**Polars 활용 레시피 100제**

**Polars 데이터 분석 레시피 100제 (초판)**

**발 행**｜2025년 04월 01일

**저 자**｜Evan

**펴낸이**｜한건희

**펴낸곳**｜주식회사 부크크

**출판사등록**｜2014.07.15(제2014-16호)

**주 소**｜서울특별시 금천구 가산디지털1로 119 SK트윈타워 A동 305호

**전 화**｜1670-8316

**이메일**｜info@bookk.co.kr

**ISBN**｜979-11-410-0000-0

**www.bookk.co.kr**

**ⓒ Evan 2025**

본 책은 저작자의 지적 재산으로서 무단 전재와 복제를 금합니다.

Polars 데이터 분석 레시피 100제

(초판)

Evan 지음

Table of Contents

[Chapter 1. Polars 시작하기 8](#_Toc191118814)

[레시피 1 – 데이터프레임 생성 8](#_Toc191118815)

[레시피 2 – 데이터 불러오기 11](#_Toc191118816)

[레시피 3 – Polars 데이터프레임 주요 속성 12](#_Toc191118817)

[데이터프레임 크기 확인 12](#_Toc191118818)

[컬럼명 확인 12](#_Toc191118819)

[데이터 타입 확인 13](#_Toc191118820)

[데이터프레임의 기본 정보 13](#_Toc191118821)

[레시피 4 – 데이터 호환성 14](#_Toc191118822)

[NumPy 배열과의 호환 14](#_Toc191118823)

[Pandas DataFrame과의 호환 15](#_Toc191118824)

[레시피 5 – Polars Series 생성 16](#_Toc191118825)

[레시피 6 – Polars Series 생성 18](#_Toc191118826)

[레시피 7 – Polars DataFrame에서 Series로 변형 19](#_Toc191118827)

[레시피 8 – Polars LazyFrame 생성 22](#_Toc191118828)

[LazyFrame 22](#_Toc191118829)

[LazyFrame vs. DataFrame 22](#_Toc191118830)

[레시피 9 – Polars LazyFrame로 파일 읽어오기 25](#_Toc191118831)

[scan\_csv() 25](#_Toc191118832)

[read\_csv() 27](#_Toc191118833)

[Chapter 4. Polars를 활용한 데이터 선택 28](#_Toc191118834)

[레시피 10 – 컬럼 선택 select() 29](#_Toc191118835)

[레시피 11 – 컬럼 추가 및 변경 with\_columns() 32](#_Toc191118836)

[레시피 12 – 정규표현식을 활용한 컬럼 선택 33](#_Toc191118837)

[레시피 13 – Polars selectors 35](#_Toc191118838)

[레시피 14 – 조건에 맞는 행 추출 filter 함수 37](#_Toc191118839)

[특정 값보다 큰 경우 37](#_Toc191118840)

[특정 값보다 작은 경우 37](#_Toc191118841)

[특정 값보다 같은 경우 38](#_Toc191118842)

[특정 값과 다른 경우 38](#_Toc191118843)

[값이 Null인 경우 38](#_Toc191118844)

[값이 NULL이 아닌 경우 39](#_Toc191118845)

[특정 문자열 포함 39](#_Toc191118846)

[리스트 값 중 하나와 일치 39](#_Toc191118847)

[여러 조건 AND (&) 40](#_Toc191118848)

[특정 문자열 포함 40](#_Toc191118849)

[Chapter 5. Polars를 활용한 데이터 가공 41](#_Toc191118850)

[레시피 15 – 사칙연산을 활용한 데이터 생성 41](#_Toc191118851)

[레시피 16 – 집계 함수를 활용한 새로운 컬럼 생성 42](#_Toc191118852)

[레시피 17 – 행 인덱스 추가 with\_row\_index 43](#_Toc191118853)

[레시피 18 – 데이터 수정 주요 메서드 44](#_Toc191118854)

[rename() 44](#_Toc191118855)

[map\_elements() 45](#_Toc191118856)

[fill\_null() 결측값 채우기 48](#_Toc191118857)

[레시피 19 – 컬럼 삭제 49](#_Toc191118858)

[drop()을 활용한 컬럼 삭제 49](#_Toc191118859)

[cs.exclude()을 활용한 컬럼 제거 50](#_Toc191118860)

[Chapter 6. Method Chaining 51](#_Toc191118861)

[레시피 20 – Method Chaining 코드 소개 52](#_Toc191118862)

[레시피 21 – 데이터 전처리 파이프라인 54](#_Toc191118863)

[레시피 22 – 데이터 집계 함수 파이프라인 55](#_Toc191118864)

[레시피 23 – 조건부 파이프라인 58](#_Toc191118865)

[Chapter 7. 데이터 수집 속도 비교 61](#_Toc191118866)

[레시피 24 – 데이터 수집 속도 비교 64](#_Toc191118867)

[레시피 25 – 메모리 사용 측정 비교 65](#_Toc191118868)

[Chapter 8. 데이터 Input & Output 67](#_Toc191118869)

[레시피 26 – CSV 파일 입출력 68](#_Toc191118870)

[CSV 파일 읽기 69](#_Toc191118871)

[CSV 파일 내보내기 70](#_Toc191118872)

[내보낸 파일 확인 70](#_Toc191118873)

[레시피 27 – Parquet 파일 입출력 71](#_Toc191118874)

[CSV 파일에서 parquet으로 내보내기 72](#_Toc191118875)

[parquet으로 파일 불러오기 73](#_Toc191118876)

[Hive 형태의 파티션된 parquet으로 파일 불러오기 74](#_Toc191118877)

[Chapter 1. 책을 출간하면서 77](#_Toc191118878)

[책을 집필하면서 느끼는 소회 77](#_Toc191118879)

[무엇을 배울 수 있는가 77](#_Toc191118880)

[Chapter 2. 개발환경설정 79](#_Toc191118881)

[1. Google Colab 79](#_Toc191118882)

[Colab에서 패키지 설치 및 업그레이드 79](#_Toc191118883)

[Colab에서 패키지 버전 확인 80](#_Toc191118884)

[2. 주요 라이브러리 80](#_Toc191118885)

[Local에서의 패키지 버전 확인 82](#_Toc191118886)

취업 준비생 분들에게 이 책을 바칩니다.

IT/빅데이터로 전업하고자 하는 많은 사람이 있습니다. 이 글의 저자, Sara, Evan도 전공자는 아니었습니다. 열심히 독학을 하면서, 이 분야로 진입하기 위해 매우 열심히 공부를 하며, 취업 준비를 하고 있습니다.

Evan은 IT/빅데이터와 관련된 첫 직장을 구하기 위해 100여군데의 이력서를 썼고, 같은 회사에 3년 내내 지원하고 떨어지기기를 반복한적도 있습니다. 정규직도 아니고 계약직에. 그래서, Evan은 취업 준비생의 고뇌와 고충을 매우 잘 이해하고 있습니다. 2011년 필리핀에서 만났던 그들도 생존의 위협을 느꼈듯이, 현재 취업 준비생들도 생존의 위협을 느끼고 있음을. Evan 역시 생존의 위협속에서 매일 하루 생존하기 위해 열심히 살고 있습니다.

Sara의 블로그 타이틀은 Grit 입니다. Grit의 핵심은 열정과 끈기, 단 3-6개월이 아닌 3-6년간의 노력이라고 합니다. 이 책이 취업 준비생 분들의 직업의 성공, 인생의 성공에 하나의 디딤돌이 되는 좋은 입문서가 되기를 기원합니다.

# Chapter 1. Polars 시작하기

**실습파일명 : ch01\_get\_started.ipynb**

## 레시피 1 – 데이터프레임 생성

데이터 분석의 첫 걸음은 데이터를 효과적으로 다룰 수 있는 구조를 만드는 것이다. Polars는 강력하고 직관적인 데이터프레임 기능을 제공하여 데이터 처리를 쉽고 빠르게 할 수 있다. 이번 레시피에서는 Polars 데이터프레임을 생성하는 세 가지 기본적인 방법을 살펴보도록 한다.

첫 번째는 파이썬 딕셔너리를 사용하는 방법이다. 딕셔너리의 키는 컬럼명이 되고 값은 리스트 형태의 데이터가 된다. 이 방법은 데이터가 이미 컬럼 단위로 정리되어 있을 때 유용하다.

|  |
| --- |
| df1 = pl.DataFrame({  "A": [1, 2, 3, 4, 5],  "B": ["a", "b", "c", "d", "e"],  "C": [1.1, 2.2, 3.3, 4.4, 5.5]  })  print("\n딕셔너리로 생성한 데이터프레임:")  print(df1)  [결과] |

텍스트, 스크린샷, 도표, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

두 번째는 중첩된 리스트를 사용하는 방법이다. 각 내부 리스트는 하나의 행을 나타내며, schema 매개변수를 통해 컬럼명을 별도로 지정해준다. 행 단위로 데이터가 구성되어 있을 때 적합한 방법이다.

|  |
| --- |
| df2 = pl.DataFrame([  [1, "a", 1.1],  [2, "b", 2.2],  [3, "c", 3.3],  [4, "d", 4.4],  [5, "e", 5.5]  ], schema=["A", "B", "C"])  print("\n리스트로 생성한 데이터프레임:")  print(df2)  [결과] |

텍스트, 스크린샷, 번호, 도표이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

마지막으로 Polars의 Series 객체들로 데이터프레임을 구성할 수 있다. 각 Series는 이름과 데이터를 가지며, 이들을 리스트로 묶어 데이터프레임을 생성한다. 이 방법은 개별 컬럼에 대한 세밀한 제어가 필요할 때 유용하다.

|  |
| --- |
| df3 = pl.DataFrame([  pl.Series("A", [1, 2, 3, 4, 5]),  pl.Series("B", ["a", "b", "c", "d", "e"]),  pl.Series("C", [1.1, 2.2, 3.3, 4.4, 5.5])  ])  print("\nSeries로 생성한 데이터프레임:")  print(df3)  [결과] |

텍스트, 스크린샷, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

## 레시피 2 – 데이터 불러오기

Polars는 다양한 형식의 데이터를 불러올 수 있다. 문법은 기존의 pandas와 비슷하기 때문에 코드 작성하는 데 큰 어려움은 없다.

|  |  |
| --- | --- |
| 파일 형식 | 메서드 |
| CSV | pl.read\_csv() |
| Excel | pl.read\_excel() |
| Parquet | pl.read\_parquet() |
| JSON | pl.read\_json() |

아래 예제에서는 CSV 파일을 불러오는 방법을 살펴본다. 데이터셋은 소스코드에서 확인이 가능하다.

|  |
| --- |
| df = pl.read\_csv("data/titanic\_dataset.csv")  print("\ntitanic 데이터셋:")  df.head(1)  [결과] |

텍스트, 폰트, 라인, 대수학이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

## 레시피 3 – Polars 데이터프레임 주요 속성

pandas와 Polars의 주요 기본 문법은 유사한 것이 많다. 본 장에서는 데이터프레임의 크기, 컬럼명, 데이터 타입, 기본 통계량을 확인하는 메서드를 사용한다. 기존의 pandas를 활용한 독자들은 어렵지 않게 이해할 수 있다.

### 데이터프레임 크기 확인

pandas와 동일한 문법을 사용하며 (행 수, 열 수)를 튜플로 반환한다.

|  |
| --- |
| print("\n데이터프레임의 크기(shape):")  df.shape  [결과]  데이터프레임의 크기(shape):  (891, 12) |

### 컬럼명 확인

pandas와 동일한 문법을 사용하며 리스트로 반환한다.

|  |
| --- |
| print("\n데이터프레임의 컬럼명:")  print(df.columns)  [결과]  데이터프레임의 컬럼명:  ['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp', 'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked'] |

### 데이터 타입 확인

pandas와 동일한 문법을 사용하며, 리스트로 반환한다.

|  |
| --- |
| print("\n데이터프레임의 데이터 타입:")  print(df.dtypes)  [결과]  데이터프레임의 데이터 타입:  [Int64, Int64, Int64, String, String, Float64, Int64, Int64, String, Float64, String, String] |

### 데이터프레임의 기본 정보

pandas와 동일한 문법을 사용하며, 리스트로 반환한다.

|  |
| --- |
| print("\n데이터프레임의 기본 정보:")  print(df.describe())  [결과] |

텍스트, 번호, 평행이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

## 레시피 4 – 데이터 호환성

다른 데이터 형식과의 호환성을 알아보도록 한다. Polars는 다른 데이터 관련 라이브러리인 NumPy 배열과의 호환, pandas 데이터프레임과의 호환성이 뛰어나다.

