Part 53
-
Redis

Version: 2019-12-10

How to Use Redis With Python

In this tutorial, you'll learn how to use Python with Redis, which is a lightning fast in-memory key-value store that can be used for anything from A to Z.
This tutorial is built for the Python programmer who may have zero to little Redis experience. We'll tackle two tools at once and introduce both Redis itself as well as one of its Python client libraries, redis-py.
redis-py (which you import as just redis) is one of many Python clients for Redis, but it has the distinction of being billed as "currently the way to go for Python" by the Redis developers themselves. It lets you call Redis commands from Python, and get back familiar Python objects in return.

Installing Redis

First, download the Redis source code as a tarball:

```
$ redisurl="http://download.redis.io/redis-stable.tar.gz"
$ curl -s -o redis-stable.tar.gz $redisurl
```

Next, switch over to root and extract the archive's source code to `/usr/local/lib/`:

```
$ sudo su root
$ mkdir -p /usr/local/lib/
$ chmod a+w /usr/local/lib/
$ tar -C /usr/local/lib/ -xzf redis-stable.tar.gz
```

Optionally, you can now remove the archive itself:

```
$ rm redis-stable.tar.gz
```

This will leave you with a source code repository at `/usr/local/lib/redis-stable/`. Redis is written in C, so you'll need to compile, link, and install with the make utility:

```
$ cd /usr/local/lib/redis-stable/
$ make && make install
```

Using `make install` does two actions:

1. The first make command compiles and links the source code.
2. The make install part takes the binaries and copies them to `/usr/local/bin/` so that you can run them from anywhere (assuming that `/usr/local/bin/` is in PATH).

Here are all the steps so far:

```
$ redisurl="http://download.redis.io/redis-stable.tar.gz"
$ curl -s -o redis-stable.tar.gz $redisurl
$ sudo su root
$ mkdir -p /usr/local/lib/
$ chmod a+w /usr/local/lib/
$ tar -C /usr/local/lib/ -xzf redis-stable.tar.gz
$ rm redis-stable.tar.gz
$ cd /usr/local/lib/redis-stable/
$ make && make install
```

At this point, take a moment to confirm that Redis is in your PATH and check its version:

```
$ redis-cli --version
redis-cli 5.0.3
```

If your shell can't find redis-cli, check to make sure that `/usr/local/bin/` is on your PATH environment variable, and add it if not.
In addition to `redis-cli`, `make install` actually leads to a handful of different executable files (and one symlink) being placed at `/usr/local/bin/`:

```
$ # A snapshot of executables that come bundled with Redis
$ ls -hFG /usr/local/bin/redis-* | sort
/usr/local/bin/redis-benchmark*
/usr/local/bin/redis-check-aof*
/usr/local/bin/redis-check-rdb*
/usr/local/bin/redis-cli*
/usr/local/bin/redis-sentinel@
/usr/local/bin/redis-server*
```

While all of these have some intended use, the two you'll probably care about most are redis-cli and redis-server, which we'll outline shortly. But before we get to that, setting up some baseline configuration is in order.

Configuring Redis

Redis is highly configurable. While it runs fine out of the box, let's take a minute to set some bare-bones configuration options that relate to database persistence and basic security:

```
$ sudo su root
$ mkdir -p /etc/redis/
$ touch /etc/redis/6379.conf
```

Now, write the following to `/etc/redis/6379.conf`. We'll cover what most of these mean gradually throughout the tutorial:

```
# /etc/redis/6379.conf

port              6379
daemonize         yes
save              60 1
bind              127.0.0.1
tcp-keepalive     300
dbfilename        dump.rdb
dir               ./
rdbcompression    yes
```

Redis configuration is self-documenting, with the sample `redis.conf` file located in the Redis source for your reading pleasure. If you're using Redis in a production system, it pays to block out all distractions and take the time to read this sample file in full to familiarize yourself with the ins and outs of Redis and fine-tune your setup.
Some tutorials, including parts of Redis' documentation, may also suggest running the Shell script `install_server.sh` located in `redis/utils/install_server.sh`. You're by all means welcome to run this as a more comprehensive alternative to the above, but take note of a few finer points about `install_server.sh`:

- It will inject a fuller set of configuration options into /etc/redis/6379.conf.
- It will write a System V init script to /etc/init.d/redis_6379 that will let you do sudo service redis_6379 start.

The Redis quickstart guide also contains a section on a more proper Redis setup, but the configuration options above should be totally sufficient for this tutorial and getting started.

Security Note: A few years back, the author of Redis pointed out security vulnerabilities in earlier versions of Redis if no configuration was set. Redis 3.2 (the current version 5.0.3 as of March 2019) made steps to prevent this intrusion, setting the protected-mode option to yes by default.
We explicitly set bind 127.0.0.1 to let Redis listen for connections only from the localhost interface, although you would need to expand this whitelist in a real production server. The point of protected-mode is as a safeguard that will mimic this bind-to-localhost behavior if you don't otherwise specify anything under the bind option.

Ten Minutes to Redis

Getting Started

Redis has a client-server architecture and uses a request-response model. This means that you (the client) connect to a Redis server through TCP connection, on port 6379 by default. You request some action (like some form of reading, writing, getting, setting, or updating), and the server *serves* you back a response.

There can be many clients talking to the same server, which is really what Redis or any client-server application is all about. Each client does a (typically blocking) read on a socket waiting for the server response.

The cli in `redis-cli` stands for command line interface, and the server in `redis-server` is for, well, running a server. In the same way that you would run python at the command line, you can run `redis-cli` to jump into an interactive REPL (Read Eval Print Loop) where you can run client commands directly from the shell.

First, however, you'll need to launch `redis-server` so that you have a running Redis server to talk to. A common way to do this in development is to start a server at localhost (IPv4 address 127.0.0.1), which is the default unless you tell Redis otherwise. You can also pass `redis-server` the name of your configuration file, which is akin to specifying all of its key-value pairs as command-line arguments:

```
$ redis-server /etc/redis/6379.conf
31829:C 07 Mar 2019 08:45:04.030 # oO0OoO0OoO0Oo Redis is starting
oO0OoO0OoO0Oo
31829:C 07 Mar 2019 08:45:04.030 # Redis version=5.0.3, bits=64,
commit=00000000, modified=0, pid=31829, just started
31829:C 07 Mar 2019 08:45:04.030 # Configuration loaded
```

We set the daemonize configuration option to yes, so the server runs in the background. (Otherwise, use --daemonize yes as an option to redis-server.)

Now you're ready to launch the Redis REPL. Enter `redis-cli` on your command line. You'll see the server's `host:port` pair followed by a `>` prompt:

```
127.0.0.1:6379>
```

Here's one of the simplest Redis commands, PING, which just tests connectivity to the server and returns "PONG" if things are okay:

```
127.0.0.1:6379> PING
PONG
```

Redis commands are case-insensitive, although their Python counterparts are most definitely not.

Note**:** As another sanity check, you can search for the process ID of the Redis server with pgrep:

```
$ pgrep redis-server
26983
```

To kill the server, use `pkill redis-server` from the command line.

