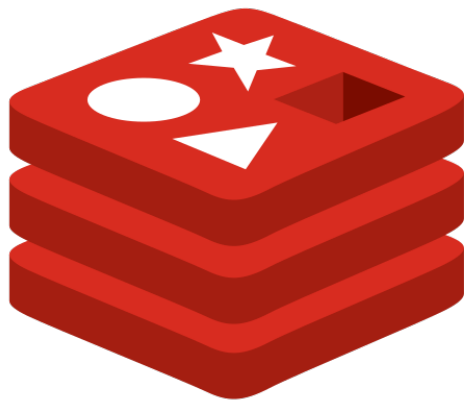


#### Exercise #4: Key-value store with Redis

**Redis** is an open-source, in-memory key-value data store known for its flexibility, performance, and broad language support. Redis doesn't use structured query language (otherwise known as SQL) to store, manipulate, and retrieve data. Instead, it comes with its own set of commands for managing and accessing data.

Redis has some main peculiarities that sets it apart. For example, Redis holds its database entirely in the memory, using the disk only for persistence.



# redis

#### Tasks:

- **Task 1. Access a Redis server**
- **Task 2. Data types: String**
- **Task 3. Data types: List**
- Task 4. Data types: Set
- Task 5. Data types: Hash
- Task 6. Data types: Zset

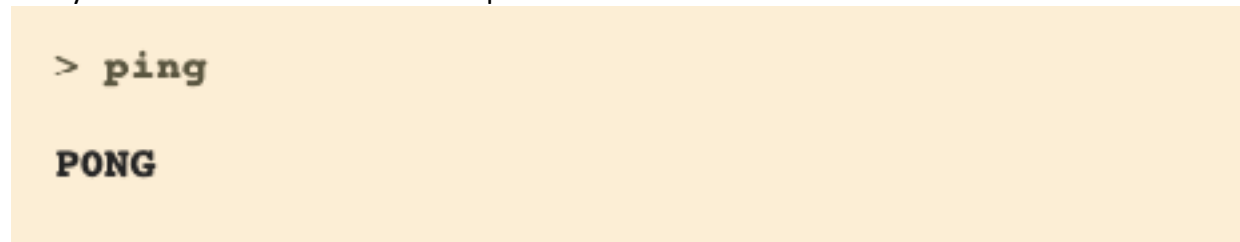
## Task 1. Access a Redis server

Go to <https://try.redis.io/> in order to test several commands that demonstrate Redis capacity as an in-memory data store.

All commands will be tested in the “terminal” provided by the website



And you'll observe the results in the panel.



In order to check if Redis is working properly, send a **PING** command (*pong message should be received*)

### Working with Keys

The essence of a key-value store is the ability to store some data, called a value, inside a key. The value can be retrieved later only if we know the specific key it was stored in. There is no direct way to search for a key by value. In some sense, it is like a very large hash/dictionary, but it is persistent, i.e. when your application ends, the data doesn't go away.

Redis keys are binary safe, this means that you can use any binary sequence as a key, from a string like "foo" to the content of a JPEG file. The empty string is also a valid key.

However, consider the following for a key identifier:

- Very long keys are not a good idea. For instance a key of 1024 bytes is a bad idea not only memory-wise, but also because the lookup of the key in the dataset may require several costly key-comparisons.
- Very short keys are often not a good idea. There is little point in writing "u1000flw" as a key if you can instead write "user:1000:followers". The latter is more readable and the added space is minor compared to the space used by the key object itself and the value object. While short keys will obviously consume a bit less memory, your job is to find the right balance.

- Try to stick with a schema. For instance "object-type:id" is a good idea ("user:1000"). Dots or dashes are often used for multi-word fields ("comment:4321:reply.to") or "comment:4321:reply-to".
- The maximum allowed key size is 512 MB.

## Task 2. Data types: String

The **Redis String** type is the simplest type of value you can associate with a Redis key.

Since Redis keys are strings, when we use the string type as a value too, we are mapping a string to another string. The string data type is useful for a number of use cases, like caching HTML fragments or pages.

To set a new key-value pair, use the **SET** command.

```
> set message hello  
  
OK
```

You just set a key "message" with a value "hello".

