To connect to Redis:
```
import redis
r = redis.Redis(host='localhost', port=6379, db=0)
```

String/Numbers:

```
# SET key value
# O(1)
# Set key to hold the string value. If key already holds a value, it is overwritten, regardless of its type.
r.set('mykey', 'Hello')
r.set('mykey2', 'World')

# GET key
# O(1)
# Get the string value of key. If the key does not exist the special value nil is returned.
r.get('mykey')

# MGET key [key ...]
# O(N)
# Returns the values of all specified keys. For every key that does not hold a string value or does not exist, the special value nil is returned.
r.mget(['mykey', 'mykey2', 'nonexistantkey'])

# INCR key
# O(1)
# Increments the number stored at key by one. If the key does not exist, it is set to 0 before performing the operation.
r.delete('mykey')
r.incr('mykey', 1)
r.get('mykey')
```

Generic:

```
# KEYS pattern
# O(N)
# Returns all keys matching pattern.
r.keys('*')
```

```
# EXPIRE key seconds
# O(1)
# Set a timeout on key. After the timeout has expired, the key will
automatically be deleted.
r.expire('mykey', 10)

# SCAN cursor [MATCH pattern] [COUNT count]
# O(1) for every call. O(N) for a complete iteration, including enough
command calls for the cursor to return back to 0.
# Iterates the set of keys in the currently selected Redis database.
r.delete('mykey', 'mykey2')
scanResult = r.scan(0, match='employee_profile:*')
r.scan(scanResult[0], match='employee_profile:*')

# DEL key [key ...]
# O(N)
# Removes the specified keys.
r.delete('employee_profile:viraj', 'employee_profile:terry',
        'employee_profile:sheera')

# TTL key
# O(1)
# Returns the remaining time to live of a key that has a timeout.
r.ttl('employee_profile:nicol')

# INFO [section ...]
# O(1)
# Returns information and statistics about the server, with the following
sections: server, clients, memory, persistence, stats, replication, cpu,
commandstats, latencystats, sentinel, cluster, modules, keyspace, errorstats
r.info('keyspace')
```

Hashes:

```
# HSET key field value [field value ...]

# O(N)
# Sets the specified fields to their respective values in the hash stored at key.
r.hset('h_employee_profile:nicol', 'name', 'Nicol')
```

```
# HGETALL key
# O(N)
# Returns all fields and values of the hash stored at key.
r.hgetall('h_employee_profile:nicol')

# HGET key field
# O(1)
# Returns the value associated with field in the hash stored at key.
r.hget('h_employee_profile:nicol', 'name')
```

Sets:

```
# SADD key member [member ...]
# O(N)
# Add the specified members to the set stored at key.
r.sadd('myset', 'Hello')
```

Sorted Sets:

```
# ZADD key score member [score member ...]
# O(log(N))
# Adds all the specified members with the specified scores to the sorted set
stored at key.
r.zadd('myzset', {'one': 1, 'two': 2, 'three': 3})

# ZRANGE key start stop [WITHSCORES]
# O(log(N)+M)
# Returns the specified range of elements in the sorted set stored at key.
r.zrange('myzset', 0, -1, withscores=True)
r.zrange('myzset', 0, -1)
```

Lists:

```
# LPOP key [count]
# O(N)
# Removes and returns the first element(s) of the list stored at key.
r.rpush('mylist', 'one', 'two', 'three', 'four', 'five')
r.lpop('mylist')
```

```
r.lpop('mylist', 2)

# LRANGE key start stop
# O(S+N)
# Returns the specified elements of the list stored at key.
r.delete('mylist')
r.rpush('mylist', 'one', 'two', 'three', 'four', 'five')
r.lrange('mylist', 0, -1)
r.lrange('mylist', -3, 2)

# LPUSH key element [element ...]
# O(N)
# Inserts specified values at the head of the list stored at key.
r.delete('mylist')
r.lpush('mylist', 'world')
r.lpush('mylist', 'hello')
r.lrange('mylist', 0, -1)
```