Redis as a Python Dictionary
Redis stands for Remote Dictionary Service. "You mean, like a Python dictionary?" you may ask.
Yes. Broadly speaking, there are many parallels you can draw between a Python dictionary (or generic hash table) and what Redis is and does:

- A Redis database holds `key:value` pairs and supports commands such as `GET`, `SET`, and `DEL`, as well as several hundred additional commands.
- Redis `keys` are always strings.
- Redis `values` may be a number of different data types. We'll cover some of the more essential value data types in this tutorial: string, list, hashes, and sets. Some advanced types include geospatial items and the new stream type.
- Many Redis commands operate in `constant O(1)` time, just like retrieving a value from a Python dict or any hash table.

Redis creator Salvatore Sanfilippo would probably not love the comparison of a Redis database to a plain-vanilla Python dict. He calls the project a "data structure server" (rather than a key-value store, such as memcached) because, to its credit, Redis supports storing additional types of *key:value* data types besides *string:string*. But for our purposes here, it's a useful comparison if you're familiar with Python's dictionary object.
Let's jump in and learn by example. Our first toy database (with ID 0) will be a mapping of `country:capital city`, where we use `SET` to set key-value pairs:

```
127.0.0.1:6379> SET Bahamas Nassau
OK
127.0.0.1:6379> SET Croatia Zagreb
OK
127.0.0.1:6379> GET Croatia
"Zagreb"
127.0.0.1:6379> GET Japan
(nil)
```

The corresponding sequence of statements in pure Python would look like this:

```
>>> capitals = {}
>>> capitals["Bahamas"] = "Nassau"
>>> capitals["Croatia"] = "Zagreb"
>>> capitals.get("Croatia")
'Zagreb'
>>> capitals.get("Japan")  # None
```

We use `capitals.get("Japan")` rather than `capitals["Japan"]` because Redis will return `nil` rather than an error when a key is not found, which is analogous to Python's None.
Redis also allows you to set and get multiple key-value pairs in one command, `MSET` and `MGET`, respectively:

```
127.0.0.1:6379> MSET Lebanon Beirut Norway Oslo France Paris
OK
127.0.0.1:6379> MGET Lebanon Norway Bahamas
1) "Beirut"
2) "Oslo"
3) "Nassau"
The closest thing in Python is with dict.update():
>>>
>>> capitals.update({
...     "Lebanon": "Beirut",
...     "Norway": "Oslo",
...     "France": "Paris",
... })
>>> [capitals.get(k) for k in ("Lebanon", "Norway", "Bahamas")]
['Beirut', 'Oslo', 'Nassau']
```

We use `.get()` rather than `.__getitem__()` to mimic Redis' behavior of returning a null-like value when no key is found.
As a third example, the `EXISTS` command does what it sounds like, which is to check if a key exists:

```
127.0.0.1:6379> EXISTS Norway
(integer) 1
127.0.0.1:6379> EXISTS Sweden
(integer) 0
```

Python has the in keyword to test the same thing, which routes to dict.__contains__(key):

```
>>> "Norway" in capitals
True
>>> "Sweden" in capitals
False
```

These few examples are meant to show, using native Python, what's happening at a high level with a few common Redis commands. There's no client-server component here to the Python examples, and redis-py has not yet entered the picture. This is only meant to show Redis functionality by example.
Here's a summary of the few Redis commands you've seen and their functional Python equivalents:

```
SET Bahamas Nassau
capitals["Bahamas"] = "Nassau"

GET Croatia
capitals.get("Croatia")

MSET Lebanon Beirut Norway Oslo France Paris
capitals.update(
    {
        "Lebanon": "Beirut",
        "Norway": "Oslo",
        "France": "Paris",
    }
)

MGET Lebanon Norway Bahamas
[capitals[k] for k in ("Lebanon", "Norway", "Bahamas")]

EXISTS Norway
"Norway" in capitals
```

The Python Redis client library, `redis-py`, that you'll dive into shortly in this article, does things differently. It encapsulates an actual TCP connection to a Redis server and sends raw commands, as bytes serialized using the REdis Serialization Protocol (RESP), to the server. It then takes the raw reply and parses it back into a Python object such as bytes, int, or even datetime.datetime.

Note: So far, you've been talking to the Redis server through the interactive **`redis-cli`** REPL. You can also issue commands directly, in the same way that you would pass the name of a script to the python executable, such as python `myscript.py`.

More Data Types in Python vs Redis
Before you fire up the redis-py Python client, it also helps to have a basic grasp on a few more Redis data types. To be clear, all Redis keys are strings. It's the value that can take on data types (or structures) in addition to the string values used in the examples so far.

A `hash` is a mapping of *string:string*, called `field-value` pairs, that sits under one top-level key:

```
127.0.0.1:6379> HSET realpython url "https://realpython.com/"
(integer) 1
127.0.0.1:6379> HSET realpython github realpython
(integer) 1
127.0.0.1:6379> HSET realpython fullname "Real Python"
(integer) 1
```

This sets three field-value pairs for one `key`, "realpython". If you're used to Python's terminology and objects, this can be confusing. A Redis hash is roughly analogous to a Python dict that is nested one level deep:

```
data = {
    "realpython": {
        "url": "https://realpython.com/",
        "github": "realpython",
        "fullname": "Real Python",
    }
}
```

Redis' fields are akin to the Python keys of each nested key-value pair in the inner dictionary above. Redis reserves the term `key` for the top-level database key that holds the hash structure itself.
Just like there's `MSET` for basic `string:string` key-value pairs, there is also `HMSET` for hashes to set multiple pairs *within* the hash value object:

```
127.0.0.1:6379> HMSET pypa url "https://www.pypa.io/" github pypa fullname
"Python Packaging Authority"
OK
127.0.0.1:6379> HGETALL pypa
1) "url"
2) "https://www.pypa.io/"
3) "github"
4) "pypa"
5) "fullname"
6) "Python Packaging Authority"
```

Using `HMSET` is probably a closer parallel for the way that we assigned data to a nested dictionary above, rather than setting each nested pair as is done with `HSET`.

Two additional value types are `lists` and `sets`, which can take the place of a hash or string as a Redis value. They are largely what they sound like, so I won't take up your time with additional examples. Hashes, lists, and sets each have some commands that are particular to that given data type, which are in some cases denoted by their initial letter:

- Hashes: Commands to operate on hashes begin with an `H`, such as `HSET`, `HGET`, or `HMSET`.
- Sets: Commands to operate on sets begin with an `S`, such as `SCARD`, which gets the number of elements at the set value corresponding to a given key.
- Lists: Commands to operate on lists begin with an `L` or `R`. Examples include `LPOP` and
- The L or R refers to which side of the list is operated on. A few list commands are also prefaced with a B, which means blocking. A blocking operation doesn't let other operations interrupt it while it's executing. For instance, BLPOP executes a blocking left-pop on a list structure.

