# 🐍 Python Days 31-40: Intermediate+ Python, APIs & Capstone Projects

This section covers intermediate-level projects using Tkinter, introduces the powerful world of Application Programming Interfaces (APIs), and culminates in two major capstone projects that integrate multiple advanced skills.

## Day 31: Flash Card App Capstone Project

The goal is to build a language-learning flashcard app using **Tkinter** for the UI and **Pandas** for data management.

### Key Concepts

1. **Reading Data with Pandas:** Instead of manually handling CSV files, we use the Pandas library for robust and easy data manipulation.
    - `pandas.read_csv()` : Reads a CSV file into a DataFrame, a powerful table-like data structure.
    - `DataFrame.to_dict(orient="records")` : Converts the DataFrame into a list of dictionaries, where each dictionary represents a row (e.g., `[{'French': 'partie', 'English': 'part'}, ...]` ). This format is ideal for working with individual flashcards.
2. **Tkinter UI & Canvas:**
    - **Canvas Widget:** Used for drawing shapes and displaying images and text. It's perfect for creating the flashcard look.
    - `canvas.create_image()` : Places an image on the canvas.
    - `canvas.create_text()` : Places text on the canvas. We'll create text elements for the language title and the word itself.
    - `canvas.itemconfig()` : A crucial method used to change the properties of an item already on the canvas (e.g., change the text or color).
    - `canvas.config()` : Used to change the properties of the canvas widget itself (e.g., its background color).
3. **Application Logic:**
    - **State Management:** The program needs to keep track of the `current_card` .
    - **Flipping Mechanism:** We use `window.after()` to schedule an action (like flipping the card) to happen after a certain delay without freezing the entire application.

- **Saving Progress:** When the user knows a word, it should be removed from the list of words to learn. The remaining words are saved to a new CSV file ( `words_to_learn.csv` ) so the user can pick up where they left off.

## Code Implementation Snippets

### 1. Reading the Data:

```
import pandas

try:
    # Try to load progress first
    data = pandas.read_csv("data/words_to_learn.csv")
except FileNotFoundError:
    # If no progress file, start with the full list
    original_data = pandas.read_csv("data/french_words.csv")
    to_learn = original_data.to_dict(orient="records")
else:
    to_learn = data.to_dict(orient="records")

current_card = {}

def next_card():
    global current_card, flip_timer
    window.after_cancel(flip_timer) # Cancel previous timer
    current_card = random.choice(to_learn)
    canvas.itemconfig(card_title, text="French", fill="black")
    canvas.itemconfig(card_word, text=current_card["French"], fill="black")
    canvas.itemconfig(card_background, image=card_front_img)
    flip_timer = window.after(3000, func=flip_card) # Set a new timer
```

### 2. Flipping the Card:

```
def flip_card():
    canvas.itemconfig(card_title, text="English", fill="white")
    canvas.itemconfig(card_word, text=current_card["English"], fill="white")
    canvas.itemconfig(card_background, image=card_back_img)
```

### 3. Saving Known Words:

```python
def is_known():
    to_learn.remove(current_card)
    # Create a new DataFrame with the remaining words and save it
    data = pandas.DataFrame(to_learn)
    data.to_csv("data/words_to_learn.csv", index=False) # index=False prevents pandas from writi
    next_card()
```

## Day 32: Automated Birthday Wisher

This project automates sending birthday emails. It introduces two crucial standard libraries: `smtplib` for sending emails and `datetime` for handling dates and times.

### Key Concepts

1. **`datetime` Module:**
   - A built-in Python module to work with dates and times.
   - `datetime.now()` : Gets the current date and time.
   - `now.weekday()` : Returns the day of the week as an integer (Monday is 0, Sunday is 6).
   - `now.month` , `now.day` : Access specific parts of the date.
   - This is essential for checking if today is someone's birthday.
2. **`smtplib` Module:**
   - **S**imple **M**ail **T**ransfer **P**rotocol library. It defines how to send emails between servers.
   - `smtplib.SMTP()` : Creates a connection object. You need the SMTP server address of your email provider (e.g., `smtp.gmail.com` ).
   - `connection.starttls()` : **T**ransport **L**ayer **S**ecurity. This encrypts the connection, making it secure. It's a mandatory step.
   - `connection.login(user=my_email, password=my_password)` : Logs you into your email account. **Important:** For services like Gmail, you must use an "App Password," not your regular login password, for security reasons.
   - `connection.sendmail(from_addr, to_addrs, msg)` : Sends the email. The message should be formatted correctly, including `Subject:` and the body.
   - `connection.close()` : Closes the connection.

