RedPipe Documentation

Release 1.0.0

John Loehrer

May 11, 2017

Contents

Did you try to use Redis pipelines? Did you get a pounding headache? Did you throw your laptop in frustration? Never fear. **RedPipe** will make you feel better almost immediately. If you have no idea what Redis is or why you should pipeline commands, look it up already.

CHAPTER 1

Requirements

The *redpipe* module requires Python 2.7 or higher.

It also requires redis-py or redis-py-cluster.

CHAPTER 2

What is RedPipe?

RedPipe is a wrapper around the pipeline component of redis-py or redis-py-cluster. It makes it easy to reduce network round trips when talking to *Redis*.

For more general information about redis pipelining, see the official redis documentation.

Use **RedPipe** to build pipelined redis calls in a modular reusable way. Rewrite your existing application calls via *redis-py* into efficient batches with only minimal changes.

CHAPTER 3

How Does it Work?

RedPipe makes pipeline commands work almost like un-pipelined commands in *redis-py*. You may have used pipelines before in a few spots as a last-minute performance optimization. **Redpipe** operates with a different paradigm. It assumes every call will be pipelined. And it gives you the tools to do it.

Some concepts sound complicated:

- futures for data prior to pipeline execution
- callbacks on pipeline execution
- nested pipelines

This documentation will explain all of these concepts and why they are important. All of these things together allow you to build modular functions that can be combined with other pipelined functions. You will be able to pass a pipeline into multiple functions, collect the results from each function, and then execute the pipeline to hydrate those result objects with data.

CHAPTER 4

What do I Need to Know to Start?

If you've used *redis-py*, you know most of what you need already to start using **RedPipe**.

If not, head over there and play with redis-py first. Or check out this very easy tutorial on *redis-py* basics:

http://agiliq.com/blog/2015/03/getting-started-with-redis-py/

You'll find the redpipe api looks nearly identical. That's because **RedPipe** is a wrapper around *redis-py*.

RedPipe adds the ability to pipeline commands in a more natural way.

CHAPTER 5

What Else Can it do?

You can use just the core of the **RedPipe**. Wrap your existing *redis-py* commands with **RedPipe** and profit. But the library unlocks a few other cool things too:

- Keyspaces allow you to work more easily with collections of Redis keys.
- Structs give you an object-oriented interface to working with Redis hashes.

Both components will make it easier to manipulate data in your application. And they are optimized for maximum network efficiency. You can pipeline *Keyspaces* and *Structs* just like you can with the core of **RedPipe**.

CHAPTER 6

User Documentation

This part of the documentation explains why you need RedPipe. Then it focuses on step-by-step instructions for getting the most out of RedPipe.

Rationale

Why do I need this?

Redis is really fast. If you only use redis on your laptop over a unix domain socket, go away. You probably do not need to think about pipelining.

But in production scenarios, redis is usually running on another machine. That means the client needs to talk to redis server over a network. If you are in AWS Ec2, there's a good chance your redis server is in a different availablity zone.

Your application latency is determined by the speed of your network. If you make many calls to redis to satisfy a single request, application latency can be terrible.

Each command needs to make another trip across the network. If your network round trip time is one millisecond, that doesn't seem too terrible. But if you have dozens or hundreds of redis commands, this adds up quickly.

Pipelining is Cool, But ...

Redis pipelining can dramatically reduce the number of network round trips. It boxcars a bunch of commands together over the wire. When running 50 commands against *Redis*, instead of 50 network round trips in serial order, boxcar them all together in one..

```
# this is pure redis-py code, not using redpipe here
client = redis.Redis()
with client.pipeline() as pipe:
    for i in range(0, 50):
        pipe.incr('foo%d' % i)
```

```
    # the one network round trip happens here.
    results = pipe.execute()
```

That's a **BIG** improvement in application latency. And you don't need *RedPipe* to do this. It's built into *redis-py* and almost every other redis client.

Pipelining is Hard to do

Here's the catch ... *the results aren't available until after you execute the pipeline*.

In the example above, consuming the results on pipe execute is pretty easy. All of the results are uniform and predictable from a loop. but what if they aren't?

Here's an example of pipelining heterogenous commands.

```
# redis-py code example, not redpipe!
client = redis.StrictRedis()
with client.pipeline() as pipe:
    pipe.incr('key1')
    pipe.expire('key1', 60)
    pipe.incrby('key2', '3')
    pipe.expire('key2', 60)
    key1, expire_key1, key2, expire_key2 = pipe.execute()
```

See how the results are separated from the action we want to perform? This example is a bit contrived but it illustrates a point. We have to be careful the results from the pipeline match up with the invocation order.

And what if we want to create a reusable function that can be pipelined?

Here's what I'd like to be able to do:

```
def increment_and_expire(key, num, expire, pipe):
    pipe.incrby(key, num)
    pipe.expire(expire)
    # return result of incrby operation
    # HOW????
```

I want to return the result of the *pipe.incrby* call from the function. But the result isn't available until *pipe.execute*. This happens outside the scope of the function. And the caller of the function doesn't know how many pipeline calls were invoked. So grabbing the correct value from pipe.execute() is tricky.

Also, consider the difference between the direct and pipeline interfaces in *redis-py*:

```
# redis-py non-pipelined example
result = client.incr('key1')
```

vs.

```
# redis-py pipelined example
pipe = client.pipeline()
pipe.incr('key1')
result = pipe.execute()[0]
```

Although the calls look almost the same, the way you fetch the result is very different.

Bottom line, it's inconvenient to use pipelines in *python*. And it is especially inconvenient when trying to create modular and reusable components.

How RedPipe Makes Things Easier

RedPipe makes things easier by first making it harder. It's a paradigm shift. You ready? Here it comes.

All redis calls are pipelined.

On the surface this seems unnecessary. But stick with me for a few minutes. It will unlock the tools to break up pipelined calls into modular reusable components.

The first step is to make the commands return a reference to the data immediately. We'll call this reference object a *Future*. The *redpipe.Future* object gets populated with data once the pipeline executes.

That makes the code look very much like a non-pipelined call. You invoke the redis-py method and you get a response back from that call. The response is a *redpipe.Future* object, but you don't ever need to think about that.

Once the pipeline executes, the *Future* behaves just like the underlying result.

RedPipe embraces the spirit of duck-typing.

You can iterate on a Future if the result is a list. Add or subtract from it if it is an int. Print it out like a string. In short, you should be able to use it interchangeably with the underlying *future.result* field.

This gives us the ability to create reusable building blocks.

How, wait what??

Okay, keep reading. I'll explain.

Reusable Building Blocks

here's how *RedPipe* allows me to do what I wanted to do above.

```
def increment_and_expire(key, num=1, expire=60, pipe=None):
    pipe = redpipe.pipeline(pipe)
    ref = pipe.incrby(key, num)
    pipe.expire(key, expire)
    pipe.execute()
    return ref
```

Now we have a reusable function! *redpipe.pipeline* can give us a pipeline if no pipeline is passed into the function. Or it wraps the one passed in. Let's invoke our function!

```
with redpipe.pipeline() as pipe:
    key1 = increment_and_expire('key1', pipe=pipe)
    key2 = increment_and_expire('key2', pipe=pipe)
    pipe.execute()

print(key1)
print(key2)
```

Or I can call the function all by itself without passing in a pipe.

```
print(increment_and_expire('key3'))
```

The function will always pipeline the *incrby* and *expire* commands together.

When we pass in one pipeline() into another, it creates a nested structure. When we pass in a pipeline to our function, it will combine with the other calls above it too! So you could pipeline a hundred of calls without any more complexity:

```
with redpipe.pipeline() as pipe:
    results = [increment_and_expire('key%d' % i, pipe=pipe) for i in range(0, 100)]
    pipe.execute()
print(results)
```

We have sent 200 redis commands with only 1 network round-trip. Pretty cool, eh? This only scratches the surface of what we can do.

Getting Started

Installation

To install, use pip:

```
pip install redpipe
```

or from source:

```
python setup.py install
```

Get the Source Code

RedPipe is actively developed on GitHub.

You can either clone the public repository:

```
git clone git://github.com/72squared/redpipe.git
```

Or, download the tarball:

```
curl -OL https://github.com/72squared/redpipe/tarball/master
```

Once you have a copy of the source, install it into your site-packages easily:

```
python setup.py install
```

Connect redis-py to RedPipe

To use redpipe, You need to bind your redis client instance to *RedPipe*. Use the standard redis-py client.

```
client = redis.Redis()
redpipe.connect_redis(client)
```

You only need to do this setup once during application bootstrapping.

This example just sets one connection up as a default, since that is the most common case. But you can connect multiple redis connections to *RedPipe*.

You can use *StrictRedis* if you want too. It doesn't matter. Whatever you use normally in your application.

The goal is to reuse your application's existing redis connection. RedPipe can be used to build your entire in your application. Or you can use *RedPipe* along side your existing code.

More on this later.

Using RedPipe

Using *RedPipe* is easy. We can pipeline multiple calls to redis and assign the results to variables.

```
with redpipe.pipeline() as pipe:
    foo = pipe.incr('foo')
    bar = pipe.incr('bar)
    pipe.execute()
print([foo, bar])
```

RedPipe allocates a pipeline object. Then we increment a few keys on the pipeline object. The code looks mostly like the code you might write with redis-py pipelines. The methods you call on the pipeline object are the same. But, notice that each *incr* call immediately gets a reference object back in return from each call. That part looks similar to how *redis-py* works without a pipeline.

The variables (in this case *foo* and *bar*) are empty until the pipeline executes. if you try to do any operations on them beforehand, it will raise an exception. Once we complete the *execute()* call we can consume the pipeline results. These variables, *foo* and *bar*, behave just like the underlying result once the pipeline executes. You can iterate of it, add it, multiply it, etc.

Reusable Functions

You can write a function that takes in a pipeline, and returns a result before the pipeline even executes.

Here's a quick example of what I mean:

```
def get_foo(pipe=None):
     with redpipe.pipeline(pipe=pipe) as pipe:
        pipe.setnx('foo', 'bar')
        foo = pipe.get('foo')
        pipe.execute()
        return foo
```

Now I can invoke my function. I can call it in isolation:

```
print(get_foo())
```

This will pipeline the following commands to redis:

- SETNX foo bar
- GET foo

Or I can pipeline it with other things:

```
with redpipe.pipeline() as pipe:
    foo = get_foo(pipe)
    bar = pipe.get('bar')
    pipe.execute()
```

This example will pipeline these three commands together:

- SETNX foo bar
- GET foo
- GET bar

In this example, the *foo* and *bar* variables are both *redpipe.Future* objects. They are empty until the *pipe.execute()* happens outside of the function. The *pipe.execute()* called inside the *get_foo* function in this case is a *NestedPipeline*. It passes it's stack of commands to the parent pipeline. That's because we passed a pipeline object into the *get_foo* function. The function passed that into *redpipe.pipeline* and it returned a NestedPipeline to wrap the one passed in.

Futures

When redis clients communicate with the server, they send a command, then wait for the response. The *redis-py* client reflects this design choice. But when you pipeline, you don't wait. You queue up a bunch of commands. Then you execute them. Then you gather these results and feed them back to where they need to go.

This forces the code invoking the execute method to *know* what to do with all the responses.

