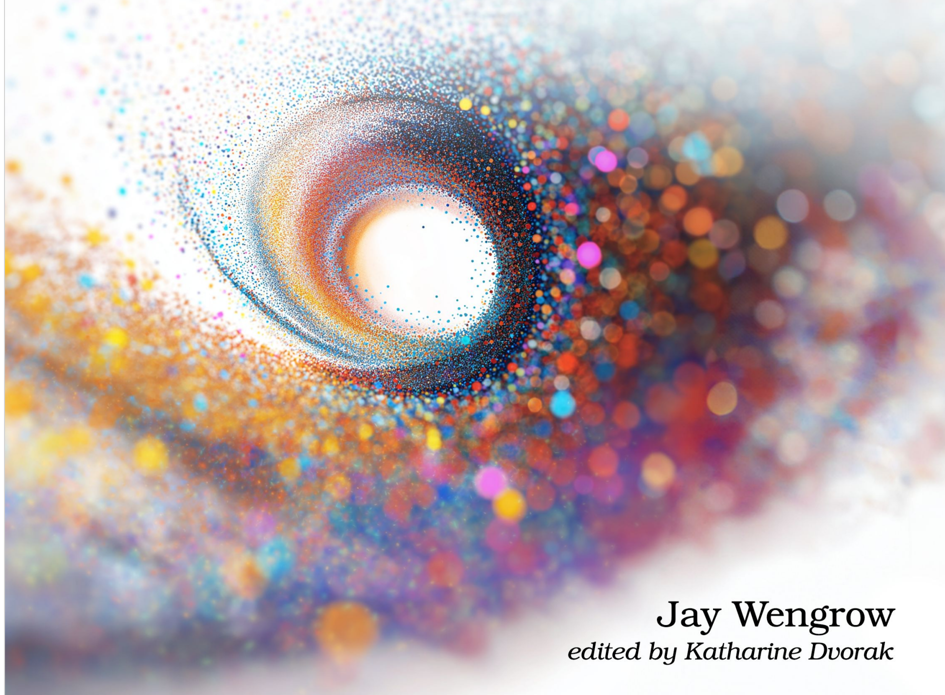
The Pragmatic Programmers

Volume 2

A Common-Sense Guide to

Data Structures and Algorithms in Python

Level Up Your Core Programming Skills



This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Copyright © The Pragmatic Programmers, LLC.

CHAPTER 4

Cache Is King

In the previous chapter, I introduced randomized algorithms and explained how randomization can boost the efficiency of all sorts of applications. In this chapter, you'll discover how randomization can also boost the speed of a process known as *caching*.

Caching is built into the hardware of virtually all computers, and affects the speed of the code you write. At the same time, you can create software that performs additional layers of caching as well.

But whether you write your own caching code or leverage the caching built into your computer, understanding caching—and how randomization can play a role in it—will allow you to take your code's speed to the next level.

In addition, caching is another key piece of the puzzle of why two algorithms can have different actual-time speeds even though they execute the same number of steps. Understanding the related concept of *spatial locality* can help you design faster algorithms even when Big O analysis tells you that you can't beat the speed of the algorithm you already have.

So, without further ado, let's dig into caching.

Caching

A fundamental rule about computers is that the farther the data is from your computer, the longer it takes for your computer to retrieve it. For example, you can open a file that's local to your computer more quickly than you can download information from the Internet. This is because the Internet consists of servers that are *outside* your computer, and it therefore takes more time for your computer to get a hold of those servers' data.

Let's take a look at an example of how this can be a major deal for the software we write.

Imagine that we're building an app that searches the web for the cheapest price available for various physical products. The user enters something like "Vroom-Master Vacuum Cleaner 3000," and our app scours the Internet to find whichever online retailer is selling it for the cheapest. Note that we're not building a massive database that stores a gazillion products and their prices. Instead, our software is searching the web each time the user searches for a particular product. In a sense, the web is our "database."

Now, say that the Vroom-Master's latest model is becoming a hot fad; everyone you know is buying one. Suddenly, our app finds itself repeatedly searching the web for the best bargain for the Vroom-Master 3000. Assuming that stores are not constantly changing their prices at a rapid clip, it's kind of a shame that our software has to search the web *each and every time* someone asks for the Vroom-Master 3000. Searching the web takes time! Wouldn't it be nice if the app could just *remember* information the first time it finds it, and then not have to search the web again and again for the same information?

Luckily for us, that's exactly what a cache does.

What Is a Cache?

A *cache* (pronounced "cash") is simply a data container that takes data that was retrieved from a far-away source and stores it locally. (The word "cache" can also be used as a verb. That is, our computer can *cache* data it gets from the web.) Because the data is more local, we'll be able to retrieve it more quickly in the future. The definition of "local" can change based on context, but for now let's say that data that lives on our computer is considered local, while data retrieved from the Internet is "far away."

