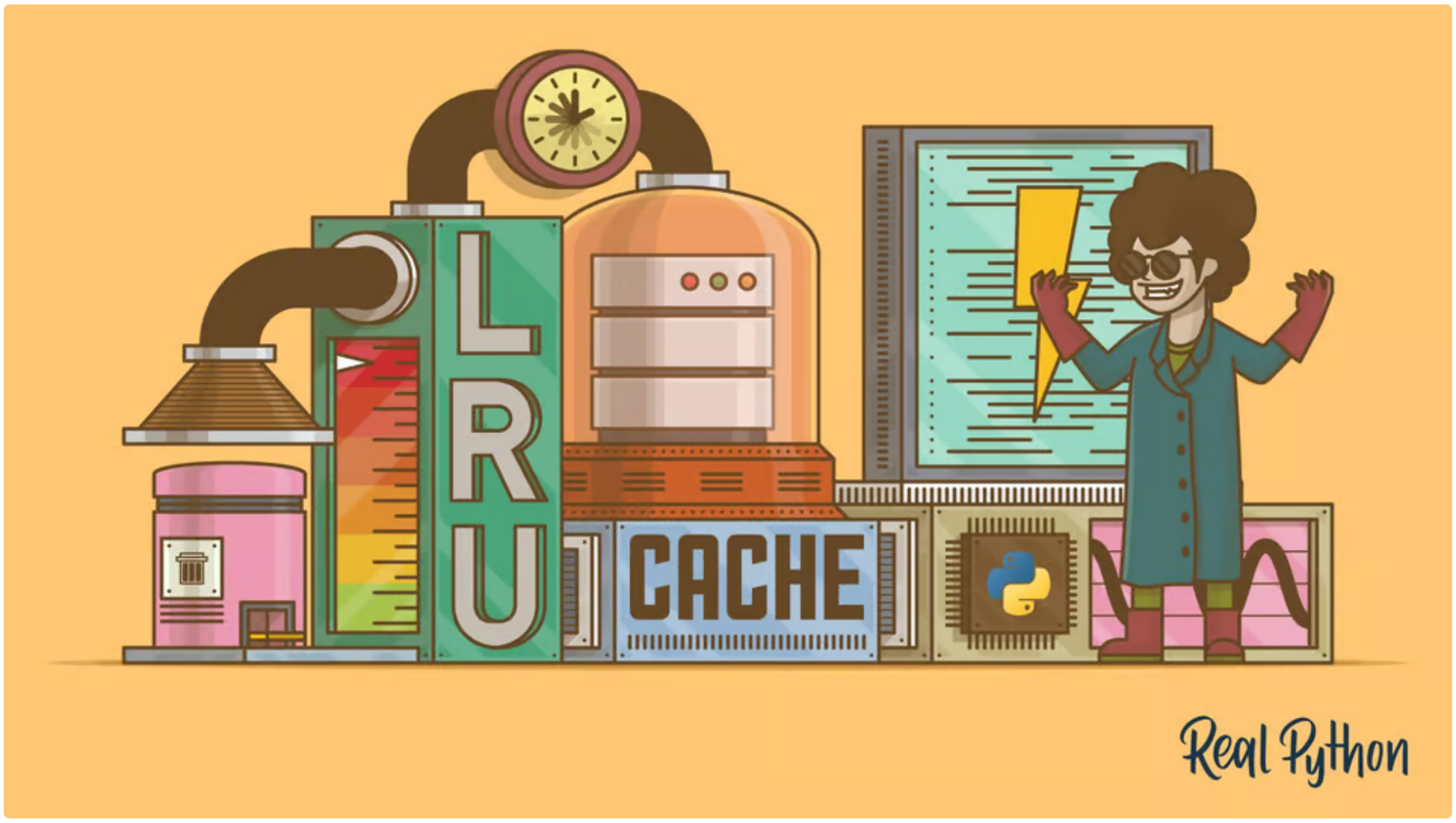


Caching in Python Using the LRU Cache Strategy

by Santiago Valdarrama Nov 04, 2020 10 Comments intermediate python

Mark as Completed

Share Share Email

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the w tutorial to deepen your understanding: **Caching in Python With lru_cache**

Help

There are many ways to achieve fast and responsive applications. **Caching** is one approach that, when used correctly, makes things much faster while decreasing the load on computing resources. Python's `functools` module comes with the `@lru_cache` decorator, which gives you the ability to cache the result of your functions using the **Least Recently Used (LRU) strategy**. This is a simple yet powerful technique that you can use to leverage the power of caching in your code.

In this tutorial, you'll learn:

- What **caching strategies** are available and how to implement them using **Python decorators**
- What the **LRU strategy** is and how it works
- How to improve performance by caching with the **`@lru_cache` decorator**
- How to expand the functionality of the `@lru_cache` decorator and make it **expire** after a specific time

By the end of this tutorial, you'll have a deeper understanding of how caching works and how to take advantage of it in Python.

Caching and Its Uses

Caching is an optimization technique that you can use in your applications to keep recent or often-used data in memory locations that are faster or computationally cheaper to access than their source.

Imagine you're building a newsreader application that fetches the latest news from different sources. As the user navigates through the list, your application downloads the articles and displays them on the screen.

What would happen if the user decided to move repeatedly back and forth between a couple of news articles? Unless you were caching the data, your application would have to fetch the same content every time! That would make your user's system sluggish and put extra pressure on the server hosting the articles.

A better approach would be to store the content locally after fetching each article. Then, the next time the user decided to open an article, your application could open the content from a locally stored copy instead of going back to the source. In computer science, this technique is called **caching**.

Implementing a Cache Using a Python Dictionary

You can implement a caching solution in Python using a dictionary.

Staying with the newsreader example, instead of going directly to the server every time you need to download an article, you can check whether you have the content in your cache and go back to the server only if you don't. You can use the article's URL as the key and its content as the value.