That's pretty inconvenient.

The **RedPipe** *Future* makes it possible to get a reference to the data before pipeline execute happens. A *Future* is a contract that says the result of this redis command will be populated into this object once the pipeline executes. And you can use a *Future* just like you would the actual result. But only after the pipeline executes. If you try to use it prior to that, it raises a *redpipe.ResultNotReady* exception. Kaboom!

Well, what use is that?

For one, it makes it easier to assign it a variable name in the context of calling the command. No confusion about how to get the result you need from the array of results returned by *pipeline.execute*. Unlike *redis-py* **RedPipe** does not return an array of results on the *execute* call. You already have the result as a variable returned from the command you called initially.

More importantly, the *Future* can be passed into a python closure that can do additional work. Tell the pipeline to execute your closure callback function after it runs. Now you have a powerful mechanism to format those results and build modular reusable components.

Still don't quite see how? Read more about callbacks.

Gotchas

There are a few gotchas to watch out for:

- isinstance() checks
- identity checks like: future is None
- trying to mutate the object like this: future += 1

You can always type cast the object into the type you expect if you need this behavior.

```
f = Future()
f.set(1)

# `f is 1` fails
assert(int(f) is 1) # works
```

This doesn't work so well for is None checks. You can't type-cast to None. You can use equality checks though.

```
f = Future()
f.set(None)
assert(f == None)
```

This is frowned upon by most lint-checks who think the *is* comparison is more appropriate. But if you do an *is* comparison, that compares the object ids. And there's no way for **RedPipe** to wrap that.

UGH!

In response, I came up a handy IS method.

```
f = Future()
f.set(None)
assert(f.IS(None))
```

Or you can access the underlying result:

```
f = Future()
f.set(None)

assert(f.result is None)
```

Hope that helps.

Examples

Other than those few caveats, you should be able to access a future object just like the underlying result.

Here are some examples if your result is numeric.

```
future = Future()
future.set(1)
assert(future == 1)
assert(future != 2)
assert(future > 0)
assert(future < 2)
assert(future >= 1)
assert(future <= 1)
assert(bool(future))
assert(float(future) == 1.0)
assert(future + 1 == 2)
assert(future * 2 == 2)
assert(future ^ 1 == 0)
assert(repr(future) == '1')
```

And here is an example if your future is a list:

```
future = Future()
future.set([1])
assert(future == [1])
assert([v for v in future] == [1])
assert(future + [2] == [1, 2])
```

And here is a dictionary:

```
future = Future()
future.set({'a': 1})
assert(future == {'a': 1})
assert(dict(future) == {'a': 1})
assert({k: v for k, v in future.items()} == {'a': 1})
```

There are many more operations supported but these are the most common. Let me know if you need more examples or explanation.

Json Serialization

The default json serializer doesn't know anything about **RedPipe** *Futures*. When it encounters a *Future*, the json encoder would normally blow up.

I monkey-patched it so it will serialize correctly.

```
future = Future()
future.set({'a': 1})
json.dumps(future)
```

The monkey-patching is pretty simple. Take a look at the source code if you are interested. If you have serious objections to this hack, let me know.

If you used a different json serializer, I can't help you. You may be able to patch those libraries on your own. Or you could also explicitly extract the result or type-cast before encoding as json.

Automatic Pipeline Execution

By default, you must call *pipe.execute* for the commands to be sent to redis. With the *autoexec* flag, you can save a step:

```
with redpipe.pipeline(pipe=pipe, autoexec=True) as pipe:
    foo_count = pipe.incr('foo')

 print(foo_count)
```

Notice we are using the *with* control-flow structure block. As you leave the block, it triggers the *__exit__* method on the pipe object. If the autoexec flag was set, the method verifies no exception was thrown and executes the pipeline. Otherwise, you must call *pipe.execute()* explicitly.

There's even a wrapper for this because it is used so often:

```
with redpipe.autoexec(pipe=pipe) as pipe:
    foo_count = pipe.incr('foo')

 print(foo_count)
```

Callbacks

What if we want to be able to combine the results of multiple operations inside a function? We need some way to wait until the pipeline executes and then combine the results. Callbacks to the rescue!

Let me show you what I mean.

(This example uses the *autoexec* flag. If you missed that section, read about it here.)

```
def incr_sum(keys, pipe=None):
    future = redpipe.Future()

    with redpipe.pipeline(pipe, autoexec=True) as pipe:
```

```
        results = [pipe.incr(key) for key in keys]

        def cb():
            future.set(sum(results))

        pipe.on_execute(cb)

    return future

# now get the value on 100 keys
print(incr_sum(["key%d" % i for i in range(0, 100)]))
```

We didn't pass in a pipeline to the function. It pipelines internally. So if we are just calling the function one time, no need to pass in a pipeline. But if we need to call it multiple times or in a loop, we can pass a pipeline in.

```
with redpipe.pipeline(autoexec=True) as pipe:
    first = incr_sum(["key%d" % i for i in range(0, 100)], pipe=pipe)
    second = incr_sum(["key%d" % i for i in range(100, 200)], pipe=pipe)

print(first)
print(second)
```

The pipeline context knows how to nest these operations. As each child context completes it passes its commands and callbacks up a level. The top pipeline context executes the functions and callbacks, creating the final result.

Use Cases

Callbacks can be used for all kinds of purposes. In fact, the internals of *RedPipe* take advantage of the callback hook for many different purposes.

Here are some examples:

- Formatting the results of a redis command before returning it
- combining multiple results from several pipelined commands into a single response
- attaching data from a pipelined call to other objects in your application

Gotchas

You can put just about anything you want into a callback. But try to avoid the trap of making subsequent network calls within a callback when building a function. It limits the reusability of your modular building block. The problem is that while the first network round-trip can be pipelined, you won't be able to pipeline the second.

Here's an example of what I mean:

```
def incr_if_gt(key, threshold, pipe=None):
    with redpipe.pipeline(pipe, autoexec=True) as pipe:
        future = redpipe.Future()
        value = pipe.get(key)

        def cb():
            if value > threshold:
                with redpipe.autoexec() as p:
                    future.set(p.incr(key))
            else:
                future.set(int(value))
```

```
        pipe.on_execute(cb)

        return future
```

While this code example certainly would work, the *p.incr(key)* command inside could not be pipelined with anything. So your get command could be pipelined with many other calls, but if it needs to increment the key, it will need to do it all alone.

Bad programmer. No cookie.

Nor can you use the pipe object from the context inside of our parent function. The reason is because when the pipe exits the with block, it resets the list of commands and callbacks.

Named Connections

So far the examples I've shown have assumed only one connection to *Redis*. But what if you need to talk to multiple backends?

How to Configure multiple Connections

RedPipe allows you to set up different connections and then refer to them:

```
redpipe.connect_redis(redis.StrictRedis(port=6379), name='users')
redpipe.connect_redis(redis.StrictRedis(port=6380), name='messages')
```

Now I can refer to those named connections inside my functions and throughout my application.

```
with redpipe.pipeline(name='users', autoexec=True) as users:
    users.hset('u{1}', 'name', 'joe')

with redpipe.pipeline(name='messages', autoexec=True) as messages:
    messages.hset('m{1}', 'body', 'hi there')
```

If you don't specify a name, it assumes a default connection set up like this:

```
redpipe.connect_redis(redis.StrictRedis(port=6379))
```

You can actually map the same redis connection to multiple names if you want. This is good for aliasing names when preparing to split up data, or for testing.

Why Named Connections are Needed

RedPipe allows you to pass in a pipeline to a function, or optionally pass in nothing. The function doesn't have to think about it. Just pass the pipe (or None) into *redpipe.pipeline* and everything looks the same under the covers. But if you have multiple connections, the named pipe passed into the function may not be the same connection. In this case, we need to always specify what connection we want to use.

If the connection is different than the one passed into the function, redpipe will still batch the two calls together in pipe execute from a logical perspective. But it needs to send commands to different instances of redis server. By specifying the connection you want to use with a named connection, you can make sure your command gets sent to the right server.

Talking to Multiple Servers in Parallel

When it's time to send those commands to the servers, redpipe batches all commands for each server and sends them out. *RedPipe* supports asynchronous execution of commands to multiple redis servers via threads. You can disable this so that the batches sent to each redis server are performed in serial order.

If you talk to only one redis backend connection at a time, *RedPipe* doesn't have to worry about parallel execution. If you execute a pipeline that combines commands to multiple backends, redpipe will use threads to talk to all backends in parallel.

If you are uncomfortable using threads in your application, you can turn it off at any time via:

```
redpipe.disable_threads()
```

To re-enable this behavior, do:

```
redpipe.enable_threads()
```

If you see any problems with asynchronous execution, let me know.

Nested Pipelines

The ability to pass one pipeline into another dramatically simplifies your application code. You can build a function that can perform a complete operation on its own. You can also effortlessly connect that function to other pipelines. You've seen it in action elsewhere in the documentation. Let's dive into what is actually happening under the covers.

How it works

The *redpipe.pipeline* function checks to see if you are passing in another pipeline object or not. If you pass in nothing, it gives you back a root-level *redpipe.pipelines.Pipeline* object. I deliberately did not expose this class at the root level of the package. You never need to instantiate it directly.

This *Pipeline* object will collect your commands. When *Pipeline.execute* is called, it obtains a *redis.StrictPipeline* and runs your pipelined commands. Simple.

If you pass in a *Pipeline* object into the *redpipe.pipeline* function, it returns a *redpipe.pipelines.NestedPipeline* object. Again, you should never need to instantiate it directly. And you can use *NestedPipeline* exactly like the *Pipeline* object.

When you execute *NestedPipeline*, it passes all the commands and callbacks queued up to its parent. The parent object is the one you passed into *redpipe.pipeline*. This may be a *Pipeline* object, or it may be another *NestedPipeline* object. It cleans itself up and defers execution responsibility to its parent.

The parent now waits for its execution method to be called. When it does, it keeps passing commands up the chain until it winds up in a *Pipeline* object. Then the commands get sent off to redis in one big batch. Then the callbacks are triggered, and everything is ready to use.

How to use it

This architecture means when you build a function, you don't need to think about what kind of pipeline you are recieving. It could be a *NestedPipeline* or a *Pipeline* or nothing at all.

Just wrap it all up in *redpipe.pipeline* and do your work.

```
class Beer(redpipe.Hash):
    keyspace = 'B'
    fields = {
        'beer_name': redpipe.TextField,
        'consumed' redpipe.Integer,
    }

def get_beer_from_fridge(beer_id, quantity=1, pipe=None):
    with redpipe.pipeline(pipe, autoexec=True) as pipe:
        b = Beer(pipe)
        b.hincrby(beer_id, 'consumed', quantity)
        return b.hgetall(beer_id)
```

Now I can grab one beer from the fridge at a time. Or I can take one in each hand. Or I can grab a case! And I can do it all in a single network transaction.

```
drinks = []
with redpipe.autoexec() as pipe:
    drinks.append(get_beer_from_fridge('schlitz', pipe=pipe))
    drinks.append(get_beer_from_fridge('guinness', 6, pipe=pipe))
print(drinks)
```

Transactions

When you talk aboout redis pipelining, most people conflate it with transactions. In fact, the *redis-py* library conflates it by making a *transaction* flag you pass into the pipeline object. There has been a lot of effort to make Redis behave in a transactional way.

