PYCON ITALIA 2026

Beyond Caching

Supercharging **Python** with **Redis**

Aastha Asthana

About Me:

- Aastha Asthana
- Engineer at Salesforce
- Used Python in most of my roles and used Redis with 100M+ monthly active users.
- From Bangalore, India 🇮🇳
- Beyond Caching: Supercharging Python with Redis Data Structures

Quick intro

✋ Raise your hand if you have used Redis

What is Redis?

- **Re**mote **Di**ctionary **S**erver
- In-memory data store
- Open source, written in C
- Lightning fast: sub-millisecond response times
  - ~0.5ms vs Postgres ~5–20ms (20–40× faster)

Why do we need it?

- Faster than traditional DB reads for hot data
- Allows shared state across app instances
- Rich data structures, not just key/value
- Sessions, rate limits, leaderboards

Redis Basics: `SET` and `GET`

```python
import redis

r = redis.Redis(host="localhost", port=6379, decode_responses=True)
r.set("greeting", "ciao pycon")
value = r.get("greeting")
print(value)  # ciao pycon
```

- `SET key value` stores a value
- `GET key` retrieves the value

The Core Idea

Practical Mapping: Python -> Redis

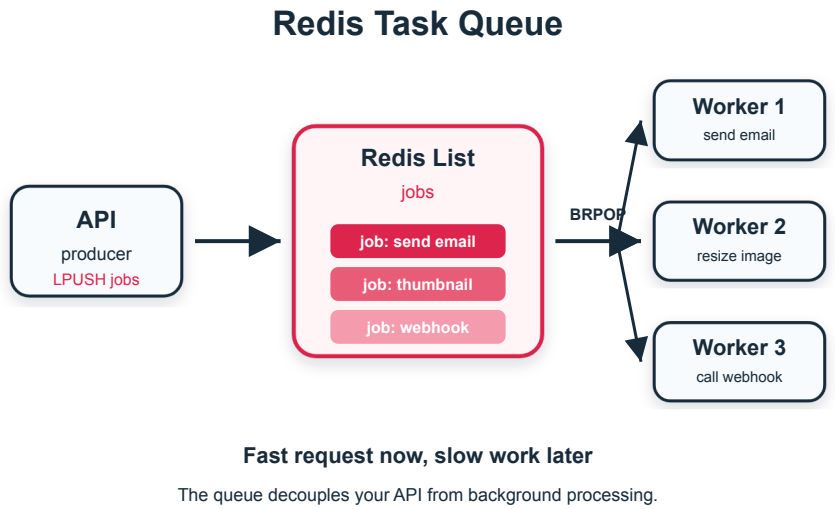
Python Type	Redis Equivalent	Primary Use Case
`list`	List	Task Queues, Activity Feeds
`dict`	Hash	User Sessions, Profiles
`sorted(set)`	Sorted Set	Leaderboards, Rate Limiting
`set`	Set	Unique Visitors, Tagging

Three Real-World Problems & Solutions

Problem #1: Heavy work blocking your API

Background Task Queues with Lists

- User clicks "Sign up"
- API does all the work synchronously:
  - send welcome email
  - generate thumbnail
  - notify other services
- User waits... and waits ❌
- **Fix:** respond immediately, do slow work in the background ✅



API Familiarity: `deque` vs Redis List

Same API you already know, now shared, persistent, and remote

```python
# Python local
from collections import deque

q = deque()
q.appendleft(task)  # add work
q.pop()             # get work
```

```python
# Redis remote (shared, persistent)
import redis

r = redis.Redis()
r.lpush("tasks", task)  # add work
r.brpop("tasks")        # worker waits
# one queue → many workers
```

- `brpop` is a **blocking pop**: worker waits until work arrives, no polling
- One shared queue → spin up as many workers as you need

Producer + Worker (Python)

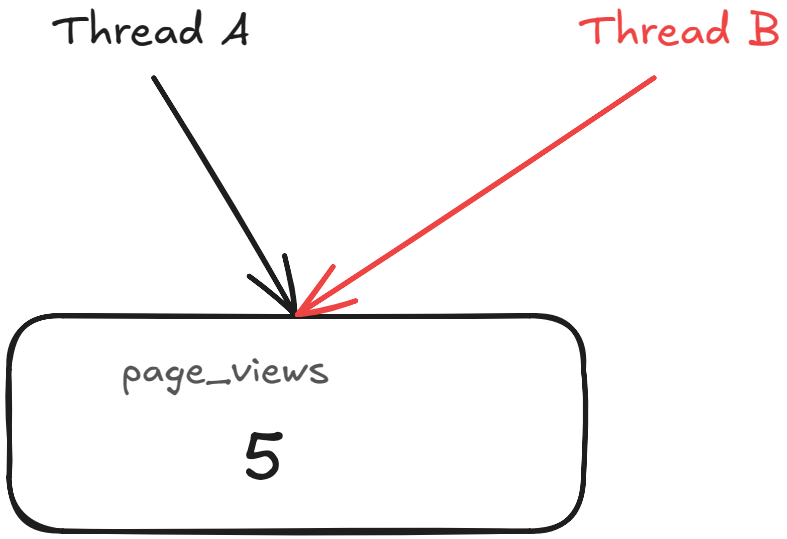
```python
import json
import redis

r = redis.Redis(host="localhost", port=6379, decode_responses=True)

# API adds background work to the right queue
r.lpush("jobs:emails", json.dumps({"user_id": 42, "template": "welcome"}))

# email worker, only listens to its own queue
_, payload = r.brpop("jobs:emails", timeout=0)
job = json.loads(payload)
print(f"Sending email to user {job['user_id']}")
```

