0x02. Redis basic

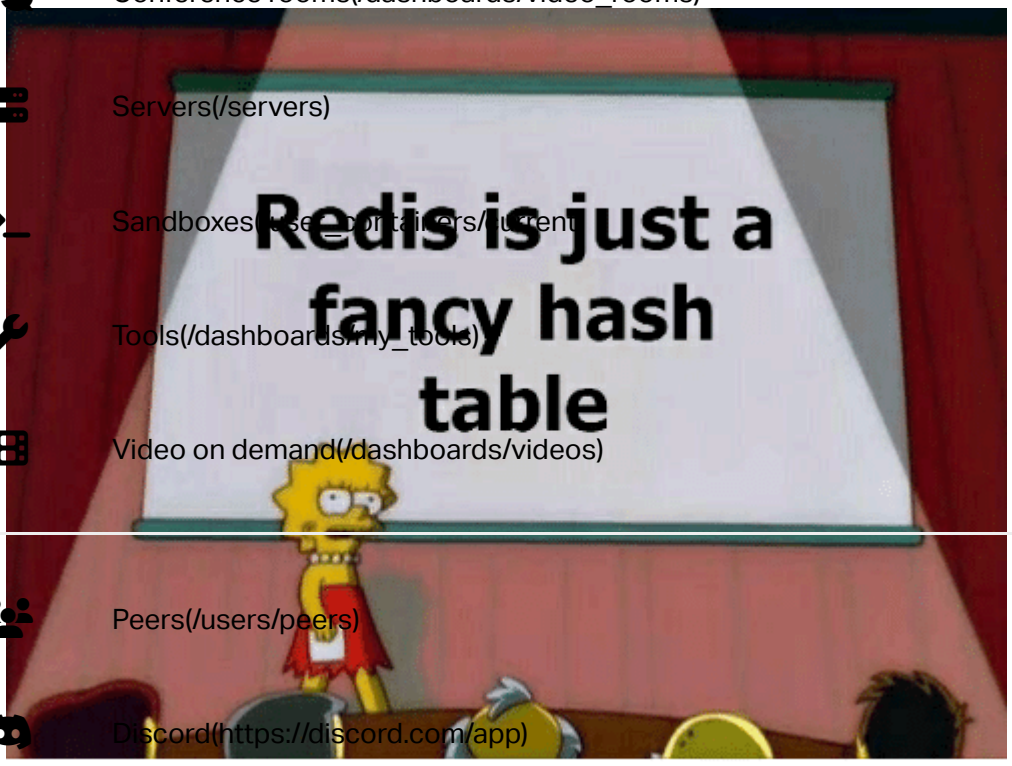
Weight: 1

Project over - took place from Jul 17, 2024 6:00 AM to Jul 18, 2024 6:00 AM

An auto QA review will be launched at the deadline

In a nutshell…

- **Auto QA review:** 27.0/27 mandatory & 4.0/6 optional
- **Altogether: 166.67%**
  - Mandatory: 100.0%
  - Optional: 66.67%
  - Calculation: 100.0% + (100.0% * 66.67%) == **166.67%**



Resources

Read or watch:

- Redis Crash Course Tutorial (/rltoken/hJVo3XwMMFFoApyX8zPXvA)
- Redis commands (/rltoken/oauvbRmxM12SxvimzqhrOg)
- Redis python client (/rltoken/imfgFhAZPlg7YMZ_tHvFZw)
- How to Use Redis With Python (/rltoken/7SluvFvgckwVgsvrfOf1CQ)

Learning Objectives

- Learn how to use redis for basic operations
- Learn how to use redis as a simple cache

Requirements

- All of your files will be interpreted/compiled on Ubuntu 18.04 LTS using python3 (version 3.7)
- All of your files should end with a new line
- A `README.md` file, at the root of the folder of the project, is mandatory
- The first line of all your files should be exactly `#!/usr/bin/env python3`
- Your code should use the `pycodestyle` style (version 2.5)
- All your modules should have documentation (`python3 -c 'print(__import__("my_module").__doc__)'`)
- All your classes should have documentation (`python3 -c 'print(__import__("my_module").MyClass.__doc__)'`)
- All your functions and methods should have documentation (`python3 -c 'print(__import__("my_module").my_function.__doc__)'` and `python3 -c 'print(__import__("my_module").MyClass.my_function.__doc__)'`)
- A documentation is not a simple word, it's a real sentence explaining what's the purpose of the module, class or method (the length of it will be verified)
- All your functions and coroutines must be type-annotated.

