redgrease

Anders Åström

May 03, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION

RedGrease is a Python client and runtime package attempting to make it as easy as possible to create and execute *RedisGears* functions on *Redis* engines with the *RedisGears Module* loaded.

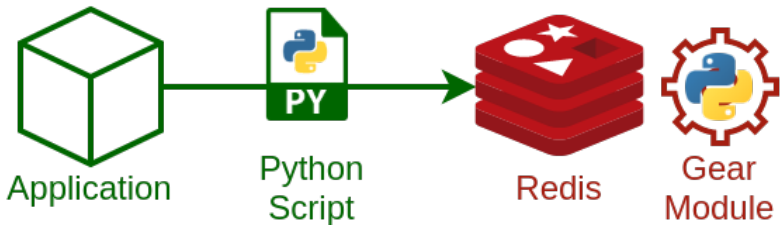


Fig. 1: Overview

RedGrease makes it easy to write concise but expressive Python functions to query and/or react to data in Redis in realtime. The functions are automatically distributed and run across the shards of the Redis cluster (if any), providing an excellent balance of performance of distributed computations and expressiveness and power of Python.

It may help you create:

- Advanced analytical queries,
- Event based and streaming data processing,
- Custom Redis commands and interactions,
- And much, much more. . .

. . . all written in Python and running distributed ON your Redis nodes.

The Gear functions may include and use third party dependencies like for example `numpy`, `requests`, `gensim` or pretty much any other Python package distribution you may need for your use-case.

If you are already familiar with Redis and RedisGears, then you can jump directly to the *What is RedGrease?* overview or the *Quickstart Guide*, otherwise you can read on to get up to speed on these technologies.

1.1 What is Redis?

Redis is a popular in-memory data structure store, used as a distributed, in-memory, key–value database, cache and message broker, with optional durability, horizontal scaling and high availability. Redis supports different kinds of abstract data structures, such as strings, lists, maps, sets, sorted sets, HyperLogLogs, bitmaps, streams, and spatial indexes. The project is developed and maintained by RedisLabs. It is open-source software released under a BSD 3-clause license.

1.2 What is Redis Gears?

RedisGears is an official extension module for Redis, also developed by RedisLabs, which allows for distributed Python computations on the Redis server itself.

From the official RedisGears site:

"RedisGears is a dynamic framework that enables developers to write and execute functions that implement data flows in Redis, while abstracting away the data's distribution and deployment. These capabilities enable efficient data processing using multiple models in Redis with infinite programmability, while remaining simple to use in any environment."

When the Redis Gears module is loaded onto the Redis engines, the Redis engine command set is extended with new commands to register, distribute, manage and run so called *Gear Functions*, written in Python, across across the shards of the Redis database.

Client applications can define and submit such Python Gear Functions, either to run immediately as 'batch jobs', or to be registered to be triggered on events, such as Redis keyspace changes, stream writes or external triggers. The Redis Gears module handles all the complexities of distribution, coordination, scheduling, execution and result collection and aggregation, of the Gear Functions.

1.2.1 What are Gear Functions?

Gear Functions are composed as a sequence of steps, or operations, such as for example Map, Filter, Aggregate, GroupBy and more.

These operations are parameterized with Python functions, that you define according to your needs.

The steps / operations are 'piped' together by the Redis Gears runtime such that the output of of one step / operation becomes the input to the subsequent step / operation, and so on.

The first step / operation of any Gear Function is always one of six available "Readers", defining the source of the input to the first step / operation:

- *KeysReader* : Redis keys and values.
- *KeysOnlyReader* : Redis keys.
- *StreamReader* : Redis Stream messages.
- *PythonReader* : Arbitrary Python generator.
- *ShardsIDReader* : Shard ID.
- *CommandReader* : Command arguments from application client.

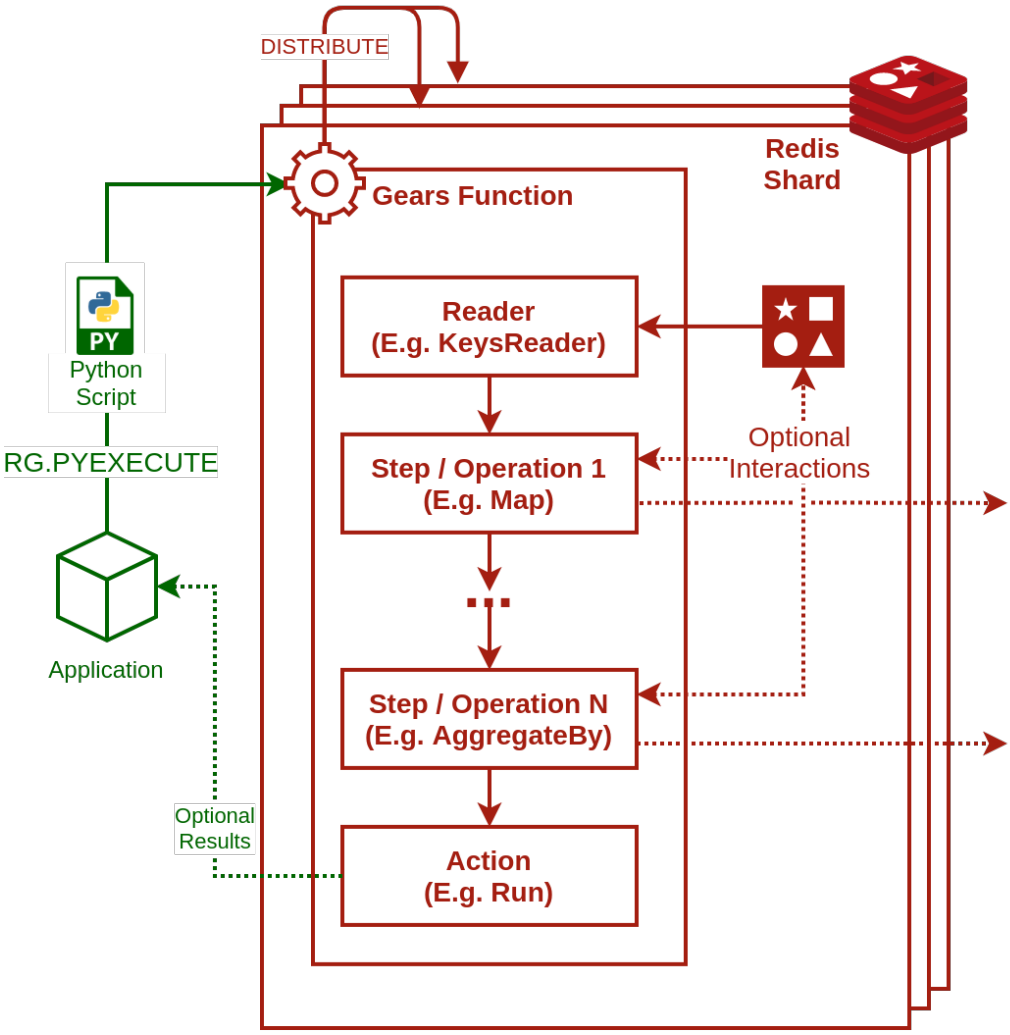


Fig. 2: Redis Gears Processing Pipeline Overview

Readers can be parameterized to narrow down the subset of data it should operate on, for example by specifying a pattern for the keys or streams it should read.

Depending on the reader type, Gear Functions can either be run immediately, on demand, as batch jobs or in an event-driven manner by registering it to trigger automatically on various types of events.

Each shard of the Redis Cluster executes its own 'instance' of the Gear Function in parallel on the relevant local shard data, unless explicit collected, or until it is implicitly reduced to its final global result at the end of the function.

You can find more details about the internals of Gear Functions in the official Documentation.

1.3 What is RedGrease?

The RedGrease package provides a number of functionalities that facilitates writing and executing Gear Functions:

1. *Redis / Redis Gears client(s)*.

   Extended versions of the redis Python client and redis-py-cluster Python client clients, but with additional pythonic functions, mapping closely (1-to-1) to the *Redis Gears command set* (e.g. `RG.PYEXECUTE`, `RG.GETRESULTS`, `RG.TRIGGER`, `RG.DUMPREGISTRATIONS` etc), outlined in the official Gears documentation.

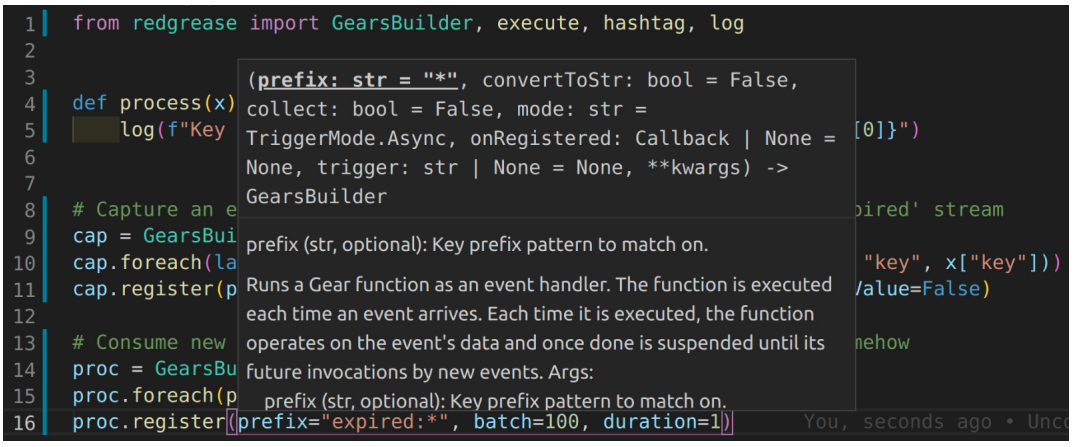
   ```
   import redgrease

   gear_script = ... # Gear function string, a GearFunction object or a␣
   ↪script file path.

   rg = redgrease.RedisGears()
   rg.gears.pyexecute(gear_script)   # <-- RG.PYEXECUTE
   ```

2. *Runtime functions* wrappers.

   The RedisGears server runtime environment automatically loads a number of special functions into the top level scope (e.g. `GearsBuilder`, `execute()`, `log()` etc). RedGrease provides placeholder versions that provide **docstrings**, **auto completion** and **type hints** during development, and does not clash with the actual runtime.

3. *Server-side Redis commands*.

   Allowing for *most* Redis (v.6) commands to be executed in the server-side function, against the local shard, as if using a Redis 'client' class, instead of *explicitly* invoking the corresponding command string using `execute()`. It is basically the redis Python client, but with `redis.Redis`.

`execute_command()` rewired to use the Gears-native *`redgrease.runtime.execute()`* instead under the hood.

```
import redgrease


# This function runs **on** the Redis server.
def download_image(annotation):
    img_id = annotation["image_id"]
    img_key = f"image:{img_id}"
    if redgrease.cmd.hexists(img_key, "image_data"):  # <- hexists
        # image already downloaded
        return img_key
    redgrease.log(f"Downloadign image for annotation: {annotation}")
    image_url = redgrease.cmd.hget(img_key, "url")  # <- hget
    response = requests.get(image_url)
    redgrease.cmd.hset(img_key, "image_data", bytes(response.content))  #
↪<- hset
    return img_key


# Redis connection (with Gears)
connection = redgrease.RedisGears()

# Automatically download corresponding image, whenever an annotation is␣
↪created.
image_keys = (
    redgrease.KeysReader()
    .values(type="hash", event="hset")
    .foreach(download_image, requirements=["requests"])
    .register("annotation:*", on=connection)
)
```

4. First class *GearFunction* objects.

   Inspired by the "remote builders" of the official redisgears-py client, but with some differences, eg:

   - Supports reuse of *Open GearFunction*, i.e. partial or incomplete Gear functions.
   - Can be *created without a Redis connection*.
   - *Requirements can be specified per step*, instead of only at execution.
   - Can be *executed in a few different convenient ways*.

```
import redgrease


def schedule(record):
    status = record.value.get("status", "new")
    redgrease.log(f"Scheduling '{status}' record: {record.key}")
    if status == "new":
        record.value["status"] = "pending"
        redgrease.cmd.hset(record.key, "status", "pending")
        redgrease.cmd.xadd("to_be_processed", {"record": record.key})
    ...
    return record
```

(continues on next page)

(continued from previous page)

```
def process(item):
    redgrease.log(f"processsing {item}")
    success = len(item["record"]) % 3  # Mock processing
    redgrease.cmd.hset(item["record"], "status", "success" if success␣
↪else "failed")


def has_status(status):
    return lambda record: record.value.get("status", None) == status


key_pattern = "record:*"

records = redgrease.KeysReader().records(type="hash")

record_listener = records.foreach(schedule).register(key_pattern,␣
↪eventTypes=["hset"])

get_failed = records.filter(has_status("failed"))

count_by_status = (
    records.countby(lambda r: r.value.get("status", "unknown"))
    .map(lambda r: {r["key"]: r["value"]})
    .aggregate({}, lambda a, r: dict(a, **r))
    .run(key_pattern)
)

process_records = (
    redgrease.StreamReader()
    .values()
    .foreach(process, requirements=["numpy"])
    .register("to_be_processed")
)

server = redgrease.RedisGears()

# Different ways of executing
server.gears.pyexecute(record_listener)
process_records.on(server)

failed = get_failed.run(key_pattern, on=server)
count = count_by_status.on(server)
```

5. *A Command Line Tool*.

   Helps running and/or loading of Gears script files onto a RedisGears instance. Particularly useful for "trigger-based" CommandReader Gears.

   It also provides a simple form of 'hot-reloading' of RedisGears scripts, by continuously monitoring directories containing Redis Gears scripts and automatically 'pyexecute' them on a Redis Gear instance if it detects modifications.

   The purpose is mainly to streamline development of 'trigger-style' Gear scripts by providing a form of hot-reloading functionality.

```
redgrease --server 10.0.2.21 --watch scripts/
```

6. A bunch of helper functions and methods for common boilerplate tasks.
    - A `redgrease.utils` module full of utils such as parsers etc.
    - Various *Syntactic sugar* and enum-like objects for common keywords etc.
    - A *Command Function Decorator*, that makes creation and execution of *`redgrease.reader.CommandReader` GearFunction* trivial, and providing a straight forward way of adding bespoke server-side Redis commands.
    - Reader-specific sugar operators, like *`KeysReader.values`* that automatically lifts out the values.
    - And more...

1.4 Example Use-Cases

The possible use-cases for Redis Gears, and subsequently RedGrease, is virtually endless, but some common, or otherwise interesting use-cases include:

- Automatic Cache-miss handling.

  Make Redis automatically fetch and cache the requested resource, so that clients do not have to handle cache-misses.

- Automatic batched write-through / write-behind.

  Make Redis automatically write back updates to slower, high latency datastore, efficiently using batch writes. Allowing clients to write high velocity updates uninterrupted to Redis, without bothering with the slow data store.

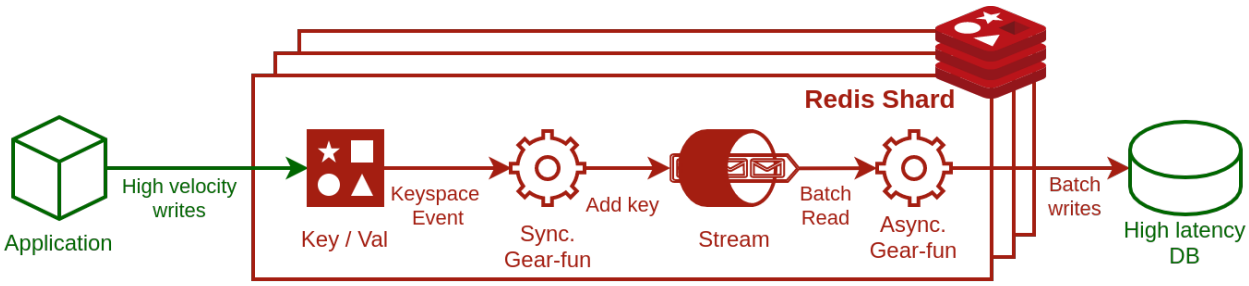


Fig. 3: Write-Through / Write-Behind example

- Advanced Data Queries and Transforms.

  Perform "Map-Reduce"-like queries on Redis datasets.

- Stream event processing.

  Trigger processes automatically when data enters Redis.

- Custom commands.

  Create custom Redis commands with arbitrarily sophisticated logic, enabling features to virtually any platform with a Redis client implementation.

1.5 Glossary

Gear Function Gear Function, written as two separate words, refer to any valid Gear function, as defined in the Redis Gears Documentation, regardless if it was constructed as a pure string, loaded from a file, or programmatically built using RedGrease's `GearFunction` constructors.

GearFunction GearFunction, written as one word, refers specifically to RedGrease objects of type `redgrease.GearFunction`.

These are constructed programmatically using either `redgrease.GearsBuilder`, any of the Reader classes such as `redgrease.KeysReader`, `redgrease.StreamReader`, `redgrease.CommandReader` etc, or function decorators such as `redgrease.trigger` and so on.

It does **not** refer to Gear Functions that are loaded from strings, either explicitly or from files.

CHAPTER

TWO

QUICKSTART GUIDE

This section aims to get you started within a few couple of minutes, while still explaining what is going on, so that someone with only limited experience with Python can follow along.

Setup TL;DR

POSIX (Linux, BSD, OSX, etc)

The Windows

```
docker run --name redis_gears --rm -d -p 127.0.0.1:6379:6379 redislabs/redisgears:1.0.
↪6

virtualenv -p python3.7 .venv
source .venv/bin/activate

pip install redgrease[all]
```

```
docker run --name redis_gears --rm -d -p 127.0.0.1:6379:6379 redislabs/redisgears:1.0.
↪6

virtualenv -p python3.7 .venv
\venv\Scripts\activate.bat

pip install redgrease[all]
```

Note: This is not tested. If anyone is using this OS, please let me know if this works or not. :)

If this was obvious to you, you can *jump straight to the first code examples*.

2.1 Running Redis Gears

The easiest way to run a Redis Engine with Redis Gears is by running one of the official Docker images, so firstly make sure that you have Docker engine installed.

(Eh? I've been living under a rock. What the hedge is Docker?)

With Docker is installed, open a terminal or command prompt and enter:

```
docker run --name redis_gears --rm -d -p 127.0.0.1:6379:6379 redislabs/redisgears:1.0.
↪6
```

This will run a single Redis engine, with the Redis Gears module loaded, inside a Docker container. The first time you issue the command it may take a little time to launch, as Docker needs to fetch the container image from Docker Hub.

Lets break the command down:

- `docker run` is the base command telling Docker that we want to run a new containerized process.
- `--name redis_gears` gives the name "redis_gears" to the container for easier identification. You can change it for whatever you like or omit it to assign a randomized name.
- `--rm` instructs Docker that we want the container to be removed, in case it stops. This is optional but makes starting and stopping easer, although the state (and stored data) will be lost between restarts.
- `-d` instructs Docker to run the container in the background as a 'daemon'. You can omit this too, but your terminal / command prompt will be hijacked by the output / logs from the container. Which could be interesting enough.
- `-p 127.0.0.1:6379:6379` instructs Docker that we want your host computer to locally (127.0.0.1) expose port 6379 and route it to 6379 in the container. This is the default port for Redis communication and this argument is necessary for application on your computer to be able to talk to the Redis engine inside the Docker container.
- `redislabs/redisgears:1.0.6` is the name and version of the Docker image that we want to run. This specific image is prepared by RedisLabs and has the Gears module pre-installed and configured.

Note: If its your first time trying Redis Gears, stick to the command above, but if you want to try running a cluster image instead, you can issue the following command:

```
docker run --name redis_gears_cluster --rm -d -p 127.0.0.1:3001:30001 -p 127.0.0.
↪1:3002:30002 -p 127.0.0.1:3003:30003 redislabs/rgcluster:1.0.6
```

This will run a 3-shard cluster exposed locally on ports 30001-30003.

Refer to the official documentation for more information and details on how how to install Redis Gears.

2.1.1 Checking Logs:

You can confirm that the container is running by inspecting the logs / output of the Redis Gears container by issuing the command:

```
docker logs redis_gears
```

- You can optionally add the argument `--follow` to continuously follow the log output.
- You can optionally add the argument `--tail 100` to start showing the logs from the 100 most recent entries only.

If you just started the single instance engine, the logs should hold 40 odd lines starting and ending something like this:

```
1:C 03 Apr 2021 07:41:37.250 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
1:C 03 Apr 2021 07:41:37.251 # Redis version=6.0.1, bits=64, commit=00000000,␣
↪modified=0, pid=1, just started
...
...
...
1:M 03 Apr 2021 07:41:37.309 * Module 'rg' loaded from /var/opt/redislabs/lib/modules/
↪redisgears.so
1:M 03 Apr 2021 07:41:37.309 * Ready to accept connections
```

2.1.2 Stopping

You can stop the container by issuing:

```
docker stop redis_gears
```

If successful, it should simply output the name of the stopped container: `redis_gears`

2.2 RedGrease Installation

For the client application environment, it is strongly recommended that you set up a virtual Python environment, with Python 3.7 specifically.

Note: The Redis Gears containers above use Python 3.7 for executing Gear functions, and using the same version the client application will enable more features.

Warning: The RedGrease client package works with any Python version from 3.6 and later, but *execution of dynamically created GearFunction objects* is only possible when the client Python version match the Python version on the Redis Gears server runtime.

If the versions mismatch, Gear function execution is limited to *execution by string* or *execution of script files*

With Python 3.7, and virtualenv installed on your system:

1. Create a virtual python3.7 environment

   ```
   virtualenv -p python3.7 .venv
   ```

   Python packages, including RedGrease that you install within this virtual environment will not interfere with the rest of your system.

2. Activate the environment

   POSIX (Linux, BSD, OSX, etc)

   The Windows

   ```
   source .venv/bin/activate
   ```

   ```
   .venv\Scripts\activate.bat
   ```

3. Install redgrease

   ```
   pip install redgrease[all]
   ```

   Note: The `[all]` portion is important, as it will include all the Redgrease extras, and include the dependencies for the RedisGears client module as well as the RedGrease Command Line Interface (CLI).

   See *here* for more details on the various extras options.

2.3 Basic Commands

In this section we'll walk through some of the basic commands and interactions with the RedGrease Gears client, including executing some very basic Gear functions.

The next chapter "*RedGrease Client*", goes into all commands in more details, but for now we'll just look at the most important things.

You can take a sneak-peek at the full code that we will walk through in this section, by expanding the block below (click " Show").

If you find this rather self-explanatory, then you can probably jump directly to the next section where we do some *RedGrease Gear Function Comparisons* with "vanilla" RedisGears functions.

Otherwise just continue reading and we'll, go through it step-by-step.

Full code of this section.

Listing 1: From examples/basics.py on the official GitHub repo:

```python
from operator import add

import redgrease

# Create connection / client for single instance Redis
r = redgrease.RedisGears()

# # Normal Redis Commands
# Clearing the database
r.flushall()

# String operations
r.set("Foo-fighter", 2021)
r.set("Bar-fighter", 63)
r.set("Baz-fighter", -747)

# Hash Opertaions
r.hset("noodle", mapping={"spam": "eggs", "meaning": 8})
r.hincrby("noodle", "meaning", 34)

# Stream operations
r.xadd("transactions:0", {"msg": "First", "from": 0, "to": 2, "amount": 1000})


# # Redis Gears Commands
# Get Statistics on the Redis Gears Python runtime
gears_runtime_python_stats = r.gears.pystats()
print(f"Gears Python runtime stats: {gears_runtime_python_stats}")

# Get info on any registered Gear functions, if any.
registered_gear_functions = r.gears.dumpregistrations()
print(f"Registered Gear functions: {registered_gear_functions}")

# Execute nothing as a Gear function
empty_function_result = r.gears.pyexecute()
print(f"Result of nothing: {empty_function_result}")

# Execute a Gear function string that just iterates through and returns the key-space.
all_records_gear = r.gears.pyexecute("GearsBuilder('KeysReader').run()")
```

(continues on next page)

(continued from previous page)

```
print("All-records gear results: [")
for result in all_records_gear:
    print(f"  {result}")
print("]")

# Gear function string to count all the keys
key_count_gearfun_str = "GearsBuilder('KeysReader').count().run()"
key_count_result = r.gears.pyexecute(key_count_gearfun_str)
print(f"Total number of keys: {int(key_count_result)}")


# # GearFunctions
# GearFunction object to count all keys
key_count_gearfun = redgrease.KeysReader().count().run()
key_count_result = r.gears.pyexecute(key_count_gearfun)
print(f"Total number of keys: {key_count_result}")


# Simple Aggregation
add_gear = redgrease.KeysReader("*-fighter").values().map(int).aggregate(0, add)
simple_sum = add_gear.run(on=r)
print(f"Multiplication of '-fighter'-keys values: {simple_sum}")
```

Let's look at some code examples of how to use RedGrease, warming up with the basics.

2.3.1 Instantiation

Naturally, the first thing is to import of the RedGrease package and *instantiate Redis Gears client / connection object*:

Listing 2: Package import and client / connection instantiation:

```
from operator import add

import redgrease

# Create connection / client for single instance Redis
r = redgrease.RedisGears()

```

This will attempt to connect to a Redis server using the default port (6379) on "localhost", which, if you followed the instructions above should be exactly what you set up and have running. There are of course arguments to set other targets, but more on that later.

The imported `add` function from the `operator` module is not part of the RedgGrease package, but we will use it later in one of the examples.

Note: If you created *a Redis cluster* above then you have to specify the initial master nodes you want to connect to:

```
import redgrease

r = redgrease.RedisGears(port=30001)
```

2.3.2 Redis Commands

The instantiated client / connection, `r`, accepts all the normal Redis commands, exactly as expected. The subsequent lines populate the Redis instance it with some data.

Listing 3: Some normal Redis commands:

```
# Clearing the database
r.flushall()

# String operations
r.set("Foo-fighter", 2021)
r.set("Bar-fighter", 63)
r.set("Baz-fighter", -747)

# Hash Opertaions
r.hset("noodle", mapping={"spam": "eggs", "meaning": 8})
r.hincrby("noodle", "meaning", 34)

# Stream operations
r.xadd("transactions:0", {"msg": "First", "from": 0, "to": 2, "amount": 1000})


```

2.3.3 Gears Commands

The client / connection also has a `gears` attribute that gives access to *RedisGears Commands*.