In theory, then, we can create our own code-based cache for the app we're building. That is, our app will take data it gets from the web and store it on the computer, tablet, or smartphone that the software is operating on. We may, for example, have our code initialize some data structure and store the data in it.

So, the first time someone searches for the Vroom-Master 3000, our app will search the web for it and then save the desired information in the user's local data structure, which serves as our cache. This way, the next time someone asks for this information, our app doesn't have to search the web again; it can instead just retrieve the data from the cache.

Now, this behavior may not be desirable for websites that are constantly changing. In this case, the cache can become what is known as *stale*. That is, the website may actually have been updated with new information, but

the app is pulling up the outdated info saved in the cache. But for websites that don't update often, the local cache can save us a lot of time.

Code Implementation: A Hash-Table Cache

We have a variety of options as to which data structure we might choose to serve as our cache. However, a hash table is a natural fit for housing a cache, since data can be stored and retrieved from a hash table in O(1) time. The following sample code demonstrates how we might use a hash table to serve as a cache:

```
import time

cache = {}

def lowest_price(product):
    if product in cache:
        return cache.get(product)
    else:
        return search_web_for(product)

def search_web_for(product):
    # Actual web-searching code goes here. Since we're not
    # actually going to search the web, we'll just use mock
    # data about the price and online shop:
    data_from_web = [799, "Jupiter Electronics"]
    # To mimic the time it takes to search the web, we'll pause for
    # one quarter of a second:
    time.sleep(0.25)
    # Cache the retrieved data:
    cache[product] = data_from_web

    return data_from_web
```

Here, the main function is lowest_price, which tells the app to first check the cache to see if it already contains data for that product. Only if the cache does *not* contain that data does the app fetch the data from the web.

In the search_web_for function, the app mimics searching the Internet by sleeping for a quarter of a second. In addition, the code uses the mock data [799, "Jupiter Electronics"] to indicate the product's lowest price and the online shop where the product is sold for that price. In this example, the product costs $799 and can be found at that price at a store called Jupiter Electronics.

Now, suppose that our software searches for products using the following commands:

```
print(lowest_price("Vroom-Master 3000"))
print(lowest_price("Dustpan Deluxe"))
print(lowest_price("Vroom-Master 3000"))
```

• Click HERE to purchase this book now. discuss

```
print(lowest_price("Vroom-Master 3000"))
print(lowest_price("Dustpan Deluxe"))
print(lowest_price("Vroom-Master 3000"))
```

The first time we search for `"Vroom-Master 3000"` and `"Dustpan Deluxe"`, it'll take a quarter of a second to obtain the data for each. However, in all subsequent requests we'll pull the data from the cache, enabling these requests to occur much more quickly.

Out of Space

Now that you know how awesome caching is, I could end the chapter here. However, there's one itty-bitty teeny-tiny little problem. If our cache keeps saving more and more information from the outside world, our computer (or tablet or smartphone) is going to run out of space really quickly. After all, we can't expect our cache to store the entire Internet!

To manage these space constraints, we need to ensure that our cache is not storing *all* data we've retrieved from the web. To do this, at some point we'll have to remove some data from our cache—or at least prevent new data from entering. This, then, is the tricky part of caching. Somehow we need to figure out what information we want to keep, and what information we can get rid of.

Eviction Policies

Here's a bit of cache jargon I'm going to use going forward. The common term for deleting something from the cache is to *evict* it. (It sounds harsh, I know.) Similarly, an *eviction policy* is an algorithm that decides what data we should evict from the cache.

Computer scientists have proposed a whole slew of different eviction policies over the years. However, the trick is to find the most *efficient* eviction policy for our cache. To define what it means for an eviction policy to be efficient, let me first introduce a few more caching terms.

Recall that each time an application makes a request for data, it first checks to see if the data is in the cache. If it is, this is called a *cache hit*. Cache hits are good, since it means that the app can grab the data from the cache rather than fetch the data from an external source. On the other hand, if when an app makes a request the data is *not* in the cache, this is known as a *cache miss*. Whenever there is a cache miss, the app has no choice but to look for the data on the Internet, or wherever the external data source is.

• Click HERE to purchase this book now. discuss

We can use these terms to help define what it means to have an efficient eviction policy. That is, an efficient eviction policy works to *increase cache hits and decrease cache misses*. (We'll see soon that we accomplish this by trying to only store data that we'll need again in the future.) The more requests that our app can fulfill by getting data from the cache, the less time our app needs to spend searching the web.

In sum, the more efficient our cache is, the faster our software will run.

Farthest-in-Future Eviction Policy

Let's think about how we might design an efficient eviction policy.

