Python project

Github 자동 업로드

https://github.com/changimbap/-HARMAN-python_project

딸깍 눌러조

장준우, 정예찬, 한규혁, 황석현, 황재윤

CONTENTS

주제 선정 이유

CHAPTER 01

1.선정 이유

GitHub

BAEKJOON ONLINE JUDGE

1. 생소한 GitHub 이용 방법

2. 간편한 업로드

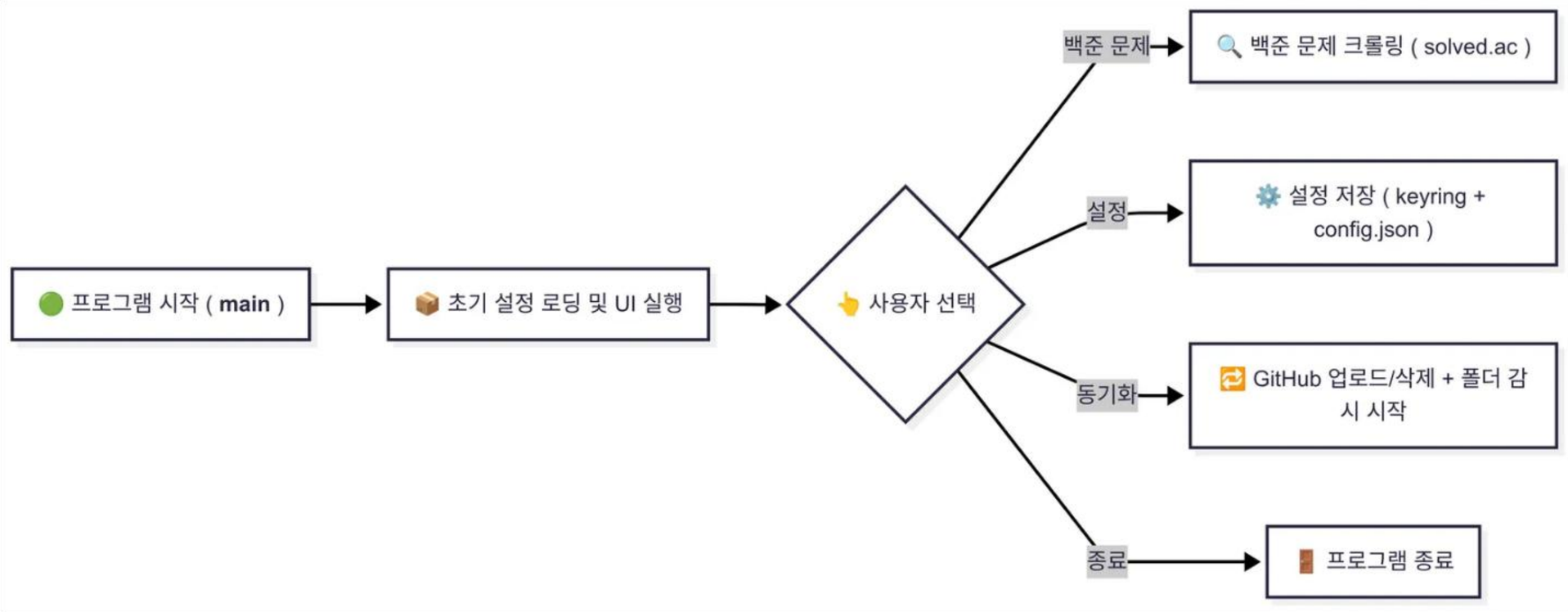
3. 포트폴리오 정리

4. 잔디 쌓기

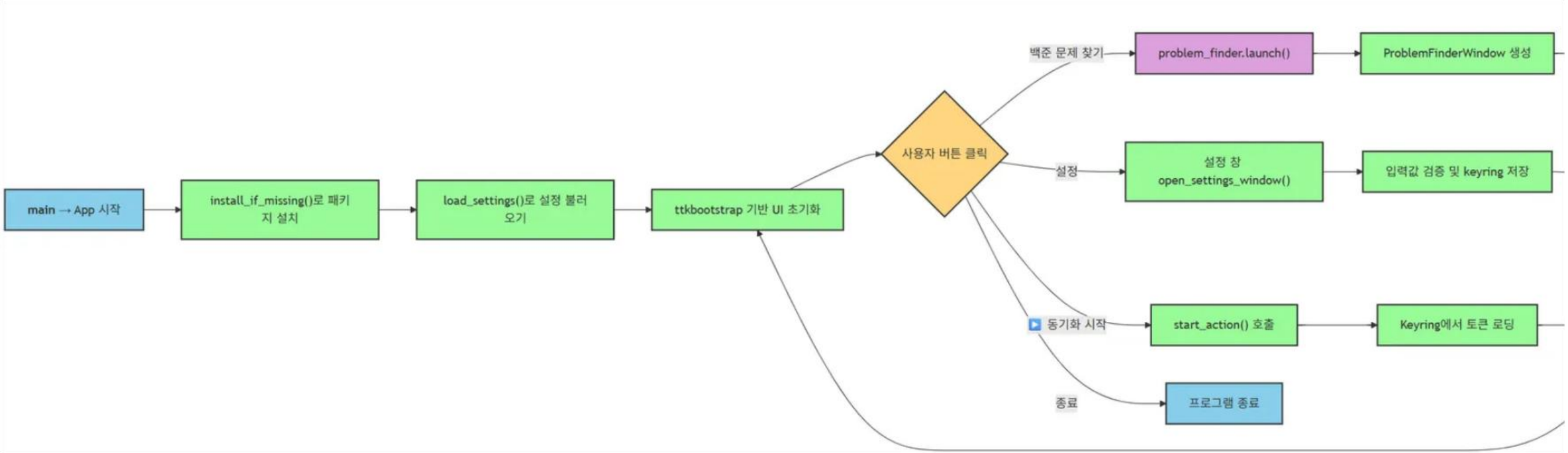
플로우 차트

CHAPTER 02

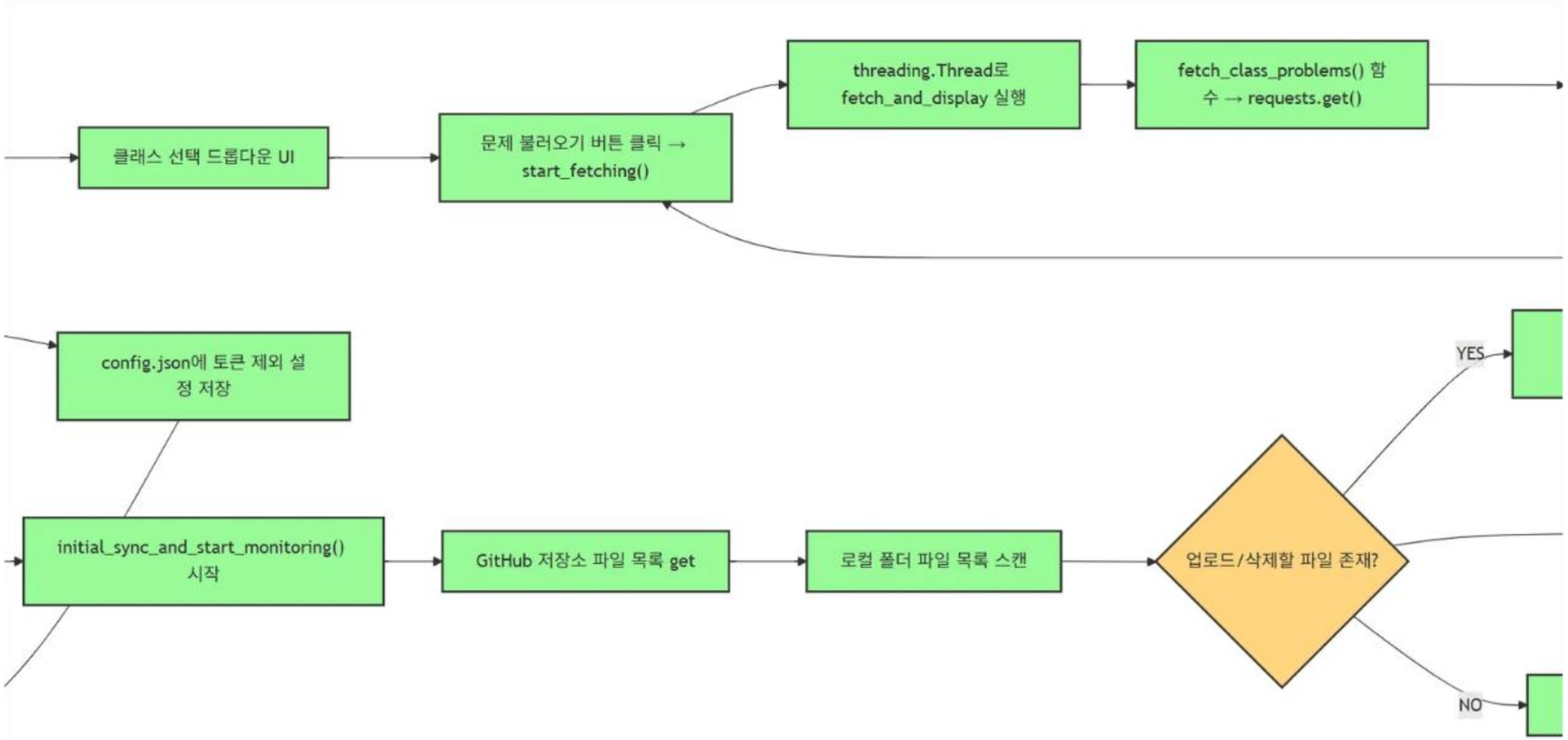
2. 플로우차트


🟢 프로그램 시작 (main)
📦 초기 설정 로딩 및 UI 실행
👆 사용자 선택
백준 문제
🔍 백준 문제 크롤링 (solved.ac)
설정
⚙️ 설정 저장 (keyring + config.json)
동기화
🔁 GitHub 업로드/삭제 + 폴더 감시 시작
종료
🚪 프로그램 종료