## Code Implementation Snippet

```
import smtplib
import datetime as dt
import random

MY_EMAIL = "your_email@gmail.com"
MY_PASSWORD = "your_app_password" # Use an app-specific password

now = dt.datetime.now()
today_tuple = (now.month, now.day)

# Assuming birthdays.csv has columns: name,email,year,month,day
import pandas
data = pandas.read_csv("birthdays.csv")
birthdays_dict = {(data_row["month"], data_row["day"]): data_row for (index, data_row) in data.:

if today_tuple in birthdays_dict:
    birthday_person = birthdays_dict[today_tuple]
    file_path = f"letter_templates/letter_{random.randint(1,3)}.txt"
    with open(file_path) as letter_file:
        contents = letter_file.read()
        contents = contents.replace("[NAME]", birthday_person["name"])

    with smtplib.SMTP("smtp.gmail.com", port=587) as connection:
        connection.starttls()
        connection.login(MY_EMAIL, MY_PASSWORD)
        connection.sendmail(
            from_addr=MY_EMAIL,
            to_addrs=birthday_person["email"],
            msg=f"Subject:Happy Birthday!\n\n{contents}"
        )
```

## Day 33: API Endpoints & ISS Overhead Notifier

This day marks the transition to working with external data via APIs. An API is a way for different software applications to communicate with each other.

## Key Concepts

1. **What is an API?**
    - **A**pplication **P**rogramming **I**nterface. It's a set of rules and definitions that allows one application to request services or data from another.
    - Think of it like a waiter in a restaurant. You (your program) don't go into the kitchen (the server/database). You give your order (an API request) to the waiter (the API), who brings you your food (the data/response).
2. **API Endpoint:** A specific URL where an API can be accessed. For example, `http://api.open-notify.org/iss-now.json` is the endpoint to get the current location of the International Space Station.
3. **`requests` Library:** The standard Python library for making HTTP requests.
    - `requests.get(url)` : Makes a GET request to the specified URL to retrieve data.
    - **Response Object:** The result of a request. It contains the status code, data, headers, etc.
    - `response.status_code` : A number indicating the result. `200` means success. `404` means not found. `401` means not authorized.
    - `response.raise_for_status()` : This is a great practice. It will automatically raise an exception if the request was unsuccessful (i.e., status code was not 200).
    - `response.json()` : If the API returns data in JSON format (which is very common), this method will automatically parse it into a Python dictionary.

## Code Implementation Snippet (ISS Notifier)

```python
import requests
from datetime import datetime
import smtplib
import time

MY_LAT = 51.507351 # Your latitude
MY_LONG = -0.127758 # Your longitude

def is_iss_overhead():
    response = requests.get(url="http://api.open-notify.org/iss-now.json")
    response.raise_for_status()
    data = response.json()

    iss_latitude = float(data["iss_position"]["latitude"])
    iss_longitude = float(data["iss_position"]["longitude"])

    # Your position is within +5 or -5 degrees of the ISS position.
    if MY_LAT-5 <= iss_latitude <= MY_LAT+5 and MY_LONG-5 <= iss_longitude <= MY_LONG+5:
        return True
    return False

def is_night():
    parameters = {
        "lat": MY_LAT,
        "lng": MY_LONG,
        "formatted": 0, # Get time in 24h format
    }
    response = requests.get("https://api.sunrise-sunset.org/json", params=parameters)
    response.raise_for_status()
    data = response.json()
    sunrise = int(data["results"]["sunrise"].split("T")[1].split(":")[0])
    sunset = int(data["results"]["sunset"].split("T")[1].split(":")[0])

    time_now = datetime.now().hour

    if time_now >= sunset or time_now <= sunrise:
        return True # It's dark
    return False

# Main loop to run the check
while True:
    time.sleep(60) # Wait 60 seconds between checks
    if is_iss_overhead() and is_night():
        # Code to send an email (from Day 32)
        print("Look up!")
```

## Day 34: API Practice - GUI Quiz App

This project combines the skills from Day 31 (Tkinter) and Day 33 (APIs) to create a quiz application that fetches questions from a public API.