### NumPy 배열과의 호환

NumPy 배열을 Polars의 DataFrame으로 변환하는 방법은 **from\_numpy()** 메서드를 활용하는 것이다. **Polars.from\_numpy()** 메서드는 NumPy 배열을 **Polars.DataFrame**으로 변환하는 기능을 제공한다. 이때, **schema** 매개변수를 사용하면 데이터프레임의 컬럼명을 지정할 수 있다.

|  |
| --- |
| # numpy 배열 생성  import numpy as np  arr = np.array([[1, 2, 3], [4, 5, 6]])  print("\nNumPy 배열:")  print(arr)  # numpy 배열을 Polars 데이터프레임으로 변환  df\_from\_np = pl.from\_numpy(arr, schema=['col1', 'col2', 'col3'])  print("\nNumPy 배열로부터 생성된 Polars 데이터프레임:")  print(df\_from\_np)  [결과] |

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

### Pandas DataFrame과의 호환

Polars는 pandas와의 호환성을 제공하여 두 라이브러리 간 데이터 변환을 쉽게 할 수 있다. pandas.DataFrame을 Polars.DataFrame으로 변환하거나 그 반대로 변환하는 기능을 지원한다. Pandas 데이터프레임을 Polars 데이터프레임으로 변환할 때는 **Polars.from\_pandas()** 메서드를 사용한다.

|  |
| --- |
| # pandas 데이터프레임 생성  import pandas as pd  pd\_df = pd.DataFrame({  'A': [1, 2, 3],  'B': ['a', 'b', 'c']  })  print("\nPandas 데이터프레임:")  print(pd\_df)  # pandas 데이터프레임을 Polars로 변환  pl\_from\_pd = pl.from\_pandas(pd\_df)  print("\nPandas 데이터프레임으로부터 생성된 Polars 데이터프레임:")  print(pl\_from\_pd)  [결과] |

텍스트, 스크린샷, 도표, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

## 레시피 5 – Polars Series 생성

**Polars.Series**는 **pandas.Series**와 유사한 1차원 데이터 구조이며, Polars.DataFrame의 개별 컬럼을 표현하는 객체이다. 하나의 데이터 타입을 가지며, 벡터 연산을 지원하여 빠르고 효율적으로 데이터를 처리할 수 있다. **Polars.Series** 객체를 생성하는 방법은 다양하게 있지만, 본 장에서는 예시로 직접 생성, NumPy 배열을 활용하여 생성, 데이터 타입을 직접 지정하여 생성하는 방법을 소개한다.

먼저 직접 생성하는 방법은 다음과 같다.

|  |
| --- |
| # Series 생성  s1 = pl.Series("a", [1, 2, 3])  print("\nSeries s1:")  print(s1)  [결과]  shape: (3,)  Series: 'B' [f64]  [  1  2  3  ] |

이번에는 NumPy 배열을 이용하여 Series를 생성한다.

|  |
| --- |
| # NumPy 배열을 이용하여 Series 생성  s\_np = pl.Series("B", np.array([0.1, 0.2, 0.3]))  print(s\_np)  [결과]  shape: (3,)  Series: 'B' [f64]  [  0.1  0.2  0.3  ] |

이번에는 데이터 타입을 Float32로 지정해서 생성하도록 한다. 이 때, Polars.datatypes.Float32 클래스를 호출한다.

|  |
| --- |
| from Polars.datatypes import Float32  # 데이터 타입을 Float32로 지정  s\_typed = pl.Series("C", [1.1, 2.2, 3.3], dtype=Float32)  print(s\_typed)  [결과]  shape: (3,)  Series: 'C' [f32]  [  1.1  2.2  3.3  ] |

## 레시피 6 – Polars Series 생성

위 코드는 **Polars.Series**의 기본 속성을 확인하는 예제이다. 먼저, "a"라는 이름을 가진 Polars.Series 객체 s1을 생성하고, [1, 2, 3, 4, 5]의 정수 리스트를 데이터로 설정한다. 이후, Series의 주요 속성을 출력하는데, **s1.name**을 통해 시리즈의 이름을 확인하고, **s1.len()**을 사용하여 시리즈의 길이를 출력한다. 또한, **s1.dtype**을 이용해 데이터 타입을 확인하며, **s1.null\_count()**를 통해 결측값(Null)의 개수를 확인할 수 있다. 마지막으로, **s1.n\_unique()**를 사용하여 시리즈 내 고유값 개수를 출력한다. 이를 통해 Polars.Series의 기본 속성을 쉽게 파악할 수 있으며, 데이터 분석 과정에서 유용하게 활용할 수 있다.

|  |
| --- |
| # Series의 기본 속성들  s1 = pl.Series("a", [1, 2, 3, 4, 5])  print("\nSeries의 기본 속성들:")  print(f"이름: {s1.name}")  print(f"길이: {s1.len()}")  print(f"데이터 타입: {s1.dtype}")  print(f"Null 값 개수: {s1.null\_count()}")  print(f"고유값 개수: {s1.n\_unique()}")  [결과]  Series의 기본 속성들:  이름: a  길이: 5  데이터 타입: Int64  Null 값 개수: 0  고유값 개수: 5 |

## 레시피 7 – Polars DataFrame에서 Series로 변형

아래 코드는 Polars.DataFrame을 생성하고, 개별 컬럼을 Polars.Series로 추출하는 예제이다. 먼저, pl.DataFrame()을 사용하여 df라는 데이터프레임을 생성하며, 컬럼 "a"에는 정수 리스트 [1, 2, 3], 컬럼 "b"에는 문자열 리스트 ["x", "y", "z"], 컬럼 "c"에는 불리언 값 [True, False, True]가 들어간다. 그 후, df["컬럼명"]을 이용해 개별 컬럼을 Series로 변환한다. "a" 컬럼은 series\_a, "b" 컬럼은 series\_b, "c" 컬럼은 series\_c로 저장되며, 각각 정수, 문자열, 불리언 타입의 Polars.Series가 된다. 마지막으로, 원본 데이터프레임과 변환된 Series 객체들을 출력하여, DataFrame에서 컬럼을 추출하는 방법과 Series로 변환된 형태를 확인할 수 있다. 이를 활용하면 DataFrame의 특정 컬럼을 독립적으로 다루거나, 벡터 연산 및 데이터 분석을 수행할 수 있다.

|  |
| --- |
| # 데이터프레임 생성  df = pl.DataFrame({  "a": [1, 2, 3],  "b": ["x", "y", "z"],  "c": [True, False, True]  })  print("\n원본 데이터프레임:")  print(df) |

텍스트, 스크린샷, 도표, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

|  |
| --- |
| # 데이터프레임의 컬럼을 Series로 변환  series\_a = df["a"]  series\_b = df["b"]  series\_c = df["c"]  print("\n추출된 Series들:")  print("Series a:")  print(series\_a)  print("\nSeries b:")  print(series\_b)  print("\nSeries c:")  print(series\_c)  [결과]  추출된 Series들:  Series a:  shape: (3,)  Series: 'a' [i64]  [  1  2  3  ]  Series b:  shape: (3,)  Series: 'b' [str]  [  "x"  "y"  "z"  ]  Series c:  shape: (3,)  Series: 'c' [bool]  [  true  false  true  ] |

## 레시피 8 – Polars LazyFrame 생성

### LazyFrame

LazyFrame은 Polars에서 제공하는 **지연 실행(Lazy Execution) 데이터프레임**으로, DataFrame과 달리 즉시 연산을 수행하지 않고, 최적화된 실행 계획을 세운 후 연산을 실행하는 방식이다. 즉, 데이터를 다룰 때 모든 연산을 바로 적용하는 것이 아니라, 필요할 때 한꺼번에 실행하여 성능을 극대화할 수 있다. 이는 대용량 데이터 처리에서 특히 유용하다.

### LazyFrame vs. DataFrame

|  |  |  |
| --- | --- | --- |
| 비교항목 | LazyFrame | DataFrame |
| 실행방식 | 지연 실행(Lazy Execution) | 즉시 실행(Eager Execution) |
| 속도 | 대용량 데이터 최적화 | 작은 데이터에 적합 |
| 메모리 사용 | 최적화 후 실행되어 효율적 | 더 많이 사용 |
| API 지원 | 동일한 연산 지원  .collect() 호출해야 실행 | .filter(), .select() 등 지원 |

아래 코드는 Polars.LazyFrame을 사용하여 딕셔너리 데이터를 기반으로 지연 실행(Lazy Execution) 방식의 데이터프레임을 생성하는 예제이다. 먼저, data라는 이름의 딕셔너리를 정의하고, "a", "b", "c"라는 세 개의 키를 가진 리스트를 값으로 설정한다. "a"는 정수 리스트 [1, 2, 3], "b"는 문자열 리스트 ["x", "y", "z"], "c"는 불리언 값 [True, False, True]로 구성된다. 그 후, pl.LazyFrame(data)를 사용하여 해당 데이터를 LazyFrame 객체 lf로 변환한다.

|  |
| --- |
| # 딕셔너리에서 LazyFrame 직접 생성  data = {  "a": [1, 2, 3],  "b": ["x", "y", "z"],  "c": [True, False, True]  }  lf = pl.LazyFrame(data)  print("\n딕셔너리로부터 직접 생성된 LazyFrame:")  print(lf)  [결과]  딕셔너리로부터 직접 생성된 LazyFrame:  naive plan: (run LazyFrame.explain(optimized=True) to see the optimized plan)  DF ["a", "b", "c"]; PROJECT \*/3 COLUMNS |

 Polars에서 LazyFrame을 DataFrame으로 변환하는 예시를 보여주고 있다. LazyFrame은 실제 연산을 지연시키고 쿼리 최적화를 수행할 수 있게 해주는 Polars의 중요한 기능입니다. 대규모 데이터셋을 처리할 때 특히 유용하다.

|  |
| --- |
| # LazyFrame을 DataFrame으로 수집  df = lf.collect()  print("\n수집된 DataFrame:")  print(df)  [결과] |

텍스트, 도표, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

|  |
| --- |
| # 기본 데이터프레임 생성  df = pl.DataFrame({  "a": [1, 2, 3],  "b": ["x", "y", "z"],  "c": [True, False, True]  })  # LazyFrame으로 변환  lf = df.lazy()  print("\nLazyFrame으로 변환:")  print(lf)  [결과]  LazyFrame으로 변환:  naive plan: (run LazyFrame.explain(optimized=True) to see the optimized plan)  DF ["a", "b", "c"]; PROJECT \*/3 COLUMNS |

위 코드는 Polars.DataFrame에서 **.lazy()**를 사용하여 LazyFrame으로 변환하는 방법을 보여준다.

## 레시피 9 – Polars LazyFrame로 파일 읽어오기

### scan\_csv()

scan\_csv()는 Polars의 **Lazy API** 기능을 활용하여 CSV 파일을 즉시 로드하는 것이 아니라, **지연 평가(Lazy Evaluation) 방식**으로 읽어들이는 함수이다. 이 메서드를 활용하면, 파일 전체를 즉시 읽어들이지 않고, 필요한 데이터만 읽도록 최적화되어 있다.

|  |
| --- |
| # CSV 파일을 LazyFrame으로 읽기  lf = pl.scan\_csv("data/titanic\_dataset.csv")  # 객체 확인  print("데이터 객체 확인")  print(type(lf))  print(lf)  [결과]  데이터 객체 확인  <class 'Polars.lazyframe.frame.LazyFrame'>  naive plan: (run LazyFrame.explain(optimized=True) to see the optimized plan)  Csv SCAN [data/titanic\_dataset.csv]  PROJECT \*/12 COLUMNS |

먼저 읽어들인 객체는 **LazyFrame**인 것을 확인할 수 있다. 그 후에 출력된 navie plan 이하의 절을 간단하게 요약하면 CSV 파일에서 12개의 컬럼을 포함한 데이터를 읽지만, 실제 연산은 아직 실행되지 않음을 의미한다. 상세하게 요약하면 다음과 같다.

* **Csv SCAN [data/titanic\_dataset.csv]**는 CSV 파일을 스캔하지만 실제 데이터 로드는 지연됨.
* **PROJECT \*/12 COLUMNS** → 12개의 컬럼을 포함한 데이터를 선택할 계획이지만, 최적화 과정을 거쳐 불필요한 컬럼은 제거될 수도 있음.

|  |
| --- |
| # LazyFrame을 DataFrame으로 변환하여 첫 몇 행 확인  print("\n데이터 미리보기:")  print(lf.head(1))  [결과]  데이터 미리보기:  naive plan: (run LazyFrame.explain(optimized=True) to see the optimized plan)  SLICE[offset: 0, len: 1]  Csv SCAN [data/titanic\_dataset.csv]  PROJECT \*/12 COLUMNS |

이 실행 계획은 **LazyFrame**을 사용하여 data/titanic\_dataset.csv 파일에서 **최초 1개의 행(row)만 가져오는 연산**을 설명하는 것이다. 기존 출력값과 달라지는 것은 **SLICE[offset: 0, len: 1]**이다**.** 이 부분에 대해상세하게 살펴보면 다음과 같다.

* **SLICE 연산**은 데이터의 일부만 선택하는 것을 의미한다.
* offset: 0 → **첫 번째 행부터 선택**한다.
* len: 1 → **1개의 행만 가져온다.**

다시 정리하면, **데이터프레임의 첫 번째 행을 미리보기 위해 슬라이싱하는 과정**을 나타낸다.

