ChatGPT

Python 콘솔 기반 치환 도구 PRD

1. 프로그램 개요 (목적 및 기능 요약)

이 Python 콘솔 프로그램은 **대량의 문자열 치환 작업을 자동화**하기 위해 설계되었습니다. 사용자로부터 메뉴를 통해 입력을 받아 세 가지 주요 기능을 수행합니다:

- **YAML 생성:** 사용자 제공 엑셀 파일(원본 파일 경로, 대상 파일 경로, 치환 전/후 문자열 쌍 정보)을 읽어 작업 정의 YAML 파일을 생성합니다. 여러 치환 작업을 **job 세트** 단위로 묶어 YAML로 저장합니다.
- **미리보기:** YAML에 정의된 각 작업에 따라 원본 파일 내용을 메모리상에서 치환하고, **치환 전후의 차이(diff)**를 콘솔에 출력합니다. 사용자는 이 **미리보기(diff)**를 통해 실제 적용 결과를 사전에 확인할 수 있습니다.
- **실행:** YAML 정의에 따라 실제 원본 파일들을 읽어 지정된 대상 파일로 복사하면서 문자열 치환을 수행합니다. 모든 치환 결과에 대한 **로그와 요약 정보를 텍스트 파일**로 저장하고, 치환된 파일들을 생성합니다.

이 프로그램의 목적은 다수의 파일에서 여러 문자열을 일괄적으로 치환해야 하는 작업을 편리하게 수행하고, 실행 전에 결과를 검토할 수 있도록 하는 것입니다. 인코딩은 UTF-8, 줄바꿈은 LF를 유지하여 원본 파일의 형식을 해치지 않고 치환을 적용합니다.

2. 사용자 인터페이스 흐름

프로그램은 **콘솔 메뉴 방식의 UI**를 제공합니다. 프로그램을 실행하면 다음과 같은 메뉴가 콘솔에 표시됩니다:

```
1. YAML 생성 (엑셀 -> YAML)
2. 미리보기 (YAML 기반 diff 출력)
3. 실행 (치환 적용 및 로그 저장)
0. 종료
```

사용자는 원하는 기능에 해당하는 번호를 입력하여 선택합니다. 각 메뉴의 동작 흐름은 다음과 같습니다:

- **1. YAML 생성:** 사용자가 `1`을 입력하면 프로그램이 엑셀 파일 경로를 묻습니다 (예: `input.xlsx`). 입력받은 경로의 엑셀 파일을 읽어 작업 정의를 추출한 후, 출력할 YAML 파일 경로를 묻습니다 (예: `jobs.yaml`). 프로그램은 openpyxl 라이브러리를 사용하여 엑셀 데이터를 읽고, PyYAML 등을 이용해 YAML 파일을 생성합니다. 완료 후 "YAML 파일 생성 완료" 등의 메시지를 출력하고 다시 메뉴를 표시합니다.

- **2. 미리보기:** 사용자가 `2`를 선택하면 프로그램이 YAML 파일 경로를 묻습니다 (예: `jobs.yaml`). 해당 YAML을 파싱하여 각 작업에 대해 원본 파일을 읽고, 정의된 문자열 치환을 **메모리상 적용**합니다. 그 결과를 원본과 비교하여 **diff 형식**으로 콘솔에 출력합니다. Python 표준 라이브러리 `difflib`를 사용하여 unified diff 등의 형식으로 변화 내용을 생성하며, 변경 전 라인은 `-`, 변경 후 라인은 `+`로 표시합니다. 예를 들어 치환 전후 내용 차이를 unified diff로 출력하면 다음과 같습니다:

  ```
  --- example.xml (원본)
  +++ example.xml (치환 적용 미리보기)
  @@
  ```

```
- <name>apple</name>
+ <name>banana</name>
```

각 작업(job)별로 구분하여 diff 결과를 출력하여 사용자에게 치환 결과를 미리 검토할 수 있게 합니다. 모든 diff 출력이 끝나면 메뉴를 다시 표시합니다.