### Key Concepts

1. **API Parameters:**
   - Many APIs allow you to customize the request by adding parameters to the URL. For example, you might request 10 questions of a specific category and difficulty.
   - The `requests` library makes this easy: you pass a dictionary of parameters to the `params` argument in `requests.get()`.
   - `parameters = {"amount": 10, "type": "boolean"}`
   - `response = requests.get(url="https://opentdb.com/api.php", params=parameters)`
2. **Type Hinting & Data Classes:**
   - As applications get more complex, using type hints ( `question_text: str` ) improves code readability and helps catch errors.
   - This is a step towards using tools like **Pydantic** (mentioned in your NEETPrepGPT plan) which uses type hints to perform data validation, parsing, and serialization. It ensures the data you get from an API matches the structure you expect.
3. **HTML Character Entities:**
   - Data from APIs often contains HTML character entities (e.g., `&quot;` for a quote mark, `&#039;` for an apostrophe).
   - The built-in `html` module can be used to decode these: `html.unescape(text)`.
4. **Structuring with Classes:**
   - The project is structured into multiple classes to separate concerns:
     - `Question` : A simple data model for a single question (text and answer).
     - `QuizBrain` : The logic engine. It keeps track of the score, the current question number, and checks answers.
     - `UI` : The Tkinter class responsible for displaying the question, score, and buttons. This separation makes the code much cleaner and easier to manage.

## Code Implementation Snippet (UI Class)

```python
from tkinter import *
from quiz_brain import QuizBrain
import html

THEME_COLOR = "#375362"

class QuizInterface:
    def __init__(self, quiz_brain: QuizBrain):
        self.quiz = quiz_brain
        self.window = Tk()
        self.window.title("Quizzler")
        self.window.config(padx=20, pady=20, bg=THEME_COLOR)

        # ... (score label setup) ...

        self.canvas = Canvas(width=300, height=250, bg="white")
        self.question_text = self.canvas.create_text(
            150, 125,
            width=280, # Text wrapping
            text="Some Question Text",
            fill=THEME_COLOR,
            font=("Arial", 20, "italic")
        )
        self.canvas.grid(row=1, column=0, columnspan=2, pady=50)

        # ... (button setup) ...

        self.get_next_question()
        self.window.mainloop()

    def get_next_question(self):
        self.canvas.config(bg="white")
        if self.quiz.still_has_questions():
            self.score_label.config(text=f"Score: {self.quiz.score}")
            q_text = self.quiz.next_question()
            self.canvas.itemconfig(self.question_text, text=html.unescape(q_text))
        else:
            self.canvas.itemconfig(self.question_text, text="You've reached the end of the quiz
            # Disable buttons

    def true_pressed(self):
        self.give_feedback(self.quiz.check_answer("True"))

    def false_pressed(self):
        self.give_feedback(self.quiz.check_answer("False"))

    def give_feedback(self, is_right):
        if is_right:
            self.canvas.config(bg="green")
        else:
            self.canvas.config(bg="red")
        self.window.after(1000, self.get_next_question)
```

## Day 35: API Keys, Authentication & Environment Variables

This project, a "Rain Alert" app, introduces secure API practices by using authentication and environment variables to protect sensitive information like API keys.

### Key Concepts

1. **API Authentication & API Keys:**
    - Most powerful APIs require you to authenticate to track usage, prevent abuse, and sometimes charge for services.
    - The most common method is an **API Key**, which is a unique string of characters you include in your request to identify yourself.
    - This is often passed as a URL parameter (e.g., `&appid=YOUR_API_KEY`).
2. **Environment Variables:**
    - **NEVER** hard-code sensitive information like API keys, passwords, or tokens directly into your source code. If you upload your code to a public repository like GitHub, your keys will be exposed.
    - **Environment Variables** are variables stored outside your program, in the operating system. Your program can then read these variables at runtime.
    - Python's `os` module is used to access them: `api_key = os.environ.get("OWM_API_KEY")`.
    - This is a fundamental practice for production-grade applications like your NEETPrepGPT project.
3. **Twilio API for Sending SMS:**
    - An example of a service API that allows you to programmatically send and receive SMS messages.
    - It requires an Account SID and an Auth Token for authentication.

