire code block. This will cause errors. You should only copy lines of code that start with `In [#]:`. This indicates a line of input. Copy everything after the `:`. Lines of code that start with `Out[#]:` are the expected output.



understanding input and output for the shell

First, import the default cache, and since we want to try storing some real data, also the `Post` model:

```
In [1]: from django.core.cache import cache

In [2]: from blog.models import Post
```

Then let's get a `Post` object, and put it in the cache for 30 seconds.

```
In [3]: post_pk = 1

In [4]: p = Post.objects.get(pk=1)

In [5]: cache.set(f"post_{post_pk}", p, 30)
```

Notice we're using the `Post` object's primary key as part of its cache key, so we can be sure the cache key is unique.

Then, let's get `Post` back out of the cache (although if it's been more than 30 seconds since you've set it, you'll need to put it back into the cache otherwise this won't work):

```
In [6]: p1 = cache.get(f"post_{post_pk}")

In [7]: p == p1
Out[7]: True
```

Now wait 30 seconds, and try to retrieve the `Post` object from the cache again.

```
In [8]: print(cache.get(f"post_{post_pk}"))
Out[8]: None
```

The `get()` method now returns `None`.

If we don't want to wait for the timeout to elapse, then we can just use the `delete()` method to remove an item from the cache. For example, if a `Post` was updated we could remove its old, cached version. `delete()` takes two arguments, the `key` to delete and the cache key `version`, which is optional. It will return `True` if the value existed in the cache prior to deletion, or `False` if it did not.

```
In [9]: cache.get(f"post_{post_pk}")

In [10]: cache.delete(f"post_{post_pk}")
Out[10]: True

In [11]: cache.delete("a made up cache key")
Out[11]: False
```

If we try to fetch a value that doesn't exist in the cache, we'll get back `None`, so how can we differentiate between a value that's not in the cache and a stored `None` value? The trick is to pass a sentinel object as the default to `get()`. If you get the same sentinel back, you know the key wasn't set. If you get `None`, you'll know the value `None` was set.

```
In [12]: sentinel = object()

In [13]: cache.set("current_user", None, 30)

In [14]: u = cache.get("current_user", sentinel)

In [15]: u is None
Out[15]: True

In [16]: u is sentinel
Out[16]: False
```

Now wait 30 seconds for the value to timeout.

```
In [17]: u = cache.get("current_user", sentinel)

In [18]: u is sentinel
Out[18]: True
```

Lower Level Cache API Part 2

Lower Level Cache API Part 2

Before we continue, we need to import again `cache` and `Post`.

```
In [1]: from django.core.cache import cache

In [2]: from blog.models import Post
```

For some cache backends, it's much faster to set and get many objects at once. To set many items at once, use the `set_many()` method. This takes three arguments, `data`, `timeout`, and `version`. As with the `set()` method, the latter two are optional. `data` is a dictionary of keys and values to set in the cache. For example, we could pre-load all the `Post` objects and store them in the cache.

```
In [3]: all_posts = Post.objects.all()

In [4]: posts_to_cache = {f"post_{post.pk}": post for post in
        all_posts}


In [5]: posts_to_cache
Out[5]: {'post_7': <Post: Leo: A Personal Profile>, 'post_6':
        <Post: Django vs Python>, 'post_5': <Post: Breaking
        News>, 'post_4': <Post: A Real Post>, 'post_3': <Post:
        Yet another test post!>, 'post_1': <Post: An Example
        Post>, 'post_2': <Post: Advanced Django: A Review>}


In [6]: cache.set_many(posts_to_cache, 30)
Out[6]: []
```

After loading all the `Post` objects, they're stored in a dictionary, keyed by a string in the format `post_{post.pk}`. We then add all these to the cache at once, by calling `set_many()` with a timeout of 30 seconds. `set_many()` returns a list of all the keys that failed to be set in the cache. In our cache, they were all stored successfully so we got back and empty list.

