Day 1: RAG 파이프라인 최소 구조 - 완전 가이드

> **GitHub Repository**: RAG-on-Local-CPU-minimal

> 3파일로 이해하는 RAG의 뼈대

🎯 학습 목표

Day 1의 핵심은 다음 한 문장입니다:

> **RAG는 복잡한 기술이 아니라 단계적으로 연결된 파이프라인이다.**

이 Day를 마치면:

- ☑ `main → pipeline → loader` 흐름 이해
- ☑ RAG 파이프라인의 기본 구조 파악
- ☑ Python 모듈 분리의 기본 이해
- ☑ 왜 이런 구조로 설계했는지 이해

📂 프로젝트 구조

```
Rag_minimal_day1/
├── main.py          # 진입점 (시작)
├── pipeline.py      # RAG 파이프라인 (흐름)
└── loader.py        # 문서 로더 (데이터)
```

🤔 왜 이런 구조인가?

3파일 구조의 의도

1. 관심사 분리 (Separation of Concerns)

```
main.py      → "무엇을 실행할까?"
pipeline.py  → "어떤 순서로 처리할까?"
loader.py    → "데이터를 어떻게 가져올까?"
```

각 파일이 **하나의 책임**만 가지면:

- ☑ 코드 읽기 쉬움
- ☑ 수정할 때 영향 범위 명확
- ☑ 재사용 가능

2. 확장성

```
# loader.py만 교체하면 다른 데이터 소스 사용 가능
loader_file.py      # 파일에서 읽기
loader_db.py        # DB에서 읽기
loader_api.py       # API에서 읽기
```

3. 테스트 용이성

```
# 각 파일을 독립적으로 테스트 가능
test_loader()    # loader만 테스트
test_pipeline()  # pipeline만 테스트
```

왜 파이프라인 패턴인가?

RAG는 **여러 단계의 연결**입니다:

```
입력 → 문서검색 → 재정렬 → LLM생성 → 출력
```

Day 1에서는 단순하지만:

```
입력 → 문서로딩 → 출력
```

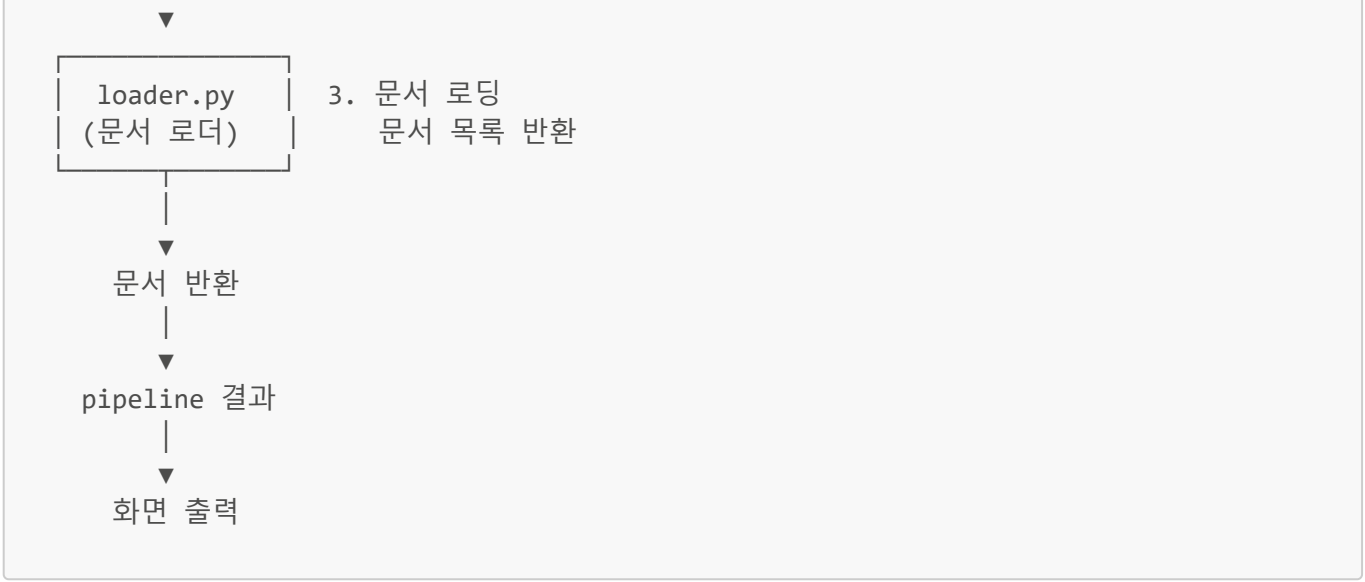
앞으로 단계가 추가됩니다:

- Day 5: Vector DB 검색 추가
- Day 6: Reranker 추가
- Day 7: LLM 생성 추가

pipeline.py는 이 모든 단계를 연결하는 허브 역할입니다.

