
redis

Using Redis In-Memory Storage for your Python Applications

Todd | NoSQL | 11 Min Read | February 05

It's been a hell of a decade for software development. I've personally enjoyed observing endless moaning and complaining from fellow developers about the topic of "new frameworks." Standard rhetoric would have one think our careers are somehow under siege as a result of learning new technologies. I can't help but wonder how the same contingency of professionals has reacted to the rise of cloud services. To those who fear to learn, I can only imagine the sentiment they hold for implementing entirely new paradigms that come with budding cloud services. Alternatively, there are people like myself: grown children masquerading in an adult world. While we may appear to be older, we've simply reallocated our life's obsessions from Lego sets to cloud services.

I've fallen victim to the more "new technology" hype cycles than I'd like to admit. Straying down the path of dead-end tools is exhausting. I'm sure you'll recall the time the entire world arbitrarily decided to replace relational databases with NoSQL alternatives. I recognize how impulse architectural decisions create devastating technical debt. Thus, I would never think to push tools that have yet to stand the test of time and usefulness in the wild. Redis is one of such tools, perhaps ranking somewhere in my top 5 of useful new technologies in the past decade.

Why Redis Tho?

Why might I choose to proverbially shit on NoSQL databases prior to advocating for Redis: a NoSQL datastore? Make no mistake: Redis has almost *zero* similarities to NoSQL databases like MongoDB, in both intent and execution. MongoDB stores records on disk space, as do SQL databases. Allocating disk space to store a record comes with the implication that the information being stored is intended to persist, like user accounts, blog posts, permissions, or whatever. Most data worth saving falls into this category.

Surely not everything we do on the internet is worth saving forever. It would be weird (and inefficient) if we stored information like *items in a user's shopping cart* or *the last page of our app a user had visited*. This sort of information could be useful in the short term, but let's not fuck up the ACID-compliant databases our businesses depend on with endless disk I/O. Luckily for us, there's a little thing called RAM, which has handled similar scenarios since the dawn of computing. Redis is an in-memory database which stores values as a key/value pairs. Reading and writing to memory is faster than writing disk, which makes memory suitable for storing *secondary* data. This data might enable better user experiences while simultaneously keeping our databases clean. If we decide at a later time that data in memory is worth keeping, we can always write to disk (such as a SQL database) later on.

In-memory data stores like Redis or Memcached lie somewhere in your architecture between your app and your database. A good way to think of these services would be as alternatives to storing information via cookies. Cookies are bound to a single browser and can't reliably be depended on to work as expected, as any user can purge or disable browser cookies at any time. Storing session data in the cloud eliminates this problem with the added benefit of having session data shared across devices! Our user's shopping cart can now be visible from any of their devices as opposed to whichever browser they had been using at the time.

Today we'll familiarize ourselves with Python's go-to Redis package: redis-py. redis-py is obnoxiously referred to as `redis` within Python, presumably because the author of redis-py loves inflicting pain. See for yourself by visiting the official docs of redis-py: it's a single page of every method in the library listed in *alphabetical order* — pretty hilarious shit.

If you plan on using Redis with a Python web framework, you may be better off using a framework-specific package (such as Flask-Redis) over redis-py. Don't worry: nearly all Redis Python libraries follow the exact syntax of redis-py with some minor benefit of framework-specific integration. No matter which library you'll use, everything below still applies.

Setting Up Redis

If you don't have a Redis instance just yet, the folks over at Redis Labs offer a generous free tier for broke losers like you and me. They're quite a reputable Redis host, probably because they invented Redis in the first place. Once you have an instance set up, take note of the host, password, and port.