- **3. 실행:** 사용자가 `3`을 선택하면 프로그램이 YAML 파일 경로를 묻습니다 (예: `jobs.yaml`). 지정된 YAML을 읽어 순차적으로 각 작업을 실행합니다. 실행 과정에서는:

- 원본 파일을 읽어 메모리상으로 내용을 가져옵니다.
- 지정된 대상 파일 경로로 원본 파일의 내용을 복사한 뒤 (또는 내용을 바로 복사하여) **지정된 문자열 치환을 모두 적용**합니다.
- 치환된 결과를 대상 파일로 저장하여, 원본 파일을 건드리지 않고 새로운 파일에 결과를 기록합니다.
- **로그 파일**과 **요약 파일**을 생성합니다. 로그 파일에는 각 파일에서 수행된 상세 치환 내역이나 치환 횟수 등이 기록되고, 요약 파일에는 작업별로 몇 건의 치환이 이루어졌는지 등의 총괄 결과가 정리됩니다.

모든 작업이 완료되면 "치환 작업이 완료되었습니다" 등의 메시지를 출력하고 메뉴를 다시 표시합니다 (또는 프로그램을 종료합니다).

- **0. 종료:** 사용자가 `0` 또는 지정된 종료 명령을 입력하면 프로그램이 종료됩니다.

사용자는 위 메뉴를 **반복적으로 실행**할 수 있습니다. 예를 들어 YAML을 생성한 후 바로 미리보기를 하거나, 미리보기 후 실행을 수행하는 등 순차적 이용이 가능합니다. 각 단계에서 잘못된 입력(예: 존재하지 않는 파일 경로 등)에 대한 예외 처리가 되어 있으며, 에러 메시지를 출력하고 필요한 경우 재입력을 요청합니다.

3. 주요 클래스 및 함수 설명

이 프로그램은 단일 파일로 구현되며, 다음과 같은 **주요 함수와 간단한 데이터 구조**를 사용합니다 (복잡도가 낮아 별도의 클래스보다는 함수 중심으로 설계):

- `generate_yaml_from_excel(excel_path, yaml_path)` – 엑셀 파일을 읽어 YAML 파일을 생성하는 함수입니다. 엑셀은 `openpyxl`을 사용하여 읽습니다. 엑셀의 각 행은 하나의 치환 정의를 나타내며, 동일한 원본-대상 파일에 대한 여러 행을 하나의 작업(job)으로 묶습니다. 이 함수는 모든 행을 읽어 `(source_file, destination_file)`을 키로 그룹화한 후, 각 그룹마다 `{'source': ..., 'destination': ..., 'replacements': [...]}` 구조의 딕셔너리를 만들고, 이를 리스트로 YAML 파일에 덤프(dump)합니다. PyYAML 라이브러리를 통해 Python의 딕셔너리 리스트를 YAML 형식으로 쉽게 출력할 수 있습니다.

- `preview_diff(yaml_path)` – YAML 파일에 정의된 작업들을 순차적으로 읽어 미리보기를 수행하는 함수입니다. `yaml.safe_load()`로 YAML을 파싱하여 작업 리스트를 얻은 뒤, 각 작업에 대해:

- 원본 파일을 읽어 텍스트를 메모리에 로드합니다.
- 텍스트 복사본을 만들고, 정의된 치환 문자열들을 모두 적용합니다 (Python의 `str.replace()` 또는 정규식 적용 등으로 수행).
- Python의 `difflib` 모듈을 사용해 원본 텍스트와 치환 적용 텍스트의 **차이(diff)**를 계산합니다. `difflib.unified_diff()` 함수를 사용하여 Unified Diff 형식의 결과를 얻으며, 파일명과 몇 줄의 문맥을 포함한 사람이 읽기 좋은 diff를 생성합니다.
- diff 결과를 콘솔에 출력합니다. 각 작업별로 `--- 파일명 (before)`와 `+++ 파일명 (after)` 헤더를 표시하고 변경 내용을 보여줍니다. 이 과정은 파일에 쓰기는 하지 않고 **화면 출력만** 수행합니다.