## Code Implementation Snippet

```python
import requests
import os
from twilio.rest import Client # Example SMS library

# --- Getting Environment Variables ---
# In your terminal (for Linux/Mac): export OWM_API_KEY="your_key_here"
# In your terminal (for Windows): set OWM_API_KEY="your_key_here"
# Or use a .env file and a library like python-dotenv

OWM_Endpoint = "https://api.openweathermap.org/data/2.5/onecall"
api_key = os.environ.get("OWM_API_KEY")
account_sid = os.environ.get("TWILIO_ACCOUNT_SID")
auth_token = os.environ.get("TWILIO_AUTH_TOKEN")



weather_params = {
    "lat": 28.6667, # Your location
    "lon": 77.2167,
    "appid": api_key,
    "exclude": "current,minutely,daily" # Get only hourly forecast
}

response = requests.get(OWM_Endpoint, params=weather_params)
response.raise_for_status()
weather_data = response.json()

# Slice to get the next 12 hours of weather data
weather_slice = weather_data["hourly"][:12]

will_rain = False
for hour_data in weather_slice:
    condition_code = hour_data["weather"][0]["id"]
    if int(condition_code) < 700: # Codes below 700 indicate some form of precipitation
        will_rain = True

if will_rain:
    client = Client(account_sid, auth_token)
    message = client.messages.create(
        body="It's going to rain today. Remember to bring an ☂️",
        from_='+15017122661', # Your Twilio phone number
        to='+911234567890' # Your verified phone number
    )
    print(message.status)
```

## Day 36: Stock Trading News Alert Project

This project integrates two different APIs: one for stock price data (Alpha Vantage) and another for news (News API). It demonstrates how to combine and process data from multiple sources to create a useful alert.

### Key Concepts

1. **Chaining API Calls:** The logic is sequential:
   - **Step 1:** Call the Stock API to get recent price data.
   - **Step 2:** Analyze the data. Calculate the percentage difference between the last two days.
   - **Step 3 (Conditional):** If the percentage change is significant (e.g., > 5%), then proceed to call the News API.
   - **Step 4:** Send an alert (SMS or email) with the stock change and the top 3 related news headlines.
2. **Data Slicing and Manipulation:**
   - The stock API returns a time series of data. You need to access the most recent two days.
   - Python's list slicing ( `data_list[0:2]` ) is perfect for this.
   - Converting string values for prices to floats ( `float()` ) is necessary for calculations.

## Code Implementation Snippet (Logic)

```
import requests
import os

STOCK_NAME = "TSLA"
COMPANY_NAME = "Tesla Inc"

STOCK_ENDPOINT = "https://www.alphavantage.co/query"
NEWS_ENDPOINT = "https://newsapi.org/v2/everything"

STOCK_API_KEY = os.environ.get("STOCK_API_KEY")
NEWS_API_KEY = os.environ.get("NEWS_API_KEY")

# --- Step 1 & 2: Get stock data and calculate difference ---
stock_params = {
    "function": "TIME_SERIES_DAILY",
    "symbol": STOCK_NAME,
    "apikey": STOCK_API_KEY
}
response = requests.get(STOCK_ENDPOINT, params=stock_params)
data = response.json()["Time Series (Daily)"]
data_list = [value for (key, value) in data.items()]
yesterday_data = data_list[0]
yesterday_closing_price = float(yesterday_data["4. close"])

day_before_yesterday_data = data_list[1]
day_before_yesterday_closing_price = float(day_before_yesterday_data["4. close"])

difference = yesterday_closing_price - day_before_yesterday_closing_price
up_down = " ▲ " if difference > 0 else " ▼ "
diff_percent = round((difference / yesterday_closing_price) * 100)

# --- Step 3: If difference is significant, get news ---
if abs(diff_percent) > 5: # Check for a 5% or greater change
    news_params = {
        "apiKey": NEWS_API_KEY,
        "qInTitle": COMPANY_NAME, # Search for company name in title
    }
    news_response = requests.get(NEWS_ENDPOINT, params=news_params)
    articles = news_response.json()["articles"]
    three_articles = articles[:3] # Get first 3 articles
```

```
# --- Step 4: Format and send alerts ---
formatted_articles = [f"{STOCK_NAME}: {up_down}{diff_percent}%\nHeadline: {article['title']]

# Send each article as a separate message/email
for article in formatted_articles:
    # Code to send SMS/email goes here
    print(article)
```

## Day 37: Habit Tracker with Pixela API (POST, PUT, DELETE)

This project moves beyond just fetching data ( `GET` ) to modifying data on a server using more advanced HTTP requests: `POST` , `PUT` , and `DELETE` .