Here's an example of how this caching technique might look:

Python

```python
import requests

cache = dict()

def get_article_from_server(url):
    print("Fetching article from server...")
    response = requests.get(url)
    return response.text

def get_article(url):
    print("Getting article...")
    if url not in cache:
        cache[url] = get_article_from_server(url)

    return cache[url]

get_article("https://realpython.com/sorting-algorithms-python/")
get_article("https://realpython.com/sorting-algorithms-python/")
```

Save this code to a `caching.py` file, install the `requests` library, then run the script:

Shell

```
$ pip install requests
$ python caching.py
Getting article...
Fetching article from server...
Getting article...
```

Notice how you get the string `"Fetching article from server..."` printed a single time despite calling `get_article()` twice, in lines 17 and 18. This happens because, after accessing the article for the first time, you put its URL and content in the `cache` dictionary. The second time, the code doesn't need to fetch the item from the server again.

Caching Strategies

There's one big problem with this cache implementation: the content of the dictionary will grow indefinitely! As the user downloads more articles, the application will keep storing them in memory, eventually causing the application to crash.

To work around this issue, you need a strategy to decide which articles should stay in memory and which should be removed. These caching strategies are algorithms that focus on managing the cached information and choosing which items to discard to make room for new ones.

There are several different strategies that you can use to evict items from the cache and keep it from growing past from its maximum size. Here are five of the most popular ones, with an explanation of when each is most useful:

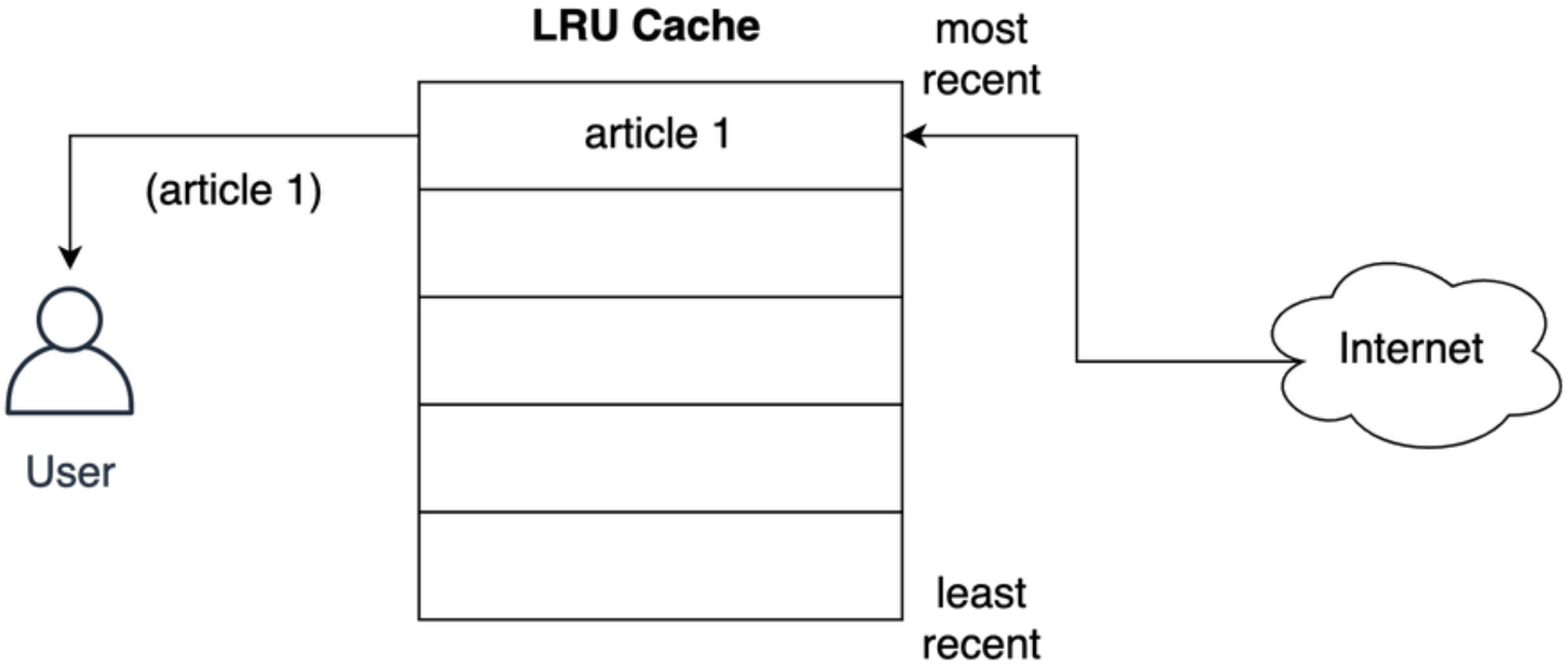
Strategy	Eviction policy	Use case
First-In/First-Out (FIFO)	Evicts the oldest of the entries	Newer entries are most likely to be reused
Last-In/First-Out (LIFO)	Evicts the latest of the entries	Older entries are most likely to be reused
Least Recently Used (LRU)	Evicts the least recently used entry	Recently used entries are most likely to be reused
Most Recently Used (MRU)	Evicts the most recently used entry	Least recently used entries are most likely to be reused
Least Frequently Used (LFU)	Evicts the least often accessed entry	Entries with a lot of hits are more likely to be reused

In the sections below, you'll take a closer look at the LRU strategy and how to implement it using the `@lru_cache` decorator from Python's `functools` module.