🔄 실행 흐름도

```
┌─────────────┐
│   main.py   │  1. 프로그램 시작
│  (시작점)   │     query 정의
└─────────────┘
       │
       ▼
┌─────────────┐
│ pipeline.py │  2. RAG 파이프라인 실행
│ (파이프라인) │     각 단계 연결
└─────────────┘
       │
```



⚙ VS Code에서 실행하기

1. VS Code 열기

Windows 사용자 (PowerShell):

```
cd Rag_minimal_day1
code .
```

macOS/Linux 사용자 (Bash):

```
cd Rag_minimal_day1
code .
```

2. Python 파일 실행

방법 1: PowerShell/Bash 사용

Windows (PowerShell):

```
python main.py
```

macOS/Linux (Bash):

```
python main.py
# 또는
python3 main.py
```

방법 2: VS Code 실행 버튼

- `main.py` 파일 열기
- 우측 상단 ▶ 버튼 클릭 (OS 무관)

예상 출력:

```
=== QUERY ===
RAG가 뭐야?

=== DOCUMENTS ===
- (1) RAG 개념: RAG는 Retrieval Augmented Generation의 약자이다.
- (2) LLM 개념: LLM은 대규모 언어 모델을 문맥을 기반으로 답변한다.
- (3) Pipeline: AI 시스템은 단계별 파이프라인으로 구성된다.
```

🐛 디버깅 방법

사전 준비

VS Code Python 확장 설치 (필수)

1. VS Code 열기
2. 왼쪽 확장 아이콘 클릭 (또는 `Ctrl + Shift + X`)
3. "Python" 검색
4. Microsoft의 Python 확장 설치

방법 1: print() 사용

loader.py에 디버깅 추가:

```python
def load_documents():
    print("📂 문서 로딩 시작")  # 디버깅용
    docs = ["doc1.txt", "doc2.txt", "doc3.txt"]
    print(f"☑ {len(docs)}개 로딩 완료")  # 결과 확인
    return docs
```

실행 결과:

```
📂 문서 로딩 시작
☑ 3개 로딩 완료
=== QUERY ===
RAG가 뭐야?
...
```

방법 2: VS Code 디버거 사용

1. 브레이크포인트 설정

- `main.py` 파일 열기
- 7번 줄 (print 문) 왼쪽 클릭 → 빨간 점 생성
- 원하는 만큼 여러 줄에 설정 가능

2. 디버그 실행

- **F5** 누르기
- "Python Debugger" 선택 (처음 한 번만)
- 브레이크포인트에서 코드가 멈춤 (노란색 배경)

3. 디버그 컨트롤

- **F5**: 다음 브레이크포인트까지 실행 (Continue)
- **F10**: 다음 줄로 이동 (Step Over)
- **F11**: 함수 안으로 들어가기 (Step Into)
- **Shift + F11**: 함수에서 나오기 (Step Out)

4. 변수 확인

- **왼쪽 패널 "VARIABLES"**: 모든 변수 값 확인
  - Locals: query, result 등
- **변수 위에 마우스 오버**: 값 미리보기
- **WATCH 패널**: 특정 변수 추적 가능

5. 디버그 화면 구성

```
왼쪽 패널:
├─ VARIABLES    # 현재 변수들
├─ WATCH        # 감시할 변수 추가
├─ CALL STACK   # 함수 호출 순서
└─ BREAKPOINTS  # 브레이크포인트 목록

하단:
└─ Python Debug Console  # 디버그 중 명령 실행
```

디버깅 팁:

```
main.py:7 에서 시작 (print 문)
  ↓ F11 (Step Into)
pipeline.py 진입
  ↓ F11 (Step Into)
loader.py 진입
  ↓ 왼쪽 VARIABLES에서 docs 확인
  ↓ F10 (Step Over)
반환값 확인
```

📖 코드 Deep Dive

main.py 한 줄씩 이해하기

```python
# Line 1: pipeline 모듈에서 pipeline 함수 가져오기
from pipeline import pipeline

# Line 3-4: 메인 실행 블록
# 다른 파일에서 import해도 실행 안 되게
def main():
    # Line 5: 사용자 질문 정의
    query = "RAG가 뭐야?"

    # Line 6: 파이프라인 실행
    # query를 전달하고 result 받기
    result = pipeline(query)

    # Line 8-9: 결과 출력
    print(f"=== QUERY ===")
    print(result["query"])

    # Line 11-13: 문서 목록 출력
    print(f"\n=== DOCUMENTS ===")
    for doc in result["documents"]:
        print(f"- {doc}")

# Line 15-16: 스크립트 직접 실행 시만 main() 호출
if __name__ == "__main__":
    main()
```

