# Exercise – Redis 101

For students who cannot use Docker on your machine, refer to this guideline to install Redis

<https://stackjava.com/redis/huong-dan-cai-dat-redis-server-tren-windows.html>

Installation msi can be found here: <https://github.com/microsoftarchive/redis/releases/tag/win-3.2.100>

Go to command line and type > redis-cli to begin

### Installation

Installation on Windows is Next next as normal

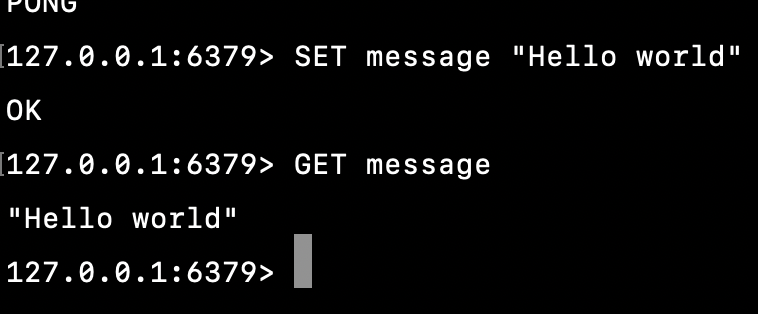
## 2. Data types In Redis

**(\*): Full documentation at https://redis.io/topics/data-types**

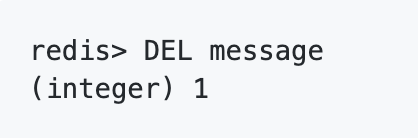
# 2.1. STRING

* GET/SET/DEL:
* GET – GET a key to a string/number
* SET – SET a key to a string/number
* EXISTS – Sees if a key EXISTS or not
* DEL – DELete a key

GET / SET



DEL: To delete specific key, use



Excercises

1. Set the key “sherlock:para1” to the first sentence of the “Adventures of Sherlock Holmes”, the paragraph that begins ‘To Sherlock Holmes she is always THE woman.’ Text is available from  <http://www.gutenberg.org/cache/epub/1661/pg1661.txt>). Confirm this, by GETting the paragraph back out again.
2. Continue setting keys of the form “sherlock:paraXXXX”, where XXXX is the number of the paragraph, based on text above. Retain the code you use here, as you will be able to adapt it for later exercises.

## Handling expire key

SETNX **–** set a key if not exist. In that case, it is equal to [SET](https://redis.io/commands/set). When key already holds a value, no operation is performed

KEYS pattern – returns all the keys that match the given pattern

TYPE – returns the type of value associated with a key

Expiration/persisting:

EXPIRE key timeout – Set a timeout on key

EXPIREAT key when – As EXPIRE above, but the timeout is set as an absolute UNIX timestamp (seconds since January 1, 1970).

SETEX – SET a key to a string, with an expiration (a Time-To-Live)

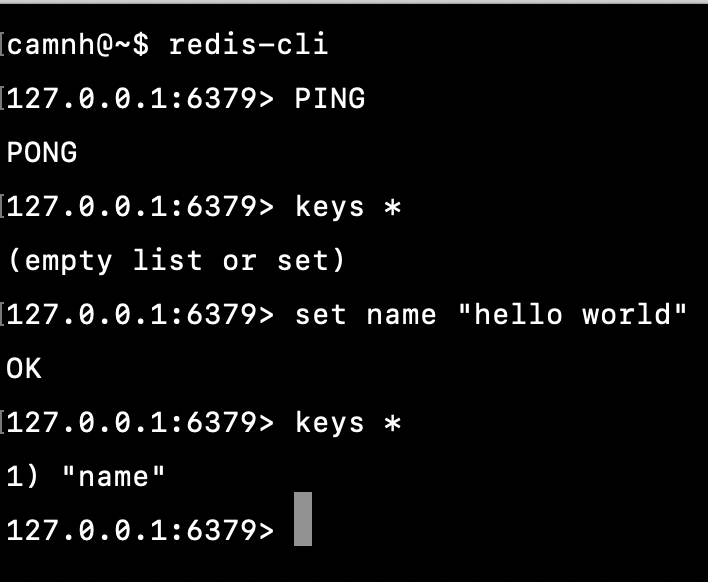
PERSIST key – Will remove any expiration associated with this key, giving it timeout of

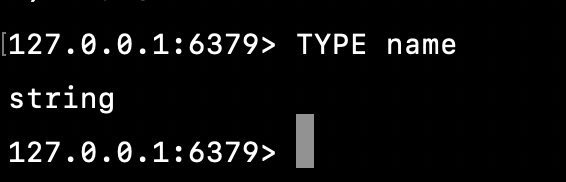
-1, thus persisting it.

TTL key – How many more seconds will key exist before being removed.