DataFrame의 head()와 같이 출력을 하려면 .collect() 함수가 필요하다.

|  |
| --- |
| # 데이터 확인  print("\n데이터 확인:")  lf.head(1).collect()  [결과] |

텍스트, 폰트, 영수증, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

### read\_csv()

read\_csv()을 활용해서 데이터를 읽어들인 후, 만약 lazy DataFrame으로 변경하고자 한다면, .lazy()를 사용한다. 아래 코드로 확인한다.

|  |
| --- |
| pl\_df = pl.read\_csv("data/titanic\_dataset.csv")  pl\_df.lazy().head(1).collect()  [결과] |

텍스트, 폰트, 라인, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

# Chapter 2. Polars를 활용한 데이터 선택

**실습파일명 : ch02\_selecting\_cols\_filtering.ipynb**

Polars에서 특정 컬럼을 선택하는 방법은 다양하며, 데이터 분석의 목적과 상황에 따라 유연하게 활용할 수 있다. 가장 기본적인 방식은 **select** 함수를 사용하는 것으로, 원하는 컬럼만 명시적으로 지정하여 선택할 수 있다. 이 방법은 전체 데이터 프레임에서 특정 컬럼만 보고 싶을 때 가장 직관적이고 간단한 방식이다.

만약 기존 컬럼을 가공하여 새로운 컬럼을 추가하거나 값을 변형하고 싶다면 **with\_columns**를 사용할 수 있다. 이 함수는 원본 데이터에 영향을 주지 않고, 변형된 컬럼을 추가한 새로운 데이터 프레임을 반환한다. 예를 들어, 특정 컬럼의 값을 두 배로 만들거나, 문자열 컬럼을 대문자로 변환하는 등의 작업을 수행할 때 유용하다.

때로는 특정한 패턴을 가지는 컬럼을 선택해야 할 수도 있다. 이때는 **matches** 함수를 활용하여 컬럼 이름에 특정 문자열이 포함되었는지 확인하고, 해당 컬럼만 선택할 수 있다. 예를 들어, ‘price’라는 단어를 포함하는 모든 컬럼을 선택하거나, 특정 접두사 또는 접미사를 가진 컬럼만 추출하는 방식이 가능하다. 이러한 기능을 사용하면 컬럼이 많을 때도 원하는 데이터를 쉽게 선택할 수 있다.

컬럼을 선택할 때 데이터 타입을 기준으로 필터링하는 방법도 있다. is\_numeric이나 is\_utf8 등의 함수를 사용하면 숫자형 컬럼만 추출하거나, 문자열 데이터를 포함하는 컬럼만 선택할 수 있다. 이 방식은 데이터 전처리 과정에서 특정 타입의 데이터만 분석해야 할 때 매우 유용하다.

또한, 특정 조건을 만족하는 행을 필터링한 후 필요한 컬럼만 선택하는 방법도 있다. filter 함수를 사용하면 특정 값 이상인 행만 선택하거나, 결측치가 없는 행만 남길 수 있다. 이와 같은 방법을 활용하면 데이터를 보다 정제된 상태로 가공할 수 있다.

마지막으로, LazyFrame을 사용할 경우 lazy().select를 활용하여 동적으로 컬럼을 선택할 수도 있다. LazyFrame은 큰 데이터를 다룰 때 성능을 최적화하는 기능을 제공하며, 계산을 지연시켜 불필요한 연산을 최소화할 수 있다. 따라서, 메모리 효율성을 고려해야 하는 경우에는 LazyFrame을 적극 활용하는 것이 좋다.

이처럼 Polars에서는 select, with\_columns, matches, filter 등 다양한 함수를 제공하여 컬럼을 선택하는 여러 방법을 지원한다. 이러한 기능들을 조합하여 사용하면 데이터 분석 과정을 보다 효율적으로 수행할 수 있으며, 복잡한 데이터에서도 필요한 정보를 빠르게 추출할 수 있다.

## 레시피 10 – 컬럼 선택 select()

select를 사용하여 특정 컬럼만 선택할 수 있습니다. 예를 들어, Name, Age, Sex 컬럼만 선택한다.

|  |
| --- |
| import polars as pl  df = pl.read\_csv('data/titanic\_dataset.csv')  # 기본 컬럼 선택  df.select(['Name', 'Age', 'Survived']).head(2)  [결과] |

텍스트, 폰트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

여러 컬럼을 선택 할 때는 pl.col을 활용할 수 있다. pl.col은 **Polars에서 특정 컬럼을 선택하거나 조작할 때 사용하는 함수**다. 컬럼 이름을 문자열로 전달하면 해당 컬럼을 선택할 수 있으며, 여러 컬럼을 리스트 형태로 지정할 수도 있다.

|  |
| --- |
| df.select(pl.col(['Name', 'Age'])).head(2)  [결과] |

텍스트, 폰트, 화이트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

이번에는 모든 컬럼을 선택한다. 이 때, pl.all은 Polars에서 데이터프레임의 모든 컬럼을 선택할 때 사용하는 함수다.

|  |
| --- |
| df.select(pl.all()).head(2)  [결과] |

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

특정 컬럼을 제외하고 선택할 때 pl.exclude를 사용할 수 있다. pl.exclude()는 **특정 컬럼(또는 특정 타입의 컬럼)을 제외하고 나머지 모든 컬럼을 선택할 때 사용하는 함수**이다. 여기에서는 'PassengerId' 컬럼만 제외하고 선택하기로 했다.

|  |
| --- |
| df.select(pl.exclude('PassengerId')).head(2) # 특정 컬럼 제외하고 선택  [결과] |

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Float64 타입의 컬럼만 선택한 후, 상위 1개의 행 출력하는 것을 보여준다. 이와 비슷하게 정수형(pl.Int64) 컬럼만 선택할 수 있고, 또는 문자열(pl.Utf8) 컬럼만 선택할 수도 있다.

|  |
| --- |
| df.select(pl.col(pl.Float64)).head(1) # float64 타입 컬럼들  [결과] |

텍스트, 폰트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

다음 코드는 각 컬럼을 지정하여 세부적으로 변경한 후 출력하는 코드를 보여준다. 컬럼명을 변경하거나, 수치형 컬럼이나 문자열 컬럼을 조작해서 출력할 수 있는 예시 코드로 활용할 수 있다.

|  |
| --- |
| df.select([  pl.col('Age').alias('age\_years'), # 컬럼명 변경  pl.col('Fare').round(2), # 반올림  pl.col('Name').str.to\_uppercase() # 대문자 변환  ]).head(1)  [결과] |

텍스트, 폰트, 라인, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

## 레시피 11 – 컬럼 추가 및 변경 with\_columns()

기존 컬럼을 변형하여 새로운 컬럼을 추가하거나, 기존 컬럼을 변경할 수 있다. 예를 들어, Age 값을 2배로 변환하는 새로운 컬럼을 추가하도록 한다.

|  |
| --- |
| df.with\_columns((pl.col("Age") \* 2).alias("Age\_Doubled")).head(2)  [결과] |

텍스트, 폰트, 라인, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

원본 데이터프레임(df)에 새로운 컬럼을 추가하거나 기존 컬럼을 변경한 후, 상위 1개의 행을 출력하는 역할을 한다. 해당 코드는 앞장의 select() 함수와 유사하지만, 차이점이 존재한다. select() 함수는 특정 컬럼만 선택하여 변환시키는 것이 목적이지만, with\_columns() 함수는 기존 컬럼을 수정하거나 또는 새로운 컬럼을 추가하는 것이 목적이다.

|  |
| --- |
| # 기존 컬럼 변환  df.with\_columns([  pl.col('Age').alias('age\_years'), # 컬럼명 변경  pl.col('Fare').round(2), # 반올림  pl.col('Name').str.to\_uppercase() # 대문자 변환  ]).head(1)  [결과] |

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

## 레시피 12 – 정규표현식을 활용한 컬럼 선택

정규표현식(Regex, Regular Expression)은 **문자열에서 특정한 패턴을 검색, 매칭, 치환, 분리**하는 데 사용하는 강력한 도구이다. Polars에서도 특정 컬럼을 선택할 때, 정규표현식을 활용하여 선택할 수 있다. 다음 예시 코드는 알파벳으로만 이루어진 0~4글자 컬럼을 선택하도록 한다. 각 코드에 대해 설명하면 다음과 같다.

* ^: 문자열 시작
* [a-zA-Z] : 알파벳 대소문자
* {0, 4} : 0~4글자
* $ : 문자열 끝

|  |
| --- |
| df.select(pl.col(r'^[a-zA-Z]{0, 4}$')).head(2)  [결과] |

텍스트, 폰트, 라인, 화이트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

이번에는 S로 시작하는 컬럼만 가져오도록 정규표현식을 작성해본다.

* ^S : S로 시작하는
* .\* : 그 뒤에 어떤 문자(.)가 0개 이상(\*) 올 수 있는
* $ : 문자열 끝

|  |
| --- |
| df.select(pl.col(r"^S.\*$")).head(2)  [결과] |

텍스트, 스크린샷, 디자인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

이번에는 d로 끝나는 컬럼만 가져오도록 정규표현식을 작성해본다.

* ^ : 문자열의 시작
* .\* : 임의의 문자(.) 0개 이상 반복(\*)
* d$ : d로 끝나는 문자열($는 문자열 끝)

|  |
| --- |
| df.select(pl.col(r"^.\*d$")).head(2)  [결과] |

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

## 레시피 13 – Polars selectors

Polars는 데이터프레임의 컬럼을 보다 유연하게 선택할 수 있도록 다양한 선택자(selector)를 제공한다. Selectors는 Polars의 DataFrame 또는 LazyFrame에서 컬럼 이름(name), 데이터 타입(dtype), 기타 속성을 기반으로 보다 직관적인 방식으로 컬럼을 선택할 수 있도록 도와주는 기능이다. 이 선택자들은 기존의 col() 표현식을 확장하고 통합하여, 더 유연하고 직관적인 방식으로 컬럼을 선택하고 연산을 수행할 수 있도록 한다. 또한, 선택된 여러 개의 컬럼에 대해 동일한 표현식을 자동으로 적용(브로드캐스트) 할 수도 있다. 즉, Selectors를 사용하면 단순히 컬럼을 선택하는 것뿐만 아니라, 특정 조건을 만족하는 컬럼들을 한 번에 변환하거나 연산을 수행할 수도 있다. 이러한 기능은 특히 다수의 컬럼을 동적으로 선택하고 처리해야 하는 경우 매우 유용하다. 자세한 것은 공식문서를 참조하도록 한다.[[1]](#footnote-1)

간단한 예시로 숫자형(numeric) 데이터 타입의 컬럼만 선택하여 첫 두행을 출력하는 코드를 작성한다.

|  |
| --- |
| import polars.selectors as cs  # 예시: 숫자형(numeric) 데이터 타입의 컬럼만 선택하여 첫 두 행을 출력  df.select(cs.numeric()).head(2)  [결과] |

텍스트, 폰트, 라인, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

이번에는 matches()를 활용하여 대소문자 구분 없이 ‘a’ 또는 ‘A’가 포함된 컬럼 선택하는 코드를 작성하였다. cs.matches(pattern)는 컬럼 이름이 특정 정규표현식(pattern)과 일치하는 컬럼을 선택하는 Selector 기능이다. 정규표현식에 대한 설명을 하면 다음과 같다.

* (?i) : 대소문자 구분 없이 검색 (case-insensitive)
* .\* : 임의의 문자(.) 0개 이상 반복 (\*)
* a : 문자 'a' 포함 여부 검사
* a.\* : 'a' 앞뒤로 어떤 문자든 있을 수 있음

|  |
| --- |
| import polars.selectors as cs  df.select(cs.matches(r"(?i).\*a.\*")).head(2)  [결과] |

텍스트, 폰트, 라인, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

cs.starts\_with()는 특정 패턴을 포함하는 컬럼을 선택할 수 있는 함수이다. 컬럼 이름이 “P”로 시작하는 모든 컬럼을 선택하는 코드를 작성하면 다음과 같다.

|  |
| --- |
| df.select(cs.starts\_with("P")).head(2)  [결과] |

텍스트, 영수증, 폰트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

## 레시피 14 – 조건에 맞는 행 추출 filter 함수

filter() 함수는 특정 조건을 만족하는 행(row)만 추출할 때 사용된다. 이 함수는 SQL의 WHERE 절과 유사하게 동작하며, 논리 연산자와 함께 다양한 조건을 적용할 수 있다. 숫자, 문자열, NULL 값 등 다양한 조건을 활용할 수 있고, AND (&), OR (|) 연산자를 사용해 복합 조건 적용도 가능하다.