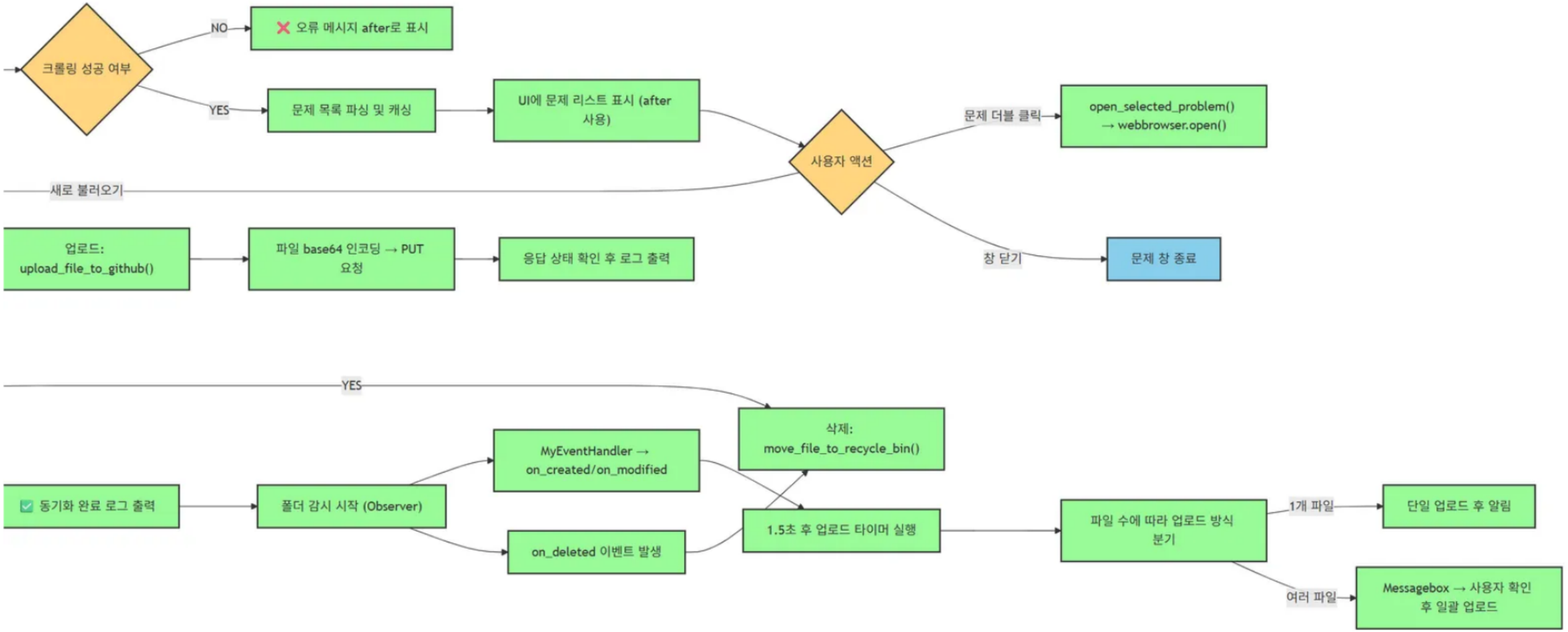
2. 플로우차트


main → App 시작
install_if_missing()로 패키지 설치
load_settings()로 설정 불러오기
ttkbootstrap 기반 UI 초기화
사용자 버튼 클릭
백준 문제 찾기
problem_finder.launch()
ProblemFinderWindow 생성
설정
설정 창 open_settings_window()
입력값 검증 및 keyring 저장
동기화 시작
start_action() 호출
Keyring에서 토큰 로딩
종료
프로그램 종료

2. 플로우차트


클래스 선택 드롭다운 UI
문제 불러오기 버튼 클릭 → start_fetching()
threading.Thread로 fetch_and_display 실행
fetch_class_problems() 함수 → requests.get()
config.json에 토큰 제외 설정 저장
initial_sync_and_start_monitoring() 시작
GitHub 저장소 파일 목록 get
로컬 폴더 파일 목록 스캔
업로드/삭제할 파일 존재?
YES
NO

2. 플로우차트


크롤링 성공 여부
NO
❌ 오류 메시지 after로 표시
YES
문제 목록 파싱 및 캐싱
UI에 문제 리스트 표시 (after 사용)
사용자 액션
문제 더블 클릭
open_selected_problem() → webbrowser.open()
새로 불러오기
창 닫기
문제 창 종료
업로드: upload_file_to_github()
파일 base64 인코딩 → PUT 요청
응답 상태 확인 후 로그 출력
YES
삭제: move_file_to_recycle_bin()
MyEventHandler → on_created/on_modified
☑ 동기화 완료 로그 출력
폴더 감시 시작 (Observer)
on_deleted 이벤트 발생
1.5초 후 업로드 타이머 실행
파일 수에 따라 업로드 방식 분기
1개 파일
단일 업로드 후 알림
여러 파일
Messagebox → 사용자 확인 후 일괄 업로드