### Key Concepts

1. **More HTTP Verbs:**
   - **`GET`** : Retrieve data from a server.
   - **`POST`** : Send data to a server to create a new resource (e.g., create a user, post a new data point).
   - **`PUT`** : Update an existing resource on the server (e.g., change the color of a graph).
   - **`DELETE`** : Remove a resource from the server.
2. **`requests` Library for Advanced Requests:**
   - `requests.post(url, json=payload, headers=headers)`
   - `requests.put(url, json=payload, headers=headers)`
   - `requests.delete(url, headers=headers)`
   - **`json` parameter:** Used to send a Python dictionary as the JSON body of the request.
   - **`headers` parameter:** Used to send additional information, like authentication tokens. Some APIs require tokens to be sent in the request header for security.

## Code Implementation Snippets

```python
import requests
from datetime import datetime
import os

USERNAME = "your-username"
TOKEN = os.environ.get("PIXELA_TOKEN") # Stored as environment variable
GRAPH_ID = "graph1"

pixela_endpoint = "https://pixe.la/v1/users"
user_params = {
    "token": TOKEN,
    "username": USERNAME,
    "agreeTermsOfService": "yes",
    "notMinor": "yes",
}

# --- 1. POST - Create a user (only need to run once) ---
# response = requests.post(url=pixela_endpoint, json=user_params)
# print(response.text)

# --- 2. POST - Create a graph definition ---
graph_endpoint = f"{pixela_endpoint}/{USERNAME}/graphs"
graph_config = {
    "id": GRAPH_ID,
    "name": "Cycling Graph",
    "unit": "Km",
    "type": "float",
    "color": "sora"
}
headers = {
    "X-USER-TOKEN": TOKEN
}
# response = requests.post(url=graph_endpoint, json=graph_config, headers=headers)
# print(response.text)

# --- 3. POST - Post a value (a pixel) to the graph ---
pixel_creation_endpoint = f"{pixela_endpoint}/{USERNAME}/graphs/{GRAPH_ID}"
today = datetime.now()
pixel_data = {
    "date": today.strftime("%Y%m%d"), # Format date as YYYYMMDD
    "quantity": "15.5",
}
response = requests.post(url=pixel_creation_endpoint, json=pixel_data, headers=headers)
print(response.text)

# --- 4. PUT - Update a pixel ---
update_endpoint = f"{pixela_endpoint}/{USERNAME}/graphs/{GRAPH_ID}/{today.strftime('%Y%m%d')}"
new_pixel_data = {
    "quantity": "10.2"
}
# response = requests.put(url=update_endpoint, json=new_pixel_data, headers=headers)
# print(response.text)


# --- 5. DELETE - Delete a pixel ---
delete_endpoint = f"{pixela_endpoint}/{USERNAME}/graphs/{GRAPH_ID}/{today.strftime('%Y%m%d')}"
# response = requests.delete(url=delete_endpoint, headers=headers)
# print(response.text)
```

# Day 38: Workout Tracking with Google Sheets

This project uses a Natural Language Processing (NLP) API from Nutritionix to interpret plain English workout descriptions and then posts the structured data to a Google Sheet via the Sheety API.

## Key Concepts

1. **Natural Language Processing (NLP) APIs:**
   - These APIs are trained to understand and process human language.
   - In this case, you send a sentence like `"ran 3 miles and swam for 20 minutes"` , and the API returns structured data like exercises, durations, and calories burned.
2. **Sheety API:**
   - A simple service that turns any Google Sheet into an API.
   - This is an example of an "API-as-a-service" that simplifies integration. You can use `GET` to read rows, `POST` to add a new row, `PUT` to update a row, and `DELETE` to remove one.
3. **Authentication:**
   - **Nutritionix:** Uses an App ID and API Key in the request headers.
   - **Sheety:** Can use "Basic Authentication" or a "Bearer Token," also sent in the headers. Bearer Token auth is common in modern APIs (like the ones used in your NEETPrepGPT project with JWT).

