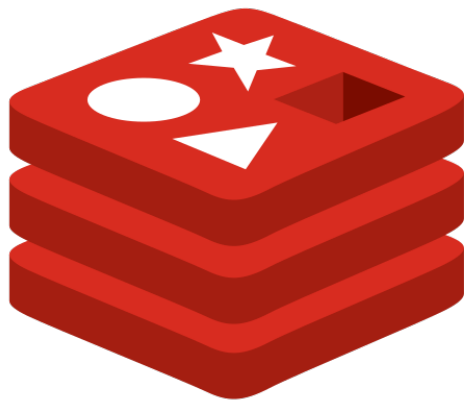


Exercise #4: Key-value store with Redis

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

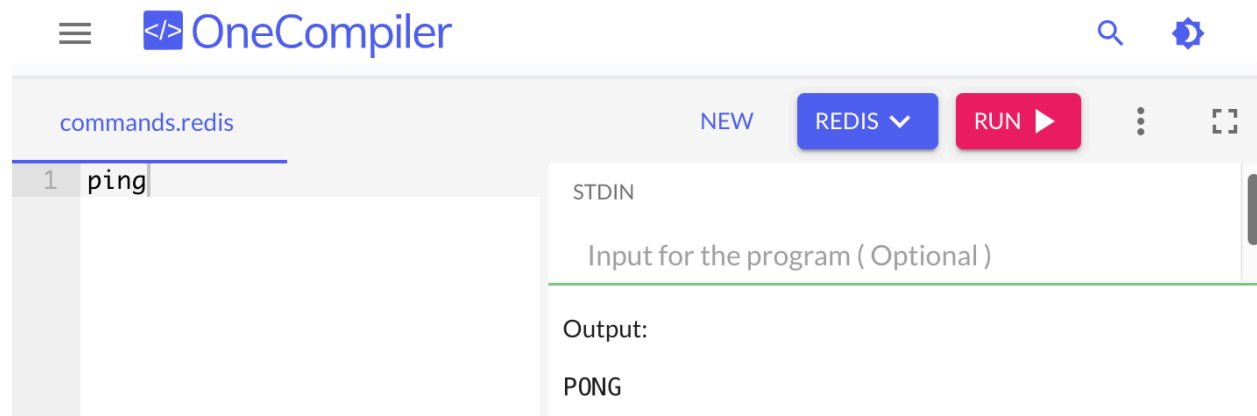
Tasks:

- **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://onecompiler.com/redis/> 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. Click the RUN button to see the results:



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

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
```

Output:

OK

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

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

```
set message hello
```

```
get message
```

Output:

OK

hello

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

```
get greeting
```

Output:

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

```
set message hello
```

```
get message
```

Output:

OK

```
set message goodbye
```

```
get message
```

hello

OK

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.

<pre>set message hello get message set message 'good morning' nx get message set message 'good morning' xx get message</pre>	<p>Output:</p> <pre>OK hello hello OK good morning</pre>
--	--

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

<pre>set message hello get message del message get message</pre>	<p>Output:</p> <pre>OK hello 1</pre>
--	--------------------------------------

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 the expiration time is reached, the key will be accessible.

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

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

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

TTL command returns -2 if the key does not exist.

```
set mission 'This message will self-destruct in 30 seconds' ex 30
get mission
ttl mission
get mission
set message 'hello'
ttl message
ttl greeting
```

Output:

```
OK
This message will self-destruct in 30 seconds
30
This message will self-destruct in 30 seconds
OK
-1
-2
```

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

	Output:
set message hello	OK
ttl message	-1
expire message 10	1
ttl message	10

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 expire set in milliseconds:

	Output:
set message hello	OK
ttl message	-1
pexpire message 30251	1
pttl message	30251
ttd message	30

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.

	Output:
set counter 100	OK
incr counter	101
incr counter	102
incrby counter 50	152
decr counter	151
decrby counter 20	131

<pre>set salary 256.75 incr salary incrbyfloat salary 20.6 incrbyfloat salary -12.5</pre>	<pre>Output: OK ERR value is not an integer or out of range 277.350000000000000000000000000001 264.850000000000000000000000000001</pre>
---	---

	Output:
set salary 256.75	OK
exists salary	1
exists price	0

- String
- List
- Set
- Hash
- Zset

```
set test 10
type test
```

Output:

OK
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 countries 'Czech Republic'  
lpush countries Mexico  
rpush countries Italy  
lpush countries Japan  
lpush countries Nigeria
```

Output:

```
1  
2  
3  
4  
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.