Note**:** One noteworthy feature of Redis' list type is that it is a linked list rather than an array. This means that appending is O(1) while indexing at an arbitrary index number is O(N).
Here is a quick listing of commands that are particular to the string, hash, list, and set data types in Redis:

Type	**Commands**
Sets	SADD, SCARD, SDIFF, SDIFFSTORE, SINTER, SINTERSTORE, SISMEMBER, SMEMBERS, SMOVE, SPOP, SRANDMEMBER, SREM, SSCAN, SUNION, SUNIONSTORE
Hashes	HDEL, HEXISTS, HGET, HGETALL, HINCRBY, HINCRBYFLOAT, HKEYS, HLEN, HMGET, HMSET, HSCAN, HSET, HSETNX, HSTRLEN, HVALS
Lists	BLPOP, BRPOP, BRPOPLPUSH, LINDEX, LINSERT, LLEN, LPOP, LPUSH, LPUSHX, LRANGE, LREM, LSET, LTRIM, RPOP, RPOPLPUSH, RPUSH, RPUSHX
Strings	APPEND, BITCOUNT, BITFIELD, BITOP, BITPOS, DECR, DECRBY, GET, GETBIT, GETRANGE, GETSET, INCR, INCRBY, INCRBYFLOAT, MGET, MSET, MSETNX, PSETEX, SET, SETBIT, SETEX, SETNX, SETRANGE, STRLEN

This table isn't a complete picture of Redis commands and types. There's a smorgasbord of more advanced data types, such as geospatial items, sorted sets, and HyperLogLog. At the Redis commands page, you can filter by data-structure group. There is also the data types summary and introduction to Redis data types.
Since we're going to be switching over to doing things in Python, you can now clear your toy database with `FLUSHDB` and quit the redis-cli REPL:

```
127.0.0.1:6379> FLUSHDB
OK
127.0.0.1:6379> QUIT
```

This will bring you back to your shell prompt. You can leave redis-server running in the background, since you'll need it for the rest of the tutorial also.

Using redis-py: Redis in Python

First Steps

redis-py is a well-established Python client library that lets you talk to a Redis server directly through Python calls:

```
$ python -m pip install redis
```

Next, make sure that your Redis server is still up and running in the background. You can check with pgrep redis-server, and if you come up empty-handed, then restart a local server with redis-server `/etc/redis/6379.conf`.
Now, let's get into the Python-centric part of things. Here's the "hello world" of redis-py:

```
>>> import redis
>>> r = redis.Redis()
>>> r.mset({"Croatia": "Zagreb", "Bahamas": "Nassau"})
True
>>> r.get("Bahamas")
b'Nassau'
```

Redis, used in Line 2, is the central class of the package and the workhorse by which you execute (almost) any Redis command. The TCP socket connection and reuse is done for you behind the scenes, and you call Redis commands using methods on the class instance r.
Notice also that the type of the returned object, `b'Nassau'` in Line 6, is Python's `bytes` type, not `str`. It is `bytes` rather than `str` that is the most common return type across `redis-py`, so you may need to call `r.get("Bahamas").decode("utf-8")` depending on what you want to actually do with the returned.
Does the code above look familiar? The methods in almost all cases match the name of the Redis command that does the same thing. Here, you called `r.mset()` and `r.get()`, which correspond to `MSET` and `GET` in the native Redis API.
This also means that `HGETALL` becomes `r.hgetall()`, `PING` becomes `r.ping()`, and so on. There are a few exceptions, but the rule holds for the large majority of commands.
While the Redis command arguments usually translate into a similar-looking method signature, they take Python objects. For example, the call to `r.mset()` in the example above uses a Python `dict` as its first argument, rather than a sequence of bytestrings.
We built the Redis instance r with no arguments, but it comes bundled with a number of parameters if you need them:

```
# From redis/client.py
class Redis(object):
    def __init__(self, host='localhost', port=6379,
                 db=0, password=None, socket_timeout=None,
                 # …
```

You can see that the default *hostname:port* pair is `localhost:6379`, which is exactly what we need in the case of our locally kept redis-server instance.
The `db` parameter is the database number. You can manage multiple databases in Redis at once, and each is identified by an integer. The max number of databases is 16 by default. When you run just `redis-cli` from the command line, this starts you at database 0. Use the `-n` flag to start a new database, as in `redis-cli -n 5`.

Allowed Key Types
One thing that's worth knowing is that redis-py requires that you pass it keys that are `bytes`, `str`, `int`, or `float`. (It will convert the last 3 of these types to bytes before sending them off to the server.)
Consider a case where you want to use calendar dates as keys:

```
>>> import datetime
>>> today = datetime.date.today()
>>> visitors = {"dan", "jon", "alex"}
>>> r.sadd(today, *visitors)
Traceback (most recent call last):
# …

redis.exceptions.DataError: Invalid input of type: 'date'.
Convert to a byte, string or number first.
```

You'll need to explicitly convert the Python date object to str, which you can do with .isoformat():

```
>>> stoday = today.isoformat()  # Python 3.7+, or use str(today)
>>> stoday
'2019-03-10'
>>> r.sadd(stoday, *visitors)  # sadd: set-add
3
>>> r.smembers(stoday)
{b'dan', b'alex', b'jon'}
>>> r.scard(today.isoformat())
3
```

Example: PyHats.com
It's time to break out a fuller example. Let's pretend we've decided to start a lucrative website, PyHats.com, that sells outrageously overpriced hats to anyone who will buy them, and hired you to build the site.
You'll use Redis to handle some of the product catalog, inventorying, and bot traffic detection for PyHats.com.
It's day one for the site, and we're going to be selling three limited-edition hats. Each hat gets held in a Redis hash of field-value pairs, and the hash has a key that is a prefixed random integer , such as `hat:56854717`. Using the `hat:` prefix is Redis convention for creating a sort of namespace within a Redis database:

```
import random

random.seed(444)
hats = {f"hat:{random.getrandbits(32)}": i for i in (
    {
        "color": "black",
        "price": 49.99,
     "style": "fitted",
        "quantity": 1000,
        "npurchased": 0,
    },
    {
        "color": "maroon",
        "price": 59.99,
        "style": "hipster",
        "quantity": 500,
        "npurchased": 0,
    },
    {
        "color": "green",
        "price": 99.99,
        "style": "baseball",
        "quantity": 200,
        "npurchased": 0,
    })
}
```

Let's start with database 1 since we used database 0 in a previous example:

```
>>> r = redis.Redis(db=1)
```

To do an initial write of this data into Redis, we can use `.hmset()` (hash multi-set), calling it for each dictionary. The "multi" is a reference to setting multiple field-value pairs, where "field" in this case corresponds to a key of any of the nested dictionaries in hats:

```
>>> with r.pipeline() as pipe:
...     for h_id, hat in hats.items():
...         pipe.hmset(h_id, hat)
...     pipe.execute()
Pipeline<ConnectionPool<Connection<host=localhost,port=6379,db=1>>>
Pipeline<ConnectionPool<Connection<host=localhost,port=6379,db=1>>>
Pipeline<ConnectionPool<Connection<host=localhost,port=6379,db=1>>>
[True, True, True]

>>> r.bgsave()
True
```

The code block above also introduces the concept of Redis pipelining, which is a way to cut down the number of round-trip transactions that you need to write or read data from your Redis server. If you would have just called `r.hmset()` three times, then this would necessitate a back-and-forth round trip operation for each row written.

With a pipeline, all the commands are buffered on the client side and then sent at once, in one fell swoop, using `pipe.hmset()` in Line 3. This is why the three True responses are all returned at once, when you call `pipe.execute()` in Line 4. You'll see a more advanced use case for a pipeline shortly.