To retrieve that value, you can run the **GET** command:

```
> get message  
  
"hello"
```

If you try to get a nonexistent key, Redis will not return anything.

```
> get greeting  
  
(nil)
```

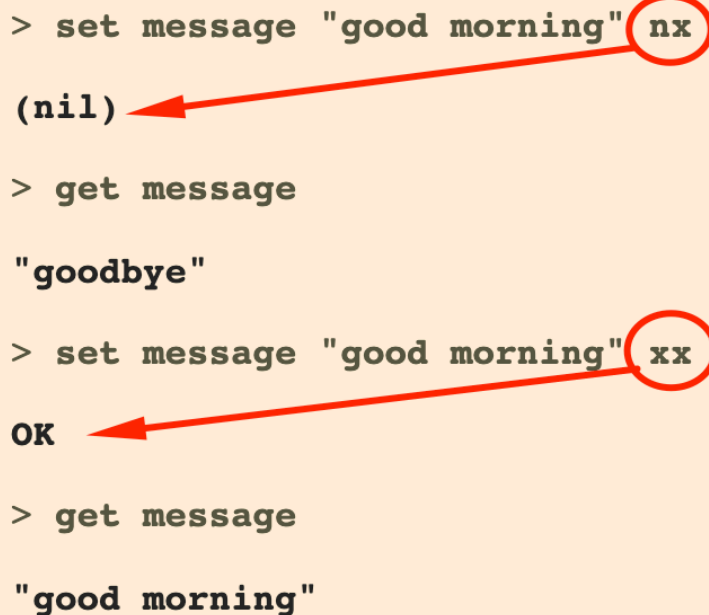
SET will replace any existing value already stored into the key, in the case that the key already exists.

```
> set message goodbye  
  
OK  
  
> get message  
  
"goodbye"
```

Values can be strings (including binary data) of every kind, for instance you can store a jpeg image inside a value. A value can't be bigger than 512 MB.

The SET command has interesting options, that are provided as additional arguments. For example, to ask SET to fail if the key already exists, or the opposite, that it only succeed if the key already exists:

- NX -- Only set the key if it does not already exist.
- XX -- Only set the key if it already exists.



```
> set message "good morning" nx
(nil)
> get message
"goodbye"
> set message "good morning" xx
OK
> get message
"good morning"
```

To delete a key-value pair, use **DEL**:

```
> del message
(integer) 1
> get message
(nil)
```

When you run DEL, it returns (integer) 1. This is because Redis is returning the number of affected items. In this case, you only deleted a pair, so it returns 1.

An important Redis feature is known as **key expiration**, which lets you set a timeout for a key, also known as a "time to live" (TTL). When the time to live elapses, the key is automatically destroyed.

A few important notes about key expiration:

- They can be set both using seconds or milliseconds precision.
- The expire time resolution is always 1 millisecond.

You can specify an expire time (in seconds) for a key when you create it with the **EX** parameter. Before that time, the key will be accessible.

```
> set mission "This message will self-destruct in 10 seconds" ex 10  
  
OK  
  
> get mission  
  
"This message will self-destruct in 10 seconds"
```

The **TTL** command returns the remaining time to live (in seconds) of a key that has a timeout.

```
> ttl mission  
  
(integer) 47
```

After the expiry time, the key will no longer be accessible.

```
> get mission  
  
(nil)
```

TTL command returns -1 if the key exists but has no associated expire.

```
> ttl message  
  
(integer) -1
```

If the key already exists, you can use the **EXPIRE** command to set an expiry time for that key.

```
> expire message 30  
  
(integer) 1  
  
> get message  
  
"good morning"  
  
> ttl message  
  
(integer) 20  
  
> get message  
  
(nil)
```

Use **PEXPIRE** to set an expiry time for a key in milliseconds. **PTTL** returns the remaining time to live of a key that has an expiry set in milliseconds:

```
> set message "hello"  
  
OK  
  
> pexpire message 30251  
  
(integer) 1  
  
> pttl message  
  
22971  
  
> ttl message  
  
(integer) 21
```

Even if strings are the basic values of Redis, there are interesting operations you can perform with them. For instance, one is atomic increment with the **INCR** command. There are other similar commands like **INCRBY**, **DECR** and **DECRBY**. Internally it's always the same command, acting in a slightly different way.

```
> set counter 100
OK
> incr counter
(integer) 101
> incr counter
(integer) 102
> incrby counter 50
(integer) 152
> decr counter
(integer) 151
> decrby counter 20
(integer) 131
```

