

PROJECT COURSE

# Gmail Cleanup - Practical Scripting

6 Chapters

Philip Osztromok

Generated with Claude

GMAIL CLEANUP - PRACTICAL SCRIPTING

# Contents


| | | |
|---|---|---|
| **Chapter 1** | Configuration & API Setup | 2 |
| **Chapter 2** | Configuration Management | 9 |
| **Chapter 3** | Gmail API Fundamentals | 13 |
| **Chapter 4** | Building Email Utilities | 17 |
| **Chapter 5** | Complete Cleanup Workflow | 20 |
| **Chapter 6** | Generalization & Extension | 24 |

# ⚙️ Configuration & API Setup

> *Before you can access Gmail from Python, you need to set up the Gmail API and create authentication credentials. This chapter walks you through the complete setup process: creating a Google Cloud project, obtaining API credentials, and organizing your project structure for secure credential management.*

## 📋 Overview

To interact with Gmail from Python, you need:

1. A Google Cloud project (free tier available)
2. Gmail API enabled in that project
3. OAuth 2.0 credentials (client ID and secret)
4. Python libraries installed
5. Secure credential storage

## 🔧 Step-by-Step Setup

### Step 1: Create a Google Cloud Project

> **1.** Go to Google Cloud Console

> **2.** Click "Create Project" in the top-left dropdown

3. Enter project name: "Gmail Cleanup Utility"

4. Click "Create" and wait for it to be created (2-3 minutes)

## Step 2: Enable the Gmail API

1. In the Cloud Console, go to "APIs & Services" → "Library"

2. Search for "Gmail API"

3. Click on it and press "Enable"

4. Wait for the API to be enabled (30 seconds)

## Step 3: Create OAuth 2.0 Credentials

1. Go to "APIs & Services" → "Credentials"

2. Click "Create Credentials" → "OAuth client ID"

3. Select "Desktop application"

4. Click "Create" and download the JSON file

5. Save it as `credentials.json` in your project

## 📁 Project Structure

Organize your Gmail cleanup project like this:

```
gmail-cleanup/
├── .env                        ← Gmail account email (loaded at ru
├── .env.example                ← Template for .env
├── credentials.json            ← Google OAuth credentials (from St
├── token.json                 ← Generated after first auth (gitign
├── requirements.txt           ← Python dependencies
├── gmail_utils.py             ← Gmail utility functions
├── cleanup.py                 ← Main cleanup script
├── .gitignore                 ← Don't commit secrets!
└── tests/
    └── test_gmail_utils.py
```

🔒 **CRITICAL:** Never commit `credentials.json` or `token.json` to Git! Add them to `.gitignore`.

## 📦 Python Dependencies

Create `requirements.txt`:

```
google-auth-oauthlib==1.1.0
google-auth-httplib2==0.2.0
google-api-python-client==2.100.0
python-dotenv==1.0.0
```

Install them:

```
pip install -r requirements.txt
```

# 🔐 Understanding OAuth 2.0

Gmail API uses OAuth 2.0 for secure authentication. Here's how it works:

1. **First Run:** User visits Google login page (browser opens automatically)
2. **Grant Access:** User clicks "Allow" to grant your app access to Gmail
3. **Token Saved:** Python saves access token to `token.json`
4. **Future Runs:** Script reads token from file, no login needed (until token expires)

> 💡 **Key Concept**
>
> You only need to authorize once! After that, the script can access Gmail without manual login.

# 🔧 Configuration File Setup

Create `.env.example` (template for users):

```
# Gmail account to manage
GMAIL_USER=your-email@gmail.com

# API settings
API_REQUEST_TIMEOUT=10
MAX_RESULTS=100

# Cleanup settings
```

```
DELETE_OLDER_THAN_DAYS=365
MIN_ATTACHMENT_SIZE_MB=10
```

Create `.env` with your actual values (don't commit this!):

```
GMAIL_USER=emubantam@gmail.com
API_REQUEST_TIMEOUT=10
MAX_RESULTS=100
DELETE_OLDER_THAN_DAYS=365
MIN_ATTACHMENT_SIZE_MB=10
```