개발 내용 및 기술 구현

CHAPTER 03

3. 핵심 코드

① GitHub API를 이용한 파일 생성 및 수정

```python
def upload_file_to_github(local_path, repo_path, settings, log_queue):
    log_queue.put(f"- 처리 대상 (추가/수정): {os.path.basename(local_path)}")
    url =
f"https://api.github.com/repos/{settings['username']}/{settings['repo']}/contents/{repo_path}"
    headers = {"Authorization": f"token {settings['token']}"}
    try:
        with open(local_path, "rb") as file:
            content_encoded = base64.b64encode(file.read()).decode('utf-8')
    except (FileNotFoundError, PermissionError) as e:
        log_queue.put(f"    ❌ 파일 읽기 오류: {e}")
        return
    sha = None # sha : 파일의 고유 식별자 (GitHub)
    try:
        response_get = requests.get(url, headers=headers)
        if response_get.status_code == 200: sha = response_get.json().get('sha')
    except: pass
    data = {"message": f"Sync: Update {repo_path}", "content": content_encoded}
    if sha: data["sha"] = sha
    log_queue.put(f"    🚀 '{repo_path}' 경로로 업로드를 시도합니다...")
    try:
        response_put = requests.put(url, headers=headers, data=json.dumps(data))
        if response_put.status_code in [200, 201]:
            log_queue.put(f"    ✅ '{os.path.basename(local_path)}' 업로드 성공!")
        else:
            log_queue.put(f"    ❌ 업로드 실패! (코드: {response_put.status_code})")
    except Exception as e:
        log_queue.put(f"    ❌ 네트워크 오류 (업로드 중): {e}")
```

3. 핵심 코드

② 초기 동기화

```python
def initial_sync_and_start_monitoring(settings, log_queue, stop_event):
    log_queue.put("🔄 초기 동기화를 시작합니다...")

    remote_files = get_github_repo_file_list(settings, log_queue) # 깃허브 저장소의 파일 목록 확인
    if remote_files is None:
        log_queue.put("초기 동기화 실패. 감시를 시작하지 않습니다.")
        log_queue.put("STOP_MONITORING_UI")
        return
    watch_folder = settings['folder']
    if not os.path.isdir(watch_folder):
        log_queue.put(f"오류: '{watch_folder}'는 유효한 폴더가 아닙니다.")
        log_queue.put("STOP_MONITORING_UI")
        return
    local_files = set()
    for root, _, files in os.walk(watch_folder): # os.walk : 내 컴퓨터 폴더의 목록 확인
        for filename in files:
            local_path = os.path.join(root, filename)
            repo_path = os.path.relpath(local_path, watch_folder).replace("\\", "/")
            local_files.add(repo_path)
    files_to_delete = remote_files - local_files
    files_to_upload = local_files - remote_files

    if not files_to_delete and not files_to_upload:
        log_queue.put("✅ 로컬과 깃허브 저장소가 이미 동기화 상태입니다.")
    else:
        total_tasks = len(files_to_delete) + len(files_to_upload)
        current_task = 0

        for repo_path in files_to_delete:
            move_file_to_recycle_bin(repo_path, settings, log_queue)
        for repo_path in files_to_upload:
```

3. 핵심 코드

③ 삭제 시 /recycle_bin 폴더로 이동

```python
def move_file_to_recycle_bin(repo_path, settings, log_queue):
    # 지정된 경로의 파일을 깃허브의 _recycle_bin 폴더로 이동
    log_queue.put(f"- 처리 대상 (휴지통 이동): {os.path.basename(repo_path)}")

    get_url =
f"https://api.github.com/repos/{settings['username']}/{settings['repo']}/contents/{repo_path}"
    headers = {"Authorization": f"token {settings['token']}"}

    original_content_encoded = None
    original_sha = None
    try:
        response_get = requests.get(get_url, headers=headers)
        if response_get.status_code == 200:
            data = response_get.json()

            # ...
    # ...
    # 3-(2.1). 휴지통에 저장할 새 경로와 파일명을 만들기
    # ...
    # 3-(2.2). 휴지통 경로에 새 파일을 생성
    put_url =
f"https://api.github.com/repos/{settings['username']}/{settings['repo']}/contents/{recycle_bin_path}"
    put_data = {
        "message": f"Recycle: Move {repo_path}",
        "content": original_content_encoded
    }
     # ...
    # 3-(2.3). 휴지통으로 복사가 성공했을 때만 원본 파일을 삭제
    del_url = get_url
    del_data = {"message": f"Sync: Delete {repo_path} (moved to recycle bin)", "sha": original_sha}
    try:
        response_del = requests.delete(del_url, headers=headers, data=json.dumps(del_data))
        if response_del.status_code == 200:
            log_queue.put(f"  ✅ '{os.path.basename(repo_path)}' 휴지통으로 이동 완료!")
       # ...
```