Note: The Redis docs provide an example of doing this same thing with the `redis-cli`, where you can pipe the contents of a local file to do mass insertion.
Let's do a quick check that everything is there in our Redis database:

```
>>> pprint(r.hgetall("hat:56854717"))
{b'color': b'green',
 b'npurchased': b'0',
 b'price': b'99.99',
 b'quantity': b'200',
 b'style': b'baseball'}

>>> r.keys()  # Careful on a big DB. keys() is O(N)
[b'56854717', b'1236154736', b'1326692461']
```

The first thing that we want to simulate is what happens when a user clicks *Purchase*. If the item is in stock, increase its `npurchased` by 1 and decrease its quantity (inventory) by 1. You can use `.hincrby()` to do this:

```
>>> r.hincrby("hat:56854717", "quantity", -1)
199
>>> r.hget("hat:56854717", "quantity")
b'199'
>>> r.hincrby("hat:56854717", "npurchased", 1)
1
```

Note: HINCRBY still operates on a hash value that is a string, but it tries to interpret the string as a base-10 64-bit signed integer to execute the operation.
This applies to other commands related to incrementing and decrementing for other data structures, namely INCR, INCRBY, INCRBYFLOAT, ZINCRBY, and HINCRBYFLOAT. You'll get an error if the string at the value can't be represented as an integer.
It isn't really that simple, though. Changing the `quantity` and `npurchased` in two lines of code hides the reality that a click, purchase, and payment entails more than this. We need to do a few more checks to make sure we don't leave someone with a lighter wallet and no hat:

- Step 1: Check if the item is in stock, or otherwise raise an exception on the backend.
- Step 2: If it is in stock, then execute the transaction, decrease the quantity field, and increase the npurchased field.
- Step 3: Be alert for any changes that alter the inventory in between the first two steps (a race condition).

Step 1 is relatively straightforward: it consists of an `.hget()` to check the available quantity.

Step 2 is a little bit more involved. The pair of increase and decrease operations need to be executed atomically: either both should be completed successfully, or neither should be (in the case that at least one fails).
With client-server frameworks, it's always crucial to pay attention to atomicity and look out for what could go wrong in instances where multiple clients are trying to talk to the server at once. The answer to this in Redis is to use a transaction block, meaning that either both or neither of the commands get through.
In redis-py, Pipeline is a transactional pipeline class by default. This means that, even though the class is actually named for something else (pipelining), it can be used to create a transaction block also.
In Redis, a transaction starts with `MULTI` and ends with `EXEC`:

```
1 127.0.0.1:6379> MULTI
2 127.0.0.1:6379> HINCRBY 56854717 quantity -1
3 127.0.0.1:6379> HINCRBY 56854717 npurchased 1
4 127.0.0.1:6379> EXEC
```

`MULTI` (Line 1) marks the start of the transaction, and `EXEC` (Line 4) marks the end. Everything in between is executed as one all-or-nothing buffered sequence of commands. This means that it will be impossible to decrement quantity (Line 2) but then have the balancing npurchased increment operation fail (Line 3).

Step 3 is the trickiest. Let's say that there is one lone hat remaining in our inventory. In between the time that User A checks the quantity of hats remaining and actually processes their transaction, User B also checks the inventory and finds likewise that there is one hat listed in stock. Both users will be allowed to purchase the hat, but we have 1 hat to sell, not 2, so we're on the hook and one user is out of their money. Not good.
Redis has a clever answer for the dilemma in Step 3: it's called optimistic locking, and is different than how typical locking works in an RDBMS such as PostgreSQL. Optimistic locking, in a nutshell, means that the calling function (client) does not acquire a lock, but rather monitors for changes in the data it is writing to *during the time it would have held a lock*. If there's a conflict during that time, the calling function simply tries the whole process again. You can effect optimistic locking by using the `WATCH` command (`.watch()` in redis-py), which provides a check-and-set behavior.

Let's introduce a big chunk of code and walk through it afterwards step by step. You can picture `buyitem()` as being called any time a user clicks on a *Buy Now* or *Purchase* button. Its purpose is to confirm the item is in stock and take an action based on that result, all in a safe manner that looks out for race conditions and retries if one is detected:

```
import logging
import redis

logging.basicConfig()

class OutOfStockError(Exception):
    """Raised when PyHats.com is all out of today's hottest hat"""

def buyitem(r: redis.Redis, itemid: int) -> None:
    with r.pipeline() as pipe:
        error_count = 0
        while True:
            try:
                # Get available inventory, watching for changes
                # related to this itemid before the transaction
                pipe.watch(itemid)
                nleft: bytes = r.hget(itemid, "quantity")
                if nleft > b"0":
                    pipe.multi()
                    pipe.hincrby(itemid, "quantity", -1)
                    pipe.hincrby(itemid, "npurchased", 1)
                    pipe.execute()
                    break
                else:
                    # Stop watching the itemid and raise to break out
                    pipe.unwatch()
                    raise OutOfStockError(
                        f"Sorry, {itemid} is out of stock!"
                    )
            except redis.WatchError:
                # Log total num. of errors by this user to buy this item,
                # then try the same process again of WATCH/HGET/MULTI/EXEC
                error_count += 1
```

```
                logging.warning(
                    "WatchError #%d: %s; retrying",
                    error_count, itemid
                )
    return None
```

The critical line occurs at Line 16 with `pipe.watch(itemid)`, which tells Redis to monitor the given itemid for any changes to its value. The program checks the inventory through the call to `r.hget(itemid, "quantity")`, in Line 17:

```
pipe.watch(itemid)
nleft: bytes = r.hget(itemid, "quantity")
if nleft > b"0":
    # Item in stock. Proceed with transaction.
```

If the inventory gets touched during this short window between when the user checks the item stock and tries to purchase it, then Redis will return an error, and redis-py will raise a WatchError (Line 30). That is, if any of the hash pointed to by itemid changes after the `.hget()` call but before the subsequent `.hincrby()` calls in Lines 20 and 21, then we'll re-run the whole process in another iteration of the while True loop as a result.
This is the "optimistic" part of the locking: rather than letting the client have a time-consuming total lock on the database through the getting and setting operations, we leave it up to Redis to notify the client and user only in the case that calls for a retry of the inventory check.
One key here is in understanding the difference between client-side and server-side operations:
`nleft = r.hget(itemid, "quantity")`
This Python assignment brings the result of `r.hget()` client-side. Conversely, methods that you call on pipe effectively buffer all of the commands into one, and then send them to the server in a single request:

```
pipe.multi()
pipe.hincrby(itemid, "quantity", -1)
pipe.hincrby(itemid, "npurchased", 1)
pipe.execute()
```