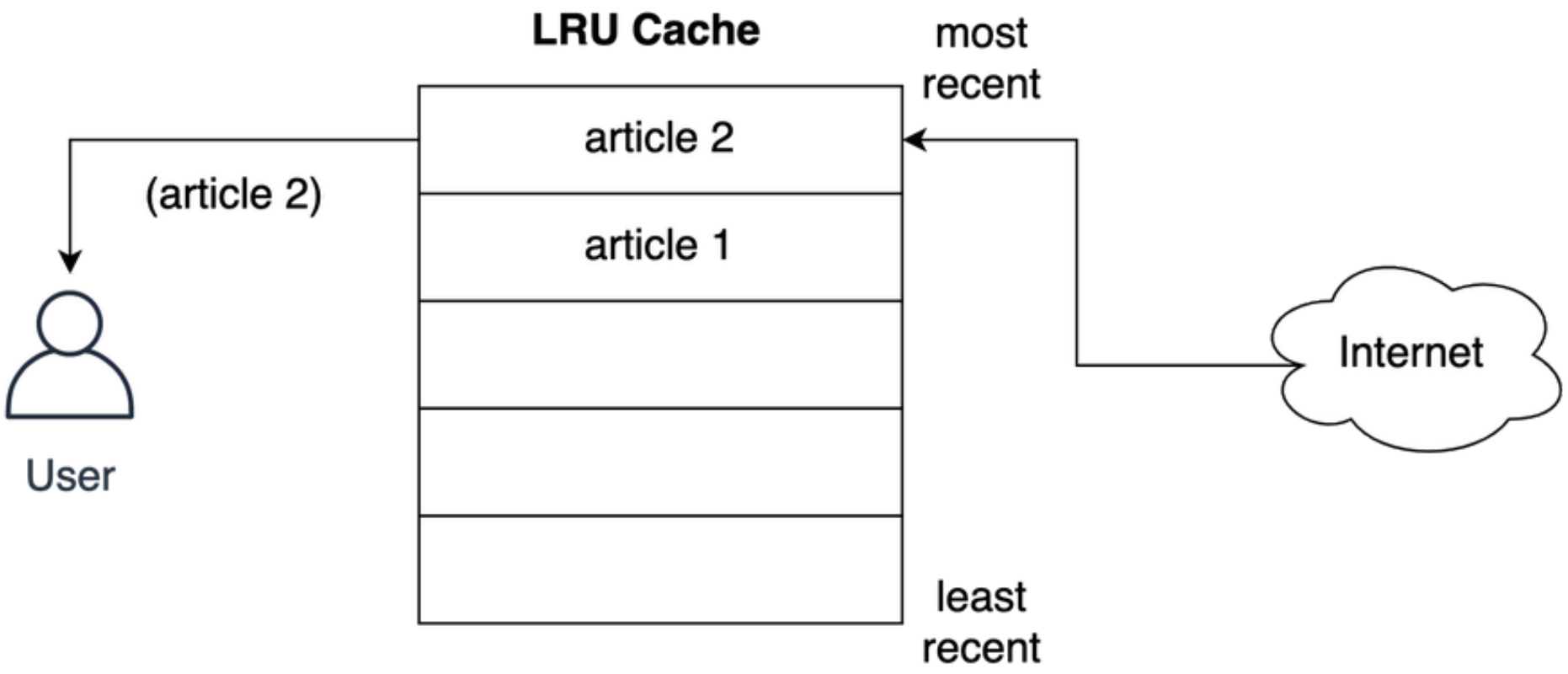
Diving Into the Least Recently Used (LRU) Cache Strategy

A cache implemented using the LRU strategy organizes its items in order of use. Every time you access an entry, the LRU algorithm will move it to the top of the cache. This way, the algorithm can quickly identify the entry that's gone unused the longest by looking at the bottom of the list.

The following figure shows a hypothetical cache representation after your user requests an article from the network:



Notice how the cache stores the article in the most recent slot before serving it to the user. The following figure shows what happens when the user requests a second article:



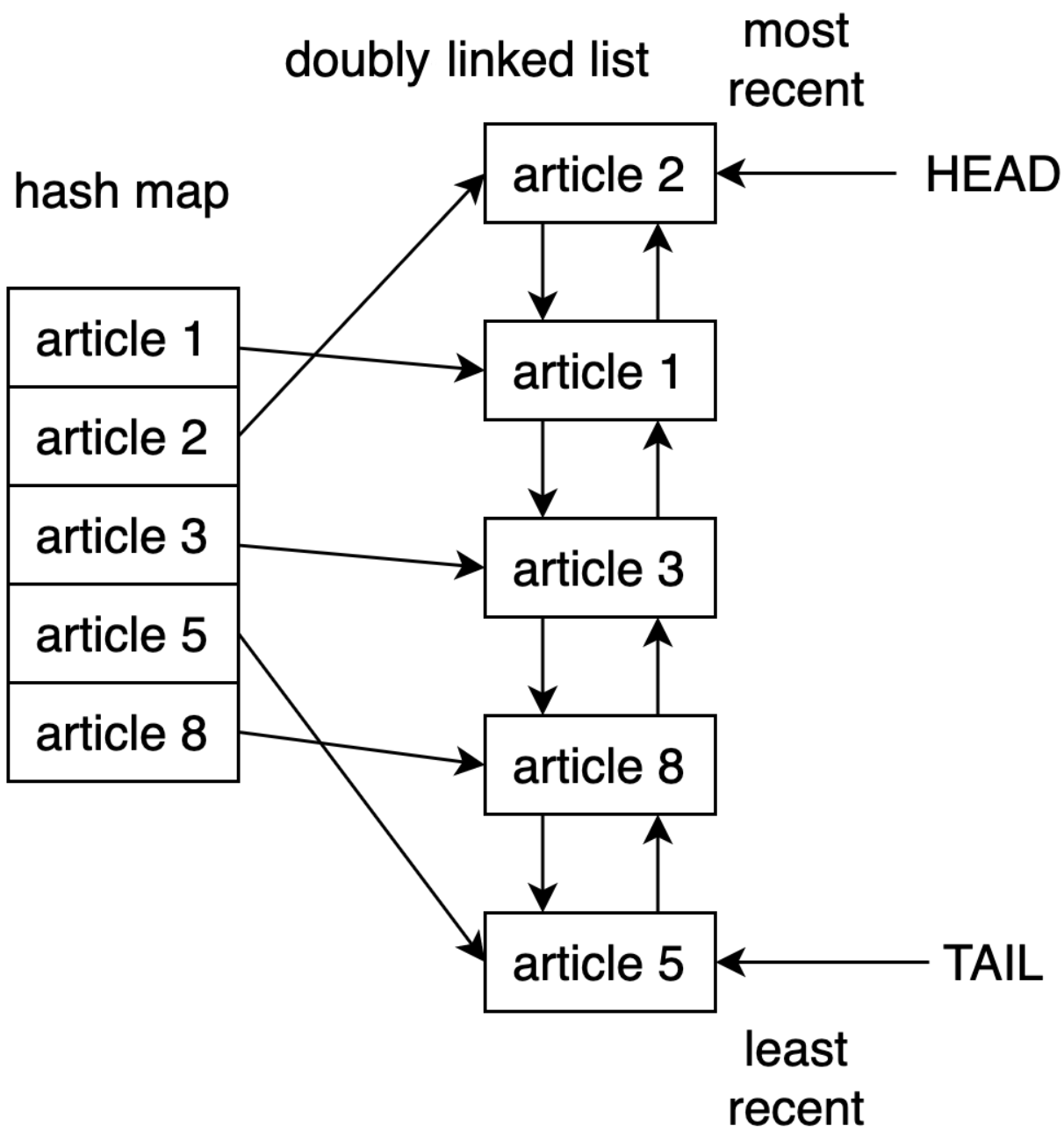
The second article takes the most recent slot, pushing the first article down the list.