```
rpush countries 'Czech Republic'
lpush countries Mexico
rpush countries Italy
lpush countries Japan
lpush countries Nigeria
lrange countries 0 -1
```

Output:

1
2
3
4
5

Nigeria
Japan
Mexico
Czech Republic
Italy

```
lrange countries -1 0
```

```
lrange countries 0 2
```

Nigeria
Japan
Mexico

```
lrange countries 3 3
```

Czech Republic

You can push multiple elements into a list in a single call:

```
rpush countries 'Czech Republic'
lpush countries Mexico
rpush countries Italy
lpush countries Japan
lpush countries Nigeria
lrange countries 0 -1
```

```
lrange countries -1 0
```

```
lrange countries 0 2
```

```
lrange countries 3 3
```

```
lpush countries Spain 'United States' Egypt
```

```
rpush countries India Germany Australia
```

```
lrange countries 0 -1
```

Output:

1
2
3
4
5

Nigeria
Japan
Mexico
Czech Republic
Italy

Nigeria
Japan
Mexico
Czech Republic

8

11

Egypt
United States
Spain
Nigeria
Japan
Mexico
Czech Republic
Italy
India
Germany
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:

The diagram illustrates the state of a Redis list named 'countries' after a series of operations. On the left, the commands are listed: `lrange countries 0 -1` (initial view), `lrange countries -1 0`, `lrange countries 0 2`, `lrange countries 3 3`, `lpush countries Spain 'United States' Egypt`, `rpush countries India Germany Australia`, `lrange countries 0 -1` (after push), `lpop countries`, `rpop countries`, `rpopt countries`, and `lrange countries 0 -1` (final view). On the right, the list elements are shown in a vertical stack. Red boxes and arrows highlight the elements being popped. The first three red boxes (Egypt, Australia, Germany) are connected by arrows to the `lpop countries` command. The next three red boxes (United States, Spain, Nigeria) are connected by arrows to the `rpopt countries` command. The final red box (Mexico) is connected by an arrow to the `rpopt countries` command. The final list state, after all operations, is: Czech Republic, Italy, Nigeria, Japan, Mexico, Czech Republic, 8, 11, Egypt, United States, Spain, Nigeria, Japan, Mexico, Czech Republic, Italy, India, Germany, Australia, Egypt, Australia, Germany, United States, Spain, Nigeria, Japan, Mexico, Czech Republic, Italy, India.

```
lrange countries 0 -1
lrange countries -1 0
lrange countries 0 2
lrange countries 3 3
lpush countries Spain 'United States' Egypt
rpush countries India Germany Australia
lrange countries 0 -1
lpop countries
rpopt countries
rpopt countries
lrange countries 0 -1
```

Czech Republic
Italy
Nigeria
Japan
Mexico
Czech Republic
8
11
Egypt
United States
Spain
Nigeria
Japan
Mexico
Czech Republic
Italy
India
Germany
Australia
Egypt
Australia
Germany
United States
Spain
Nigeria
Japan
Mexico
Czech Republic
Italy
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.

```

rpush countries 'Czech Republic'
lpush countries Mexico
rpush countries Italy
lpush countries Japan
lpush countries Nigeria
lrange countries 0 -1

lrange countries -1 0

lrange countries 0 2

lrange countries 3 3
lpush countries Spain 'United States' Egypt
rpush countries India Germany Australia
lrange countries 0 -1

lpop countries
rpop countries
rpop countries

lrange countries 0 -1

ltrim countries 0 6
lrange countries 0 -1

```

```

11
Egypt
United States
Spain
Nigeria
Japan
Mexico
Czech Republic
Italy
India
Germany
Australia
Egypt
Australia
Germany
United States
Spain
Nigeria
Japan
Mexico
Czech Republic
Italy
India
OK
United States
Spain
Nigeria
Japan
Mexico
Czech Republic
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:

```

rpush countries 'Czech Republic'
lpush countries Mexico
rpush countries Italy
lpush countries Japan
lpush countries Nigeria
lrange countries 0 -1

lrange countries -1 0

lrange countries 0 2

lrange countries 3 3
lpush countries Spain 'United States' Egypt
rpush countries India Germany Australia
lrange countries 0 -1

lpop countries
rpop countries
rpop countries

lrange countries 0 -1

ltrim countries 0 6
lrange countries 0 -1

lpush countries France
ltrim countries 0 6
lrange countries 0 -1