This is not a goal for **RedPipe**.

> **RedPipe** was written to improve network i/o.

Most of the concepts for **RedPipe** came from a project that uses Redis Cluster. It's not practical or supported to use transactions there. Any kind of atomic multi-step operation is limited to a single key, and is best accomplished with a LUA script.

I haven't disallowed transactions. But I'm not going out of my way to try to support it either.

You can turn transactions on or off in setting up your connection.

```
client = redis.StrictRedis()
redpipe.connect_redis(client, transaction=False)
```

I welcome discussion. If this is a pain point for you, *let me know <https://github.com/72squared/redpipe/issues>*.

Working with Keyspaces

Usually when working with *Redis*, developers often group a collection of keys that are similar under a keyspace. Use a key pattern with a prefix and curly braces around the unique identifier for that record. For example, for a list of followers for user ids *1* and *2*, I might have keys *F{1}* and *F{2}*.

This keyspace functions as a virtual table, like what you might have in a typical RDBMS. Except that each key is really independent. We just use a naming convention to group them together.

Example of a Sorted Set Keyspace

RedPipe gives you a way to easily manipulate these keyspaces.

Here's an example of a sorted set:

```
class Followers(redpipe.SortedSet):
    keyspace = 'F'
    connection = 'default'

key1 = '1'
key2 = '2'
with redpipe.pipeline(name='default') as pipe:
    f = Followers(pipe=pipe)
    f.zadd(key1, 'a', score=1)
    f.zadd(key2, 'a', score=2)
    f1_members = f.zrange(key1, 0, -1)
    f2_members = f.zrange(key2, 0, -1)
    pipe.execute()

print(f1_members)
print(f2_members)
```

We can specify what named connection we want to use with the *connection* variable. Or you can omit it if you are using just one default connection to redis.

You will notice the interface provided by the keyspace object *redpipe.SortedSet* looks just like *redis-py* functions. Except it omits the name of the key. That's because the key name is already specified in the constructor.

Supported Keyspace Types

All of the *redis-py* sorted set functions are exposed on the in the example above.

In a similar way, we support the other *Redis* primitives:

- strings
- sets
- lists
- hashes
- sorted sets
- hyperloglog
- geo (in progress)

All the commands associated with each data type are exposed for each. See the official redis documentation for more information, or refer to redis-py.

Character Encoding in Keyspaces

When you use *redpipe.pipeline()* directly, **RedPipe** disables automatic character decoding. That's because there's no way to know how to decode responses for every single request that goes through redis. The dump/restore commands, for example, never should automatically decode the binary data. It's not utf-8. And if you are pickling python objects and storing them in redis, character encoding makes no sense.

With a Keyspace, though, it's entirely appropriate to map the binary data in redis to appropriate encodings. That's because you are defining some application

There are some defaults you can tune per keyspace that you define:

- keyparse
- valueparse

We treat these as utf-8 encoded unicode strings, controlled by the formatter *redpipe.TextField*. There are many other data types you can use.

They control how to encode the key and the values in the redis data structures.

In addition, *redpipe.Hash* gives you additional ways to encode and decode data for each individual member of the Hash.

Fields in Hashes

Often you want to store data in Hashes that maps to a particular data type. For example, a boolean flag, an integer, or a float. Redis stores all the values as byte strings and doesn't interpret. In the Keyspace, we default to treating all fields as unicode that is stored in redis as utf-8 binary strings. If you need something different, you can set up explicit mappings for other data types in *redpipe.Hash*. This is not required but it makes life easier.

```
class User(redpipe.Hash):
    keyspace = 'U'
    fields = {
        'first_name': redpipe.TextField,
        'last_name': redpipe.TextField,
        'admin': redpipe.BooleanField,
        'last_seen': redpipe.FloatField,
        'encrypted_secret': redpipe.BinaryField,
    }
```

You can see we defined a few fields and gave them types that we can use in python. The fields will perform basic data validation on the input and correctly serialize and deserialize from a *Redis* hash key.

```
key = '1'
with redpipe.autoexec() as pipe:
    u = User(pipe=pipe)
    data = {
        'first_name': 'Fred',
        'last_name': 'Flitstone',
        'admin': True,
        'last_seen': time.time(),
    }
    u.hmset(key, data)
    ref = u.hgetall(key)

assert(ref == data)
```

You can see this allows us to set booleans, ints and other data types into the hash and get the same values back.

Data Types defined for Keyspaces

Here's a list of all the different data types you can represent so far:

- BooleanField

- FloatField
- IntegerField
- TextField
- AsciiField
- BinaryField
- ListField
- DictField
- StringListField

If you don't see the one you want, you can always write your own. It's pretty easy. You just need an object that provides two methods:

- encode
- decode

The encode method that converts your python data structure into binary string. And the decode method to will convert it back consistently into your original python structure.

Strict or No?

Redis-py gives you two different interfaces:

- Redis
- StrictRedis

They provide the same functionality. *Redis* rewrites the the order of arguments to be more intuitive since the server order of arguments can be confusing in some cases. Whereas *StrictRedis* gives an interface that conforms to the same argument order that the server presents.

Keyspace classes conform to the *Redis* interface. It doesn't matter which type of object you pass into *redpipe.connect_redis*. The Keyspace object knows the right thing to do and will pass the arguments through correctly. It does this by using keyword arguments when it can do so and when there is ambiguity about the order of the command arguments. In some cases, keyword arguments cannot be used because *Redis* and *StrictRedis* used different keyword arguments. In those rare cases, the Keyspace classes bypass the issue and invoke *execute_command* directly.

Scanning the Keys in a Keyspace

When you use the *scan* command on a keyspace, **RedPipe** automatically builds a pattern that matches the keyspace you are using. Any additional patterns you pass in are searched for inside of that pattern. So you should be able easily iterate through a list of all keys in the keyspace.

The scan commands don't seem to work quite right in redis-py-cluster. I'm working with the package maintainer to try to get that squared away.

Structs

A *Struct* in redpipe is a dictionary-like object with persistence built in.

Easy, Efficient I/O

You want to be able to load data and persist it into a hash and still preserve the data-type of the original structure. We gave *redpipe.Hash* the ability to type-cast variables stored in redis. But we could make it more convenient to fetch and save data as objects.

That's where *redpipe.Struct* comes in.

Defining a Struct

Here's an example of how to define a *Struct*.

```
# assume we already set up our connection

# set up a struct object.
class User(redpipe.Struct):
    keyspace = 'U'
    key_name = 'user_id'
    fields = {
        'name': redpipe.TextField,
        'last_seen': redpipe.IntegerField,
        'admin': redpipe.BooleanField,
        'page_views': redpipe.IntegerField,
    }
```

A lot of this looks very similar to how we defined *redpipe.Hash*. That's because struct is built on top of the Hash object. It allows you to access data from a hash in a more object oriented manner.

The *Struct* does not enforce required fields on any of this data. Just as a redis hash object does not. It is up to your application logic to enforce these constraints.

The rule is that if the element is in the hash, it will be coerced into the appropriate data type by the *fields* definition. If an element in the hash is not mentioned in the *fields* it is coerced into a *TextField*.

You can override this default behavior by defining *valueparse*.

```
class User(redpipe.Struct):
    keyspace = 'U'
    key_name = 'user_id'
    fields = {
        # ...
    }
    valueparse = redpipe.AsciiField
```

This example will force all values not listed in *fields* to be set as ascii values in redis. (It does not coerce values already in redis to be ascii tho. It will treat them as text.)

You can specify an alternate redis connection if you are using multiple redis connections in your app.

```
class User(redpipe.Struct):
    keyspace = 'U'
    key_name = 'user_id'
    fields = {
        # ...
    }
    connection = 'users'
```

The string value *users* refers to a connection you have added in application bootstrapping. See the Named Connections section of this documentation.

Creating New Structs

Let's create a few user objects using our *Struct*. The first argument is always either the key or the data.

We pass in a pipeline so we can combine the save operation with other network i/o.

```
with redpipe.autoexec() as pipe:
    # create a few users
    ts = int(time.time())
    u1 = User({'user_id': '1', 'name': 'Jack', 'last_seen': ts}, pipe=pipe)
    u2 = User({'user_id': '2', 'name': 'Jill', 'last_seen': ts}, pipe=pipe)

# these model objects print out a json dump representation of the data.
print("first batch: %s" % [u1, u2])

# we can access the data like we would dictionary keys
assert(u1['name'] == 'Jack')
assert(u2['name'] == 'Jill')
assert(isinstance(u1['last_seen'], int))
assert(u1['user_id'] == '1')
assert(u2['user_id'] == '2')
```

When we exit the context, all the structs are saved to *Redis* in one pipeline operation. It also automatically loads the other fields in the hash. Since the commands are batched together, you can write the fields then read the hash in one pass. If you don't want it to read, you can set the fields to an empty array.

Accessing the Data

RedPipe exposes the variables from redis as elements like a dictionary:

```
user = User({'user_id': '1', 'name': 'Jack'})
assert(user['name'] == 'Jack')
```

Here, we accessed the name field of the redis hash as a dictionary element on the user object. This data is loaded from redis inside the object on instantiation by calling *hgetall()* on the key. The data is cached inside the struct.

You can coerce the objects into dictionaries.

```
user = User({'user_id': '1', 'name': 'Jack'})
assert(dict(user) == user)
```

This just takes all the internal data and returns it as a dictionary. If you don't define a *_key_name* on the object, it defaults to the field name '_key'. This primary key is not stored inside the hash. It is embedded in the redis object key name. This is more efficient that storing it both in the name of the key and as an element of the hash. It also avoids problems of accidentally creating a mismatch.

You can compare the user *Struct* to a dictionary for equality.

```
user = User({'user_id': '1', 'name': 'Jack'})
assert(dict(user) == user)
```

There is an *__eq__* magic method on *Struct* that allows this comparison.

You can iterate on the object like a dictionary:

```
user = User({'user_id': '1', 'name': 'Jack'})
assert({k: v for k, v in user.items()} == user)
```

You can see the user object has an *items* method. There is also a *iteritems* method for backward compatibility with python 2. The *iteritems* method is a generator, whereas *items* returns a list of key/value tuples.

You can access an unknown data element like you would a dictionary:

```
user = User({'user_id': '1', 'name': 'Jack'})
assert(user.get('name', 'unknown') == 'Jack')
```

The *get* method allows you to pass in a default if no key is found. It defaults to *None*.

You can check for key existence:

```
user = User({'user_id': '1', 'name': 'Jack'})
assert('name' in user)
assert('non-existent-name' not in user)
```

The magic method *__contains__* looks for the key in the internal dictionary, or the *_key_name* field.

You can check the length of a struct:

```
user = User({'user_id': '1', 'name': 'Jack'})
assert(len(user) == 2)
```

This will include the primary key, so it should never be less than 1. A *Struct* object will always have a primary key.

You can get the keys of a struct:

```
user = User({'user_id': '1', 'name': 'Jack'})
# returns a list but we don't know the order
# coerce to a set for comparison
assert(set(user.keys()) == {'user_id', 'name'})
```

The *_key_name* will show up in this list. If no *_key_name* is defined, you will see *_key* in the list of keys.