If the value is a floating-point number, use the INCRBYFLOAT command. Pass a negative parameter if you want to decrement the value:

```
> set salary 256.75
OK
> incr salary
(error) ERR value is not an integer or out of range
> incrbyfloat salary 20.6
277.35
> incrbyfloat salary -12.5
264.85
```



The EXISTS command returns 1 or 0 to signal if a given key exists or not in the data store.

```
> exists salary  
  
(integer) 1  
  
> exists price  
  
(integer) 0
```

---

Redis actually has 5 different data types. What you just learned above was setting a key-value pair for a **string** data type. The 5 data types are:

- String
- List
- Set
- Hash
- Zset

Let's go through the other 4. But before, how to check the kind of value store at a key? Use the TYPE command:

```
> set test 10  
  
OK  
  
> type test  
  
"string"
```

### Task 3. Data types: List

Redis lists are implemented via **Linked Lists**. This means that even if you have millions of elements inside a list, the operation of adding a new element in the head or in the tail of the list is performed in constant time. The speed of adding a new element with the **LPUSH** command to the head of a list with ten elements is the same as adding an element to the head of list with 10 million elements.

What's the downside? Accessing an element by index is not so fast in linked lists (where the operation requires an amount of work proportional to the index of the accessed element).

Redis Lists are implemented with linked lists because for a database system it is crucial to be able to add elements to a very long list in a very fast way. Another strong advantage, as you'll see in a moment, is that Redis Lists can be taken at constant length in constant time.

When fast access to the middle of a large collection of elements is important, there is a different data structure that can be used, called **sorted sets** (will be covered later).

Commands:

- The **LPUSH** command adds a new element into a list, on the left (at the head).
- The **RPUSH** command adds a new element into a list, on the right (at the tail).

Both RPUSH and LPUSH commands will create a list if it doesn't exist before adding the first element. Each time you add to the list, Redis returns its size.

```
> rpush countryList "Czech Republic"
(integer) 1
> lpush countryList "Mexico"
(integer) 2
> rpush countryList "Italy"
(integer) 3
> lpush countryList "Japan"
(integer) 4
> lpush countryList "Nigeria"
(integer) 5
```

The **LRange** command extracts ranges of elements from lists. It takes two indexes: the first and the last element of the range to return. Both the indexes can be negative, telling Redis to start counting from the end: so -1 is the last element, -2 is the penultimate element of the list, and so forth.

```
> lrange countryList 0 -1
```

```
1) "Nigeria"  
2) "Japan"  
3) "Mexico"  
4) "Czech Republic"  
5) "Italy"
```

```
> lrange countryList -1 0
```

```
(empty list or set)
```

```
> lrange countryList 0 2
```

```
1) "Nigeria"  
2) "Japan"  
3) "Mexico"
```

```
> lrange countryList 3 3
```

```
1) "Czech Republic"
```

You are free to push multiple elements into a list in a single call:

```
> lpush countryList "Spain" "United States" "Egypt"
```

```
(integer) 8
```

```
> rpush countryList "India" "Germany" "Australia"
```

```
(integer) 11
```

```
> lrange countryList 0 -1
```

- 1) "Egypt"
- 2) "United States"
- 3) "Spain"
- 4) "Nigeria"
- 5) "Japan"
- 6) "Mexico"
- 7) "Czech Republic"
- 8) "Italy"
- 9) "India"
- 10) "Germany"
- 11) "Australia"

An important operation defined on Redis lists is the ability to pop elements. Popping elements is the operation of both retrieving the element from the list, and eliminating it from the list, at the same time. You can pop elements from left and right, similarly to how you can push elements in both sides of the list with **RPOP** and **LPOP** commands:

```
> lpop countryList
```

```
"Egypt"
```

```
> rpop countryList
```

```
"Australia"
```

```
> rpop countryList
```

```
"Germany"
```

```
> lrange countryList 0 -1
```

- 1) "United States"
- 2) "Spain"
- 3) "Nigeria"
- 4) "Japan"
- 5) "Mexico"
- 6) "Czech Republic"
- 7) "Italy"
- 8) "India"

Redis returns a NULL value to signal that there are no elements in the list if you try to pop from an empty list.