No data comes back to the client side in the middle of the transactional pipeline. You need to call `.execute()` (Line 19) to get the sequence of results back all at once.
Even though this block contains two commands, it consists of exactly one round-trip operation from client to server and back.
This means that the client can't immediately *use* the result of `pipe.hincrby(itemid, "quantity", -1)`, from Line 20, because methods on a Pipeline return just the pipe instance itself. We haven't asked anything from the server at this point. While normally `.hincrby()` returns the resulting value, you can't immediately reference it on the client side until the entire transaction is completed.
There's a catch-22: this is also why you can't put the call to `.hget()` into the transaction block. If you did this, then you'd be unable to know if you want to increment the `npurchased` field yet, since you can't get real-time results from commands that are inserted into a transactional pipeline.
Finally, if the inventory sits at zero, then we UNWATCH the item ID and raise an OutOfStockError (Line 27), ultimately displaying that coveted *Sold Out* page that will make our hat buyers desperately want to buy even more of our hats at ever more outlandish prices:

```
else:
    # Stop watching the itemid and raise to break out
    pipe.unwatch()
    raise OutOfStockError(
        f"Sorry, {itemid} is out of stock!"
    )
```

Here's an illustration. Keep in mind that our starting quantity is 199 for hat 56854717 since we called `.hincrby()` above. Let's mimic 3 purchases, which should modify the quantity and `npurchased` fields:

```
>>> buyitem(r, "hat:56854717")
>>> buyitem(r, "hat:56854717")
>>> buyitem(r, "hat:56854717")
>>> r.hmget("hat:56854717", "quantity", "npurchased")  # Hash multi-get
[b'196', b'4']
```

Now, we can fast-forward through more purchases, mimicking a stream of purchases until the stock depletes to zero. Again, picture these coming from a whole bunch of different clients rather than just one Redis instance:

```
>>> # Buy remaining 196 hats for item 56854717 and deplete stock to 0
>>> for _ in range(196):
...     buyitem(r, "hat:56854717")
>>> r.hmget("hat:56854717", "quantity", "npurchased")
[b'0', b'200']
```

Now, when some poor user is late to the game, they should be met with an OutOfStockError that tells our application to render an error message page on the frontend:

```
>>> buyitem(r, "hat:56854717")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 20, in buyitem
__main__.OutOfStockError: Sorry, hat:56854717 is out of stock!
```

Looks like it's time to restock.

Using Key Expiry
Let's introduce key expiry, which is another distinguishing feature in Redis. When you expire a key, that key and its corresponding value will be automatically deleted from the database after a certain number of seconds or at a certain timestamp.
In redis-py, one way that you can accomplish this is through `.setex()`, which lets you set a basic *string:string* key-value pair with an expiration:

```
>>> from datetime import timedelta

>>> # setex: "SET" with expiration
>>> r.setex(
...     "runner",
...     timedelta(minutes=1),
...     value="now you see me, now you don't"
... )
True
```

You can specify the second argument as a number in seconds or a timedelta object, as in Line 6 above. I like the latter because it seems less ambiguous and more deliberate.
There are also methods (and corresponding Redis commands, of course) to get the remaining lifetime (time-to-live) of a key that you've set to expire:

```
r.ttl("runner")  # "Time To Live", in seconds
58
r.pttl("runner")  # Like ttl, but milliseconds
54368
```

Below, you can accelerate the window until expiration, and then watch the key expire, after which `r.get()` will return None and `.exists()` will return 0:

```
r.get("runner")  # Not expired yet
b"now you see me, now you don't"

r.expire("runner", timedelta(seconds=3))  # Set new expire window
True
# Pause for a few seconds
r.get("runner")
r.exists("runner")  # Key & value are both gone (expired)
0
```

The table below summarizes commands related to key-value expiration, including the ones covered above. The explanations are taken directly from redis-py method docstrings:

Signature	Purpose
`r.setex(name, time, value)`	Sets the value of key name to value that expires in time seconds, where time can be represented by an int or a Python timedelta object
`r.psetex(name, time_ms, value)`	Sets the value of key name to value that expires in time_ms milliseconds, where time_ms can be represented by an int or a Python timedelta object
`r.expire(name, time)`	Sets an expire flag on key name for time seconds, where time can be represented by an int or a Python timedelta object
`r.expireat(name, when)`	Sets an expire flag on key name, where when can be represented as an int indicating Unix time or a Python datetime object
`r.persist(name)`	Removes an expiration on name
`r.pexpire(name, time)`	Sets an expire flag on key name for time milliseconds, and time can be represented by an int or a Python timedelta object
`r.pexpireat(name, when)`	Sets an expire flag on key name, where when can be represented as an int representing Unix time in milliseconds (Unix time * 1000) or a Python datetime object
`r.pttl(name)`	Returns the number of milliseconds until the key name will expire
`r.ttl(name)`	Returns the number of seconds until the key name will expire

PyHats.com, Part 2

A few days after its debut, PyHats.com has attracted so much hype that some enterprising users are creating bots to buy hundreds of items within seconds, which you've decided isn't good for the long-term health of your hat business.

Now that you've seen how to expire keys, let's put it to use on the backend of PyHats.com. We're going to create a new Redis client that acts as a consumer (or watcher) and processes a stream of incoming IP addresses, which in turn may come from multiple HTTPS connections to the website's server.

The watcher's goal is to monitor a stream of IP addresses from multiple sources, keeping an eye out for a flood of requests from a single address within a suspiciously short amount of time.

Some middleware on the website server pushes all incoming IP addresses into a Redis list with `.lpush()`. Here's a crude way of mimicking some incoming IPs, using a fresh Redis database:

```
>>> r = redis.Redis(db=5)
>>> r.lpush("ips", "51.218.112.236")
1
>>> r.lpush("ips", "90.213.45.98")
2
>>> r.lpush("ips", "115.215.230.176")
3
>>> r.lpush("ips", "51.218.112.236")
4
```

As you can see, `.lpush()` returns the length of the list after the push operation succeeds. Each call of `.lpush()` puts the IP at the beginning of the Redis list that is keyed by the string "ips". In this simplified simulation, the requests are all technically from the same client, but you can think of them as potentially coming from many different clients and all being pushed to the same database on the same Redis server.

Now, open up a new shell tab or window and launch a new Python REPL. In this shell, you'll create a new client that serves a very different purpose than the rest, which sits in an endless while True loop and does a blocking left-pop `BLPOP` call on the ips list, processing each address:

```
# New shell window or tab

import datetime
import ipaddress

import redis

# Where we put all the bad egg IP addresses
blacklist = set()
MAXVISITS = 15

ipwatcher = redis.Redis(db=5)

while True:
    _, addr = ipwatcher.blpop("ips")
    addr = ipaddress.ip_address(addr.decode("utf-8"))
    now = datetime.datetime.utcnow()
    addrts = f"{addr}:{now.minute}"
    n = ipwatcher.incrby(addrts, 1)
    if n >= MAXVISITS:
        print(f"Hat bot detected!:  {addr}")
        blacklist.add(addr)
    else:
        print(f"{now}:  saw {addr}")
    _ = ipwatcher.expire(addrts, 60)
```

Let's walk through a few important concepts.
The ipwatcher acts like a consumer, sitting around and waiting for new IPs to be pushed on the "ips" Redis list. It receives them as bytes, such as `b"51.218.112.236"`, and makes them into a more proper address object with the ipaddress module:

```
_, addr = ipwatcher.blpop("ips")
addr = ipaddress.ip_address(addr.decode("utf-8"))
```

Then you form a Redis string key using the address and minute of the hour at which the ipwatcher saw the address, incrementing the corresponding count by 1 and getting the new count in the process:

```
now = datetime.datetime.utcnow()
addrts = f"{addr}:{now.minute}"
n = ipwatcher.incrby(addrts, 1)
```

If the address has been seen more than `MAXVISITS`, then it looks as if we have a PyHats.com web scraper on our hands trying to create the next tulip bubble. Alas, we have no choice but to give this user back something like a dreaded 403 status code.
We use `ipwatcher.expire(addrts, 60)` to expire the *(address minute)* combination 60 seconds from when it was last seen. This is to prevent our database from becoming clogged up with stale one-time page viewers.