- `execute_replacements(yaml_path, log_path, summary_path)` – YAML에 명시된 작업들을 실제로 실행하는 함수입니다. `yaml.safe_load()`로 작업 리스트를 가져와 순차 처리하며:

  - 원본 파일을 읽어 텍스트를 가져옵니다.
  - 대상 파일 경로에 원본 파일의 내용을 복사하되, 동시에 지정된 모든 문자열 치환을 적용하여 **새로운 텍스트**를 만듭니다.
  - 치환이 적용된 새 텍스트를 대상 파일로 저장합니다 (필요 시 디렉토리 생성 등을 처리).
  - 각 치환에 대해 **치환 횟수**를 계산합니다 (예: `original_text.count(from_str)` 등으로 원본에서의 발생 횟수를 세어 치환된 개수 파악).
  - 로그 파일에 상세 내역을 기록합니다. 예를 들어 `source_file -> destination_file: 'apple' -> 'banana' 5건 치환` 같은 형식으로 각 치환별로 기록하거나, 또는 변경된 라인 정보를 기록할 수 있습니다. 로그 파일에는 시간, 파일 경로, 치환 내용 등의 정보도 포함됩니다.
  - 요약 파일에는 작업 단위별 요약 정보를 기록합니다. 예를 들어 각 작업마다 `"파일 A -> B: X개 문자열 치환"`과 같이 총 치환된 건수를 요약하거나, 전체 처리한 파일 수와 전체 치환 건수 합계를 기록합니다.

모든 작업을 처리한 후 파일들을 닫고 종료합니다. 작업 진행 중 오류가 발생하면 해당 내용을 로그에 남기며, 다음 작업을 이어서 진행하거나 전체 중단 여부는 요구사항에 따라 결정합니다 (본 설계에서는 치환 실패 시 프로그램이 예외를 던지고 종료하도록 가정하고, 필요한 경우 try-except으로 처리).

- **기타 도우미 함수:**
- `load_jobs_from_yaml(yaml_path)` : YAML 파일을 읽어 job 리스트(`jobs`)를 반환합니다. (`yaml.safe_load` 사용)
- `save_jobs_to_yaml(jobs, yaml_path)` : job 리스트를 YAML 파일로 저장합니다 (`yaml.dump` 사용).
- `apply_replacements(text, replacements)` : 주어진 텍스트 문자열에 대하여 `[{"from": .., "to": ..}, ...]` 리스트를 순차 적용하여 치환된 새로운 문자열을 반환합니다.
- `compute_diff(original_text, modified_text, fromfile, tofile)` : 두 버전의 텍스트에 대해 difflib으로 unified diff 결과를 리스트로 반환합니다.
- `log_change(log_file, message)` : 로그 파일에 문자열을 기록합니다 (줄 단위 기록).

프로그램은 단일 파일로 구성되지만, 위와 같은 함수 단위로 논리를 분리하여 **유지보수와 확장**을 용이하게 합니다. 또한 특정 상태(예: 현재 메뉴 모드, 최근 사용한 파일 경로 등)를 필요로 하면 전역 변수 또는 `main` 함수 스코프 내 변수를 활용할 수 있습니다.

4. 파일 입출력 형식 설명

이 섹션에서는 프로그램에서 다루는 **입출력 파일 형식**과 예시를 설명합니다:

- **엑셀 파일 (입력):** 엑셀 파일은 치환 작업의 원본 정보입니다. **컬럼은 총 4개**이며 각각 아래와 같습니다:
- `source_file` : 원본 파일 경로 (치환을 적용할 대상 XML 또는 텍스트 파일 경로)
- `destination_file` : 대상 파일 경로 (원본 파일을 복사하여 치환 결과를 저장할 경로)
- `from` : 치환할 문자열 (변경 전 문자열 패턴)
- `to` : 변경할 문자열 (변경 후 문자열 패턴)

엑셀 파일은 **한 행(row)**당 하나의 치환 규칙을 나타냅니다. **동일한 원본 파일**에 대해 여러 치환 규칙이 존재할 수 있으며, 이 경우 각 규칙이 별도의 행으로 나열됩니다. 예를 들어 엑셀에 다음과 같은 데이터가 있을 수 있습니다:

source_file	destination_file	from	to
`data/input1.xml`	`data/output1.xml`	`apple`	`banana`

source_file	destination_file	from	to
`data/input1.xml`	`data/output1.xml`	`cat`	`dog`
`data/input2.xml`	`data/output2.xml`	`hello`	`world`