Install Redis on Ubuntu 18.04

```
$ sudo apt-get -y install redis-server
$ pip3 install redis
$ sed -i "s/bind .*/bind 127.0.0.1/g" /etc/redis/redis.conf
```

Use Redis in a container

- Redis server is stopped by default - when you are starting a container, you should start it with: `service redis-server start`

Tasks

0. Writing strings to Redis

mandatory

Score: 100.0% (*Checks completed: 100.0%*)

Create a `Cache` class. In the `__init__` method, store an instance of the Redis client as a private variable named `_redis` (using `redis.Redis()`) and flush the instance using `flushdb`.

Create a `store` method that takes a `data` argument and returns a string. The method should generate a random key (e.g. using `uuid`), store the input data in Redis using the random key and return the key.

Type-annotate `store` correctly. Remember that `data` can be a `str`, `bytes`, `int` or `float`.

```
bob@dylan:~$ cat main.py
#!/usr/bin/env python3
"""
Main file
"""
import redis

Cache = __import__('exercise').Cache

cache = Cache()

data = b"hello"
key = cache.store(data)
print(key)

local_redis = redis.Redis()
print(local_redis.get(key))
bob@dylan:~$ python3 main.py
3a3e8231-b2f6-450d-8b0e-0f38f16e8ca2
b'hello'
bob@dylan:~$
```

Repo:

- GitHub repository: `alx-backend-storage`
- Directory: `0x02-redis_basic`
- File: `exercise.py`

Check submission | Get a sandbox | View results

1. Reading from Redis and recovering original type

mandatory

Score: 100.0% (*Checks completed: 100.0%*)

Redis only allows to store string, bytes and numbers (and lists thereof). Whatever you store as single elements, it will be returned as a byte string. Hence if you store `"a"` as a UTF-8 string, it will be returned as `b"a"` when retrieved from the server.

In this exercise we will create a `get` method that take a `key` string argument and an optional `Callable` argument named `fn`. This callable will be used to convert the data back to the desired format.

Remember to conserve the original `Redis.get` behavior if the key does not exist.

Also, implement 2 new methods: `get_str` and `get_int` that will automatically parametrize `Cache.get` with the correct conversion function.

The following code should not raise:

```
cache = Cache()

TEST_CASES = {
    b"foo": None,
    123: int,
    "bar": lambda d: d,
}

for value, fn in TEST_CASES.items():
    key = cache.store(value)
    assert cache.get(key, fn=fn) == value
```

Repo:

- GitHub repository: `alx-backend-storage`
- Directory: `0x02-redis_basic`
- File: `exercise.py`

Check submission | >_ Get a sandbox | View results

2. Incrementing values

mandatory

Score: 100.0% (*Checks completed: 100.0%*)

Familiarize yourself with the `INCR` command and its python equivalent.

In this task, we will implement a system to count how many times methods of the `Cache` class are called.

Above `Cache` define a `count_calls` decorator that takes a single `method Callable` argument and returns a `Callable`.

As a key, use the qualified name of `method` using the `__qualname__` dunder method.

Create and return function that increments the count for that key every time the method is called and returns the value returned by the original method.

Remember that the first argument of the wrapped function will be `self` which is the instance itself, which lets you access the Redis instance.

Protip: when defining a decorator it is useful to use `functool.wraps` to conserve the original function's name, docstring, etc. Make sure you use it as described here (/rltoken/eRjLY2hVLrkDcNkcDJDK3g).

Decorate `Cache.store` with `count_calls`.

```
bob@dylan:~$ cat main.py
#!/usr/bin/env python3
""" Main file """

Cache = __import__('exercise').Cache

cache = Cache()

cache.store(b"first")
print(cache.get(cache.store.__qualname__))

cache.store(b"second")
cache.store(b"third")
print(cache.get(cache.store.__qualname__))

bob@dylan:~$ ./main.py
b'1'
b'3'
bob@dylan:~$
```

Repo:

- GitHub repository: `alx-backend-storage`
- Directory: `0x02-redis_basic`
- File: `exercise.py`

Check submission | Get a sandbox | View results

3. Storing lists

mandatory

Score: 100.0% (Checks completed: 100.0%)

Familiarize yourself with redis commands `RPUSH`, `LPUSH`, `LRANGE`, etc.

In this task, we will define a `call_history` decorator to store the history of inputs and outputs for a particular function.