If you execute this code block in a new shell, you should immediately see this output:

```
2019-03-11 15:10:41.489214:  saw 51.218.112.236
2019-03-11 15:10:41.490298:  saw 115.215.230.176
2019-03-11 15:10:41.490839:  saw 90.213.45.98
2019-03-11 15:10:41.491387:  saw 51.218.112.236
```

The output appears right away because those four IPs were sitting in the queue-like list keyed by "ips", waiting to be pulled out by our ipwatcher. Using `.blpop()` (or the BLPOP command) will block until an item is available in the list, then pops it off. It behaves like Python's `Queue.get()`, which also blocks until an item is available.
Besides just spitting out IP addresses, our ipwatcher has a second job. For a given minute of an hour (minute 1 through minute 60), ipwatcher will classify an IP address as a hat-bot if it sends 15 or more `GET` requests in that minute.
Switch back to your first shell and mimic a page scraper that blasts the site with 20 requests in a few milliseconds:

```
for _ in range(20):
    r.lpush("ips", "104.174.118.18")
Finally, toggle back to the second shell holding ipwatcher, and you should
see an output like this:
2019-03-11 15:15:43.041363:  saw 104.174.118.18
2019-03-11 15:15:43.042027:  saw 104.174.118.18
2019-03-11 15:15:43.042598:  saw 104.174.118.18
2019-03-11 15:15:43.043143:  saw 104.174.118.18
2019-03-11 15:15:43.043725:  saw 104.174.118.18
2019-03-11 15:15:43.044244:  saw 104.174.118.18
2019-03-11 15:15:43.044760:  saw 104.174.118.18
2019-03-11 15:15:43.045288:  saw 104.174.118.18
2019-03-11 15:15:43.045806:  saw 104.174.118.18
2019-03-11 15:15:43.046318:  saw 104.174.118.18
2019-03-11 15:15:43.046829:  saw 104.174.118.18
2019-03-11 15:15:43.047392:  saw 104.174.118.18
2019-03-11 15:15:43.047966:  saw 104.174.118.18
2019-03-11 15:15:43.048479:  saw 104.174.118.18
Hat bot detected!:  104.174.118.18
Hat bot detected!:  104.174.118.18
Hat bot detected!:  104.174.118.18
Hat bot detected!:  104.174.118.18
Hat bot detected!:  104.174.118.18
Hat bot detected!:  104.174.118.18
```

Now, `Ctrl+C` out of the while True loop and you'll see that the offending IP has been added to your blacklist:

```
>>> blacklist
{IPv4Address('104.174.118.18')}
```

Can you find the defect in this detection system? The filter checks the minute as .minute rather than the *last 60 seconds* (a rolling minute). Implementing a rolling check to monitor how many times a user has been seen in the last 60 seconds would be trickier. There's a crafty solution using using Redis' sorted sets at ClassDojo. Josiah Carlson's *Redis in Action* also presents a more elaborate and general-purpose example of this section using an IP-to-location cache table.

Persistence and Snapshotting
One of the reasons that Redis is so fast in both read and write operations is that the database is held in memory (RAM) on the server. However, a Redis database can also be stored (persisted) to disk in a process called snapshotting. The point behind this is to keep a physical backup in binary format so that data can be reconstructed and put back into memory when needed, such as at server startup.
You already enabled snapshotting without knowing it when you set up basic configuration at the beginning of this tutorial with the save option:

```
# /etc/redis/6379.conf

port              6379
daemonize         yes
save              60 1
bind              127.0.0.1
tcp-keepalive     300
dbfilename        dump.rdb
dir               ./
rdbcompression    yes
```

The format is save <seconds> <changes>. This tells Redis to save the database to disk if both the given number of seconds and number of write operations against the database occurred. In this case, we're telling Redis to save the database to disk every 60 seconds if at least one modifying write operation occurred in that 60-second timespan. This is a fairly aggressive setting versus the sample Redis config file, which uses these three save directives:

```
# Default redis/redis.conf
save 900 1
save 300 10
save 60 10000
```

An RDB snapshot is a full (rather than incremental) point-in-time capture of the database. (RDB refers to a Redis Database File.) We also specified the directory and file name of the resulting data file that gets written:

```
# /etc/redis/6379.conf

port              6379
daemonize         yes
save              60 1
bind              127.0.0.1
tcp-keepalive     300
dbfilename        dump.rdb
dir               ./
rdbcompression    yes
```

This instructs Redis to save to a binary data file called dump.rdb in the current working directory of wherever redis-server was executed from:

```
$ file -b dump.rdb
data
```

You can also manually invoke a save with the Redis command `BGSAVE`:

```
127.0.0.1:6379> BGSAVE
Background saving started
```

The “BG” in BGSAVE indicates that the save occurs in the background. This option is available in a redis-py method as well:

```
>>> r.lastsave()  # Redis command: LASTSAVE
datetime.datetime(2019, 3, 10, 21, 56, 50)
>>> r.bgsave()
True
>>> r.lastsave()
datetime.datetime(2019, 3, 10, 22, 4, 2)
```

This example introduces another new command and method, `.lastsave()`. In Redis, it returns the Unix timestamp of the last DB save, which Python gives back to you as a datetime object. Above, you can see that the `r.lastsave()` result changes as a result of `r.bgsave()`. `r.lastsave()` will also change if you enable automatic snapshotting with the save configuration option.
To rephrase all of this, there are two ways to enable snapshotting:

1. Explicitly, through the Redis command `BGSAVE` or redis-py method `.bgsave()`
2. Implicitly, through the save configuration option (which you can also set with `.config_set()` in redis-py)

RDB snapshotting is fast because the parent process uses the fork() system call to pass off the time-intensive write to disk to a child process, so that the parent process can continue on its way. This is what the *background* in `BGSAVE` refers to.
There’s also `SAVE` (`.save()` in redis-py), but this does a synchronous (blocking) save rather than using `fork()`, so you shouldn’t use it without a specific reason.
Even though `.bgsave()` occurs in the background, it’s not without its costs. The time for `fork()` itself to occur can actually be substantial if the Redis database is large enough in the first place.
If this is a concern, or if you can’t afford to miss even a tiny slice of data lost due to the periodic nature of RDB snapshotting, then you should look into the append-only file (AOF) strategy that is an alternative to snapshotting. AOF copies Redis commands to disk in real time, allowing you to do a literal command-based reconstruction by replaying these commands.