Enough chit chat, let's dig into Python. You know what to do:

```
$ pip install redis
```

Redis URIs

Like regular databases, we can connect to our Redis instance by creating a connection string URI. Here's what a real-life Redis URI might look like:

Example Redis URI

```
redis://:hostname.redislabs.com@mypassword:12345/0
```

Here's what's happening piece-by-piece:

Anatomy of a Redis URI

```
[CONNECTION_METHOD]:[HOSTNAME]@[PASSWORD]:[PORT]/[DATABASE]
```

- CONNECTION_METHOD: This is a suffix preceding all Redis URIs specifying *how* you'd like to connect to your instance. `redis://` is a standard connection, `rediss://` (double S) attempts to connect over SSL, `redis-socket://` is reserved for Unix domain sockets, and `redis-sentinel://` is a connection type for high-availability Redis clusters... presumably for people less broke than ourselves.
- HOSTNAME: The URL or IP your Redis instance. If you're using a cloud-hosted instance, chances are this will look like an AWS EC2 address. This is a small side-effect of modern-day capitalism shifting to a model where all businesses are AWS resellers with preconfigured software.
- PASSWORD: Redis instance have passwords yet lack users, presumably because a memory-based data store would inherently have a hard time managing persistent usernames.
- PORT: Is your preferred port of call after pillaging British trade ships. Just making sure you're still here.
- DATABASE: If you're unsure of what this is supposed to be, set it to `0`. People always do that.

Create a Redis Client

Awesome, we have our URI. Let's connect to Redis by creating a Redis client object:

Connecting to Redis

```
import redis
from config import redis_uri

r = redis.StrictRedis(url=redis_uri)
```

Why StrictRedis, you might ask? There are two ways to instantiate Redis clients: `redis.Redis()`, and `redis.StrictRedis()`. StrictRedis makes an effort to properly enforce Redis datatypes in ways that old Redis instances did not. `redis.Redis()` is backward-compatible with legacy Redis instances with trash data, where `redis.StrictRedis()` is not. When in doubt, use StrictRedis.

There are many other arguments we can (and should) pass into `redis.StrictRedis()` to make our lives easier. I highly recommend passing the keyword argument `decode_responses=True`, as this saves you the trouble of explicitly decoding every value that you fetch from Redis. It doesn't hurt to set a character set either:

Basic Redis client configuration

```
...

r = redis.StrictRedis(url=redis_uri
                      charset="utf-8",
                      decode_responses=True)
```

Birds-Eye View of Redis

Redis' key/value store is a similar concept to Python dictionaries, hence the meaning behind the name: Remote Dictionary Service. Keys are always strings, but there are a few data types available to us to store as values. Let's populate our Redis instance with a few records to get a better look at how Redis databases work:

Creating our first key/value pairs.

```
r.set('ip_address', '0.0.0.0')
```

```
r.set('timestamp', int(time.time()))
r.set('user_agent', 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_3)')
r.set('last_page_visited', 'account')
```

`r.set([KEY], [VALUE])` is your bread-and-butter for setting single values. It's as simple as it looks: the first parameter is your pair's key, while the second is the value assigned to said key.

Like a regular database, we can connect to our Redis instance via a GUI like TablePlus to check out our data. Here's what my database looks like after running the snippet above:

KEY	VALUE	TYPE	TTL
user_agent	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_3)	STRING	-1
last_page_visited	account	STRING	-1
ip_address	0.0.0.0	STRING	-1
timestamp	1580866091	STRING	-1

It looks like everything went swimmingly, eh? We can learn quite a bit about Redis simply by looking at this table. Let's start with the type column.

Data Types in Redis

Values we store in Redis can be any of 5 data types:

- STRING: Any value we create with `r.set()` is stored as a string type. You'll notice we set the value of *timestamp* to be an integer in our Python script, but it appears here as a string. This may seem obnoxiously inconvenient, there's more to Redis strings than meets the eye. For one, Redis strings are binary-safe, meaning they can be used to store the contents of nearly anything, including images or serialized objects. Strings also have several built-in functions that can manipulate strings as though they were numbers, such as incrementing with the **INC** command.
- LIST: Redis lists are mutable arrays of strings, where each string is sorted by the order in which they first appeared. Once a list is created, new items can be appended to the end of the array with the RPUSH command, or added at the zero-index via the LPUSH command. It's also possible to limit the maximum number of items in a list using the **LTRIM** command. If you're the kind of nerd that likes to do things like build LRU caches, you'll likely recognize the immediate benefit of this.
- SET: A set is an *unordered* list of strings. Like Python sets, Redis sets cannot contain duplicates. Sets have the unique ability to perform unions or intersections between other sets, which is a great way to combine or compare data quickly.
- ZSET: Stick with me here... ZSETs are sets which *are* ordered- they retain the same constraint of disallowing duplicates. Items in the list can have their order changed after creation, making ZSETs very useful for ranking unique items. They're somewhat similar to ordered dictionaries in Python, except with a key per value (which would be unnecessary, as all set values are unique).
- HASH: Redis hashes are key/value pairs in themselves, allowing you to assign a collection of key/value pairs as the value of a key/value pair. Hashes cannot be nested, because that would be crazy.