Now we can retrieve an individual `Post` by its cache key:

```
In [7]: cache.get("post_2")
Out[7]: <Post: Advanced Django: A Review>
```

To compliment `set_many()`, there's also the `get_many()` method. It takes two arguments, `keys`, a list of keys to retrieve, and `version` (which is optional). It returns a dictionary containing only the items that exist in the cache.

```
In [8]: cache.get_many(["post_1", "post_2", "post_1000"])
Out[8]: {'post_1': <Post: An Example Post>, 'post_2': <Post:
        Advanced Django: A Review>}
```

Since `post_1000` was not in the cache it's not in the returned dictionary. If none of the keys you requested were in the cache, then a empty dictionary will be returned.

Unlike the `get()` method, there's no `default` argument, so cached `None` values will be in the returned dictionary. There's no need to use a sentinel object to check for these.

```
In [9]: cache.set("none_value", None, 30)

In [10]: cache.get_many(["none_value"])
Out[10]: {'none_value': None}

# wait 30 seconds

In [11]: cache.get_many(["none_value"])
Out[11]: {}
```

If you want to remove multiple keys at once, there's also the `delete_many()` method, which takes a list of keys to remove from the cache, and a `version` argument. It doesn't return anything.

The final cache method we're going to look at in depth is `get_or_set()`. This will get a value from the cache, or set it if it doesn't already exist. It takes four arguments, `key`, `default` (the value to store and retrieve, if `key` is not set in the cache), `timeout` (optional) and `version` (optional).

```
In [12]: cache.set("key1", "value1")

In [13]: cache.get_or_set("key1", "value2")
Out[13]: 'value1'

In [14]: cache.get_or_set("key2", "value3")
Out[14]: 'value3'

In [15]: cache.get_or_set("key2", "value4")
Out[15]: 'value3'
```

`default` can be also be a function that will be called to get the value. As we mentioned right at the start of this section, a common pattern is to check if a value is in cache, if not, get it, and then set it. For example, to retrieve/cache the latest `Post` object, we could do something like this (**Note,** the following code does not go into the Python shell):

```
def get_latest_post():
    return Post.objects.first()


p = cache.get("latest_post")

if not p:
    p = get_latest_post()
    p.set("latest_post", p)
```

This could be simplified to a single `get_or_set` call:

```
p = cache.get_or_set("latest_post", get_latest_post)
```

Note that there are no brackets after `get_latest_post`, as we're just passing in a reference to the function rather than calling it and passing in the result. The `get_or_set()` method will only call the function if the cache key is not set.

That's the last cache method we're going to look at in detail, but here's a brief overview of a few more that you may find useful:

- `add(key, value, timeout=DEFAULT_TIMEOUT, version=None)`: Set a value in the cache but only if they key doesn't already exist. Returns `True` if the value was set or `False` if it was not (i.e. the key was already in the cache).
- `touch(key, timeout=DEFAULT_TIMEOUT, version=None)`: Refresh the timeout of the `key`, that is key will now expire `timeout` seconds from now. Returns `True` if the timeout was updated successfully or `False` if it wasn't; `False` will be returned if the key is not in the cache.
- `incr(key, delta=1, version=None)`: Increment an integer value that's set in for `key`, by `delta`, then return the new value. Will raise a `ValueError` if the `key` is not set. For some backends, like Memcache, this operation is atomic.
- `decr(key, delta=1, version=None)`: The opposite of `incr()`.
- `has_key(key, version=None)`: Return `True` if `key` is in the cache and not expired, `False` otherwise.
- `clear()`: Removes **all** the values from the cache.

That's all we're going to cover for using the cache in Django. As usual, the official cache documentation goes more in depth with other options and configuration for more specific use cases.

In the next assignment, we're going to look at getting more performance by optimizing the way we use the database.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish caching"
```

- Push to GitHub:

```
git push
```