Many ORMS set the data as attributes of the object. *RedPipe* does not. This makes it easier to differentiate methods of the object from the data. It also avoids difficulty of data elements that don't obey the pythonic naming conventions of object attributes.

You can have a element name that would otherwise be illegal.

```
# this wouldn't work, syntax error
# user.full-name
# but this will!
user['full-name']
```

Modifying Structs

Let's read those two users we created and modify them.

```
with redpipe.autoexec() as pipe:
    users = [User('1', pipe=pipe), User('2', pipe=pipe)]
    ts = int(time.time())
    users[0].update({'name':'Bobby', 'last_seen': ts}, pipe=pipe)
    users[1].remove(['last_seen'])
```

```
print([dict(u1), dict(u2)])
```

When you pass just the key into the object it reads from the database. Then we can change the fields we want at any point. Or we can remove fields we no longer want.

Fields that are undefined can still be accessed as basic strings.

We can remove a field and return it like we would popping an item from a dict:

```
with redpipe.autoexec() as pipe:
    user = User({'user_id': '1', 'name': 'Jack'}, pipe=pipe)
    name = user.pop('name', pipe=pipe)

assert(name == 'Jack')
assert(user.get('name', None) is None)
```

This doesn't just pop the data from the local data structure. It also pops it from redis. Use at your own risk.

You don't have to use a pipeline if you don't want to:

```
user = User({'user_id': '1', 'name': 'Jack'})
name = user.pop('name')

assert(name == 'Jack')
assert(user.get('name', None) is None)
```

But then you pay for two network round-trips.

If you want to ensure you don't modify redis accidentally, coerce your user object into a dictionary.

You can increment a field:

```
with redpipe.autoexec() as pipe:
    user = User({'user_id': '1', 'name': 'Jack'}, pipe=pipe)
    user.incr('page_views', pipe=pipe)

assert(user['page_views'], 1)
```

As with the pop example, you can use a pipe or not. There's also a *decr* method which is the inverse.

Using the Underlying Hash

Because the struct is based on a *redpipe.Hash* object, you can access the underlying Hash. This is pretty helpful if you need to extend the functionality of your struct. From our earlier *User* struct example:

```
username = User.core().hget('1', 'name')
```

More on this later.

Deleting Structs

to delete all the data in a struct, use the same syntax as you would for a dictionary:

```
user = User('1')
user.clear()
```

Of course you can pipeline it:

```
with redpipe.autoexec() as pipe:
    user = User('1')
    user.clear(pipe)
```

If you need to delete a record without loading the record, you can call the Struct class method:

```
with redpipe.autoexec() as pipe:
    User.delete(['1', '2', '3'], pipe=pipe)
```

Extra Fields

I touched on it briefly before, but you can store arbitrary data in a struct too.

```
user = User({'user_id': '1', 'arbitrary_field': 'foo'})
assert(user['arbitrary_field'] == 'foo')
```

The data will be simple string key-value pairs by default. But you can add type-casting at any point easily in the *fields* dictionary.

Why Struct and not Model?

I chose the name *Struct* because it implies a single, standalone data structure. You clearly define data structure of the struct. And you can instantiate the struct with many records. The word *Struct* doesn't imply indexes or one-to-many relationships the way the word *Model* does.

Why no ORM?

An Object-Relational Mapping can make life much simpler. Automatic indexes, foreign keys, unique constraints, etc. It hides all that pesky complexity from you. If you want a good ORM for redis, check out ROM. Or redish. Both are pretty cool.

RedPipe does not provide you with an ORM solution.

Choose *Redpipe* if you really care about optimizing your network i/o.

To optimize redis i/o, you need to batch command operations together as much as possible. ORMs often hide things like automatic unique constraints and indexes beneath the covers. It bundles lots of multi-step operations together, where one operation feeds another. That makes it tricky to ensure you are batching those operations efficently as possible over the network.

RedPipe exposes low level redis command primitives. Inputs and outputs. This allows you to construct building blocks that can be pipelined efficiently.

Latency

Pipelining isn't a magic bullet. If you pipeline 10 thousand commands together, you have to wait until all 10k commands execute and stream back over the wire.

Most of the time, you will find a happy middle ground where 10 or 20 different commands can easily be combined together. This will make a difference.

When in doubt, profile your code. Look for the slow spots. If you dozens or hundreds of network round-trips to redis, RedPipe can help!

Project Status

RedPipe is based on what I've learned over the past 3 years. We run a really big open-source redis cluster where all of our data is stored in Redis. So these ideas were tested by fire in real production environments.

However, RedPipe is a complete rewrite of the original concepts. I took the opportunity to write it from scratch, taking advantage of all I learned. There may be a few bugs that have crept in during this big rewrite and refactor.

That's not an excuse for sloppy code or mistakes.

I believe in well tested code. If you find issues, let me know right away. I'll fix it and write a regression test.

Road Map

Here's my current backlog:

- distributed hash, so we can spread an index out over multiple keys
- better benchmarking
- Tutorials and Examples

Another way of defining the roadmap is listing what I expect **NOT** to be supported:

- Unique Constraints on Struct
- one-to-many indexes on Struct
- many-to-many indexes on Struct
- required fields on Struct

All of these start forcing me down the road of requiring network i/o in ways that you can't control. These operations are best left up to your application logic to handle.

You can still build indexes and unique constraints using redpipe SortedSets, Lists, Sets, Hashes etc. But you do so separately from Struct as their own first-class objects.

This allows you to access and control the indexes separately from the objects. Don't see this as a deficiency in the framework. See it as a feature.

Open Source Redis Cluster Support

RedPipe supports Redis Cluster.

```
import rediscluster
import redpipe

r = rediscluster.StrictRedisCluster(
    startup_nodes=[{'host': '0', 'port': 7000}],
)

redpipe.connect_redis(r, name='my-cluster')
```

The reason you can do this is because **RedPipe** wraps the interface.

If it quacks like a duck ...

RedPipe can support both the strict and normal interfaces. Because it is a wrapper, the commands buffer just as you send them. So if you wrap a StrictRedisCluster, the commands will be sent through as strict commands. If you wrap *RedisCluster* it follows that interface.

When you get to *Keyspaces*, **RedPipe** is more opinionated. You can use either StrictRedisCluster or RedisCluster. But it will present an interface more like the non-strict version.

Error Handling

Redis Pipelining and Errors

RedPipe is opinionated on this point. When we execute the pipeline, we always raise any errors from the redis client. Some of your commands may have run. Others may not. Any attached callbacks will not be triggered if an exception was raised. *Futures* will not have any results populated inside of them.

It is your job as an application developer when using redis to make your API behave in an idempotent way.

One way of handling this is to allow the exception to bubble up. When the call is retried later, make it pick up where it left off. Figure out how to repair any prior state and complete the operation. Also design your application to handle partially written records and handle them appropriately.

Another way is to try to roll back the changes. This is more difficult. Frankly I'm not exactly sure how it would work. I don't design my own applications this way. It seems like you could do it. But there's also a good chance that the problem that caused the exception may persist. And that multiple tries one way or another may not be able to restore you to a clean state. I think it's a losing battle.

If you choose redis, try to think about error cases and don't assume all the commands will proceed in lockstep. Ask yourself what could go wrong, and how might I recover from it when I read this dirty state,

Errors Raised by RedPipe

RedPipe raises exceptions of its own under the following scenarios:

- Trying to access the result of a Future object before it has been received.
- Misconfiguring the pipeline object.
- Invalid data type passed to a defined field in a Hash

Maybe there are others? Anyway, those are the ones that come to mind. If you run into an issue and don't understand it, let me know. I will update the documentation to help better explain it.

Unicode Support

If you use *RedPipe.pipeline* objects directly, you are writing raw bytes into redis and reading them out.

Go down a higher level of abstraction in the Keyspaces, and all keys and values are unicode characters stored as utf-8 bytes in redis. When we read the bytes out of redis we decode them back into strings in python.

Python 3 is much pickier about this. Python 2 doesn't force you to think about it and often does the right thing, but can be error prone.

I'm no expert at unicode or character encoding so if you see a bug let me know and I'll try to fix it.

Still working on more tests in this area.

Why not make all of the data utf-8 compliant?

There are some operations, like redis DUMP and redis RESTORE where the binary data shouldn't be decoded as unicode. It's a raw binary data representation. In other cases you may decide to pickle objects and store them in redis. *RedPipe* should be able to support all of this.

This part of the library is less mature than other aspects of the code.

Use at your own risk.

Please report any issues.

Licensing

The license should be the same as redis-py.

FAQ

Q: Why am I getting InvalidPipeline?

Have you configured your connection yet?

```
redis_client = redis.Redis()
redpipe.connect_redis(redis_client)
```

This will pass the redis connection to redpipe.

Q: I used *decode_responses* in redis and got an error in redpipe. WTF?

Short answer: I raised an exception on purpose.

I decided to be very opinionated. I don't want you to do this.

When talking to redis, you don't know for sure what you are getting. It might be binary data. It might not.

We wait to decode responses until we are down a layer when we know the data type of the keyspace we are using and the fields. That allows us to reuse the same connection to write and read binary data and still decode responses that should be.

If you feel this is wrong, let's chat. I'm open to discussion.

Raise it in issues.

Q: Why name it RedPipe? That's dumb.

Yeah.

I'm not the best at naming things.

Red is short for *Redis*. *Pipe* is short for *Pipelining*. Put the two together. *RedPipe.* That's the sum total of my thought process in naming my module.

Plus, no one had used it yet in *PyPi*.

:)

CHAPTER 7

Community Documentation

This part of the documentation explains the RedPipe ecosystem.

Testing

Testing is really important with any code library. It is especially important when working with database libraries. So much depends on them.

I try to be as thorough as I can in testing each facet of code.

All of the tests are contained in one file at the root of the repo:

./test.py

I could split it up, but it is convenient at the moment to have all the tests in one file. And it can find the path to the redpipe package without any special hoops to jump through.

If you see an area that has not been well tested, let me know.

Test Setup

Check out the code from GitHub.

Open a shell at the root of the repo.

Then type this command:

```
./activate
```

This will set up the virtualenv and install all the necessary test packages.

It also puts you in a shell with the virtualenv path declared.

Running the Tests

If you only want to run the test, you can just run the test script:

```
./test.py
```

When you are done, hit control-d to exit the shell.

Running Tests Against Supported Python Versions

To go through a more thorough test suite, run:

```
make test
```

This will run tox against a bunch of different python versions and print out coverage. To run this, you need the following python versions installed and discoverable in your path:

- python2.7
- python3.3
- python3.4
- python3.5

This will also print out code coverage statistics and lint tests.

I expect all of these code tests to pass fully before accepting patches to master.

Using Docker to Test

There's a docker image to help you set up all these versions of python. It will check them out and run tox.

To run the tests, type:

```
docker build . -t redpipe && docker run redpipe
```

If you need to jump in and debug stuff, do:

```
docker build . -t redpipe && docker run -it redpipe /bin/bash
```

Building Documentation

To build this documentation, there's a make command:

```
make documentation
```

This will run the *sphinx-build* command to create the local version of the docs. The docs are automatically published to Read the Docs. But it's handy to build locally before publishing.

Benchmarking

I'd like to write some standardized benchmarks on this stuff. But writing benchmarks is tricky.