Serialization Workarounds
Let's get back to talking about Redis data structures. With its hash data structure, Redis in effect supports nesting one level deep:

```
127.0.0.1:6379> hset mykey field1 value1
```

The Python client equivalent would look like this:

```
r.hset("mykey", "field1", "value1")
```

Here, you can think of "field1": "value1" as being the key-value pair of a Python dict, {"field1": "value1"}, while mykey is the top-level key:

Redis Command	Pure-Python Equivalent
`r.set("key", "value")`	`r = {"key": "value"}`
`r.hset("key", "field", "value")`	`r = {"key": {"field": "value"}}`

But what if you want the value of this dictionary (the Redis hash) to contain something other than a string, such as a list or nested dictionary with strings as values?
Here's an example using some JSON-like data to make the distinction clearer:

```
restaurant_484272 = {
    "name": "Ravagh",
    "type": "Persian",
    "address": {
        "street": {
            "line1": "11 E 30th St",
            "line2": "APT 1",
        },
        "city": "New York",
        "state": "NY",
        "zip": 10016,
    }
}
```

Say that we want to set a Redis hash with the key 484272 and field-value pairs corresponding to the key-value pairs from restaurant_484272. Redis does not support this directly, because restaurant_484272 is nested:

```
>>> r.hmset(484272, restaurant_484272)
Traceback (most recent call last):
# ...
redis.exceptions.DataError: Invalid input of type: 'dict'.
```

Convert to a byte, string or number first.
You can in fact make this work with Redis. There are two different ways to mimic nested data in redis-py and Redis:

1. Serialize the values into a string with something like `json.dumps()`
2. Use a delimiter in the key strings to mimic nesting in the values

Option 1: Serialize the Values Into a String
You can use `json.dumps()` to serialize the dict into a JSON-formatted string:

```
>>> import json
>>> r.set(484272, json.dumps(restaurant_484272))
True
```

If you call `.get()`, the value you get back will be a bytes object, so don't forget to deserialize it to get back the original object. `json.dumps()` and `json.loads()` are inverses of each other, for serializing and deserializing data, respectively:

```
>>> from pprint import pprint
>>> pprint(json.loads(r.get(484272)))
{'address': {'city': 'New York',
             'state': 'NY',
             'street': '11 E 30th St',
             'zip': 10016},
 'name': 'Ravagh',
 'type': 'Persian'}
```

This applies to any serialization protocol, with another common choice being yaml:

```
>>> import yaml  # python -m pip install PyYAML
>>> yaml.dump(restaurant_484272)
'address: {city: New York, state: NY, street: 11 E 30th St, zip: 10016}\
nname: Ravagh\ntype: Persian\n'
```

No matter what serialization protocol you choose to go with, the concept is the same: you're taking an object that is unique to Python and converting it to a bytestring that is recognized and exchangeable across multiple languages.

Option 2: Use a Delimiter in Key Strings
There's a second option that involves mimicking "nestedness" by concatenating multiple levels of keys in a Python dict. This consists of flattening the nested dictionary through recursion, so that each key is a concatenated string of keys, and the values are the deepest-nested values from the original dictionary. Consider our dictionary

```
restaurant_484272 = {
    "name": "Ravagh",
    "type": "Persian",
    "address": {
        "street": {
            "line1": "11 E 30th St",
            "line2": "APT 1",
        },
        "city": "New York",
        "state": "NY",
        "zip": 10016,
    }
}
```

We want to get it into this form:

```
{
    "484272:name":                    "Ravagh",
    "484272:type":                    "Persian",
    "484272:address:street:line1":    "11 E 30th St",
    "484272:address:street:line2":    "APT 1",
    "484272:address:city":            "New York",
    "484272:address:state":           "NY",
    "484272:address:zip":             "10016",
}
```

That's what `setflat_skeys()` below does, with the added feature that it does inplace `.set()` operations on the Redis instance itself rather than returning a copy of the input dictionary:

```
from collections.abc import MutableMapping

def setflat_skeys(
    r: redis.Redis,
    obj: dict,
    prefix: str,
    delim: str = ":",
    *,
    _autopfix=""
) -> None:

"""Flatten `obj` and set resulting field-value pairs into `r`.

    Calls `.set()` to write to Redis instance inplace and returns None.

    `prefix` is an optional str that prefixes all keys.
    `delim` is the delimiter that separates the joined, flattened keys.
    `_autopfix` is used in recursive calls to created de-nested keys.

    The deepest-nested keys must be str, bytes, float, or int.
    Otherwise a TypeError is raised.
    """
    allowed_vtypes = (str, bytes, float, int)
    for key, value in obj.items():
        key = _autopfix + key
        if isinstance(value, allowed_vtypes):
            r.set(f"{prefix}{delim}{key}", value)
        elif isinstance(value, MutableMapping):
```

```
            setflat_skeys(
                r, value, prefix, delim, _autopfix=f"{key}{delim}"
            )
        else:
            raise TypeError(f"Unsupported value type: {type(value)}")
```

The function iterates over the key-value pairs of obj, first checking the type of the value (Line 25) to see if it looks like it should stop recursing further and set that key-value pair. Otherwise, if the value looks like a dict (Line 27), then it recurses into that mapping, adding the previously seen keys as a key prefix (Line 28).
Let's see it at work:

```
>>> r.flushdb()  # Flush database: clear old entries
>>> setflat_skeys(r, restaurant_484272, 484272)

>>> for key in sorted(r.keys("484272*")):  # Filter to this pattern
...     print(f"{repr(key):35}{repr(r.get(key)):15}")
...
b'484272:address:city'             b'New York'
b'484272:address:state'            b'NY'
b'484272:address:street:line1'     b'11 E 30th St'
b'484272:address:street:line2'     b'APT 1'
b'484272:address:zip'              b'10016'
b'484272:name'                     b'Ravagh'
b'484272:type'                     b'Persian'

>>> r.get("484272:address:street:line1")
b'11 E 30th St'
```

The final loop above uses `r.keys("484272*")`, where "`484272*`" is interpreted as a pattern and matches all keys in the database that begin with "`484272`".
Notice also how `setflat_skeys()` calls just `.set()` rather than `.hset()`, because we're working with plain *string:string* field-value pairs, and the `484272` ID key is prepended to each field string.

Encryption
Another trick to help you sleep well at night is to add symmetric encryption before sending anything to a Redis server. Consider this as an add-on to the security that you should make sure is in place by setting proper values in your Redis configuration. The example below uses the cryptography package:

```
$ python -m pip install cryptography
```

To illustrate, pretend that you have some sensitive cardholder data (CD) that you never want to have sitting around in plaintext on any server, no matter what. Before caching it in Redis, you can serialize the data and then encrypt the serialized string using Fernet:

```
>>> import json
>>> from cryptography.fernet import Fernet

>>> cipher = Fernet(Fernet.generate_key())
>>> info = {
...     "cardnum": 2211849528391929,
...     "exp": [2020, 9],
...     "cv2": 842,
... }

>>> r.set(
...     "user:1000",
...     cipher.encrypt(json.dumps(info).encode("utf-8"))
... )

>>> r.get("user:1000")
b'gAAAAABcg8-LfQw9TeFZ1eXbi'  # ... [truncated]

>>> cipher.decrypt(r.get("user:1000"))
b'{"cardnum": 2211849528391929, "exp": [2020, 9], "cv2": 842}'

>>> json.loads(cipher.decrypt(r.get("user:1000")))
{'cardnum': 2211849528391929, 'exp': [2020, 9], 'cv2': 842}
```

Because info contains a value that is a list, you'll need to serialize this into a string that's acceptable by Redis. (You could use json, yaml, or any other serialization for this.) Next, you encrypt and decrypt that string using the cipher object. You need to deserialize the decrypted bytes using `json.loads()` so that you can get the result back into the type of your initial input, a dict.