## Code Implementation Snippet

```python
import requests
from datetime import datetime
import os

# --- Nutritionix API Config ---
GENDER = "male"
WEIGHT_KG = 72.5
HEIGHT_CM = 167.6
AGE = 30
APP_ID = os.environ.get("NUTRITIONIX_APP_ID")
API_KEY = os.environ.get("NUTRITIONIX_API_KEY")
exercise_endpoint = "https://trackapi.nutritionix.com/v2/natural/exercise"

# --- Sheety API Config ---
sheet_endpoint = os.environ.get("SHEETY_ENDPOINT")
SHEETY_TOKEN = os.environ.get("SHEETY_TOKEN")

exercise_text = input("Tell me which exercises you did: ")

headers = {
    "x-app-id": APP_ID,
    "x-api-key": API_KEY,
}

parameters = {
    "query": exercise_text,
    "gender": GENDER,
    "weight_kg": WEIGHT_KG,
    "height_cm": HEIGHT_CM,
    "age": AGE,
}

# --- Step 1: Get structured data from Nutritionix ---
response = requests.post(exercise_endpoint, json=parameters, headers=headers)
result = response.json()

# --- Step 2: Post the data to Google Sheets via Sheety ---
today_date = datetime.now().strftime("%d/%m/%Y")
now_time = datetime.now().strftime("%X")

bearer_headers = {
    "Authorization": f"Bearer {SHEETY_TOKEN}"
}

for exercise in result["exercises"]:
    sheet_inputs = {
        "workout": {
            "date": today_date,
            "time": now_time,
            "exercise": exercise["name"].title(),
            "duration": exercise["duration_min"],
            "calories": exercise["nf_calories"]
        }
    }

    sheet_response = requests.post(
        sheet_endpoint,
        json=sheet_inputs,
        headers=bearer_headers
    )
    print(sheet_response.text)
```

## Days 39 & 40: Capstone Project - Flight Deal Finder

This is a major two-part capstone project that brings together everything learned so far: APIs, authentication, data processing, and notifications. The goal is to build a service that checks for cheap flights to destinations you're interested in and notifies you when it finds a deal.

### Key Concepts & Project Structure

The project is broken down into several classes, promoting good **Object-Oriented Programming (OOP)** practices, which is essential for large projects like NEETPrepGPT.

1. **`DataManager` Class (Day 39):**
    - **Responsibility:** Manages all communication with the Google Sheet (via Sheety API).
    - `get_destination_data()` : Reads the data from your sheet (which contains cities you want to fly to and your target price).
    - `update_destination_codes()` : After finding the IATA codes for cities, this method writes them back to the sheet using `PUT` requests.
2. **`FlightSearch` Class (Day 39):**

- **Responsibility:** Manages all communication with the Flight Search API (Tequila by Kiwi.com).
- `get_destination_code(city_name)` : Takes a city name and queries the Tequila API to find its IATA (airport) code (e.g., "Paris" -> "PAR").
- `check_flights(...)` : The main search function. It takes an origin city code, destination city code, and date range, then queries the API for the cheapest flight.

3. **`FlightData` Class (Day 40):**
   - **Responsibility:** A simple data structure class. Its purpose is to structure the flight data found by `FlightSearch` in a clean, organized way. This prevents passing around messy dictionaries.
4. **`NotificationManager` Class (Day 40):**
   - **Responsibility:** Handles sending notifications (SMS via Twilio or Email via `smtplib` ).
   - `send_sms(message)` : Contains the logic to interact with the Twilio API.

## Main Logic Flow ( `main.py` ) (Day 40)

```python
from data_manager import DataManager
from flight_search import FlightSearch
from notification_manager import NotificationManager
from datetime import datetime, timedelta

ORIGIN_CITY_IATA = "LON" # Example: London

# Initialize all the manager classes
data_manager = DataManager()
flight_search = FlightSearch()
notification_manager = NotificationManager()

# 1. Get destination data from Google Sheet
sheet_data = data_manager.get_destination_data()

# 2. Fill in IATA Codes if they are missing in the sheet
if sheet_data[0]["iataCode"] == "":
    for row in sheet_data:
        row["iataCode"] = flight_search.get_destination_code(row["city"])
    data_manager.destination_data = sheet_data
    data_manager.update_destination_codes()

# 3. Search for flights for each destination
tomorrow = datetime.now() + timedelta(days=1)
six_month_from_today = datetime.now() + timedelta(days=(6 * 30))

for destination in sheet_data:
    flight = flight_search.check_flights(
        ORIGIN_CITY_IATA,
        destination["iataCode"],
        from_time=tomorrow,
        to_time=six_month_from_today
    )

    # 4. If a cheap flight is found, send a notification
    if flight is not None and flight.price < destination["lowestPrice"]:
        message = (
            f"Low price alert! Only £{flight.price} to fly from "
            f"{flight.origin_city}-{flight.origin_airport} to "
            f"{flight.destination_city}-{flight.destination_airport}, "
            f"from {flight.out_date} to {flight.return_date}."
        )
        notification_manager.send_sms(message=message)
```

This structured, multi-class approach is highly scalable and maintainable, representing a significant step towards building professional-quality software.