Listing 4: Some Redis Gears commands:

```
# Get Statistics on the Redis Gears Python runtime
gears_runtime_python_stats = r.gears.pystats()
print(f"Gears Python runtime stats: {gears_runtime_python_stats}")

# Get info on any registered Gear functions, if any.
registered_gear_functions = r.gears.dumpregistrations()
print(f"Registered Gear functions: {registered_gear_functions}")

# Execute nothing as a Gear function
empty_function_result = r.gears.pyexecute()
print(f"Result of nothing: {empty_function_result}")

# Execute a Gear function string that just iterates through and returns the key-space.
all_records_gear = r.gears.pyexecute("GearsBuilder('KeysReader').run()")
print("All-records gear results: [")
for result in all_records_gear:
    print(f"  {result}")
print("]")

# Gear function string to count all the keys
key_count_gearfun_str = "GearsBuilder('KeysReader').count().run()"
key_count_result = r.gears.pyexecute(key_count_gearfun_str)
print(f"Total number of keys: {int(key_count_result)}")


```

The highlighted lines show the commands *`Gears.pystats()`*, *`Gears.dumpregistrations()`* and *`Gears.`*

pyexecute() respectively and the output should look something like this:

```
Gears Python runtime stats: PyStats(TotalAllocated=41275404, PeakAllocated=11867779,␣
↪CurrAllocated=11786368)
Registered Gear functions: []
Result of nothing: True
All-records gear results: [
    b"{'event': None, 'key': 'Baz-fighter', 'type': 'string', 'value': '-747'}"
    b"{'event': None, 'key': 'Bar-fighter', 'type': 'string', 'value': '63'}"
    b"{'event': None, 'key': 'transactions:0', 'type': 'unknown', 'value': None}"
    b"{'event': None, 'key': 'Foo-fighter', 'type': 'string', 'value': '2021'}"
    b"{'event': None, 'key': 'noodle', 'type': 'hash', 'value': {'meaning': '42',
↪'spam': 'eggs'}}"
]
Total number of keys: 5
```

The command *Gears.pystats()* gets some memory usage statistics about the Redis Gears Python runtime environment on the server.

The command *Gears.dumpregistrations()* gets information about any registered Gears functions, in this cas none.

And finally, the command *Gears.pyexecute()* is the most important command, which sends a Gears function to the server for execution or registration. In the above example, we are invoking it three times:

- Firstly (line 35) - We pass nothing, i.e. no function at all, which naturally doesn't do anything, but is perfectly valid, and the call thus just returns `True`.
- Secondly (line 39) - We execute a *Raw Function String*, that reads through the Redis keys (indicated by the `'KeysReader'`) and just returns the result by running the function as a batch job (indicated by the `run()` operation). The result is consequently a list of dicts, representing the keys and their respective values and types in the Redis keyspace, I.e. the keys we added just before.
- Thirdly (lines 46-47) - We pass a very similar function, but with an additional `count()` operation, which is a Gear operation that simply aggregates and counts the incoming records, in this case all key-space records on the server. The result is simply the number or keys in the database: `5`.

There are other Gears commands too, and the next chapter, "*Redgrease Client*", will run through all of them.

2.3.4 GearFunctions

Composing Gear functions by using strings is not at all very practical, so RedGrease provides a more convenient way of constructing Gear functions programmatically, using various *GearFunction* objects.

Listing 5: GearFunction objects instead of strings:

```
52 # GearFunction object to count all keys
53 key_count_gearfun = redgrease.KeysReader().count().run()
54 key_count_result = r.gears.pyexecute(key_count_gearfun)
55 print(f"Total number of keys: {key_count_result}")
56
57
```

This Gear function does the same thing as the last function of the previous example, but instead of being composed by a string, it is composed programmatically using RedGrease's GearFunction objects, in this case using the *KeysReader* class.

The output is, just as expected:

```
Total number of keys: 5
```

Warning: Note that execution of GearFunction objects only work if your local Python environment version matches the version on the Redis Gear server, i.e. Python 3.7.

If the versions mismatch, Gear function execution is limited to *execution by string* or *execution of script files*

The final basic example shows a GearFunction that has a couple of operations stringed together.

Listing 6: Simple aggregation - Add keyspace values:

```
59 add_gear = redgrease.KeysReader("*-fighter").values().map(int).aggregate(0, add)
60 simple_sum = add_gear.run(on=r)
61 print(f"Multiplication of '-fighter'-keys values: {simple_sum}")
```

This Gear function adds the values of all the simple keys, with names ending in "-fighter", which were the first three keys created in the example.

And indeed, the result is:

```
Sum of '-fighter'-keys values: 1337
```

Here is a quick run down of how it works:

- Firstly, the *`KeysReader`* is parameterized with a key pattern `*-fighter` meaning it will only read the matching keys.
- Secondly, the *`map()`* operation uses a simple lambda function, to lift out the `value` and ensure it is an integer, on each of the keys.
- Thirdly, the *`aggregate()`* operation is used to add the values together, using the imported `add` function, starting from the value 0.
- Lastly, the *`run()`* operation is used to specify that the function should run as a batch job. The `on` argument states that we want to run it immediately on our client / connection, `r`.

The chapter "*Readers*" will go through the various types of readers, and the chapter operations will go through the various types of operations, and how to use them.

2.4 RedGrease Gear Function Comparisons

Now let's move on to some more examples of smaller Gear functions, before we move on to some more elaborate examples.

The examples in this section are basically comparisons of how the examples in the official Gears documentation, could be simplified by using RedGrease.

Note: RedGrease is backwards compatible with the "vanilla" syntax and structure, and all versions below are still perfectly valid Gear functions when executing using RedGrease.

2.4.1 Word Count

Counting of words.

Assumptions

All keys store Redis String values. Each value is a sentence.

Vanilla Version

This is the the 'Word Count' example from the official RedisGears documentation.

Listing 7: Vanilla - Word Count

```
gb = GearsBuilder()
gb.map(lambda x: x["value"])  # map records to "sentence" values
gb.flatmap(lambda x: x.split())  # split sentences to words
gb.countby()  # count each word's occurances
gb.run()
```

RedGrease Version

This is an example of how the same Gear function could be rewritten using RedGrease.

Listing 8: RedGrease - Word Count

```
from redgrease import KeysReader

KeysReader().values().flatmap(str.split).countby().run()
```

2.4.2 Delete by Key Prefix

Deletes all keys whose name begins with a specified prefix and return their count.

Assumptions

There may be keys in the database. Some of these may have names beginning with the "delete_me:" prefix.

Vanilla Version

This is the the 'Delete by Key Prefix' example from the official RedisGears documentation.

Listing 9: Vanilla - Delete by Key Prefix

```
gb = GearsBuilder()
gb.map(lambda x: x["key"])  # map the records to key names
gb.foreach(lambda x: execute("DEL", x))  # delete each key
gb.count()  # count the records
gb.run("delete_me:*")
```

RedGrease Version

This is an example of how the same Gear function could be rewritten using RedGrease.

Listing 10: RedGrease - Delete by Key Prefix

```
from redgrease import KeysReader, cmd

delete_fun = KeysReader().keys().foreach(cmd.delete).count()
delete_fun.run("delete_me:*")
```

2.4.3 Basic Redis Stream Processing

Copy every new message from a Redis Stream to a Redis Hash key.

Assumptions

An input Redis Stream is stored under the "mystream" key.

Vanilla Version

This is the the 'Basic Redis Stream Processing' example from the official RedisGears documentation.

Listing 11: Vanilla - Basic Redis Stream Processing

```
gb = GearsBuilder("StreamReader")
gb.foreach(
    lambda x: execute("HMSET", x["id"], *sum([[k, v] for k, v in x.items()], []))
)  # write to Redis Hash
gb.register("mystream")
```

RedGrease Version

This is an example of how the same Gear function could be rewritten using RedGrease.

Listing 12: RedGrease - Basic Redis Stream Processing

```
from redgrease import StreamReader, cmd

StreamReader().foreach(lambda x: cmd.hmset(x["id"], x)).register(  # write to Redis␣
↪Hash
    "mystream"
)
```

2.4.4 Automatic Expiry

Sets the time to live (TTL) for every updated key to one hour.

Assumptions

None.

Vanilla Version

This is the the 'Automatic Expiry' example from the official RedisGears documentation.

Listing 13: Vanilla - Automatic Expiry

```
gb = GB()
gb.foreach(lambda x: execute("EXPIRE", x["key"], 3600))
gb.register("*", mode="sync", readValue=False)
```

RedGrease Version

This is an example of how the same Gear function could be rewritten using RedGrease.

Listing 14: RedGrease - Automatic Expiry

```
from redgrease import KeysReader, cmd

expire = KeysReader().keys().foreach(lambda x: cmd.expire(x, 3600))
expire.register("*", mode="sync", readValue=False)
```

2.4.5 Keyspace Notification Processing

This example demonstrates a two-step process that:

1. Synchronously captures distributed keyspace events
2. Asynchronously processes the events' stream

Assumptions

The example assumes there is a `process` function defined, that does the actual processing of the deleted records. For the purpose of the example we can assume that it just outputs the name of the expired keys to the Redis logs, as follows:

```
def process(x):
    """
    Processes a message from the local expiration stream
    Note: in this example we simply print to the log, but feel free to replace
    this logic with your own, e.g. an HTTP request to a REST API or a call to an
    external data store.
    """
    log(f"Key '{x['value']['key']}' expired at {x['id'].split('-')[0]}")
```

Vanilla Version

This is the the 'Keyspace Notification Processing' example from the official RedisGears documentation.

Listing 15: Vanilla - Keyspace Notification Processing

```
# Capture an expiration event and adds it to the shard's local 'expired' stream
cap = GB("KeysReader")
cap.foreach(lambda x: execute("XADD", f"expired:{hashtag()}", "*", "key", x["key"]))
cap.register(prefix="*", mode="sync", eventTypes=["expired"], readValue=False)

# Consume new messages from expiration streams and process them somehow
proc = GB("StreamReader")
proc.foreach(process)
proc.register(prefix="expired:*", batch=100, duration=1)
```

RedGrease Version

This is an example of how the same Gear function could be rewritten using RedGrease.

Listing 16: RedGrease - Keyspace Notification Processing

```
from redgrease import KeysReader, StreamReader, cmd, hashtag, log

# Capture an expiration event and adds it to the shard's local 'expired' stream
KeysReader().keys().foreach(
    lambda key: cmd.xadd(f"expired:{hashtag()}", {"key": key})
).register(prefix="*", mode="sync", eventTypes=["expired"], readValue=False)

# Consume new messages from expiration streams and process them somehow
StreamReader().foreach(process).register(prefix="expired:*", batch=100, duration=1)
```

2.4.6 Reliable Keyspace Notification

Capture each keyspace event and store to a Stream.

Assumptions

...

Vanilla Version

This is the the 'Reliable Keyspace Notification' example from the official RedisGears documentation.

Listing 17: Vanilla - Reliable Keyspace Notification

```
GearsBuilder().foreach(
    lambda x: execute(
        "XADD", "notifications-stream", "*", *sum([[k, v] for k, v in x.items()], [])
    )
).register(prefix="person:*", eventTypes=["hset", "hmset"], mode="sync")
```

RedGrease Version

This is an example of how the same Gear function could be rewritten using RedGrease.

Listing 18: RedGrease - Reliable Keyspace Notification

```
from redgrease import KeysReader, cmd

KeysReader().foreach(lambda x: cmd.xadd("notifications-stream", x)).register(
    prefix="person:*", eventTypes=["hset", "hmset"], mode="sync"
)
```

2.5 Cache Get Command

As a final example of this quickstart tutorial, let's look at how we can build caching into Redis as a new command, with the help of Redis Gears and RedGrease.

Full code.

It may look a bit intimidating at first, but theres actually not not that much to it. Most of it is just comments, logging or testing code.

Listing 19: Simple Caching command:

```
import timeit

import redgrease

# Bind / register the function on some Redis instance.
r = redgrease.RedisGears()


# CommandReader Decorator
# The `command` decorator tunrs the function to a CommandReader,
# registerered on the Redis Gears sever if using the `on` argument
@redgrease.command(on=r, requirements=["requests"], replace=False)
def cache_get(url):
    import requests

    # Check if the url is already in the cache,
    # And if so, simply return the cached result.
    if redgrease.cmd.exists(url):
        return bytes(redgrease.cmd.get(url))

    # Otherwise fetch the url.
    response = requests.get(url)

    # Return nothing if request fails
    if response.status_code != 200:
        return bytes()

    # If ok, set the cache data and return.
    response_data = bytes(response.content)
    redgrease.cmd.set(url, response_data)

    return response_data


# Test caching on some images
```

(continues on next page)

(continued from previous page)

```
some_image_urls = [
    "http://images.cocodataset.org/train2017/000000483381.jpg",
    "http://images.cocodataset.org/train2017/000000237137.jpg",
    "http://images.cocodataset.org/train2017/000000017267.jpg",
    "http://images.cocodataset.org/train2017/000000197756.jpg",
    "http://images.cocodataset.org/train2017/000000193332.jpg",
    "http://images.cocodataset.org/train2017/000000475564.jpg",
    "http://images.cocodataset.org/train2017/000000247368.jpg",
    "http://images.cocodataset.org/train2017/000000416337.jpg",
]


# Get all the images and write them to disk
def get_em_all():

    for image_url in some_image_urls:

        # This will invoke the cache_get function **on the Redis server**
        image_data = cache_get(image_url)

        # Quick and dirty way of getting the image file name.
        image_name = image_url.split("/")[-1]

        # Write to file
        with open(image_name, "wb") as img_file:
            img_file.write(image_data.value)


# Test it
# Time how long it takes to get images when the cache is empty.
t1 = timeit.timeit(get_em_all, number=1)
print(f"Cache-miss time: {t1:.3f} seconds")

# Time how long it takes to get the images when they are all in the cache.
t2 = timeit.timeit(get_em_all, number=1)
print(f"Cache-hit time: {t2:.3f} seconds")
print(f"That is {t1/t2:.1f} times faster!")


# Clean the database
def cleanup(r: redgrease.RedisGears):

    # Unregister all registrations
    for reg in r.gears.dumpregistrations():
        r.gears.unregister(reg.id)

    # Remove all executions
    for exe in r.gears.dumpexecutions():
        r.gears.dropexecution(str(exe.executionId))

    # Clear all keys
    r.flushall()

    # Check that there are no keys
    return len(r.keys()) == 0
```

(continues on next page)

(continued from previous page)

```
# print(cleanup(r))
```

Let's go through the code, step by step, and it will hopefully make some sense.

Listing 20: Instantiation as usual:

```
1  import timeit
2
3  import redgrease
4
5  # Bind / register the function on some Redis instance.
6  r = redgrease.RedisGears()
7
8
```

The instantiation of the client / connection is business as usual.

2.5.1 Cache-Get function

Lets now go for the core of the solution; The code that we want to run on Redis for each resource request.

Listing 21: Cache handling function:

```
14     import requests
15
16     # Check if the url is already in the cache,
17     # And if so, simply return the cached result.
18     if redgrease.cmd.exists(url):
19         return bytes(redgrease.cmd.get(url))
20
21     # Otherwise fetch the url.
22     response = requests.get(url)
23
24     # Return nothing if request fails
25     if response.status_code != 200:
26         return bytes()
27
28     # If ok, set the cache data and return.
29     response_data = bytes(response.content)
30     redgrease.cmd.set(url, response_data)
31
32     return response_data
33
34
```

Look at the highlighted lines, and notice:

- The logic of handling requests with caching is simply put in a normal function, much like we would if the caching logic was handled by each client.
- The argument of the function is what we could expect, the `url` to the resource to get.
- **The function return value is either:**
  - The contents of the response to requests to the URL (line 32), or

  - A cached value (line 19)

Which is exactly what you would expect from a cached fetching function.

The really interesting part, however, is this little line, on top of the function.

Listing 22: CommandReader function decorator:

```
10 # The `command` decorator tunrs the function to a CommandReader,
11 # registerered on the Redis Gears sever if using the `on` argument
12 @redgrease.command(on=r, requirements=["requests"], replace=False)
13 def cache_get(url):
14     import requests
15
16     # Check if the url is already in the cache,
17     # And if so, simply return the cached result.
18     if redgrease.cmd.exists(url):
19         return bytes(redgrease.cmd.get(url))
```

All the Redis Gears magic is hidden in this function decorator, and it does a couple of important things:

- It embeds the function in a *CommandReader* Gear function.
- It ensures that the function is registered on our Redis server(s).
- It captures the relevant requirements, for the function to work.
- It ensures that we only register this function once.
- It creates a new function, with the same name, which when called, triggers the corresponding registered Gear function, and returns the result from the server.

This means that you can now call the decorated function, just as if it was a local function:

```
result = cache_get("http://images.cocodataset.org/train2017/000000169188.jpg")
```

This may look like it is actually executing the function locally, but the `cache_get` function is actually executed on the server.

This means that the registered `cache_get` Gear function can not only be triggered by the client that defined the decorated function, but **can be triggered by any client** by invoking the Redis Gear RG.TRIGGER command with the the functions' trigger name and arguments.

In our case, using *redis-cli* as an example:

```
> RG.TRIGGER cache_get http://images.cocodataset.org/train2017/000000169188.jpg
```

The arguments for the *@command* decorator, are the same as to the `OpenGearFunction.register()` method, inherited by the `CommandReader` class.

Note: This simplistic cache function is only for demonstrating the command function decorator. The design choices of this particular caching implementation is far from ideal for all use-cases.

For example:

- Only the response content data is returned, not response status or headers.
- Cache is never expiring.
- If multiple requests for the same resource is made in close successions, there may be duplicate external requests.
- The entire response contents is copied into memory before writing to cache.

- ... etc ...

Naturally, the solution could easily be modified to accommodate other behaviors.

2.5.2 Testing the Cache

To test the caching, we create a very simple function that iterates through some URLs and tries to get them from the cache and saving the contents to local files.

Listing 23: Test function:

```
some_image_urls = [
    "http://images.cocodataset.org/train2017/000000483381.jpg",
    "http://images.cocodataset.org/train2017/000000237137.jpg",
    "http://images.cocodataset.org/train2017/000000017267.jpg",
    "http://images.cocodataset.org/train2017/000000197756.jpg",
    "http://images.cocodataset.org/train2017/000000193332.jpg",
    "http://images.cocodataset.org/train2017/000000475564.jpg",
    "http://images.cocodataset.org/train2017/000000247368.jpg",
    "http://images.cocodataset.org/train2017/000000416337.jpg",
]


# Get all the images and write them to disk
def get_em_all():

    for image_url in some_image_urls:

        # This will invoke the cache_get function **on the Redis server**
        image_data = cache_get(image_url)

        # Quick and dirty way of getting the image file name.
        image_name = image_url.split("/")[-1]

        # Write to file
        with open(image_name, "wb") as img_file:
            img_file.write(image_data.value)


# Test it
# Time how long it takes to get images when the cache is empty.
t1 = timeit.timeit(get_em_all, number=1)
print(f"Cache-miss time: {t1:.3f} seconds")

# Time how long it takes to get the images when they are all in the cache.
t2 = timeit.timeit(get_em_all, number=1)
print(f"Cache-hit time: {t2:.3f} seconds")
print(f"That is {t1/t2:.1f} times faster!")


```

Calling the this function twice reveals that the caching does indeed seem to work.

```
Cache-miss time: 10.954 seconds
Cache-hit time: 0.013 seconds
That is 818.6 times faster!
```

We can also inspect the logs of the Redis node to confirm that the cache function was indeed executed on the server.

```
docker logs --tail 100
```

And you should indeed see that the expected log messages appear:

```
1:M 06 Apr 2021 08:58:06.314 * <module> GEARS: Cache request #1 for resource 'http://
↪images.cocodataset.org/train2017/000000416337.jpg'
1:M 06 Apr 2021 08:58:06.314 * <module> GEARS: Cache miss #1 - Downloading resource
↪'http://images.cocodataset.org/train2017/000000416337.jpg'.
1:M 06 Apr 2021 08:58:07.855 * <module> GEARS: Cache update #1 - Request status for␣
↪resource 'http://images.cocodataset.org/train2017/000000416337.jpg': 200

...

1:M 06 Apr 2021 08:58:07.860 * <module> GEARS: Cache request #2 for resource 'http://
↪images.cocodataset.org/train2017/000000416337.jpg'
```

The last piece of code is jut to clean up the database by un-registering the `cache_get` Gear function, cancel and drop any ongoing Gear function executions and flush the key-space.

Listing 24: Clean up the database:

```
def cleanup(r: redgrease.RedisGears):

    # Unregister all registrations
    for reg in r.gears.dumpregistrations():
        r.gears.unregister(reg.id)

    # Remove all executions
    for exe in r.gears.dumpexecutions():
        r.gears.dropexecution(str(exe.executionId))

    # Clear all keys
    r.flushall()

    # Check that there are no keys
    return len(r.keys()) == 0


# print(cleanup(r))
```

That wraps up the Quickstart Guide! Good luck building Gears!

Courtesy of :  Pte. Ltd.

CHAPTER

THREE

REDGREASE CLIENT

The RedGrease client / connection object gives access to the various commands defined by the RedisGears module.

3.1 Instantiation

The RedGrease client can be instantiated in a few different ways.

3.1.1 As a Redis Client

This instantiation method is useful if you are working with Redis from scratch and are not working on a legacy application that already has a bunch of Redis objects instantiated.

```
import redgrease

r = redgrease.RedisGears(host="localhost", port=6379)
```

This instantiates a Redis client object that is a subclass of the `Redis` class of the popular redis python client, but with an additional property `gears` through which the *RedisGears commands can be accessed*.

The constructor takes all the same arguments as the normal Redis client, and naturally it exposes all the same Redis command methods as the original.

It also means that if you are running a Redis instance on the default Redis port (6379) locally, as above, then you don't need any arguments to instantiate the client object: `r = redgrease.RedisGears()`.

Note: RedGrease also supports cluster-mode. Instantiating `RedisGears` will automatically try to figure out if it is against a cluster, and in that case instead instantiate an object which is a subclass of `RedisCluster` from the redis-py-cluster package (which in turn also is a subclass of `Redis` from the redis package).

The constructor takes all the same arguments as the normal Redis client, and naturally it exposes all the same Redis command methods as well.

If you want to be explicit which class to instantiate, then you can use `redgrease.Redis` and `redgrease.RedisCluster` for single and cluster mode respectively.

3.1.2 As a Gears object

If you already have code with instantiated Redis client objects, and you don't want to create more connections, then you can instantiate only the `redgrease.Gears` object directly, using your existing Redis connection.

```
import redis  # Alt. `rediscluster`
import redgrease

# Legacy code
r = redis.Redis()  # Alt. `rediscluster.RedisCluster()`
...

# New code
gears = redgrease.Gears(r)
```

This instantiates a Gears object, which only exposes the *RedisGears commands can be accessed*, and not the normal Redis commands. This object is the same object that the above RedisGears client exposes through its `gears` property.

3.2 RedisGears Commands

The commands introduced by the RedisGears Redis module can be invoked through the `Gears` object, instantiates as per above. This section gives a run-down of the various commands and what they do, in the order of usefulness to most people.

You can find all the methods and functions in the API Reference, and in this section, we specifically look at the `redgrease.Gears` class.

Command Descriptions:

- *Executing Gear Functions*
- *Get List of Executions*
- *Get Result of Asynchronous Execution*
- *Get List of Registered Event-Based Functions*
- *Un-register an Event-Based Function*
- *Trigger a 'Command-Event'*
- *Abort a Running Execution*
- *Remove an Execution*
- *Get List of Registered Dependencies*
- *Get Python Runtime Statistics*
- *Get Cluster Information*
- *Refresh Cluster Topology*
- *Get a Detailed Execution Plan*
- *Get and Set Gears Configurations*

3.2.1 Executing Gear Functions

redgrease.Gears.pyexecute()

This is the most important RedGrease command, as it is the command for executing Gear functions. There are other ways, as we will go through in the next section *Executing Gear Functions*, but under the hood they all call this method.

RedisGears Command: RG.PYEXECUTE

`Gears.`**`pyexecute`**(*gear_function: Union[str, redgrease.runtime.GearsBuilder, redgrease.gears.GearFunction] = '', unblocking=False, requirements: Optional[Iterable[Union[str, packaging.requirements.Requirement]]] = None, enforce_redgrease: Optional[Union[bool, str, packaging.version.Version, packaging.requirements.Requirement]] = None*) → redgrease.data.ExecutionResult

Execute a gear function.

Parameters

- **`gear_function`** (*`Union[str, redgrease.gears.GearFunction]`, `optional`*) – Function to execute. Either:
  - A *raw function string* containing a clear-text serialized Gears Python function as per the examples in the official documentation.
  - A *script file path*.
  - A *GearFunction object*, e.g. *`GearsBuilder`* or either of the *Readers* types.

  Note:

  - Python version must match the Gear runtime.
  - If the function is not "closed" with a *`run`* or *`register`* operation, an `run()` operation without additional arguments will be assumed, and automatically added to the function to close it.
  - The default for `enforce_redgrease` is `True`.

  Defaults to `""`, i.e. no function.

- **`unblocking`** (*`bool, optional`*) – Execute function without waiting for it to finish before returning.

  Defaults to `False`. I.e. block until the function returns or fails.

- **`requirements`** (*`Iterable[Union[None, str, redgrease.requirements.Requirement]], optional`*) – List of 3rd party package requirements needed to execute the function on the server.

  Defaults to `None`.

- **`enforce_redgrease`** (*`redgrease.requirements.PackageOption`, `optional`*) – Indicates if redgrease runtime package requirement should be added or not, and potentially which version and/or extras or source.

  It can take several optional types:

  - `None` : No enforcement. Requirements are passed through, with or without 'redgrease' runtime package.
  - `True` : Enforces latest `"redgrease[runtime]"` package on PyPi,

- `False` : Enforces that *redgrease* is **not** in the requirements list, any *redgrease* requirements will be removed from the function's requirements. Note that it will **not** force *redgrease* to be uninstalled from the server runtime.
- Otherwise, the argument's string-representation is evaluated, and interpreted as either:
  a. A specific version. E.g. `"1.2.3"`.
  b. A version qualifier. E.g. `">=1.0.0"`.
  c. Extras. E.g. `"all"` or `"runtime"`. Will enforce the latest version on PyPi, with this/these extras.
  d. Full requirement qualifier or source. E.g: `"redgrease[all]>=1.2.3"` or `"redgrease[runtime]@git+https://github.com/lyngon/redgrease.git@main"`

Defaults to `None` when `gear_function` is a *script file path* or a *raw function string*, but `True` when it is a *GearFunction object*.

Returns

The returned `ExecutionResult` has two properties: `value` and `errors`, containing the result value and any potential errors, respectively.

The value contains the result of the function, unless:

- When used in 'unblocking' mode, the value is set to the execution ID
- If the function has no output (i.e. it is closed by *register()* or for some other reason is not closed by a *run()* action), the value is True (boolean).

Any errors generated by the function are accumulated in the `errors` property, as a list.

Note: The `ExecutionResult` object itself behaves *for the most part* like its contained `value`, so for many simple operations, such as checking *True-ness*, result length, iterate results etc, it can be used directly. But the the safest was to get the actual result is by means of the `value` property.

Return type redgrease.data.ExecutionResult

Raises **`redis.exceptions.ResponseError`** – If the function cannot be parsed.

3.2.2 Get List of Executions

`redgrease.Gears.dumpexecutions()`

RedisGears Command: RG.DUMPEXECUTIONS

`Gears.`**`dumpexecutions`**(*status: Optional[Union[str, redgrease.data.ExecutionStatus]] = None*, *registered: Optional[bool] = None*) → List[redgrease.data.ExecutionInfo]

Get list of function executions. The executions list's length is capped by the 'MaxExecutions' configuration option.