Ideally, we'd evict data that will never be requested ever again. After all, the whole point of caching is to quickly deliver data on the second and third time it's requested. So if there will never be another future request for a particular item, we can safely evict its data. This enables us to save room for data that we *will* request again at some point. (How will we know what data will or won't be requested again? Good question, but hold onto it for now.)

Now, even if all the products in our sample app will be requested again at some point, we can still make our eviction policy more efficient by evicting data that will be requested *farthest into the future*. Let's see an example of what I mean.

Here's a cache that stores up to four items:

Right now, the cache is empty, which, incidentally, is called a *cold cache*. (I like calling it cold *hard* cache, but no one else calls it that.)

To keep the diagrams nice and small, instead of requesting items such as a "Vroom-Master," we'll just be requesting integers. So imagine in your mind's eye that each integer represents some particular physical product.

Here's an example sequence of requests we'll be making, from left to right:

3, 5, 2, 1, 6, 5, 4, 3, 6, 2, 4, 1

Let's begin to make our requests. First, we request the 3. There's plenty of room in the cache, so we cache the 3:

• Click HERE to purchase this book now. discuss

3, 5, 2, 1, 6, 5, 4, 3, 6, 2, 4, 1
↑

3			

We do the same for the next three requests, filling up the cache:

3, 5, 2, 1, 6, 5, 4, 3, 6, 2, 4, 1
↑ (at 5)

3	**5**		

3, 5, 2, 1, 6, 5, 4, 3, 6, 2, 4, 1
↑ (at 2)

3	5	**2**	

3, 5, 2, 1, 6, 5, 4, 3, 6, 2, 4, 1
↑ (at 1)

3	5	2	**1**

Next, we request the 6. The 6 is not currently in the cache, so this is a cache miss. Actually, the previous requests were also all cache misses, but this is the first cache miss we encounter where the cache is full.

Because our cache is full, we need to evict one of the cache's current items if we're going to cache the 6. And so, we look to our eviction policy to decide which item we'll evict.

If we look ahead to our future requests (those to the right of the 6), we'll see that of all the items currently in our cache, the 1 is the one we'll be requesting farthest into the future. That is, the 3, 5, and 2 will all be requested sooner than the 1. This is what I mean, then, by the term *farthest-in-future* eviction policy; we evict the whichever item will be requested later than all other items in the cache.

In this case, then, we'll evict the 1 and replace it with the 6:

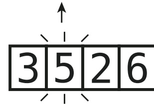
3, 5, 2, 1, 6, 5, 4, 3, 6, 2, 4, 1
↑ (at 6)

3	5	2	**6**

replaced 1

Next up, a 5 is requested. We happen to have a 5 in the cache, so we have our first cache hit!

3, 5, 2, 1, 6, 5, 4, 3, 6, 2, 4, 1

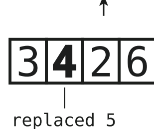


We just saved time from having to request the 5 from the web. Also, we don't have to evict anything, since we're not inserting any new data into the cache.

Next, a 4 is requested. This is a cache miss, but what item to evict?

The 3, 2, and 6 will all be requested again soon, but there won't be a request for 5 again in the near future. Perhaps there will be in a future batch of requests, but there aren't any 5's in the current list of requests. So let's evict the 5 and cache the 4:

3, 5, 2, 1, 6, 5, 4, 3, 6, 2, 4, 1

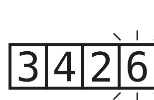


Next, we request a 3. That's a cache hit:

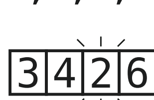
3, 5, 2, 1, 6, 5, 4, 3, 6, 2, 4, 1

We also have cache hits with the next three requests:

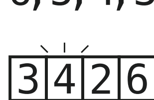
3, 5, 2, 1, 6, 5, 4, 3, 6, 2, 4, 1



3, 5, 2, 1, 6, 5, 4, 3, 6, 2, 4, 1



3, 5, 2, 1, 6, 5, 4, 3, 6, 2, 4, 1



The final request in this sequence is a 1. That's a cache miss. As to which item to evict, it's hard to say, since we're not yet privy to whatever requests may come next.

In any case, the above walk-through shows what it means to evict items that will be requested farthest into the future. But now let me explain why this eviction policy is ideal. To some, this may already be intuitive, but I'll spell it out anyway.

If we evict items that are about to be requested again soon, we'll certainly cause cache misses that could have been avoided. Now, one might argue that perhaps there might be a case where it's worthwhile keeping a farthest-in-future item if that item will be requested many times down the line. However, this not a valid concern, as the next time that item is requested, it'll be cached at that point and available for all those subsequent requests. By keeping it around in the meantime, we're taking away space from items that are being requested sooner and causing cache misses.