**Handling keys:**

An expensive, but useful command is the KEYS command. It allows you to find any key in the Redis instance that match a given pattern. The most basic pattern is just a wildcard – ‘\*’. This will attempt to return all the keys in the current db - (connect to a used Redis instance)





**Expiration and Persistence**

By default, keys you create in Redis will not expire. There are a number of situations where having keys exist temporarily is a useful tool and Redis provides a number of commands that are, frankly, wonderful for most tasks that need it.

There are three key commands used to set expirations: EXPIRE, EXPIREAT and SETEX:

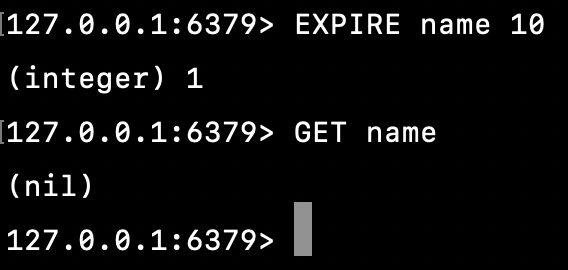
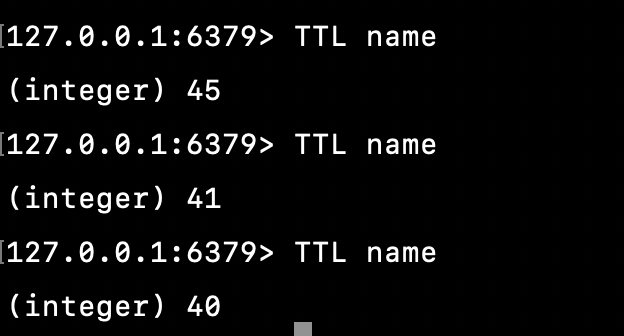


Figure 3: (cause the ‘name’’ key to expire in 10 seconds)

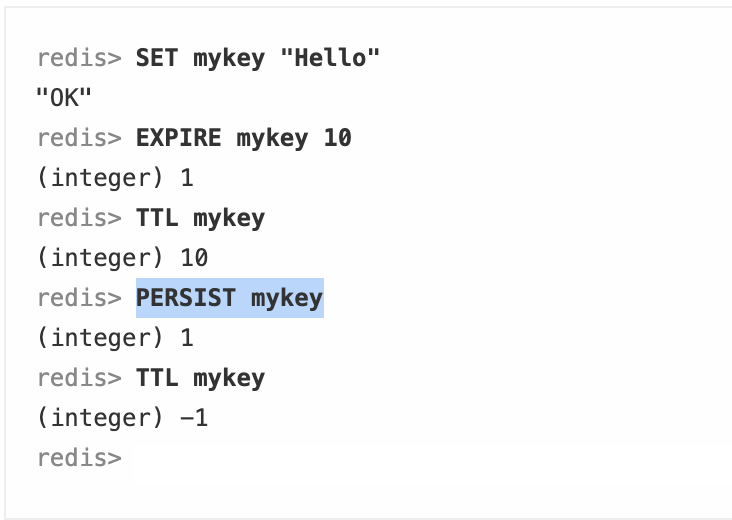
**Checking expiration times – TTL**

TTL is a handy command to check how long keys have to ‘live’. For example:



**Removing expirations – PERSIST**

Quite simply, this will remove any expiration associated with a key and will do nothing if the key is already permanent (TTL = -1)



Exercises

* 1. Consider (and if desired, build) a system that makes sure that a user of a web app can only vote on a poll once every 30 seconds. During 30 seconds, user cannot vote the same question again. If 30 seconds passed, user can re-vote.

If check with TTL command, value -2 means key is not exist, -1 means key is exist but expiration time is -1

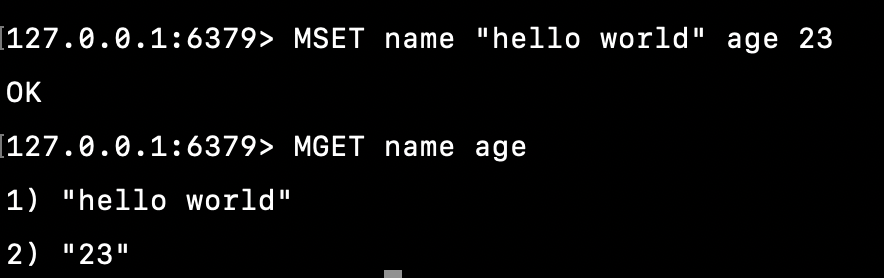
Hint: Using SETNX together with EXPIRE command.

**BATCH GET/SET**

MGET – GET the values for a number of keys simultaneously.

MSET – SET the values for a number of keys simultaneously.

MSETNX – As MSET above, but will only succeed when the keys referenced do not already exist.



**Exercise**

1. Adapt the Sherlock paragraph setting code you wrote previously (in section 1 of this) to set the paragraphs in configurable batches, using MSET.