Parameters

- **`status`** (*`Union[str, redgrease.data.ExecutionStatus], optional`*) – Only return executions that match this status. Either: "created", "running", "done", "aborted", "pending_cluster", "pending_run", "pending_receive" or "pending_termination". Defaults to None.

- **registered** (*bool, optional*) – If *True*, only return registered executions. If *False*, only return non-registered executions.

  Defaults to None.

Returns A list of ExecutionInfo, with an entry per execution.

Return type List[redgrease.data.ExecutionInfo]

3.2.3 Get Result of Asynchronous Execution

redgrease.Gears.getresults()

RedisGears Command: RG.GETRESULTS

Gears.**getresults**(*id: Union[bytes, str, redgrease.data.ExecID, redgrease.data.ExecutionInfo]*) → redgrease.data.ExecutionResult

Get the results of a function in the execution list.

Parameters **id** (*Union[redgrease.data.ExecutionInfo, redgrease.data.ExecID, bytes, str]*) – Execution identifier for the function to fetch the output for.

Returns Results and errors from the gears function, if, and only if, execution exists and is completed.

Return type redgrease.data.ExecutionResult

Raises **redis.exceptions.ResponseError** – If the the execution does not exist or is still running

3.2.4 Get List of Registered Event-Based Functions

redgrease.Gears.dumpregistrations()

RedisGears Command: RG.DUMPREGISTRATIONS

Gears.**dumpregistrations**(*reader: Optional[str] = None*, *desc: Optional[str] = None*, *mode: Optional[str] = None*, *key: Optional[str] = None*, *stream: Optional[str] = None*, *trigger: Optional[str] = None*) → List[redgrease.data.Registration]

Get list of function registrations.

Parameters

- **reader** (*str, optional*) – Only return registrations of this reader type. E.g: "StreamReader" Defaults to None.
- **desc** (*str, optional*) – Only return registrations, where the description match this pattern. E.g: "transaction*log*" Defaults to None.
- **mode** (*str, optional*) – Only return registrations, in this mode. Either "async", "async_local" or "sync". Defaults to None.
- **key** (*str, optional*) – Only return (KeysReader) registrations, where the key pattern match this key. Defaults to None.
- **stream** (*str, optional*) – Only return (StreamReader) registrations, where the stream pattern match this key. Defaults to None.
- **trigger** (*str, optional*) – Only return (CommandReader) registrations, where the trigger pattern match this key. Defaults to None.

Returns A list of Registration, with one entry per registered function.

Return type List[redgrease.data.Registration]

3.2.5 Un-register an Event-Based Function

redgrease.Gears.unregister()

RedisGears Command: RG.UNREGISTER

Gears.**unregister**(*id: Union[bytes, str, redgrease.data.ExecID, redgrease.data.Registration]*) → bool
Removes the registration of a function

Parameters **id** (*Union[redgrease.data.Registration, redgrease.data.ExecID, bytes, str]*) – Execution identifier for the function to unregister.

Returns True if successful

Return type bool

Raises **redis.exceptions.ResponseError** – If the registration ID doesn't exist or if the function's reader doesn't support the unregister operation.

3.2.6 Trigger a 'Command-Event'

redgrease.Gears.trigger()

RedisGears Command: RG.TRIGGER

Gears.**trigger**(*trigger_name: str*, **args*) → List[Any]
Trigger the execution of a registered 'CommandReader' function.

Parameters

- **trigger_name** (*str*) – The registered 'trigger' name of the function
- ***args** (*Any*) – Any additional arguments to the trigger

Returns:6 List: A list of the functions output records.

3.2.7 Abort a Running Execution

redgrease.Gears.abortexecution()

RedisGears Command: RG.ABORTEXECUTION

Gears.**abortexecution**(*id: Union[bytes, str, redgrease.data.ExecID, redgrease.data.ExecutionInfo]*) → bool
Abort the execution of a function mid-flight

Parameters **id** (*Union[redgrease.data.ExecutionInfo, redgrease.data.ExecID, bytes, str]*) – The execution id to abort

Returns True or an error if the execution does not exist or had already finished.

Return type bool

3.2.8 Remove an Execution

redgrease.Gears.dropexecution()

RedisGears Command: RG.DROPEXECUTION

Gears.**dropexecution**(*id: Union[bytes, str, redgrease.data.ExecID, redgrease.data.ExecutionInfo]*) → bool

Remove the execution of a function from the executions list.

Parameters **id** (*Union[redgrease.data.ExecutionInfo, redgrease.data.ExecID, bytes, str]*) – Execution ID to remove

Returns True if successful, or an error if the execution does not exist or is still running.

Return type bool

3.2.9 Get List of Registered Dependencies

redgrease.Gears.pydumpreqs()

RedisGears Command: RG.PYDUMPREQS

Gears.**pydumpreqs**(*name: Optional[str] = None*, *is_downloaded: Optional[bool] = None*, *is_installed: Optional[bool] = None*) → List[redgrease.data.PyRequirementInfo]

Gets all the python requirements available (with information about each requirement).

Parameters

- **name** (*str, optional*) – Only return packages with this **base name**. I.e. it is not filtering on version number, extras etc. Defaults to None.
- **is_downloaded** (*bool, optional*) – If *True*, only return requirements that have been downloaded. If *False*, only return requirements that have NOT been downloaded. Defaults to None.
- **is_installed** (*bool, optional*) – If *True*, only return requirements that have been installed. If *False*, only return requirements that have NOT been installed. Defaults to None.

Returns List of Python requirement information objects.

Return type List[redgrease.data.PyRequirementInfo]

3.2.10 Get Python Runtime Statistics

redgrease.Gears.pystats()

RedisGears Command: RG.PYSTATS

Gears.**pystats**() → redgrease.data.PyStats

Gets memory usage statisticy from the Python interpreter

Returns Python interpretere memory statistics, including total, peak and current amount of allocated memory, in bytes.

Return type redgrease.data.PyStats

3.2.11 Get Cluster Information

redgrease.Gears.infocluster()

RedisGears Command: RG.INFOCLUSTER

Gears.**infocluster**() → redgrease.data.ClusterInfo
Gets information about the cluster and its shards.

Returns Cluster information or None if not in cluster mode.

Return type redgrease.data.ClusterInfo

3.2.12 Refresh Cluster Topology

redgrease.Gears.refreshcluster()

RedisGears Command: RG.REFRESHCLUSTER

Gears.**refreshcluster**() → bool
Refreshes the local node's view of the cluster topology.

Returns True if successful.

Return type bool

Raises **redis.exceptions.ResponseError** – If not successful

3.2.13 Get a Detailed Execution Plan

redgrease.Gears.getexecution()

RedisGears Command: RG.GETEXECUTION

Gears.**getexecution**(*id: Union[bytes, str, redgrease.data.ExecID, redgrease.data.ExecutionInfo]*, *locality: Optional[redgrease.data.ExecLocality] = None*) → Mapping[bytes, redgrease.data.ExecutionPlan]
Get the execution plan details for a function in the execution list.

Parameters

- **id** (*Union[redgrease.data.ExecutionInfo, redgrease.data.ExecID, bytes, str]*) – Execution identifier for the function to fetch execution plan for.
- **locality** (*Optional[redgrease.data.ExecLocality], optional*) – Set to 'Shard' to get only local execution plan and set to 'Cluster' to collect executions from all shards. Defaults to 'Shard' in stand-alone mode, but "Cluster" in cluster mode.

Returns A dict, mapping cluster ID to ExecutionPlan

Return type Mapping[bytes, redgrease.data.ExecutionPlan]

3.3 Get and Set Gears Configurations

The above mentioned *Gears* object contains a property `config` of type *redgrease.config.Config*, through which various pre-defined and custom configuration setting can be read an written.

This object contains readable, and for certain options also writable, properties for the predefined RedisGears configurations. It also contains *getter* and *setter* methods that allows access to both the pre-defined as well as user defined configurations, in bulk.

RedisGears Commands: RG.CONFIGGET and RG.CONFIGSET

class redgrease.config.**Config**(*redis: redis.client.Redis*)

Redis Gears Config

Instantiate a Redis Gears Confg object

Parameters **redis** (*redis.Redis*) – Redis client / connection for underlying communication.

ValueTypes: Dict[str, Union[Type[T], Callable[[Any], T]]] = {'CreateVenv': <class 'b

Mapping from config name to its corresponding value type, transformer or constructor.

redis

get(**config_option: Union[bytes, str]*) → Dict[Union[bytes, str], Any]

Get the value of one or more built-in configuration or a user-defined options.

Parameters ***config_option** (*Union[bytes, str]*) – One or more names/key of configurations to get.

Returns Dict of the requested config options mapped to their corresponding values.

Return type Dict[Union[bytes, str], Any]

set(*config_dict=None*, ***config_setting*) → bool

Set a value of one ore more built-in configuration or a user-defined options.

This function offers two methods of providing the keys and values to set; either as a mapping/dict or b key-word arguments.

Parameters

- **config_dict** (*Mapping[str, Any]*) – Mapping / dict of config values to set.
- ****config_setting** (*Any*) – Key-word arguments to set as config values.

Returns *True* if all was successful, *False* oterwise

Return type bool

get_single(*config_option: Union[bytes, str]*)

Get a single config value.

Parameters **config_option** (*str*) – Name of the config to get.

Returns The value of the config option.

Return type Any

property MaxExecutions

Get the current value for the *MaxExecutions* config option.

The MaxExecutions configuration option controls the maximum number of executions that will be saved in the executions list. Once this threshold value is reached, older executions will be deleted from the list by order of their creation (FIFO).

Only executions that had finished (e.g. the 'done' or 'aborted' status) are deleted.

property MaxExecutionsPerRegistration
Get the current value for the *MaxExecutionsPerRegistration* config option.

The MaxExecutionsPerRegistration configuration option controls the maximum number of executions that are saved in the list per registration. Once this threshold value is reached, older executions for that registration will be deleted from the list by order of their creation (FIFO).

Only executions that had finished (e.g. the 'done' or 'aborted' status) are deleted.

property ProfileExecutions
Get the current value for the *ProfileExecutions* config option.

The ProfileExecutions configuration option controls whether executions are profiled.

Note: Profiling impacts performance Profiling requires reading the server's clock, which is a costly operation in terms of performance. Execution profiling is recommended only for debugging purposes and should be disabled in production.

property PythonAttemptTraceback
Get the current value for the *PythonAttemptTraceback* config option.

The PythonAttemptTraceback configuration option controls whether the engine tries producing stack traces for Python runtime errors.

property DownloadDeps
Get the current value for the *DownloadDeps* config option.

The DownloadDeps configuration option controls whether or not RedisGears will attempt to download missing Python dependencies.

property DependenciesUrl
Get the current value for the *DependenciesUrl* config option.

The DependenciesUrl configuration option controls the location from which RedisGears tries to download its Python dependencies.

property DependenciesSha256
Get the current value for the *DependenciesSha256* config option.

The DependenciesSha256 configuration option specifies the SHA265 hash value of the Python dependencies. This value is verified after the dependencies have been downloaded and will stop the server's startup in case of a mismatch.

property PythonInstallationDir
Get the current value for the *PythonInstallationDir* config option.

The PythonInstallationDir configuration option specifies the path for RedisGears' Python dependencies.

property CreateVenv
Get the current value for the *CreateVenv* config option.

The CreateVenv configuration option controls whether the engine will create a virtual Python environment.

property ExecutionThreads
Get the current value for the *ExecutionThreads* config option.

The ExecutionThreads configuration option controls the number of threads that will run executions.

property ExecutionMaxIdleTime
Get the current value for the *ExecutionMaxIdleTime* config option.

The ExecutionMaxIdleTime configuration option controls the maximal amount of idle time (in milliseconds) before execution is aborted. Idle time means no progress is made by the execution.

The main reason for idle time is an execution that's blocked on waiting for records from another shard that had failed (i.e. crashed). In that case, the execution will be aborted after the specified time limit. The idle timer is reset once the execution starts progressing again.

property PythonInstallReqMaxIdleTime

Get the current value for the *PythonInstallReqMaxIdleTime* config option.

The PythonInstallReqMaxIdleTime configuration option controls the maximal amount of idle time (in milliseconds) before Python's requirements installation is aborted. Idle time means that the installation makes no progress.

The main reason for idle time is the same as for ExecutionMaxIdleTime .

property SendMsgRetries

Get the current value for the *SendMsgRetries* config option. The SendMsgRetries configuration option controls the maximum number of retries for sending a message between RedisGears' shards. When a message is sent and the shard disconnects before acknowledging it, or when it returns an error, the message will be resent until this threshold is met.

Setting the value to 0 means unlimited retries.

CHAPTER

FOUR

EXECUTING GEAR FUNCTIONS

Gear functions can either be defined as a *Raw Function String*, in *Script File Path*, as or dynamically constructed *GearFunction Object*. There are some subtleties and variations the three types that we'll go through in their respective section, but either type can be executed using the *redgrease.Gears.pyexecute()* method.

```
import redgrease

gear_fun = ...  # Either a function string, script file path or GearFunction object

connection = redgrease.RedisGears()

result = connection.gears.pyexecute(gear_fun)

print(result.value)
print(result.errors)
```

4.1 Raw Function String

The most basic way of creating and executing Gear Functions is by passing a raw function string to the *redgrease.Gears.pyexecute()* method:

```
import redgrease

raw_gear_fun = "GearsBuilder('KeysReader').map(lambda x: x['type']).countby().run()"

rg = redgrease.RedisGears()

result = rg.gears.pyexecute(raw_gear_fun)
```

Note: You would rarely construct Gear functions this way, but it is fundamentally what happens under the hood for all the other methods of execution, and corresponds directly to the underlying RedisGears protocol.

4.2 Script File Path

A more practical way of defining Gear Functions is by putting them in a separate script file, and executing it by passing the path to the *redgrease.Gears.pyexecute()* method:

```
import redgrease

gear_script_path = "./path/to/some/gear/script.py"

rg = redgrease.RedisGears()

result = rg.gears.pyexecute(gear_script_path)
```

These scripts may be plain vanilla RedisGears functions that only use the *built-in runtime functions*, and does not import *redgrease* or use any of its features. In this case the *redgrease* package does not need to be installed on the runtime.

If the function is importing and using any RedGrease construct from the `redgrease` package, then when calling *redgrease.Gears.pyexecute()* method, the `enforce_redgrease` must be set in order to ensure that the package is installed on the RedisGears runtime.

In most cases you would just set it to `True` to get the latest stable RedGrease runtime package, but you may specify a specific version or even repository.

A notable special case is when functions in the script are only importing RedGrease modules that do not require any 3rd party dependencies (see list in the *Redgrease Extras Options* section). If this is the case then you may want to set `enforce_redgrease="redgrease"` (without the extras *"[runtime]"*), when calling *redgrease.Gears.pyexecute()*, as this is a version of redgrease without any external dependencies.

Another case is when you are only using explicitly imported *Builtin Runtime Functions* (e.g. `from redgrease.runtime import GB, logs, execute`) , and nothing else, as you in this case do not need any version of RedGrease on your RedisGears server runtime. In this case you can actually set `enforce_redgrease=False`.

More details about the various runtime installation options, which modules and functions are impacted, as well as the respective 3rd party dependencies can be found in the *Redgrease Extras Options* section.

Note: By default all Gear functions run in a shared runtime environment, and as a consequence all requirements / dependencies from different Gear functions are all installed in the same Python environment.

4.3 GearFunction Object

You can dynamically create a GearFunction object, directly in the same application as your Gears client / connection, by using any of the constructs well talk about in the *GearFunction* section, such as for example *GearsBuilder* or the "Reader" classes such as *KeysReader* and *StreamReader* etc.

GearFunction objects can be executed in three different ways; Using *pyexecute*, the *on-method* or directly in *run or execute*.

Note: There are two drawbacks of executing GearFunction objects, compared to executing Gear functions using either raw string, or by script file:

1. The Python version of local application must match the the Python version in the RedisGears runtime (Python 3.7, at the time of writing this).

When executing Gear functions using either raw string, or by script file, it doesn't matter which version of Python the application is using, as long as it is Python 3.6 or later and the the code in the raw string is compatible with the Python version in the RedisGears runtime,

2. The `redgrease[runtime]` package must be installed on the RedisGears Runtime environment.

   If you pass a GearFunction to *`redgrease.Gears.pyexecute()`*, it will attempt to install the latest stable version of `redgrease[runtime]` on the server, unless already installed, or if explicitly told otherwise using the `enforce_redgrease` argument.

   When executing Gear functions using either raw string, or by script file, *redgrease* only have to be installed if the function is importing any redgrease modules, of course.

As an example let's assume, that we have instantiated a Gear client and created a very simple "open" Gear function as follows:

```
# Function that collecs the keys per type
get_keys_by_type = redgrease.KeysReader().aggregateby(
    lambda record: record["type"],
    set(),
    lambda _, acc, rec: acc | set([rec["key"]]),
    lambda _, acc, lacc: acc | lacc,
)
```

In this example `get_keys_by_type` is our simple "open" Gear function, which groups all the keys by their type ("string", "hash", "stream" etc) , and within each group collects the keys in a set.

We call it "open" because it has not been "closed" by a *`run()`* or *`register()`* action. The output from the last operation, here *`countby()`*, can therefore be used as input for a subsequent operations, if we'd like. The chain of operations of the function is "open-ended", if you will.

Once an "open" function is terminated with either a *`run()`* or *`register()`* action, it is considered "closed", and it can be executed, but not further extended.

The *GearFunction* section, goes into more details of these concepts and the different classes of GearFunctions.

RedGrease allows for open Gear functions, such as `key_counter` to be used as a starting point for other Gear functions, so lets create two "closed" functions from it:

```
# Transform to a single dict from type to keys
get_keys_by_type_dict = (
    get_keys_by_type.map(lambda record: {record["key"]: record["value"]})
    .aggregate({}, lambda acc, rec: {**acc, **rec})
    .run()
)

# Find the most common key-type
get_commonest_type = (
    get_keys_by_type.map(lambda x: (x["key"], len(x["value"])))
    .aggregate((None, 0), lambda acc, rec: rec if rec[1] > acc[1] else acc)
    .run()
)
```

These two new functions, `get_keys_by_type_dict` and `get_commonest_type`, both extend the earlier `get_keys_by_type` function with more operations. The former function collates the results in a dictionary. The latter finds the key-type that is most common in the keyspace.

Note that both functions end with the *`run()`* action, which indicates that the functions will run as an on-demand batch-job, but also that it is 'closed' and cannot be extended further.

Let's execute these functions in some different ways.

4.3.1 Execute with `Gears.pyexecute()` Client method

The most idiomatic way of executing GearFunction objects is just to pass it to *`Gears.pyexecute()`*:

```
# Execute "closed" GearFunction object with a Gear clients' `pyexecute` method
r = redgrease.RedisGears()

result_1 = r.gears.pyexecute(get_keys_by_type_dict)
```

The result might look something like:

```
{
    'hash': {'hash:1', 'hash:0', ...},
    'string': {'string:0', 'string:1', 'string:2', ...}
    ...
}
```

If you pass an "open" gear function, like our initial `get_keys_by_type`, to *`Gears.pyexecute()`*, it will still try its best to execute it, by assuming that you meant to close it with an empty *`run()`* action in the end:

```
# Execute "open" GearFunction with a Gear clients' `pyexecute` method
result_3 = r.gears.pyexecute(get_keys_by_type)
```

The result from our function might look something like:

```
[
    {'key': 'hash', 'value': {'hash:1', 'hash:0', ...}},
    {'key': 'string', 'value': {'string:0', 'string:1', 'string:2', ...}},
    ...
]
```

4.3.2 Execute with `ClosedGearFunction.on()` GearFunction method

Another short-form way of running a closed GearFunction is to call its its *`on()`* method.

```
# Execute "closed" GearFunction object with its `on` method
result_2 = get_commonest_type.on(r)
```

This approach only works with "closed" functions, but works regardless if the function has been closed with the *`run()`* or *`register()`* action.

The result for our specific function might look something like:

```
("string", 3)
```

The API specification is as follows:

`ClosedGearFunction.`**`on`**(*gears_server*, *unblocking: bool = False*, *requirements: Optional[Iterable[str]] = None*, *replace: Optional[bool] = None*, ***kwargs*)

Execute the function on a RedisGears. This is equivalent to passing the function to *Gears.pyexecute*

Parameters

- **`gears_server`** (*[type]*) – Redis client / connection object.

- **unblocking** (*bool, optional*) – Execute function unblocking, i.e. asynchronous. Defaults to `False`.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to `None`.

Returns The result of the function, just as *Gears.pyexecute*

Return type redgrease.data.ExecutionResult

4.3.3 Execute directly in `run()` or `register()`

An even more succinct way of executing GearFunction objects is to specify the target connection directly in the action that closes the function. I.e the *`run()`* or *`register()`* action.

RedGrease has extended these methods with a couple additional arguments, which are not in the standard RedisGears API:

- `requirements` - Takes a list of requirements / external packages that the function needs installed.
- `on` - Takes a Gears (or RedisGears) client and immediately executes the function on it.

```
# Execute GearFunction using `on` argument in "closing" method
result_4 = get_keys_by_type.run(on=r)
```

This approach only works with "closed" functions, but works regardless if the function has been closed with the *`run()`* or *`register()`* action.

The result for our specific function should be identical to when we ran the function using *pyexecute*:

```
[
    {'key': 'hash', 'value': {'hash:1', 'hash:0', ...}},
    {'key': 'string', 'value': {'string:0', 'string:1', 'string:2', ...}},
    ...
]
```

4.4 `pyexecute` API Reference

`Gears.`**`pyexecute`**(*gear_function: Union[str, redgrease.runtime.GearsBuilder, redgrease.gears.GearFunction] = '', unblocking=False, requirements: Optional[Iterable[Union[str, packaging.requirements.Requirement]]] = None, enforce_redgrease: Optional[Union[bool, str, packaging.version.Version, packaging.requirements.Requirement]] = None*) → redgrease.data.ExecutionResult

Execute a gear function.

Parameters

- **gear_function** (*Union[str, redgrease.gears.GearFunction], optional*) – Function to execute. Either:
  - A *raw function string* containing a clear-text serialized Gears Python function as per the examples in the official documentation.
  - A *script file path*.
  - A *GearFunction object*, e.g. *`GearsBuilder`* or either of the *Readers* types.

Note:

  - Python version must match the Gear runtime.
  - If the function is not "closed" with a *run* or *register* operation, an `run()` operation without additional arguments will be assumed, and automatically added to the function to close it.
  - The default for `enforce_redgrease` is `True`.

  Defaults to `""`, i.e. no function.

- **unblocking** (*bool, optional*) – Execute function without waiting for it to finish before returning.

  Defaults to `False`. I.e. block until the function returns or fails.

- **requirements** (*Iterable[Union[None, str, redgrease.requirements.Requirement]], optional*) – List of 3rd party package requirements needed to execute the function on the server.

  Defaults to `None`.

- **enforce_redgrease** (*redgrease.requirements.PackageOption, optional*) – Indicates if redgrease runtime package requirement should be added or not, and potentially which version and/or extras or source.

  It can take several optional types:

  - `None` : No enforcement. Requirements are passed through, with or without 'redgrease' runtime package.
  - `True` : Enforces latest `"redgrease[runtime]"` package on PyPi,
  - `False` : Enforces that *redgrease* is **not** in the requirements list, any *redgrease* requirements will be removed from the function's requirements. Note that it will **not** force *redgrease* to be uninstalled from the server runtime.
  - Otherwise, the argument's string-representation is evaluated, and interpreted as either:
    a. A specific version. E.g. `"1.2.3"`.
    b. A version qualifier. E.g. `">=1.0.0"`.
    c. Extras. E.g. `"all"` or `"runtime"`. Will enforce the latest version on PyPi, with this/these extras.
    d. Full requirement qualifier or source. E.g: `"redgrease[all]>=1.2.3"` or `"redgrease[runtime]@git+https://github.com/lyngon/redgrease.git@main"`

  Defaults to `None` when `gear_function` is a *script file path* or a *raw function string*, but `True` when it is a *GearFunction object*.

Returns

The returned `ExecutionResult` has two properties: `value` and `errors`, containing the result value and any potential errors, respectively.

The value contains the result of the function, unless:

- When used in 'unblocking' mode, the value is set to the execution ID
- If the function has no output (i.e. it is closed by *register()* or for some other reason is not closed by a *run()* action), the value is True (boolean).

Any errors generated by the function are accumulated in the `errors` property, as a list.

Note: The `ExecutionResult` object itself behaves *for the most part* like its contained `value`, so for many simple operations, such as checking *True-ness*, result length, iterate results etc, it can be used directly. But the the safest was to get the actual result is by means of the `value` property.

Return type redgrease.data.ExecutionResult

Raises **`redis.exceptions.ResponseError`** – If the function cannot be parsed.

4.5 `on` API Reference

`ClosedGearFunction.`**`on`**(*gears_server*, *unblocking: bool = False*, *requirements: Optional[Iterable[str]] = None*, *replace: Optional[bool] = None*, ***kwargs*)

Execute the function on a RedisGears. This is equivalent to passing the function to *Gears.pyexecute*

Parameters

- **`gears_server`** (*`[type]`*) – Redis client / connection object.
- **`unblocking`** (*`bool, optional`*) – Execute function unblocking, i.e. asynchronous. Defaults to `False`.
- **`requirements`** (*`Iterable[str], optional`*) – Additional requirements / dependency Python packages. Defaults to `None`.

Returns The result of the function, just as *Gears.pyexecute*

Return type redgrease.data.ExecutionResult

CHAPTER

FIVE

BUILTIN RUNTIME FUNCTIONS

The RedisGears Python server runtime automagically expose a number of functions into the scope of any Gear functions being executed. These "builtin" runtime functions can be used in Gear Functions without importing any module or package:

- *execute*
- *atomic*
- *configGet*
- *gearsConfigGet*
- *hashtag*
- *log*
- *GearsBuilder*

Note: With the exception of the *GearsBuilder* neither these functions **cannot** be used in normal application code outside Gear functions running in the RedisGears server runtime.

RedGrease expose its own wrapped versions of these RedisGears runtime functions which, for the most part, behave exactly like the originals, but require you to import them, either from the top level `redgrease` package, or from the `redgrease.runtime` module.

But if these are the same, why would you bother with them?

The main reason to use the RedGrease versions, is that they aid during development by enabling most Integrated Development Environment (IDE) access to the doc-strings and type-annotations that the RedGrease versions are providing.

This alone can help greatly in developing gears faster (e.g. through auto-complete) and with less errors (e.g. with type checking).

Note: If you are **only** using these wrapped runtime functions in your Gear Functions, and **no other** RedGrease features, then you actually don't need RedGrease to be installed on the RedisGears server runtime. Explicitly setting `enforce_redgrease` argument to `False` when executing a function script with to *`Gears.pyexecute()`*, will not add any redgrease requirement to the function and simply ignore any explicit runtime imports.