## Capped Lists

In many use cases we just want to use lists to store the latest items, whatever they are: social network updates, logs, or anything else.

Redis allows us to use lists as a capped collection, only remembering the latest N items and discarding all the oldest items using the **LTRIM** command.

The LTRIM command is similar to LRANGE, but instead of displaying the specified range of elements it sets this range as the new list value. All the elements outside the given range are removed.

```
> lrange countryList 0 -1
```

```
1) "United States"  
2) "Spain"  
3) "Nigeria"  
4) "Japan"  
5) "Mexico"  
6) "Czech Republic"  
7) "Italy"  
8) "India"
```

```
> ltrim countryList 0 6
```

```
OK
```

```
> lrange countryList 0 -1
```

```
1) "United States"  
2) "Spain"  
3) "Nigeria"  
4) "Japan"  
5) "Mexico"  
6) "Czech Republic"  
7) "Italy"
```

The above **LTRIM** command tells Redis to take just list elements from index 0 to 6, everything else will be discarded. This allows for a very simple but useful pattern: doing a List push operation + a List trim operation together in order to add a new element and discard elements exceeding a limit:

```
> lpush countryList France
(integer) 8

> ltrim countryList 0 6

OK

> lrange countryList 0 -1

1) "France"
2) "United States"
3) "Spain"
4) "Nigeria"
5) "Japan"
6) "Mexico"
7) "Czech Republic"
```

By the way, what is the type of countryList?

```
> type countryList

"list"
```

The LLEN command returns the number of elements in a list

```
> llen countryList

(integer) 7
```

We don't have to create empty lists before pushing elements, or to remove empty lists when they no longer have elements inside. It is Redis' responsibility to delete keys when lists are left empty, or to create an empty list if the key does not exist and we are trying to add elements to it, for example, with LPUSH.

This is not specific to lists, it applies to all the Redis data types composed of multiple elements -- Streams, Sets, Sorted Sets and Hashes.

### Advanced commands

**LMOVE** command atomically returns and removes the first/last element (head/tail depending on the **wherefrom** argument) of the list stored at source and pushes the element at the first/last element (head/tail depending on the **whereto** argument) of the list stored at destination.

```
redis> RPUSH mylist "one"
(integer) 1

redis> RPUSH mylist "two"
(integer) 2

redis> RPUSH mylist "three"
(integer) 3

redis> LMOVE mylist myotherlist RIGHT LEFT
"three"

redis> LMOVE mylist myotherlist LEFT RIGHT
"one"

redis> LRANGE mylist 0 -1
1) "two"

redis> LRANGE myotherlist 0 -1
1) "three"
2) "one"
```

#### Task 4. Data types: Set

Redis Sets are unordered collections of strings. The **SADD** command adds new elements to a set.

Redis is free to return the elements in any order at every call, since there is no contract with the user about element ordering. The command **SMEMBERS** retrieves all elements in a set

```
> SADD fruits apple banana orange
```

```
(integer) 3
```

```
> smembers fruits
```

- 1) "apple"
- 2) "orange"
- 3) "banana"

```
> sadd fruits lemon
```

```
(integer) 1
```

```
> smembers fruits
```

- 1) "apple"
- 2) "lemon"
- 3) "orange"
- 4) "banana"

Redis has commands to test for membership. For example, checking if an element exists with **SISMEMBER**

```
> sismember fruits lemon
```

```
(integer) 1
```

```
> sismember fruits pear
```

```
(integer) 0
```



The **SINTER** command performs the intersection between different sets.

```
> sadd store apple melon mango lime banana

(integer) 5

> sinter fruits store

1) "apple"
2) "banana"
```

The command **SUNION** returns the members of the set resulting from the union of all the given sets.

```
> sunion fruits store

1) "lime"
2) "mango"
3) "apple"
4) "lemon"
5) "orange"
6) "melon"
7) "banana"
```

The command **SUNIONSTORE** is equal to **SUNION**, but instead of returning the resulting set, it is stored in **destination**. If destination already exists, it is overwritten.

```
> sunionstore market fruits store

7

> smembers market

1) "lime"
2) "mango"
3) "apple"
4) "lemon"
5) "orange"
6) "melon"
7) "banana"
```

**SRANDMEMBER** command returns a random element from the set value stored at key.

If a provided **count** argument is positive, return an array of distinct elements. The array's length is either count or the set's cardinality (SCARD), whichever is lower.