I am confident these optimizations make a big difference. It has been proven over and over in real-world applications.

Nevertheless, benchmarks are a good idea.

It's on my list of things to do.

Inspiration

Josiah Carlson deserves some credit.

Long ago I took inspiration from the Redis object mapper project. I tried to rewrite his library for my purposes and did a pretty bad job of it. There were some absurd patch requests that were rightly rejected.

:-)

But I learned a lot in the process. Hopefully I've grown as a developer since then. I didn't want a full-fledged ORM but there were some excellent ideas there that provided a jumping off point. *RedPipe* and *Redis Object Mapper* have different use-cases. It is worth checking out.

Disclaimers

RedPipe is based on concepts I have been using in a well-tested production environment for a long time. But some implementation details are fairly new.

There will be bugs.

Please report all issues here.

I will respond to issues promptly. Make sure to provide clear explanations of what you are seeing and give steps to reproduce the bug.

Thread Safety

Thread safety is a stated goal of *RedPipe*.

Redis-Py is considered thread safe by using atomic operations against the GIL when accessing the connection pool. Redis-Py-Cluster uses similar mechanisms.

You should not share objects produced by *redpipe.pipeline()* between threads. The main issue you will run into is how it enters and exits the with block, resetting the command stack. Another issue is ordering of commands. Frankly, I just haven't tested this behavior and don't feel it is important to support it.

You can safely use a different *redpipe.pipeline()* in each thread after setting up your connection. This is because when the *redpipe.pipeline()* object executes, it obtains a new *redis.pipeline()* object to pass the commands into. That redis-py pipeline object queues all the commands and then obtains a connection from its pool in a thread-safe way. Then it packs the commands and sends it over the wire and waits for the response before releasing it back into the connection pool.

If you see any symptoms of unsafe thread behavior, please report it here.

Character Encoding

To be honest, I never spent a whole lot of time thinking about character encoding in redis until recently. Most of the values I manipulate in *redis* are numbers and simple ascii keys. And python 2 doesn't make you think about character encoding vs bytes much at all. However, I think a good library should fully support proper character encoding. And since RedPipe is fully tested on python 3, I am making more of an effort to understand the nuances.

If you find a bug, Please report it.

Lua Scripting

Lua scripting is only barely supported in *redis-py-cluster*. You can make it work if you don't bother with script registration or evalsha. That's because it is too hard to know for sure when running a command whether or not the node in the cluster will have it already. And it gets especially complicated in pipeline scenarios and pipelined failover scenarios.

So, we choose to always send the full lua script every time. If you use short lua scripts like I do, it's not a big deal. the network penalty of bytes over the wire is small compared with the penalty of multiple network round-trips. And on the redis server, it keeps an internal hash of the compiled lua script. So there's no additional compilation penalty with sending the Lua script every time.

I know it's not quite as nice this way. But at least it is functional. If you have any ideas on how to make this better, let me know.

Contributing

I welcome new ideas. And bug fixes. But I want to keep RedPipe clear and focused. Code bloat stinks.

The changes submitted should adhere to the following principles:

- modular component that does one job well
- allows for efficient network i/o talking to redis
- doesn't nest network round-trips that defeat the point of pipelining
- exposes the power of the redis API first and foremost
- KISS: keep it simple, stupid!

If any individual component starts to feel really complex, it's time to break it up. Or time to cut it.

For a patch to be accepted, it must pass all the unit tests and flake8 tests. It should do so for all supported versions of python.

That said, I'm happy to take rough patch requests and make them suitable for merging if the idea is good. And of course, I'm happy to give credit where it is due.

Release Notes

1.0.0 (May 11, 2017

No substantive changes from 1.0.0rc3. Updating notes and removing beta flags.

1.0.0rc3 (May 10, 2017)

Use threads by default when talking to multiple backends in the same pipeline. You can disable this behavior with *redpipe.disable_threads()*.

1.0.0rc2 (May 9, 2017)

Minor changes.

- make the keyspace object call conform to redis-py
- use twine to publish to pypi
- publish wheels

1.0.0rc1 (May 7, 2017)

This marks the first RC. There are a few breaking changes, mostly easily fixed.

- better handling of Nones returned from hmget in Struct
- testing with toxiproxy to simulate slower networks in benchmarks
- using pytest-benchmark tool for benchmark comparisons
- simplifying connections so we can pass in redis or rediscluster
- fixing some compat issues with redis-py interface

0.5.0 (May 5, 2017)

More breaking changes to Struct. Solidifying the api. Making important simplifications. This will make it easier to explain and document.

- Struct and Keyspace: simplifying some variable names
- Struct: support a no_op flag to prevent read/write from redis
- Struct: no kwargs as properties of struct. a dict models it better
- Struct: specify fields to load when instantiating
- Struct: reuse remove logic in the update function for elements set to None
- Simplifying task wait and promise to use the TaskManager directly
- Future: better isinstance and is comparison checks
- make it easier to build docs
- adding Docker support for testing many versions of python

0.4.0 (May 4, 2017)

- by default, don't use transactions
- autocommit flag renamed to autoexec. *Breaking change*.
- support pickling Struct
- make repr(Struct) more standard
- cleaner connection and pipeline interfaces
- verify redis cluster support with a single-node redis cluster via redislite

0.3.2 (May 3, 2017)

After experimenting with some things, simplifying Struct back down. Some of the methods in Struct will break. Easier to explain with fewer methods and can still do everything I need to.

- cleaner support for items and iteritems in struct
- support for delete in struct
- fixed a bug with deleting multiple keys in Keyspace objects.
- simplification on json serialization detection
- test flake8 on travis
- test with hiredis

This release also improves the documentation on Struct. I hadn't bothered much up until this point. The interface was still solidifying. Starting to get to a stable place there.

0.3.1 (May 2, 2017)

Breaking changes in this release as well. Can only access data from a struct object like you would a dictionary. This is an important step because it disambiguates commands from data. And it enforces one consistent way to access data. All the methods on the *Struct* give it a dictionary interface. Easier to explain the mental model this way.

- Improvements to *redpipe.Struct*.
- Documentation improvements.

0.3.0 (April 30, 2017)

BIG REFACTOR. key no longer part of the constructor of Keyspace objects. Instead, you pass the key name to the method. This keeps the api identical in arguments in redis-py. It also allows me to support multi-key operations. This is a breaking change.

- no need for a compat layer, using six
- standardize key, value, member encoding & decoding by reusing Field interface
- key no longer part of the constructor of Keyspace objects

0.2.5 (April 30, 2017)

- support for binary field
- improving encoding and decoding in Keyspaces
- alias iteritems to items on struct
- make fields use duck-typing to validate instead of using isinstance

0.2.4 (April 28, 2017)

- better interface for async enable/disable.
- add ability to talk to multiple redis servers in parallel via threads

0.2.3 (April 27, 2017)

- renaming datatypes to keyspaces. easier to explain.
- moving documentation from readme into docs/ for readthedocs.
- support for ascii field

0.2.2 (April 26, 2017)

- better support and testing of redis cluster
- support for hyperloglog data type
- adding support for more complex field types
- support sortedset lex commands
- support for scanning

0.2.1 (April 24, 2017)

- bug fix: make sure accessing result before ready results in a consistent exception type.
- bug fix: issue when exiting with statement from python cli

0.2.0 (April 24, 2017)

- make the deferred object imitate the underlying result

0.1.1 (April 23, 2017)

- make it possible to typecast fields in the Hash data type
- better support for utf-8
- make result object traceback cleaner

0.1.0 (April 21, 2017)

- better pipelining and task management
- better support for multi pipeline use case

Old Releases

Releases prior to **1.0.0** are considered beta. The api is not officially supported. We make no guarantees about backward compatibility.

Releases less than **0.1.0** in this project are considered early alpha and don't deserve special mention.

Project Authors

Lead author and maintainer: John Loehrer - https://github.com/72squared

If you contribute to this project I will be happy to mention you.

CHAPTER 8

API Documentation

This part of the documentation provides detailed API documentation. Dig into the source code and see how everything ties together. This is what is great about open-source projects. You can see everything.

redpipe package

Redpipe makes redis pipelines easier to use in python.

Usage:

```
import redpipe
import redis

redpipe.connect_redis(redis.Redis())
with redpipe.pipeline() as pipe:
    foo = pipe.incr('foo')
    bar = pipe.incr('bar)
    pipe.execute()
print([foo, bar])
```

Module Structure

This is the structure of the top level of the package, grouped by category.

Connections

- connect_redis
- disconnect
- reset

- pipeline
- autoexec

Fields

- IntegerField
- FloatField
- TextField
- AsciiField
- BinaryField
- BooleanField
- ListField
- DictField',
- StringListField

Keyspaces

- String
- Set
- List
- SortedSet
- Hash
- HyperLogLog

Exceptions

- Error
- ResultNotReady
- InvalidOperation
- InvalidValue
- AlreadyConnected
- InvalidPipeline

Misc

- Future
- Struct
- enable_threads
- disable_threads

You shouldn't need to import the submodules directly.

Submodules

redpipe.connections module

Bind instances of the redis-py or redis-py-cluster client to redpipe. Assign named connections to be able to talk to multiple redis servers in your project.

The ConnectionManager is a singleton class.

These functions are all you will need to call from your code:

- connect_redis
- disconnect
- reset

Everything else is for internal use.

`redpipe.connections.`**`connect_redis`**(*redis_client*, *name=None*, *transaction=False*)

Connect your redis-py instance to redpipe.

Example:

```
redpipe.connect_redis(redis.StrictRedis(), name='users')
```

Do this during your application bootstrapping.

You can also pass a redis-py-cluster instance to this method.

```
redpipe.connect_redis(rediscluster.StrictRedisCluster(), name='users')
```

You are allowed to pass in either the strict or regular instance.

```
redpipe.connect_redis(redis.StrictRedis(), name='a')
redpipe.connect_redis(redis.Redis(), name='b')
redpipe.connect_redis(rediscluster.StrictRedisCluster(...), name='c')
redpipe.connect_redis(rediscluster.RedisCluster(...), name='d')
```

Parameters

- **`redis_client`** –
- **`name`** – nickname you want to give to your connection.
- **`transaction`** –

Returns

`redpipe.connections.`**`disconnect`**(*name=None*)

remove a connection by name. If no name is passed in, it assumes default.

```
redpipe.disconnect('users')
redpipe.disconnect()
```

Useful for testing.

Parameters **`name`** –

Returns None

redpipe.connections.**reset**()
remove all connections.

```
redpipe.reset()
```

Useful for testing scenarios.

Not sure when you'd want to call this explicitly unless you need an explicit teardown of your application. In most cases, python garbage collection will do the right thing on shutdown and close all the redis connections.

Returns None

redpipe.exceptions module

This module contains the set of all of redpipe exceptions.

exception redpipe.exceptions.**Error**
Bases: exceptions.Exception

Base class for all redpipe errors

exception redpipe.exceptions.**ResultNotReady**
Bases: *redpipe.exceptions.Error*

Raised when you access a data from a Future before it is assigned.

exception redpipe.exceptions.**InvalidOperation**
Bases: *redpipe.exceptions.Error*

Raised when trying to perform an operation disallowed by the redpipe api.

exception redpipe.exceptions.**InvalidValue**
Bases: *redpipe.exceptions.Error*

Raised when data assigned to a field is the wrong type

exception redpipe.exceptions.**AlreadyConnected**
Bases: *redpipe.exceptions.Error*

raised when you try to connect and change the ORM connection without explicitly disconnecting first.

exception redpipe.exceptions.**InvalidPipeline**
Bases: *redpipe.exceptions.Error*

raised when you try to use a pipeline that isn't configured correctly.

redpipe.fields module

A module for marshalling data in and out of redis and back into the python data type we expect.