The section, *Redgrease Extras Options*, goes deeper into the details of the various RedGrease extras options, and their limitations.

Another reason to use the functions from RedGrease `runtime`, is that it contains some slightly enhanced variants of the defaults, like for example, the *log*, or have alternative versions, like *hashtag3*.

And if you are going to use other RedGrease features, then you will have to load the top level `redgrease` namespace anyway, which automatically expose the runtime functions.

The RedGrease runtime functions can be imported in a few ways:

- Directly from the package top-level, e.g:

```
from redgrease import GearsBuilder, log, atomic, execute
```

- Explicitly from the `redgrease.runtime` module:

```
from redgrease.runtime import GearsBuilder, log, atomic, execute
```

- By importing the `redgrease.runtime` module:

```
import redgrease.runtime
```

It is possible to load all symbols, using `*`, although it's generally not a recommended practice, particularly not for the top level `redgrease` package.

5.1 execute

RedGrease's version of *`runtime.execute()`* behaves just like the default.

This function executes an arbitrary Redis command inside Gear functions.

Note: For more information about Redis commands refer to:

- Redis commands

Arguments

- command : the command to execute
- args : the command's arguments

Example:

```
from redgrease import execute

# Pings the server (reply should be 'PONG')
reply = execute('PING')
```

In most cases, a more convenient approach is to use *Serverside Redis Commands* to execute Redis Commands inside Gear Functions.

Longer Example:

```
from redgrease import GearsBuilder, execute

def age(x):
   ''' Extracts the age from a person's record '''
   return int(x['value']['age'])

def cas(x):
   ''' Checks and sets the current maximum '''
   k = 'age:maximum'
```

(continues on next page)

(continued from previous page)

```
   v = execute('GET', k)    # read key's current value
   v = int(v) if v else 0   # initialize to 0 if N
   if x > v:                # if a new maximum found
   execute('SET', k, x)   # set key to new value

# Event handling function registration
gb = GearsBuilder()
gb.map(age)
gb.foreach(cas)
gb.register('person:*')
```

5.1.1 `execute` API Reference

redgrease.runtime.**execute**(*command: str*, **args*) → bytes

Execute an arbitrary Redis command.

Parameters **command** (*str*) – The commant to execute

Returns Raw command response

Return type bytes

5.2 atomic

RedGrease's version of `runtime.atomic()` behaves just like the default.

Atomic provides a context manager that ensures that all operations in it are executed atomically by blocking the main Redis process.

Example:

```
from redgrease import atomic, GB, hashtag

# Increments two keys atomically
def transaction(_):
   with atomic():
      execute('INCR', f'{{{hashtag()}}}:foo')
      execute('INCR', f'{{{hashtag()}}}:bar')

gb = GB('ShardsIDReader')
gb.foreach(transaction)
gb.run()
```

5.2.1 `atomic` API Reference

class redgrease.runtime.**atomic**

The atomic() Python context is imported to the runtime's environment by default.

The context ensures that all operations in it are executed atomically by blocking he main Redis process.

5.3 configGet

RedGrease's version of *runtime.configGet()* behaves just like the default.

This function fetches the current value of a RedisGears configuration options.

Example:

```
from redgrease import configGet

# Gets the current value for 'ProfileExecutions'
foo = configGet('ProfileExecutions')
```

5.3.1 configGet API Reference

redgrease.runtime.**configGet**(*key: str*) → str
Fetches the current value of a RedisGears configuration option.

Parameters **key** (*str*) – The configuration option key

5.4 gearsConfigGet

RedGrease's version of *runtime.gearsConfigGet()* behaves just like the default.

This function fetches the current value of a RedisGears configuration options, and returns a default value if that key does not exist.

Example:

```
from redgrease import gearsConfigGet

# Gets the 'foo' configuration option key and defaults to 'bar'
foo = gearsConfigGet('foo', default='bar')
```

5.4.1 gearsConfigGet API Reference

redgrease.runtime.**gearsConfigGet**(*key: str*, *default=None*) → str
Fetches the current value of a RedisGears configuration option and returns a default value if that key does not exist.

Parameters

- **key** (*str*) – The configuration option key.
- **default** (*[type], optional*) – A default value. Defaults to None.

5.5 hashtag

RedGrease's version of *runtime.hashtag()* behaves just like the default.

This function returns a hashtag that maps to the lowest hash slot served by the local engine's shard. Put differently, it is useful as a hashtag for partitioning in a cluster.

5.5.1 hashtag API Reference

redgrease.runtime.**hashtag**() → str

Returns a hashtag that maps to the lowest hash slot served by the local engine's shard. Put differently, it is useful as a hashtag for partitioning in a cluster.

Returns A hastag that maps to the lowest hash slot served by the local engine.

Return type str

5.6 hashtag3

This function, *runtime.hashtag3()*, is not part of the default RedisGears runtime scope, and is introduced by RedGrease. It is a slightly modified version of version of *runtime.hashtag()* but adds enclosing curly braces ("{" and "}") to the hashtag, so it can be used directly inside Python f-strings.

5.6.1 hashtag3 API Reference

redgrease.runtime.**hashtag3**() → str

Provides a the same value as *hashtag*, but surrounded by curly braces.

For example, if *hashtag()* generates "06S", then `hashtag3' gives "{06S}".

This is useful for creating slot-specific keys using f-strings, inside gear functions, as the braces are already escaped. Example:

```
redgrease.cmd.set(f"{hashtag3()}",  some_value)
```

Returns A braces-enclosed hashtag string

Return type str

5.7 log

RedGrease's version of *runtime.log()* behaves almost like the default. It prints a message to Redis' log, but forces the the argument to a string before logging it.

(The built in default throws an error if the argument is not a string.)

Example:

```
from redgrease import GB, log

# Dumps every datum in the DB to the log for "debug" purposes
GB().foreach(lambda x: log(str(x), level='debug')).run()
```

5.7.1 `log` API Reference

`redgrease.runtime.`**`log`**(*message: str*, *level: str = 'notice'*)

Print a message to Redis' log.

Parameters

- **`message`** (*`str`*) – The message to output
- **`level`** (*`str, optional`*) – Message loglevel. Either:

```
- 'debug'
- 'verbose'
- 'notice'
- 'warning'
```

Defaults to 'notice'.

5.8 GearsBuilder

The *`runtime.GearsBuilder`* (as well as its short-form alias `GB`), behaves exactly like the default RedisGears version, with a couple of exceptions:

1. It has a property `gearfunction` which gives access to the constructed *GearFunction* object at that that point in the builder pipeline.
2. Any additional arguments passed to its constructor, will be passed as defaults to the *`run()`* or *`register()`* action that terminates the build.

Note: The *`runtime.GearsBuilder`* objects are mutable with respect to the operations, whereas *GearFunction* objects are immutable and returns a new function when an operation is applied.

This means that:

```
fun = GearsBuilder()
fun.map(...)
fun.aggregateby(...)
```

Creates one single function equivalent to:

```
fun = KeysReader().map(...).aggregateby(...)
```

Whereas:

```
sad = KeysReader()
sad.map(...)
sad.aggregateby(...)
```

Creates three functions; One named `sad` that is just the *KeysReader*, one which is `sad` with a *Map* and one which is `sad` with a *AggregateBy*. The latter two functions are also not bound to any variables in this example.

5.8.1 `GearsBuilder` API Reference

class redgrease.runtime.**GearsBuilder**(*reader: str = 'KeysReader'*, *defaultArg: str = '*'*, *desc: Optional[str] = None*, **args*, ***kwargs*)

The RedisGears GearsBuilder class is imported to the runtime's environment by default, and this class is a RedGrease wrapper of it.

It exposes the functionality of the function's context builder.

> **Warning:** GearsBuilder is mutable with respect to the operations.
>
> The *GearsBuilder* is a subclass of *gears.OpenGearFunction*, but unlike other OpenGearFunctions, the GearsBuilder mutates an internal GearFunction instead of creating a new one for each operation. This behavior is deliberate, in order to be consistent with the original GearsBuilder.

Gear function / process factory

Parameters

- **reader** (*str, optional*) – Input records reader Defining where the input to the gear will come from. One of:
  - `"KeysReader"`
  - `"KeysOnlyReader"`
  - `"StreamReader"`
  - `"PythonReader"`
  - `"ShardsReader"`
  - `"CommandReader"`

  Defaults to 'KeysReader'.
- **defaultArg** (*str, optional*) – Additional arguments to the reader. These are usually a key's name, prefix, glob-like or a regular expression. Its use depends on the function's reader type and action. Defaults to '*'.
- **desc** (*str, optional*) – An optional description. Defaults to None.

property gearfunction

The "open" GearFunction object at this step in the pipeline.

This GearFunction is itself immutable but can be built upon to create new GearFunctions, independently from the GearsBuilder.

Returns The current GearFunction object.

Return type *redgrease.gears.OpenGearFunction*

run(*arg: Optional[str] = None*, *convertToStr: bool = True*, *collect: bool = True*, *requirements: Optional[Iterable[str]] = None*, *on=None*, ***kwargs*) → *redgrease.gears.ClosedGearFunction*

Create a "closed" function to be *Run* as in "batch-mode".

Batch functions are executed once and exits once the data is exhausted by its reader.

Parameters

- **arg** (*str, optional*) – An optional argument that's passed to the reader as its defaultArg. It means the following:
  - A glob-like pattern for the KeysReader and KeysOnlyReader readers.

- A key name for the StreamReader reader.
- A Python generator for the PythonReader reader.

Defaults to `None`.

- **convertToStr** (*bool, optional*) – When `True`, adds a map operation to the flow's end that stringifies records. Defaults to `False`.
- **collect** (*bool, optional*) – When `True` adds a collect operation to flow's end. Defaults to `False`.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to `None`.
- **on** (*redis.Redis*) – Immediately execute the function on this RedisGears system.
- ****kwargs** – Additional parameters to the run operation.

Returns A new closed batch function, if *on* is **not** specified. An execution result, if *on* **is** specified.

Return type Union[*ClosedGearFunction*, redgrease.data.ExecutionResult]

Raises **TypeError** – If the function does not support batch mode.

register(*prefix: str = '*', convertToStr: bool = True, collect: bool = True, mode: Optional[str] = None, onRegistered: Optional[Callable[], None]] = None, eventTypes: Optional[Iterable[str]] = None, keyTypes: Optional[Iterable[str]] = None, readValue: Optional[bool] = None, batch: Optional[int] = None, duration: Optional[int] = None, onFailedPolicy: Optional[str] = None, onFailedRetryInterval: Optional[int] = None, trimStream: Optional[bool] = None, trigger: Optional[str] = None, requirements: Optional[Iterable[str]] = None, on=None, **kwargs*) → *redgrease.gears.ClosedGearFunction*

Create a "closed" function to be *Register* 'ed as an event-triggered function.

Event functions are executed each time an event arrives. Each time it is executed, the function operates on the event's data and once done is suspended until its future invocations by new events. :param prefix: Key prefix pattern to match on.

Not relevant for 'CommandReader' readers (see 'trigger'). Defaults to `"*"`.

Parameters

- **convertToStr** (*bool, optional*) – When `True` adds a map operation to the flow's end that stringifies records. Defaults to `True`.
- **collect** (*bool, optional*) – When `True` adds a collect operation to flow's end. Defaults to `False`.
- **mode** (*str, optional*) – The execution mode of the function. Can be one of:
  - `"async"`:

    Execution will be asynchronous across the entire cluster.
  - `"async_local"`:

    Execution will be asynchronous and restricted to the handling shard.
  - `"sync"`:

    Execution will be synchronous and local.

  Defaults to `"async"`.

- **onRegistered** (*Registrator, optional*) – A function that's called on each shard upon function registration. It is a good place to initialize non-serializeable objects such as network connections. Defaults to `None`.
- **eventTypes** (*Iterable[str], optional*) – For KeysReader only. A whitelist of event types that trigger execution when the KeysReader are used. The list may contain one or more:
  - Any Redis or module command
  - Any Redis event

  Defaults to `None`.
- **keyTypes** (*Iterable[str], optional*) – For KeysReader and KeysOnlyReader only. A whitelist of key types that trigger execution when using the KeysReader or KeysOnlyReader readers. The list may contain one or more from the following:
  - Redis core types:

    `"string"`, `"hash"`, `"list"`, `"set"`, `"zset"` or `"stream"`
  - Redis module types:

    `"module"`

  Defaults to `None`.
- **readValue** (*bool, optional*) – For KeysReader only. When `False` the value will not be read, so the 'type' and 'value' of the record will be set to `None`. Defaults to `True`.
- **batch** (*int, optional*) – For StreamReader only. The number of new messages that trigger execution. Defaults to 1.
- **duration** (*int, optional*) – For StreamReader only. The time to wait before execution is triggered, regardless of the batch size (0 for no duration). Defaults to 0.
- **onFailedPolicy** (*str, optional*) – For StreamReader only. The policy for handling execution failures. May be one of:
  - `"continue"`:

    Ignores a failure and continues to the next execution. This is the default policy.
  - `"abort"`:

    Stops further executions.
  - `"retry"`:

    Retries the execution after an interval specified with onFailedRetryInterval (default is one second).

  Defaults to `"continue"`.
- **onFailedRetryInterval** (*int, optional*) – For StreamReader only. The interval (in milliseconds) in which to retry in case onFailedPolicy is 'retry'. Defaults to 1.
- **trimStream** (*bool, optional*) – For StreamReader only. When `True` the stream will be trimmed after execution Defaults to `True`.
- **trigger** (*str*) – For 'CommandReader' only, and mandatory. The trigger string that will trigger the function.

- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to None.
- **on** (*redis.Redis*) – Immediately execute the function on this RedisGears system.
- ****kwargs** – Additional parameters to the register operation.

Returns A new closed event function, if *on* is **not** specified. An execution result, if *on* **is** specified.

Return type Union[*ClosedGearFunction*, redgrease.data.ExecutionResult]

Raises **TypeError** – If the function does not support event mode.

map(*op: Callable[[InputRecord], OutputRecord], requirements: Optional[Iterable[str]] = None, **kwargs*) → *redgrease.runtime.GearsBuilder*

Instance-local *Map* operation that performs a one-to-one (1:1) mapping of records.

:param op *redgrease.typing.Mapper*): Function to map on the input records. The function must take one argument as input (input record) and return something as an output (output record).

Parameters

- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to None.
- ****kwargs** – Additional parameters to the Map operation.

Returns Itself, i.e. the same GearsBuilder, but with its internal gear function updated with a Map operation as last step.

Return type *GearsBuilder*

flatmap(*op: Optional[Callable[[InputRecord], Iterable[OutputRecord]]] = None, requirements: Optional[Iterable[str]] = None, **kwargs*) → *redgrease.runtime.GearsBuilder*

Instance-local *FlatMap* operation that performs one-to-many (1:N) mapping of records.

:param op *redgrease.typing.Expander*: Function to map on the input records. The function must take one argument as input (input record) and return an iterable as an output (output records). Defaults to the 'identity-function', I.e. if input is an iterable will be expanded.

Parameters

- **optional)** – Function to map on the input records. The function must take one argument as input (input record) and return an iterable as an output (output records). Defaults to the 'identity-function', I.e. if input is an iterable will be expanded.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to None.
- ****kwargs** – Additional parameters to the FlatMap operation.

Returns Itself, i.e. the same GearsBuilder, but with its internal gear function updated with a FlatMap operation as last step.

Return type *GearsBuilder*

foreach(*op: Callable[[InputRecord], None], requirements: Optional[Iterable[str]] = None, **kwargs*) → *redgrease.runtime.GearsBuilder*

Instance-local *ForEach* operation performs one-to-the-same (1=1) mapping.

:param op *redgrease.typing.Processor*): Function to run on each of the input records. The function must take one argument as input (input record) and should not return anything.

Parameters

- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to None.
- ****kwargs** – Additional parameters to the ForEach operation.

Returns Itself, i.e. the same GearsBuilder, but with its internal gear function updated with a ForEach operation as last step.

Return type *GearsBuilder*

filter(*op: Optional[Callable[[InputRecord], bool]] = None*, *requirements: Optional[Iterable[str]] = None*, ***kwargs*) → *redgrease.runtime.GearsBuilder*

Instance-local *Filter* operation performs one-to-zero-or-one (1:bool) filtering of records.

:param op *redgrease.typing.Filterer*: Function to apply on the input records, to decide which ones to keep

The function must take one argument as input (input record) and return a bool. The input records evaluated to *True* will be kept as output records. Defaults to the 'identity-function', i.e. records are filtered based on their own trueess or falseness.

Parameters

- **optional)** – Function to apply on the input records, to decide which ones to keep. The function must take one argument as input (input record) and return a bool. The input records evaluated to *True* will be kept as output records. Defaults to the 'identity-function', i.e. records are filtered based on their own trueess or falseness.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to None.
- ****kwargs** – Additional parameters to the Filter operation.

Returns Itself, i.e. the same GearsBuilder, but with its internal gear function updated with a FIlter operation as last step.

Return type *GearsBuilder*

accumulate(*op: Optional[Callable[[T, InputRecord], T]] = None*, *requirements: Optional[Iterable[str]] = None*, ***kwargs*) → *redgrease.runtime.GearsBuilder*

Instance-local *Accumulate* operation performs many-to-one mapping (N:1) of records.

:param op *redgrease.typing.Accumulator*: Function to to apply on the input records.

The function must take two arguments as input:

- An accumulator value, and
- The input record.

It should aggregate the input record into the accumulator variable, which stores the state between the function's invocations. The function must return the accumulator's updated value. Defaults to a list accumulator, I.e. the output will be a list of all inputs.

Parameters

- **optional)** – Function to to apply on the input records. The function must take two arguments as input:
  - An accumulator value, and
  - The input record.

It should aggregate the input record into the accumulator variable, which stores the state between the function's invocations. The function must return the accumulator's updated value. Defaults to a list accumulator, I.e. the output will be a list of all inputs.

- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to None.
- ****kwargs** – Additional parameters to the Accumulate operation.

Returns Itself, i.e. the same GearsBuilder, but with its internal gear function updated with Accumulate operation as last step.

Return type *GearsBuilder*

localgroupby(*extractor: Optional[Callable[[InputRecord], Key]] = None*, *reducer: Optional[Callable[[Key, T, InputRecord], T]] = None*, *requirements: Optional[Iterable[str]] = None*, ***kwargs*) → *redgrease.runtime.GearsBuilder*

Instance-local *LocalGroupBy* operation performs many-to-less mapping (N:M) of records.

:param extractor *redgrease.typing.Extractor*: Function to apply on the input records, to extact the groupi
The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key. Defaults to the hash of the input.

Parameters **optional)** – Function to apply on the input records, to extact the grouping key. The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key. Defaults to the hash of the input.

:param reducer *redgrease.typing.Reducer*: Function to apply on the records of each group, to reduce to a si
value (per group). The function must take (a) a key, (b) an input record and (c) a variable that's called an accumulator. It performs similarly to the accumulator callback, with the difference being that it maintains an accumulator per reduced key / group. Defaults to a list accumulator, I.e. the output will be a list of all inputs, for each group.

Parameters

- **optional)** – Function to apply on the records of each group, to reduce to a single value (per group). The function must take (a) a key, (b) an input record and (c) a variable that's called an accumulator. It performs similarly to the accumulator callback, with the difference being that it maintains an accumulator per reduced key / group. Defaults to a list accumulator, I.e. the output will be a list of all inputs, for each group.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to None.
- ****kwargs** – Additional parameters to the LocalGroupBy operation.

Returns Itself, i.e. the same GearsBuilder, but with its internal gear function updated with a LocalGroupBy operation as last step.

Return type *GearsBuilder*

limit(*length: int*, *start: int = 0*, ***kwargs*) → *redgrease.runtime.GearsBuilder*

Instance-local *Limit* operation limits the number of records.

Parameters

- **length** (*int*) – The maximum number of records.
- **start** (*int, optional*) – The index of the first input record. Defaults to 0.

- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to None.
- ****kwargs** – Additional parameters to the Limit operation.

Returns Itself, i.e. the same GearsBuilder, but with its internal gear function updated with a Limit operation as last step.

Return type *GearsBuilder*

collect(***kwargs*) → *redgrease.runtime.GearsBuilder*

Cluster-global *Collect* operation collects the result records.

Parameters ****kwargs** – Additional parameters to the Collect operation.

Returns Itself, i.e. the same GearsBuilder, but with its internal gear function updated with a Collect operation as last step.

Return type *GearsBuilder*

repartition(*extractor: Callable[[InputRecord], Hashable], requirements: Optional[Iterable[str]] = None, **kwargs*) → *redgrease.runtime.GearsBuilder*

Cluster-global *Repartition* operation repartitions the records by shuffling them between shards.

:param extractor *redgrease.typing.Extractor*): Function that takes a record and calculates a key that is us

determine the hash slot, and consequently the shard, that the record should migrate to to. The function must take one argument as input (input record) and return a string (key). The hash slot, and consequently the destination shard, is determined by the value of the key.

Parameters

- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to None.
- ****kwargs** – Additional parameters to the Repartition operation.

Returns Itself, i.e. the same GearsBuilder, but with its internal gear function updated with a Repartition operation as last step.

Return type *GearsBuilder*

aggregate(*zero: Optional[T] = None, seqOp: Optional[Callable[[T, InputRecord], T]] = None, combOp: Optional[Callable[[T, T], T]] = None, requirements: Optional[Iterable[str]] = None, **kwargs*) → *redgrease.runtime.GearsBuilder*

Distributed *Aggregate* operation perform an aggregation on local data then a global aggregation on the local aggregations.

Parameters **zero** (*Any, optional*) – The initial / zero value of the accumulator variable. Defaults to an empty list.

:param seqOp *redgrease.typing.Accumulator*: A function to be applied on each of the input records, locall

shard. It must take two parameters: - an accumulator value, from previous calls - an input record The function aggregates the input into the accumulator variable, which stores the state between the function's invocations. The function must return the accumulator's updated value. Defaults to addition, if 'zero' is a number and to a list accumulator if 'zero' is a list.

Parameters **optional)** – A function to be applied on each of the input records, locally per shard. It must take two parameters: - an accumulator value, from previous calls - an input record The function aggregates the input into the accumulator variable, which stores the state between the function's invocations. The function must return the accumulator's

updated value. Defaults to addition, if 'zero' is a number and to a list accumulator if 'zero' is a list.

:param combOp *redgrease.typing.Accumulator*: A function to be applied on each of the aggregated results aggregation (i.e. the output of *seqOp*). It must take two parameters: - an accumulator value, from previous calls - an input record The function aggregates the input into the accumulator variable, which stores the state between the function's invocations. The function must return the accumulator's updated value. Defaults to re-use the *seqOp* function.

Parameters

- **optional)** – A function to be applied on each of the aggregated results of the local aggregation (i.e. the output of *seqOp*). It must take two parameters: - an accumulator value, from previous calls - an input record The function aggregates the input into the accumulator variable, which stores the state between the function's invocations. The function must return the accumulator's updated value. Defaults to re-use the *seqOp* function.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to None.
- ****kwargs** – Additional parameters to the Aggregate operation.

Returns Itself, i.e. the same GearsBuilder, but with its internal gear function updated with a Aggregate operation as last step.

Return type *GearsBuilder*

aggregateby(*extractor: Optional[Callable[[InputRecord], Key]] = None*, *zero: Optional[T] = None*, *seqOp: Optional[Callable[[Key, T, InputRecord], T]] = None*, *combOp: Optional[Callable[[Key, T, T], T]] = None*, *requirements: Optional[Iterable[str]] = None*, ***kwargs*) → *redgrease.runtime.GearsBuilder*

Distributed *AggregateBy* operation, behaves like aggregate, but separated on each key, extracted using the extractor.

:param extractor *redgrease.typing.Extractor*: Function to apply on the input records, to extact the groupi The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key. Defaults to the hash of the input.

Parameters

- **optional)** – Function to apply on the input records, to extact the grouping key. The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key. Defaults to the hash of the input.
- **zero** (*Any, optional*) – The initial / zero value of the accumulator variable. Defaults to an empty list.

:param seqOp *redgrease.typing.Accumulator*: A function to be applied on each of the input records, locall shard and group. It must take two parameters: - an accumulator value, from previous calls - an input record The function aggregates the input into the accumulator variable, which stores the state between the function's invocations. The function must return the accumulator's updated value. Defaults to a list reducer.

Parameters optional) – A function to be applied on each of the input records, locally per shard and group. It must take two parameters: - an accumulator value, from previous calls - an input record The function aggregates the input into the accumulator variable,

which stores the state between the function's invocations. The function must return the accumulator's updated value. Defaults to a list reducer.

:param combOp *redgrease.typing.Accumulator*): A function to be applied on each of the aggregated result
aggregation (i.e. the output of *seqOp*). It must take two parameters: - an accumulator value, from previous calls - an input record The function aggregates the input into the accumulator variable, which stores the state between the function's invocations. The function must return the accumulator's updated value. Defaults to re-use the *seqOp* function.

Parameters

- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to None.
- ****kwargs** – Additional parameters to the AggregateBy operation.

Returns Itself, i.e. the same GearsBuilder, but with its internal gear function updated with a AggregateBy operation as last step.

Return type *GearsBuilder*

groupby(*extractor: Optional[Callable[[InputRecord], Key]] = None*, *reducer: Optional[Callable[[Key, T, InputRecord], T]] = None*, *requirements: Optional[Iterable[str]] = None*, ***kwargs*) → *redgrease.runtime.GearsBuilder*
Cluster-local *GroupBy* operation performing a many-to-less (N:M) grouping of records.

:param extractor *redgrease.typing.Extractor*: Function to apply on the input records, to extact the groupi
The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key. Defaults to the hash of the input.

Parameters **optional)** – Function to apply on the input records, to extact the grouping key. The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key. Defaults to the hash of the input.

:param reducer *redgrease.typing.Reducer*: Function to apply on the records of each group, to reduce to a si
value (per group). The function must take (a) a key, (b) an input record and (c) a variable that's called an accumulator. It performs similarly to the accumulator callback, with the difference being that it maintains an accumulator per reduced key / group. Defaults to a list reducer.

Parameters

- **optional)** – Function to apply on the records of each group, to reduce to a single value (per group). The function must take (a) a key, (b) an input record and (c) a variable that's called an accumulator. It performs similarly to the accumulator callback, with the difference being that it maintains an accumulator per reduced key / group. Defaults to a list reducer.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to None.
- ****kwargs** – Additional parameters to the GroupBy operation.

Returns Itself, i.e. the same GearsBuilder, but with its internal gear function updated with a GroupBy operation as last step.

Return type *GearsBuilder*

batchgroupby(*extractor: Optional[Callable[[InputRecord], Key]] = None*, *reducer: Optional[Callable[[Key, Iterable[T]], InputRecord]] = None*, *requirements: Optional[Iterable[str]] = None*, ***kwargs*) → *redgrease.runtime.GearsBuilder*

Cluster-local *GroupBy* operation, performing a many-to-less (N:M) grouping of records.