### 특정 값보다 큰 경우

30세 이상 승객을 선택하는 코드를 작성하면 다음과 같다.

|  |
| --- |
| df.filter(pl.col("Age") > 30).head(1)  [결과] |

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

### 특정 값보다 작은 경우

탑승 요금이 20미만인 승객 선택하는 코드를 작성하면 다음과 같다.

|  |
| --- |
| df.filter(pl.col("Fare") < 20).head(1)  [결과] |

텍스트, 폰트, 라인, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

### 특정 값보다 같은 경우

여성 승객만 선택하는 코드를 작성하면 다음과 같다.

|  |
| --- |
| df.filter(pl.col("Sex") == "female").head(1)  [결과] |

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

### 특정 값과 다른 경우

3등석이 아닌 승객을 선택하는 코드를 작성하면 다음과 같다.

|  |
| --- |
| df.filter(pl.col("Pclass") != 3).head(1)  [결과] |

텍스트, 폰트, 라인, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

### 값이 Null인 경우

Age값이 NULL(결측치)인 고객을 선택하는 코드를 작성하면 다음과 같다.

|  |
| --- |
| df.filter(pl.col("Age").is\_null()).head(1)  [결과] |

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

### 값이 NULL이 아닌 경우

Cabin이 NULL이 아닌 승객만 선택하는 코드를 작성하면 다음과 같다.

|  |
| --- |
| df.filter(pl.col("Cabin").is\_not\_null()).head(1)  [결과] |

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

### 특정 문자열 포함

Name 컬럼에 “Smith”가 포함된 승객을 선택하는 코드를 작성하면 다음과 같다.

|  |
| --- |
| df.filter(pl.col("Name").str.contains("Smith")).head(1)  [결과] |

텍스트, 폰트, 라인, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

### 리스트 값 중 하나와 일치

Embarked가 "C" 또는 "Q" 인 승객을 선택하는 코드를 작성하면 다음과 같다.

|  |
| --- |
| df.filter(pl.col("Embarked").is\_in(["C", "Q"])).head(1)  [결과] |

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

### 여러 조건 AND (&)

18세 이상 & 여성 승객을 선택하는 코드를 작성하면 다음과 같다.

|  |
| --- |
| df.filter((pl.col("Age") > 18) & (pl.col("Sex") == "female")).head(1)  [결과] |

텍스트, 폰트, 라인, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

### 특정 문자열 포함

요금이 100 초과 또는 1등석인 승객만 선택하는 코드를 작성하면 다음과 같다.

|  |
| --- |
| df.filter((pl.col("Fare") > 100) | (pl.col("Pclass") == 1)).head(1)  [결과] |

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

# Chapter 3. Polars를 활용한 데이터 가공

**실습파일명 : ch03\_creating\_modifying\_deleting.ipynb**

데이터 분석 과정에서 데이터프레임을 생성하고, 기존 데이터를 수정하거나 불필요한 데이터를 삭제하는 작업은 필수적이다. Polars는 이러한 작업을 효율적으로 처리할 수 있도록 고성능 API를 제공한다. 이 장에서는 Polars를 활용하여 새로운 데이터프레임을 생성하고(Creating), 기존 데이터를 수정하며(Modifying), 특정 컬럼이나 행을 삭제하는(Deleting) 방법을 살펴본다. 컬럼을 생성하는 것은 레시피 11에서 간단하게 주로 다뤘다.

## 레시피 15 – 사칙연산을 활용한 데이터 생성

Fare에 10% 추가 요금을 부과, 클래스별 운임 계산, 가족 규모 계산, 미성년자 여부 판별하여 각각 개별적으로 컬럼을 생성하는 코드를 작성하였다. 새로 생성된 컬럼, Fare\_with\_surcharge, Fare\_per\_class, Family\_size, IsMinor을 확인한다.

|  |
| --- |
| df.with\_columns([  # Fare에 10% surcharge(추가 요금)를 부과한 새로운 컬럼 생성  (pl.col("Fare") \* 1.1).alias("Fare\_with\_surcharge"),    # Pclass(객실 등급)으로 나눈 Fare를 계산 (클래스별 운임 비교용)  (pl.col("Fare") / pl.col("Pclass")).alias("Fare\_per\_class"),    # 가족 규모 계산: 형제/자매(SibSp) + 부모/자녀(Parch) + 본인(1명 추가)  (pl.col("SibSp") + pl.col("Parch") + 1).alias("Family\_size"),    # 미성년자인지 여부: Age가 존재하는 경우 18세 미만이면 True 반환  (pl.col("Age") < 18).alias("IsMinor"),  ]).head(1)  [결과] |

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

## 레시피 16 – 집계 함수를 활용한 새로운 컬럼 생성

with\_columns()를 사용하여 전체 데이터의 최대값, 평균값을 기반으로 새로운 컬럼을 추가하는 코드를 작성한다. 새롭게 생성된 컬럼을 확인한다.

|  |
| --- |
| df.with\_columns([  # Fare 컬럼의 최대값을 새로운 컬럼 "max\_fare"로 저장  pl.col("Fare").max().alias("max\_fare"),  # Fare 컬럼의 평균값을 새로운 컬럼 "ave\_fare"로 저장  pl.col("Fare").mean().alias("ave\_fare"),  # 최대 운임과 평균 운임의 차이를 계산하여 "Fare\_diff" 컬럼 생성  (pl.col("Fare").max() - pl.col("Fare").mean()).alias("Fare\_diff")  ]).head(1)  [결과] |

텍스트, 폰트, 라인, 대수학이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

## 레시피 17 – 행 인덱스 추가 with\_row\_index

데이터프레임에 행 인덱스를 추가하고 싶다면, with\_row\_index()를 사용한다. 기본적으로 새로운 컬럼(index)이 생성되며, 0부터 시작하는 정수값이 할당된다. 기존 원본 데이터프레임(여기서는 df)에는 영향을 주지 않고, 새로운 데이터프레임을 반환한다.

|  |
| --- |
| df.with\_row\_index().head(1)  [결과] |

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

## 레시피 18 – 데이터 수정 주요 메서드

일반적으로는 with\_columns()를 활용하지만, 여기에서는 그 외의 주요 메서드 rename(), map\_elements(), fill\_null()를 활용하여 데이터프레임을 변형하는 방법에 대한 설명을 하고자 한다. 이 메서드들은 데이터를 보다 직관적으로 정리하고, 분석을 위한 전처리 과정에서 필수적으로 사용된다.

### rename()

데이터프레임에서 컬럼명을 변경하는 작업은 가독성을 높이고, 분석 목적에 맞게 데이터를 정리하는 과정에서 중요하다. 예를 들어, "Pclass"라는 컬럼명을 "Passenger\_Class"로 변경하면, 해당 데이터가 승객 등급을 나타낸다는 의미를 보다 명확하게 전달할 수 있다. 또한, "Fare" 컬럼을 "Ticket\_Price"로 변경하면, 운임 정보라는 것이 직관적으로 이해된다. 이 메서드는 기존 데이터프레임의 컬럼명을 유지하면서, 일부 컬럼명을 새로운 이름으로 대체하는 방식으로 동작한다.  
한 번에 여러 개의 컬럼명을 변경할 수도 있으며, 이를 통해 코드의 가독성을 높이고 유지보수성을 개선할 수 있다.

|  |
| --- |
| df.rename({  "Pclass": "Passenger\_Class",  "Fare": "Ticket\_Price"  }).head(1)  [결과] |

텍스트, 폰트, 영수증, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

### map\_elements()

특정 컬럼의 데이터를 변경하거나 가공해야 할 때, map\_elements() 메서드를 활용하면 유용하다. Polars의 map\_elements()는 각 행(요소)에 대해 개별적으로 함수를 적용하는 표현식(Expression) 메서드이다. 예를 들어, 승객의 나이(Age)를 기준으로 "미성년자(Child)" 와 "성인(Adult)"으로 분류하고 싶다면, map\_elements()를 사용하여 각 행의 값에 대해 조건을 적용하고, 변환된 값을 새로운 컬럼으로 추가할 수 있다. 이와 같은 방식은 단순한 수치 데이터뿐만 아니라, 복잡한 문자열 변환, 조건부 데이터 변형 등 다양한 상황에서 활용할 수 있다. 또한, map\_elements()는 각 행을 순회하면서 값을 변경하기 때문에 유연성이 높지만, 연산량이 많을 경우 성능을 고려해야 한다. 나이가 18세 미만이면 ‘Child’, 이상이면 ‘Adult’로 분류하는 코드를 작성한다.

|  |
| --- |
| df.with\_columns(  pl.col("Age")  .map\_elements(lambda x: "Child" if x < 18 else "Adult", return\_dtype=pl.String)  .alias("age\_group")  ).head(1)  [결과] |

텍스트, 폰트, 영수증, 화이트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.공식문서[[2]](#footnote-2)에서 보면 경고문이 있다. Polars 공식 문서에서는 map\_elements()가 Polars의 네이티브 표현식(Expression API)보다 훨씬 느리다고 경고하고 있다. 따라서 가능하면 기존의 표현식(**sqrt(), list.eval(), struct.field()**)을 활용하는 것이 더 빠르고 효율적이다. 효율적인 코드로 다시 작성하면 다음과 같다.

|  |
| --- |
| df.with\_columns(  pl.when(pl.col("Age") < 18)  .then(pl.lit("Child"))  .otherwise(pl.lit("Adult"))  .alias("age\_group")  ).head(1)  [결과] |

텍스트, 폰트, 영수증, 화이트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

실제 차이가 나는지 속도를 비교해보도록 한다. 가상으로 1억개의 데이터를 만들고 비교하는 코드를 구성하였다. 결과를 확인해보면 약 14배 가량 차이가 속도 차이가 나는 것을 확인할 수 있다.

|  |
| --- |
| # 비교  import polars as pl  import timeit  import numpy as np  # 샘플 데이터 생성 (100만 개의 Age 데이터)  df = pl.DataFrame({"Age": np.arange(0, 100000000) % 100})  # map\_elements() 방식  def test\_map\_elements():  df.with\_columns(  pl.col("Age")  .map\_elements(lambda x: "Child" if x < 18 else "Adult", return\_dtype=pl.String)  .alias("age\_group")  ).head(1)  # when().then().otherwise() 방식  def test\_when\_then():  df.with\_columns(  pl.when(pl.col("Age") < 18)  .then(pl.lit("Child"))  .otherwise(pl.lit("Adult"))  .alias("age\_group")  ).head(1)  # 실행 시간 측정  map\_elements\_time = timeit.timeit(test\_map\_elements, number=10)  when\_then\_time = timeit.timeit(test\_when\_then, number=10)  # 결과 출력  print(f"map\_elements() 실행 시간: {map\_elements\_time:.6f}초")  print(f"when().then().otherwise() 실행 시간: {when\_then\_time:.6f}초")  [결과]  map\_elements() 실행 시간: 131.874113초  when().then().otherwise() 실행 시간: 9.131302초 |

### fill\_null() 결측값 채우기

데이터셋에는 NULL 값(결측치, NaN) 이 포함되어 있을 수 있으며, 이러한 값들은 데이터 분석 및 머신러닝 모델 학습 과정에서 문제가 될 수 있다. 따라서, fill\_null() 메서드를 사용하면 NULL 값을 특정 값(예: 평균, 중앙값, 0, 특정 문자열 등)으로 대체할 수 있다. 예를 들어, Age 컬럼에 NULL 값이 존재하는 경우, 해당 값을 전체 데이터의 평균 연령(Age.mean())으로 대체하면 보다 일관성 있는 데이터셋을 유지할 수 있다. 이와 같은 결측값 처리는 데이터 분석의 신뢰성을 높이는 데 필수적인 과정이다. 먼저 결측값 채우기에 앞서 결측값을 확인하는 코드를 작성한다. 실제 Age 컬럼에 177개의 누락값이 있음을 확인했다.

|  |
| --- |
| df.select(pl.all().is\_null().sum())  [결과] |

텍스트, 폰트, 영수증, 화이트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

평균값을 대체하는 코드는 다음과 같다. df2로 저장한 후, 결과를 확인한다.

|  |
| --- |
| df2 = df.with\_columns(  df["Age"].fill\_null(df["Age"].mean())  )  df2.select(pl.all().is\_null().sum())  [결과] |

텍스트, 폰트, 영수증, 화이트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

## 레시피 19 – 컬럼 삭제

데이터 분석을 진행하다 보면 필요 없는 컬럼을 제거해야 하는 경우가 많다. Polars에서는 여러 가지 방법을 통해 컬럼을 삭제할 수 있으며, 사용 목적이나 데이터의 형태에 따라 가장 적합한 방법을 선택하는 것이 중요하다. 이 장에서는 Polars에서 컬럼을 삭제하는 다양한 방법을 설명하고, 각각의 방법이 어떻게 동작하는지 살펴본다.

### drop()을 활용한 컬럼 삭제

컬럼을 삭제하는 가장 직관적인 방법은 drop() 메서드를 사용하는 것이다. drop()은 하나의 컬럼을 삭제할 수도 있고, 여러 개의 컬럼을 동시에 제거할 수도 있다. 예시 코드는 다음과 같다.

|  |
| --- |
| df.drop('Age').head(1)  [결과] |

텍스트, 폰트, 화이트, 영수증이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

다중 컬럼을 삭제할 때는 리스트 형태로 입력한다.

|  |
| --- |
| df.drop(["PassengerId", "Ticket", "Cabin"]).head(1)  [결과] |

텍스트, 영수증, 폰트, 화이트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

### cs.exclude()을 활용한 컬럼 제거

Polars에서는 컬럼 선택을 더 직관적으로 수행할 수 있도록 Column Selectors(cs)를 제공한다. 이 기능을 활용하면 cs.exclude()를 사용하여 특정 컬럼을 제외할 수 있다. 이 때에는 select()과 같이 사용되어야 함을 기억한다.

|  |
| --- |
| import polars.selectors as cs  # cs.exclude  df.select(cs.exclude("Age")).head(1)  [결과] |

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

만약 여러 개의 컬럼을 삭제하고 싶다면, cs.exclude()에 리스트를 전달하면 된다.

|  |
| --- |
| df.select(cs.exclude(["Age", "Fare"])).head(1)  [결과] |

텍스트, 영수증, 폰트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

# Chapter 4. Method Chaining

**실습파일명 : ch04\_method\_chaining.ipynb**

Method Chaining은 여러 개의 메서드를 연속적으로 호출하는 프로그래밍 패턴으로, 각 메서드가 객체를 반환하고 그 객체에 대해 다시 메서드를 호출하는 방식이다. 이를 활용하면 한 줄의 코드에서 여러 작업을 연결하여 수행할 수 있어 코드가 더욱 간결해지고 가독성이 향상된다.