# ⚠️ Common Setup Issues

## Issue 1: "Gmail API has not been used in project"

**Solution:** Make sure you enabled the Gmail API in Step 2. Give it 30 seconds to activate.

## Issue 2: "Credentials file not found"

**Solution:** Make sure `credentials.json` is in the same folder as your Python script.

## Issue 3: "2-factor authentication enabled"

**Solution:** Gmail requires 2FA. This is good for security! The OAuth flow handles it.

# 💻 Coding Challenges

### Challenge 1: Project Setup

Set up the complete project structure:

1. Create gmail-cleanup/ directory

2. Create .env and .env.example files
3. Create requirements.txt with Gmail API dependencies
4. Create credentials.json from Google Cloud Console
5. Create .gitignore with appropriate entries

**Goal:** Complete the setup without errors.
→ Solution

### Challenge 2: Install Dependencies

Install and verify Python dependencies:

1. pip install -r requirements.txt
2. Create a test script that imports all libraries
3. Print library versions to verify installation

**Goal:** Verify all dependencies are correctly installed.
→ Solution

### Challenge 3: OAuth Authentication

Create a script that performs initial OAuth setup:

1. Read credentials.json
2. Perform OAuth flow (opens browser)
3. Save token.json for future use
4. Verify that token.json was created

**Goal:** Understand and complete the OAuth authentication flow.
→ Solution

## 🎯 What's Next

Chapter 2 will teach you how to load and manage configuration securely with environment variables and .env files, then Chapter 3 will connect to the Gmail API and start querying your mailbox.

# 🔧 Configuration Management

> *Configuration should never be hardcoded! This chapter teaches you how to load settings from environment variables using `.env` files, validate them, and handle errors gracefully. This is a critical skill for secure, portable applications.*

# 📋 Why Configuration Management Matters

Consider this dangerous code:

```
# ❌ BAD: Hardcoded credentials!
GMAIL_USER = "my-email@gmail.com"
API_TIMEOUT = 10
```

Problems:

- Email is visible in the code
- Can't change settings without editing the code
- Different values needed for development vs. production
- Easy to accidentally commit secrets to Git

The solution: **.env files + load_dotenv()**

# 🔐 The .env Pattern

Create a `config.py` module:

```
import os
from pathlib import Path
from dotenv import load_dotenv

# Load .env file from same directory as this script
ENV_FILE = Path(__file__).parent / ".env"
load_dotenv(ENV_FILE)

class Config:
    """Application configuration."""

    def __init__(self):
        # Gmail settings
        self.gmail_user = os.environ.get('GMAIL_USER')
        self.api_timeout = int(os.environ.get('API_REQUEST_TIME
        self.max_results = int(os.environ.get('MAX_RESULTS', '1

        # Cleanup settings
        self.days_old = int(os.environ.get('DELETE_OLDER_THAN_D
        self.min_attachment_mb = int(os.environ.get('MIN_ATTACH

    def validate(self) -> bool:
        """Validate required configuration."""
        if not self.gmail_user:
            print("❌ Missing GMAIL_USER in .env")
```

## 📝 Using Configuration in Your Script

```
from config import Config

def main():
    # Load configuration
    config = Config()
```

```
    # Validate it
    if not config.validate():
        return False

    # Use it
    print(f"Cleaning emails from: {config.gmail_user}")
    print(f"Deleting emails older than: {config.days_old} days")

    return True

if __name__ == "__main__":
    main()
```

# ⚠️ Common Configuration Errors

## Error 1: .env file not found

**Cause:** File is in wrong directory or has wrong name

**Fix:** Make sure `.env` is in the same folder as `config.py`

## Error 2: Variables are None

**Cause:** Variable name in code doesn't match .env

**Fix:** Case matters! Use exact same names

## Error 3: Type conversion errors

**Cause:** Trying to convert non-numeric value to int

**Fix:** Validate that .env values are correct type

# 💻 Coding Challenges

### Challenge 1: Create config.py

Build a `Config` class that:

1. Loads .env file from current directory
2. Reads GMAIL_USER, API_REQUEST_TIMEOUT, MAX_RESULTS
3. Uses defaults for timeout (10) and max_results (100)
4. Returns all values

**Goal:** Create reusable configuration module.
→ Solution

### Challenge 2: Configuration Validation

Extend `Config.validate()` to check:

1. GMAIL_USER is not empty
2. GMAIL_USER contains @ symbol (valid email)
3. API_REQUEST_TIMEOUT is positive number
4. Return True if all valid, False otherwise