The LRU strategy assumes that the more recently an object has been used, the more likely it will be needed in the future, so it tries to keep that object in the cache for the longest time.

Peeking Behind the Scenes of the LRU Cache

One way to implement an LRU cache in Python is to use a combination of a doubly linked list and a hash map. The **head element** of the doubly linked list would point to the most recently used entry, and the **tail** would point to the least recently used entry.

The figure below shows the potential structure of the LRU cache implementation behind the scenes:



Using the hash map, you can ensure access to every item in the cache by mapping each entry to the specific location in the doubly linked list.

This strategy is very fast. Accessing the least recently used item and updating the cache are operations with a runtime of $O(1)$.

> **Note:** For a deeper understanding of Big O notation, together with several practical examples in Python, check out Big O Notation and Algorithm Analysis with Python Examples.

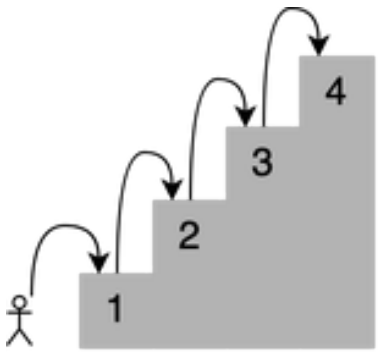
Since version 3.2, Python has included the `@lru_cache` decorator for implementing the LRU strategy. You can use this decorator to wrap functions and cache their results up to a maximum number of entries.

Using @lru_cache to Implement an LRU Cache in Python

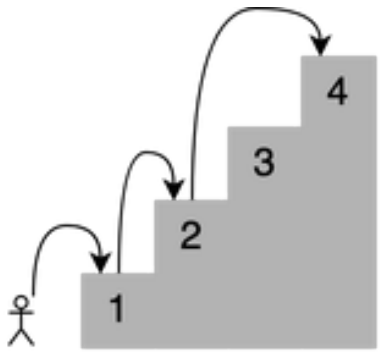
Just like the caching solution you implemented earlier, `@lru_cache` uses a dictionary behind the scenes. It caches the function's result under a key that consists of the call to the function, including the supplied arguments. This is important because it means that these arguments have to be **hashable** for the decorator to work.

Playing With Stairs

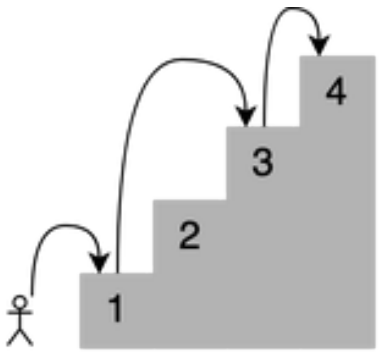
Imagine you want to determine all the different ways you can reach a specific stair in a staircase by hopping one, two, or three stairs at a time. How many paths are there to the fourth stair? Here are all the different combinations:



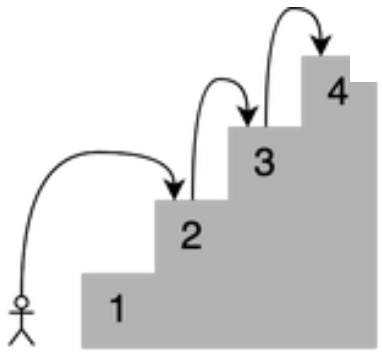
1 → 1 → 1 → 1



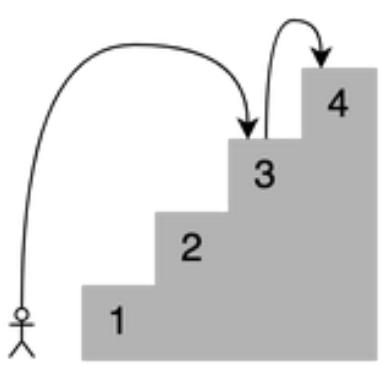
1 → 1 → 2



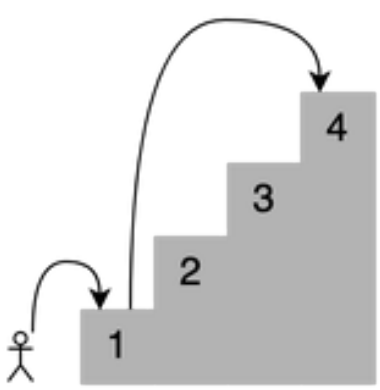
1 → 2 → 1



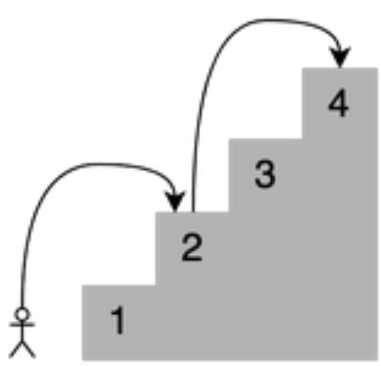
2 → 1 → 1



3 → 1



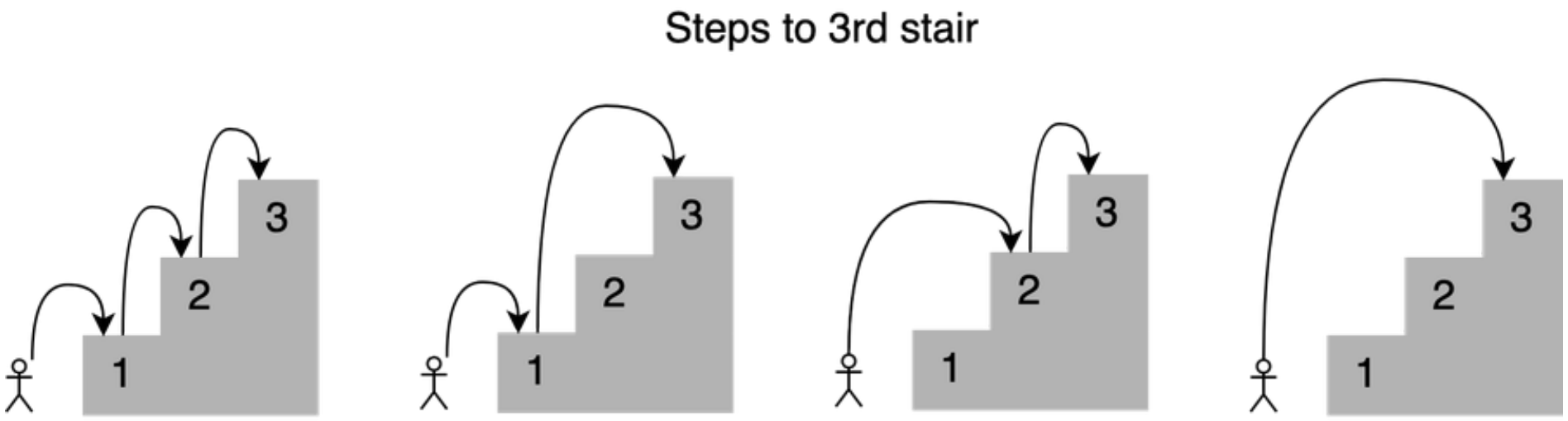
1 → 3

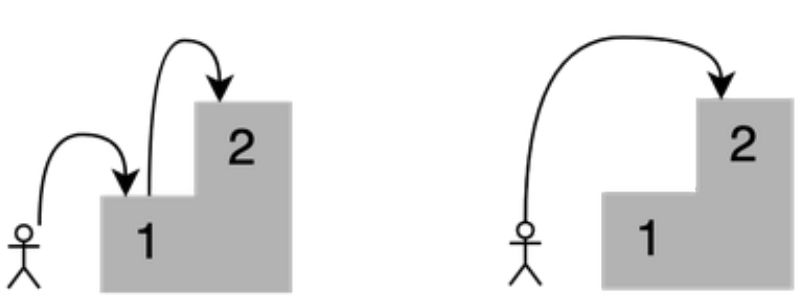


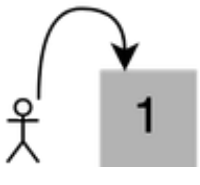
2 → 2

You could frame a solution to this problem by stating that, to reach your current stair, you can jump from one stair, two stairs, or three stairs below. Adding up the number of jump combinations you can use to get to each of those points should give you the total number of possible ways to reach your current position.

For example, the number of combinations for reaching the fourth stair will equal the total number of different ways you can reach the third, second, and first stair:







As shown in the picture, there are seven different ways to reach the fourth stair. Notice how the solution for a given stair builds upon the answers to smaller subproblems. In this case, to determine the different paths to the fourth stair, you can add up the four ways of reaching the third stair, the two ways of reaching the second stair, and the one way of reaching the first stair.

This approach is called **recursion**. If you want to learn more, then check out Recursion in Python: An Introduction for an introduction to the topic.

Here's a function that implements this recursion:

Python

```python
def steps_to(stair):
    if stair == 1:
        # You can reach the first stair with only a single step
        # from the floor.
        return 1
    elif stair == 2:
        # You can reach the second stair by jumping from the
        # floor with a single two-stair hop or by jumping a single
        # stair a couple of times.
        return 2
    elif stair == 3:
        # You can reach the third stair using four possible
        # combinations:
        # 1. Jumping all the way from the floor
        # 2. Jumping two stairs, then one
        # 3. Jumping one stair, then two
        # 4. Jumping one stair three times
        return 4
    else:
        # You can reach your current stair from three different places:
        # 1. From three stairs down
        # 2. From two stairs down
        # 2. From one stair down
        #
        # If you add up the number of ways of getting to those
        # those three positions, then you should have your solution.
        return (
            steps_to(stair - 3)
            + steps_to(stair - 2)
            + steps_to(stair - 1)
        )

print(steps_to(4))
```

Save this code into a file named `stairs.py` and run it with the following command:

Shell

```
$ python stairs.py
7
```

Great! The code works for `4` stairs, but how about counting how many steps to reach a higher place in the staircase? Change the stair number in line 33 to `30` and rerun the script:

Shell

```
$ python stairs.py
53798080
```

Wow, more than 53 million combinations! That's a lot of hops!

Timing Your Code

When finding the solution for the thirtieth stair, the script took quite a bit of time to finish. To get a baseline, you can measure how long it takes for the code to run.

To accomplish this, you can use Python's `timeit` module. Add the following lines after line 33:

Python

```
35 setup_code = "from __main__ import steps_to"
36 stmt = "steps_to(30)"
37 times = repeat(setup=setup_code, stmt=stmt, repeat=3, number=10)
38 print(f"Minimum execution time: {min(times)}")
```

You also need to import the `timeit` module at the top of the code:

Python

```
1 from timeit import repeat
```

Here's a line-by-line explanation of these additions:

- **Line 35** imports the name of `steps_to()` so that `timeit.repeat()` knows how to call it.
- **Line 36** prepares the call to the function with the stair number you want to reach, which in this case is `30`. This is the statement that will be executed and timed.
- **Line 37** calls `timeit.repeat()` with the setup code and the statement. This will call the function `10` times, returning the number of seconds each execution took.
- **Line 38** identifies and prints the shortest time returned.

> **Note:** A common misconception is that you should find the average time of each run of the function instead of selecting the shortest time.
>
> Time measurements are noisy because the system runs other processes concurrently. The shortest time is always the least noisy, making it the best representation of the function's runtime.