Note: Fernet uses AES 128 encryption in CBC mode. See the cryptography docs for an example of using AES 256. Whatever you choose to do, use cryptography, not pycrypto (imported as Crypto), which is no longer actively maintained.
If security is paramount, encrypting strings before they make their way across a network connection is never a bad idea.

Compression
One last quick optimization is compression. If bandwidth is a concern or you're cost-conscious, you can implement a lossless compression and decompression scheme when you send and receive data from Redis. Here's an example using the bzip2 compression algorithm, which in this extreme case cuts down on the number of bytes sent across the connection by a factor of over 2,000:

```
>>> import bz2

>>> blob = "i have a lot to talk about" * 10000
>>> len(blob.encode("utf-8"))
260000

>>> # Set the compressed string as value
>>> r.set("msg:500", bz2.compress(blob.encode("utf-8")))
>>> r.get("msg:500")
b'BZh91AY&SY\xdaM\x1eu\x01\x11o\x91\x80@\x002l\x87\'  # ... [truncated]
>>> len(r.get("msg:500"))
122
>>> 260_000 / 122  # Magnitude of savings
2131.1475409836066

>>> # Get and decompress the value, then confirm it's equal to the original
>>> rblob = bz2.decompress(r.get("msg:500")).decode("utf-8")
>>> rblob == blob
True
```

The way that serialization, encryption, and compression are related here is that they all occur client-side. You do some operation on the original object on the client-side that ends up making more efficient use of Redis once you send the string over to the server. The inverse operation then happens again on the client side when you request whatever it was that you sent to the server in the first place.

Using Hiredis

It's common for a client library such as redis-py to follow a protocol in how it is built. In this case, redis-py implements the REdis Serialization Protocol, or RESP.

Part of fulfilling this protocol consists of converting some Python object in a raw bytestring, sending it to the Redis server, and parsing the response back into an intelligible Python object. For example, the string response "OK" would come back as `"+OK\r\n"`, while the integer response 1000 would come back as `":1000\r\n"`. This can get more complex with other data types such as RESP arrays.

A parser is a tool in the request-response cycle that interprets this raw response and crafts it into something recognizable to the client. redis-py ships with its own parser class, PythonParser, which does the parsing in pure Python. (See `.read_response()` if you're curious.)

However, there's also a C library, Hiredis, that contains a fast parser that can offer significant speedups for some Redis commands such as `LRANGE`. You can think of Hiredis as an optional accelerator that it doesn't hurt to have around in niche cases.

All that you have to do to enable redis-py to use the Hiredis parser is to install its Python bindings in the same environment as redis-py:

```
$ python -m pip install hiredis
```

What you're actually installing here is `hiredis-py`, which is a Python wrapper for a portion of the hiredis C library.

The nice thing is that you don't really need to call `hiredis` yourself. Just `pip` install it, and this will let `redis-py` see that it's available and use its `HiredisParser` instead of `PythonParser`. Internally, `redis-py` will attempt to import hiredis, and use a `HiredisParser` class to match it, but will fall back to its PythonParser instead, which may be slower in some cases:

```
# redis/utils.py
try:
    import hiredis
    HIREDIS_AVAILABLE = True
except ImportError:
    HIREDIS_AVAILABLE = False


# redis/connection.py
if HIREDIS_AVAILABLE:
    DefaultParser = HiredisParser
else:
    DefaultParser = PythonParser
```

Using Enterprise Redis Applications
While Redis itself is open-source and free, several managed services have sprung up that offer a data store with Redis as the core and some additional features built on top of the open-source Redis server:

- Amazon ElastiCache for Redis: This is a web service that lets you host a Redis server in the cloud, which you can connect to from an Amazon EC2 instance. For full setup instructions, you can walk through Amazon's ElastiCache for Redis launch page.
- Microsoft's Azure Cache for Redis: This is another capable enterprise-grade service that lets you set up a customizable, secure Redis instance in the cloud.

The designs of the two have some commonalities. You typically specify a custom name for your cache, which is embedded as part of a DNS name, such as `demo.abcdef.xz.0009.use1.cache.amazonaws.com` (AWS) or `demo.redis.cache.windows.net` (Azure).
Once you're set up, here are a few quick tips on how to connect.
From the command line, it's largely the same as in our earlier examples, but you'll need to specify a host with the h flag rather than using the default localhost. For Amazon AWS, execute the following from your instance shell:

```
$ export REDIS_ENDPOINT="demo.abcdef.xz.0009.use1.cache.amazonaws.com"
$ redis-cli -h $REDIS_ENDPOINT
```

For Microsoft Azure, you can use a similar call. Azure Cache for Redis uses SSL (port 6380) by default rather than port 6379, allowing for encrypted communication to and from Redis, which can't be said of TCP. All that you'll need to supply in addition is a non-default port and access key:

```
$ export REDIS_ENDPOINT="demo.redis.cache.windows.net"
$ redis-cli -h $REDIS_ENDPOINT -p 6380 -a <primary-access-key>
```

The `-h` flag specifies a host, which as you've seen is 127.0.0.1 (localhost) by default.
When you're using `redis-py` in Python, it's always a good idea to keep sensitive variables out of Python scripts themselves, and to be careful about what read and write permissions you afford those files. The Python version would look like this:

```
>>> import os
>>> import redis

>>> # Specify a DNS endpoint instead of the default localhost
>>> os.environ["REDIS_ENDPOINT"]
'demo.abcdef.xz.0009.use1.cache.amazonaws.com'
>>> r = redis.Redis(host=os.environ["REDIS_ENDPOINT"])
```

That's all there is to it. Besides specifying a different host, you can now call command-related methods such as `r.get()` as normal.

Note: If you want to use solely the combination of `redis-py` and an AWS or Azure Redis instance, then you don't really need to install and make Redis itself locally on your machine, since you don't need either redis-cli or redis-server.
If you're deploying a medium- to large-scale production application where Redis plays a key role, then going with AWS or Azure's service solutions can be a scalable, cost-effective, and security-conscious way to operate.

Wrapping Up
That concludes our whirlwind tour of accessing Redis through Python, including installing and using the Redis REPL connected to a Redis server and using `redis-py` in real-life examples. Here's some of what you learned:

- `redis-py` lets you do (almost) everything that you can do with the Redis CLI through an intuitive Python API.
- Mastering topics such as persistence, serialization, encryption, and compression lets you use Redis to its full potential.
- Redis transactions and pipelines are essential parts of the library in more complex situations.
- Enterprise-level Redis services can help you smoothly use Redis in production.

Redis has an extensive set of features, some of which we didn't really get to cover here, including server-side Lua scripting, sharding, and master-slave replication. If you think that Redis is up your alley, then make sure to follow developments as it implements an updated protocol, RESP3.