이 예에서 `data/input1.xml`에 대해서 두 개의 치환 (`apple->banana` , `cat->dog`)이 정의되어 있으며, 결과는 `data/output1.xml`에 저장됩니다. `input2.xml`에 대해서는 한 개의 치환(`hello->world`)이 정의되어 있고 `output2.xml`에 저장됩니다.

- **YAML 파일 (입출력):** YAML 파일은 위 엑셀에서 정의한 작업들을 구조화하여 표현합니다. YAML의 최상위 계층은 **작업 목록 (jobs)**이며, 각 작업은 원본 파일 하나에 대한 일련의 치환 리스트로 구성됩니다. YAML 파일의 구조 예시는 다음과 같습니다:

```yaml
jobs:
  - source: "data/input1.xml"
    destination: "data/output1.xml"
    replacements:
      - from: "apple"
        to: "banana"
      - from: "cat"
        to: "dog"
  - source: "data/input2.xml"
    destination: "data/output2.xml"
    replacements:
      - from: "hello"
        to: "world"
```

위 YAML에서 `jobs` 아래에 두 개의 작업이 있으며, 첫 번째 작업은 `input1.xml`을 읽어 `output1.xml`에 쓰면서 두 종류의 문자열 치환을 적용하는 내용입니다. 두 번째 작업은 `input2.xml`에 한 가지 치환을 적용합니다. YAML은 사람이 읽기 쉽고 편집하기 쉬운 포맷으로, 설정 파일로 적합합니다. 프로그램은 이 YAML을 읽어 각 작업을 순차적으로 처리합니다.

- **로그 파일 (출력): 실행(3번 메뉴)** 시 생성되는 로그 파일은 치환 작업의 상세 내역을 기록합니다. 로그 파일은 일반적인 텍스트 (.txt) 형식이며, 예를 들어 다음과 같은 내용이 포함됩니다 (포맷 예시):

```
[2025-05-19 00:00:00] Started replacement jobs (total 2 jobs)
[2025-05-19 00:00:00] Job1: data/input1.xml -> data/output1.xml
[2025-05-19 00:00:01]    "apple" -> "banana": 3 occurrences replaced
[2025-05-19 00:00:01]    "cat" -> "dog": 1 occurrence replaced
[2025-05-19 00:00:01] Job1 completed: 4 replacements in total.
[2025-05-19 00:00:01] Job2: data/input2.xml -> data/output2.xml
[2025-05-19 00:00:02]    "hello" -> "world": 2 occurrences replaced
[2025-05-19 00:00:02] Job2 completed: 2 replacements in total.
[2025-05-19 00:00:02] All jobs completed. Total files: 2, Total replacements:
6.
```

위 예시는 로그 파일에 타임스탬프와 함께 각 작업의 진행 상황과 세부 치환 결과(치환 문자열, 치환 횟수)를 기록한 형태입니다. 정확한 로그 포맷은 구현에 따라 달라질 수 있으나, **어떤 파일에서 어떤 치환이 몇 건 일어났는지** 추적할 수 있어야 합니다. 만약 치환 과정에서 오류가 발생하면 해당 오류 내용도 로그에 기록됩니다.

- **요약 파일 (출력):** 요약 파일은 모든 작업 결과의 요약 정보를 담은 텍스트 파일입니다. 로그 파일이 상세 내역이라면, 요약 파일은 전체를 조망할 수 있는 요약본입니다. 예를 들어 다음과 같습니다:

```
치환 작업 요약:
- data/input1.xml -> data/output1.xml: 4건 치환 (2종류 문자열)
- data/input2.xml -> data/output2.xml: 2건 치환 (1종류 문자열)
------------------------------
총 파일 수: 2, 총 치환 횟수: 6
```

요약에는 각 파일별 치환된 총 횟수와 고유 치환 종류 개수를 기록하고, 마지막에 전체 통계를 제공합니다. 이를 통해 사용자는 작업 결과를 한눈에 파악할 수 있습니다. (로그와 요약을 별도 파일로 두는 대신 하나의 파일로 합칠 수도 있지만, 요구사항에 따라 분리하여 제공하는 것으로 합니다.)