3. 핵심 코드

④ Threading 사용

```python
# 감시 시작 버튼 클릭 시
def start_action(self):
    # ...
    # '초기 동기화 및 감시'라는 무거운 작업을 별도 스레드에서 실행
    threading.Thread(target=initial_sync_and_start_monitoring, ...).start()

# 백준 문제 찾기 버튼 클릭 시
def open_problem_finder_window(self):
    # 독립된 problem_finder 모듈의 launch 함수를 호출하여 새 창을 띄움
    problem_finder.launch(self.root)

# 백그라운드의 결과를 UI에 표시
def check_log_queue(self):
    # 0.1초마다 queue를 확인하여 새로운 메시지가 있으면 로그 창에 업데이트
    while not self.log_queue.empty():
        # ...
    self.root.after(100, self.check_log_queue)
```

3. 핵심 코드

⑤ 보안 강화 : keyring 을 이용한 토큰 관리

```python
import keyring

# 토큰 저장 시 (설정 창)
def save_and_close():
    # ...
    # OS 보안 저장소에 "서비스이름", "사용자이름", "토큰"을 저장
    keyring.set_password("github_auto_uploader", new_settings["username"], new_settings["token"])

    # 실제 파일에는 토큰을 빈 값으로 저장
    settings_for_file["token"] = ""
    save_settings(settings_for_file)

# 토큰 로드 시 (감시 시작)
def start_action(self):
    # ...
    # 저장된 사용자 이름을 'key'로 사용하여 OS에서 토큰을 불러옴
    token = keyring.get_password("github_auto_uploader", self.settings["username"])
    active_settings["token"] = token
    # ...
```

3. 핵심 코드

⑥ 웹 크롤링 : solved.ac 웹페이지에서 문제 데이터 추출

```python
def fetch_class_problems(class_num: str) -> list[tuple[str, str]]:
    """solved.ac 클래스 문제 목록을 크롤링하는 함수"""
    url = f"https://solved.ac/class/{class_num}"
    headers = {"User-Agent": "Mozilla/5.0"}
    res = requests.get(url, headers=headers)
    res.raise_for_status() # 웹 요청이 실패하면 에러 발생

    # 'lxml' 파서를 사용해 HTML을 분석 가능한 객체로 변환
    soup = BeautifulSoup(res.text, "lxml")

    problems = []
    # CSS 선택자를 이용해 문제 테이블(tbody)를 정확히 찾아냄
    problem_table = soup.select_one("table tbody")
    if not problem_table: return []

    # 테이블의 모든 행(tr)을 순회
    for row in problem_table.find_all("tr"):
        cols = row.find_all("td") # 각 행의 모든 열(td)을 찾음
        if len(cols) >= 2:
            problem_id = cols[0].text.strip() # 첫 번째 열에서 문제 번호 추출
            title = cols[1].text.strip()      # 두 번째 열에서 문제 제목 추출
            problems.append((problem_id, title))
    return problems
```

트러블 슈팅

CHAPTER 04

4. 트러블 슈팅

GUI

1. 문제 현상
초기에는 코드를 실행하면 웹서버가 구동되고, 사용자는 브라우저를 직접 열어 주소를 입력해야 프로그램을 사용할 수 있는 구조를 구성함.
이로 인해 프로그램 실행 절차가 번거로워져 사용자가 매번 주소 입력 과정을 거쳐야 했음.

2. 원인
Flask는 웹 애플리케이션용 프레임워크로 설계되어 있어, 로컬에서 바로 실행해 간편하게 쓰기에는 적합하지 않음.

3. 해결책
Python 내장 GUI인 tkinter로 UI를 재구성하여 웹서버 없이도 .py 파일 하나로 바로 실행할 수 있도록 변경함.