Note: Using this operation may cause a substantial increase in memory usage during runtime. Consider using the GroupBy

:param extractor *redgrease.typing.Extractor*: Function to apply on the input records, to extact the groupi

The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key. Defaults to the hash of the input.

Parameters **optional)** – Function to apply on the input records, to extact the grouping key. The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key. Defaults to the hash of the input.

:param reducer *redgrease.typing.Reducer*): Function to apply on the records of each group, to reduce to a s

value (per group). The function must take (a) a key, (b) an input record and (c) a variable that's called an accumulator. It performs similarly to the accumulator callback, with the difference being that it maintains an accumulator per reduced key / group. Default is the length (*len*) of the input.

Parameters ****kwargs** – Additional parameters to the BatchGroupBy operation.

Returns Itself, i.e. the same GearsBuilder, but with its internal gear function updated with a BatchGroupBy operation as last step.

Return type *GearsBuilder*

sort(*reverse: bool = True*, *requirements: Optional[Iterable[str]] = None*, ***kwargs*) → *redgrease.runtime.GearsBuilder*

Sort the records

Parameters

- **reverse** (*bool, optional*) – Sort in descending order (higher to lower). Defaults to True.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to None.
- ****kwargs** – Additional parameters to the Sort operation.

Returns Itself, i.e. the same GearsBuilder, but with its internal gear function updated with a Sort operation as last step.

Return type *GearsBuilder*

distinct(***kwargs*) → *redgrease.runtime.GearsBuilder*

Keep only the *Distinct* values in the data.

Parameters ****kwargs** – Additional parameters to the Distinct operation.

Returns Itself, i.e. the same GearsBuilder, but with its internal gear function updated with a Distinct operation as last step.

Return type *GearsBuilder*

count(***kwargs*) → *redgrease.runtime.GearsBuilder*

Count the number of records in the execution.

Parameters ****kwargs** – Additional parameters to the Count operation.

Returns Itself, i.e. the same GearsBuilder, but with its internal gear function updated with a Count operation as last step.

Return type *GearsBuilder*

countby(*extractor: Callable[[InputRecord], Hashable] = <function GearsBuilder.<lambda>>, requirements: Optional[Iterable[str]] = None, **kwargs*) → *redgrease.runtime.GearsBuilder*

Distributed *CountBy* operation counting the records grouped by key.

:param extractor *redgrease.typing.Extractor*): Function to apply on the input records, to extact the group

The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key. Defaults to 'lambda x: str(x)'.

Parameters

- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to None.
- ****kwargs** – Additional parameters to the CountBy operation.

Returns Itself, i.e. the same GearsBuilder, but with its internal gear function updated with a CountBy operation as last step.

Return type *GearsBuilder*

avg(*extractor: Callable[[InputRecord], float] = <function GearsBuilder.<lambda>>, requirements: Optional[Iterable[str]] = None, **kwargs*) → *redgrease.runtime.GearsBuilder*

Distributed *Avg* operation, calculating arithmetic average of the records.

:param extractor *redgrease.typing.Extractor*): Function to apply on the input records, to extact the group

The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key. Defaults to 'lambda x: float(x)'.

Parameters

- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to None.
- ****kwargs** – Additional parameters to the map operation.

Returns

A new "open" gear function with an avg operation as last step. GearsBuilder - The same GearBuilder, but with updated function.

Note that for GearBuilder this method does **not** return a new GearFunction, but instead returns the same GearBuilder, but with its internal function updated.

Return type *OpenGearFunction*

property reader

The reader type, generating the initial input records to the GearFunction.

Returns Either `"KeysReader"`, `"KeysOnlyReader"`, `"StreamReader"`, `"PythonReader"`, `"ShardsIDReader"`, `"CommandReader"` or `None` (If no reader is defined).

Return type str

property supports_batch_mode

Indicates if the function can run in Batch-mode, by closing it with a *run* action.

Returns `True` if the function supports batch mode, `False` if not.

Return type bool

property supports_event_mode

Indicates if the function can run in Event-mode, by closing it with a *register* action.

Returns `True` if the function supports event mode, `False` if not.

Return type bool

Now we are finally ready to start building some Gear Functions.

Courtesy of :  Pte. Ltd.

CHAPTER

SIX

GEARFUNCTION

GearFunction objects are RedGrease's representation of RedisGears Gear functions. There are a couple of different classes of GearFunctions to be aware of.

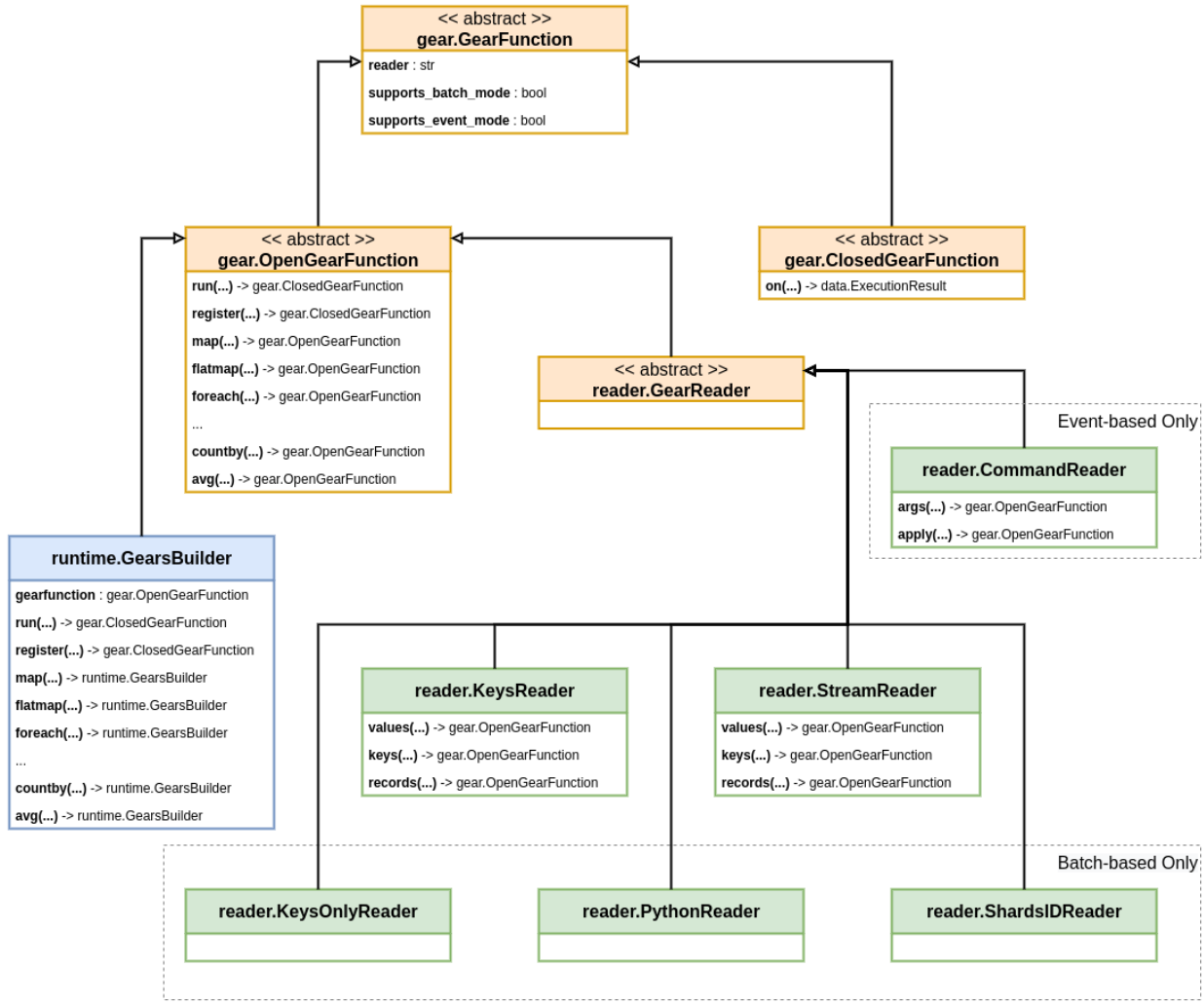


Fig. 1: UML Relationships between different GearFunction

These function objects can be dynamically constructed in either your application code or in separate script files. They are constructed using either the *GearsBuilder* (blue) or any of the *Readers* (green):

- *KeysReader*
- *KeysOnlyReader*
- *StreamReader*
- *PythonReader*
- *ShardsIDReader*
- *CommandReader*

These are responsible for reading data from different sources into records, either in batch on-demand or when some event occurs.

GearFunctions are created by "applying" operations, such as *Map*, *GroupBy*, *Aggregate* etc, to either any of the *Readers*, a *GearsBuilder*, or some other GearFunction.

With the exception of the *GearsBuilder*, "applying" an operation on a GearFunction, doesn't actually modify it, but instead creates a new function with that operation added.

All Gear functions therefore consists of a chain of zero or more operations, with some reader at the beginning.

Some of the Reader types have additional reader-specific operations, that can only be "applied" as as their first operation.

Gear functions are "terminated" by either one of two special *Actions*. Either *Run*, for immediate "batch-mode" execution, or *Register*, for registering the function for "event-mode" execution.

Some Readers support both "batch-mode" and "event-mode", but some only support one of the modes.

GearFunctions that **have not** been "terminated" by either of the *Actions*, are referred to as an *Open GearFunction*, as they **can be extended** with more operations, creating new GearFunctions.

GearFunctions that **have** been "terminated" by either of the *Actions*, are referred to as a *Closed GearFunction*, as they **cannot be extended** with more operations, creating new GearFunctions, but they can be executed.

Note: This "open/closed" terminology is not explicitly used by RedisLabs in their RedisGears Function Documentation, but it was the most intuitive terminology I could think of, to describe how their design of GearsBuilder actually behaves.

Early versions of RedGrease used the term "partial" instead of "open". This was deemed a bit confusing, because "partial functions" is a very specific, but completely different, thing in mathematics. It was also sometimes confused with "partial application", which in computer science is yet another completely different (but powerful) concept.

Every *Open GearFunction*, including the *GearsBuilder*, implement the default set of operations.

When a GearFunction is executed, the Reader reads its data, and pass each record to its first operation, which modifies, filters or aggregates these records into some new output records, which in turn are passed to the next operation and so on, until the last operation.

The output of the final operation is then either, returned to the caller if it was a "batch-mode" execution **and** `unblocking` was not set to `True` in *`Gears.pyexecute()`*, or stored for later retrieval otherwise.

6.1 Open GearFunction

You would never instantiate a *gears.OpenGearFunction* yourself, but all "open" *GearFunctions* that has not yet been "closed" with the *Run* or *Register Actions*, inherits from this class.

It is this class that under the hood is responsible for "applying" operations and *Actions*, and thus creating new GearFunctions.

This includes both the *runtime.GearsBuilder* as well as the *Readers*:

class redgrease.gears.**OpenGearFunction**(*operation: redgrease.gears.Operation*, *input_function: Optional[*redgrease.gears.OpenGearFunction*] = None*, *requirements: Optional[Iterable[str]] = None*)

An open Gear function is a Gear function that is not yet "closed" with a *Run* action or a *Register* action.

Open Gear functions can be used to create new "open" gear functions by applying operations, or it can create a closed Gear function by applying either the *Run* action or a *Register* action.

run(*arg: Optional[str] = None*, *convertToStr: bool = True*, *collect: bool = True*, *requirements: Optional[Iterable[str]] = None*, *on=None*, ***kwargs*) → *redgrease.gears.ClosedGearFunction*[InputRecord]

Create a "closed" function to be *Run* as in "batch-mode".

Batch functions are executed once and exits once the data is exhausted by its reader.

Parameters

- **arg** (*str, optional*) – An optional argument that's passed to the reader as its defaultArg. It means the following:
  - A glob-like pattern for the KeysReader and KeysOnlyReader readers.
  - A key name for the StreamReader reader.
  - A Python generator for the PythonReader reader.

  Defaults to `None`.
- **convertToStr** (*bool, optional*) – When `True`, adds a map operation to the flow's end that stringifies records. Defaults to `False`.
- **collect** (*bool, optional*) – When `True` adds a collect operation to flow's end. Defaults to `False`.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to `None`.
- **on** (*redis.Redis*) – Immediately execute the function on this RedisGears system.
- ****kwargs** – Additional parameters to the run operation.

Returns A new closed batch function, if *on* is **not** specified. An execution result, if *on* **is** specified.

Return type Union[*ClosedGearFunction*, redgrease.data.ExecutionResult]

Raises **TypeError** – If the function does not support batch mode.

register(*prefix: str = '*', convertToStr: bool = True, collect: bool = True, mode: Optional[str] = None, onRegistered: Optional[Callable[], None]] = None, eventTypes: Optional[Iterable[str]] = None, keyTypes: Optional[Iterable[str]] = None, readValue: Optional[bool] = None, batch: Optional[int] = None, duration: Optional[int] = None, onFailedPolicy: Optional[str] = None, onFailedRetryInterval: Optional[int] = None, trimStream: Optional[bool] = None, trigger: Optional[str] = None, requirements: Optional[Iterable[str]] = None, on=None, **kwargs*) → *redgrease.gears.ClosedGearFunction*[InputRecord]

Create a "closed" function to be *Register* 'ed as an event-triggered function.

Event functions are executed each time an event arrives. Each time it is executed, the function operates on the event's data and once done is suspended until its future invocations by new events. :param prefix: Key prefix pattern to match on.

Not relevant for 'CommandReader' readers (see 'trigger'). Defaults to `"*"`.

Parameters

- **convertToStr** (*bool, optional*) – When `True` adds a map operation to the flow's end that stringifies records. Defaults to `True`.
- **collect** (*bool, optional*) – When `True` adds a collect operation to flow's end. Defaults to `False`.
- **mode** (*str, optional*) – The execution mode of the function. Can be one of:
  - `"async"`:

    Execution will be asynchronous across the entire cluster.
  - `"async_local"`:

    Execution will be asynchronous and restricted to the handling shard.
  - `"sync"`:

    Execution will be synchronous and local.

  Defaults to `"async"`.
- **onRegistered**(*Registrator, optional*) – A function that's called on each shard upon function registration. It is a good place to initialize non-serializeable objects such as network connections. Defaults to `None`.
- **eventTypes** (*Iterable[str], optional*) – For KeysReader only. A whitelist of event types that trigger execution when the KeysReader are used. The list may contain one or more:
  - Any Redis or module command
  - Any Redis event

  Defaults to `None`.
- **keyTypes** (*Iterable[str], optional*) – For KeysReader and KeysOnlyReader only. A whitelist of key types that trigger execution when using the KeysReader or KeysOnlyReader readers. The list may contain one or more from the following:
  - Redis core types:

    `"string"`, `"hash"`, `"list"`, `"set"`, `"zset"` or `"stream"`
  - Redis module types:

"module"

Defaults to None.

- **readValue** (*bool, optional*) – For KeysReader only. When False the value will not be read, so the 'type' and 'value' of the record will be set to None. Defaults to True.
- **batch** (*int, optional*) – For StreamReader only. The number of new messages that trigger execution. Defaults to 1.
- **duration** (*int, optional*) – For StreamReader only. The time to wait before execution is triggered, regardless of the batch size (0 for no duration). Defaults to 0.
- **onFailedPolicy** (*str, optional*) – For StreamReader only. The policy for handling execution failures. May be one of:
  - "continue":

    Ignores a failure and continues to the next execution. This is the default policy.
  - "abort":

    Stops further executions.
  - "retry":

    Retries the execution after an interval specified with onFailedRetryInterval (default is one second).

  Defaults to "continue".
- **onFailedRetryInterval** (*int, optional*) – For StreamReader only. The interval (in milliseconds) in which to retry in case onFailedPolicy is 'retry'. Defaults to 1.
- **trimStream** (*bool, optional*) – For StreamReader only. When True the stream will be trimmed after execution Defaults to True.
- **trigger** (*str*) – For 'CommandReader' only, and mandatory. The trigger string that will trigger the function.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to None.
- **on** (*redis.Redis*) – Immediately execute the function on this RedisGears system.
- ****kwargs** – Additional parameters to the register operation.

Returns A new closed event function, if *on* is **not** specified. An execution result, if *on* **is** specified.

Return type Union[*ClosedGearFunction*, redgrease.data.ExecutionResult]

Raises **TypeError** – If the function does not support event mode.

map(*op: Callable[[InputRecord], OutputRecord]*, *requirements: Optional[Iterable[str]] = None*, ***kwargs*) → *redgrease.gears.OpenGearFunction*[*redgrease.typing.OutputRecord*]

Instance-local *Map* operation that performs a one-to-one (1:1) mapping of records.

Parameters

- **op** (*redgrease.typing.Mapper*) – Function to map on the input records. The function must take one argument as input (input record) and return something as an output (output record).

- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to `None`.
- ****kwargs** – Additional parameters to the *Map* operation.

Returns A new "open" gear function with a *Map* operation as last step.

Return type *OpenGearFunction*

flatmap(*op: Optional[Callable[[InputRecord], Iterable[OutputRecord]]] = None*, *requirements: Optional[Iterable[str]] = None*, ***kwargs*) → *redgrease.gears.OpenGearFunction*[Iterable[*redgrease.typing.OutputRecord*]]

Instance-local *FlatMap* operation that performs one-to-many (1:N) mapping of records.

Parameters

- **op** (*redgrease.typing.Expander*, optional) – Function to map on the input records. The function must take one argument as input (input record) and return an iterable as an output (output records). Defaults to the 'identity-function', I.e. if input is an iterable will be expanded.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to `None`.
- ****kwargs** – Additional parameters to the *FlatMap* operation.

Returns A new "open" gear function with a *FlatMap* operation as last step.

Return type *OpenGearFunction*

foreach(*op: Callable[[InputRecord], None]*, *requirements: Optional[Iterable[str]] = None*, ***kwargs*) → *redgrease.gears.OpenGearFunction*[*redgrease.typing.InputRecord*]

Instance-local *ForEach* operation performs one-to-the-same (1=1) mapping.

Parameters

- **op** (*redgrease.typing.Processor*) – Function to run on each of the input records. The function must take one argument as input (input record) and should not return anything.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to `None`.
- ****kwargs** – Additional parameters to the *ForEach* operation.

Returns A new "open" gear function with a *ForEach* operation as last step.

Return type *OpenGearFunction*

filter(*op: Optional[Callable[[InputRecord], bool]] = None*, *requirements: Optional[Iterable[str]] = None*, ***kwargs*) → *redgrease.gears.OpenGearFunction*[*redgrease.typing.InputRecord*]

Instance-local *Filter* operation performs one-to-zero-or-one (1:bool) filtering of records.

Parameters

- **op** (*redgrease.typing.Filterer*, optional) – Function to apply on the input records, to decide which ones to keep. The function must take one argument as input (input record) and return a bool. The input records evaluated to `True` will be kept as output records. Defaults to the 'identity-function', i.e. records are filtered based on their own trueness or falseness.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to `None`.
- ****kwargs** – Additional parameters to the *Filter* operation.

Returns A new "open" gear function with a *Filter* operation as last step.

Return type *OpenGearFunction*

accumulate(*op: Optional[Callable[[T, InputRecord], T]] = None*, *requirements: Optional[Iterable[str]] = None*, ***kwargs*) → *redgrease.gears.OpenGearFunction*[T]

Instance-local *Accumulate* operation performs many-to-one mapping (N:1) of records.

Parameters

- **op** (*redgrease.typing.Accumulator*, optional) – Function to to apply on the input records. The function must take two arguments as input:
  - An accumulator value, and
  - The input record.

  It should aggregate the input record into the accumulator variable, which stores the state between the function's invocations. The function must return the accumulator's updated value. Defaults to a list accumulator, I.e. the output will be a list of all inputs.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to `None`.
- ****kwargs** – Additional parameters to the *Accumulate* operation.

Returns A new "open" gear function with *Accumulate* operation as last step.

Return type *OpenGearFunction*

localgroupby(*extractor: Optional[Callable[[InputRecord], Key]] = None*, *reducer: Optional[Callable[[Key, T, InputRecord], T]] = None*, *requirements: Optional[Iterable[str]] = None*, ***kwargs*) → *redgrease.gears.OpenGearFunction*[Dict[*redgrease.typing.Key*, T]]

Instance-local *LocalGroupBy* operation performs many-to-less mapping (N:M) of records.

Parameters

- **extractor** (*redgrease.typing.Extractor*, optional) – Function to apply on the input records, to extact the grouping key. The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key. Defaults to the hash of the input.
- **reducer** (*redgrease.typing.Reducer*, optional) – Function to apply on the records of each group, to reduce to a single value (per group). The function must take (a) a key, (b) an input record and (c) a variable that's called an accumulator. It performs similarly to the accumulator callback, with the difference being that it maintains an accumulator per reduced key / group. Defaults to a list accumulator, I.e. the output will be a list of all inputs, for each group.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to `None`.
- ****kwargs** – Additional parameters to the *LocalGroupBy* operation.

Returns A new "open" gear function with a *LocalGroupBy* operation as last step.

Return type *OpenGearFunction*

limit(*length: int*, *start: int = 0*, ***kwargs*) → *redgrease.gears.OpenGearFunction*[*redgrease.typing.InputRecord*]

Instance-local *Limit* operation limits the number of records.

Parameters

- **length** (*int*) – The maximum number of records.

- **start** (*int, optional*) – The index of the first input record. Defaults to 0.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to `None`.
- ****kwargs** – Additional parameters to the *Limit* operation.

Returns A new "open" gear function with a *Limit* operation as last step.

Return type *OpenGearFunction*

collect(***kwargs*) → *redgrease.gears.OpenGearFunction*[*redgrease.typing.InputRecord*]

Cluster-global *Collect* operation collects the result records.

Parameters ****kwargs** – Additional parameters to the *Collect* operation.

Returns A new "open" gear function with a *Collect* operation as last step.

Return type *OpenGearFunction*

repartition(*extractor: Callable[[InputRecord], Hashable], requirements: Optional[Iterable[str]] = None, **kwargs*) → *redgrease.gears.OpenGearFunction*[*redgrease.typing.InputRecord*]

Cluster-global *Repartition* operation repartitions the records by shuffling them between shards.

Parameters

- **extractor** (`Extractor`) – Function that takes a record and calculates a key that is used to determine the hash slot, and consequently the shard, that the record should migrate to to. The function must take one argument as input (input record) and return a string (key). The hash slot, and consequently the destination shard, is determined by the value of the key.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to `None`.
- ****kwargs** – Additional parameters to the *Repartition* operation.

Returns A new "open" gear function with a *Repartition* operation as last step.

Return type *OpenGearFunction*

aggregate(*zero: Optional[T] = None, seqOp: Optional[Callable[[T, InputRecord], T]] = None, combOp: Optional[Callable[[T, T], T]] = None, requirements: Optional[Iterable[str]] = None, **kwargs*) → *redgrease.gears.OpenGearFunction*[T]

Distributed *Aggregate* operation perform an aggregation on local data then a global aggregation on the local aggregations.

Parameters

- **zero** (*Any, optional*) – The initial / zero value of the accumulator variable. Defaults to an empty list.
- **seqOp** (*redgrease.typing.Accumulator*, optional) – A function to be applied on each of the input records, locally per shard. It must take two parameters: - an accumulator value, from previous calls - an input record The function aggregates the input into the accumulator variable, which stores the state between the function's invocations. The function must return the accumulator's updated value. Defaults to addition, if 'zero' is a number and to a list accumulator if 'zero' is a list.
- **combOp** (*redgrease.typing.Accumulator*, optional) – A function to be applied on each of the aggregated results of the local aggregation (i.e. the output of *seqOp*). It must take two parameters: - an accumulator value, from previous calls - an input record The function aggregates the input into the accumulator variable,

which stores the state between the function's invocations. The function must return the accumulator's updated value. Defaults to re-use the *seqOp* function.

- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to None.
- ****kwargs** – Additional parameters to the ref:*op_aggregate* operation.

Returns A new "open" gear function with a ref:*op_aggregate* operation as last step.

Return type *OpenGearFunction*

aggregateby(*extractor: Optional[Callable[[InputRecord], Key]] = None*, *zero: Optional[T] = None*, *seqOp: Optional[Callable[[Key, T, InputRecord], T]] = None*, *combOp: Optional[Callable[[Key, T, T], T]] = None*, *requirements: Optional[Iterable[str]] = None*, ***kwargs*) → *redgrease.gears.OpenGearFunction*[Dict[*redgrease.typing.Key*, T]]

Distributed *AggregateBy* operation, behaves like aggregate, but separated on each key, extracted using the extractor.

Parameters

- **extractor** (*redgrease.typing.Extractor*, optional) – Function to apply on the input records, to extact the grouping key. The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key. Defaults to the hash of the input.
- **zero** (*Any, optional*) – The initial / zero value of the accumulator variable. Defaults to an empty list.
- **seqOp** (*redgrease.typing.Accumulator*, optional) – A function to be applied on each of the input records, locally per shard and group. It must take two parameters: - an accumulator value, from previous calls - an input record The function aggregates the input into the accumulator variable, which stores the state between the function's invocations. The function must return the accumulator's updated value. Defaults to a list reducer.
- **combOp** (*redgrease.typing.Accumulator*) – A function to be applied on each of the aggregated results of the local aggregation (i.e. the output of *seqOp*). It must take two parameters: - an accumulator value, from previous calls - an input record The function aggregates the input into the accumulator variable, which stores the state between the function's invocations. The function must return the accumulator's updated value. Defaults to re-use the *seqOp* function.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to None.
- ****kwargs** – Additional parameters to the *AggregateBy* operation.

Returns A new "open" gear function with a *AggregateBy* operation as last step.

Return type *OpenGearFunction*

groupby(*extractor: Optional[Callable[[InputRecord], Key]] = None*, *reducer: Optional[Callable[[Key, T, InputRecord], T]] = None*, *requirements: Optional[Iterable[str]] = None*, ***kwargs*) → *redgrease.gears.OpenGearFunction*[Dict[*redgrease.typing.Key*, T]]

Cluster-local *GroupBy* operation performing a many-to-less (N:M) grouping of records.

Parameters

- **extractor** (*redgrease.typing.Extractor*, optional) – Function to apply on the input records, to extact the grouping key. The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key. Defaults to the hash of the input.

- **reducer** (*redgrease.typing.Reducer*, optional) – Function to apply on the records of each group, to reduce to a single value (per group). The function must take (a) a key, (b) an input record and (c) a variable that's called an accumulator. It performs similarly to the accumulator callback, with the difference being that it maintains an accumulator per reduced key / group. Defaults to a list reducer.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to `None`.
- ****kwargs** – Additional parameters to the *GroupBy* operation.

Returns A new "open" gear function with a *GroupBy* operation as last step.