## Commands acting on numbers

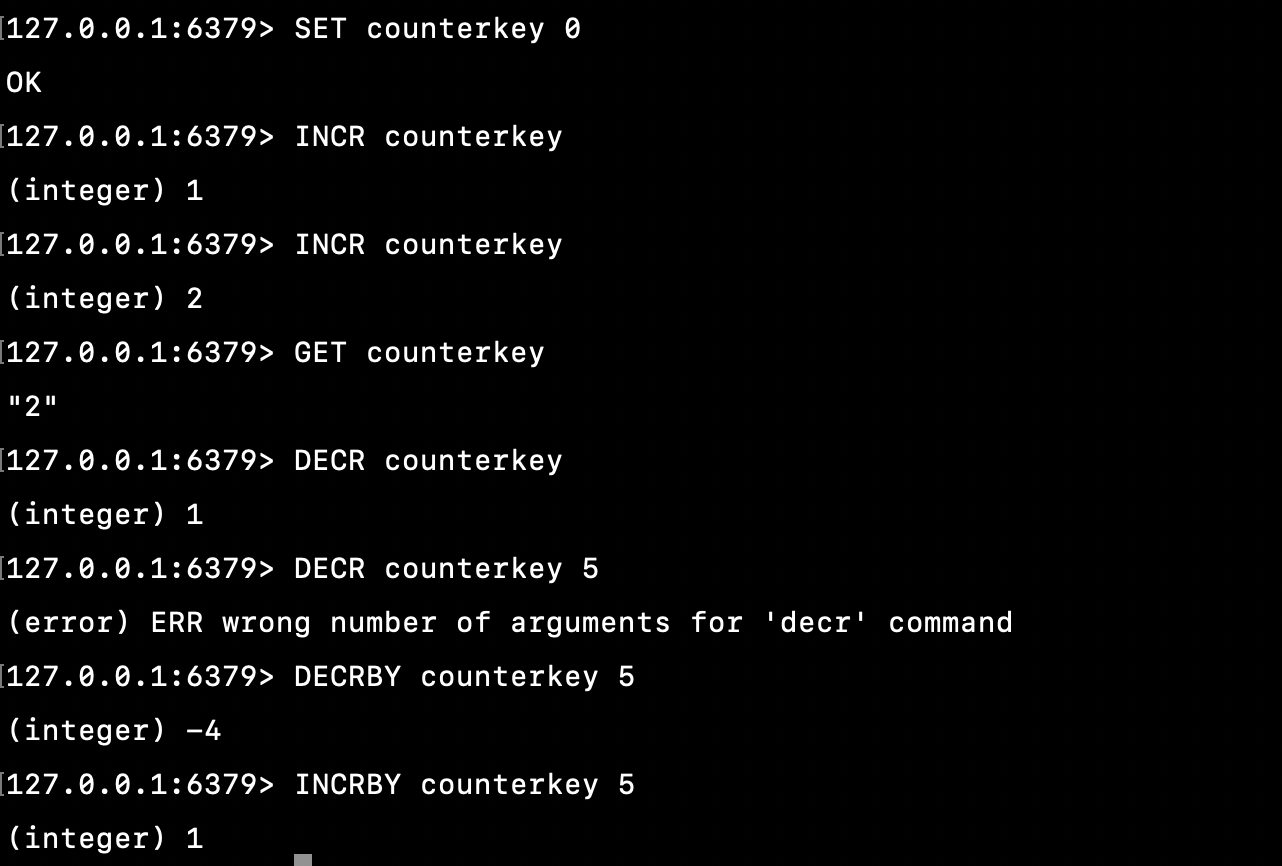
Simple maths:

INCR key – “key = key + 1”

INCRBY key integer – “key = key + integer”

DECR key – “key = key – 1”

DECRBY key integer – “key = key – integer”



Excercises

1. A client wants to have a click counter on their specific website, with a counter per each page. How would you build it using Redis?

# HASH

Hash is useful when deal with object since storage is optimized when using Hash. For example, storing user information includes first\_name, last\_name, address inside Hash. Normally, Hash is used to make statistics of something.

**Basic Hash commands**

**Commands covered here:**

HSET key field value - SET 'field' to be equal to the “value” in the hash 'key' --> "key.field = value"

HGET key field - GET the value of 'field' in the hash 'key'

HGETALL key - GET all the field:value pairs held within the hash 'key'

HDEL key field - DEL 'field' from the hash 'key'

HSETNX - as HSET above, but only succeeds if 'field' doesn't already exist.

**Basic Hash Usage:**

Hashes were introduced into Redis in one of the later 2.x versions. Before this, many applications used conventions to impose a hash-like hierarchy on the keys in the Redis db. You may still see documentation or code that uses commands of the sort:

redis> SET users:10001:passwordhash "md5:....."

OK

Redis doesn't ascribe any particular behavior to the colon or structure in the key used here. It's just another key to Redis. However, it helps segregate keys into groups. One way of utilizing hashes to improve this might be to do the following:

redis> HSET user:10001 passwordhash "...."

OK

In other words, use a hash per user to contain all the user-specific data. Note that the hash key doesn't need to exist before using it in this way. Redis will instantiate an empty hash and associate it with the key you use automatically.