핵심 포인트:

- `from pipeline import pipeline`: 다른 파일의 함수 사용
- `if __name__ == "__main__"`: 모듈화의 핵심 패턴
- `result = pipeline(query)`: 파이프라인에 모든 것 위임

pipeline.py 한 줄씩 이해하기

```python
# Line 1: loader 모듈에서 load_documents 함수 가져오기
from loader import load_documents

# Line 3: pipeline 함수 정의
def pipeline(query):
    # Line 4: 문서 로딩 단계
    # 이 한 줄이 "Retrieve" 단계
    documents = load_documents()

    # Line 6-9: 결과 딕셔너리 구성
    # 나중에 더 많은 정보 추가 예정
    return {
```

```
        "query": query,
        "documents": documents
    }
```

핵심 포인트:

- `documents = load_documents()`: 데이터 가져오기 위임
- `return {...}`: 딕셔너리로 여러 정보 반환
- 현재는 단순하지만, 앞으로 단계 추가 예정

loader.py 한 줄씩 이해하기

```
# Line 1: 함수 정의
def load_documents():
    # Line 2-6: 하드코딩된 문서 목록
    # Day 2부터 파일에서 읽어옴
    return [
        "(1) RAG 개념: RAG는 Retrieval Augmented Generation의 약자이다.",
        "(2) LLM 개념: LLM은 대규모 언어 모델을 문맥을 기반으로 답변한다.",
        "(3) Pipeline: AI 시스템은 단계별 파이프라인으로 구성된다."
    ]
```

핵심 포인트:

- 현재는 고정된 데이터
- 나중에 파일, DB, API로 교체 가능
- 리스트 형태로 반환

💡 학습 효율을 높이는 팁

1. 디버깅 순서 추천

처음 실행:

```
1. 그냥 실행 (python main.py)
   → 어떤 출력이 나오는지 확인

2. print() 추가
   → 각 단계마다 print 찍어보기

3. VS Code 디버거
   → 브레이크포인트로 한 줄씩 따라가기
```

효과적인 브레이크포인트 위치:

```
# main.py
query = "RAG가 뭐야?"      # ← 여기 (변수 초기화)
result = pipeline(query)  # ← 여기 (함수 호출 전)

# pipeline.py
documents = load_documents()  # ← 여기 (데이터 로딩)

# loader.py
return [...]  # ← 여기 (반환 직전)
```

2. 코드 읽는 방법

Top-Down 방식 (추천):

```
1. main.py부터 읽기
2. pipeline() 호출 발견
3. pipeline.py로 이동
4. load_documents() 호출 발견
5. loader.py로 이동
```

Bottom-Up 방식:

```
1. loader.py부터 읽기 (가장 단순)
2. pipeline.py 읽기
3. main.py 읽기 (전체 흐름)
```

3. 변수 이름 이해하기

```
query       # 사용자의 질문
documents   # 검색된 문서들
result      # 최종 결과 (딕셔너리)
```

좋은 변수 이름의 특징:

- 역할이 명확함
- 복수형(documents)과 단수형(document) 구분
- 약어 사용 최소화

🔧 직접 해보기: 간단한 확장

실습 1: 문서 추가하기

loader.py 수정:

```
def load_documents():
    return [
        "(1) RAG 개념: RAG는 Retrieval Augmented Generation의 약자이다.",
        "(2) LLM 개념: LLM은 대규모 언어 모델을 문맥을 기반으로 답변한다.",
        "(3) Pipeline: AI 시스템은 단계별 파이프라인으로 구성된다.",
        # 👇 여기에 추가
        "(4) Vector DB: 문서를 벡터로 저장하여 빠르게 검색한다.",
        "(5) Embedding: 텍스트를 숫자 벡터로 변환하는 과정이다."
    ]
```

실행 결과:

```
=== DOCUMENTS ===
- (1) RAG 개념: ...
- (2) LLM 개념: ...
- (3) Pipeline: ...
- (4) Vector DB: ...  ← 추가됨!
- (5) Embedding: ... ← 추가됨!
```