```

```

Germany
Australia
Egypt
Australia
Germany
United States
Spain
Nigeria
Japan
Mexico
Czech Republic
Italy
India
OK
United States
Spain
Nigeria
Japan
Mexico
Czech Republic
Italy
8
OK
France
United States
Spain
Nigeria
Japan
Mexico
Czech Republic

```

What is the **TYPE** of **countries**? The **LLen** command returns the number of elements in a list

```

rpush countries 'Czech Republic'    1
lpush countries Mexico              2
rpush countries Italy                3
type countries                      list
llen countries                      3

```

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.

Task 4. Data types: Set

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

```
sadd fruits lemon  
smembers fruits
```

Output:

```
3  
apple  
banana  
orange  
1  
apple  
banana  
orange  
lemon
```

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

```
sadd fruits apple banana orange  
smembers fruits
```

```
sadd fruits lemon  
smembers fruits
```

```
sismember fruits lemon  
sismember fruits pear
```

Output:

```
3  
apple  
banana  
orange  
1  
apple  
banana  
orange  
lemon  
1  
0
```

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

```
sadd fruits apple banana orange lemon  
sadd store apple melon mango lime banana  
sinter fruits store
```

Output:

```
4  
5  
apple  
banana
```

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

```
sadd fruits apple banana orange lemon
sadd store apple melon mango lime banana
sunion fruits store
```

Output:

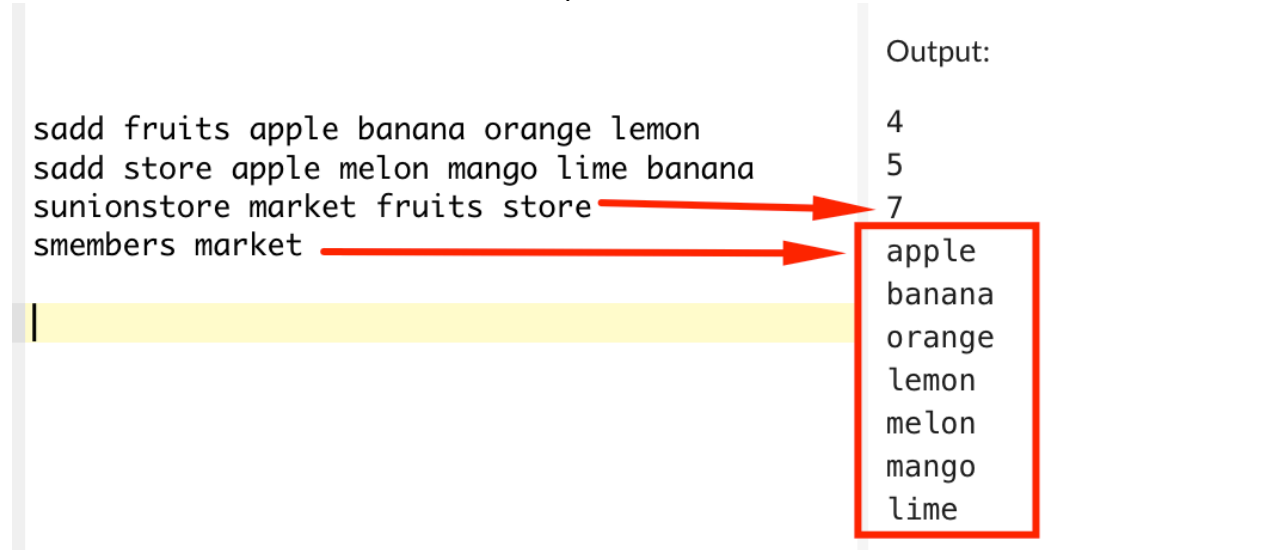
```
4
5
apple
banana
orange
lemon
melon
mango
lime
```

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.

```
sadd fruits apple banana orange lemon
sadd store apple melon mango lime banana
sunionstore market fruits store
smembers market
```