Return type *OpenGearFunction*

batchgroupby(*extractor: Optional[Callable[[InputRecord], Key]] = None*, *reducer: Optional[Callable[[Key, Iterable[T]], InputRecord]] = None*, *requirements: Optional[Iterable[str]] = None*, ***kwargs*) → *redgrease.gears.OpenGearFunction*[Dict[*redgrease.typing.Key*, T]]

Cluster-local *GroupBy* operation, performing a many-to-less (N:M) grouping of records.

Note: Using this operation may cause a substantial increase in memory usage during runtime. Consider using the GroupBy

Parameters

- **extractor** (*redgrease.typing.Extractor*, optional) – Function to apply on the input records, to extact the grouping key. The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key. Defaults to the hash of the input.
- **reducer** (*redgrease.typing.Reducer*) – Function to apply on the records of each group, to reduce to a single value (per group). The function must take (a) a key, (b) an input record and (c) a variable that's called an accumulator. It performs similarly to the accumulator callback, with the difference being that it maintains an accumulator per reduced key / group. Default is the length (*len*) of the input.
- ****kwargs** – Additional parameters to the *GroupBy* operation.

Returns A new "open" gear function with a *GroupBy* operation as last step.

Return type *OpenGearFunction*

sort(*reverse: bool = True*, *requirements: Optional[Iterable[str]] = None*, ***kwargs*) → *redgrease.gears.OpenGearFunction*[*redgrease.typing.InputRecord*]

Sort the records

Parameters

- **reverse** (*bool, optional*) – Sort in descending order (higher to lower). Defaults to `True`.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to `None`.
- ****kwargs** – Additional parameters to the *Sort* operation.

Returns A new "open" gear function with a *Sort* operation as last step.

Return type *OpenGearFunction*

distinct(***kwargs*) → *redgrease.gears.OpenGearFunction*[*redgrease.typing.InputRecord*]

Keep only the *Distinct* values in the data.

Parameters ****kwargs** – Additional parameters to the *Distinct* operation.

Returns A new "open" gear function with a *Distinct* operation as last step.

Return type *OpenGearFunction*

count(***kwargs*) → *redgrease.gears.OpenGearFunction*[int]
Count the number of records in the execution.

Parameters ****kwargs** – Additional parameters to the *Count* operation.

Returns A new "open" gear function with a *Count* operation as last step.

Return type *OpenGearFunction*

countby(*extractor: Callable[[InputRecord], Hashable] = <function OpenGearFunction.<lambda>>, requirements: Optional[Iterable[str]] = None, **kwargs*) → *redgrease.gears.OpenGearFunction*[Dict[Hashable, int]]
Distributed *CountBy* operation counting the records grouped by key.

Parameters

- **extractor** (*redgrease.typing.Extractor*) – Function to apply on the input records, to extact the grouping key. The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key. Defaults to `lambda x: str(x)`.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to `None`.
- ****kwargs** – Additional parameters to the *CountBy* operation.

Returns A new "open" gear function with a *CountBy* operation as last step.

Return type *OpenGearFunction*

avg(*extractor: Callable[[InputRecord], float] = <function OpenGearFunction.<lambda>>, requirements: Optional[Iterable[str]] = None, **kwargs*) → *redgrease.gears.OpenGearFunction*[float]
Distributed *Avg* operation, calculating arithmetic average of the records.

Parameters

- **extractor** (*redgrease.typing.Extractor*) – Function to apply on the input records, to extact the grouping key. The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key. Defaults to `lambda x: float(x)`.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to `None`.
- ****kwargs** – Additional parameters to the *Avg* operation.

Returns A new "open" gear function with an *Avg* operation as last step.

Return type *OpenGearFunction*

6.2 Closed GearFunction

class redgrease.gears.**ClosedGearFunction**

Closed Gear functions are GearsFunctions that have been "closed" with a *Run* action or a *Register* action.

Closed Gear functions cannot add more operations, but can be executed in RedisGears.

on(*gears_server*, *unblocking: bool = False*, *requirements: Optional[Iterable[str]] = None*, *replace: Optional[bool] = None*, ***kwargs*)

Execute the function on a RedisGears. This is equivalent to passing the function to *Gears.pyexecute*

Parameters

- **gears_server** (*[type]*) – Redis client / connection object.
- **unblocking** (*bool, optional*) – Execute function unblocking, i.e. asynchronous. Defaults to `False`.
- **requirements** (*Iterable[str], optional*) – Additional requirements / dependency Python packages. Defaults to `None`.

Returns The result of the function, just as *Gears.pyexecute*

Return type redgrease.data.ExecutionResult

6.2.1 GearsBuilder

If you are familiar with RedisGears from before, then the `runtime.Gearsbuilder` should be very familiar. In fact the RedGrease version is designed to be backwards compatible with the Context Builder of RedisGears, with the same name.

The the *GearsBuilder* is technically a part of on the *Builtin Runtime Functions* and is exposed both through *redgrease.runtime.GearsBuilder* as well as `redgrease.GearsBuilder`.

Check out the *Builtin Runtime Functions* and specifically the section on the *GearsBuilder* for more details.

6.2.2 Readers

6.3 KeysReader

class redgrease.reader.**KeysReader**(*default_key_pattern: str = '*'*, *desc: Optional[str] = None*, *requirements: Optional[Iterable[str]] = None*)

KeysReader is a convenience class for GearsBuilder("KeysReader", ...)

Instantiate a KeysReader "open" Gear function.

Parameters

- **default_key_pattern** (*str, optional*) – Default Redis key pattern for the keys (and its values, type) to read. Defaults to "*".
- **desc** (*str, optional*) – An optional description. Defaults to None.
- **requirements** (*Iterable[str], optional*) – Package dependencies for the gear train. Defaults to None.

values(*type=Ellipsis*, *event=Ellipsis*) → *redgrease.gears.OpenGearFunction*

Filter out and select the values of the records only.

Parameters

- **type** (*Union[str, Container[str]], optional*) – A single string, or a container of several strings, representing the Redis type(s) of keys to select. Valid values include: “string”, “hash”, “list”, “set”, “zset”, “stream” or “module”. Defaults to ... (Ellipsis), meaning any type.
- **event** (*Union[str, Container[str]], optional*) – A single string, or a container of several strings, representing the Redis command or event `that . Defaults to ... (Ellipsis), meaning any event.

Returns A new “open” gear function generating the matching values.

Return type *OpenGearFunction*

keys(*type=Ellipsis*, *event=Ellipsis*) → *redgrease.gears.OpenGearFunction*[str]

Filter out and select the keys of the records only.

Parameters

- **type** (*Union[str, Container[str]], optional*) – A single string, or a container of several strings, representing the Redis type(s) of keys to select. Valid values include: “string”, “hash”, “list”, “set”, “zset”, “stream” or “module”. Defaults to ... (Ellipsis), meaning any type.
- **event** (*Union[str, Container[str]], optional*) – A single string, or a container of several strings, representing the Redis command or event `that . Defaults to ... (Ellipsis), meaning any event.

Returns A new “open” gear function generating the matching keys.

Return type *OpenGearFunction*[str]

records(*type=Ellipsis*, *event=Ellipsis*) → *redgrease.gears.OpenGearFunction*[redgrease.utils.Record]

Filter out and map the records to *redgrease.utils.Record* objects.

This provides the fields *key*, *value*, *type* and *event* as typed attributes on an object, instead of items in a *dict*, making it a little bit more pleasant to work with.

Parameters

- **type** (*Union[str, Container[str]], optional*) – A single string, or a container of several strings, representing the Redis type(s) of keys to select. Valid values include: “string”, “hash”, “list”, “set”, “zset”, “stream” or “module”. Defaults to ... (Ellipsis), meaning any type.
- **event** (*Union[str, Container[str]], optional*) – A single string, or a container of several strings, representing the Redis command or event `that. Defaults to ... (Ellipsis), meaning any event.

Returns A new “open” gear function generating the matching Record values.

Return type *OpenGearFunction*[redgrease.utils.Record]

6.4 KeysOnlyReader

class redgrease.reader.**KeysOnlyReader**(*default_key_pattern: str = '*', desc: Optional[str] = None, requirements: Optional[Iterable[str]] = None*)

KeysOnlyReader is a convenience class for GearsBuilder("KeysOnlyReader", ...)

Instantiate a KeysOnlyReader "open" Gear function.

Parameters

- **default_key_pattern** (*str, optional*) – Default Redis keys pattern for the keys to read. Defaults to "*".
- **desc** (*str, optional*) – An optional description. Defaults to None.
- **requirements** (*Iterable[str], optional*) – Package dependencies for the gear train. Defaults to None.

6.5 StreamReader

class redgrease.reader.**StreamReader**(*default_key_pattern: str = '*', desc: Optional[str] = None, requirements: Optional[Iterable[str]] = None*)

StreamReader is a convenience class for GearsBuilder("StreamReader", ...)

Instantiate a StreamReader "open" Gear function.

Parameters

- **default_key_pattern** (*str, optional*) – Default Redis keys pattern for the redis stream(s) to read. Defaults to "*".
- **desc** (*str, optional*) – An optional description. Defaults to None.
- **requirements** (*Iterable[str], optional*) – Package dependencies for the gear train. Defaults to None.

values() → *redgrease.gears.OpenGearFunction*[Dict]

Select the values of the stream only.

Returns A new "open" gear function generating values.

Return type *OpenGearFunction*

keys() → *redgrease.gears.OpenGearFunction*[str]

Select the keys, i.e. stream names, only.

Returns A new "open" gear function generating names.

Return type *OpenGearFunction*[str]

records() → *redgrease.gears.OpenGearFunction*[redgrease.utils.StreamRecord]

Filter out and map the records to *redgrease.utils.StreamRecord* objects.

This provides the fields *key*, *id* and *value* as typed attributes on an object, instead of items in a *dict*, making it a little bit more pleasant to work with.

Returns A new "open" gear function generating the matching Record values.

Return type *OpenGearFunction*[redgrease.utils.Record]

6.6 PythonReader

class redgrease.reader.**PythonReader**(*desc: Optional[str] = None*, *requirements: Optional[Iterable[str]] = None*)

PythonReader is a convenience class for GearsBuilder("PythonReader", ...)

Instantiate a PythonReader "open" Gear function.

Parameters

- **desc** (*str, optional*) – An optional description. Defaults to None.
- **requirements** (*Iterable[str], optional*) – Package dependencies for the gear train. Defaults to None.

6.7 ShardsIDReader

class redgrease.reader.**ShardsIDReader**(*desc: Optional[str] = None*, *requirements: Optional[Iterable[str]] = None*)

ShardsIDReader is a convenience class for GearsBuilder("ShardsIDReader", ...)

Instantiate a ShardsIDReader "open" Gear function.

Parameters

- **desc** (*str, optional*) – An optional description. Defaults to None.
- **requirements** (*Iterable[str], optional*) – Package dependencies for the gear train. Defaults to None.

6.8 CommandReader

class redgrease.reader.**CommandReader**(*desc: Optional[str] = None*, *requirements: Optional[Iterable[str]] = None*)

CommandReader is a convenience class for GearsBuilder("CommandReader", ...)

Instantiate a CommandReader "open" Gear function.

Parameters

- **desc** (*str, optional*) – An optional description. Defaults to None.
- **requirements** (*Iterable[str], optional*) – Package dependencies for the gear train. Defaults to None.

args(*max_count: Optional[int] = None*) → *redgrease.gears.OpenGearFunction*

Ignore the trigger name and take only the arguments following the trigger.

Parameters **max_count** (*int, optional*) – Maximum number of args to take. Any additional args will be truncated. Defaults to None.

Returns A new "open" gear function only generating the trigger arguments.

Return type *redgrease.gears.OpenGearFunction*

apply(*fun: Callable[[...], OutputRecord]*, *requirements: Optional[Iterable[str]] = None*, ***kwargs*) → *redgrease.gears.OpenGearFunction*[OutputRecord]

Apply a function to the trigger arguments.

Parameters **fun** (Callable[..., *redgrease.typing.OutputRecord*]) – The function to call with the trigger arguments.

Returns A new "open" gear function generating the results of the function.

Return type redgrease.gears.OpenGearFunction[*redgrease.typing.OutputRecord*]

CHAPTER

SEVEN

OPERATIONS

This section goes through the various operations available to *Open GearFunction* in more detail.

7.1 Map

class redgrease.gears.**Map**(*op: Callable[[InputRecord], OutputRecord]*, ***kwargs*)

The local Map operation performs the one-to-one (1:1) mapping of records.

It requires one mapper function.

op

The mapper function to map on all input records.

Type *redgrease.typing.Mapper*

Instantiate a Map operation.

Parameters **op** (*redgrease.typing.Mapper*) – Function to map on the input records. The function must take one argument as input (input record) and return something as an output (output record).

7.2 FlatMap

class redgrease.gears.**FlatMap**(*op: Callable[[InputRecord], Iterable[OutputRecord]]*, ***kwargs*)

The local FlatMap operation performs one-to-many (1:N) mapping of records.

It requires one expander function that maps a single input record to potentially multiple output records.

FlatMap is nearly identical to the Map operation in purpose and use. Unlike regular mapping, however, when FlatMap returns a sequence / iterator, each element in the sequence is turned into a separate output record.

op

The mapper function to map on all input records.

Type *redgrease.typing.Expander*

Instantiate a FlatMap operation.

Parameters **op** (*redgrease.typing.Expander*) – Function to map on the input records. The function must take one argument as input (input record) and return an iterable as an output (output records).

7.3 ForEach

class redgrease.gears.**ForEach**(*op: Callable[[InputRecord], None]*, ***kwargs*)
The local ForEach operation performs one-to-the-same (1=1) mapping.

It requires one processor function to perform some work that's related to the input record.

Its output record is a copy of the input, which means anything the callback returns is discarded.

Parameters **op** (*redgrease.typing.Processor*) – Function to run on the input records.

Instantiate a ForEach operation.

Parameters **op** (*redgrease.typing.Processor*) – Function to run on each of the input records. The function must take one argument as input (input record) and should not return anything.

7.4 Filter

class redgrease.gears.**Filter**(*op: Callable[[InputRecord], bool]*, ***kwargs*)
The local Filter operation performs one-to-zero-or-one (1:(0|1)) filtering of records.

It requires a filterer function.

An input record that yields a falsehood will be discarded and only truthful ones will be output.

Parameters **op** (*redgrease.typing.Filterer*) – Predicate function to run on the input records.

Instantiate a Filter operation.

Parameters **op** (*redgrease.typing.Filterer*) – Function to apply on the input records, to decide which ones to keep. The function must take one argument as input (input record) and return a bool. The input records evaluated to `True` will be kept as output records.

7.5 Accumulate

class redgrease.gears.**Accumulate**(*op: Callable[[T, InputRecord], T]*, ***kwargs*)
The local Accumulate operation performs many-to-one mapping (N:1) of records.

It requires one accumulator function.

Once input records are exhausted its output is a single record consisting of the accumulator's value.

Parameters **op** (*redgrease.typing.Accumulator*) – Accumulation function to run on the input records.

Instantiate an Accumulate operation.

Parameters **op** (*redgrease.typing.Accumulator*) – Function to to apply on the input records. The function must take two arguments as input:

- the input record, and
- An accumulator value.

It should aggregate the input record into the accumulator variable, which stores the state between the function's invocations. The function must return the accumulator's updated value.

7.6 LocalGroupBy

class redgrease.gears.**LocalGroupBy**(*extractor: Callable[[InputRecord], Key]*, *reducer: Callable[[Key, T, InputRecord], T]*, ***kwargs*)

The local LocalGroupBy operation performs many-to-less mapping (N:M) of records.

The operation requires two functions, an extractor and a reducer.

The output records consist of the grouping key and its respective reduce value.

extractor
Function that extracts the key to group by from input records.

Type *redgrease.typing.Extractor*

reducer
Function that reduces the records in each group to an output record.

Type *redgrease.typing.Reducer*

Instantiate a LocalGroupBy operator.

Parameters

- **extractor** (*redgrease.typing.Extractor*) – Function to apply on the input records, to extact the grouping key. The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key.
- **reducer** (*redgrease.typing.Reducer*) – Function to apply on the records of each group, to reduce to a single value (per group). The function must take (a) a key, (b) an input record and (c) a variable that's called an accumulator. It performs similarly to the accumulator callback, with the difference being that it maintains an accumulator per reduced key / group.

7.7 Limit

class redgrease.gears.**Limit**(*length: int*, *start: int = 0*, ***kwargs*)

The local Limit operation limits the number of records.

It accepts two numeric arguments: a starting position in the input records "array" and a maximal number of output records.

start
Starting index (0-based) of the input record to start from

Type int

length
The maximum number of records to let through.

Type int

Instantiate a Limit operation

Parameters

- **length** (*int*) – The maximum number of records.
- **start** (*int, optional*) – The index of the first input record. Defaults to 0.

7.8 Collect

class redgrease.gears.**Collect**(***kwargs*)

The global Collect operation collects the result records from all of the shards to the originating one.

Instantiate a Collect operation.

7.9 Repartition

class redgrease.gears.**Repartition**(*extractor: Callable[[InputRecord], Key]*, ***kwargs*)

The global Repartition operation repartitions the records by them shuffling between shards.

It accepts a single key extractor function. The extracted key is used for computing the record's new placement in the cluster (i.e. hash slot). The operation then moves the record from its original shard to the new one.

Attributes:

extractor (redgrease.typing.Extractor): A function deciding the destination shard of an input record.

Instantiate a Repartition operation

Parameters **extractor** (*redgrease.typing.Extractor*) – Function that takes a record and calculates a key that is used to determine the hash slot, and consequently the shard, that the record should migrate to to. The function must take one argument as input (input record) and return a string (key). The hash slot, and consequently the destination shard, is determined by hthe value of the key.

7.10 Aggregate

class redgrease.gears.**Aggregate**(*zero: Any*, *seqOp: Callable[[T, InputRecord], T]*, *combOp: Callable[[T, InputRecord], T]*, ***kwargs*)

The Aggregate operation performs many-to-one mapping (N:1) of records.

Aggregate provides an alternative to the local accumulate operation as it takes the partitioning of data into consideration. Furthermore, because records are aggregated locally before collection, its performance is usually superior.

It requires a zero value and two accumulator functions for computing the local and global aggregates.

The operation is made of these steps:

1. The local accumulator is executed locally and initialized with the zero value.
2. A global collect moves all records to the originating engine.
3. The global accumulator is executed locally by the originating engine.

Its output is a single record consisting of the accumulator's global value.

zero
The initial / zero value for the accumulator variable.

Type Any

seqOp
A local accumulator function, applied locally on each shard.

Type *redgrease.typing.Accumulator*

combOp
A global accumulator function, applied on the results of the local accumulations.

Type *redgrease.typing.Accumulator*

Instantiates an Aggregate operation.

Parameters

- **zero** (*Any*) – The initial / zero value of the accumulator variable.
- **seqOp** (*redgrease.typing.Accumulator*) – A function to be applied on each of the input records, locally per shard. It must take two parameters: - an accumulator value, from previous calls - an input record The function aggregates the input into the accumulator variable, which stores the state between the function's invocations. The function must return the accumulator's updated value.
- **combOp** (*redgrease.typing.Accumulator*) – A function to be applied on each of the aggregated results of the local aggregation (i.e. the output of *seqOp*). It must take two parameters: - an accumulator value, from previous calls - an input record The function aggregates the input into the accumulator variable, which stores the state between the function's invocations. The function must return the accumulator's updated value.

7.11 AggregateBy

class redgrease.gears.**AggregateBy**(*extractor: Callable[[InputRecord], Key]*, *zero: Any*, *seqOp: Callable[[Key, T, InputRecord], T]*, *combOp: Callable[[Key, T, InputRecord], T]*, ***kwargs*)

AggregateBy operation performs many-to-less mapping (N:M) of records.

It is similar to the Aggregate operation but aggregates per key. It requires a an extractor callback, a zero value and two reducers callbacks for computing the local and global aggregates.

The operation is made of these steps:

1. Extraction of the groups using extractor.
2. The local reducer is executed locally and initialized with the zero value.
3. A global repartition operation that uses the extractor.
4. **The global reducer is executed on each shard once it is repartitioned with its** relevant keys.

Output list of records, one for each key. The output records consist of the grouping key and its respective reducer's value.

extractor
Function that extracts the key to group by from input records.

Type *redgrease.typing.Extractor*

zero
The initial / zero value for the accumulator variable.

Type Any

seqOp
A local accumulator function, applied locally on each shard.

Type *redgrease.typing.Accumulator*

combOp
A global accumulator function, applied on the results of the local accumulations.

Type *redgrease.typing.Accumulator*

Instantiate an AggregateBy operation.

Parameters

- **extractor** (*redgrease.typing.Extractor*) – Function to apply on the input records, to extact the grouping key. The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key.
- **zero** (*Any*) – The initial / zero value of the accumulator variable.
- **seqOp** (*redgrease.typing.Accumulator*) – A function to be applied on each of the input records, locally per shard and group. It must take two parameters: - an accumulator value, from previous calls - an input record The function aggregates the input into the accumulator variable, which stores the state between the function's invocations. The function must return the accumulator's updated value.
- **combOp** (*redgrease.typing.Accumulator*) – A function to be applied on each of the aggregated results of the local aggregation (i.e. the output of *seqOp*). It must take two parameters: - an accumulator value, from previous calls - an input record The function aggregates the input into the accumulator variable, which stores the state between the function's invocations. The function must return the accumulator's updated value.

7.12 GroupBy

class redgrease.gears.**GroupBy**(*extractor: Callable[[InputRecord], Key]*, *reducer: Callable[[Key, T, InputRecord], T]*, ***kwargs*)

GroupBy * operation performs a many-to-less (N:M) grouping of records. It is similar to AggregateBy but uses only a global reducer. It can be used in cases where locally reducing the data isn't possible.

The operation requires two functions; an extractor a reducer.

The operation is made of these steps:

```
1. A global repartition operation that uses the extractor.
2. The reducer is locally invoked.
```

Output is a locally-reduced list of records, one for each key. The output records consist of the grouping key and its respective accumulator's value.

extractor
Function that extracts the key to group by from input records.

Type *redgrease.typing.Extractor*

reducer
Function that reduces the records of each group to a value

Type *redgrease.typing.Reducer*

Instantiate a GroupBy operation.

Parameters

- **extractor** (*redgrease.typing.Extractor*) – Function to apply on the input records, to extact the grouping key. The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key.

- **reducer** (*redgrease.typing.Reducer*) – Function to apply on the records of each group, to reduce to a single value (per group). The function must take (a) a key, (b) an input record and (c) a variable that's called an accumulator. It performs similarly to the accumulator callback, with the difference being that it maintains an accumulator per reduced key / group.

7.13 BatchGroupBy

class redgrease.gears.**BatchGroupBy**(*extractor: Callable[[InputRecord], Key], reducer: Callable[[Key, Iterable[InputRecord]], OutputRecord], **kwargs*)

BatchGroupBy operation performs a many-to-less (N:M) grouping of records.

> Instead of using BatchGroupBy, prefer using the GroupBy operation as it is more efficient and performant. Only use BatchGroupBy when the reducer's logic requires the full list of records for each input key.

The operation requires two functions; an extractor a batch reducer.

The operation is made of these steps:

```
1. A global repartition operation that uses the extractor
2. A local localgroupby operation that uses the batch reducer
```

Once finished, the operation locally outputs a record for each key and its respective accumulator value.

> Using this operation may cause a substantial increase in memory usage during runtime.

extractor
: Function that extracts the key to group by from input records.

  Type *redgrease.typing.Extractor*

reducer
: Function that reduces the records of each group to a value

  Type *redgrease.typing.Reducer*

Instantiate a BatchGroupBy operation.

Parameters

- **extractor** (*redgrease.typing.Extractor*) – Function to apply on the input records, to extact the grouping key. The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key.
- **reducer** (*redgrease.typing.Reducer*) – Function to apply on the records of each group, to reduce to a single value (per group). The function must take (a) a key, (b) an input record and (c) a variable that's called an accumulator. It performs similarly to the accumulator callback, with the difference being that it maintains an accumulator per reduced key / group.

7.14 Sort

class redgrease.gears.**Sort**(*reverse: bool = True*, ***kwargs*)

Sort operation sorts the records.

It allows to control the sort order.

The operation is made of the following steps:

```
1. A global aggregate operation collects and combines all records.
2. A local sort is performed on the list.
3. The list is flatmapped to records.
```

Using this operation may cause an increase in memory usage during runtime due to the list being copied during the sorting operation.

reverse

Defines if the sorting order is descending (`True`) or ascending (`False`).

Type bool

Instantiate a Sort operation.

Parameters **reverse** (*bool, optional*) – Sort in descending order (higher to lower). Defaults to `True`.

7.15 Distinct

class redgrease.gears.**Distinct**(***kwargs*)

The Distinct operation returns distinct records.

It requires no arguments.

The operation is made of the following steps:

```
1. A aggregate operation locally reduces the records to sets that are then
    collected and unionized globally.
2. A local flatmap operation turns the set into records.
```

Instantiate a Distinct operation.

7.16 Count

class redgrease.gears.**Count**(***kwargs*)

The Count operation counts the number of input records.

It requires no arguments.

The operation is made of an aggregate operation that uses local counting and global summing accumulators.

Instantiate a Count operation.

7.17 CountBy

class redgrease.gears.**CountBy**(*extractor: Callable[[InputRecord], Key]*, ***kwargs*)

The CountBy operation counts the records grouped by key.

It requires a single extractor function.

The operation is made of an aggregateby operation that uses local counting and global summing accumulators.

extractor

Function that extracts the key to group by from input records.

Type *redgrease.typing.Extractor*

Instantiate a CountBy operation.

Parameters **extractor** (*redgrease.typing.Extractor*) – Function to apply on the input records, to extact the grouping key. The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key.

7.18 Avg

class redgrease.gears.**Avg**(*extractor: Callable[[InputRecord], Key]*, ***kwargs*)

The Avg operation returns the arithmetic average of records.

It has an optional value extractor function.

The operation is made of the following steps:

```
1. A aggregate operation locally reduces the records to tuples of sum and count
   that are globally combined.
2. A local map operation calculates the average from the global tuple.
```

extractor

Function that extracts the key to group by from input records.

Type *redgrease.typing.Extractor*

Instantiate an Avg operation.

Parameters **extractor** (*redgrease.typing.Extractor*) – Function to apply on the input records, to extact the grouping key. The function must take one argument as input (input record) and return a string (key). The groups are defined by the value of the key.

CHAPTER

EIGHT

ACTIONS

Actions closes open GearFunctions, and indicates the running mode of the function, as follows:

Action	Execution Mode
Run	"batch-mode"
Register	"event-mode"

8.1 Run

class redgrease.gears.**Run**(*arg: Optional[str] = None*, *convertToStr: bool = True*, *collect: bool = True*, ***kwargs*)

Run action

The Run action runs a Gear function as a batch. The function is executed once and exits once the data is exhausted by its reader.