Now run the script again:

Shell

```
$ python stairs.py
53798080
Minimum execution time: 40.014977024000004
```

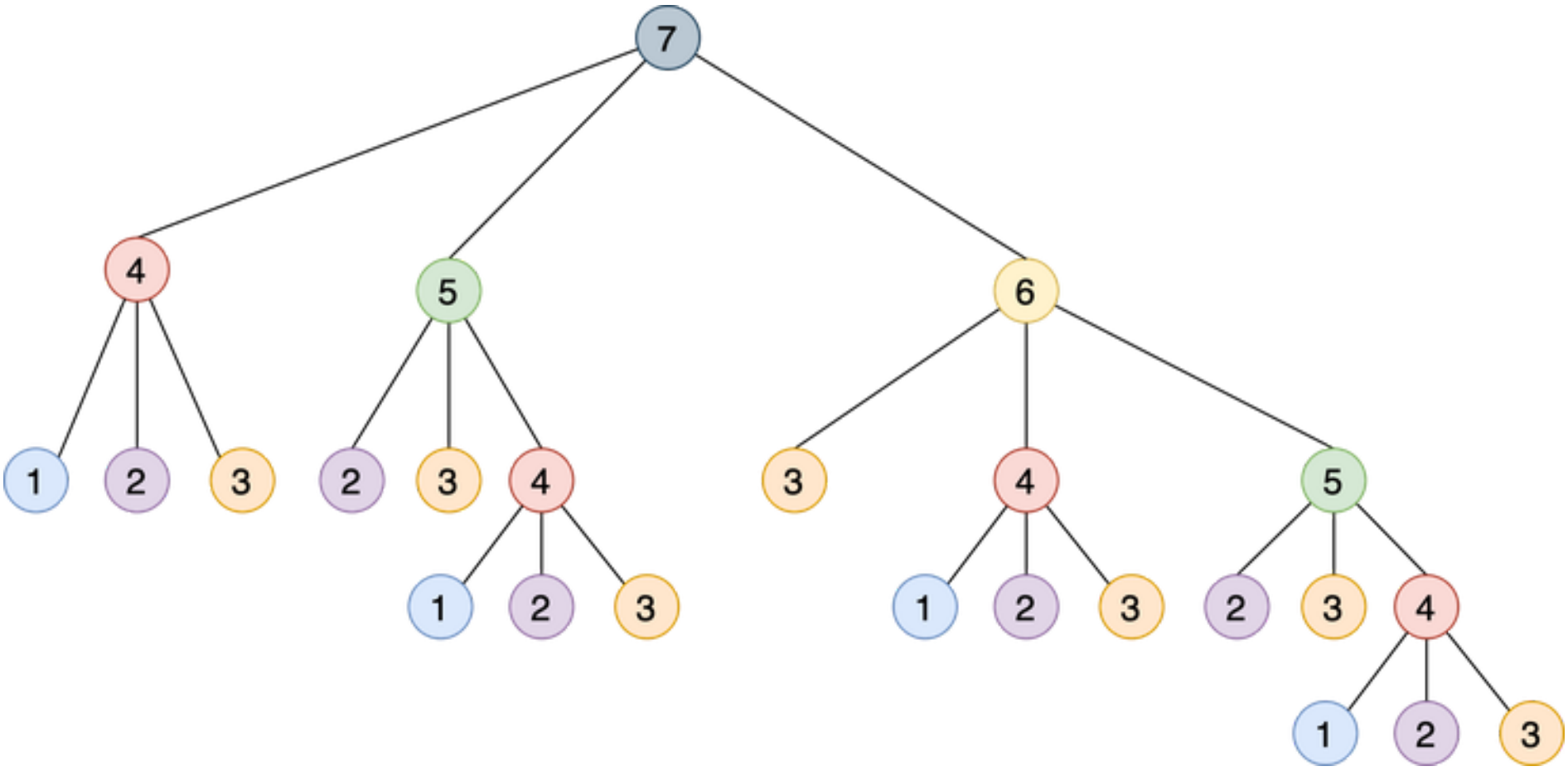
The number of seconds you'll see depends on your specific hardware. On my system, the script took forty seconds, which is quite slow for just thirty stairs!

> **Note:** You can learn more about the `timeit` module in the official Python documentation.

A solution that takes this long is a problem, but you can improve it using memoization.

Using Memoization to Improve the Solution

This recursive implementation solves the problem by breaking it into smaller steps that build upon each other. The following figure shows a tree in which every node represents a specific call to `steps_to()`:



Notice how you need to call `steps_to()` with the same argument multiple times. For example, `steps_to(5)` is computed two times, `steps_to(4)` is computed four times, `steps_to(3)` seven times, and `steps_to(2)` six times. Calling the same function more than once adds computation cycles that aren't necessary—the result will always be the same.

To fix this problem, you can use a technique called **memoization**. This approach ensures that a function doesn't run for the same inputs more than once by storing its result in memory and then referencing it later when necessary. This scenario sounds like the perfect opportunity to use Python's `@lru_cache` decorator!

> **Note:** For more information on **memoization** and using `@lru_cache` to implement it, check out Memoization in Python.

With just two changes, you can considerably improve the algorithm's runtime:

1. Import the `@lru_cache` decorator from the `functools` module.
2. Use `@lru_cache` to decorate `steps_to()`.

Here's what the top of the script will look like with the two updates:

Python

```python
from functools import lru_cache
from timeit import repeat

@lru_cache
def steps_to(stair):
    if stair == 1:
```

Running the updated script produces the following result:

Shell

```
$ python stairs.py
53798080
Minimum execution time: 7.999999999987184e-07
```

Caching the result of the function takes the runtime from 40 seconds down to 0.0008 milliseconds! That's a fantastic improvement!

> **Note:** In Python 3.8 and above, you can use the `@lru_cache` decorator without parentheses if you're not specifying any parameters. In previous versions, you may need to include the parentheses: `@lru_cache()`.

Remember, behind the scenes, the `@lru_cache` decorator stores the result of `steps_to()` for each different input. Every time the code calls the function with the same parameters, instead of computing an answer all over again, it returns the correct result directly from memory. This explains the massive improvement in performance when using `@lru_cache`.

Unpacking the Functionality of @lru_cache

With the `@lru_cache` decorator in place, you store every call and answer in memory to access later if requested again. But how many calls can you save before running out of memory?

Python's `@lru_cache` decorator offers a `maxsize` attribute that defines the maximum number of entries before the cache starts evicting old items. By default, `maxsize` is set to `128`. If you set `maxsize` to `None`, then the cache will grow indefinitely, and no entries will be ever evicted. This could become a problem if you're storing a large number of different calls in memory.

Here's an example of `@lru_cache` using the `maxsize` attribute:

Python

```
from functools import lru_cache
from timeit import repeat

@lru_cache(maxsize=16)
def steps_to(stair):
    if stair == 1:
```

In this case, you're limiting the cache to a maximum of `16` entries. When a new call comes in, the decorator's implementation will evict the least recently used of the existing `16` entries to make a place for the new item.