- One queue per task type, each worker pool scales independently

Problem #2: Race conditions and Lost updates

Atomic Counters with Hashes

- Two concurrent users trying to modify same resource
- Both read 5, both write 6, you lost an update ❌
- **You need:** one atomic operation, not three steps
- `HINCRBY` = read + modify + write, uninterruptible ✅



API Familiarity: `dict` vs Redis Hash

Same idea, now shared, atomic, and safe across servers

```python
# Python local
session = {"page_views": 0}

session["page_views"] += 1
# NOT safe: read → +1 → write
# another thread can sneak in!
```

```python
# Redis remote (safe!)
import redis
r = redis.Redis()

r.hset("session", "page_views", 0)
r.hincrby("session", "page_views", 1)
# atomic: no thread can interfere
```

- `+=` is 3 steps under the hood, any thread can sneak in between
- `hincrby` does all 3 in one shot, nothing can interrupt it

The Race Condition Problem

```
# Without HINCRBY, the wrong way:

# Both threads read at nearly the same time
Thread A: views = int(r.hget("session", "page_views"))  # 5
Thread B: views = int(r.hget("session", "page_views"))  # 5

# Both increment locally
Thread A: views += 1  # 6
Thread B: views += 1  # 6

# Both write back
Thread A: r.hset("session", "page_views", 6)
Thread B: r.hset("session", "page_views", 6)  # overwrites A's write!

# Expected 7, got 6 ❌
```

The Atomic Solution

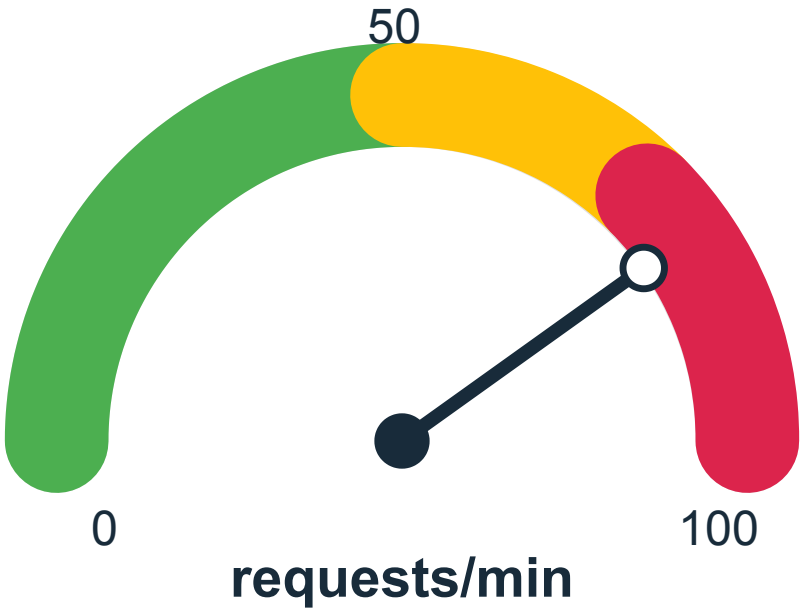
```
# With HINCRBY, the right way:

# Initialize
r.hset("session", "page_views", 5)

# Both threads increment
Thread A: new_count = r.hincrby("session", "page_views", 1)  # returns 6
Thread B: new_count = r.hincrby("session", "page_views", 1)  # returns 7

# Expected: 7  |  Actual: 7 ✅
```

- Redis does Read → Modify → Write in one atomic operation
- No other thread can interfere between steps!
- **Faster:** 1 network round-trip instead of 3

Problem #3: API Abuse & Traffic Spikes

Rate Limiting with Sorted Sets

- 1000 requests/second -> API goes down ❌
- Simple counter? Easy to exploit at minute boundaries
- **You need:** a true sliding window, shared across servers
- Sorted Sets make it possible ✅