The Run action can only be the last operation of any GearFunction, and it effectivesly 'closes' it to further operations.

arg

Argument that's passed to the reader, overriding its defaultArg. It means the following:

- A glob-like pattern for the KeysReader and KeysOnlyReader readers.
- A key name for the StreamReader reader.
- A Python generator for the PythonReader reader.

Type str, optional

convertToStr

When *True*, adds a map operation to the flow's end that stringifies records.

Type bool

collect

When *True*, adds a collect operation to flow's end.

Type bool

Instantiate a Run action

Parameters

- **arg** (*Optional[str], optional*) – Optional argument that's passed to the reader, overriding its defaultArg. It means the following:

  - A glob-like pattern for the KeysReader and KeysOnlyReader readers.
  - A key name for the StreamReader reader.
  - A Python generator for the PythonReader reader.

  Defaults to `None`.

- **convertToStr** (*bool, optional*) – When *True*, adds a map operation to the flow's end that stringifies records. Defaults to `True`.
- **collect** (*bool, optional*) – When *True*, adds a collect operation to flow's end. Defaults to `True`.

8.2 Register

class redgrease.gears.**Register**(*prefix: str = '*', convertToStr: bool = True, collect: bool = True, mode: str = 'async', onRegistered: Optional[Callable[], None]] = None, **kwargs*)

Register action

The Register action registers a function as an event handler. The function is executed each time an event arrives. Each time it is executed, the function operates on the event's data and once done it is suspended until its future invocations by new events.

The Register action can only be the last operation of any GearFunction, and it effectivesly 'closes' it to further operations.

prefix
: Key prefix pattern to match on. Not relevant for 'CommandReader' readers (see 'trigger').

  Type str

convertToStr
: When *True*, adds a map operation to the flow's end that stringifies records.

  Type bool

collect
: When *True*, adds a collect operation to flow's end.

  Type bool

mode
: The execution mode of the triggered function.

  Type str

onRegistered
: A function callback that's called on each shard upon function registration.s

  Type Callable

Instantiate a Register action

Parameters

- **prefix** (*str, optional*) – Key prefix pattern to match on. Not relevant for 'CommandReader' readers (see 'trigger'). Defaults to '*'.
- **convertToStr** (*bool, optional*) – When `True` adds a map operation to the flow's end that stringifies records. Defaults to `True`.

- **collect** (*bool, optional*) – When `True` adds a collect operation to flow's end. Defaults to `False`.
- **mode** (*str, optional*) – The execution mode of the function. Can be one of:

  - ``"async"``:

    Execution will be asynchronous across the entire cluster.

  - "async_local":

    Execution will be asynchronous and restricted to the handling shard.

  - "sync":

    Execution will be synchronous and local

  Defaults to *redgrease.TriggerMode.Async* ("async")

- **onRegistered** (*Registrator, optional*) – A function callback that's called on each shard upon function registration. It is a good place to initialize non-serializeable objects such as network connections. Defaults to `None`.

CHAPTER

NINE

OPERATION CALLBACK TYPES

This section runs through the various type signatures that the function callbacks used in the operations must follow.

9.1 Registrator

redgrease.typing.**Registrator**

"Type definition for Registrator functions.

I.e. callback functions that may be called on each shard upon function registration. Such functions provide a good place to initialize non-serializable objects such as network connections.

An function of Registrator type shoud take no arguments, nor return any value.

alias of Callable[[], None]

9.2 Extractor

redgrease.typing.**Extractor**

Type definition for Extractor functions.

Extractor functions are used in the following operations:

- *LocalGroupBy*
- *Repartition*
- *AggregateBy*
- *GroupBy*
- *BatchGroupBy*
- *CountBy*
- *Avg*

Extractor functions extracts or calculates the value that should be used as (grouping) key, from an input record of the operation.

Parameters (InputRecord) - A single input-record, of the same type as the

operations' input type.

Returns A any 'Hashable' value.

Return type Key

Example - Count users per supervisor:

```
# Function of "Extractor" type
# Extracts the "supervisor" for a user,
# If the user has no supervisor, then the user is considered its own supervisor.
def supervisor(user)
    return user.get("supervisor", user["id"])

KeysReader("user:*").values().countby(supervisor).run()
```

alias of Callable[[InputRecord], Key]

9.3 Mapper

redgrease.typing.**Mapper**

Type definition for Mapper functions.

Mapper functions are used in the following operations:

- *Map*

Mapper functions transforms a value from the operations input to some new value.

Parameters (InputRecord) - A single input-record, of the same type as the

operations' input type.

Returns A any value.

Return type OutputRecord

alias of Callable[[InputRecord], OutputRecord]

9.4 Expander

redgrease.typing.**Expander**

Type definition forExpander functions.

Expander functions are used in the following operations:

- *FlatMap*

Expander functions transforms a value from the operations input into several new values.

Parameters (InputRecord) - A single input-record, of the same type as the

operations' input type.

Returns An iterable sequence of values, for example a list, each of which becomes an input to the next operation.

Return type Iterable[OutputRecord]

alias of Callable[[InputRecord], Iterable[OutputRecord]]

9.5 Processor

redgrease.typing.**Processor**
Type definition forProcessor functions.

Processor functions are used in the following operations:

- *ForEach*

Processor functions performs some side effect using a value from the operations input.

Parameters (InputRecord) - A single input-record, of the same type as the

operations' input type.

Returns Nothing.

Return type None

alias of Callable[[InputRecord], None]

9.6 Filterer

redgrease.typing.**Filterer**
Type definition forFilterer functions.

Filterer functions are used in the following operations:

- *Filter*

Filter functions evaluates a value from the operations input to either `True` or `False`.

Parameters (InputRecord) - A single input-record, of the same type as the

operations' input type.

Returns Either `True` or `False`.

Return type bool

alias of Callable[[InputRecord], bool]

9.7 Accumulator

redgrease.typing.**Accumulator**
Type definition forAccumulator functions.

Accumulator functions are used in the following operations:

- *Accumulate*
- *Aggregate*

Accumulator functions takes a variable that's also called an accumulator, as well as an input record. It aggregates inputs into the accumulator variable, which stores the state between the function's invocations. The function must return the accumulator's updated value after each call.

Parameters

- (T) - An accumulator value.

- (InputRecord) - A single input-record, of the same type as the operations' input

type.

Returns The updated accumulator value.

Return type T

alias of Callable[[T, InputRecord], T]

9.8 Reducer

`redgrease.typing.`**`Reducer`**

Type definition forReducer functions.

Reducer functions are used in the following operations:

- *LocalGroupBy*
- *AggregateBy*
- *GroupBy*

Reducer functions receives a key, a variable that's called an accumulator and an an input. It performs similarly to the `redgrease.typing.Accumulator callback`, with the difference being that it maintains an accumulator per reduced key.

Parameters

- (Key) - A key value for the group.
- (T) - An accumulator value.
- (InputRecord) - A single input-record, of the same type as the operations' input

type.

Returns The updated accumulator value.

Return type T

alias of Callable[[Key, T, InputRecord], T]

9.9 BatchReducer

`redgrease.typing.`**`BatchReducer`**

Type definition forBatchReducer functions.

BatchReducer functions are used in the following operations:

- *BatchGroupBy*

BatchReducer functions receives a key and a list of input records. It performs similarly to the `redgrease.typing.Reducer callback`, with the difference being that it is input with a list of records instead of a single one. It is expected to return an accumulator value for these records

Parameters

- (Key) - A key value for the group.

- (Iterable[InputRecord]) - A collection of input-record, of the same type as the

operations' input type.

Returns A reduced output value.

Return type OutputRecord

alias of Callable[[Key, Iterable[InputRecord]], OutputRecord]

CHAPTER

TEN

SERVERSIDE REDIS COMMANDS

With the RedGrease *runtime package installed on the RedGrease server*, you can execute Redis commands to the local shards using "serverside commands" instead of using `redgrease.runtime.execute()` inside your Gear functions.

Serverside Redis Commands, behaves almost identical to a normal Redis client, except that you do not have to instantiate it.

Inside any Gear function, you can simply invoke Redis commands just as you would in your client using `redgrease.cmd`:

Example:

```
redgrease.cmd.set("Foo", "Bar")
```

Warning: Serverside Redis Commands have the following limitations:

A. **Only executes commands against the local shard.** This is also the case for both the `redgrease.runtime.execute()` function as well as the the RedisGears default builtin *execute()* function too.

B. **Blocking commands, such as "BRPOP" or "BLPOP" ect, are not supported.** This is also the case for both the `redgrease.runtime.execute()` function as well as the the RedisGears default builtin *execute()* function too.

Courtesy of :  Pte. Ltd.

CHAPTER

ELEVEN

SYNTACTIC SUGAR

Various minor things to make life easier, prettier, more concise and/or less error prone.

11.1 Command Function Decorator

The *@redgrease.command* decorator can be put on functions to immediately turn them into *CommandReader* GearFunctions.

The decorator takes all the same parameters as the *CommandReader.register()*, but the `trigger` argument is optional, and the name of the function is used as default.

The decorator also takes a boolean keyword argument `replace` which indicates what should be done if a function with the same trigger has already been registered, as follows:

- `None` - Default - A `redgrease.exceptions.DuplicateTriggerError` error is raised.
- `True` - The registered function is replaced.
- `False` - Nothing. No error is raised, but the existing registered function remains.

The decorator also takes the keyword `on`, which if provided with a Redis Gears client connection, attempts to register the function as a command reader on the server.

The GearFunction can be triggered by simply locally calling the decorated function, just as if it was a local function. Under the hood, however, this will send the trigger and any passed arguments to the server, which in turn runs the registered CommandReader function on the Redis shards, and then retuns the function results to the caller.

Listing 1: examples/cache_get_command.py

```
2
3  import redgrease
4
5  # Bind / register the function on some Redis instance.
6  r = redgrease.RedisGears()
7
8
9  # CommandReader Decorator
10 # The `command` decorator tunrs the function to a CommandReader,
11 # registerered on the Redis Gears sever if using the `on` argument
12 @redgrease.command(on=r, requirements=["requests"], replace=False)
13 def cache_get(url):
14     import requests
15
16     # Check if the url is already in the cache,
17     # And if so, simply return the cached result.
```

(continues on next page)

(continued from previous page)

```
    if redgrease.cmd.exists(url):
        return bytes(redgrease.cmd.get(url))

    # Otherwise fetch the url.
    response = requests.get(url)

    # Return nothing if request fails
    if response.status_code != 200:
        return bytes()

    # If ok, set the cache data and return.
    response_data = bytes(response.content)
    redgrease.cmd.set(url, response_data)

    return response_data


```

Note: The `on` argument is actually optional, but in that case the decorated function cannot be called locally as described above.

Instead the function name becomes a *"Closed" CommandReader GearFunction*, which can be turned into a function as per above again, by calling the *`on()`* method:

```
import redgrease

# ``on`` argument **not** provided => ``foo`` becomes a Closed GearFunction
@redgrease.command()
def foo(arg1, arg2):
    redgrease.cmd.log(f"Pretending to do something with {arg1} and {arg2}")


r = redgrease.RedisGears()

# Call ``on`` on foo with a Gears client
# => result is a local triggering function
do_foo = foo.on(r)

# Call / trigger the function
do_foo("this", "that")
```

redgrease.**command**(*trigger: Optional[str] = None*, *prefix: str = '*'*, *collect: bool = True*, *mode: str = 'async'*, *onRegistered: Optional[Callable[], None]] = None*, ***kargs*) → Callable[[Callable], *redgrease.gears.ClosedGearFunction*]

Decorator for creation of CommandReader + Trigger GearFunctions

Parameters

- **trigger** (*str, optional*) – The trigger string Will be a the function name if not specified (None). The special value ... (Ellipsis) wil give the function a unique trigger. Defaults to None
- **prefix** (*str, optional*) – Register prefix. Same as for the *register* operation. Defaults to "*".
- **collect** (*bool, optional*) – Add a *collect' operation to the end of the function.*

Same as for the `register operation. Defaults to True.

- **mode** (*str, optional*) – The execution mode of the triggered function. Same as for the *register* operation. Defaults to redgrease.sugar.TriggerMode.Async.
- **onRegistered** (*redgrease.typing.Registrator*, optional) – A function callback thats called on each shard upon function registration. It is a good place to initialize non-serializeable objects such as network connections. Same as for the *register* operation. Defaults to None.

Returns A ClosedGearFunction generator.

Return type Callable[[Callable], *ClosedGearFunction*]

11.2 Keywords

Moderately useful symbols that can be used instead of strings for various RedisGears Keywords.

11.2.1 Reader Types

`ReaderType.`**`KeysReader = 'KeysReader'`**
KeysReader

`ReaderType.`**`KeysOnlyReader = 'KeysOnlyReader'`**
KeysOnlyReader

`ReaderType.`**`StreamReader = 'StreamReader'`**
StreamReader

`ReaderType.`**`PythonReader = 'PythonReader'`**
PythonReader

`ReaderType.`**`ShardsIDReader = 'ShardsIDReader'`**
ShardsIDReader

`ReaderType.`**`CommandReader = 'CommandReader'`**
CommandReader

Example:

```
from redgrease import GearsBuilder, ReaderType

gb = GearsBuilder(ReaderType.KeysReader).run()
```

11.2.2 Trigger Modes

`TriggerMode.`**`Async = 'async'`**
Async

`TriggerMode.`**`AsyncLocal = 'async_local'`**
AsyncLocal

`TriggerMode.`**`Sync = 'sync'`**
Sync

Example:

```
from redgrease import GearsBuilder, KeysReader, TriggerMode

fun = KeysReader().register(mode=TriggerMode.AsyncLocal)
```

11.2.3 Key types

`KeyType.String = 'string'`
: String

`KeyType.Hash = 'hash'`
: Hash

`KeyType.List = 'list'`
: List

`KeyType.Set = 'set'`
: Set

`KeyType.ZSet = 'zset'`
: ZSet

`KeyType.Stream = 'stream'`
: Stream

`KeyType.Module = 'module'`
: Module

Example:

```
from redgrease import KeysReader, KeyType

fun = KeysReader().register(keyTypes=[KeyType.List, KeyType.Set, KeyType.ZSet])
```

11.2.4 Failure Policies

`FailurePolicy.Continue = 'continue'`
: Continue

`FailurePolicy.Abort = 'abort'`
: Abort

`FailurePolicy.Retry = 'retry'`
: Retry

Example:

```
from redgrease import StreamReader, FailurePolicy

fun = StreamReader().register(onFailedPolicy=FailurePolicy.Abort)
```

11.2.5 Log Levels

`LogLevel.Debug = 'debug'`
: Debug

`LogLevel.Verbose = 'verbose'`
: Verbose

`LogLevel.Notice = 'notice'`
: Notice

`LogLevel.Warning = 'warning'`
: Warning

Example:

```
from redgrease import KeysOnlyReader, LogLevel

fun = KeysOnlyReader().map(lambda k: log(f"Processing key: {k}", level=LogLevel.
↪Debug).run()
```

CHAPTER

TWELVE

COMMAND LINE TOOL

Sorry, this section is under construction!

redgrease can be invoked from the CLI:

```
redgrease --help
usage: redgrease [-h] [-c PATH] [--index-prefix PREFIX] [-r] [--script-pattern␣
↪PATTERN] [--requirements-pattern PATTERN] [--unblocking-pattern PATTERN] [-i␣
↪PATTERN] [-w [SECONDS]] [-s [SERVER]] [-p PORT] [-l LOG_CONFIG] dir_path [dir_path .
↪..]

Scans one or more directories for Redis Gears scripts, and executes them in a Redis␣
↪Gears instance or cluster. Can optionally run continuously, monitoring and re-
↪loading scripts whenever changes are detected. Args that start with '--' (eg. --
↪index-prefix) can also be set in a config file
(./*.conf or /etc/redgrease/conf.d/*.conf or specified via -c). Config file syntax␣
↪allows: key=value, flag=true, stuff=[a,b,c] (for details, see syntax at https://goo.
↪gl/R74nmi). If an arg is specified in more than one place, then command-line values␣
↪override environment variables which override
config file values which override defaults.

positional arguments:
   dir_path              One or more directories containing Redis Gears scripts to␣
↪watch

   optional arguments:
   -h, --help            show this help message and exit
   -c PATH, --config PATH
                         Config file path [env var: CONFIG_FILE]
   --index-prefix PREFIX
                         Redis key prefix added to the index of monitored/executed␣
↪script files. [env var: INDEX_PREFIX]
   -r, --recursive       Recursively watch subdirectories. [env var: RECURSIVE]
   --script-pattern PATTERN
                         File name pattern (glob-style) that must be matched for␣
↪scripts to be loaded. [env var: SCRIPT_PATTERN]
   --requirements-pattern PATTERN
                         File name pattern (glob-style) that must be matched for␣
↪requirement files to be loaded. [env var: REQUIREMENTS_PATTERN]
   --unblocking-pattern PATTERN
                         Scripts with file paths that match this regular expression,␣
↪will be executed with the 'UNBLOCKING' modifier, i.e. async execution. Note that␣
↪the pattern is a 'search' pattern and not anchored to the start of the path string.␣
↪[env var: UNBLOCKING_PATTERN]
```

(continues on next page)

(continued from previous page)

```
  -i PATTERN, --ignore PATTERN
                        Ignore files matching this pattern. [env var: IGNORE]
  -w [SECONDS], --watch [SECONDS]
                        If set, the directories will be continuously monitored for␣
↪updates/modifications to scripts and requirement files, and automatically loaded/
↪rerun. The flag takes an optional value specifying the duration, in seconds, to␣
↪wait for further updates/modifications to files,
                        before executing. This 'hysteresis' period is to prevent␣
↪malformed scripts to be unnecessarily loaded during coding. If no value is supplied,
↪ the duration is defaulting to 5 seconds. [env var: WATCH]
  -s [SERVER], --server [SERVER]
                        Redis Gears host server IP or hostname. [env var: SERVER]
  -p PORT, --port PORT  Redis Gears host port number [env var: PORT]
  -l LOG_CONFIG, --log-config LOG_CONFIG
                        [env var: LOG_CONFIG]
```

Courtesy of :  Pte. Ltd.

CHAPTER

THIRTEEN

UTILS MODULE

Utility and boilerplate functions, such as parsers, value transformers etc.

class redgrease.utils.**CaseInsensitiveDict**(*data*)

Case insensitive dict implementation. Assumes string keys only. Heavily derived from redis.client **`https://github.com/andymccurdy/redis-py/blob/master/redis/client.py`_**

get(*k*, *default=None*)

Return the value for key if key is in the dictionary, else default.

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

redgrease.utils.**as_is**(*value: T*) → T

Passthrough parser / identity function

Parameters **value** (*T*) – Input value

Returns The value, unmodified.

Return type T

redgrease.utils.**str_if_bytes**(*value: T*) → Union[T, str]

Parses byte values into a string, non-byte values passthrough unchanged Slightly modified from redis.utils, as it is not exported

Parameters **value** (*T*) – Any serialized Redis value

Returns Either a string or the input unchanged

Return type Union[T, str]

redgrease.utils.**safe_str**(*value: Any*) → str

Parse anything to a string

Parameters **value** (*Any*) – Input value

Returns String

Return type str

redgrease.utils.**safe_str_upper**(*value: Any*) → str

Parse anything to an uppercase string

Parameters **value** (*Any*) – Input value

Returns Parsed uppercase string

Return type str

redgrease.utils.**bool_ok**(*value: Any*) → bool
Parse redis response as bool, such that:

"Ok" => True

Anything else => False

Same name as in redis.client but slightly different implementation. Should be better for long non-Ok replies, e.g. images, erroneously passed to it

Parameters **value** (*Any*) – Input value

Returns Parsed boolean

Return type bool

redgrease.utils.**optional**(*constructor: Union[Type[T], Callable[[Any], T]]*) → Union[Type[Optional[T]], Callable[[Any], Optional[T]]]
Create parser that accepts None values, but otherwise behaves like the provided parser.

Parameters **constructor** (*Constructor[T]*) – constructor to apply, unless the value is None.

redgrease.utils.**safe_bool**(*input: Any*) → bool
Parse a bool, slightly more accepting

allowing for literal "True"/"False", integer 0 or1, as well as "Ok" and "yes"/"no" values.

Parameters **input** (*Any*) – Input value

Returns Parsed boolean

Return type bool

redgrease.utils.**to_int_if_bool**(*value: Any*) → Union[int, Any]
Transforms any boolean into an integer As booleans are not natively supported as a separate datatype in Redis

True => 1

False => 0

Parameters **value** (*Union[bool, Any]*) – A boolean value

Returns Integer representation of the bool

Return type Union[int, Any]

redgrease.utils.**to_redis_type**(*value: Any*) → Union[bytes, int, float]
Attempts to serialize a value to a Redis-native type. I.e. either: bytes, int or float It will serialize most primitive types (str, bool, int, float), as well as any complex type that implements __bytes__ method

Parameters **value** (*Any*) – Value to serialize for Redis

Returns A serialized version

Return type RedisType

redgrease.utils.**to_list**(*mapping: Optional[Dict[Key, Val]], key_transform: Callable[[Key], Any] = <function str_if_bytes>, val_transform: Callable[[Val], Any] = <function to_redis_type>*) → List
Flattens a Dict to a list consisting of the of keys and values altertnating. This is useful generating the arguments for Redis commands.

Parameters

- **mapping** (*Dict[Key, Val]*) – The dict to flatten.

- **key_transform** (*Callable[[Key], Any], optional*) – Transformation function for the keys. Defaults to 'str_if_bytes'
- **val_transform** (*Callable[[Val], Any], optional*) – Transformation function for the values. Defaults to 'to_redis_type'.

Returns Flattened list of the transformed keys and values

Return type List

redgrease.utils.**transform**(*value: Any*, *constructor: Union[Type[T], Callable[[Any], T], Dict[Key, Union[Type[T], Callable[[Any], T]]]]*, *key: Optional[Key] = None*) → T

Applies a transformation to a value. The tranformation function could either be passed directly or in a dictionary along with a key to look it up. This is mostly only useful as a helper function for constructors of composite types, where the value may need to be transformed differently depending on the field/key.

Parameters

- **value** (*Any*) – Value to transform
- **constructor** (*Union[Constructor[T], Dict[Any, Constructor[T]]]*) – Transformation function or a dict of transformation functions. If a dict is used, the key argument is used to look up the transformation. If the key is not present in the dict, but there is a 'None'-key mapping to a functsion, this function will be used as a default. Otherwise, the value will be returned untransformed.
- **key** (*Key*) – key to use to look up the appropriate transforpation

Returns Transformed value

Return type T

redgrease.utils.**to_dict**(*items: Iterable*, *keys: Optional[Iterable] = None*, *key_transform: Optional[Union[Type[Key], Callable[[Any], Key], Dict[Any, Union[Type[Key], Callable[[Any], Key]]]]] = None*, *val_transform: Optional[Union[Type[Val], Callable[[Any], Val], Dict[Key, Union[Type[Val], Callable[[Any], Val]]]]] = None*) → Dict[Key, Val]

Folds an iterable of values into a dict. This is useful for parsing Redis' list responseses into a more manageable structure. It can be used on lists of alternating unnamed Key and values, i.e:

```
[key_1, value_1, key_2, value_2, ... ]
```

eg:

```
to_dict(["foo", 42, 13, 37]) == {"foo": 42, 13: 37}

to_dict(["foo", 42, 13, 37], key_transform=str) == {"foo": 42, "13": 37}

to_dict(["foo", 42, 13, 37], val_transform=str) == {"foo": "42", 13: "37"}

to_dict(
    ["foo", 42, 13, 37],
    val_transform={"foo":int, 13:float}
) == {"foo": 42, 13: 37.0}

to_dict(
    ["foo", 42, 13, 37],
    key_transform=str,
    val_transform={"foo":int, "13":float}
) == {"foo": 42, "13": 37.0}
```

Parameters

- **items** (*Iterable*) – Iterable to "fold" into a dict
- **key_transform** (*Union[Constructor[Key], Dict[Any, Constructor[Key]]], optional*) – Transformation function / type / constructor to apply to keys. It can either be a callable, which is then applied to all keys. Altertnatively, it can be a mapping from 'raw' key to a specific transform for that key Defaults to None (No key transformation).
- **val_transform** (*Union[Constructor[Val], Dict[Key, Constructor[Val]]], optional*) – Transformation function / type / constructor to apply to values. It can either be a callable, which is then applied to all values. Altertnatively, it can be a mapping from (transformed) key to a specific transform for the value of that key Defaults to None (No value transformation).

Returns Folded dictionary

Return type Dict[Key, Val]

redgrease.utils.**to_kwargs**(*items: Iterable*) → Dict[str, Any]

Folds an iterable of values into a 'kwargs-compatible' dict. This is useful for constructing objects from Redis' list responseses, by means of an intermediate kwargs dict that can be passed to for example a constructor. It behaves exactly as 'to_dict' but enforces keys to be parsed to strings.

- **Alternating unnamed Key and values, i.e::** [key_1, value_1, key_2, value_2, ...]

eg:

- input: `["foo", 42, 13, 37]`
- output: `{"foo": 42, "13": 37}`

Parameters **items** (*Iterable*) – Iterable to "fold" into a dict

Returns Folded dictionary

Return type Dict[str, Any]

redgrease.utils.**list_parser**(*item_parser: Union[Type[T], Callable[[Any], T]]*) → Callable[[Iterable], List[T]]

Creates a list parser for lists of objects created with a given constructor. E.g:

```
parser = list_parser(bool)
parser(['', 1, None])
```

=> `[False, True, False]`

Parameters **item_parser** (*Constructor[T]*) – The constructor to apply to each element.

Returns Function that takes maps the constructor on to the iterable and returns the result as a list.

Return type Callable[[Iterable[Any]], List[T]]

redgrease.utils.**dict_of**(*constructors: Dict[Key, Union[Type[Any], Callable[[Any], Any]]]*) → Callable[[Iterable, Iterable[Key]], Dict[Key, Any]]

Creates a parser that parses a list of values to a dict, according to a dict of named constructors.

The generated parser takes both the iterable of values to parse, as well as, an equally long, iterable of names/keys to to use to lookup the corresponding parser/constructor in the constructor lookup dict.

The parser for the Nth value is using the parser found by looking up the Nth name/key in the key list in the lookup dict. This key is also used as the key in the resulting dict.

E.g:

```
parser = dict_of({"b":bool, "i":int, "s":str, "f":float})
parser([0,1,0,1], ["b","f","s","i"])
```

=> `{"b":False, "f":1.0, "i":1, "s":"0"}`

Parameters **constructors** (*Dict[str, Constructor[Any]]*) – Dict of named constructors

Returns Dict parser

Return type Callable[.., Dict[str, Any]]

class redgrease.utils.**Record**(*key: str*, *value: Optional[Any] = None*, *type: Optional[str] = None*, *event: Optional[str] = None*, ***kwargs*)

Class representing a Redis Record, as generated by KeysReader.

key
The name of the Redis key.

Type str

value
The value corresponting to the key. *None* if deleted.