Output:

```
4
5
7
apple
banana
orange
lemon
melon
mango
lime
```



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.

```
sadd fruits apple banana orange lemon
sadd store apple melon mango lime banana
sunionstore market fruits store
```

```
srandmember market
srandmember market
srandmember market 4
srandmember market -4
```

Output:

```
4
5
7
```

```
lime
lemon
```

```
apple
melon
mango
lime
```

```
mango
mango
lime
apple
```

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

```
sadd fruits apple banana orange lemon
sadd store apple melon mango lime banana
sdiff fruits store
```

```
sdiff store fruits
```

Output:

```
4
5
```

```
orange
lemon
```

```
melon
mango
lime
```

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

```
sadd fruits apple banana orange lemon
sadd store apple melon mango lime banana
sdiffstore supermarket fruits store
smembers supermarket
```

Output:

```
4
5
2
```

```
orange
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:

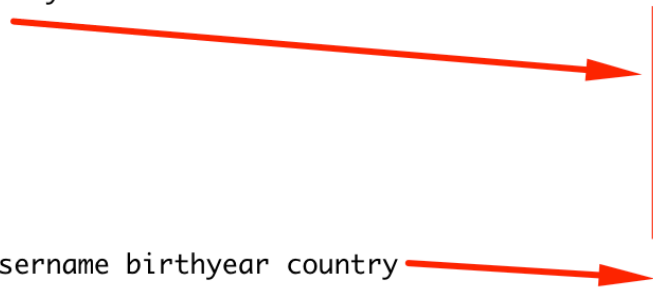
```
hset user:1000 username luis birthyear 1983 verified true
hget user:1000 username
hget user:1000 birthyear
hgetall user:1000
```

Output:

```
3
luis
1983
username
luis
birthyear
1983
verified
true
```

hmget user:1000 username birthyear country

```
luis
1983
```



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

```
hset user:1000 username luis birthyear 1983 verified true
hincrby user:1000 birthyear 10
hincrby user:1000 birthyear -5
```

Output:

```
3
1993
1988
```

Task 6. Data types: Zset

Sorted sets are another data type. They combine capabilities of Sets and Hashes. 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.

	Output:
<code>zadd hackers 1940 'Alan Key'</code>	1
<code>zadd hackers 1957 'Sophie Wilson'</code>	1
<code>zadd hackers 1953 'Richard Stallman'</code>	1
<code>zadd hackers 1949 'Anita Borg'</code>	1
<code>zadd hackers 1965 'Yukihiro Matsumoto'</code>	1
<code>zadd hackers 1914 'Hedy Lamarr'</code>	1
<code>zadd hackers 1916 'Claude Shannon'</code>	1
<code>zadd hackers 1969 'Linus Torvalds'</code>	1
<code>zadd hackers 1912 'Alan Turing'</code>	1

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.

```
zadd hackers 1940 'Alan Key'
zadd hackers 1957 'Sophie Wilson'
zadd hackers 1953 'Richard Stallman'
zadd hackers 1949 'Anita Borg'
zadd hackers 1965 'Yukihiro Matsumoto'
zadd hackers 1914 'Hedy Lamarr'
zadd hackers 1916 'Claude Shannon'
zadd hackers 1969 'Linus Torvalds'
zadd hackers 1912 'Alan Turing'
```

```
zrange hackers 0 -1
```

Output:

```
1
1
1
1
1
1
1
1
1
1
Alan Turing
Hedy Lamarr
Claude Shannon
Alan Key
Anita Borg
Richard Stallman
Sophie Wilson
Yukihiro Matsumoto
Linus Torvalds
```

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

```
zadd hackers 1940 'Alan Key'
zadd hackers 1957 'Sophie Wilson'
zadd hackers 1953 'Richard Stallman'
zadd hackers 1949 'Anita Borg'
zadd hackers 1965 'Yukihiro Matsumoto'
zadd hackers 1914 'Hedy Lamarr'
zadd hackers 1916 'Claude Shannon'
zadd hackers 1969 'Linus Torvalds'
zadd hackers 1912 'Alan Turing'
```

```
zrevrange hackers 0 -1
```

Output:

```
1
1
1
1
1
1
1
1
1
1
Linus Torvalds
Yukihiro Matsumoto
Sophie Wilson
Richard Stallman
Anita Borg
Alan Key
Claude Shannon
Hedy Lamarr
Alan Turing
```

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

	Output:
zadd hackers 1940 'Alan Key'	1
zadd hackers 1957 'Sophie Wilson'	1
zadd hackers 1953 'Richard Stallman'	1
zadd hackers 1949 'Anita Borg'	1
zadd hackers 1965 'Yukihiro Matsumoto'	1
zadd hackers 1914 'Hedy Lamarr'	1
zadd hackers 1916 'Claude Shannon'	1
zadd hackers 1969 'Linus Torvalds'	1
zadd hackers 1912 'Alan Turing'	1
zrange hackers 0 -1 withscores	Alan Turing 1912 Hedy Lamarr 1914 Claude Shannon 1916 Alan Key 1940 Anita Borg 1949 Richard Stallman 1953 Sophie Wilson 1957 Yukihiro Matsumoto 1965 Linus Torvalds 1969

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:

zadd hackers 1940 'Alan Key'	1
zadd hackers 1957 'Sophie Wilson'	1
zadd hackers 1953 'Richard Stallman'	1
zadd hackers 1949 'Anita Borg'	1
zadd hackers 1965 'Yukihiro Matsumoto'	1
zadd hackers 1914 'Hedy Lamarr'	1
zadd hackers 1916 'Claude Shannon'	1
zadd hackers 1969 'Linus Torvalds'	1
zadd hackers 1912 'Alan Turing'	1
zrangebyscore hackers -inf 1950	Alan Turing Hedy Lamarr Claude Shannon Alan Key Anita Borg

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

<pre>zadd hackers 1940 'Alan Key' zadd hackers 1957 'Sophie Wilson' zadd hackers 1953 'Richard Stallman' zadd hackers 1949 'Anita Borg' zadd hackers 1965 'Yukihiro Matsumoto' zadd hackers 1914 'Hedy Lamarr' zadd hackers 1916 'Claude Shannon' zadd hackers 1969 'Linus Torvalds' zadd hackers 1912 'Alan Turing' zremrangebyscore hackers 1940 1960 zrange hackers 0 -1</pre>	<p>Output:</p> <pre>1 1 1 1 1 1 1 1 1 1 4 Alan Turing Hedy Lamarr Claude Shannon Yukihiro Matsumoto Linus Torvalds</pre>
--	--

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

<pre>zadd hackers 1940 'Alan Key' zadd hackers 1957 'Sophie Wilson' zadd hackers 1953 'Richard Stallman' zadd hackers 1949 'Anita Borg' zadd hackers 1965 'Yukihiro Matsumoto' zadd hackers 1914 'Hedy Lamarr' zadd hackers 1916 'Claude Shannon' zadd hackers 1969 'Linus Torvalds' zadd hackers 1912 'Alan Turing' zremrangebyscore hackers 1940 1960 zrange hackers 0 -1 zrank hackers 'Claude Shannon' zrank hackers 'John Doe'</pre>	<p>Output:</p> <pre>1 1 1 1 1 1 1 1 1 1 4 Alan Turing Hedy Lamarr Claude Shannon Yukihiro Matsumoto Linus Torvalds 2</pre>
---	--

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

```
zadd hackers 1940 'Alan Key'
zadd hackers 1957 'Sophie Wilson'
zadd hackers 1953 'Richard Stallman'
zadd hackers 1949 'Anita Borg'
zadd hackers 1965 'Yukihiro Matsumoto'
zadd hackers 1914 'Hedy Lamarr'
zadd hackers 1916 'Claude Shannon'
zadd hackers 1969 'Linus Torvalds'
zadd hackers 1912 'Alan Turing'
```

```
zremrangebyscore hackers 1940 1960
zrange hackers 0 -1
```

```
zrevrank hackers 'Hedy Lamarr'
```

Output:

```
1
1
1
1
1
1
1
1
1
1
4
Alan Turing
Hedy Lamarr
Claude Shannon
Yukihiro Matsumoto
Linus Torvalds
3
```

Let's add the following instructions:

```
ZADD hackersv2 0 'Sophie Wilson' 0 'Alan Kay' 0 'Richard Stallman'
zrange hackersv2 0 -1
```

Output:

```
3
Alan Kay
Richard Stallman
Sophie Wilson
```

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

```
ZADD hackersv2 0 'Sophie Wilson' 0 'Alan Kay' 0 'Richard Stallman'
set message "hello"
keys *
```

Output:

```
3
OK
hackersv2
message
```

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"
```