특히 Polars와 같은 데이터 처리 라이브러리에서는 Method Chaining을 적극적으로 활용하면 더욱 효율적인 데이터 처리가 가능하다. 일반적으로 데이터를 변환하거나 필터링할 때, 중간 변수를 여러 개 생성하여 단계별로 처리하는 방식은 코드가 길어지고 복잡해질 수 있다. 그러나 Method Chaining을 사용하면 이러한 중간 변수를 만들 필요 없이 연속적인 연산을 하나의 흐름으로 작성할 수 있다.

이러한 방식은 메모리를 절약하는 데에도 도움이 된다. 중간 변수 없이 연산을 수행하면 불필요한 데이터 복사가 줄어들어 보다 효율적으로 데이터를 다룰 수 있다. 또한, 데이터 처리의 흐름이 직관적으로 표현되기 때문에, 여러 단계를 거치는 데이터 변환 과정에서도 로직을 쉽게 이해하고 유지 보수할 수 있다.

결과적으로 Polars에서 Method Chaining을 활용하면 코드가 짧고 명확해지며, 성능과 가독성을 동시에 높일 수 있다. 따라서 데이터 분석 및 처리 과정에서 Method Chaining을 적극적으로 활용하는 것이 바람직하다.

## 레시피 20 – Method Chaining 코드 소개

Method Chaining은 여러 메서드를 연속적으로 호출하는 프로그래밍 패턴으로, 각 메서드가 객체를 반환하고 그 객체에 대해 다시 메서드를 호출하는 방식이다. 이를 활용하면 중간 변수를 사용하지 않고 연속적인 데이터 변환 작업을 수행할 수 있어 코드가 더욱 간결하고 직관적으로 표현된다.

이번 예제에서는 **iris 데이터셋**을 활용하여, 특정 컬럼을 선택한 후 정렬하는 과정을 Method Chaining을 통해 구현한다. 이 데이터에서 **sepal\_length, sepal\_width, species** 3개의 컬럼만 선택한 뒤, sepal\_length 컬럼을 기준으로 내림차순 정렬하는 코드를 작성한다.

|  |
| --- |
| # 간단한 메서드 체이닝 예시  import polars as pl  import seaborn as sns  # iris 데이터셋 로드  df = pl.from\_pandas(sns.load\_dataset('iris'))  # select와 sort를 체이닝으로 연결  result = (df  .select(['sepal\_length', 'sepal\_width', 'species'])  .sort('sepal\_length', descending=True)  )  print(result.head(3))  [결과] |

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

위 문법에서 주요한 특징은 크게 3가지다. 괄호() 사용, 들여쓰기, 점 연산자다. 먼저 괄호() 사용에 관해 설명하면 다음과 같다. Method Chaining을 여러 줄로 나눌 때 괄호()를 사용하면 전체 체인을 하나의 표현식으로 묶어줄 수 있다.  
이렇게 하면 코드가 더욱 읽기 쉬워지고, 여러 줄에 걸쳐 연결된 메서드들을 명확하게 표현할 수 있다. 또한, 괄호 없이 줄바꿈을 하면 SyntaxError가 발생할 수 있으므로, 긴 체인을 작성할 때는 반드시 괄호를 사용하는 것이 좋다.

두번째는 들여쓰기다. Method Chaining을 사용할 때는 체인된 메서드들이 일정한 들여쓰기로 정렬되어야 한다. 이를 통해 코드 구조를 더욱 명확하게 표현할 수 있으며, 어떤 작업들이 연속적으로 수행되는지 쉽게 이해할 수 있다. 들여쓰기가 일관되지 않으면 코드 가독성이 떨어지고, 유지보수가 어려워질 수 있다. df 객체에서 .select(), .sort() 메서드가 같은 수준에서 들여쓰기 되어 처리 순서가 명확하게 보인다. 이런 방식은 Python의 PEP 8 스타일 가이드에서도 권장되는 코드 스타일이다.

세번째는 점(.) 연산자다. Method Chaining에서는 점(.) 연산자가 매우 중요한 역할을 한다. 점 연산자는 각 메서드를 연결하는 기능을 하며, 이전 메서드의 반환값을 기반으로 다음 메서드를 호출할 수 있도록 한다. 이러한 방식 덕분에, 중간 변수를 만들지 않고도 연속적인 연산을 수행할 수 있다.

## 레시피 21 – 데이터 전처리 파이프라인

이 코드는 Polars의 Method Chaining을 활용해 데이터를 간결하고 직관적으로 처리하는 방법을 보여준다. 컬럼 선택, 필터링, 새로운 컬럼 추가의 세 가지 작업을 한 줄의 흐름으로 표현하여 가독성을 높이고, 중간 변수를 줄이며, 성능을 최적화할 수 있다. 먼저, select()를 사용해 sepal\_length, sepal\_width, petal\_length, species 컬럼만 선택하여 핵심 정보만 유지한다. 이후, filter()를 이용해 sepal\_length > 5.0인 행만 남겨 특정 조건을 만족하는 데이터만 추출한다. 마지막으로, with\_columns()를 통해 sepal\_length과 petal\_length의 평균값을 계산해 avg\_sepal\_length, avg\_petal\_length 컬럼을 추가한다. Polars의 Method Chaining을 활용하면 데이터 전처리를 간결하고 효율적으로 수행할 수 있으며, 불필요한 중간 변수를 생성하지 않아 메모리 사용도 최적화된다.

|  |
| --- |
| df\_processed = (df  .select(['sepal\_length', 'sepal\_width', 'petal\_length', 'species'])  .filter(pl.col('sepal\_length') > 5.0)  .with\_columns([  pl.col('sepal\_length').mean().alias('avg\_sepal\_length'),  pl.col('petal\_length').mean().alias('avg\_petal\_length'  ])  )  print("기본 전처리된 데이터:")  print(df\_processed.head(3))  [결과] |

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

## 레시피 22 – 데이터 집계 함수 파이프라인

Method Chaining에서 가장 유용하게 활용되는 부분이 집계함수를 활용하는 부분일 것이다. 크게 보면 그룹화(group\_by), 집계(agg), 정렬(sort)로 구성되어 있는 것을 확인할 수 있다. 이 코드는 Polars의 Method Chaining을 활용하여 iris 데이터셋을 종(species)별로 그룹화한 후, 평균 및 표준편차를 계산하고 정렬하는 과정을 수행한다. 이러한 과정을 통해 종별 데이터의 평균 및 표준편차를 한눈에 비교할 수 있는 집계 테이블을 생성할 수 있다.

|  |
| --- |
| df\_aggregated = (df  .group\_by('species')  .agg([  pl.col('sepal\_length').mean().alias('avg\_sepal\_length'),  pl.col('petal\_length').mean().alias('avg\_petal\_length'),  pl.col('sepal\_width').std().alias('std\_sepal\_width')  ])  .sort('avg\_sepal\_length', descending=True)  )  print("\n종별 집계 데이터:")  print(df\_aggregated)  [결과] |

텍스트, 영수증, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Polars의 Method Chaining을 활용하다보면 어떤 부분에서 이득이 있는지 궁금할 수 있다. 그러나 실제 속도면에서 50% 개선 된 것을 확인할 수 있다. 아래 코드를 연속해서 확인한다. 특정 컬럼을 지정하고, petal\_length를 기준으로 필터링을 진행하고, petal\_width를 기준으로 내림차순을 정렬한 코드다. 결과는 다음과 같이 나올 것이다.

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Method Chaining을 하지 않을 때는 아래와 같이 코드를 작성할 수 있다. Google Colab이나 Jupyter Notebook에서는 코드셀에 %%time을 입력하면 시간 측정을 해줄 수 있다.

|  |
| --- |
| %%time  df = pl.from\_pandas(sns.load\_dataset('iris'))  # select, filter, sort를 독립적으로 사용  df = df.select(['species', 'petal\_length', 'petal\_width'])  # 그 다음 petal\_length가 4.0보다 큰 행만 필터링  df = df.filter(pl.col('petal\_length') > 4.0)  # 마지막으로 petal\_width를 기준으로 내림차순 정렬  df = df.sort('petal\_width', descending=True)  print("필터링 및 정렬된 iris 데이터:")  print(df.head(5))  [결과]  CPU times: total: 0 ns  Wall time: 3.44 ms |

다음 코드는 Method Chaining을 사용한 코드다.

|  |
| --- |
| %%time  df = pl.from\_pandas(sns.load\_dataset('iris'))  # select, filter, sort를 Method Chaining으로 사용  df = (df  .select(['species', 'petal\_length', 'petal\_width'])  .filter(pl.col('petal\_length') > 4.0)  .sort('petal\_width', descending=True)  )  print("필터링 및 정렬된 iris 데이터:")  print(df.head(5))  [결과]  CPU times: total: 0 ns  Wall time: 2.16 ms |

결과는 동일하지만, Method Chaining을 하게 되면 속도가 2.16 ms로 기존의 코드 실행 속도 3.44 ms보다 1.28ms 약 0.5배 정도 더 빠른 것을 확인할 수 있다. 이는 대용량 데이터를 다룰 때는 속도 차이가 더 크게 날 수 있음을 의미하기도 한다.

## 레시피 23 – 조건부 파이프라인

이 코드의 목적은 Iris 데이터셋을 Polars의 DataFrame 형식으로 변환하고, 특정 조건에 따라 데이터를 분류한 후, 그룹별 개수를 집계하여 정렬하는 것이다. 먼저 전체 코드와 결과를 살펴보도록 한다.

|  |
| --- |
| # iris 데이터셋을 Polars DataFrame으로 변환  df = pl.from\_pandas(sns.load\_dataset('iris'))  # 조건부 파이프라인 구성  df\_conditional = (df  # with\_columns: 새로운 컬럼을 추가  .with\_columns([  pl.when(pl.col('sepal\_length') > 6.0)  .then(pl.lit('large'))  .otherwise(pl.lit('small'))  .alias('size\_category')  ])  # group\_by: 지정된 컬럼들로 그룹화  .group\_by(['species', 'size\_category'])  # agg: 그룹별 집계 연산 수행  .agg(  cnt=pl.len() # 각 그룹의 행 개수를 계산  )  # sort: 지정된 컬럼들로 정렬  .sort(['species', 'size\_category'])  )  print("\n조건부 분류 결과:")  print(df\_conditional)  [결과] |

텍스트, 영수증, 도표, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

이번 예제에서 핵심 코드는 다음과 같다. 구체적으로 하나씩 살펴보면 with\_columns()를 통해서 size\_category라는 컬럼을 추가하는데, 이 때, pl.when().then().otherwise() 문법이 핵심이라고 볼 수 있다. 이 코드는 SQL의 CASE WHEN과 유사한 조건부 로직을 가지고 있다. 먼저, pl.when()에서는 pl.col() 함수를 활용하여 논리구조를 만든다. 만약 참일 경우, then() 함수 내부에 pl.lit()을 입력하는데, 이는 문자열로 변환하겠다는 뜻을 가지고 있다. 만약 거짓일 경우 .othersie() 함수를 활용한다. 즉, ‘sepal\_length가 6.0보다 크다’라는 조건이 참이면, large로 입력하고, 만약 조건이 거짓이면 small로 입력한 후, size\_category로 변경 저장하는 코드다.

|  |
| --- |
| # 조건부 Method Chaining 구성  df\_conditional = (df  # with\_columns: 새로운 컬럼을 추가  .with\_columns([  pl.when(pl.col('sepal\_length') > 6.0)  .then(pl.lit('large'))  .otherwise(pl.lit('small'))  .alias('size\_category')  ]) |

다음 코드는 group\_by 연산을 하는 코드다. 기존 Pandas를 활용한 사용자의 경우, groupby() 함수를 주로 사용하는데, Polars에서는 group\_by() 언더바(\_)가 추가가 되기 때문에 이 부분만 주의하면 코드상의 문법적 오류는 크게 줄일 수 있다. 그룹별 집계 연산을 수행하는 것으로 agg() 함수를 사용하는데, 이 부분은 Pandas 문법과 유사하기 때문에 활용하는데 어렵지 않다. 그룹바이 연산이 완료가 된 이후에는 sort() 함수로 정렬한 것을 확인할 수 있다.

그룹화로 집계 연산하는 것을 수행하는 데 이 부분은 Chapter 10장에서 보다 상세하게 다룰 예정이다.

|  |
| --- |
| # group\_by: 지정된 컬럼들로 그룹화      .group\_by(['species', 'size\_category'])      # agg: 그룹별 집계 연산 수행      .agg(          cnt=pl.len()  # 각 그룹의 행 개수를 계산      )      # sort: 지정된 컬럼들로 정렬      .sort(['species', 'size\_category']) |

# Chapter 5. 데이터 수집 속도 비교

**실습파일명 : ch05\_comparison\_pandas\_polars.ipynb**

데이터 분석 작업에서 **데이터를 얼마나 빠르게 로드하고 처리할 수 있는가**는 매우 중요한 요소이다. 특히, 데이터 크기가 커질수록 처리 속도의 차이는 더욱 극명하게 드러난다. Pandas와 Polars는 모두 강력한 데이터 프레임 라이브러리이지만, 데이터 수집 속도(data ingestion speed)와 처리 방식에서 큰 차이를 보인다.