To get back the passwordhash from this example, you can use the command HGET:

redis> HGET user:10001 passwordhash

"......."

Placing related information into a single hash has certain benefits, such as the following:

**Getting all the data from a hash:**

redis> HSET myhash field1 "Hello"

(integer) 1

redis> HSET myhash field2 "World"

(integer) 1

redis> HGETALL myhash

1) "field1"

2) "Hello"

3) "field2"

4) "World"

Even though this appears to return the fields in the same order they were issued, note that this is a hash and that you cannot trust the field-value pairs to be returned in any particular order.

**Deleting fields:**

The ordinary DEL command will remove the entire hash:

redis> DEL myhash

OK

redis> EXISTS myhash

(integer) 0

Use the related command, HDEL, to remove a single field from a hash:

redis> HDEL myhash field1

OK

## More Hash Commands

HEXISTS key field - Does 'field' exist in the hash 'key'?

HKEYS key - Get all the 'fields' from the hash 'key'

HVALS key - Get all the values from the hash 'key'

HLEN key - Get the number of fields in the hash 'key' (not the actual size of the values it contains.)

HMGET key field1 field2 … - Batch retrieve the field values from the hash 'key'

HMSET key field1 value1 field2 value2 …- Batch set the field values for the hash 'key'

HINCRBY key field increment - INCRement the field value in hash 'key'

## Batch Hash functions

HMSET and HMGET are two commands that can really speed up transactions with the server, especially when you are doing large numbers of lookups and settings to one or more hashes.

**Exercises:**

1. Store sample records in users in each hash like this below

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| id | name | Age | Gender | Nationality |
| 1 | camnh | 28 | Male | vietnam |
| 2 | minhpt | 24 | Male | korea |
| 3 | nganntt | 23 | Female | china |

Based on table above, enter commands performs listed requirements

* Print all data in hash (no need to specify keys to show)
* Print name of hash key of record id = 1
* Increase age of record Id = 3 from 23 to 24.
* Get all keys for record id = 2
* Unset person name value where id = 3
* Delete person record in hash where id = 3

# LIST

## Pushing and Popping values

The linked list structure in Redis is particularly useful and fast, for a given type of operation. Pushing objects onto either end of a list is uniformly fast - it doesn't matter if there are 2 or 2 million items in a given list, the time to append or prefix a value to it is uniform.

What it isn't, is an array. Retrieval by index or by range is not nearly as fast as you might expect if this were an indexed array of some sort.

Each list has a 'left' and a 'right', corresponding to either end of the list, and most of the basic commands that work on lists are prefixed by either an 'L' or an 'R' respectively, to let you know which end they work on.

Commands covered here:

LPUSH list\_key value - Push the value onto the left of the list

RPUSH list\_key value - As LPUSH, but pushes the item onto the right of the list

LPOP list\_key - Removes or 'pop's the leftmost item of the list\_key list

RPOP list\_key - As LPOP, but acts on the right.

RPOPLPUSH - A cunning command which pops a value from the right of one list and pushes it onto the left of another as a single, atomic action.

## Basic actions:

Creating a list is often done by using either LPUSH or RPUSH and having Redis auto-create one.

redis> LPUSH mylist "Welcome"

(integer) 1 <---- Note that the response of a PUSH is the length of the list.

redis> LPUSH mylist "to"

(integer) 2

redis> LPUSH mylist "the"

(integer) 3

redis> LPUSH mylist "jungle"

(integer) 4

You may dimly remember data-structure ideas like FIFO/FILO, queues, stacks and deques and these are all easily implementable with redis lists:

For example:

Using the above list - we pushed onto the lefthand side, so to use it as a queue - FIFO remember - we have to pop from the right:

redis> RPOP mylist

"Welcome"

redis> RPOP mylist

"to"

redis> RPOP mylist

"the"

redis> RPOP mylist

"jungle"

redis> RPOP mylist

(nil)

Note that this would work equally if we used RPUSH and LPOP, pushing onto the right and popping from the left. Also, the (nil) response reappears here, indicating that the list is empty. This command returns immediately - non-blocking - and so can be used for periodic polling and so on.

A stack is another structure that is readily implementable - FILO, first-in, last-out. The only thing to remember is to POP from the same side that you are PUSHing onto:

redis> LPUSH stack "Cool"

(integer) 1

redis> LPUSH stack "are"

(integer) 2

redis> LPUSH stack "Stacks"

(integer) 3

redis> LPOP stack

"Stacks"

redis> LPOP stack

"are"

redis> LPOP stack

"Cool"

The linked list in Redis is a double-ended list, which is the 'deque' as mentioned above.

## Cunning command: RPOPLPUSH

In a single atomic action, it will RPOP a value from one list, and LPUSH it onto another. It cannot (or rather, really shouldn't) fail in between. It is very effective when used to distribute tasks. A worker process can use its own private work list, claiming items to work on from the main task list using RPOPLPUSH. Once the task is complete, the worker should pass on the task with a report to a logger, and can also send a complete receipt to the process asking for the work to be done.

