Redis-py is the standard Python client for interacting with Redis, an in-memory data structure store. It is maintained by the Redis Company itself, ensuring that it stays up-to-date with the latest Redis features. The GitHub repository for Redis-py can be found at [redis/redis-py](https://github.com/redis/redis-py).
Connecting to the Server

To connect to a Redis server using Python, you use the `redis.Redis` class. Here's an example of how to establish a connection:

```python
import redis

redis_client = redis.Redis(
    host='localhost',
    port=6379,
    db=2,
    decode_responses=True
)
```

- **host**: This is the address of the Redis server. If you're running Redis locally or in a Docker container, this will typically be `localhost` or `127.0.0.1`.
- **port**: This is the port on which the Redis server is running. The default port for Redis is `6379`, but if you've configured a different port, you'll need to specify it here.
- **db**: Redis supports multiple databases (numbered 0 to 15). You can specify which database you want to connect to using the `db` parameter.
- **decode_responses**: By default, Redis returns data as bytes. Setting `decode_responses=True` ensures that the data is automatically decoded into strings, which is often more convenient for Python applications.

Redis Command List

Redis supports a wide variety of commands for different data structures, such as strings, lists, hashes, sets, and more. The full list of Redis commands can be found on the official Redis documentation website: [Redis Commands](https://redis.io/commands). You can filter the commands based on the data structure you're working with.

The Redis-py documentation provides detailed information on how to use these commands in Python: [Redis-py Documentation](https://redis-py.readthedocs.io/en/stable/).

The following sections highlight some of the most commonly used Redis commands, but this is not an exhaustive list. For a complete list of commands, refer to the official documentation.

String Commands

Redis strings are the most basic type of data structure in Redis. They can store text, numbers, or binary data. Here are some common string commands:

- **set(key, value)**: Sets the value of a key. For example:
  ```python
  r.set('clickCount:/abc', 0)
  ```
  This sets the value of the key `clickCount:/abc` to `0`.

- **get(key)**: Retrieves the value of a key. For example:
  ```python
  val = r.get('clickCount:/abc')
  ```
  This retrieves the value of the key `clickCount:/abc`.

- **incr(key)**: Increments the value of a key by 1. For example:
  ```python
  r.incr('clickCount:/abc')
  ```
  This increments the value of `clickCount:/abc` by 1.

- **mset({key1: val1, key2: val2, ...})**: Sets multiple key-value pairs in a single command. For example:
  ```python
  redis_client.mset({'key1': 'val1', 'key2': 'val2', 'key3': 'val3'})
  ```
  This sets the values for `key1`, `key2`, and `key3` in one operation.

- **mget(key1, key2, ...)**: Retrieves the values of multiple keys in a single command. For example:
  ```python
  print(redis_client.mget('key1', 'key2', 'key3'))
  ```
  This retrieves the values for `key1`, `key2`, and `key3` and returns them as a list.

Other useful string commands include `setex()` (set with expiration), `setnx()` (set if not exists), `getex()` (get with expiration), `getdel()` (get and delete), `incrby()` (increment by a specified amount), `decr()` (decrement), `strlen()` (get the length of the value), and `append()` (append to the value).

List Commands

Redis lists are ordered collections of strings. They are often used to implement queues, stacks, or other data structures that require ordered elements. Here are some common list commands:

- **rpush(key, value1, value2, ...)**: Adds one or more elements to the end of a list. For example:
  ```python
  redis_client.rpush('names', 'mark', 'sam', 'nick')
  ```
  This adds the elements `mark`, `sam`, and `nick` to the list stored under the key `names`.

- **lrange(key, start, end)**: Retrieves a range of elements from a list. For example:
  ```python
  print(redis_client.lrange('names', 0, -1))
  ```
  This retrieves all elements from the list `names`.

- **lpush(key, value1, value2, ...)**: Adds one or more elements to the beginning of a list. For example:
  ```python
  redis_client.lpush('names', 'alice')
  ```
  This adds `alice` to the beginning of the list `names`.

- **lpop(key)**: Removes and returns the first element of a list. For example:
  ```python
  redis_client.lpop('names')
  ```
  This removes and returns the first element from the list `names`.

- **rpop(key)**: Removes and returns the last element of a list. For example:
  ```python
  redis_client.rpop('names')
  ```
  This removes and returns the last element from the list `names`.

Other useful list commands include `lset()` (set the value of an element at a specific index), `lrem()` (remove elements from the list), `llen()` (get the length of the list), and `lpos()` (find the position of an element in the list).

Hash Commands

Redis hashes are maps between string fields and string values. They are often used to represent objects, such as user profiles or session data. Here are some common hash commands:

- **hset(key, mapping)**: Sets the values of multiple fields in a hash. For example:
  ```python
  redis_client.hset('user-session:123', mapping={
      'first': 'Sam',
      'last': 'Uelle',
      'company': 'Redis',
      'age': 30
  })
  ```
  This sets the fields `first`, `last`, `company`, and `age` in the hash stored under the key `user-session:123`.

- **hgetall(key)**: Retrieves all fields and values from a hash. For example:
  ```python
  print(redis_client.hgetall('user-session:123'))
  ```
  This retrieves all fields and values from the hash `user-session:123`.

- **hget(key, field)**: Retrieves the value of a specific field in a hash. For example:
  ```python
  print(redis_client.hget('user-session:123', 'first'))
  ```
  This retrieves the value of the `first` field from the hash `user-session:123`.

- **hkeys(key)**: Retrieves all fields in a hash. For example:
  ```python
  print(redis_client.hkeys('user-session:123'))
  ```
  This retrieves all fields from the hash `user-session:123`.

Other useful hash commands include `hdel()` (delete fields from a hash), `hexists()` (check if a field exists in a hash), `hlen()` (get the number of fields in a hash), and `hstrlen()` (get the length of the value of a field).

Pipelines

Redis pipelines allow you to send multiple commands to the server in a single batch, reducing the number of round-trips between the client and server. This can significantly improve performance, especially when you need to execute many commands in sequence.

Here's an example of using a pipeline:

```python
r = redis.Redis(decode_responses=True)
pipe = r.pipeline()

for i in range(5):
    pipe.set(f"seat:{i}", f"#{i}")

set_5_result = pipe.execute()
print(set_5_result)  # Output: [True, True, True, True, True]

pipe = r.pipeline()

# Chain pipeline commands together.
get_3_result = pipe.get("seat:0").get("seat:3").get("seat:4").execute()
print(get_3_result)  # Output: ['#0', '#3', '#4']
```

In this example, the first pipeline sets five keys in one batch, and the second pipeline retrieves three keys in another batch. Using pipelines reduces network overhead and improves performance.

Redis in Machine Learning

Redis is often used in machine learning (ML) workflows, particularly for storing and serving feature data. Feature stores are a critical component of ML systems, as they allow for the efficient storage and retrieval of features used for training and inference.

In a typical ML workflow, raw data is transformed into features, which are then stored in a feature store. Redis can serve as an online feature store, providing low-latency access to features for real-time inference. For example, after aggregating user data (e.g., average purchase price), the results can be stored in Redis for quick access during inference.

In contrast, historical feature data used for training models is often stored in more traditional data warehouses or cloud storage solutions like Amazon S3. Redis complements these systems by providing fast access to features needed for real-time predictions.