To see what happens with this new addition to the code, you can use `cache_info()`, provided by the `@lru_cache` decorator, to inspect the number of **hits** and **misses** and the current size of the cache. For clarity, remove the code that times the runtime of the function. Here's how the final script looks after all the modifications:

Python

```python
from functools import lru_cache
from timeit import repeat

@lru_cache(maxsize=16)
def steps_to(stair):
    if stair == 1:
        # You can reach the first stair with only a single step
        # from the floor.
        return 1
    elif stair == 2:
        # You can reach the second stair by jumping from the
        # floor with a single two-stair hop or by jumping a single
        # stair a couple of times.
        return 2
    elif stair == 3:
        # You can reach the third stair using four possible
        # combinations:
        # 1. Jumping all the way from the floor
        # 2. Jumping two stairs, then one
        # 3. Jumping one stair, then two
        # 4. Jumping one stair three times
        return 4
    else:
        # You can reach your current stair from three different places:
        # 1. From three stairs down
        # 2. From two stairs down
        # 2. From one stair down
        #
        # If you add up the number of ways of getting to those
        # those three positions, then you should have your solution.
        return (
            steps_to(stair - 3)
            + steps_to(stair - 2)
            + steps_to(stair - 1)
        )

print(steps_to(30))

print(steps_to.cache_info())
```

If you call the script again, then you'll see the following result:

Shell

```
$ python stairs.py
53798080
CacheInfo(hits=52, misses=30, maxsize=16, currsize=16)
```

You can use the information returned by `cache_info()` to understand how the cache is performing and fine-tune it to find the appropriate balance between speed and storage.

Here's a breakdown of the properties provided by `cache_info()`:

- **`hits=52`** is the number of calls that `@lru_cache` returned directly from memory because they existed in the cache.
- **`misses=30`** is the number of calls that didn't come from memory and were computed. Since you're trying to find the number of steps to reach the thirtieth stair, it makes sense that each of these calls missed the cache the first time they were made.
- **`maxsize=16`** is the size of the cache as you defined it with the `maxsize` attribute of the decorator.
- **`currsize=16`** is the current size of the cache. In this case, it shows that your cache is full.

If you need to remove all the entries from the cache, then you can use `cache_clear()` provided by `@lru_cache`.

Adding Cache Expiration

Imagine you want to develop a script that monitors *Real Python* and prints the number of characters in any article that contains the word `python`.

Real Python provides an Atom feed, so you can use the `feedparser` library to parse the feed and the `requests` library to load the contents of the article as you did before.

Here's an implementation of the monitor script:

Python

```python
import feedparser
import requests
import ssl
import time

if hasattr(ssl, "_create_unverified_context"):
    ssl._create_default_https_context = ssl._create_unverified_context

def get_article_from_server(url):
    print("Fetching article from server...")
    response = requests.get(url)
    return response.text

def monitor(url):
    maxlen = 45
    while True:
        print("\nChecking feed...")
        feed = feedparser.parse(url)

        for entry in feed.entries[:5]:
            if "python" in entry.title.lower():
                truncated_title = (
                    entry.title[:maxlen] + "..."
                    if len(entry.title) > maxlen
                    else entry.title
                )
                print(
                    "Match found:",
                    truncated_title,
                    len(get_article_from_server(entry.link)),
                )

        time.sleep(5)

monitor("https://realpython.com/atom.xml")
```

Save this script to a file called `monitor.py`, install the `feedparser` and `requests` libraries, and run the script. It will run continuously until you stop it by pressing `^ Ctrl` + `C` in your terminal window:

Shell

```
$ pip install feedparser requests
$ python monitor.py

Checking feed...
Fetching article from server...
The Real Python Podcast – Episode #28: Using ... 29520
Fetching article from server...
Python Community Interview With David Amos 54256
Fetching article from server...
Working With Linked Lists in Python 37099
Fetching article from server...
Python Practice Problems: Get Ready for Your ... 164888
Fetching article from server...
The Real Python Podcast – Episode #27: Prepar... 30784

Checking feed...
Fetching article from server...
The Real Python Podcast – Episode #28: Using ... 29520
Fetching article from server...
Python Community Interview With David Amos 54256
Fetching article from server...
Working With Linked Lists in Python 37099
Fetching article from server...
Python Practice Problems: Get Ready for Your ... 164888
Fetching article from server...
The Real Python Podcast – Episode #27: Prepar... 30784
```

Here's a step-by-step explanation of the code:

- **Lines 6 and 7**: This is a workaround to an issue when `feedparser` tries to access content served over HTTPS. See the note below for more information.
- **Line 16**: `monitor()` will loop indefinitely.
- **Line 18**: Using `feedparser`, the code loads and parses the feed from *Real Python*.
- **Line 20**: The loop goes through the first `5` entries on the list.
- **Lines 21 to 31**: If the word `python` is part of the title, then the code prints it along with the length of the article.
- **Line 33**: The code sleeps for `5` seconds before continuing.
- **Line 35**: This line kicks off the monitoring process by passing the URL of the *Real Python* feed to `monitor()`.

Every time the script loads an article, the message `"Fetching article from server..."` is printed to the console. If you let the script run long enough, then you'll see how this message shows up repeatedly, even when loading the same link.

> **Note:** For more information about the issue with `feedparser` accessing content served over HTTPS, check out issue 84 on the `feedparser` repository. PEP 476 describes how Python started enabling certificate verification by default for `stdlib` HTTP clients, which is the underlying cause of this error.

This is an excellent opportunity to cache the article's contents and avoid hitting the network every five seconds. You could use the `@lru_cache` decorator, but what happens if the article's content is updated?

The first time you access the article, the decorator will store its content and return the same data every time after. If the post is updated, then the monitor script will never realize it because it will be pulling the old copy stored in the cache. To solve this problem, you can set your cache entries to expire.

Evicting Cache Entries Based on Both Time and Space

The `@lru_cache` decorator evicts existing entries only when there's no more space to store new listings. With sufficient space, entries in the cache will live forever and never get refreshed.

This presents a problem for your monitoring script because you'll never fetch updates published for previously cached articles. To get around this problem, you can update the cache implementation so it expires after a specific time.

You can implement this idea into a new decorator that extends `@lru_cache`. If the caller tries to access an item that's past its lifetime, then the cache won't return its content, forcing the caller to fetch the article from the network.