**Goal:** Practice data validation.
→ Solution

### Challenge 3: Error Handling & Logging

Enhance config loading with error handling:

1. Catch missing .env file gracefully
2. Handle type conversion errors
3. Print helpful error messages (what's wrong, how to fix)
4. Test with missing/invalid values

**Goal:** Build robust configuration system.
→ Solution

# Gmail API Fundamentals

> *Now that you have credentials and configuration set up, it's time to connect to Gmail and start querying emails. This chapter teaches the Gmail API structure, how to authenticate, query emails, and parse the responses.*

## Connecting to Gmail

```
from google.auth.transport.requests import Request
from google.oauth2.credentials import Credentials
from google_auth_oauthlib.flow import InstalledAppFlow
from googleapiclient.discovery import build

def get_gmail_service():
    """Authenticate and return Gmail service."""
    SCOPES = ['https://www.googleapis.com/auth/gmail.modify']

    creds = None

    # Load existing token if available
    try:
        creds = Credentials.from_authorized_user_file('token.js
    except:
        pass

    # If no valid credentials, perform OAuth flow
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())
        else:
            flow = InstalledAppFlow.from_client_secrets_file(
                'credentials.json', SCOPES
```

# 🔍 Querying Emails

## Gmail Search Syntax

| Query | Meaning |
| --- | --- |
| `before:2020/01/01` | Emails before January 1, 2020 |
| `from:sender@example.com` | Emails from specific sender |
| `filename:pdf` | Emails with PDF attachments |
| `size:>10M` | Emails larger than 10MB |
| `has:attachment` | Emails with attachments |

## Query Example

```
def find_old_emails(service, days_old: int) -> list:
    """Find emails older than N days."""
    from datetime import datetime, timedelta

    cutoff_date = (datetime.now() - timedelta(days=days_old)).str
    query = f"before:{cutoff_date}"

    try:
        results = service.users().messages().list(
            userId='me',
            q=query,
            maxResults=10
        ).execute()

        messages = results.get('messages', [])
        print(f"✅ Found {len(messages)} emails older than {days_
```

```
        return messages

    except Exception as e:
        print(f"❌ Error: {e}")
        return []
```

## 📨 Parsing Email Details

```
def get_email_details(service, message_id: str) -> dict:
    """Get details about a single email."""
    try:
        message = service.users().messages().get(
            userId='me',
            id=message_id
        ).execute()

        headers = message['payload']['headers']
        subject = next(h['value'] for h in headers if h['name']
        sender = next(h['value'] for h in headers if h['name']
        date_str = next(h['value'] for h in headers if h['name'

        size = int(message.get('sizeEstimate', 0))

        return {
            'id': message_id,
            'subject': subject,
            'from': sender,
            'date': date_str,
            'size_bytes': size,
            'size_mb': size / 1024 / 1024
        }

    except Exception as e:
```

## 💻 Coding Challenges

### Challenge 1: Connect to Gmail API

Create `gmail_utils.py` with:

1. Write `get_gmail_service()` function
2. Handle OAuth authentication and token caching
3. Return authenticated Gmail service object

**Goal:** Master Gmail API authentication.
→ Solution

### Challenge 2: Query Emails

Create functions to find:

1. Emails older than N days
2. Emails from specific sender
3. Emails with attachments larger than N MB

**Goal:** Practice Gmail search queries.
→ Solution

### Challenge 3: Parse and Report

Create a script that:

1. Finds 10 old emails
2. Gets details for each (subject, sender, date, size)
3. Displays them in a table
4. Calculates total storage

**Goal:** End-to-end email discovery and reporting.
→ Solution

# 🔧 Building Email Utilities

> *Now that you understand the Gmail API, it's time to build reusable utility functions. This chapter teaches modular design: creating helper functions that you can use in different ways, handle errors gracefully, and test independently.*

## 📦 Utility Module Design

```
class GmailCleanup:
    """Reusable Gmail cleanup utilities."""

    def __init__(self, service, config):
        self.service = service
        self.config = config
        self.deleted_count = 0
        self.archived_count = 0
        self.total_storage_freed = 0

    def find_emails(self, query: str, limit: int = 10) -> list:
        """Find emails matching query."""
        try:
            results = self.service.users().messages().list(
                userId='me',
                q=query,
                maxResults=limit
            ).execute()
            return results.get('messages', [])
        except Exception as e:
            print(f"❌ Error finding emails: {e}")
            return []

    def delete_email(self, message_id: str) -> bool:
```