For example:

Main task queue:

(task descriptions in the list)

redis> LPUSH jobqueue "Task encoded with JSON or XML or something else."

(integer) 2003004 <-- lots of jobs, such as lookup X or upload 'Foo.png'

or

(task descriptions as a reference)

redis> SET task\_reference:1000 "Task description"

OK

redis> LPUSH jobqueue task\_reference:1000

(integer) 2001...

Each worker (of which you can run many of) is given an worker ID of some sort, eg worker\_1, \_2, etc.

To claim a job to work on (if you are 'worker\_1'):

redis> RPOPLPUSH jobqueue worker\_1:jobs

"Task encoded wi...."

Once job is complete, the task - from 'worker\_1:jobs' in this case - can be passed onto a logger for example, or simply RPOP'd off and ignored.

By minimising the points where the task is not held in a persistent store, you minimise the decisions you have to make about how robust you will handle the task descriptions in the worker code. Treating the Redis store as the canonical version, and the jobs held in the worker process's memory as copies of this is a useful way to think about it.

## Common usage pattern:

One common pattern with lists worth mentioning here is the use of lists for arbitrary ordering of references, where the references are commonly keys in Redis. For example:

redis> SET comment:10000 "....."

OK

redis> SET comment:10003 "..... #dev8d"

OK

redis> SET comment:10005 "....."

OK

redis> SET comment:10004 "..... #dev8d"

OK

... and so on ...

redis> LPUSH dev8d\_thread comment:10003

(integer) 1

redis> LPUSH dev8d\_thread comment:10004

(integer) 2

More usefully, the values of the keys being referenced will be more interesting types of things, like hashes, lists or sets. This allows the 'objects' (comments, users, things more generally) to be recorded once, but indexed and referred to many times.

## List utility commands

A number of utlity commands for lists exist, some obvious in use and some that should be used sparingly.

LLEN list\_key - Returns the length of the list.

## LRANGE

LRANGE <list> start end - A useful command, when used with care. The server responds with the range of values from the list, as you may expect. However, there are some important caveats to be aware of:

* Consistency with range functions in various programming languages

Note that if you have a list of numbers from 0 to 100, LRANGE list 0 10 will return 11 elements, that is, the rightmost item is included.

* Out-of-range indexes

Out of range indexes will not produce an error. If start is larger than the end of the list, an empty list is returned. If stop is larger than the actual end of the list, Redis will treat it like the last element of the list.

redis> RPUSH mylist "one"

(integer) 1

redis> RPUSH mylist "two"

(integer) 2

redis> RPUSH mylist "three"

(integer) 3

Start and end set to same value - returns just that value. If the last number is negative, count from right to left

redis> LRANGE mylist 0 0

1) "one"

Minus numbers work from the right end of the list, as opposed to the left end. Note that the indexes start at 0, so the end '2' refers to the third and last element of the list.

redis> LRANGE mylist -3 2

1) "one"

2) "two"

3) "three"

An example of the out-of-range edge case:

redis> LRANGE mylist -100 100

1) "one"

2) "two"

3) "three"

redis> LRANGE mylist 5 10

(empty list or set)

LINDEX list\_key value - Get a value from the list at the given index along from the left. As the structure is not an array but a list, this is a much more expensive operation than working on the elements at either end of the list.

Important to note that if you use LPUSH, then LINDEX might work counter-intuitively to how you might expect, as it also works from the lefthand side:

redis> LPUSH m 1

(integer) 1

redis> LPUSH m 2

(integer) 2

redis> LPUSH m 3

(integer) 3

List is [3, 2, 1] not the other way around. An index of -1 results in the last element of the list, or the 'first-in':

redis> LINDEX m -1

"1"

redis> LINDEX m -2

"2"

redis> LINDEX m -3

"3"

redis> LINDEX m 1

"2"

redis> LINDEX m 2

"1"

LREM (from official documentation: <http://redis.io/commands/lrem> )

Removes the first count occurrences of elements equal to value from the list stored at key. The count argument influences the operation in the following ways:

count > 0: Remove elements equal to value moving from head to tail.

count < 0: Remove elements equal to value moving from tail to head.

count = 0: Remove all elements equal to value.

For example, LREM list -2 "hello" will remove the last two occurrences of "hello" in the list stored at list.