If called with a negative count, the behavior changes and the command is allowed to return the same element multiple times. In this case, the number of returned elements is the absolute value of the specified count.

```
> srandmember market
```

```
"melon"
```

```
> srandmember market
```

```
"banana"
```

```
> srandmember market 4
```

```
1) "mango"
```

```
2) "lemon"
```

```
3) "banana"
```

```
4) "melon"
```

```
> srandmember market -4
```

```
1) "apple"
```

```
2) "mango"
```

```
3) "mango"
```

```
4) "lemon"
```

**SDIFF** command returns the members of the set resulting from the difference between the first set and all the successive sets.

```
> sdiff fruits store
```

```
1) "orange"
```

```
2) "lemon"
```

```
> sdiff store fruits
```

```
1) "lime"
```

```
2) "mango"
```

```
3) "melon"
```

**SDIFFSTORE** command is equal to **SDIFF**, but instead of returning the resulting set, it is stored in destination.

```
> sdiffstore supermarket fruits store
```

```
2
```

```
> smembers supermarket
```

```
1) "orange"
```

```
2) "lemon"
```

## Task 5. Data types: Hash

While hashes are handy to represent objects, actually the number of fields you can put inside a hash has no practical limits (other than available memory), so you can use hashes in many different ways inside your application.

The command HSET sets multiple fields of the hash, while HGET retrieves a single field. HMGET is similar to HGET but returns an array of values:

```
> hset user:1000 username antirez birthyear 1977 verified 1
(integer) 3
> hget user:1000 username
"antirez"
> hget user:1000 birthyear
"1977"
> hgetall user:1000
1) "username"
2) "antirez"
3) "birthyear"
4) "1977"
5) "verified"
6) "1"
```

```
> hmget user:1000 username birthyear no-such-field
1) "antirez"
2) "1977"
3) (nil)
```

There are commands that are able to perform operations on individual fields as well, like HINCRBY:

```
> hincrby user:1000 birthyear 10  
(integer) 1987  
> hincrby user:1000 birthyear 10  
(integer) 1997
```

## Task 6. Data types: Zset

Sorted sets are a data type which is similar to a mix between a Set and a Hash. Like sets, sorted sets are composed of unique, non-repeating string elements, so in some sense a sorted set is a set as well.

However, while elements inside sets are not ordered, every element in a sorted set is associated with a floating point value, called the score (this is why the type is also similar to a hash, since every element is mapped to a value).

Moreover, elements in a sorted sets are taken in order (so they are not ordered on request, order is a peculiarity of the data structure used to represent sorted sets). They are ordered according to the following rule:

- If B and A are two elements with a different score, then  $A > B$  if  $A.score > B.score$ .
- If B and A have exactly the same score, then  $A > B$  if the A string is lexicographically greater than the B string. B and A strings can't be equal since sorted sets only have unique elements.

```
> zadd hackers 1940 "Alan Kay"
(integer) 1
> zadd hackers 1957 "Sophie Wilson"
(integer) 1
> zadd hackers 1953 "Richard Stallman"
(integer) 1
> zadd hackers 1949 "Anita Borg"
(integer) 1
> zadd hackers 1965 "Yukihiro Matsumoto"
(integer) 1
> zadd hackers 1914 "Hedy Lamarr"
(integer) 1
> zadd hackers 1916 "Claude Shannon"
(integer) 1
> zadd hackers 1969 "Linus Torvalds"
(integer) 1
```

```
> zadd hackers 1912 "Alan Turing"
```

The **ZADD** command is similar to SADD but takes one additional argument (placed before the element to be added) which is the score. ZADD is also variadic, so you are free to specify multiple score-value pairs.