Data Expiration Dates

Our Redis database contains a fourth column labeled ttl. So far, each of our rows has a value of `-1` for this column. When this number is set to a positive integer, it represents the *number of seconds remaining before the data expires*. We've established that Redis is a great way to store temporarily useful data, but usually *not* valuable enough to hold on to. This is where setting an expiration on a new value comes in handy: it automatically ensures our instance's memory isn't bogged down with information that has become irrelevant.

Revisiting our example where we stored some user session information, let's try setting a value with an expiration:

Add an expiration to the **last_page_visited** key

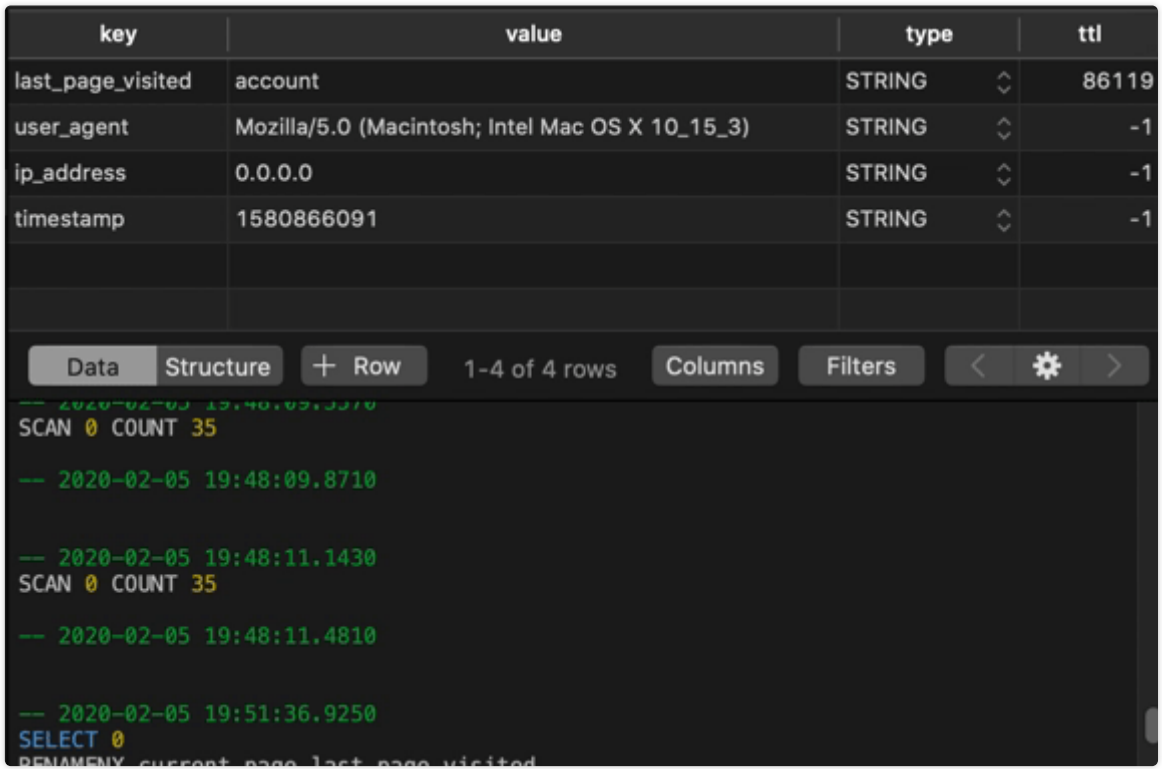
```
...

r.set('last_page_visited', 'account', 86400)
```

This time around we pass a third value to `r.set()` which represents the number of seconds our pair will be stored before self-destructing. Let's check out the database now:

KEY	VALUE	TYPE	TTL
user_agent	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_3)	STRING	-1
last_page_visited	account	STRING	86400
ip_address	0.0.0.0	STRING	-1
timestamp	1580866091	STRING	-1

If we use our GUI to refresh our table periodically, we can actually see this number counting down:



A Redis key slowly approaching death.