실습 2: 디버깅 메시지 추가하기

각 파일에 print 추가:

```
# main.py
def main():
    print("🚀 프로그램 시작")  # 추가
    query = "RAG가 뭐야?"
    result = pipeline(query)
    print("☑ 프로그램 완료")  # 추가

# pipeline.py
def pipeline(query):
    print(f"📋 파이프라인 시작: {query}")  # 추가
    documents = load_documents()
    print(f"📋 파이프라인 완료")  # 추가
    return {...}

# loader.py
def load_documents():
    print("🗁 문서 로딩 중...")  # 추가
    docs = [...]
    print(f"🗁 {len(docs)}개 문서 로딩 완료")  # 추가
    return docs
```

실행 결과:

```
🚀 프로그램 시작
📋 파이프라인 시작: RAG가 뭐야?
📂 문서 로딩 중...
📂 3개 문서 로딩 완료
📋 파이프라인 완료
=== QUERY ===
RAG가 뭐야?
...
☑ 프로그램 완료
```

실습 3: 쿼리 변경하기

main.py 수정:

```python
def main():
    # 다른 질문으로 바꿔보기
    query = "LLM이 뭐야?"        # 원래: "RAG가 뭐야?"
    # query = "Pipeline이 뭐야?"  # 또는 이것으로
    result = pipeline(query)
```

출력 변화 확인:

```
=== QUERY ===
LLM이 뭐야?  ← 변경됨!

=== DOCUMENTS ===
(문서는 동일)
```

🧠 파일별 역할 요약

main.py

```python
# 역할: 프로그램 시작점
# - 사용자 쿼리 정의
# - pipeline 호출
# - 결과 출력

# 핵심 개념:
# - if __name__ == "__main__"
# - from ... import ...
```

pipeline.py

```
# 역할: RAG 파이프라인 중심
# - 여러 단계를 연결하는 허브
# - Day 2부터 단계가 계속 추가됨
# - 현재: loader만 호출

# 핵심 개념:
# - 파이프라인 패턴
# - 딕셔너리 반환
# - 단계적 확장
```

loader.py

```
# 역할: 문서 가져오기
# - Day 1: 하드코딩된 문서 목록
# - 향후: 파일 / DB / API로 교체 가능

# 핵심 개념:
# - 데이터 소스 추상화
# - 리스트 반환
# - 교체 가능성
```

☑ Day 1 완료 기준

다음을 이해하고 실습했다면 완료입니다:

이해

- ✔ `main → pipeline → loader` 흐름을 설명할 수 있다
- ✔ RAG가 "단계적 파이프라인"임을 이해했다
- ✔ 왜 3파일 구조인지 이해했다
- ✔ 각 파일의 역할과 책임을 설명할 수 있다

실습

- ✔ VS Code에서 코드를 실행할 수 있다
- ✔ 브레이크포인트로 디버깅할 수 있다
- ✔ print()로 흐름을 추적할 수 있다
- ✔ 문서를 추가하거나 쿼리를 변경할 수 있다

체득

- ✔ 코드를 한 줄씩 따라갈 수 있다
- ✔ 변수 값의 변화를 추적할 수 있다
- ✔ pipeline에 단계가 추가될 수 있음을 인지했다

➡ SOON 다음 단계 (Day 2)

Day 2에서는 다음을 추가합니다:

1. 실제 파일 읽기

```
# loader.py
def load_documents():
    with open("data/docs.txt") as f:
        return f.read().split("\n")
```

2. 설정 분리

```
# config.yaml
data_path: "data/docs.txt"
```

3. Docker 실행

```
docker build -t rag-day2 .
docker run rag-day2
```

4. 경로 처리

```
BASE_DIR = Path(__file__).parent
data_path = BASE_DIR / "data" / "docs.txt"
```

Day 1의 3파일 구조에 **실무 패턴**이 추가됩니다!

💬 학습 후 자가 점검

다음 질문에 답할 수 있나요?

1. **main.py의 역할은?**
    - 프로그램 시작점, 파이프라인 호출, 결과 출력
2. **pipeline.py에서 앞으로 추가될 단계는?**
    - Vector 검색, Reranking, LLM 생성 등
3. **loader.py를 교체하면 어떻게 되나?**
    - 다른 데이터 소스 사용 가능 (파일, DB, API)
4. **왜 3개 파일로 나눴나?**
    - 관심사 분리, 확장성, 테스트 용이성

5. **브레이크포인트는 어디에 찍어야 하나?**
    - 변수 초기화, 함수 호출 전, 반환 직전

모두 답할 수 있다면 Day 1 완료! 🎉

📜 라이선스

⭐ **이 프로젝트가 도움이 되었다면 Star를 눌러주세요!**