Pandas는 **행(row) 중심의 연산 방식**을 채택하고 있으며, 데이터를 한꺼번에 메모리에 로드하여 처리한다. 즉, 데이터가 메모리에 한 번에 올라와야 연산이 가능하므로, 메모리가 충분한 환경에서는 빠른 속도를 보일 수 있지만, **메모리보다 큰 데이터를 처리할 때는 성능이 급격히 저하된다**. 특히, 데이터 크기가 기가바이트(GB) 단위를 넘어가면, Pandas는 메모리 부족으로 인해 속도가 느려지거나 실행이 불가능할 수도 있다.

Polars는 Pandas와 달리 **컬럼(column) 중심의 연산 방식**을 사용하며, 멀티스레딩을 기본적으로 활용하여 연산 속도를 극대화한다. 또한, Polars의 가장 뛰어난 기능 중 하나인 **스트리밍 모드(Streaming Mode)**는 지연 실행 **API(Lazy API)**의 일부로, **메모리보다 훨씬 큰 데이터를 배치 단위로 처리할 수 있도록 설계**되었다. 스트리밍 모드를 활용하면 데이터를 한 번에 로드하지 않고 **필요한 부분만 읽어 처리할 수 있기 때문에** 메모리 부담이 크게 줄어든다. 만약 이 기능이 없다면, Pandas처럼 한정된 메모리 환경에서 대용량 데이터를 다루는 것은 불가능했을 것이다. 다만, **모든 지연 실행 연산이 스트리밍 모드에서 지원되는 것은 아니다**. 현재도 지속적으로 개발이 진행 중이므로, 일부 연산은 스트리밍 방식으로 실행되지 않을 수도 있다. 하지만 Polars 엔진은 자동으로 **해당 연산이 스트리밍 모드에서 실행 가능한지 판단**하며, 만약 실행할 수 없다면 **일반(non-streaming) 모드로 자동 전환하여 실행**한다. 향후 Polars의 스트리밍 모드는 더욱 정교해지면서 지원하는 연산의 범위도 점점 확대될 것으로 기대된다.

필자는 Pandas와 Polars와의 성능 비교를 위해 150MB의 데이터를 Kaggle에서 구해서 진행하기로 하였다.[[3]](#footnote-3) 해당 데이터셋은 소스코드 저장소에 없기 때문에 반드시 다운로드를 받아서 실습하기를 바란다.

텍스트, 스크린샷, 웹 페이지, 웹사이트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

필자는 프로젝트 디렉터리에서 data/ch05/ 폴더에 해당 파일을 넣어놓고 실습을 진행하였다. 먼저 파일의 사이즈를 확인하는 코드는 다음과 같이 작성하였다.

|  |
| --- |
| import os  file\_path = 'data/ch05/creditcard.csv'  file\_size = os.path.getsize(file\_path)  def convert\_size(size\_bytes):  for unit in ['B', 'KB', 'MB', 'GB', 'TB']:  if size\_bytes < 1024.0:  return f"{size\_bytes:.2f} {unit}"  size\_bytes /= 1024.0  print(f"File size: {convert\_size(file\_size)}")  [결과]  File size: 143.84 MB |

## 레시피 24 – 데이터 수집 속도 비교

해당 데이터를 가져오는 코드는 다음과 같이 작성할 수 있다. 코드 작성 후, 속도를 비교해보면 알겠지만, Polars가 Pandas에 비해 속도가 약 8~9배 빠른 것을 확인할 수 있다.

|  |
| --- |
| import pandas as pd  import polars as pl  import time  file\_path = 'data/ch05/creditcard.csv'  # Pandas 데이터 로드 시간 측정  start\_time = time.time()  df\_pandas = pd.read\_csv(file\_path)  pandas\_time = time.time() - start\_time  print(f"Pandas 로드 시간: {pandas\_time:.2f}초")  # Polars (Lazy API + Streaming) 데이터 로드 시간 측정  start\_time = time.time()  # Time 컬럼을 Float64로 지정  df\_polars\_lazy = pl.scan\_csv(file\_path, schema\_overrides={"Time": pl.Float64}).collect()  polars\_lazy\_time = time.time() - start\_time  print(f"Polars (Lazy API + Streaming Mode) 로드 시간: {polars\_lazy\_time:.2f}초")  [결과]  Pandas 로드 시간: 1.31초  Polars (Lazy API + Streaming Mode) 로드 시간: 0.15초 |

## 레시피 25 – 메모리 사용 측정 비교

데이터 처리 라이브러리인 **Pandas**와 **Polars**는 각각의 방식으로 데이터를 로드하고 처리한다. 하지만 두 라이브러리의 메모리 사용 방식에는 큰 차이가 있으며, 특히 **Polars의 Lazy API(스트리밍 모드)**는 Pandas보다 훨씬 적은 메모리를 사용하면서도 대용량 데이터를 효율적으로 처리할 수 있다. 이를 확인하기 위해, psutil 라이브러리를 사용하여 **각 라이브러리가 CSV 파일을 로드할 때의 메모리 사용량**을 비교하는 코드를 작성하였다. 전체 코드는 다음과 같다.

|  |
| --- |
| import pandas as pd  import polars as pl  # psutil 라이브러리를 사용하여 프로세스의 메모리 사용량을 측정하는 함수  import psutil  import os  import gc  def get\_memory\_usage():  process = psutil.Process(os.getpid())  return process.memory\_info().rss  def convert\_size(size\_bytes):  for unit in ['B', 'KB', 'MB', 'GB', 'TB']:  if size\_bytes < 1024.0:  return f"{size\_bytes:.2f} {unit}"  size\_bytes /= 1024.0  # 파일 경로  file\_path = 'data/ch05/creditcard.csv'  # 초기 메모리 사용량 측정  initial\_memory = get\_memory\_usage()  # Pandas 메모리 사용량 측정  df\_pandas = pd.read\_csv(file\_path)  pandas\_memory = get\_memory\_usage() - initial\_memory  print(df\_pandas.head(1))  # 메모리 정리  del df\_pandas  \_ = gc.collect()  # 다시 초기 메모리 측정  initial\_memory = get\_memory\_usage()  # Polars Lazy 메모리 사용량 측정  df\_polars\_lazy = pl.scan\_csv(file\_path, schema\_overrides={"Time": pl.Float64})  polars\_memory = get\_memory\_usage() - initial\_memory  print(df\_polars\_lazy.head(1).collect())  del df\_polars\_lazy  \_ = gc.collect()  print(f"Pandas 메모리 사용량: {convert\_size(pandas\_memory)}")  print(f"Polars Lazy 메모리 사용량: {convert\_size(polars\_memory)}")  print(f"메모리 사용량 차이: {convert\_size(abs(pandas\_memory - polars\_memory))}")  [결과] |

텍스트, 폰트, 스크린샷, 문서이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

동일한 결과를 만들어내지만, 실제로 메모리 사용량은 Pandas에 비해서 거의 사용되지 않고 있음을 확인할 수 있다. 그럼 Polars에서 scan\_csv()를 사용하면 메모리 사용량이 증가하지 않는 이유는 무엇일까? scan\_csv()는 Polars의 Lazy API(지연 실행 모드)를 사용하는 함수이다. 이 방식은 데이터를 **즉시 메모리에 로드하지 않고**, 필요할 때만 데이터를 로드하여 처리하기 때문에 get\_memory\_usage()로 측정할 때 메모리 사용량이 거의 변하지 않는다. 즉, 실제 데이터를 로드는 하지 않고 일종의 준비단계로 인식하면 된다.

# Chapter 6. 데이터 Input & Output

**실습파일명 : ch06\_file\_input\_output.ipynb**

데이터 분석과 처리 작업에서는 **다양한 파일 형식을 다룰 줄 아는 것이 필수적**이다. 특히, Polars는 **고성능 데이터프레임 라이브러리**로서 **CSV, Parquet, JSON, Avro, IPC(Arrow), Excel** 등 여러 형식을 효과적으로 지원한다. 이번 장에서는 **Polars 1.22** 버전에서 자주 사용되는 주요 파일 형식을 **효율적으로 읽고 쓰는 방법**을 살펴본다. 이번에 활용하는 데이터셋은 customer\_shopping\_data다. 이번 장에서 사용할 customer\_shopping\_data 데이터셋은 온라인/오프라인 쇼핑몰의 고객 구매 데이터를 담고 있다. 이 데이터셋을 통해 다양한 파일 형식의 입출력 방법을 실습해 보겠다.

* invoice\_no: 거래 고유 번호 (예: "I138884")
* customer\_id: 고객 식별자 (예: "C241288")
* gender: 고객 성별 ("Female", "Male")
* age: 고객 나이
* category: 상품 카테고리 (예: "Clothing")
* quantity: 구매 수량
* price: 상품 가격
* payment\_method: 결제 방식 (예: "Credit Card")
* invoice\_date: 구매 날짜 (예: "5/8/2022")
* shopping\_mall: 쇼핑몰 이름 (예: "Kanyon")

구조화된 거래 데이터로, 다양한 데이터 타입(문자열, 숫자, 날짜)을 포함하며, 실제 비즈니스 환경에서 흔히 볼 수 있는 형태의 데이터다. 다양한 파일 형식으로의 변환과 처리 실습에 적합하며, 데이터 크기가 적절하여 학습용으로 활용하기 좋다. 이 데이터셋을 통해 CSV, Parquet, JSON, Excel, Delta Lake 등 다양한 파일 형식의 특징과 장단점을 실제로 경험해보면서, 각 형식에 따른 데이터 처리 방법의 차이를 이해할 수 있다.

## 레시피 26 – CSV 파일 입출력

CSV(Comma-Separated Values)는 가장 기본적이며 널리 사용되는 데이터 저장 형식이다. 이 형식은 텍스트 기반이며, 각 데이터 값이 쉼표(,)로 구분되어 저장된다. CSV는 사람이 직접 읽기 쉽고, 대부분의 데이터 분석 및 스프레드시트 도구에서 기본적으로 지원된다. Polars에서는 CSV 파일을 다루는 방식으로 read\_csv()과 lazy 방식인 scan\_csv()을 제공한다. 먼저 read\_csv() 방식은 데이터를 즉시 메모리로 로드하여 사용할 수 있으며, 작은 파일을 처리할 때 적합하다. scan\_csv()은 Chapter 7장에서도 설명했듯이 데이터를 한 번에 메모리로 로드하지 않고, 필요할 때마다 읽어오는 방식이다. 본장에서는 두 메서드의 문법 비교 차이를 확인 하는 선에서 읽어주기를 바란다.

### CSV 파일 읽기

먼저 read\_csv() 함수 코드는 기존의 pandas 문법과 동일하다.

|  |
| --- |
| import polars as pl  df\_read = pl.read\_csv('data/ch06/input/customer\_shopping\_data.csv')  df\_read.head(1)  [결과] |

텍스트, 폰트, 라인, 대수학이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

이번에는 scan\_csv() 함수다. 결과는 동일하지만 collect() 함수가 추가로 사용된 것을 기억해야 한다.

|  |
| --- |
| import polars as pl  df\_scan = pl.scan\_csv('data/ch06/input/customer\_shopping\_data.csv')  df\_scan.head(1).collect()  [결과] |

텍스트, 폰트, 라인, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

### CSV 파일 내보내기

CSV파일로 내보낼 때는 write\_csv() 함수를 사용하면 된다. 문법의 차이만 확인하도록 한다. 먼저 객체가 DataFrame일 경우에는 Pandas 문법과 마찬가지로 그대로 사용한다. 필자는 빈 output 폴더를 하나 만들었다.

|  |
| --- |
| df\_read.write\_csv('data/ch06/output/customer\_shopping\_data(read).csv') |

만약 LazyFrame일 경우에는 collect() 데이터 값이 호출된 상태에서 내보내기를 해야 한다.

|  |
| --- |
| df\_scan.collect().write\_csv('…/customer\_shopping\_data(scan).csv') |

### 내보낸 파일 확인

내보낸 파일에 대한 정보를 확인하기 위한 메서드를 하나 만들었다.

|  |
| --- |
| import os  import glob  def print\_file\_info(directory\_path: str) -> None:  """  지정된 디렉토리의 CSV 파일들의 정보를 출력하는 함수    매개변수:  directory\_path: CSV 파일들이 있는 디렉토리 경로  """  # Get all CSV files in output directory  output\_files = glob.glob(f'{directory\_path}/\*')    for file in output\_files:  size = os.path.getsize(file) / (1024 \* 1024) # Convert to MB  print(f"File: {file}")  print(f"Size: {size:.2f} MB")  print(f"Absolute path: {os.path.abspath(file)}")  print()  # 사용자 정의 함수 호출  print\_file\_info('data/ch06/output')  [결과] |

텍스트, 스크린샷, 폰트, 대수학이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

## 레시피 27 – Parquet 파일 입출력

Parquet은 **빅데이터 처리**를 위해 설계된 고성능 **컬럼 기반 저장 형식**이다. CSV와 달리, 데이터가 **컬럼 단위로 저장**되므로 특정 컬럼만 선택하여 읽는 것이 가능하며, 대량의 데이터에서도 빠른 속도로 분석할 수 있다. 또한, **고효율 압축 및 인코딩 기법**이 적용되어 있어 저장 공간을 절약하는 데 유리하다. Parquet의 주요 장점은 다음과 같다.