Working With Each Data Type

I'm sure you love hearing me drone on, but we both know why you're here: to copy & paste some code you need for your day job. I'll spoil you with a few cheatsheets which demonstrate some common use-cases when working with each of Redis' 5 datatypes.

Strings

If strings contain integer values, there are many methods we can use to modify the string as though it were an integer. Check out how we use `.incr()`, `.decr()`, and `.incrby()` here:

Redis string manipulation.

```
# Create string value
r.set('index', '1')
logger.info(f"index: {r.get('index')}")

# Increment string by 1
r.incr('index')
logger.info(f"index: {r.get('index')}")

# Decrement string by 1
r.decr('index')
logger.info(f"index: {r.get('index')}")

# Increment string by 3
r.incrby('index', 3)
logger.info(f"index: {r.get('index')}")
```

This assigns a value of `'1'` to `index`, increments the value of `index` by 1, decrements the value of `index` by 1, and finally increments the value of `index` by 3:

String manipulation output.

```
index: 1
index: 2
index: 1
index: 4
```

Lists

Below we add items to a Redis list using a combination of `.lpush()` and `.rpush()`, as well as popping an item out using `.lpop()`. You know, typical list stuff:

Redis list manipulation.

```
r.lpush('my_list', 'A')
logger.info(f"my_list: {r.lrange('my_list', 0, -1)}")

# Push second string to list from the right.
r.rpush('my_list', 'B')
logger.info(f"my_list: {r.lrange('my_list', 0, -1)}")

# Push third string to list from the right.
r.rpush('my_list', 'C')
logger.info(f"my_list: {r.lrange('my_list', 0, -1)}")

# Remove 1 instance from the list where the value equals 'C'.
r.lrem('my_list', 1, 'C')
logger.info(f"my_list: {r.lrange('my_list', 0, -1)}")

# Push a string to our list from the left.
r.lpush('my_list', 'C')
logger.info(f"my_list: {r.lrange('my_list', 0, -1)}")

# Pop first element of our list and move it to the back.
r.rpush('my_list', r.lpop('my_list'))
logger.info(f"my_list: {r.lrange('my_list', 0, -1)}")
```

Here's what our list looks like after each command:

List manipulation output.

```
my_list: ['A']
my_list: ['A', 'B']
my_list: ['A', 'B', 'C']
my_list: ['A', 'B']
my_list: ['C', 'A', 'B']
my_list: ['A', 'B', 'C']
```

Sets

Redis sets are powerful party due to their ability to interact with other sets. Below we create two separate sets and perform `.sunion()` and `.sinter()` on both:

Redis set manipulation.

```
# Add item to set 1
r.sadd('my_set_1', 'Y')
logger.info(f"my_set_1: {r.smembers('my_set_1')}'")

# Add item to set 1
r.sadd('my_set_1', 'X')
logger.info(f"my_set_1: {r.smembers('my_set_1')}'")

# Add item to set 2
r.sadd('my_set_2', 'X')
logger.info(f"my_set_2: {r.smembers('my_set_2')}'")

# Add item to set 2
r.sadd('my_set_2', 'Z')
logger.info(f"my_set2: {r.smembers('my_set_2')}'")

# Union set 1 and set 2
```

```
logger.info(f"Union: {r.sunion('my_set_1', 'my_set_2')}")

# Interset set 1 and set 2
logger.info(f"Intersect: {r.sinter('my_set_1', 'my_set_2')}")
```

As expected, our union combines the set without duplicates, and the intersect finds values common to both sets:

Set manipulation output.

```
my_set_1: {'Y'}'
my_set_1: {'X', 'Y'}'
my_set_2: {'X'}'
my_set2: {'X', 'Ƶ'}'
Union: {'X', 'Ƶ', 'Y'}
Intersect: {'X'}
```