Note that non-existing keys are treated like empty lists, so when key does not exist, the command will always return 0.

redis> RPUSH mylist "hello"

(integer) 1

redis> RPUSH mylist "hello"

(integer) 2

redis> RPUSH mylist "foo"

(integer) 3

redis> RPUSH mylist "hello"

(integer) 4

redis> LREM mylist -2 "hello"

(integer) 2

redis> LRANGE mylist 0 -1

1) "hello"

2) "foo"

LSET list\_key index value (also from official documentation <http://redis.io/commands/lrem>):

Sets the list element at index to value. For more information on the index argument, see LINDEX. For example, "LSET mylist 4 foo" is equivalent to "mylist[4] = 'foo'"

An error is returned for out of range indexes.

redis> RPUSH mylist "one"

(integer) 1

redis> RPUSH mylist "two"

(integer) 2

redis> RPUSH mylist "three"

(integer) 3

redis> LSET mylist 0 "four"

OK

redis> LSET mylist -2 "five"

OK

redis> LRANGE mylist 0 -1

1) "four"

2) "five"

3) "three"

## Exercises:

Create a list contain 3 numbers 1, 2, 3

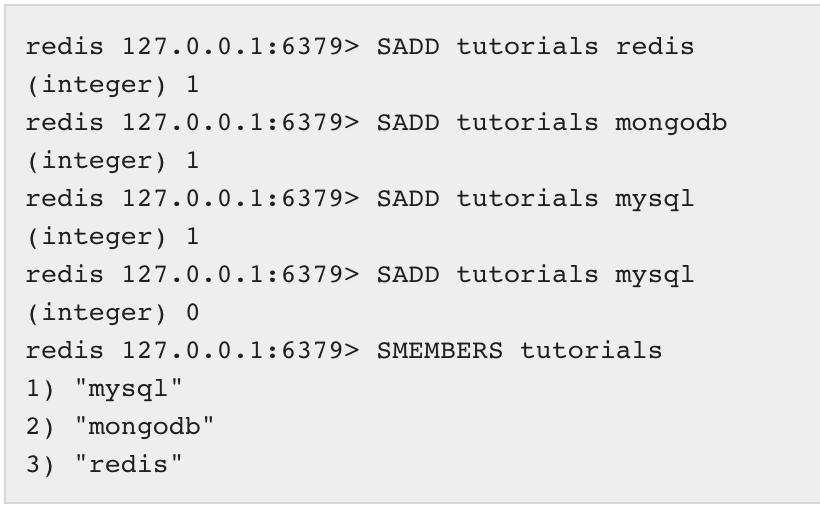
* Add new id = 4 with data of your choice to list created above
* Show number of ids stored in list (expected: 1 2 3 4)
* Remove 4 from list created above and print result (expected: 1 2 3)
* Add id = 5 into head of the list (expected: 5 1 2 3)
* Pop out id = 5 out of list
* Get items in list (expected: 1 2 3)
* Get number of items in list (expected 3)
* Get element name with index = 2 (expected 3)

# SET

Redis Sets are an unordered collection of unique strings. Unique means sets do not allow repetition of data in a key. It’s suitable for unique things such as IP Address statistics.

In Redis set add, remove, and test for the existence of members in O(1) (constant time regardless of the number of elements contained inside the Set). The maximum length of a list is 232 - 1 elements (4294967295, more than 4 billion of elements per set).

A problem can use SET in Redis is to track IP address of user visiting a given blog post.



Below are methods for a set

|  |  |
| --- | --- |
| **Sr.No** | **Command & Description** |
| 1 | [SADD key member1 [member2]](https://www.tutorialspoint.com/redis/sets_sadd.htm)  Adds one or more members to a set |
| 2 | [SCARD key](https://www.tutorialspoint.com/redis/sets_scard.htm)  Gets the number of members in a set |
| 3 | [SDIFF key1 [key2]](https://www.tutorialspoint.com/redis/sets_sdiff.htm)  Subtracts multiple sets |
| 4 | [SDIFFSTORE destination key1 [key2]](https://www.tutorialspoint.com/redis/sets_sdiffstore.htm)  Subtracts multiple sets and stores the resulting set in a key |
| 5 | [SINTER key1 [key2]](https://www.tutorialspoint.com/redis/sets_sinter.htm)  Intersects multiple sets |
| 6 | [SINTERSTORE destination key1 [key2]](https://www.tutorialspoint.com/redis/sets_sinterstore.htm)  Intersects multiple sets and stores the resulting set in a key |
| 7 | [SISMEMBER key member](https://www.tutorialspoint.com/redis/sets_sismember.htm)  Determines if a given value is a member of a set |
| 8 | [SMEMBERS key](https://www.tutorialspoint.com/redis/sets_sismember.htm)  Gets all the members in a set |
| 9 | [SMOVE source destination member](https://www.tutorialspoint.com/redis/sets_smove.htm)  Moves a member from one set to another |
| 10 | [SPOP key](https://www.tutorialspoint.com/redis/sets_spop.htm)  Removes and returns a random member from a set |
| 11 | [SRANDMEMBER key [count]](https://www.tutorialspoint.com/redis/sets_srandmember.htm)  Gets one or multiple random members from a set |
| 12 | [SREM key member1 [member2]](https://www.tutorialspoint.com/redis/sets_srem.htm)  Removes one or more members from a set |
| 13 | [SUNION key1 [key2]](https://www.tutorialspoint.com/redis/sets_sunion.htm)  Adds multiple sets |
| 14 | [SUNIONSTORE destination key1 [key2]](https://www.tutorialspoint.com/redis/sets_sunionstore.htm)  Adds multiple sets and stores the resulting set in a key |
| 15 | [SSCAN key cursor [MATCH pattern] [COUNT count]](https://www.tutorialspoint.com/redis/sets_sscan.htm)  Incrementally iterates set elements |