* **CSV 대비 작은 저장 공간:** 불필요한 데이터를 반복 저장하지 않아 용량이 절약된다.
* **데이터 타입 보존:** 스키마 정보가 포함되어 있어 데이터 타입이 유지된다.
* **필요한 컬럼만 선택 가능:** 전체 데이터를 불러올 필요 없이 특정 컬럼만 읽을 수 있다.
* **파티셔닝 지원:** 데이터를 폴더 단위로 분할하여 저장하면, 필요한 데이터만 빠르게 조회 가능하다.

Polars에서는 **read\_parquet**과 **write\_parquet** 메서드를 사용하여 Parquet 파일을 효율적으로 다룰 수 있다. 먼저 관련 라이브러리를 설치한다. Pyarrow을 설치하면 Arrow 테이블, 배열 등을 쉽게 생성하고 조작할 수 있다. 또한 Parquet 파일 읽기/쓰기 기능을 제공한다. 즉, 대용량 데이터 처리에 대한 고성능 연산을 지원하므로, 효율적으로 메모리를 활용하고자 할 때 유용하다.

|  |
| --- |
| !pip install pyarrow |

### CSV 파일에서 parquet으로 내보내기

Python의 Polars를 활용해서 CSV 파일을 불러와서 Parquet 파일로 내보내는 코드를 작성한다.

|  |
| --- |
| import polars as pl  df\_read = pl.read\_csv('data/ch06/input/customer\_shopping\_data.csv')  # Parquet 파일로 내보내기  df\_read.write\_parquet('data/ch06/input/customer\_shopping\_data.parquet') |

CSV 파일은 텍스트 형식으로 되어 있어, 동일한 데이터를 다룰 때 Parquet 형식에 비해 저장 크기가 더 클 수 있다. Parquet으로 변환 후 읽고 쓰는 속도가 빨라질 수 있으며, 용량 절감 효과도 기대할 수 있다.

### parquet으로 파일 불러오기

Python의 Polars를 활용해서 CSV 파일을 불러와서 Parquet 파일로 내보내는 코드를 작성한다. CSV 파일 불러올 때 두가지 방식으로 불러온 것처럼 Parquet 파일 불러올 때도 두가지 방식이 있다. 먼저 일반적인 DataFrame 방식으로 파일을 불러오도록 한다.

|  |
| --- |
| # 일부 컬럼만 선택해서 가져오기  df\_read\_parquet = pl.read\_parquet(  'data/ch06/input/customer\_shopping\_data.parquet',  columns=['invoice\_no', 'customer\_id', 'gender', 'price'  )  df\_read\_parquet.head(1)  [결과] |

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

이번에는 LazyDataFrame 방식으로 파일을 불러오도록 한다. 결과는 동일하다. 다만 문법은 기존의 Lazy DataFrame 처럼 출력을 하려면 collect() 사용해야 한다.

|  |
| --- |
| # scan\_parquet 사용  df\_scan\_parquet = pl.scan\_parquet(  'data/ch06/input/customer\_shopping\_data.parquet'  ).select(['invoice\_no', 'customer\_id', 'gender', 'price'])  df\_scan\_parquet.head(1).collect()  [결과] |

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

### Hive 형태의 파티션된 parquet으로 파일 불러오기

Apache Hive는 대규모 데이터를 저장하고 분석하기 위한 **데이터 웨어하우스**(Data Warehouse) 솔루션이다. 주로 **Hadoop 생태계** 위에서 동작하며, 대규모 데이터를 쉽고 직관적으로 분석할 수 있도록 SQL과 유사한 쿼리 언어(HiveQL)를 제공합니다. 이를 통해 사용자들은 복잡한 Java MapReduce 코드를 직접 작성하지 않고도, 익숙한 SQL 문법에 가깝게 데이터를 조회하고 가공할 수 있습니다. Hive를 사용하는 가장 중요한 목적 중의 하나는 대규모 배치 처리에 적합하기 때문이며, 이 때, 파티셔닝(Partitioning) 또는 버킷팅(Bucketing)을 활용하게 된다. 큰 테이블을 특정 컬럼 기준으로 물리적으로 분할(파티셔닝)하거나, 해시 함수를 이용해 버킷팅(Bucketing)할 수 있다. 이를 통해 쿼리 성능을 높이고, 필요한 파티션만 조회하는 등의 최적화가 가능하다. 본 예제에서는 사전에 이미 만들어 둔 파티션된 파일을 불러오는 예제와 파티션된 형태로 파일을 내보내는 예제 코드를 확인한다.

먼저 파티션된 파일을 불러오는 코드는 다음과 같다. 주요 매개변수를 확인하면 다음과 같이 정리할 수 있다.

**use\_pyarrow=True**의 경우, Polars가 내부적으로 Parquet를 읽을 때, PyArrow 기반의 리더(reader)를 사용하도록 지정한다. Polars는 자체 C++ 파서 혹은 PyArrow 파서를 사용할 수 있는데, Hive 파티셔닝처럼 특정 옵션을 활용하려면 PyArrow 사용을 지정해야 하는 경우가 있다.

**pyarrow\_options={'partitioning': 'hive'}**의 경우, 디렉터리 경로가 **Hive 스타일**로 파티셔닝되어 있음을 의미한다. 예를 들어, 디렉터리 구조가 country=US/year=2022/와 같이 컬럼명=값 형태로 구성된 경우, PyArrow가 자동으로 파티션 정보를 인식하고 컬럼(country, year)으로 로드하게 된다.

|  |
| --- |
| # Hive 형태의 파티션된 parquet 파일 읽기  df\_read\_parquet\_hive = pl.read\_parquet(  'data/ch06/input/venture\_funding\_deals\_partitioned',  use\_pyarrow=True,  pyarrow\_options={'partitioning': 'hive'}  )  df\_read\_parquet\_hive.head(5)  [결과] |

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

### Hive 형태의 파티션된 parquet으로 파일 내보내기

이번에는 파티션된 형태로 파일을 내보내기를 해보겠다. 먼저 csv 파일을 불러와서 df\_read 객체로 저장한다. 그 다음 write\_parquet 함수를 사용할 것이다.

|  |
| --- |
| # partitioned parquet 파일 내보내기  df\_read = pl.read\_csv('data/ch06/input/customer\_shopping\_data.csv')  df\_read.write\_parquet(  'data/ch06/output/customer\_shopping\_data\_output\_partitioned.parquet',  partition\_by='gender',  ) |

파일을 내보낼 때 가장 중요한 옵션은 partition\_by 매개변수이다. 여기에서는 gender 컬럼 값을 기준으로 파티션을 나누어 저장한다. 예를 들어서 df\_read가 gender 컬럼에 'Male', 'Female' 두 가지 값만 가지고 있다면, 최종적으로 생성되는 디렉터리는 다음과 같은 구조가 된다. 실제 구조를 보면 다음과 같다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

## 레시피 28 – Delta Lake 파일 입출력

**Delta Lake**는 오픈 소스 프로젝트로서, 데이터 레이크(Data Lake) 환경에서 **ACID 트랜잭션(원자성, 일관성, 고립성, 내구성)**을 보장하고, 스키마 관리, Time Travel(버전 관리) 등을 지원하는 스토리지 레이어(Storage Layer)입니다. Apache Spark 환경에서 주로 활용되며, 기존의 Parquet 포맷 기반 데이터 레이크가 가지던 문제점(데이터 무결성, 업데이트·삭제 작업의 어려움, 스키마 진화 문제 등)을 해결해 줄 수 있다.

**Delta Lake**는 테이블 형태의 데이터셋을 더 쉽고, 더 견고하며, 더 높은 성능으로 다룰 수 있도록 해주는 오픈 소스 라이브러리이다. 새로운 데이터가 유입될 때 데이터셋에 행을 추가하거나 제거하거나 업데이트할 수 있으며, 이전 버전의 데이터셋으로 되돌아가는 ‘시간 여행(Time Travel)’ 기능도 제공한다. 또한 작은 파일을 큰 파일로 통합하여 데이터셋 저장을 최적화할 수도 있다. **Delta Lake**는 로컬 파일 시스템이나 클라우드에 저장된 데이터를 관리하기 위해 사용할 수 있다. Pandas, Polars, DuckDB, DataFusion 등의 데이터 처리 라이브러리와 연동이 가능하여, 다양한 환경에서 손쉽게 활용할 수 있다.

**Delta Lake**는 데이터셋을 관리하기 위해 ‘레이크하우스(Lakehouse)’ 프레임워크 방식을 따른다. 이 방식에서 사용자는 DeltaTable 객체로 데이터셋을 제어하고, 실제 파일 관리는 **Delta Lake**가 담당한다. DeltaTable 내부적으로는 데이터가 고성능 Parquet 파일 형식으로 저장되고, 메타데이터는 ‘트랜잭션 로그(Transaction Log)’라고 불리는 일련의 JSON 파일에 기록된다. **Delta Lake**는 원래 Databricks에서 개발한 **Delta Lake**프로토콜을 Rust로 재구현한 것으로, Rust와 Python API를 모두 지원한다. 특히 Java, Spark, Databricks에 대한 종속성이 없어 보다 가벼운 환경에서 사용할 수 있다.

먼저 관련 라이브러리를 설치한다.[[4]](#footnote-4)

|  |
| --- |
| !pip install deltalake |

### CSV 파일에서 Delta로 내보내기

CSV파일에서 df\_read 객체로 불러들인 후, Delta로 내보내기를 시작한다. 매개변수 **mode='overwrite'**는 이미 같은 위치에 데이터가 존재한다면 **덮어쓰기(overwrite)**한다. 즉, 기존 데이터나 메타데이터를 모두 삭제하고 새로운 내용을 기록한다. 또다른 매개변수 **delta\_write\_options={'partition\_by': 'category'}**는 파티션 설정이다. 'partition\_by': 'category'는 DataFrame 내의 "category" 컬럼을 기준으로 데이터를 구분하여 저장한다. 파티셔닝을 해두면, 해당 컬럼을 기준으로 필터링(예: WHERE category = 'B')할 때 **I/O를 절약**하며 **쿼리 성능**을 높일 수 있다.

|  |
| --- |
| df\_read.write\_delta(  'data/ch06/output/customer\_shopping\_data\_delta\_partitioned',  mode='overwrite',  delta\_write\_options={  'partition\_by' : 'category'  }  )  print\_file\_info('data/ch06/output') |

결과를 보면 Category를 분리해서 저장을 한 것을 알 수 있다.

텍스트, 스크린샷, 소프트웨어, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

### Delta 파일 불러오기

이번에는 파일을 불러오도록 한다. 이 때에는 read\_delta() 함수와 scan\_delta()를 활용하면 된다. 파일은 'data/ch06/input/’ 폴더 내에 있는 ‘venture\_funding\_deals\_delta’를 가져오도록 한다. 두 함수의 결과는 동일하게 나오지만, 출력하는 방법은 약간 다르다. 먼저 DataFrame 형태의 객체로 불러들일 때는 read\_delta() 함수를 활용하여 가져오는 것이다.

|  |
| --- |
| # pl.read\_delta  df\_read\_delta = pl.read\_delta('…/venture\_funding\_deals\_delta')  df\_read\_delta.head(1) |

이번에는 Lazy DataFrame 형태의 객체로 불러들일 때는 scan\_delta() 함수를 활용하여 가져오는 것이다.

|  |
| --- |
| # pl.read\_delta  df\_scan\_delta = pl.scan\_delta('…/venture\_funding\_deals\_delta')  df\_scan\_delta.collect().head(1) |

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

### Delta 파일로 내보내기

df\_read\_delta 객체를 output 폴더로 내보내기를 한다. 매개변수는 크게 2가지가 있다. mode='overwrite'는 만약 지정된 경로에 기존 데이터가 있으면, 완전히 덮어쓰게(overwrite) 된다. 이전 버전의 데이터 및 트랜잭션 로그가 모두 대체된다고 보시면 된다. delta\_write\_options={'partition\_by':'Industry', 'schema\_mode':'overwrite'}의 경우, Industry 컬럼을 기준으로 파티셔닝한다. 예를 들어, Industry가 'Tech', 'Healthcare', 'Finance' 등의 값을 가질 경우, 각 값별 디렉터리가 생성되어 데이터가 저장된다. 또 하나의 옵션은 schema\_mode: 'overwrite'인데, 이는 새로운 데이터 프레임의 스키마와 기존 스키마가 다를 때, **기존 스키마를 덮어쓰는 방식**으로 동작한다. 예를 들어, 추가된 컬럼이 있거나 기존 컬럼의 타입이 변경되었더라도, 충돌 없이 새로운 스키마를 적용하게 된다. 스키마 진화(Schema Evolution) 관련 기능으로 볼 수 있으며, 'overwrite' 모드는 기존 스키마 정보를 무시하고 새 스키마를 완전히 반영한다.

|  |
| --- |
| df\_read\_delta.write\_delta(  'data/ch06/output/venture\_funding\_deals\_partitioned',  mode='overwrite',  delta\_write\_options={'partition\_by':'Industry', 'schema\_mode' : 'overwrite'}  )  [결과] |

텍스트, 스크린샷, 소프트웨어, 컴퓨터 아이콘이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

## 레시피 29 – JSON 파일 입출력

Polars는 크게 두 가지 JSON 형태를 다룰 때 유용한 메서드를 제공한다. 단일 JSON 객체와 NDJSON(Newline Delimited JSON) 파일 입출력을 지원한다. JSON과 NDJSON은 두개 모두 JavaScript Object Notation(JSON)을 기반으로 한 데이터 표현 방식이지만 파일 구조와 사용 목적 측면에서 차이가 있다.