Sorted Sets

Adding records to sorted sets using `.zadd()` has a bit of an interesting syntax. Take note below how adding a sorted set records expects a dictionary in the format of `{[VALUE]: [INDEX]}`:

Redis sorted set manipulation.

```
# Initialize sorted set with 3 values
r.zadd('top_songs_set', {'Never Change - Jay Ƶ': 1,
                         'Rich Girl - Hall & Oats': 2,
                         'The Prayer - Griz': 3})
logger.info(f"top_songs_set: {r.zrange('top_songs_set', 0, -1)}'")
```

```
# Add item to set with conflicting value
r.zadd('top_songs_set', {'Can\'t Figure it Out - Bishop Lamont': 3})
logger.info(f"top_songs_set: {r.zrange('top_songs_set', 0, -1)}'")

# Shift index of a value
r.zincrby('top_songs_set', 3, 'Never Change - Jay Ƶ')
logger.info(f"top_songs_set: {r.zrange('top_songs_set', 0, -1)}'")
```

Items in a sorted set can never share the same index, so when attempt to insert a value at an index where one exists, the existing value (and those following) get *pushed down* to make room. We're also able to change the indexes of values after we've created them:

Sorted set manipulation output.

```
top_songs_set: ['Never Change - Jay Ƶ',
                'Rich Girl - Hall & Oats',
                'The Prayer - Griz']
top_songs_set: ['Never Change - Jay Ƶ',
                'Rich Girl - Hall & Oats',
                'Can\'t Figure it Out - Bishop Lamont',
                'The Prayer - Griz']
top_songs_set: ['Rich Girl - Hall & Oats',
                'Can\'t Figure it Out - Bishop Lamont',
                The Prayer - Griz',
                'Never Change - Jay Ƶ']
```

Hashes

Alright, I didn't do much of anything interesting with hashes. Sue me. Still, creating and fetching a hash value is *kind of* cool, right?

Redis hash creation.

```
record = {
    "name": "Hackers and Slackers",
    "description": "Mediocre tutorials",
    "website": "https://hackersandslackers.com/",
    "github": "https://github.com/hackersandslackers"
}
r.hmset('business', record)
logger.info(f"business: {r.hgetall('business')}")
```

The output is exactly the same as the input... no surprises there:

Hash output.

```
business: {'name': 'Hackers and Slackers',
           'description': 'Mediocre tutorials',
           'website': 'https://hackersandslackers.com/',
           'github': 'https://github.com/hackersandslackers'}
```

Go Forth

There are plenty of Redis and redis-py features we've left unexplored, but you should have more than enough to get going. More importantly, hopefully, this tutorial has hopefully had some part in demonstrating *why* Redis could be useful in your stack… or not! If I can claim any part in preventing awful system architecture, that's a victory.

Oh yeah, and check out the repo for this tutorial here:

hackersandslackers/redis-python-tutorial

Contribute to hackersandslackers/redis-python-tutorial development by creating an account on GitHub.

hackersandslackers • GitHub

NoSQL | Python | Software Development | Architecture

Todd Birchard's avatar

Todd Birchard

116 Posts | New York City | Website | Twitter

Engineer with an ongoing identity crisis. Breaks everything before learning best practices. Completely normal and emotionally stable.

ADD A COMMENT

M↓ Markdown

python-poetry-package-manager

automate-ssh-scp-python-paramiko

faster-python-apis-in-two-minutes-really-just-one-line-of-code-to-change

Package Python Projects the Proper Way with Poetry

Python, Software Development

SSH & SCP in Python with Paramiko

Python, Automation

Faster Python APIs in Two Minutes

Python, REST APIs

Monthly Newsletter

Toss us your email and we'll promise to only give you the good stuff.

Your name

Your email address

Sign Up

Support us

We started sharing these tutorials to help and inspire new scientists and engineers around the world. If Hackers and Slackers has been helpful to you, feel free to buy us a coffee to keep us going :).

Buy us a coffee

Community of hackers obsessed with data science, data engineering, and analysis. Openly pushing a pro-robot agenda.

Pages

About

Series

Join

Search

Donate

Series

Code Snippet Corner

Building Flask Apps

Data Analysis with Pandas

Rise of Google Cloud

Learning Apache Spark

Creating APIs in AWS

Working with MySQL