## Exercises:

Create a set to hold a list contain 3 numbers: 1,2,3

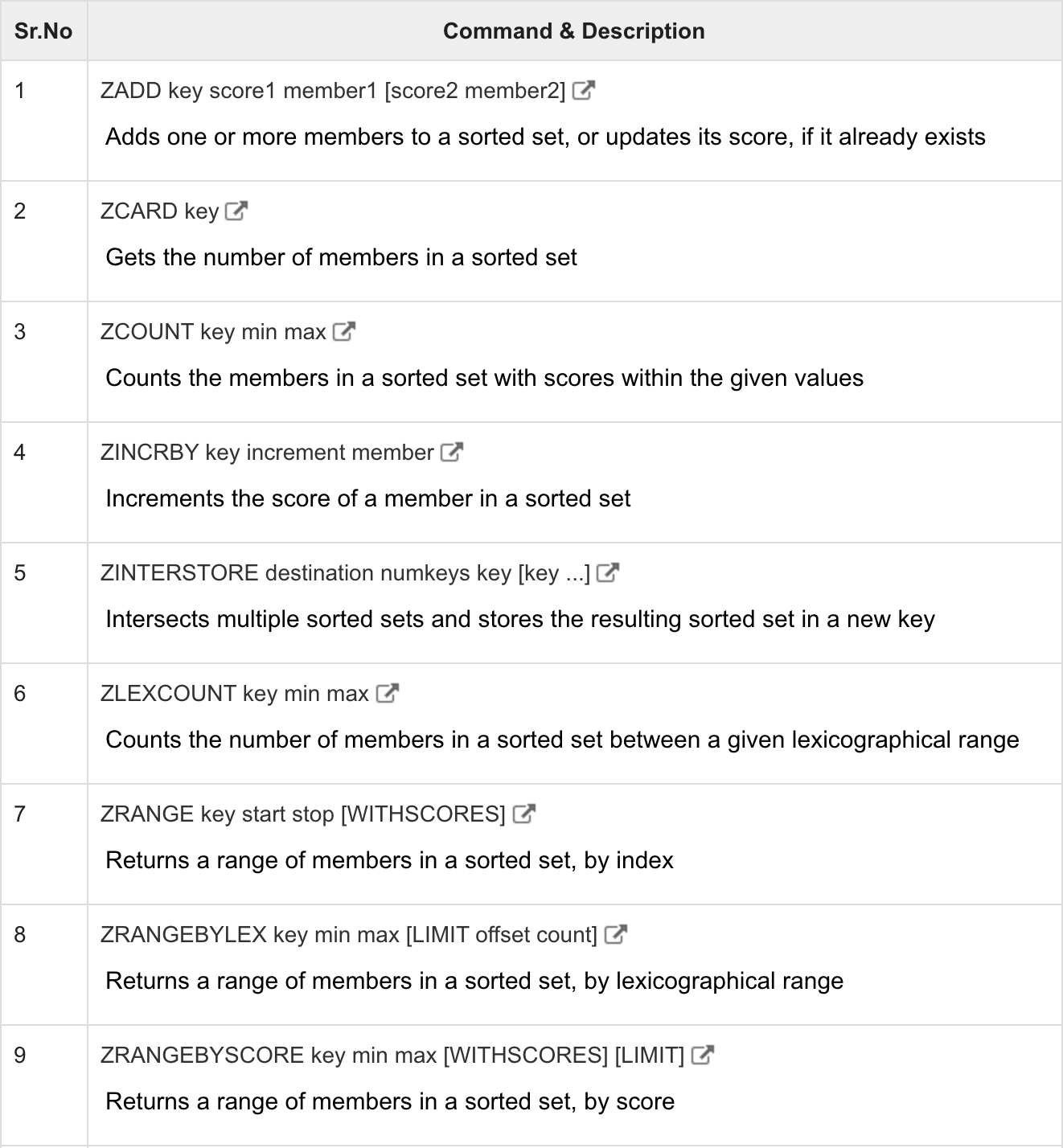
* Add new id = 4 to set created above
* Show number of ids stored in list (expected: 1 2 3 4)
* Remove set item 3 from list created above (expected: 1 2 4)
* Get number of items in sets
* Now create a new set called “users1” which include 3 elements 2, 4, 5. Write Redis query to get common values between “users” and “users1” (expected 2,4)
* Use set above, show the difference between 2 sets **users** and **users1** respectively
* Use set above, show the difference between 2 sets **users1** and **users** respectively. Is the result different from result generated from previous question