4. 트러블 슈팅

백준 크롤링

1. 문제 현상
백준 문제 데이터를 flask에서 api로 DB 데이터를 가져오는 과정에서 일부 정보(문제 본문 등)가 불완전하거나 누락되어 있음.
문제 번호가 뒤죽박죽 (ex A+B를 출력하시오 문제가 1000번 문제) ➔ 난이도 선정 과정에서 어려움이 있을 것이라 생각함.

2. 원인
백준 자체의 문제 내용과 문제 번호를 보기 편하게 하기 위해선 크롤링 없이는 힘듦.

3. 해결책
API 방식에서 크롤링 방식으로 전환함.
requests + beautifulsoup 을 이용해 백준 사이트에서 직접 HTML 파싱하여 데이터 수집함.
문제 상세 내용을 보기 위해 브라우저 자동 연결 기능 탑재함.

4. 트러블 슈팅

GitHub

1. 문제 현상
파일 삭제 시 GitHub Repository의 파일이 바로 삭제가 되는 상황이 발생함.

2. 원인
원래 의도했던 컨셉에 맞지만, 안전 장치 없이 파일이 바로 삭제 되는 것이 위험하다고 판단함.

3. 해결책
파일을 삭제할 때 Repository 내에 /recycle_bin (휴지통) 경로를 만들어 그곳으로 넣을 수 있게 수정함.

4. 트러블 슈팅

GitHub

1. 문제 현상
파일 업로드/알림이 두 번씩 발생함.

2. 원인
파일 저장 시, on_created(생성)와 on_modified(수정) 이벤트가 거의 동시에 발생하여 각각 업로드를 시도함.

3. 해결책
파일 하나하나 디바운싱(Debouncing) 타이머를 도입함.
짧은 시간 내에 발생하는 모든 이벤트를 담아두었다가, 이벤트들이 끝난 뒤 한 번에 모아서 처리하도록 로직 수정.

4. 트러블 슈팅

GitHub

1. 문제 현상
로컬 폴더를 비우면 다음 실행 시 404 오류가 발생함.

2. 원인
저장소가 완전히 비워지면 기본 브랜치(main/master)가 사라져, 다음 동기화 시 파일 목록을 조회하지 못 함.

3. 해결책
초기 동기화 함수가 404 오류를 받았을 때, 이를 치명적 오류가 아닌 **'파일이 없는 정상 상태'**로 인지하고 빈 목록을 반환하도록 수정함.

시연 영상

CHAPTER 05

시원 엄살

CHAPTER 05

추가 개선사항

CHAPTER 06

6. 추가 개선 사항

1. .exe 배포

실행 파일(.exe)로 변환하여 사용이 더 용이하게 한다.

2. 시간 설정

설정한 시간에 GitHub에 자동으로 업로드 되도록 한다.

3. 문제 추천

백준 문제를 추천해준다.

프로젝트 소감

장O우

팀원마다 다른 접근 방식이 인상적이었다.
개발하면서 기능에 중심을 뒀지만 코드의 복잡성과
유지보수에서 다양한 의견을 듣고 복잡한 게 항상
좋은 것은 아니라는 것을 느꼈다.

정O찬

개인적으로 Github 관리는 막연히 어렵다고만
생각하며 주저했습니다. 하지만 자동 업로드
프로그램을 직접 개발하며 GitHub를 활용하는
첫걸음을 뗄 수 있어 매우 뿌듯한 경험이었습니다.

한O혁

팀원들과 협업하며 문제를 해결해 가는 과정이 가장
값진 경험이었고 프로그램을 완성해 가면서 성취감을
느꼈다.

황O현

수업시간에 배운 개념을 복습하고 코드를 살펴보면서 다시
정리할 수 있었다. 또, 문제점을 발견했을 때 해결 방안을
찾아보고 분석하며 공부할 수 있는 기회가 되었다.

황O윤

반복적이고 번거로운 작업을 자동화하면서 더 많은
사람들이 쉽게 사용할 수 있도록 고민한 경험이 도움이
되었다.

THANK YOU!
감사합니다!

딸각 눌러조