Used extensively in the *redpipe.keyspaces* module for type-casting keys and values.

class redpipe.fields.**IntegerField**
Bases: object

Used for integer numeric fields.

classmethod decode(*value*)
read bytes from redis and turn it back into an integer.

Parameters **value** – bytes

Returns int

classmethod encode(*value*)
take an integer and turn it into a string representation to write into redis.

Parameters **value** – int

Returns str

class redpipe.fields.**FloatField**
Bases: object

Numeric field that supports integers and floats (values are turned into floats on load from persistence).

classmethod decode(*value*)
decode the bytes from redis back into a float

Parameters **value** – bytes

Returns float

classmethod encode(*value*)
encode a floating point number to bytes in redis

Parameters **value** – float

Returns bytes

class redpipe.fields.**TextField**
Bases: object

A unicode string field.

Encoded via utf-8 before writing to persistence.

classmethod decode(*value*)
take bytes from redis and turn them into unicode string

Parameters **value** –

Returns

classmethod encode(*value*)
take a valid unicode string and turn it into utf-8 bytes

Parameters **value** – unicode, str

Returns bytes

class redpipe.fields.**AsciiField**
Bases: *redpipe.fields.TextField*

Used for ascii-only text

PATTERN = <_sre.SRE_Pattern object>

classmethod encode(*value*)
take a list of strings and turn it into utf-8 byte-string

Parameters **value** –

Returns

class redpipe.fields.**BinaryField**
Bases: object

A bytes field. Not encoded.

classmethod decode(*value*)
read binary data from redis and pass it on through.

Parameters **value** – bytes

Returns bytes

classmethod encode(*value*)
write binary data into redis without encoding it.

Parameters **value** – bytes

Returns bytes

class redpipe.fields.**BooleanField**
Bases: object

Used for boolean fields.

classmethod decode(*value*)
convert from redis bytes into a boolean value

Parameters **value** – bytes

Returns bool

classmethod encode(*value*)
convert a boolean value into something we can persist to redis. An empty string is the representation for False.

Parameters **value** – bool

Returns bytes

class redpipe.fields.**ListField**
Bases: object

A list field. Marshalled in and out of redis via json. Values of the list can be any arbitrary data.

classmethod decode(*value*)
take a utf-8 encoded byte-string from redis and turn it back into a list

Parameters **value** – bytes

Returns list

classmethod encode(*value*)
take a list and turn it into a utf-8 encoded byte-string for redis.

Parameters **value** – list

Returns bytes

class redpipe.fields.**DictField**
Bases: object

classmethod decode(*value*)
decode the data from a json string in redis back into a dict object.

Parameters **value** – bytes

Returns dict

classmethod encode(*value*)
encode the dict as a json string to be written into redis.

Parameters **value** – dict

Returns bytes

class redpipe.fields.**StringListField**
Bases: object

Used for storing a list of strings, serialized as a comma-separated list.

classmethod decode(*value*)
decode the data from redis. :param value: bytes :return: list

classmethod encode(*value*)
the list it so it can be stored in redis.

Parameters **value** – list

Returns bytes

redpipe.futures module

The *Future()* object in **RedPipe** gives us the ability to make the pipeline interface of redis-py look like the non-pipelined interface. You call a command and get a response back. Only the response is not the actual data. It is an empty container called a *Future*. There is a callback attached to that empty container. When the pipeine is executed, the pipeline injects the response into the container.

This Future container is a very special kind of python object. It can imitate anything it contains. If there is an integer inside, it behaves like an integer. If it holds a dictionary, it behaves like a dictionary. If it holds a list, it behaves like a list. Your application should be able to use it interchangeably.

There are a few gotchas to watch out for:

- isinstance() checks
- identity checks like: future is None
- trying to mutate the object like this: future += 1

You can always type cast the object into the type you expect if you need this behavior.

```
f = Future()
f.set(1)

# f is 1 fails
assert(int(f) is 1)
```

This doesn't work so well for is None checks. You can use equality checks though. Or you can use our handy IS method. Or you can access the underlying result

```
f = Future()
f.set(None)

assert(f == None)
assert(f.IS(None))
assert(f.result is None)
```

Hope that helps.

Other than those few caveats, you should be able to access a future object just like the underlying result.

Here are some examples if your result is numeric.

```
future = Future()
future.set(1)
assert(future == 1)
assert(future != 2)
assert(bool(future))
assert(float(future) == 1.0)
assert(future + 1 == 2)
assert(future * 2 == 2)
assert(future ^ 1 == 0)
assert(repr(future) == '1')
```

And here is an example if your future is a list:

```
future = Future()
future.set([1])
assert(future == [1])
assert([v for v in future] == [1])
assert(future + [2] == [1, 2])
```

And here is a dictionary:

```
future = Future()
future.set({'a': 1})
assert(future == {'a': 1})
assert(dict(future) == {'a': 1})
assert({k: v for k, v in future.items()} == {'a': 1})
```

There are many more operations supported but these are the most common. Let me know if you need more examples or explanation.

class redpipe.futures.**Future**

Bases: object

An object returned from all our Pipeline calls.

IS(*other*)

Allows you to do identity comparisons on the underlying object.

Parameters **other** – Mixed

Returns bool

id()

Get the object id of the underlying result.

isinstance(*other*)

allows you to check the instance type of the underlying result.

Parameters **other** –

Returns

result

Get the underlying result. Usually one of the data types returned by redis-py.

Returns None, str, int, list, set, dict

set(*data*)

Write the data into the object. Note that I intentionally did not declare *result* in the constructor. I want an error to happen if you try to access it before it is set.

Parameters **data** – any python object

Returns None

redpipe.futures.**IS**(*instance*, *other*)
Support the *future is other* use-case. Can't override the language so we built a function. Will work on non-future objects too.

Parameters

- **instance** – future or any python object
- **other** – object to compare.

Returns

redpipe.futures.**ISINSTANCE**(*instance*, *A_tuple*)
Allows you to do isinstance checks on futures. Really, I discourage this because duck-typing is usually better. But this can provide you with a way to use isinstance with futures. Works with other objects too.

Parameters

- **instance** –
- **A_tuple** –

Returns

redpipe.keyspaces module

This module provides a way to access keys grouped under a certain keyspace. A keyspace is a convention used often in redis where many keys are grouped logically together. In the SQL world, you could think of this as a table. But in redis each key is independent whereas a record in a table is controlled by the schema.

Examples of a group of keys in a keyspace:

- user{A}
- user{B}
- user{C}

It is inconvient to refer to keys this way. The identifiers for our user records are A, B, C. In addition, we usually know that a user record is always a redis hash. And we know that it has certain fields that have different data types.

These keyspace classes in this module allow you to easily manipulate these keys.

```
redpipe.connect_redis(redis.Redis(
        # connection params go here.
    ), name='user_redis_db')

class User(redpipe.Hash):
    keyspace = 'user'
    fields = {
        'name': redpipe.TextField,
        'created_at': redpipe.TextField,
    }
    connection = 'user_redis_db'


user_a = User().hgetall('A')
```

This Keyspace object exposes all the hash-related redis commands as normal. Internally, it rewrites the key name to be 'user{A}' for you automatically. You can pass in a pipeline to the constructor. No matter what pipeline you pass in, it routes your commands to the *user_redis_db* that you set up.

There's also support for character encoding and complex data types.

class `redpipe.keyspaces.`**`String`**(*pipe=None*)

Bases: `redpipe.keyspaces.Keyspace`

Manipulate a String key in Redis.

`append`(*name*, *value*)

Appends the string `value` to the value at `key`. If `key` doesn't already exist, create it with a value of `value`. Returns the new length of the value at `key`.

Parameters

- **`name`** – str the name of the redis key
- **`value`** – str

Returns Future()

`bitcount`(*name*, *start=None*, *end=None*)

Returns the count of set bits in the value of `key`. Optional `start` and `end` paramaters indicate which bytes to consider

Parameters

- **`name`** – str the name of the redis key
- **`start`** – int
- **`end`** – int

Returns Future()

`get`(*name*)

Return the value of the key or None if the key doesn't exist

Parameters **`name`** – str the name of the redis key

Returns Future()

`getbit`(*name*, *offset*)

Returns a boolean indicating the value of `offset` in key

Parameters

- **`name`** – str the name of the redis key
- **`offset`** – int

Returns Future()

`incr`(*name*, *amount=1*)

increment the value for key by 1

Parameters

- **`name`** – str the name of the redis key
- **`amount`** – int

Returns Future()

`incrby`(*name*, *amount=1*)

increment the value for key by value: int

Parameters

- **name** – str the name of the redis key
- **amount** – int

Returns Future()

incrbyfloat(*name*, *amount=1.0*)
increment the value for key by value: float

Parameters

- **name** – str the name of the redis key
- **amount** – int

Returns Future()

psetex(*name*, *value*, *time_ms*)
Set the value of key `name` to `value` that expires in `time_ms` milliseconds. `time_ms` can be represented by an integer or a Python timedelta object

set(*name*, *value*, *ex=None*, *px=None*, *nx=False*, *xx=False*)
Set the value at key `name` to `value`

`ex` sets an expire flag on key `name` for `ex` seconds.

`px` sets an expire flag on key `name` for `px` milliseconds.

`nx` if set to True, set the value at key `name` to `value` if it does not already exist.

`xx` if set to True, set the value at key `name` to `value` if it already exists.

Returns Future()

setbit(*name*, *offset*, *value*)
Flag the `offset` in the key as `value`. Returns a boolean indicating the previous value of `offset`.

Parameters

- **name** – str the name of the redis key
- **offset** – int
- **value** –

Returns Future()

setex(*name*, *value*, *time*)
Set the value of key to `value` that expires in `time` seconds. `time` can be represented by an integer or a Python timedelta object.

Parameters

- **name** – str the name of the redis key
- **value** – str
- **time** – secs

Returns Future()

setnx(*name*, *value*)
Set the value as a string in the key only if the key doesn't exist.

Parameters

- **name** – str the name of the redis key

- **value** –

Returns Future()

setrange(*name*, *offset*, *value*)
Overwrite bytes in the value of `name` starting at `offset` with `value`. If `offset` plus the length of `value` exceeds the length of the original value, the new value will be larger than before. If `offset` exceeds the length of the original value, null bytes will be used to pad between the end of the previous value and the start of what's being injected.

Returns the length of the new string. :param name: str the name of the redis key :param offset: int :param value: str :return: Future()

strlen(*name*)
Return the number of bytes stored in the value of the key

Parameters **name** – str the name of the redis key

Returns Future()

substr(*name*, *start*, *end=-1*)
Return a substring of the string at key `name`. `start` and `end` are 0-based integers specifying the portion of the string to return.