With sorted sets it is trivial to return a list of hackers sorted by their birth year because actually they are already sorted.

```
> zrange hackers 0 -1
```

- 1) "Alan Turing"**
- 2) "Hedy Lamarr"**
- 3) "Claude Shannon"**
- 4) "Alan Kay"**
- 5) "Anita Borg"**
- 6) "Richard Stallman"**
- 7) "Sophie Wilson"**
- 8) "Yukihiro Matsumoto"**
- 9) "Linus Torvalds"**

Use ZREVRANGE instead of ZRANGE to return the elements in reverse order:

```
> zrevrange hackers 0 -1
```

- 1) "Linus Torvalds"**
- 2) "Yukihiro Matsumoto"**
- 3) "Sophie Wilson"**
- 4) "Richard Stallman"**
- 5) "Anita Borg"**
- 6) "Alan Kay"**
- 7) "Claude Shannon"**
- 8) "Hedy Lamarr"**
- 9) "Alan Turing"**

It is possible to return scores as well, using the WITHSCORES argument:

```
> zrange hackers 0 -1 withscores
```

```
1) "Alan Turing"  
2) 1912.0  
3) "Hedy Lamarr"  
4) 1914.0  
5) "Claude Shannon"  
6) 1916.0  
7) "Alan Kay"  
8) 1940.0  
9) "Anita Borg"  
10) 1949.0  
11) "Richard Stallman"  
12) 1953.0  
13) "Sophie Wilson"  
14) 1957.0  
15) "Yukihiro Matsumoto"  
16) 1965.0  
17) "Linus Torvalds"  
18) 1969.0
```

Sorted sets are powerful. They can operate on ranges. Let's get all the individuals that were born up to 1950 inclusive. We use the ZRANGEBYSCORE command to do it:

```
> zrangebyscore hackers -inf 1950
```

```
1) "Alan Turing"  
2) "Hedy Lamarr"  
3) "Claude Shannon"  
4) "Alan Kay"  
5) "Anita Borg"
```

We asked Redis to return all the elements with a score between negative infinity and 1950 (both extremes are included).



It's also possible to remove ranges of elements. Let's remove all the hackers born between 1940 and 1960 from the sorted set:

```
> zremrangebyscore hackers 1940 1960
```

```
4
```

```
> zrange hackers 0 -1
```

- 1) "Alan Turing"
- 2) "Hedy Lamarr"
- 3) "Claude Shannon"
- 4) "Yukihiro Matsumoto"
- 5) "Linus Torvalds"

Another extremely useful operation defined for sorted set elements is the get-rank operation. It is possible to ask what is the position of an element in the set of the ordered elements.

```
> zrank hackers "Claude Shannon"
```

```
2
```

The ZREVRANK command is also available in order to get the rank, considering the elements sorted a descending way.

```
> zrevrank hackers "Hedy Lamarr"
```

```
3
```

Let's add the following instruction:

```
ZADD hackersv2 0 "Alan Kay" 0 "Sophie Wilson" 0 "Richard Stallman" 0  
"Anita Borg" 0 "Yukihiro Matsumoto" 0 "Hedy Lamarr" 0 "Claude Shannon"  
0 "Linus Torvalds" 0 "Alan Turing"
```

And then

```
> zrange hackersv2 0 -1
```

```
1) "Alan Kay"  
2) "Alan Turing"  
3) "Anita Borg"  
4) "Claude Shannon"  
5) "Hedy Lamarr"  
6) "Linus Torvalds"  
7) "Richard Stallman"  
8) "Sophie Wilson"  
9) "Yukihiro Matsumoto"
```

The command **KEYS \*** displays all available keys in your Redis server:

```
> keys *
```

- 1) "countryList"
- 2) "counter"
- 3) "name"
- 4) "test"
- 5) "key:1"
- 6) "salary"
- 7) "key"
- 8) "India"
- 9) "salari"

## Appendix

If you want to install Redis server in your computer, here are the instructions:

<https://redis.io/docs/getting-started/>

# Getting started with Redis

How to get up and running with Redis

This is a guide to getting started with Redis. You'll learn how to install, run, and experiment with the Redis server process.

## Install Redis

How you install Redis depends on your operating system. See the guide below that best fits your needs:

- [Install Redis from Source](#)
- [Install Redis on Linux](#)
- [Install Redis on macOS](#)
- [Install Redis on Windows](#)

Once you have Redis up and running, you can connect using **redis-cli**.

```
$ redis-cli
redis 127.0.0.1:6379> ping
PONG
redis 127.0.0.1:6379> set mykey somevalue
OK
redis 127.0.0.1:6379> get mykey
"somevalue"
```