5. 프로그램 실행 흐름도

아래는 본 프로그램의 전체 실행 흐름을 나타낸 **흐름도(flowchart)**입니다. 사용자의 메뉴 선택에 따라 프로그램이 어떤 동작을 수행하고 다시 메뉴로 돌아오는지 보여줍니다:

- **프로그램 시작** → 콘솔에 **메뉴 표시** (1.YAML 생성, 2.미리보기, 3.실행, 0.종료).
- **[1 입력] YAML 생성**: 엑셀 파일 경로 입력 → 엑셀에서 데이터 읽어 YAML 작성 → YAML 파일 출력 → 완료 메시지 → **메뉴로 복귀**.
- **[2 입력] 미리보기**: YAML 파일 경로 입력 → YAML 파싱 → 각 작업별 원본 내용에 치환 적용 → **diff 결과 출력 (콘솔)** → **메뉴로 복귀**.
- **[3 입력] 실행**: YAML 파일 경로 입력 → YAML 파싱 → 각 작업별 치환 실행 (원본→대상 복사 및 치환) → **로그 파일 및 요약 파일 저장** → 완료 메시지 출력 → **메뉴로 복귀**.
- **[0 입력] 종료**: 프로그램 종료 메시지 출력 후 **프로그램 종료**.

위 흐름에서 메뉴로 복귀한 이후 사용자는 원하면 계속해서 다른 기능을 수행할 수 있습니다 (예: YAML 생성 후 미리보기 연속 실행). 모든 기능 동작 후에는 결국 다시 초기 메뉴로 돌아오거나 사용자가 종료를 선택할 때까지 반복합니다.

6. 확장 가능성 및 향후 개선

이 프로그램은 기본적인 문자열 치환 기능에 초점을 맞추어 설계되었지만, 구조를 확장하여 다양한 추가 기능을 도입할 수 있습니다:

- **정규식 치환:** 현재는 단순한 문자열 리터럴 치환만 지원하지만, PyYAML 설정에 **정규식 사용 플래그**를 추가하고 Python의 `re` 모듈을 사용하여 `re.sub()`를 적용하면 정규 표현식 패턴에 대한 치환도 가능할 것입니다. 예를 들어 YAML의 각 replacement 항목에 `regex: true` 필드를 추가하여 정규식 여부를 표시하고, 코드에서 이를 감지하여 `str.replace` 대신 `re.sub`를 수행하도록 확장할 수 있습니다.

- **미리보기 개선:** 현재 unified diff 또는 ndiff 형식으로 콘솔에 출력하는 diff를 HTML 보고서나 GUI 창으로 보여줄 수 있습니다. 예를 들어 **컬러 하이라이트**된 diff 결과를 생성하거나, 웹 브라우저로 볼 수 있는 HTML 파일로 저장하는 기능을 추가할 수 있습니다. Python의 `difflib` 모듈은 HTML diff를 생성하는 예시도 제공하고 있어 이를 활용할 수 있습니다.

- **GUI 연동:** 콘솔 환경에 익숙하지 않은 사용자를 위해 **그래픽 사용자 인터페이스(GUI)**를 제공할 수 있습니다. Python의 Tkinter나 PyQt 등을 사용하여 파일 선택 대화상자, 진행 상태 표시, 결과 뷰어 등을 갖춘 GUI 어플리케이션으로 확장하면 편의성이 증가합니다. 현재 구조에서 함수 분리가 잘 되어 있으므로, GUI 이벤트에서 `generate_yaml_from_excel`, `preview_diff`, `execute_replacements` 등을 호출하는 형태로 충분히 연동 가능합니다.

