Asynchronous Task Queues

Kay Ashaolu - Instructor

Aishwarya Sriram - TA

Chapter 2: A Full Python Refresher

Object-Oriented Programming Fundamentals

- Transition from procedural to object-based design
- Emphasis on encapsulation, inheritance, and composition
- Practical coding examples for real-world analogies

Introduction to Object-Oriented Programming (OOP)

- OOP models real-world entities (students, devices, etc.)
- Shifts code from procedural (functions + data) to encapsulated objects
- Improves code organization & readability

Data: Dictionary vs. Object

Traditional Approach:

```
1 student = {"name": "Rolf", "grades": (90, 88, 87)}
2 def average(seq):
3     return sum(seq) / len(seq)
4 print(average(student["grades"]))
```

Limitation: Lacks semantic connection between data and behavior

Defining a Python Class

- Use the `class` keyword to create a blueprint
- The `__init__` method initializes instance attributes
- The `self` parameter references the instance

```
1 class Student:
2     def __init__(self):
3         self.name = "Rolf"
4         self.grades = (90, 88, 87)
```

Adding Behavior with Methods

- Methods are functions defined within a class
- Access & modify instance attributes via `self`
- Encapsulates data and functionality together

```
1 class Student:
2     def __init__(self):
3         self.name = "Rolf"
4         self.grades = (90, 88, 87)
5
6     def average_grade(self):
7         return sum(self.grades) / len(self.grades)
```

Example: Calculating a Student's Average

- Create a Student object and call its method

```
student = Student()
print(student.average_grade())  # Outputs: 88.33...
```

- Emphasizes encapsulated data & behavior

Class Inheritance in Python

- Inheritance allows a class to derive properties and methods from another
- Models “is-a” relationships (e.g., Printer is a Device)
- Reduces redundancy by reusing code

```python
class Device:
    def __init__(self, name, connected_by):
        self.name = name
        self.connected_by = connected_by
        self.connected = True

    def __str__(self):
        return f"device {self.name} {self.connected_by}"

    def disconnect(self):
        self.connected = False
        print("disconnected")
```

Extending with the Printer Class

- Printer inherits from Device and adds extra features

```
1  class Printer(Device):
2      def __init__(self, name, connected_by, capacity):
3          super().__init__(name, connected_by)
4          self.capacity = capacity
5          self.remaining_pages = capacity
6
7      def __str__(self):
8          return f"{super().__str__()} - remaining pages {self.remaining_pa
9
10     def print_pages(self, pages):
11         if not self.connected:
12             print("printer is not connected")
13             return
14         print(f"printing {pages} pages")
15         self.remaining_pages -= pages
```

Class Composition vs. Inheritance

When to Use Composition

- **Conceptual Clarity:**
  - A Book is not a Bookshelf.
  - Bookshelf *has-a* collection of Book objects, rather than being one.
- **Technical Benefits:**
  - **Modularity:** Changes in one component (e.g., Book) do not force changes in the container (Bookshelf).
  - **Flexibility:** Easier to mix and match behaviors without rigid parent-child constraints.
  - **Reduced Coupling:** Keeps classes focused on their primary responsibilities..

Class Composition vs. Inheritance

When to Use Composition

- **Example Comparison:**
  - Inheritance: A Book inheriting from Bookshelf forces unnecessary attributes.
  - Composition: A Bookshelf holds Book objects, reflecting real-world relationships.

Composition Example: Bookshelf & Book

- **Using Composition:**

```python
class Book:
    def __init__(self, title):
        self.title = title

    def __str__(self):
        return f"Book: {self.title}"

class Bookshelf:
    def __init__(self, *books):
        self.books = books

    def __str__(self):
        return f"Bookshelf with {len(self.books)} books"
```

- Clear separation: A bookshelf contains books; a book remains an independent entity.

Summary: Chapter 2 (Python OOP)

- Transition from dictionaries to objects
- Use of methods, inheritance, and composition
- Composition offers flexibility and modularity over inheritance in many scenarios

Chapter 12: Task Queues with rq & Sending Emails

Background Processing in Web Architecture

- Offload heavy or time-consuming tasks
- Enhance API responsiveness by processing tasks asynchronously
- Use of Redis as a message broker with the rq library

What is a Queue Data Structure?

- **Definition:**
  - A queue is a First-In-First-Out (FIFO) data structure
  - Items are added at the rear and removed from the front

What is a Queue Data Structure?

- **Comparison:**
  - **Dictionary:** Key-value mapping with fast lookup
  - **Array (List):** Ordered collection accessed by index
  - **Queue:** Enforces order for processing tasks sequentially

What is a Queue Data Structure?

- **Real-World Analogy:**
  - Think of a queue as a line at a ticket counter: first come, first served.

Setting Up Redis for Task Queues

- Redis acts as an in-memory data store and message broker
- Use Render.com or Docker to host Redis

```
# Example Docker command to run Redis locally:
docker run -p 6379:6379 redis
```

Integrating rq with a Flask Application

- **rq (Redis Queue):** A Python library for managing task queues
- Enqueue tasks from your Flask app to be processed asynchronously
- Steps include:
  1. Installing rq (`pip install rq`)
  2. Connecting to Redis
  3. Enqueuing background tasks (e.g., sending emails)

Code Example: Enqueueing Tasks

- **tasks.py**: Define the email sending task

```python
import os
from dotenv import load_dotenv
load_dotenv()

def send_user_registration_email(email, username):
    # Simulated email sending function
    print(f"Sending registration email to {email} for {username}")
```

Flask App Integration with rq

- **app.py**: Connect Flask with Redis and enqueue tasks

```python
import os
import redis
from rq import Queue
from flask import Flask, request, current_app
from tasks import send_user_registration_email

app = Flask(__name__)
connection = redis.from_url(os.getenv("REDIS_URL"))
app.queue = Queue('emails', connection=connection)

@app.route('/register', methods=['POST'])
def register():
    # ... (user registration logic)
    email = request.form['email']
    username = request.form['username']
    current_app.queue.enqueue(send_user_registration_email, email, username)
    return "User created successfully", 201
```

Processing Background Tasks with rq Worker

- Run the worker as a separate process to consume queued tasks
- The worker monitors the Redis queue and processes tasks asynchronously

```
# Docker example command:
docker run -w /app rest-api-recording-email sh -c "rq worker -u $REDIS_URL emails"
```

Recap: Chapter 12 (Task Queues)

- **Task Queue:** Offloads heavy tasks to improve API responsiveness
- **Redis:** In-memory data store serving as the broker
- **rq Library:** Simplifies task management and background processing
- **Workflow:** Enqueue tasks from Flask → Worker processes tasks → e.g., Sending emails

Questions?