Type Any

type
The core Redis type. Either 'string', 'hash', 'list', 'set', 'zset' or 'stream'.

Type str

event
The event that triggered the execution. (*None* if the execution was created via the run function.)

Type str

class redgrease.utils.**StreamRecord**(*key: str*, *id: Optional[str] = None*, *value: Optional[Any] = None*, ***kwargs*)

Class representing a Redis Record, as generated by KeysReader.

key
The name of the Redis key.

Type str

value
The value corresponting to the key. *None* if deleted.

Type Any

type
The core Redis type. Either 'string', 'hash', 'list', 'set', 'zset' or 'stream'.

Type str

event
The event that triggered the execution. (*None* if the execution was created via the run function.)

Type str

redgrease.utils.**record**(*rec: Any*) → redgrease.utils.Record

Create a Record

Parameters **rec** (*Any*) – the value to parse. Either a string (key only) or a dict with at minimum the key *key* present and optionally any of the keys *value*, *type* and/or *event*.

Returns Parsed Record object.

Return type Record

redgrease.utils.**stream_record**(*rec: Any*) → redgrease.utils.StreamRecord
Create a Record

Parameters **rec** (*Any*) – the value to parse. Either a string (key only) or a dict with at minimum the key *key* present and optionally any of the keys *value*, *type* and/or *event*.

Returns Parsed Record object.

Return type Record

redgrease.utils.**compose**(**function: Callable*) → Callable
Compose functions. I.e:

```
lambda x: f(g(x))
```

can be written:

```
compose(f, g)``
```

Parameters ***function** (*Callable*) – Any number of functions to compose together. Output type of function N must be the input type of function N+1.

Returns A composed function with input type same as the firs function, and output type same as the last function.

Return type Callable

redgrease.utils.**dict_filter**(***kwargs*) → Callable[[Dict[str, Any]], bool]
Create a dictionary matching predicate function.

This function takes any number of keyword arguments, and returns a predicate function that takes a single *dict* as argument and returns a *bool*.

The predicate function returns *True* if, and only if, for all keyword arguments:

1. The keyword argument name is a key in the dict, and
2. Depending on the value, V, of the keyword argument, either:
   - **V is Container type (excluding `str`)** The dict value for the key is present in V
   - **V is a Type (e.g. `bool`)** The dict value for the key is of type V.
   - **V is any value, except `...` (Ellipsis)** The dict value for the key equals V.
   - **V is `...` (Ellipsis)** The dict value can be any value.

redgrease.utils.**passfun**(*fun: Optional[T] = None*, *default: Optional[T] = None*) → T
Create a Python 'function' object from any 'Callable', or constant.

RedisGears operator callbacks ony accept proper 'function's and not every type of 'Callable', such as for example 'method's (e.g. *redgrease.cmd.incr*) or 'method-desriptor's (e.g. *str.split*), which forces users to write seemingly "unnecessary" lambda-functions to wrap these.

This function ensures that the argument is a proper 'function', and thus will be accepted as RedisGears operator callback (assuming the type signature is correct).

It can also be used to create ‘constant’-functions, if passing a non-callable, or to create the ‘identity-function`, if called with no arguments.

Parameters

- **fun** (*Optional[T], optional*) – Callable to turn into a ‘function` Alternatively a constant, to use to create a constant function, i.e. a function that alway return the same thing, regarding of input. If None, and no default, the ‘identity-function’ is returned. Defaults to None.
- **default** (*Optional[T], optional*) – Default Callable to use as fallback if the ‘fun’ argument isn’t a callable. Defaults to None.

Returns [description]

Return type T

CHAPTER

FOURTEEN

TYPING MODULE

Type variables, Type aliases and Protocol Types.

redgrease.typing.**InputRecord**
: Type variable for the input value of a GearFunction step / operation.

  alias of TypeVar('InputRecord', contravariant=True)

redgrease.typing.**OutputRecord**
: Type variable for the output value of a GearFunction step / operation.

  alias of TypeVar('OutputRecord', covariant=True)

redgrease.typing.**Key**
: Type variable for a Keys used in extractor functions in GroupBy operations and similar.

  alias of TypeVar('Key', contravariant=True)

redgrease.typing.**Val**
: Type variable for intermediate values inside a step / operation.

  alias of TypeVar('Val')

redgrease.typing.**Constructor**
: Joint type for primitive Types and (single argument) object constructors

  alias of Union[Type[T], Callable[[Any], T]]

redgrease.typing.**RedisType**
: Types native to Redis

  alias of Union[bytes, int, float]

redgrease.typing.**SafeType**
: Types that Redis happily accepts as input without any manipulation.

  alias of Union[bytes, int, float, str]

redgrease.typing.**SupportedType**
: Types that RedGrease supports

  alias of Union[bool, str, bytes, int, float]

redgrease.typing.**RedisKey**
: Accepted types for Redis Keys

  alias of Union[str, bytes]

redgrease.typing.**Record**
: The type of a record from KeysReader and others.

  alias of Dict

redgrease.typing.**Registrator**
"Type definition for Registrator functions.

I.e. callback functions that may be called on each shard upon function registration. Such functions provide a good place to initialize non-serializable objects such as network connections.

An function of Registrator type shoud take no arguments, nor return any value.

alias of Callable[[], None]

redgrease.typing.**Extractor**
Type definition for Extractor functions.

Extractor functions are used in the following operations:

- *LocalGroupBy*
- *Repartition*
- *AggregateBy*
- *GroupBy*
- *BatchGroupBy*
- *CountBy*
- *Avg*

Extractor functions extracts or calculates the value that should be used as (grouping) key, from an input record of the operation.

Parameters (InputRecord) - A single input-record, of the same type as the

operations' input type.

Returns A any 'Hashable' value.

Return type Key

Example - Count users per supervisor:

```
# Function of "Extractor" type
# Extracts the "supervisor" for a user,
# If the user has no supervisor, then the user is considered its own supervisor.
def supervisor(user)
    return user.get("supervisor", user["id"])

KeysReader("user:*").values().countby(supervisor).run()
```

alias of Callable[[InputRecord], Key]

redgrease.typing.**Mapper**
Type definition for Mapper functions.

Mapper functions are used in the following operations:

- *Map*

Mapper functions transforms a value from the operations input to some new value.

Parameters (InputRecord) - A single input-record, of the same type as the

operations' input type.

Returns A any value.

Return type OutputRecord

alias of Callable[[InputRecord], OutputRecord]

redgrease.typing.**Expander**
Type definition forExpander functions.

Expander functions are used in the following operations:

- *FlatMap*

Expander functions transforms a value from the operations input into several new values.

Parameters (InputRecord) - A single input-record, of the same type as the

operations' input type.

Returns An iterable sequence of values, for example a list, each of which becomes an input to the next operation.

Return type Iterable[OutputRecord]

alias of Callable[[InputRecord], Iterable[OutputRecord]]

redgrease.typing.**Processor**
Type definition forProcessor functions.

Processor functions are used in the following operations:

- *ForEach*

Processor functions performs some side effect using a value from the operations input.

Parameters (InputRecord) - A single input-record, of the same type as the

operations' input type.

Returns Nothing.

Return type None

alias of Callable[[InputRecord], None]

redgrease.typing.**Filterer**
Type definition forFilterer functions.

Filterer functions are used in the following operations:

- *Filter*

Filter functions evaluates a value from the operations input to either `True` or `False`.

Parameters (InputRecord) - A single input-record, of the same type as the

operations' input type.

Returns Either `True` or `False`.

Return type bool

alias of Callable[[InputRecord], bool]

redgrease.typing.**Accumulator**
Type definition forAccumulator functions.

Accumulator functions are used in the following operations:

- *Accumulate*
- *Aggregate*

Accumulator functions takes a variable that's also called an accumulator, as well as an input record. It aggregates inputs into the accumulator variable, which stores the state between the function's invocations. The function must return the accumulator's updated value after each call.

Parameters

- (T) - An accumulator value.
- (InputRecord) - A single input-record, of the same type as the operations' input

type.

Returns The updated accumulator value.

Return type T

alias of Callable[[T, InputRecord], T]

`redgrease.typing.`**`Reducer`**

Type definition forReducer functions.

Reducer functions are used in the following operations:

- *LocalGroupBy*
- *AggregateBy*
- *GroupBy*

Reducer functions receives a key, a variable that's called an accumulator and an an input. It performs similarly to the `redgrease.typing.Accumulator callback`, with the difference being that it maintains an accumulator per reduced key.

Parameters

- (Key) - A key value for the group.
- (T) - An accumulator value.
- (InputRecord) - A single input-record, of the same type as the operations' input

type.

Returns The updated accumulator value.

Return type T

alias of Callable[[Key, T, InputRecord], T]

`redgrease.typing.`**`BatchReducer`**

Type definition forBatchReducer functions.

BatchReducer functions are used in the following operations:

- *BatchGroupBy*

BatchReducer functions receives a key and a list of input records. It performs similarly to the `redgrease.typing.Reducer callback`, with the difference being that it is input with a list of records instead of a single one. It is expected to return an accumulator value for these records

Parameters

- (Key) - A key value for the group.
- (Iterable[InputRecord]) - A collection of input-record, of the same type as the

operations' input type.

Returns A reduced output value.

Return type OutputRecord

alias of Callable[[Key, Iterable[InputRecord]], OutputRecord]

CHAPTER

FIFTEEN

DATA MODULE

Datatypes and parsers for the various structures, specific to Redis Gears.

These datatypes are returned from various redgrease functions, merely for the purpose of providing more convenient structure, typing and documentation compared to the native 'list-based' structures.

They are generally not intended to be instantiated by end-users.

class redgrease.data.**ExecID**(*shard_id: str = '00000000000000000000000000000000000000000000'*, *sequence: int = 0*)

Execution ID

shard_id

Shard Identifier

Type str

sequence

Sequence number

Type in

Method generated by attrs for class ExecID.

static parse(*value: Union[str, bytes]*) → redgrease.data.ExecID

Parses a string or bytes representation into a *redgrease.data.ExecID*

Returns The parsed ExecID

Return type redgrease.data.ExecID

Raises **ValueError** – If the the value cannot be parsed.

class redgrease.data.**ExecutionResult**(*value: T*, *errors: Optional[List] = None*)

Common class for all types of execution results. Generic / Polymorphic on the result type (T) of the Gears function.

Redis Gears specifies a few different commands for getting the results of a function execution (*pyexecute*, *getresults*, *getresultsblocking* and *trigger*), each potentially having more than one different possible value type, depending on context.

In addition, while most gears functions result in collection of values, some functions (notably those ending with a *count* or *avg* operation) semantically always have scalar results, but are still natively returned as a list.

redgrease.data.ExecutionResult is a unified result type for all scenarios, aiming at providing as intuitive API experience as possible.

It is generic and walks and quacks just like the main result type (T) it wraps. With some notable exceptions:

- It has an additional property *errors* containing any accumulated errors
- Scalar results, behaves like scalars, but **also** like a single element list.

This behavior isn't always perfect but gives for the most part an intuitive api experience.

If the behavior in some situations are confusing, the raw wrapped value can be accessed through the *value* property.

property value
Gets the raw result value of the Gears function execution.

Returns The result value / values

Return type T

redgrease.data.**parse_execute_response**(*response*) → redgrease.data.ExecutionResult
Parses raw responses from *pyexecute*, *getresults* and *getresultsblocking* into a *redgrease.data.ExecuteResponse* object.

Parameters **response** (*Any*) – The raw gears function response. This is most commonly a tuple of a list with the actual results and a list of errors List[List[Union[T, Any]]].

For some scenarios the response may take other forms, like a simple *Ok* (e.g. in the absence of a closing *run()* operation) or an execution ID (e.g. for non-blocking executions).

Returns A parsed execution response

Return type ExecutionResult[T]

redgrease.data.**parse_trigger_response**(*response*) → redgrease.data.ExecutionResult
Parses raw responses from *trigger* into a *redgrease.data.ExecuteResponse* object.

Parameters

- **response** (*Any*) – The gears function response. This is a tuple of a list with the actual results and a list of errors List[List[Union[T, Any]]].
- **pickled** (*bool, optional*) – Indicates if the response is pickled and need to be unpickled. Defaults to False.

Returns A parsed execution response

Return type ExecutionResult[T]

class redgrease.data.**ExecutionStatus**(*value*)
Representation of the various states an execution could be in.

created = b'created'
Created - The execution has been created.

running = b'running'
Running - The execution is running.

done = b'done'
Done - Thee execution is done.

aborted = b'aborted'
Aborted - The execution has been aborted.

pending_cluster = b'pending_cluster'
Pending Cluster - Initiator is waiting for all workers to finish.

pending_run = b'pending_run'
Pending Run - Worker is pending ok from initiator to execute.

pending_receive = b'pending_receive'
Pending Receive - Initiator is pending acknowledgement from workers on receiving execution.

pending_termination = b'pending_termination'
Pending Termination - Worker is pending termination messaging from initiator

class redgrease.data.**ExecLocality**(*value*)
Locality of execution: Shard or Cluster

class redgrease.data.**RedisObject**
Base class for many of the more complex Redis Gears configuration values

classmethod from_redis(*params*) → redgrease.data.RedisObject
Default parser

Assumes the object is seralized as a list of alternating attribute names and values.

Note: This method should not be invoked directly on the 'RedisObject' base class. It should be only be invoked on subclasses of RedisObjects.

Returns Returns the RedisObject subclass if, and only if, its constructor argument names and value types exactly match the names and values in the input list.

Return type RedisObject

Raises **TypeError** – If either the input list contains attributes not defined in the subclass constructor, or if the subclass defines mandatory constructor arguments that are not present in the input list.

redgrease.data.**parse_PD**(*value: Union[str, bytes]*) → Dict
Parses str or bytes to a dict.

Used for the 'PD' field in the 'Registration' type, returned by 'getregistrations'.

Parameters **value** (*Union[str,bytes]*) – Serialized version of a dict.

Returns A dictionary

Return type Dict

class redgrease.data.**ExecutionInfo**(*executionId*, *status*, *registered*)
Return object for *redgrease.client.Redis.dumpexecutions* command.

Method generated by attrs for class ExecutionInfo.

executionId: redgrease.data.ExecID
The execution Id

status: redgrease.data.ExecutionStatus
The status

registered: bool
Indicates whether this is a registered execution

class redgrease.data.**RegData**(*mode*, *numTriggered: int*, *numSuccess: int*, *numFailures: int*, *numAborted: int*, *lastError: str*, *args*, *status=None*)
Object reprenting the values for the *Registration.RegistrationData*, part of the return value of *redgrease.client.dupregistrations* command.

Method generated by attrs for class RegData.

mode: str
Registration mode.

numTriggered: int
A counter of triggered executions.

numSuccess: int
A counter of successful executions.

numFailures: int
A counter of failed executions.

numAborted: int
A conter of aborted executions.

lastError: str
The last error returned.

args: Dict[str, Any]
Reader-specific arguments

status: Optional[bool]
Undocumented status field

class redgrease.data.**Registration**(*id*, *reader*, *desc: str*, *RegistrationData*, *PD*)
Return object for *redgrease.client.Redis.dumpregistrations* command. Contains the information about a function registration.

Method generated by attrs for class Registration.

id: redgrease.data.ExecID
The registration ID.

reader: str
The Reader.

desc: str
The description.

RegistrationData: redgrease.data.RegData
Registration Data, see *RegData*.

PD: Dict[Any, Any]
Private data

class redgrease.data.**ExecutionStep**(*type*, *duration: int*, *name*, *arg*)
Object reprenting a 'step' in the *ExecutionPlan.steps*, attribute of the return value of *redgrease.client.getexecution* command.

Method generated by attrs for class ExecutionStep.

type: str
Step type.

duration: int
The step's duration in milliseconds (0 when *ProfileExecutions* is disabled)

name: str
Step callback

arg: str
Step argument

class redgrease.data.**ExecutionPlan**(*status*, *shards_received: int*, *shards_completed: int*, *results: int*, *errors: int*, *total_duration: int*, *read_duration: int*, *steps*)
Object representing the execution plan for a given shard in the response from the *redgrease.client.Redis.getexecution* command.

Method generated by attrs for class ExecutionPlan.

status: redgrease.data.ExecutionStatus
The current status of the execution.

shards_received: int
Number of shards that received the execution.

shards_completed: int
Number of shards that completed the execution.

results: int
Count of results returned.

errors: int
Count of the errors raised.

total_duration: int
Total execution duration in milliseconds.

read_duration: int
Reader execution duration in milliseconds.

steps: List[redgrease.data.ExecutionStep]
The steps of the execution plan.

static parse(*res: Iterable[bytes]*) → Dict[str, redgrease.data.ExecutionPlan]
Parse the raw results of *redgrease.client.Redis.getexecution* into a dict that maps from shard identifiers to ExecutionStep objects.

Returns Execution plan mapping.

Return type Dict[str, ExecutionPlan]

class redgrease.data.**ShardInfo**(*id*, *ip*, *port: int*, *unixSocket*, *runid*, *minHslot: int*, *maxHslot: int*, *pendingMessages: int*)
Object representing a shard in the *ClusterInfo.shards* attribute in the response from *redgrease.client.Redis.infocluster* command.

Method generated by attrs for class ShardInfo.

id: str
The shard's identifier int the cluster.

ip: str
The shard's IP address.

port: int
The shard's port.

unixSocket: str
The shards UDS.

runid: str
The engines run identifier.

minHslot: int
Lowest hash slot served by the shard.

maxHslot: int
Highest hash slot served by the shard.

pendingMessages: int
Number of pending messages

class redgrease.data.**ClusterInfo**(*my_id: str*, *my_run_id: str*, *shards*)
Information about the Redis Gears cluster.

Return object for *redgrease.client.Redis.infocluster* command.

Method generated by attrs for class ClusterInfo.

my_id: str
The identifier of the shard the client is connected to.

shards: List[redgrease.data.ShardInfo]
List of the all the shards in the cluster.

static parse(*res: bytes*) → Optional[redgrease.data.ClusterInfo]
Parses the response from *redgrease.client.Redis.infocluster* into a *ClusterInfo* object.

If the client is not connected to a Redis Gears cluster, *None* is returned.

Returns A ClusterInfo object or None (if not in cluster mode).s

Return type Optional[ClusterInfo]

class redgrease.data.**PyStats**(*TotalAllocated: int*, *PeakAllocated: int*, *CurrAllocated: int*)
Memory usage statistics from the Python interpreter. As returned by *redgrease.client.Redis.pystats*

Method generated by attrs for class PyStats.

TotalAllocated: int
A total of all allocations over time, in bytes.

PeakAllocated: int
The peak allocations, in bytes.

CurrAllocated: int
The currently allocated memory, in bytes.

class redgrease.data.**PyRequirementInfo**(*GearReqVersion: int*, *Name*, *IsDownloaded*, *IsInstalled*, *CompiledOs*, *Wheels*)
Information about a Python requirement / dependency.

Method generated by attrs for class PyRequirementInfo.

GearReqVersion: int
An internally-assigned version of the requirement. (note: this isn't the package's version)

Name: str
The name of the requirement as it was given to the 'requirements' argument of the *pyexecute* command.

IsDownloaded: bool
True if the requirement wheels was successfully download, otherwise *False*.

IsInstalled: bool
True if the requirement wheels was successfully installed, otherwise *False*.

CompiledOs: str
The underlying Operating System

Wheels: List[str]
A List of Wheels required by the requirement.

CHAPTER

SIXTEEN

ADVANCED CONCEPTS

This section discuss some more advanced topics, considerations an usage patterns.

16.1 Redgrease Extras Options

It is always recommended to install either the redgrease package with either `redgrease[client]`, `redgrease[cli]` or `redgrease[all]` package options on your clients.

However for the RedisGears server runtime, you may want to be more prudent with what you install. Therefore, RedGrease strives to give you visibility and control of control of how much of RedGrease you want to install on on your RedisGear server.

For the server you may want to consider these different options:

- `redgrease[runtime]`

  This is the default and recommended package is to install on your server as this gives you access to every runtime feature in RedGrease, including all the *GearFunction* constructs, the *Serverside Redis Commands*, as well as the *Builtin Runtime Functions*.

  This is what *`Gears.pyexecute()`* is going to automatically assume for you when you pass any dynamic *GearFunction* to it, or if you set the `enforce_redgrease` argument to either `True` or just a version string (e.g. `"0.3.12"`).

  This option installs all the dependencies that are needed for any of the RedGrease runtime features, but none of the dependencies that are only required for the RedGrease client features.

- `redgrease`

  You can also install just the bare `redgrease` package without any dependencies. This limits the RedGrease functionalities that you can use to the ones provided by the *"Clean" RedGrease Modules*.

  Most notably this will prevent you from using the *Serverside Redis Commands*.

  To enforce this option, ensure that any calls to *`Gears.pyexecute()`* explicitly set the `enforce_redgrease` argument to `"redgrease"` (without extras). Version qualifiers are supported (e.g. `"redgrease>0.3"`).

- Nothing

  Yes, In some cases RedGrease may be of value even if it is **not** installed in the RedisGears runtime environment. If your are only executing GearFunctions by *Script File Path*, and the script itself is either:

A. Not importing `redgrease` at all (obviously), or

B. Only using explicitly imported *Builtin Runtime Functions*, I.e. only importing RedGrease with:

```
from redgrease.runtime import ...
```

If this is the case, you can enforce that RedGrease is **not** added to the runtime requirements at all by ensuring that any calls to *`Gears.pyexecute()`* explicitly set the `enforce_redgrease` argument to `False`. This option will not add any redgrease requirement to the function and simply ignore any explicit runtime imports.

Note: This only applies to explicit imports of symbols in the `runtime` module, and not to imports of the module itself.

I.e, imports of the form:

```
from redgrease.runtime import GB, hashtag
```

Or:

```
from redgrease.runtime import *
```

But not:

```
import redgrease.runtime
```

Nor:

```
from redgrease import GB, hashtag
```

16.1.1 Dependency Packages per Option

The dependencies of the different extras options are as follows:

- `redgrease`
  - Clean. No dependencies. See *"Clean" RedGrease Modules* for a list of RedGrease modules that can be used.
- `redgrease[runtime]`
  - attrs - This dependency may be removed in future versions.
  - cloudpickle - This dependency may be replaced with dill in future versions.
  - redis
  - packaging - This dependency may be moved to the `client` extra in future versions.
  - wrapt - This dependency may be removed in future versions.
- `redgrease[client]`
  - All the dependencies of `redgrease[runtime]`, plus
  - typing-extensions
  - redis-py-cluster - This dependency may be moved to a new `cluster` extra in future versions.

- `redgrease[cli]`
  - All the dependencies of `redgrease[client]`, plus
  - watchdog
  - ConfigArgParse
  - pyyaml
- `redgrease[all]`
  - All dependencies above

16.1.2 "Clean" RedGrease Modules

The "clean" RedGrease modules, that can be used without extra dependencies are:

- `redgrease.runtime` - Wrapped versions of the built-in runtime functions, but with docstrings and type hints.
- `redgrease.reader` - GearFunction constructors for the various Reader types.
- `redgrease.func` - Function decorator for creating `CommandReader` functions.
- `redgrease.utils` - A bunch of helper functions.
- `redgrease.sugar` - Some trivial sugar for magic strings and such.
- *`redgrease.typing`* - A bunch of type helpers, typically not needed to be imported in application code.
- `redgrease.gears` - The core internals of RedGrease, rarely needed to be imported in application code.
- `redgrease.hysteresis` - A helper module, specifically for the RedGrease CLI. Not intended to be imported in application code.

16.2 Python 3.6 and 3.8+

Dynamically created *GearFunction* objects can only be executed if the client Python version match (major and minor version) the Python version of the RedisGears runtime. At the moment of writing, RedisGears version 1.0.6, is relying on Python 3.7.

RedGrease does however support using any Python version after Python 3.6 inclusive, for all other functionalities.

You are still able to construct an run Gear functions using the RedGrease *GearFunction* objects, but only if executed using *Script File Path*.

This means that you need to:

1. Put your Gear Function code in a separate file from your application code.
2. Ensure that the Gear Function script, only use Python 3.7 constructs.
3. Execute the function by passing the script file path to `redgrease.client.Gears.pyexecute()`.

16.3 Legacy Gear Scripts

You do not have to change any of your existing legacy scripts to start using RedGrease.

RedGrease support running "vanilla" RedisGears Gear functions, i.e. without any RedGrease features, by execution using *Script File Path*.

If you however need to modify any of your legacy scripts, it may be a good idea to add `from redgrease import execute, atomic, configGet, gearsConfigGet, hashtag, log, GearsBuilder, GB` to the import section of your script so that you get the benefits of autocompletion and write-time type checking (assuming your IDE supports it).

CHAPTER

SEVENTEEN

SUPPORT

If you are having issues, or questions, feel free to check out the FAQ below or search the issues on the RedGrease GitHub repository.

If you can't find a satisfactory answer, feel to post it as an "issue" in RedGrease GitHub repository:

- Post a question
- Request a new feature

These will be addressed on a best-effort basis.

17.1 Professional Support

RedGrease is backed by Lyngon Pte. Ltd. and professional support with SLA's can be provided on request. For inquiries please send a mail to support@lyngon.com.

17.2 FAQ

17.2.1 Q: Can I use RedGrease for Commercial Applications?

A: Yes

RedGrease is licensed under the MIT license, which is a very permissive licence and allows for commercial use. The same goes for Open Source Redis, which is licensed under the 3-Clause-BSD licence, and is similarly permissive. However, the RedisGears module is licensed under a custom Redis Source Available License (RSAL), which limits usage to "non-database products".

17.2.2 Q: Can I run Redis Gears / RedGrease on AWS ElastiCache?

A: No

Unfortunately AWS ElastiCache does not (yet) support Redis modules, including Redis Gears.

17.2.3 Q: Why are there so many spelling mistakes?

A: The author suffers from mild dyslexia and has a had time spotting when a word isn't right.

Hopefully the sub-par spelling is not indicative of the quality of the software.

17.3 Reporting issues

- Report a bug

CHAPTER

EIGHTEEN

CONTRIBUTE

Feel free to submit PRs.

18.1 Development Setup

After cloning or forking the repository, it is recommended to do the following:

1. In the project root, create and activate a Python 3.7 virtual environment. E.g:

```
cd redgrease

virtualenv -p python3.7 .venv

source .venv/bin/activate
```

2. Install the Development, Test **and** all package requirements:

```
pip install -r src/requirements-dev.txt
```

3. Install *redgrease* in "develop" mode:

```
pip install -e .
```

4. [optional] If you want to build the docs:

```
pip install -r docs/source/requirements.txt
```

Note: It is highly recommended to check / lint the code regularly (continuously) with:

```
black src/
flake8 src/
isort src/
mypy src/
```

18.2 Local Testing

Sorry, this section is under construction!

To run the test, docker needs to be installed as it is used to spin up clean Redis instances.

Courtesy of :  Pte. Ltd.

Courtesy of :  Pte. Ltd.

PYTHON MODULE INDEX

r

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z