Parameters

- **name** – str the name of the redis key
- **start** – int
- **end** – int

Returns Future()

class redpipe.keyspaces.**Set**(*pipe=None*)
Bases: redpipe.keyspaces.Keyspace

Manipulate a Set key in redis.

sadd(*name*, *values*, **args*)
Add the specified members to the Set.

Parameters

- **name** – str the name of the redis key
- **values** – a list of values or a simple value.

Returns Future()

scard(*name*)
How many items in the set?

Parameters **name** – str the name of the redis key

Returns Future()

sdiff(*keys*, **args*)
Return the difference of sets specified by `keys`

Parameters

- **keys** – list
- **args** – tuple

Returns Future()

sdiffstore(*dest*, **keys*)
Store the difference of sets specified by `keys` into a new set named `dest`. Returns the number of keys in the new set.

sinter(*keys*, **args*)
Return the intersection of sets specified by `keys`

Parameters

- **keys** – list or str
- **args** – tuple

Returns Future

sinterstore(*dest*, *keys*, **args*)
Store the intersection of sets specified by `keys` into a new set named `dest`. Returns the number of keys in the new set.

sismember(*name*, *value*)
Is the provided value is in the `Set`?

Parameters

- **name** – str the name of the redis key
- **value** – str

Returns Future()

smembers(*name*)
get the set of all members for key

Parameters **name** – str the name of the redis key

Returns

spop(*name*)
Remove and return (pop) a random element from the Set.

Parameters **name** – str the name of the redis key

Returns Future()

srandmember(*name*, *number=None*)
Return a random member of the set.

Parameters **name** – str the name of the redis key

Returns Future()

srem(*name*, **values*)
Remove the values from the Set if they are present.

Parameters

- **name** – str the name of the redis key
- **values** – a list of values or a simple value.

Returns Future()

sscan(*name*, *cursor=0*, *match=None*, *count=None*)
Incrementally return lists of elements in a set. Also return a cursor indicating the scan position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

Parameters

- **name** – str the name of the redis key
- **cursor** – int
- **match** – str
- **count** – int

sscan_iter(*name*, *match=None*, *count=None*)
Make an iterator using the SSCAN command so that the client doesn't need to remember the cursor position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

Parameters

- **name** – str the name of the redis key
- **match** – str
- **count** – int

sunion(*keys*, **args*)
Return the union of sets specified by `keys`

Parameters

- **keys** – list or str
- **args** – tuple

Returns Future()

sunionstore(*dest*, *keys*, **args*)
Store the union of sets specified by `keys` into a new set named `dest`. Returns the number of members in the new set.

class redpipe.keyspaces.**List**(*pipe=None*)
Bases: redpipe.keyspaces.Keyspace

Manipulate a List key in redis

blpop(*keys*, *timeout=0*)
LPOP a value off of the first non-empty list named in the `keys` list.

If none of the lists in `keys` has a value to LPOP, then block for `timeout` seconds, or until a value gets pushed on to one of the lists.

If timeout is 0, then block indefinitely.

brpop(*keys*, *timeout=0*)
RPOP a value off of the first non-empty list named in the `keys` list.

If none of the lists in `keys` has a value to LPOP, then block for `timeout` seconds, or until a value gets pushed on to one of the lists.

If timeout is 0, then block indefinitely.

brpoplpush(*src*, *dst*, *timeout=0*)
Pop a value off the tail of `src`, push it on the head of `dst` and then return it.

This command blocks until a value is in `src` or until `timeout` seconds elapse, whichever is first. A `timeout` value of 0 blocks forever.

lindex(*name*, *index*)
Return the value at the index *idx*

Parameters

- **name** – str the name of the redis key
- **index** – the index to fetch the value.

Returns Future()

llen(*name*)
Returns the length of the list.

Parameters **name** – str the name of the redis key

Returns Future()

lpop(*name*)
Pop the first object from the left.

Parameters **name** – str the name of the redis key

Returns Future()

lpush(*name*, **values*)
Push the value into the list from the *left* side

Parameters

- **name** – str the name of the redis key
- **values** – a list of values or single value to push

Returns Future()

lrange(*name*, *start*, *stop*)
Returns a range of items.

Parameters

- **name** – str the name of the redis key
- **start** – integer representing the start index of the range
- **stop** – integer representing the size of the list.

Returns Future()

lrem(*name*, *value*, *num=1*)
Remove first occurrence of value.

Can't use redis-py interface. It's inconstistent between redis.Redis and redis.StrictRedis in terms of the kwargs. Better to use the underlying execute_command instead.

Parameters

- **name** – str the name of the redis key
- **num** –
- **value** –

Returns Future()

lset(*name*, *index*, *value*)
Set the value in the list at index *idx*

Parameters

- **name** – str the name of the redis key
- **value** –
- **index** –

Returns Future()

ltrim(*name*, *start*, *end*)
Trim the list from start to end.

Parameters

- **name** – str the name of the redis key
- **start** –
- **end** –

Returns Future()

rpop(*name*)
Pop the first object from the right.

Parameters **name** – str the name of the redis key

Returns the popped value.

rpoplpush(*src*, *dst*)
RPOP a value off of the `src` list and atomically LPUSH it on to the `dst` list. Returns the value.

rpush(*name*, **values*)
Push the value into the list from the *right* side

Parameters

- **name** – str the name of the redis key
- **values** – a list of values or single value to push

Returns Future()

class `redpipe.keyspaces.`**SortedSet**(*pipe=None*)
Bases: `redpipe.keyspaces.Keyspace`

Manipulate a SortedSet key in redis.

zadd(*name*, *members*, *score=1*, *nx=False*, *xx=False*, *ch=False*, *incr=False*)
Add members in the set and assign them the score.

Parameters

- **name** – str the name of the redis key
- **members** – a list of item or a single item
- **score** – the score the assign to the item(s)
- **nx** –
- **xx** –
- **ch** –
- **incr** –

Returns Future()

zcard(*name*)
Returns the cardinality of the SortedSet.

Parameters **name** – str the name of the redis key

Returns Future()

zincrby(*name*, *value*, *amount=1*)
Increment the score of the item by *value*

Parameters

- **name** – str the name of the redis key
- **value** –
- **amount** –

Returns

zlexcount(*name*, *min*, *max*)
Return the number of items in the sorted set between the lexicographical range `min` and `max`.

Parameters

- **name** – str the name of the redis key
- **min** – int or '-inf'
- **max** – int or '+inf'

Returns Future()

zrange(*name*, *start*, *end*, *desc=False*, *withscores=False*, *score_cast_func=<type 'float'>*)
Returns all the elements including between `start` (non included) and `stop` (included).

Parameters

- **name** – str the name of the redis key
- **start** –
- **end** –
- **desc** –
- **withscores** –
- **score_cast_func** –

Returns

zrangebylex(*name*, *min*, *max*, *start=None*, *num=None*)
Return the lexicographical range of values from sorted set `name` between `min` and `max`.

If `start` and `num` are specified, then return a slice of the range.

Parameters

- **name** – str the name of the redis key
- **min** – int or '-inf'
- **max** – int or '+inf'
- **start** – int
- **num** – int

Returns Future()

zrangebyscore(*name*, *min*, *max*, *start=None*, *num=None*, *withscores=False*, *score_cast_func=<type 'float'>*)

Returns the range of elements included between the scores (min and max)

Parameters

- **name** – str the name of the redis key
- **min** –
- **max** –
- **start** –
- **num** –
- **withscores** –
- **score_cast_func** –

Returns Future()

zrank(*name*, *value*)

Returns the rank of the element.

Parameters

- **name** – str the name of the redis key
- **value** – the element in the sorted set

zrem(*name*, **values*)

Remove the values from the SortedSet

Parameters

- **name** – str the name of the redis key
- **values** –

Returns True if **at least one** value is successfully removed, False otherwise

zremrangebylex(*name*, *min*, *max*)

Remove all elements in the sorted set between the lexicographical range specified by `min` and `max`.

Returns the number of elements removed. :param name: str the name of the redis key :param min: int or -inf :param max: into or +inf :return: Future()

zremrangebyrank(*name*, *min*, *max*)

Remove a range of element between the rank `start` and `stop` both included.

Parameters

- **name** – str the name of the redis key
- **min** –
- **max** –

Returns Future()

zremrangebyscore(*name*, *min*, *max*)

Remove a range of element by between score `min_value` and `max_value` both included.

Parameters

- **name** – str the name of the redis key
- **min** –

- **max** –

Returns Future()

zrevrange(*name*, *start*, *end*, *withscores=False*, *score_cast_func=<type 'float'>*)
Returns the range of items included between `start` and `stop` in reverse order (from high to low)

Parameters

- **name** – str the name of the redis key
- **start** –
- **end** –
- **withscores** –
- **score_cast_func** –

Returns

zrevrangebylex(*name*, *max*, *min*, *start=None*, *num=None*)

Return the reversed lexicographical range of values from the sorted set between `max` and `min`.

If `start` and `num` are specified, then return a slice of the range.

Parameters

- **name** – str the name of the redis key
- **max** – int or '+inf'
- **min** – int or '-inf'
- **start** – int
- **num** – int

Returns Future()

zrevrangebyscore(*name*, *max*, *min*, *start=None*, *num=None*, *withscores=False*, *score_cast_func=<type 'float'>*)
Returns the range of elements between the scores (min and max).

If `start` and `num` are specified, then return a slice of the range.

`withscores` indicates to return the scores along with the values. The return type is a list of (value, score) pairs

score_cast_func' a callable used to cast the score return value

Parameters

- **name** – str the name of the redis key
- **max** – int
- **min** – int
- **start** – int
- **num** – int
- **withscores** – bool
- **score_cast_func** –

Returns Future()

zrevrank(*name*, *value*)
Returns the ranking in reverse order for the member

Parameters

- **name** – str the name of the redis key
- **member** – str

zscan(*name*, *cursor=0*, *match=None*, *count=None*, *score_cast_func=<type 'float'>*)
Incrementally return lists of elements in a sorted set. Also return a cursor indicating the scan position.

`match` allows for filtering the members by pattern

`count` allows for hint the minimum number of returns

`score_cast_func` a callable used to cast the score return value

zscan_iter(*name*, *match=None*, *count=None*, *score_cast_func=<type 'float'>*)
Make an iterator using the ZSCAN command so that the client doesn't need to remember the cursor position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

`score_cast_func` a callable used to cast the score return value

zscore(*name*, *value*)
Return the score of an element

Parameters

- **name** – str the name of the redis key
- **value** – the element in the sorted set key

Returns Future()

zunionstore(*dest*, *keys*, *aggregate=None*)
Union multiple sorted sets specified by `keys` into a new sorted set, `dest`. Scores in the destination will be aggregated based on the `aggregate`, or SUM if none is provided.

class `redpipe.keyspaces.`**Hash**(*pipe=None*)
Bases: `redpipe.keyspaces.Keyspace`

Manipulate a Hash key in Redis.

fields = {}

hdel(*name*, **keys*)
Delete one or more hash field.

Parameters

- **name** – str the name of the redis key
- **keys** – on or more members to remove from the key.

Returns Future()

hexists(*name*, *key*)
Returns `True` if the field exists, `False` otherwise.

Parameters

- **name** – str the name of the redis key

- **key** – the member of the hash

Returns Future()

hget(*name*, *key*)
Returns the value stored in the field, None if the field doesn't exist.