The Problem with Simple Counters

```
# Fixed window counter (wrong):

12:00:59 → 100 requests ✅ (counter = 100)
12:01:01 → 100 requests ✅ (counter reset to 0, now 100 again)

# 200 requests in 2 seconds! ❌
```

- **Why?** Counter resets at minute boundary

How Sorted Set Solves It

- Each request stored with timestamp
- True sliding window: always checks "last 60 seconds"
-

```
# No matter when requests come:
# 00:59 → checks requests from 23:59 to 00:59
# 01:01 → checks requests from 00:01 to 01:01
# 01:30 → checks requests from 00:30 to 01:30

# Always a true 60-second window! ✅
```

What you'd write in plain Python

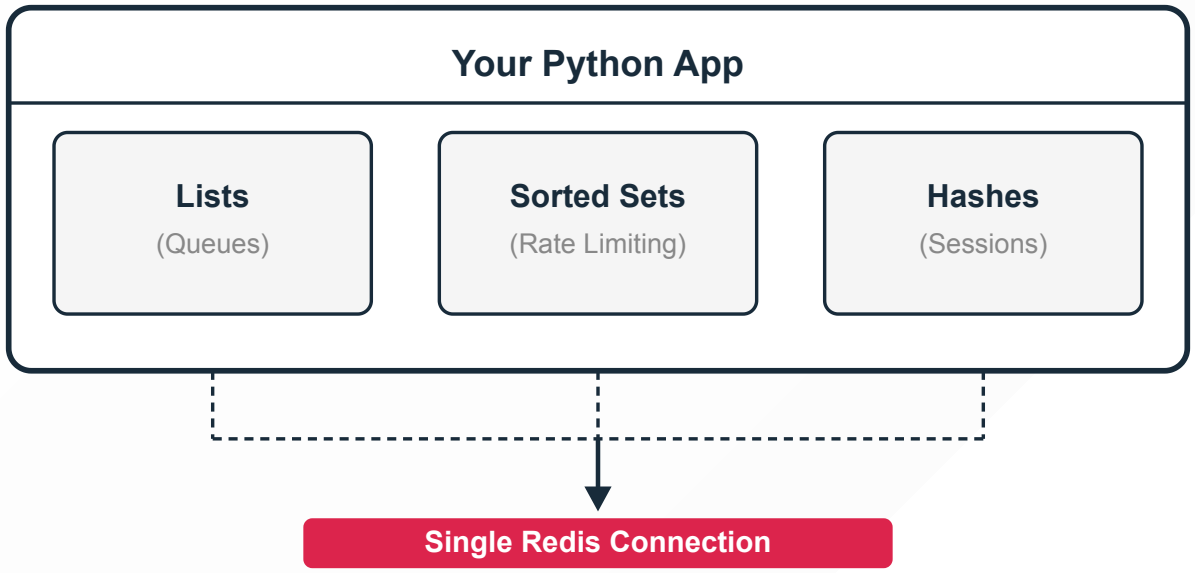
```
# Per-process, no coordination — broken across servers
requests = []
now = time.time()
requests.append(now)
requests = [t for t in requests if t > now - 60]
if len(requests) > 100:
    return "Rate limited!"
```

- Works on **one** process: append, filter by time, check length
- Run it on 10 servers → 10 separate lists, no shared limit ❌
- Need: one **shared** sliding window across the whole fleet

The Sliding Window Solution

```
import time
import redis

r = redis.Redis(host="localhost", port=6379, decode_responses=True)

user_id = "alice"
now = time.time()
window = 60  # 60 seconds
limit = 100

# Add current request with timestamp as score
request_id = f"{user_id}:{now}"
r.zadd(f"rate:{user_id}", {request_id: now})

# Remove requests older than 60 seconds
r.zremrangebyscore(f"rate:{user_id}", 0, now - window)

# Count requests in current window
count = r.zcard(f"rate:{user_id}")

if count > limit:
    return "Rate limited! Try again later."
```

Architecture Summary



One tool. Minimal infrastructure. Maximum performance.

When NOT to reach for Redis

RAM Limits

Redis holds data in memory.
If your dataset is TB-scale,
use **PostgreSQL / SQL**.

Complex Joins

Redis is not relational.
For many-to-many queries,
use a **SQL database**.

Hard Durability

Redis is a speed layer.
Keep source-of-truth in
a **primary database**.

Summary

Redis is more than just a cache:

📋 **Lists** → Background task queues. Simple, persistent, scalable
🔐 **Hashes** → Atomic counters, sessions. Safe from race conditions
🛡️ **Sorted Sets** → Rate limiting, leaderboards. True sliding window

Grazie! 🇮🇹

Questions?

Find me at:

- 🐦 Twitter/X: @asthana_aastha
- 💼 LinkedIn: https://www.linkedin.com/in/aasthaasthana/

Resources:

- redis.io/commands
- redis-py docs