- **대용량 파일 및 성능:** 매우 큰 파일의 경우 현재 구현은 파일 전체를 메모리에 읽어 처리하는데, 메모리 사용량 문제가 발생할 수 있습니다. 이를 개선하기 위해 **스트리밍 방식 치환**(예: 한 줄씩 읽어 쓰면서 치환)이나 메모리 맵 등을 활용할 수 있습니다. 또한 치환 작업이 매우 많은 경우 진행률을 표시하거나 멀티스레딩/멀티프로세싱으로 작업 병렬화를 고려할 수 있습니다.

- **구조 및 배포:** 향후 해당 코드를 모듈화하여 다른 Python 코드에서 import하여 사용할 수 있는 라이브러리 형태로 만들 수 있습니다. 또는 CLI 툴로 패키징하여 `pip`으로 설치 가능하게 하거나, 실행 파일로 만들어 배포하는 것도 가능합니다.

이러한 개선 사항들은 현재 프로그램의 핵심 동작을 변경하지 않으면서도 사용자 편의성과 적용 범위를 넓힐 수 있는 방향입니다. 초기 버전에서는 요구된 기능(엑셀->YAML, diff 미리보기, 실행 및 로그)이 충실히 구현되는 것을 목표로 하고, 이후 단계적으로 확장을 고려합니다.

7. 전체 프로그램 Python 코드 (단일 파일)