Parameters

- **name** – str the name of the redis key
- **key** – the member of the hash

Returns Future()

hgetall(*name*)
Returns all the fields and values in the Hash.

Parameters **name** – str the name of the redis key

Returns Future()

hincrby(*name*, *key*, *amount=1*)
Increment the value of the field.

Parameters

- **name** – str the name of the redis key
- **increment** – int
- **field** – str

Returns Future()

hincrbyfloat(*name*, *key*, *amount=1.0*)
Increment the value of the field.

Parameters

- **name** – str the name of the redis key
- **key** – the name of the ement in the hash
- **amount** – float

Returns Future()

hkeys(*name*)
Returns all fields name in the Hash.

Parameters **name** – str the name of the redis key

Returns Future

hlen(*name*)
Returns the number of elements in the Hash.

Parameters **name** – str the name of the redis key

Returns Future()

hmget(*name*, *keys*, **args*)
Returns the values stored in the fields.

Parameters

- **name** – str the name of the redis key

- **fields** –

Returns Future()

hmset(*name*, *mapping*)
Sets or updates the fields with their corresponding values.

Parameters

- **name** – str the name of the redis key
- **mapping** – a dict with keys and values

Returns Future()

hscan(*name*, *cursor=0*, *match=None*, *count=None*)
Incrementally return key/value slices in a hash. Also return a cursor indicating the scan position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

hscan_iter(*name*, *match=None*, *count=None*)
Make an iterator using the HSCAN command so that the client doesn't need to remember the cursor position.

`match` allows for filtering the keys by pattern

`count` allows for hint the minimum number of returns

hset(*name*, *key*, *value*)
Set `member` in the Hash at `value`.

Parameters

- **name** – str the name of the redis key
- **value** –
- **key** – the member of the hash key

Returns Future()

hsetnx(*name*, *key*, *value*)
Set `member` in the Hash at `value`.

Parameters

- **name** – str the name of the redis key
- **value** –
- **key** –

Returns Future()

hvals(*name*)
Returns all the values in the Hash Unfortunately we can't type cast these fields. it is a useless call anyway imho.

Parameters **name** – str the name of the redis key

Returns Future()

class `redpipe.keyspaces.`**HyperLogLog**(*pipe=None*)
Bases: `redpipe.keyspaces.Keyspace`

Manipulate a HyperLogLog key in redis.

pfadd(*name*, **values*)
Adds the specified elements to the specified HyperLogLog.

Parameters

- **name** – str the name of the redis key
- **values** – list of str

pfcount(**sources*)
Return the approximated cardinality of the set observed by the HyperLogLog at key(s).

Parameters **name** – str the name of the redis key

pfmerge(*dest*, **sources*)
Merge N different HyperLogLogs into a single one.

Parameters

- **dest** –
- **sources** –

Returns

redpipe.luascripts module

Some utility lua scripts used to extend some functionality in redis. It also let's me exercise the eval code path a bit.

redpipe.pipelines module

This is where the magic happens. The most important components of redpipe are here. The Pipeline and NestedPipeline classes and the pipeline function enable Use to pass pipeline functions into each other and attach redis calls to them.

The main function exposed here is the *pipeline* function. You will use it everywhere, so get used to this syntax:

```
def incr(name, pipe=None):
    with redpipe.autoexec(pipe=pipe) as pipe:
        return pipe.incr(name)

with redpipe.autoexec() as pipe:
    a = incr('a', pipe=pipe)
    b = incr('b', pipe=pipe)

print([a, b])
```

Look at the *incr* function. The call to *redpipe.pipeline* will return a *Pipeline* object if None is passed in. And if a Pipeline object is passed in, it will return a *NestedPipeline* object. Those two objects present the same interface but behave very differently.

Pipeline objects execute your pipelined calls. *NestedPipeline* objects pass their commands up the chain to the parent pipeline they wrap. This could be another *NestedPipeline* object, or a Pipeline() object.

redpipe.pipelines.**pipeline**(*pipe=None*, *name=None*, *autoexec=False*)
This is the foundational function for all of redpipe. Everything goes through here. create pipelines, nest pipelines, get pipelines for a specific name. It all happens here.

Here's a simple example:

```
with pipeline() as pipe:
    pipe.set('foo', 'bar')
    foo = pipe.get('foo')
    pipe.execute()
print(foo)
> bar
```

Now let's look at how we can nest a pipeline.

```
def process(key, pipe=None):
    with pipeline(pipe, autoexec=True) as pipe:
        return pipe.incr(key)

with pipeline() as pipe:
    key1 = process('key1', pipe)
    key2 = process('key2', pipe)
    pipe.execute()

print([key1, key2])

> [1, 1]
```

Parameters

- **pipe** – a Pipeline() or NestedPipeline() object, or None
- **name** – str, optional. the name of the connection to use.
- **autoexec** – bool, if true, implicitly execute the pipe

Returns Pipeline or NestedPipeline

redpipe.pipelines.**autoexec**(*pipe=None*, *name=None*)

create a pipeline with a context that will automatically execute the pipeline upon leaving the context if no exception was raised.

Parameters

- **pipe** –
- **name** –

Returns

redpipe.structs module

The Struct is a convenient way to access data in a hash. Makes it possible to load data from redis as an object and access the fields. Then store changes back into redis.

class redpipe.structs.**Struct**(*_key_or_data*, *pipe=None*, *fields=None*, *no_op=False*)

Bases: object

load and store structured data in redis using OOP patterns.

If you pass in a dictionary-like object, redpipe will write all the values you pass in to redis to the key you specify.

By default, the primary key name is *_key*. But you should override this in your Struct with the *key_name* property.

```
class Beer(redpipe.Struct):
    fields = {'name': redpipe.TextField}
    key_name = 'beer_id'

beer = Beer({'beer_id': '1', 'name': 'Schlitz'})
```

This will store the data you pass into redis. It will also load any additional fields to hydrate the object. **RedPipe** does this in the same pipelined call.

If you need a stub record that neither loads or saves data, do:

```
beer = Beer({'beer_id': '1'}, no_op=True)
```

You can later load the fields you want using, load.

If you pass in a string we assume it is the key of the record. redpipe loads the data from redis:

```
beer = Beer('1')
assert(beer['beer_id'] == '1')
assert(beer['name'] == 'Schlitz')
```

If you need to load a record but only specific fields, you can say so.

```
beer = Beer('1', fields=['name'])
```

This will exclude all other fields.

RedPipe cares about pipelining and efficiency, so if you need to bundle a bunch of reads or writes together, by all means do so!

```
beer_ids = ['1', '2', '3']
with redpipe.pipeline() as pipe:
    beers = [Beer(i, pipe=pipe) for i in beer_ids]
print(beers)
```

This will pipeline all 3 together and load them in a single pass from redis.

The following methods all accept a pipe:

- __init__
- update
- incr
- decr
- pop
- remove
- clear
- delete

You can pass a pipeline into them to make sure that the network i/o is combined with another pipeline operation. The other methods on the object are about accessing the data already loaded. So you shouldn't need to pipeline them.

clear(*pipe=None*)

delete the current redis key.

Parameters **pipe** –

Returns

connection = None

copy()
like the dictionary copy method.

Returns

decr(*field*, *amount=1*, *pipe=None*)
Inverse of incr function.

Parameters

- **field** –
- **amount** –
- **pipe** –

Returns Pipeline, NestedPipeline, or None

default_fields = 'all'

classmethod delete(*keys*, *pipe=None*)
Delete one or more keys from the Struct namespace.

This is a class method and unlike the *clear* method, can be invoked without instantiating a Struct.

Parameters

- **keys** – the names of the keys to remove from the keyspace
- **pipe** – Pipeline, NestedPipeline, or None

Returns None

fields = {}

get(*item*, *default=None*)
works like the dict get method.

Parameters

- **item** –
- **default** –

Returns

incr(*field*, *amount=1*, *pipe=None*)
Increment a field by a given amount. Return the future

Also update the field.

Parameters

- **field** –
- **amount** –
- **pipe** –

Returns

items()
We return the list of key/value pair tuples. Similar to iteritems but in list form instead of as a generator. The reason we do this is because python2 code probably expects this to be a list. Not sure if I could care, but just covering my bases.

Example:

```
u = User('1')
data = {k: v for k, v in u.items()}
```

Or:

```
u = User('1')
for k, v in u.items():
    print("%s: %s" % (k, v)
```

Returns list, containing key/value pair tuples.

iteritems()
Support for the python 2 iterator of key/value pairs. This includes the primary key name and value.

Example:

```
u = User('1')
data = {k: v for k, v in u.iteritems()}
```

Or:

```
u = User('1')
for k, v in u.iteritems():
    print("%s: %s" % (k, v)
```

Returns generator, a list of key/value pair tuples

key

key_name = '_key'

keys()
Get a list of all the keys in the Struct. This includes the primary key name, and all the elements that are set into redis.

Note: even if you define fields on the Struct, those keys won't be returned unless the fields are actually written into the redis hash.

```
u = User('1')
assert(u.keys() == ['_key', 'name'])
```

Returns list

keyspace = None

load(*fields=None*, *pipe=None*)
Load data from redis. Allows you to specify what fields to load. This method is also called implicitly from the constructor.

Parameters

- **fields** – 'all', 'defined', or array of field names
- **pipe** – Pipeline(), NestedPipeline() or None

Returns None

persisted
Not certain I want to keep this around. Is it useful?

Returns

pop(*name*, *default=None*, *pipe=None*)
works like the dictionary pop method.

IMPORTANT!

This method removes the key from redis. If this is not the behavior you want, first convert your Struct data to a dict.

Parameters

- **name** –
- **default** –
- **pipe** –

Returns

remove(*fields*, *pipe=None*)
remove some fields from the struct. This will remove data from the underlying redis hash object. After the pipe executes successfully, it will also remove it from the current instance of Struct.

Parameters

- **fields** – list or iterable, names of the fields to remove.
- **pipe** – Pipeline, NestedPipeline, or None

Returns None

update(*changes*, *pipe=None*)
update the data in the Struct.

This will update the values in the underlying redis hash. After the pipeline executes, the changes will be reflected here in the local struct.

If any values in the changes dict are None, those fields will be removed from redis and the instance.

The changes should be a dictionary representing the fields to change and the values to change them to.

Parameters

- **changes** – dict
- **pipe** – Pipeline, NestedPipeline, or None

Returns None

redpipe.tasks module

When sending commands to multiple redis backends in one redpipe.pipeline, this module gives us an api to allow threaded async communication to those different backends, improving parallelism.

The AsynchronousTask is well tested and should work well. But if you see any issues, you can easily disable this in your application.

```
redpipe.disable_threads()
```

Please report any issues.

redpipe.tasks.**enable_threads**()
used to enable threaded behavior when talking to multiple redis backends in one pipeline execute call. Otherwise we don't need it. :return: None

redpipe.tasks.**disable_threads**()
used to disable threaded behavior when talking to multiple redis backends in one pipeline execute call. Use this option if you are really concerned about python threaded behavior in your application. Doesn't apply if you are only ever talking to one redis backend at a time. :return: None

redpipe.version module

Utility for grabbing the redpipe version string from the VERSION file.

Python Module Index

r

Index

I

K

L

P

R

S

T

U

Z