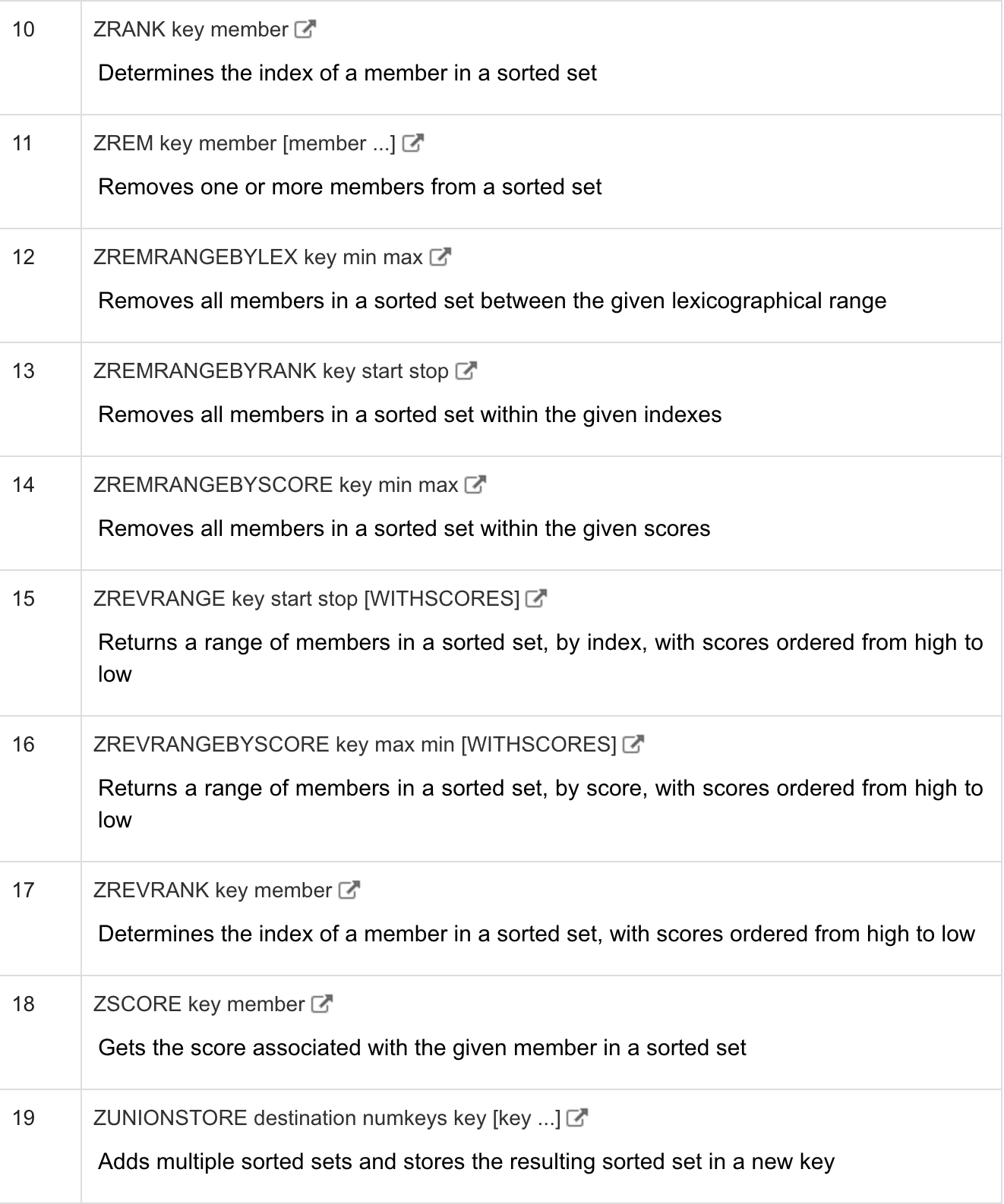
# ZSET (Sorted Set)

Sorted set characteristic is the same as set, however, it’s different to set that sorted set can be automatically sorted based on score. It’s suitable for a problem of sorting unique result (game leaderboard).

A problem which can use ZSET is

* Take a leaderboard in a massive online game, where every time a new score is submitted you update it using [ZADD](https://redis.io/commands/zadd). You can easily take the top users using [ZRANGE](https://redis.io/commands/zrange), you can also, given a user name, return its rank in the listing using [ZRANK](https://redis.io/commands/zrank). Using ZRANK and ZRANGE together you can show users with a score similar to a given user. All very *quickly*.







**Exercises:**

Store sample records in users in each set like this below

|  |  |
| --- | --- |
| id | score |
| 1 | 4 |
| 2 | 2 |
| 3 | 3 |

Based on table above, enter commands performs listed requirements

* Add those data to ZSET
* Add new item id = 4 and score 10
* Remove that item with id = 4
* Increase item’s score with id = 1 by 1 (expected result: 5)
* Get score of items with id = 1
* Show number of items in ZSet (expected number of items: 3)
* Print the list of items in normal and reverse order