아래는 위에서 설명한 기능을 모두 포함하는 **Python 코드 전체**입니다. 이 코드는 한 파일에 구성되어 있으며, 모듈 임포트, 함수 정의, 그리고 `if __name__ == "__main__":` 블록에서의 메뉴 구성이 포함됩니다. 실제 사용 시 필요한 라이브러리(openpyxl, PyYAML 등)를 사전에 설치해야 합니다.

```
import openpyxl
import yaml
import difflib
import os
import datetime

def generate_yaml_from_excel(excel_path, yaml_path):
  """엑셀 파일을 읽어 YAML 파일을 생성한다."""
    wb = openpyxl.load_workbook(excel_path)
    sheet = wb.active  # 첫 번째 시트를 사용
    jobs_dict = {}  # (source, dest)를 키로 치환 리스트를 모음
    for row in sheet.iter_rows(min_row=2, values_only=True):  # 2행부터 데이터 읽기 (1행은 헤더)
        source_file, destination_file, from_str, to_str = row
        if source_file is None or destination_file is None or from_str is None or to_str is
None:
            continue  # 빈 행은 건너뜀
        key = (str(source_file).strip(), str(destination_file).strip())
        if key not in jobs_dict:
            jobs_dict[key] = []
```

```
            jobs_dict[key].append({"from": str(from_str), "to": str(to_str)})
    # jobs_dict를 YAML용 구조로 변환
    jobs = []
    for (source, dest), replacements in jobs_dict.items():
        jobs.append({
            "source": source,
            "destination": dest,
            "replacements": replacements
        })
    # YAML 파일로 저장
    with open(yaml_path, 'w', encoding='utf-8') as yf:
        yaml.safe_dump({"jobs": jobs}, yf, allow_unicode=True)
    return len(jobs)

def apply_replacements(text, replacements):
  """여러 치환 규칙을 순차적으로 적용하여 새로운 텍스트 반환."""
    new_text = text
    for repl in replacements:
        # 단순 문자열 치환 (정규식의 경우 필요시 re.sub로 변경 가능)
        from_str = repl.get("from", "")
        to_str = repl.get("to", "")
        if from_str:
            new_text = new_text.replace(from_str, to_str)
    return new_text

def compute_diff(original_text, modified_text, fromfile="[Before]", tofile="[After]"):
  """두 텍스트 버전에 대한 unified diff를 생성하여 리스트로 반환."""
    original_lines = original_text.splitlines(keepends=True)
    modified_lines = modified_text.splitlines(keepends=True)
    diff_lines = difflib.unified_diff(original_lines, modified_lines,
                                      fromfile=fromfile, tofile=tofile, lineterm='')
    return list(diff_lines)

def preview_diff(yaml_path):
  """YAML 파일에 정의된 각 작업에 대해 diff 미리보기를 콘솔에 출력."""
    try:
        with open(yaml_path, 'r', encoding='utf-8') as yf:
            data = yaml.safe_load(yf)
    except FileNotFoundError:
        print(f"YAML 파일을 찾을 수 없습니다: {yaml_path}")
        return
    jobs = data.get("jobs", []) if data else []
    for job in jobs:
        source = job.get("source")
        dest = job.get("destination", "(preview)")
        replacements = job.get("replacements", [])
        if not source or not replacements:
            continue
        try:
            with open(source, 'r', encoding='utf-8') as sf:
                original_text = sf.read()
```

```
            except FileNotFoundError:
                print(f"\n[오류] 원본 파일을 찾을 수 없습니다: {source}")
                continue
            # 치환 적용 (미리보기이므로 파일 저장 안 함)
            modified_text = apply_replacements(original_text, replacements)
            # diff 계산
            diff_lines = compute_diff(original_text, modified_text, fromfile=source, tofile=f"{dest}
(preview)")
            # diff 출력
            print(f"\n*** {source} vs {dest} 미리보기 diff ***")
            for line in diff_lines:
                print(line)
            print("*** 끝 ***")
        print("\n미리보기가 완료되었습니다.\n")

def execute_replacements(yaml_path, log_path, summary_path):
    """YAML 정의에 따라 실제 파일 치환을 수행하고 로그/요약을 작성."""
    try:
        with open(yaml_path, 'r', encoding='utf-8') as yf:
            data = yaml.safe_load(yf)
    except FileNotFoundError:
        print(f"YAML 파일을 찾을 수 없습니다: {yaml_path}")
        return
    jobs = data.get("jobs", []) if data else []
    if not jobs:
        print("YAML에 실행할 작업이 없습니다.")
        return
    # 로그와 요약 파일 열기
    log_f = open(log_path, 'w', encoding='utf-8')
    summary_f = open(summary_path, 'w', encoding='utf-8')
    start_time = datetime.datetime.now()
    log_f.write(f"[{start_time}] 치환 작업 시작 (총 {len(jobs)}개 작업)\n")
    summary_lines = []
    total_replacements = 0
    job_index = 0
    for job in jobs:
        job_index += 1
        source = job.get("source")
        dest = job.get("destination")
        replacements = job.get("replacements", [])
        if not source or not dest or not replacements:
            continue
        log_f.write(f"[{datetime.datetime.now()}] 작업{job_index}: {source} -> {dest}\n")
        # 원본 파일 읽기
        try:
            with open(source, 'r', encoding='utf-8') as sf:
                original_text = sf.read()
        except FileNotFoundError:
            error_msg = f"[{datetime.datetime.now()}]  [오류] 원본 파일을 찾을 수 없습니다: {source}
\n"
            log_f.write(error_msg)
```

```
                print(error_msg, end="")
                continue
            modified_text = original_text
            job_replacement_count = 0
            # 필요 시 대상 디렉토리 생성
            os.makedirs(os.path.dirname(dest), exist_ok=True)
            # 각 치환 적용 및 로그 기록
            for repl in replacements:
                from_str = repl.get("from", "")
                to_str = repl.get("to", "")
                if from_str == "" or to_str is None:
                    continue
                # 치환 적용
                new_text = modified_text.replace(from_str, str(to_str))
                # 치환 횟수 계산 (단순히 발생 횟수로 추정)
                count = modified_text.count(from_str) if from_str else 0
                modified_text = new_text
                job_replacement_count += count
                total_replacements += count
                log_f.write(f"[{datetime.datetime.now()}]   \"{from_str}\" -> \"{to_str}\": {count}건 치환
\n")
            # 대상 파일 쓰기
            with open(dest, 'w', encoding='utf-8') as df:
                df.write(modified_text)
            log_f.write(f"[{datetime.datetime.now()}] 작업{job_index} 완료: 총 {job_replacement_count}
건 치환\n")
            summary_lines.append(f"- {source} -> {dest}: {job_replacement_count}건 치환
({len(replacements)}종류 문자열)")
        # 전체 요약 정보 작성
        end_time = datetime.datetime.now()
        log_f.write(f"[{end_time}] 모든 작업 완료. 총 파일: {len(jobs)}, 총 치환 횟수: {total_replacements}
\n")
        summary_f.write("치환 작업 요약:\n")
        for line in summary_lines:
            summary_f.write(line + "\n")
        summary_f.write("-" * 30 + "\n")
        summary_f.write(f"총 파일 수: {len(jobs)}, 총 치환 횟수: {total_replacements}\n")
        # 파일 닫기
        log_f.close()
        summary_f.close()
        print(f"치환 작업을 완료했습니다. 로그: {log_path}, 요약: {summary_path}\n")

def main():
    """메인 함수 - 메뉴 구성 및 사용자 입력 처리."""
    while True:
        print("======= 문자열 치환 도구 =======")
        print("1. YAML 생성 (엑셀 -> YAML)")
        print("2. 미리보기 (YAML 기반 diff 출력)")
        print("3. 실행 (치환 적용 및 로그 저장)")
        print("0. 종료")
        choice = input("원하는 기능의 번호를 입력하세요: ").strip()
```

```
        if choice == '1':
            excel_path = input("엑셀 파일 경로를 입력하세요: ").strip()
            yaml_path = input("생성할 YAML 파일 경로를 입력하세요: ").strip()
            try:
                job_count = generate_yaml_from_excel(excel_path, yaml_path)
                print(f"YAML 파일이 생성되었습니다. (총 {job_count}개 작업) -> {yaml_path}\n")
            except Exception as e:
                print(f"[오류] YAML 생성 실패: {e}\n")
        elif choice == '2':
            yaml_path = input("YAML 파일 경로를 입력하세요: ").strip()
            preview_diff(yaml_path)
        elif choice == '3':
            yaml_path = input("YAML 파일 경로를 입력하세요: ").strip()
            log_path = input("로그 파일 경로를 입력하세요 (예: result.log): ").strip()
            summary_path = input("요약 파일 경로를 입력하세요 (예: summary.txt): ").strip()
            execute_replacements(yaml_path, log_path, summary_path)
        elif choice == '0' or choice.lower() == 'q' or choice.lower() == 'quit':
            print("프로그램을 종료합니다. 안녕히 가세요!")
            break
        else:
            print("올바른 번호를 입력하세요.\n")

# 프로그램 진입점
if __name__ == "__main__":
    main()
```