> **Note:** For more information about Python decorators, check Primer on Python Decorators and Python Decorators 101.

Here's a possible implementation of this new decorator:

Python

```
1  from functools import lru_cache, wraps
2  from datetime import datetime, timedelta
3
4  def timed_lru_cache(seconds: int, maxsize: int = 128):
5      def wrapper_cache(func):
6          func = lru_cache(maxsize=maxsize)(func)
7          func.lifetime = timedelta(seconds=seconds)
8          func.expiration = datetime.utcnow() + func.lifetime
9
10         @wraps(func)
11         def wrapped_func(*args, **kwargs):
12             if datetime.utcnow() >= func.expiration:
13                 func.cache_clear()
14                 func.expiration = datetime.utcnow() + func.lifetime
15
16             return func(*args, **kwargs)
17
18         return wrapped_func
19
20     return wrapper_cache
```

Here's a breakdown of this implementation:

- **Line 4**: The `@timed_lru_cache` decorator will support the lifetime of the entries in the cache (in seconds) and the maximum size of the cache.
- **Line 6**: The code wraps the decorated function with the `lru_cache` decorator. This allows you to use the cache functionality already provided by `lru_cache`.
- **Lines 7 and 8**: These two lines instrument the decorated function with two attributes representing the lifetime of the cache and the actual date when it will expire.
- **Lines 12 to 14**: Before accessing an entry in the cache, the decorator checks whether the current date is past the expiration date. If that's the case, then it clears the cache and recomputes the lifetime and expiration date.

Notice how, when an entry is expired, this decorator clears the entire cache associated with the function. The lifetime applies to the cache as a whole, not to individual articles. A more sophisticated implementation of this strategy would evict entries based on their individual lifetimes.

Caching Articles With the New Decorator

You can now use your new `@timed_lru_cache` decorator with the `monitor` script to prevent fetching the content of an article every time you access it.

Putting the code together in a single script for simplicity, you end up with the following:

Python

```python
import feedparser
import requests
import ssl
import time

from functools import lru_cache, wraps
from datetime import datetime, timedelta

if hasattr(ssl, "_create_unverified_context"):
    ssl._create_default_https_context = ssl._create_unverified_context

def timed_lru_cache(seconds: int, maxsize: int = 128):
    def wrapper_cache(func):
        func = lru_cache(maxsize=maxsize)(func)
        func.lifetime = timedelta(seconds=seconds)
        func.expiration = datetime.utcnow() + func.lifetime

        @wraps(func)
        def wrapped_func(*args, **kwargs):
            if datetime.utcnow() >= func.expiration:
                func.cache_clear()
                func.expiration = datetime.utcnow() + func.lifetime

            return func(*args, **kwargs)

        return wrapped_func

    return wrapper_cache

@timed_lru_cache(10)
def get_article_from_server(url):
    print("Fetching article from server...")
    response = requests.get(url)
    return response.text

def monitor(url):
    maxlen = 45
    while True:
        print("\nChecking feed...")
        feed = feedparser.parse(url)

        for entry in feed.entries[:5]:
            if "python" in entry.title.lower():
                truncated_title = (
                    entry.title[:maxlen] + "..."
                    if len(entry.title) > maxlen
                    else entry.title
                )
                print(
                    "Match found:",
                    truncated_title,
                    len(get_article_from_server(entry.link)),
                )

        time.sleep(5)

monitor("https://realpython.com/atom.xml")
```

Notice how line 30 decorates `get_article_from_server()` with the `@timed_lru_cache` and specifies a validity of `10` seconds. Any attempt to access the same article from the server within `10` seconds of having fetched it will return the contents from the cache and never hit the network.

Run the script and take a look at the results:

Shell

```
$ python monitor.py

Checking feed...
Fetching article from server...
Match found: The Real Python Podcast – Episode #28: Using ... 29521
Fetching article from server...
Match found: Python Community Interview With David Amos 54254
Fetching article from server...
Match found: Working With Linked Lists in Python 37100
Fetching article from server...
Match found: Python Practice Problems: Get Ready for Your ... 164887
Fetching article from server...
Match found: The Real Python Podcast – Episode #27: Prepar... 30783

Checking feed...
Match found: The Real Python Podcast – Episode #28: Using ... 29521
Match found: Python Community Interview With David Amos 54254
Match found: Working With Linked Lists in Python 37100
Match found: Python Practice Problems: Get Ready for Your ... 164887
Match found: The Real Python Podcast – Episode #27: Prepar... 30783

Checking feed...
Match found: The Real Python Podcast – Episode #28: Using ... 29521
Match found: Python Community Interview With David Amos 54254
Match found: Working With Linked Lists in Python 37100
Match found: Python Practice Problems: Get Ready for Your ... 164887
Match found: The Real Python Podcast – Episode #27: Prepar... 30783

Checking feed...
Fetching article from server...
Match found: The Real Python Podcast – Episode #28: Using ... 29521
Fetching article from server...
Match found: Python Community Interview With David Amos 54254
Fetching article from server...
Match found: Working With Linked Lists in Python 37099
Fetching article from server...
Match found: Python Practice Problems: Get Ready for Your ... 164888
Fetching article from server...
Match found: The Real Python Podcast – Episode #27: Prepar... 30783
```

Notice how the code prints the message `"Fetching article from server..."` the first time it accesses the matching articles. After that, depending on your network speed and computing power, the script will retrieve the articles from the cache either one or two times before hitting the server again.

The script tries to access the articles every `5` seconds, and the cache expires every `10` seconds. These times are probably too short for a real application, so you can get a significant improvement by adjusting these configurations.

Conclusion

Caching is an essential optimization technique for improving the performance of any software system. Understanding how caching works is a fundamental step toward incorporating it effectively in your applications.

In this tutorial, you learned:

- What the different **caching strategies** are and how they work
- How to use Python's **`@lru_cache`** decorator
- How to **create a new decorator** to extend the functionality of `@lru_cache`
- How to measure your code's **runtime** using the `timeit` module
- What **recursion** is and how to solve a problem using it
- How **memoization** improves runtime by storing intermediate results in memory

The next step to implementing different caching strategies in your applications is looking at the `cachetools` module. This library provides several collections and decorators covering some of the most popular caching strategies that you can start using right away.