Everytime the original function will be called, we will add its input parameters to one list in redis, and store its output into another list.

In `call_history`, use the decorated function's qualified name and append `":inputs"` and `":outputs"` to create input and output list keys, respectively.

`call_history` has a single parameter named `method` that is a `Callable` and returns a `Callable`.

In the new function that the decorator will return, use `rpush` to append the input arguments. Remember that Redis can only store strings, bytes and numbers. Therefore, we can simply use `str(args)` to normalize. We can ignore potential `kwargs` for now.

Execute the wrapped function to retrieve the output. Store the output using `rpush` in the `"...:outputs"` list, then return the output.

Decorate `Cache.store` with `call_history`.

```
bob@dylan:~$ cat main.py
#!/usr/bin/env python3
""" Main file """
Cache = __import__('exercise').Cache

cache = Cache()

s1 = cache.store("first")
print(s1)
s2 = cache.store("secont")
print(s2)
s3 = cache.store("third")
print(s3)

inputs = cache._redis.lrange("{}:inputs".format(cache.store.__qualname__), 0, -1)
outputs = cache._redis.lrange("{}:outputs".format(cache.store.__qualname__), 0, -
1)

print("inputs: {}".format(inputs))
print("outputs: {}".format(outputs))

bob@dylan:~$ ./main.py
04f8dcaa-d354-4221-87f3-4923393a25ad
a160a8a8-06dc-4934-8e95-df0cb839644b
15a8fd87-1f55-4059-86aa-9d1a0d4f2aea
inputs: [b"('first',)", b"('secont',)", b"('third',)"]
outputs: [b'04f8dcaa-d354-4221-87f3-4923393a25ad', b'a160a8a8-06dc-4934-8e95-df0c
b839644b', b'15a8fd87-1f55-4059-86aa-9d1a0d4f2aea']
bob@dylan:~$
```

Repo:

- GitHub repository: `alx-backend-storage`
- Directory: `0x02-redis_basic`
- File: `exercise.py`

Check submission | Get a sandbox | View results

4. Retrieving lists

mandatory

Score: 100.0% (*Checks completed: 100.0%*)

In this tasks, we will implement a `replay` function to display the history of calls of a particular function.

Use keys generated in previous tasks to generate the following output:

```
>>> cache = Cache()
>>> cache.store("foo")
>>> cache.store("bar")
>>> cache.store(42)
>>> replay(cache.store)
Cache.store was called 3 times:
Cache.store(*('foo',)) -> 13bf32a9-a249-4664-95fc-b1062db2038f
Cache.store(*('bar',)) -> dcddd00c-4219-4dd7-8877-66afbe8e7df8
Cache.store(42,) -> 3e782f2b-ecd8-4925-a3ce-e2efdee08d20
```

Tip: use `lrange` and `zip` to loop over inputs and outputs.

Repo:

- GitHub repository: `alx-backend-storage`
- Directory: `0x02-redis_basic`
- File: `exercise.py`

Check submission | Get a sandbox | View results

5. Implementing an expiring web cache and tracker

#advanced

Score: 66.67% (*Checks completed: 66.67%*)

In this tasks, we will implement a `get_page` function (prototype: `def get_page(url: str) -> str:`). The core of the function is very simple. It uses the `requests` module to obtain the HTML content of a particular URL and returns it.

Start in a new file named `web.py` and do not reuse the code written in `exercise.py`.

Inside `get_page` track how many times a particular URL was accessed in the key `"count:{url}"` and cache the result with an expiration time of 10 seconds.

Tip: Use `http://slowwly.robertomurray.co.uk` to simulate a slow response and test your caching.

Bonus: implement this use case with decorators.

Repo:

- GitHub repository: `alx-backend-storage`
- Directory: `0x02-redis_basic`
- File: `web.py`

Check submission | Mark submission | Get a sandbox | View results

(/)

Home(/)

My Planning(/planning/me)

Projects(/projects/current)

QA Reviews I can make(/corrections/to_review)

Evaluation quizzes(/dashboards/my_current_evaluation_quizzes)

Curriculums(/dashboards/my_curriculums)

Concepts(/concepts)

Conference rooms(/dashboards/video_rooms)

Servers(/servers)

Sandboxes(/user_containers/current)

Tools(/dashboards/my_tools)

Video on demand(/dashboards/videos)

Peers(/users/peers)

Discord(https://discord.com/app)

My Profile(/users/my_profile)