위 코드에서는 메뉴 입력에 따라 각각의 기능이 동작하며, 예외 상황(파일 미존재 등)을 출력하도록 최소한의 처리를 포함하고 있습니다. 실제 운영 환경에 맞게 경로 처리나 인코딩 에러 처리, 예외 handling 등을 보완할 수 있습니다.

주요 구현 포인트:

- 엑셀 처리: `openpyxl.load_workbook`과 `sheet.iter_rows`를 사용하여 데이터를 읽고, **job 단위로 그룹화**하여 YAML 생성.
- YAML 처리: `yaml.safe_dump`과 `yaml.safe_load`를 사용하여 YAML 파일을 쓰고 읽습니다.
- Diff 미리보기: Python 표준 `difflib.unified_diff`를 통해 **Unified Diff** 형식의 차이를 계산하여 콘솔에 출력합니다.
- 파일 치환: 문자열의 `replace` 메서드를 사용하여 모든 치환을 적용하며, 치환 횟수 집계를 위해 `count`를 활용합니다. (`re.subn` 등을 사용하면 정규식 치환과 치환 횟수 추출을 동시에 할 수도 있습니다.)
- 로그/요약: `datetime` 모듈로 타임스탬프를 기록하고, 최종 통계를 요약 파일에 작성합니다.

이상으로 프로그램의 설계 요구사항에 따른 상세 내용과 코드 구현을 모두 기술하였습니다. 프로그램을 실행하기 전에 필요한 라이브러리를 설치하고 (예: `pip install openpyxl pyyaml`), 엑셀 파일과 YAML 파일 경로 등을 준비하면, 콘솔을 통해 메뉴 기반으로 손쉽게 치환 작업을 수행할 수 있습니다.