# 🎯 Key Utility Functions

## Find Old Emails

```
def find_old_emails(self, days: int) -> list:
    from datetime import datetime, timedelta
    cutoff = (datetime.now() - timedelta(days=days)).strftime('%Y
    query = f"before:{cutoff}"
    return self.find_emails(query)
```

## Find Large Attachments

```
def find_large_attachments(self, min_size_mb: int) -> list:
    min_bytes = min_size_mb * 1024 * 1024
    query = f"has:attachment size:{min_bytes}"
    return self.find_emails(query)
```

## Find from Sender

```
def find_from_sender(self, sender: str) -> list:
    query = f"from:{sender}"
    return self.find_emails(query)
```

# 💻 Coding Challenges

### Challenge 1: Build GmailCleanup Class

Create `GmailCleanup` class with methods:

1. `find_emails(query, limit)`
2. `delete_email(message_id)`
3. `get_email_details(message_id)`

**Goal:** Build reusable email utility class.
→ Solution

### Challenge 2: Specialized Finders

Add methods to find:

1. Emails older than N days
2. Emails with attachments > N MB
3. Emails from specific sender

**Goal:** Build specialized queries.
→ Solution

### Challenge 3: Statistics & Reporting

Extend class with:

1. Track deleted/archived count
2. Calculate storage freed
3. Generate cleanup report

**Goal:** Build comprehensive reporting.
→ Solution

# 🔄 Complete Cleanup Workflow

> *Now it's time to bring everything together: configuration, API connection, utility functions, and safety checks. This chapter teaches you how to orchestrate a complete, safe cleanup workflow that finds emails, previews them, and deletes or archives them with full reporting.*

## 📋 The Cleanup Workflow

**Step 1: Load Configuration**
Read .env file and validate settings

**Step 2: Authenticate**
Connect to Gmail API using OAuth

**Step 3: Find Emails**
Query for emails matching cleanup criteria

**Step 4: Preview Results**
Display what will be deleted (safety check!)

**Step 5: Confirm Action**
Ask user before deleting (never automatic!)

**Step 6: Execute Cleanup**
Delete or archive emails

**Step 7: Report Results**
Show what was deleted and storage freed

# 🔧 Main Cleanup Script

```
import sys
from config import Config
from gmail_utils import get_gmail_service, GmailCleanup

def preview_emails(emails: list, cleanup: GmailCleanup, limit:
    """Show sample of emails to be deleted."""
    print("\n📋 PREVIEW (first 5):")
    print("-" * 80)

    for email in emails[:limit]:
        details = cleanup.get_email_details(email['id'])
        if details:
            print(f"Subject: {details['subject']}")
            print(f"From: {details['from']}")
            print(f"Date: {details['date']}")
            print(f"Size: {details['size_mb']:.2f} MB")
            print("-" * 80)

def confirm_action(email_count: int) -> bool:
    """Ask user to confirm deletion."""
    print(f"\n⚠️  About to delete {email_count} emails!")
    response = input("Type 'yes' to confirm: ")
    return response.lower() == 'yes'

def main():
```

# ⚠️ Safety Features

- ✅ Configuration validation (catches errors early)
- ✅ Preview of emails before deletion (visual confirmation)

- ✅ User confirmation prompt (prevents accidents)
- ✅ Error handling throughout (graceful failures)
- ✅ Comprehensive reporting (what happened)

# 💻 Coding Challenges

### Challenge 1: Safe Deletion

Create a workflow that:

1. Finds 10 old emails
2. Displays subject, sender, date for preview
3. Asks user to confirm (input validation)
4. Deletes only if confirmed

**Goal:** Build safe, user-friendly deletion.
→ Solution

### Challenge 2: Multi-Criteria Cleanup

Build cleanup that finds and deletes:

1. Emails older than 365 days
2. Emails with large attachments
3. Emails from specific senders

**Goal:** Handle multiple cleanup criteria.
→ Solution

### Challenge 3: Complete Workflow

Build the full cleanup script with:

1. Configuration loading and validation

2. Gmail authentication
3. Finding and previewing emails
4. User confirmation
5. Deletion and reporting

**Goal:** End-to-end cleanup automation.

→ Solution

# 🎯 Generalization & Extension

> *You've built a complete Gmail cleanup utility! Now learn to extend these patterns to other email services, schedule automatic cleanup, and build variations for different use cases. This final chapter teaches how to think generically and reuse your knowledge.*

## 🔄 Generalizing the Pattern

The pattern you learned works for ANY email service:

| Email Service | API | Differences |
|---|---|---|
| Gmail | Gmail API v1 | OAuth, label-based organization |
| Outlook/Office 365 | Microsoft Graph | Folder-based, different auth |
| Apple Mail | IMAP protocol | Standard email protocol |
| Yahoo Mail | OAuth + IMAP | Similar to Gmail |

**The core pattern stays the same:**

1. Authenticate securely
2. Query for emails matching criteria
3. Preview before deleting
4. Execute action with user confirmation
5. Report results

# Scheduling Automatic Cleanup

Extend your script to run automatically with a scheduler:

```
# On Linux/Mac: Add to crontab
# 0 2 * * * /usr/bin/python3 /home/user/cleanup.py
# (Runs at 2 AM daily)

# On Windows: Use Task Scheduler
# Or use Python's APScheduler library

from apscheduler.schedulers.background import BackgroundScheduler
import time

def schedule_cleanup():
    scheduler = BackgroundScheduler()
    # Run cleanup every day at 2 AM
    scheduler.add_job(main, 'cron', hour=2, minute=0)
    scheduler.start()

    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        scheduler.shutdown()
```

# Extension Ideas

## 1. Selective Cleanup

Delete only specific types of emails:

- From promotional senders only
- With specific keywords in subject
- Larger than N MB

- With certain file types attached

## 2. Archive Instead of Delete

Move emails to archive instead of deleting:

```
def archive_instead_of_delete(self, message_id):
    # Remove from INBOX, keep in ALL_MAIL
    self.service.users().messages().modify(
        userId='me',
        id=message_id,
        body={'removeLabelIds': ['INBOX']}
    ).execute()
```

## 3. Logging & Notifications

Track cleanup history and send reports:

```
import logging
from datetime import datetime

logging.basicConfig(
    filename='cleanup.log',
    level=logging.INFO,
    format='%(asctime)s - %(message)s'
)

def log_cleanup(stats):
    logging.info(f"Deleted {stats['deleted']} emails, freed {stat
```

## 4. Integration with Other Services

- Send cleanup report via email
- Log to cloud storage (Google Drive, Dropbox)
- Integrate with monitoring tools (Slack, Discord)

- Store stats in database for analysis

# 💡 Common Patterns You've Learned

These patterns apply to ANY scripting project:

| Pattern | When to Use |
|---|---|
| Configuration from .env | Any project with settings |
| API authentication | Working with external services |
| Error handling & validation | Robust production code |
| User confirmation | Destructive operations |
| Progress reporting | Long-running operations |
| Logging & monitoring | Debugging and auditing |

# 💻 Final Challenges

### Challenge 1: Extend to Another Email Service

Adapt the pattern to Outlook/Microsoft Graph:

1. Study Microsoft Graph email API
2. Modify authentication (OAuth for Azure)
3. Update query syntax (Outlook filters)
4. Create equivalent cleanup utility

**Goal:** Apply patterns to different API.
→ Solution

### Challenge 2: Archive Instead of Delete

Create version that archives instead:

1. Archive old emails to archive folder
2. Keep in Gmail but hidden
3. Report storage saved
4. Make it reversible

**Goal:** Safer alternative to deletion.
→ Solution

### Challenge 3: Production-Ready Utility

Build production-ready tool with:

1. Scheduling (APScheduler)
2. Logging to file
3. Error notifications (email)
4. Comprehensive documentation

**Goal:** Utility you'd use daily.
→ Solution

# 🎓 Course Complete!

### ✅ What You've Mastered

- Setting up APIs and authentication securely
- Configuration management with environment variables
- Connecting to external services from Python
- Building reusable utility classes

- Creating safe, user-friendly automation
- Generalizing patterns to other problems

You now have the skills to build automation utilities for almost any service or problem. The patterns you learned here scale from simple scripts to complex enterprise automation. Keep practicing, and keep building! 🚀