Streams:

```
# XADD key field value [field value ...]
# O(1) for new entries, O(N) when trimming where N is the number of evicted
values
# Appends the specified stream entry to the stream at the specified key.
r.xadd('temperatures:us-ny:10007',
      {'temp_f': 87.2, 'pressure': 29.69, 'humidity': 46})
r.xadd('temperatures:us-ny:10007',
      {'temp_f': 83.1, 'pressure': 29.21, 'humidity': 46.5})
r.xadd('temperatures:us-ny:10007',
      {'temp_f': 81.9, 'pressure': 28.37, 'humidity': 43.7})

# XREAD [COUNT count] [BLOCK milliseconds] STREAMS key [key ...] ID [ID
...]
# Read data from one or multiple streams, only returning entries with an ID
greater than the last received ID reported by the caller.
r.xread({'temperatures:us-ny:10007': '0-0'})
```

JSON

```
# JSON.SET key path value
# O(M+N) where M is the original size and N is the new size
# Set the JSON value at path in key.
r.json().set('employee_profile:nicol', '.', {
    'name': 'nicol', 'age': 24, 'single': True, 'skills': []})
r.json().set('employee_profile:nicol', '$.name', 'Nicol')

# JSON.GET key [path [path ...]]
# O(N)
# Return the value at path in JSON serialized form
r.json().get('employee_profile:nicol', '.')

# JSON.ARRAPPEND key [path] value [value ...]
# O(1) for each value added, O(N) for multiple values added where N is the
size of the key
# Append the value(s) to the array at path in key after the last element in the
array.
r.json().set('employee_profile:nicol', '$.skills', [])
r.json().arrappend('employee_profile:nicol', '$.skills', 'python')
r.json().get('employee_profile:nicol', '$.skills')

# JSON.ARRINDEX key path value [start [stop]]
# O(N)
# Search for the first occurrence of a JSON value in an array.
r.json().arrindex('employee_profile:nicol', '$.skills', 'python')
r.json().arrindex('employee_profile:nicol', '$.skills', 'java')
```

Search and Query

```
try:
    r.ft('idx-employees').dropindex()
except:
    pass

# FT.CREATE index [ON HASH | JSON] [PREFIX count prefix [prefix ...]]
SCHEMA field_name [AS alias] TEXT | TAG | NUMERIC | GEO | VECTOR |
GEOSHAP [SORTABLE [UNF]] [NOINDEX] [ field_name [AS alias] TEXT |
TAG | NUMERIC | GEO | VECTOR | GEOSHAPE [ SORTABLE [UNF]]
[NOINDEX] ...]
```

```
# O(K) where K is the number of fields in the document, O(N) for keys in the
keyspace
# Creates a new search index with the given specification.
schema = (TextField('$.name', as_name='name', sortable=True),
NumericField('$.age', as_name='age', sortable=True),
        TagField('$.single', as_name='single'), TagField('$.skills[*]',
as_name='skills'))

r.ft('idx-employees').create_index(schema, definition=IndexDefinition(
   prefix=['employee_profile:'], index_type=IndexType.JSON))

# FT.INFO index
# O(1)
# Return information and statistics on the index.
r.ft('idx-employees').info()

# FT.SEARCH index query
# O(N)
# Search the index with a textual query, returning either documents or just ids
r.ft('idx-employees').search('Nicol')
r.ft('idx-employees').search("@single:{false}")
r.ft('idx-employees').search("@skills:{python}")
r.ft('idx-employees').search(Query("*").add_filter(NumericFilter('age', 30, 40)))
r.json().arrappend('employee_profile:karol', '$.skills', 'python', 'java', 'c#')
r.ft('idx-employees').search(Query("@skills:{java}, @skills:{python}"))

# FT.AGGREGATE index query
# O(1)
# Run a search query on an index, and perform aggregate transformations on
the results, extracting statistics etc from them
r.ft('idx-
employees').aggregate(aggregations.AggregateRequest("*").group_by('@age'
,

reducers.count().alias('count')).sort_by("@age")).rows

r.ft('idx-
employees').aggregate(aggregations.AggregateRequest("@skills:{python}").gr
oup_by('@skills',

reducers.tolist('@name').alias('names'))).rows
```