### JSON

JSON(JavaScript Object Notation)은 originally 자바스크립트에서 파생되었지만, 언어에 구애받지 않고 광범위하게 사용되는 데이터 표준 포맷이다. 가볍고 사람이 읽기 쉬운 구조를 지니고 있으며, 객체(Object)나 배열(Array)을 통해 여러 형태의 데이터를 표현할 수 있습니다. 예를 들어, 어떤 사용자의 이름과 나이를 관리하고 싶다면 다음과 같은 **JSON 배열** 형태로 표현할 수 있다. 예시 파일 형태는 다음과 같이 출력될 수 있다.

|  |
| --- |
| [  {  "name": "Alice",  "age": 25  },  {  "name": "Bob",  "age": 30  },  {  "name": "Charlie",  "age": 35  }  ] |

위 예시 파일은 [ ... ]를 사용하여 전체 데이터를 하나의 배열로 감싸고, 배열 안에 { "키": "값" } 형태의 JSON 객체들이 나열되어 있다. 이와 달리 하나의 큰 JSON 객체({ ... }) 안에 여러 중첩 필드를 담을 수도 있다. 이처럼 JSON은 다양한 방식으로 계층적·구조적 데이터를 표현할 수 있어, 소규모부터 중규모 규모의 데이터 교환이나 설정 관리에 널리 쓰인다. 보통 RESTful API나 설정 파일, 데이터 교환 포맷 등으로 많이 사용된다. 그러나 JSON 파일이 매우 커질 경우에는 전체를 한꺼번에 파싱해야 하므로 메모리 사용량이 급증할 수 있다. 또한, 배치 작업보다 실시간 스트리밍 처리나 레코드 단위의 증분 처리가 중요한 환경에서는 JSON 파일 하나가 지나치게 커지면 다루기 어렵다. 이런 상황에서 더 적합한 대안 중 하나가 NDJSON이다.

### NDJSON

NDJSON은 **Newline Delimited JSON**의 약자로, 말 그대로 한 줄(line)에 하나의 JSON 객체를 담는 포맷이다. JSON 구조를 유지하면서도, 각 레코드를 줄바꿈 문자로 구분하기 때문에 대용량 처리를 할 때 효과적이다. 일반적인 NDJSON 파일은 다음과 같이 생겼다.

|  |
| --- |
| {"name": "Alice", "age": 25}  {"name": "Bob", "age": 30}  {"name": "Charlie", "age": 35} |

이 예시는 JSON 배열과 겉보기에는 비슷해 보이지만, 중요한 차이는 배열의 대괄호([와 ])나 쉼표가 전혀 없다는 점이다. 대신 각 행이 하나의 JSON 레코드를 완결적으로 표현합니다. 이로 인해 여러 장점이 생긴다. 우선, 파일이 매우 커도 한 줄씩 읽고 처리할 수 있으므로, 스트리밍 형태로 데이터를 파싱할 수 있다. 예컨대 로그 파일에서 이벤트가 계속 추가되는 상황이라면, 새로운 JSON 객체를 한 줄씩 이어붙이는 것만으로도 손쉽게 데이터가 축적된다. 또한 분산 처리 환경에서 라인 단위로 작업을 나누기 쉬워, 대규모 배치 파이프라인이나 실시간 분석 시스템에서 NDJSON이 빈번히 사용된다.

다만, NDJSON은 사람이 직접 보기에는 상대적으로 불편한 면이 있을 수 있다. JSON 배열처럼 예쁘게 들여쓰기를 하거나, 전체 구조를 한눈에 파악하기가 쉽지 않다. 그럼에도 불구하고 로그나 이벤트 처리, 대규모 데이터 스트리밍 등 실무에서는 이 포맷을 선호하는 사례가 많다.

### JSON과 NDJSON의 선택 기준

JSON과 NDJSON 모두 내부적으로 **JSON 형식**을 사용한다는 공통점을 지니지만, 실제 활용 목적이 조금 다르다. 일반적으로 크고 복잡하지 않은 데이터를 한 번에 주고받을 때는 JSON 배열이나 객체를 사용해도 충분하다. 그러나 데이터를 한 줄씩 받아들이거나 파일이 매우 커서 전체를 한꺼번에 메모리에 올리기 어려운 경우에는 NDJSON이 훨씬 유리하다.

예를 들어, 웹 API의 응답을 단순히 전달하고자 할 때나, 애플리케이션 설정 파일처럼 완결된 형태의 구조가 필요한 경우엔 일반 JSON이 적합하다. 반면에 로그 수집 서버에서 계속 쌓이는 이벤트 데이터를 저장하거나, 빅데이터 분석용 파이프라인에서 레코드 단위로 분산 처리해야 한다면 NDJSON으로 저장하는 것이 합리적인 선택이다.

추가로, JSON 배열 형식을 NDJSON으로 변환하거나, NDJSON을 다시 JSON 배열로 변환하는 방법도 어렵지 않다. 예를 들어 파이썬의 표준 라이브러리 json을 이용하면 한 줄씩 읽어서 객체 리스트를 만든 다음, 리스트 전체를 JSON 배열로 직렬화하는 식으로 쉽게 변환할 수 있다.

### Polars DataFrame에서 JSON으로 내보내기

아래 코드는 df\_read라는 DataFrame을 JSON 파일로 저장한 뒤, 해당 파일에서 일부 레코드를 로드하고 출력하는 과정을 보여준다. 이 때에는 write\_json() 함수를 사용한다.

|  |
| --- |
| # 내보낸 JSON 파일 출력  import json  df\_read.write\_json('data/ch06/output/customer\_shopping\_data.json')  with open('data/ch06/output/customer\_shopping\_data.json', 'r') as f:  json\_data = json.load(f)  # 1-2번째째 레코드만 출력  json\_formatted\_str = json.dumps(json\_data[0:2], indent=2)  print(json\_formatted\_str)  [결과] |

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

아래 코드는 df\_read라는 DataFrame을 **NDJSON 포맷**으로 저장한 뒤, 해당 파일에서 1-5번째 (레코드)만 출력하는 예시이다. 이 때에는 write\_ndjson() 함수를 사용한다.

|  |
| --- |
| # NDJSON  df\_read.write\_ndjson('data/ch06/output/customer\_shopping\_data.jsonl')  # NDJSON 파일 출력  with open('data/ch06/output/customer\_shopping\_data.jsonl', 'r') as f:  # 1-5번째 레코드만 출력  for i, line in enumerate(f):  if i < 5:  print(line, end='')  [결과] |

텍스트, 폰트, 화이트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

### JSON과 NDJSON 파일 읽기

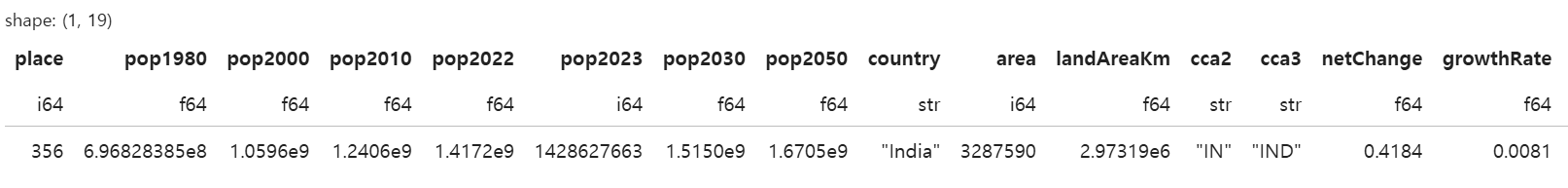
이번에는 input 폴더에 있는 두개의 파일(world\_population)을 연속해서 가져오도록 한다. polars에서는 read\_json()과 read\_ndjson()를 제공하고 있다. 결과는 동일하기 때문에 두개의 코드를 연속해서 확인한다.

먼저 json 파일을 불러온다.

|  |
| --- |
| df\_json\_read = pl.read\_json('data/ch06/input/world\_population.json')  df\_json\_read.head(1) |

두번째는 jsonl 파일을 불러온다.

|  |
| --- |
| df\_jsonl\_read = pl.read\_ndjson('data/ch06/input/world\_population.jsonl')  df\_jsonl\_read.head(1) |



Lazy DataFrame으로 파일을 읽고 있다면, scan\_json()과 scan\_njson() 함수를 사용하면 될 것이다.

## 레시피 30 – Excel 파일 입출력

Pandas와 마찬가지로 Polars에서도 Excel 파일 입출력은 쉽게 할 수 있다. 그런데, Excel 파일 입출력을 하려면 사전에 Excel 관련 파이썬 라이브러리들을 설치해야 한다. 먼저 주요 라이브러리들을 설치하도록 한다.

Google Colab이나 Jupyter Notebook 셀 모드에서 라이브러리 설치 시, !를 입력한 상태에서 다음 코드를 실행한다. 만약 일반 Local에서 터미널에서 설치를 원한다면 !만 제거하면 된다.

|  |
| --- |
| !pip install xlsx2csv xlsxwriter openpyxl |

Polars에서는 각각 설치된 라이브러리들을 활용하여 실행되기 때문에 기능적으로 큰 차이점은 느끼지 못할 수 있다. 그러나 각 라이브러리들은 모두 Excel 파일을 다루지만, 기능적인 면에서 약간씩 차이가 존재한다. 이론적인 측면에서 접근하여 세부적으로 살펴보도록 하겠다.

### xlsx2csv[[5]](#footnote-5)

xlsx2csv는 “XLSX에서 CSV로 변환”하는 데 특화된 간단하면서도 강력한 도구다. Excel 파일의 내부 구조를 깊이 파악하거나 복잡한 처리를 수행하기보다는, 말 그대로 **XLSX 형태의 시트 데이터를 CSV 파일로 떨어뜨려주는** 역할에 집중한다. 이 라이브러리는 Excel의 셀 서식, 차트, 매크로 등 복잡한 기능을 해석하지 않고, 순수한 문자열과 숫자 값만 추출해 CSV로 만드는 것을 목표로 한다.

이러한 특성 덕분에 xlsx2csv는 **읽기 전용** 도구로서 대규모 혹은 다량의 Excel 파일을 한꺼번에 변환해야 할 때 매우 유용하다. 가령 수십만 줄이 들어 있는 Excel 파일을 매일 새벽에 분석 시스템에 넘겨야 한다면, 우선 xlsx2csv로 CSV 파일을 만든 뒤, 이를 Pandas나 Spark 같은 분석 도구에서 곧바로 불러올 수 있다. Excel 파일을 직접 열지 않고도 빠른 변환이 가능하므로, 자동화 파이프라인을 구성할 때 시간이 크게 절약된다.

또한 xlsx2csv는 **커맨드 라인**(CLI) 툴로 사용하기에도 적합하다. 시스템 환경에서 “xlsx2csv input.xlsx output.csv” 형태로 간단히 호출할 수 있고, 여러 시트를 각각 CSV 파일로 나누는 등의 기능도 지원한다. 다만, 이 라이브러리는 **기존 XLSX 파일을 수정**하거나 **새로운 XLSX 파일을 생성**하는 기능은 전혀 제공하지 않는다. 따라서 “Excel 파일에서 데이터를 읽고, 그 결과를 다시 Excel로 저장하고 싶다”라는 시나리오에는 맞지 않는다. xlsx2csv는 오직 변환에 집중하므로, **일괄 처리가 필요한 큰 파일**을 효율적으로 CSV파일로 변환하여 다른 워크플로우에 넘길 때 가장 잘 쓰이는 도구라고 할 수 있다.

### xlsxwriter[[6]](#footnote-6)

xlsxwriter는 이름에서 드러나듯, “Excel(xlsx) 파일을 작성”하기 위한 파이썬 라이브러리이다. 이 라이브러리는 오로지 **쓰기 전용**이므로, 이미 존재하는 Excel 파일을 열어보거나 수정하는 기능은 제공하지 않는다. 대신 새로운 워크북(Workbook)을 만들고, 원하는 워크시트(Worksheet)를 생성하거나 삭제할 수 있으며, 셀 서식, 스타일 지정, 차트 삽입, 수식 입력, 피벗 테이블 구성 등에 이르기까지 **Excel 보고서 작성에 필요한 거의 모든 작업**을 코드로 자동화할 수 있다.

특히 xlsxwriter가 주목받는 이유 중 하나는 **Excel의 시각적이고 복합적인 요소들을 매우 세밀하게 제어**할 수 있다는 점이다. 엑셀 문서에서 셀 배경색, 폰트 스타일, 셀 테두리, 셀 병합 등 온갖 기능을 파이썬 코드로 손쉽게 적용할 수 있으니, 반복적인 보고서를 자동 생성해야 하는 업무 환경에서 각광받는다. 예를 들어, 매주 월요일마다 새 데이터를 바탕으로 색상과 차트가 포함된 Excel 요약 보고서를 자동 발행해야 한다면, xlsxwriter 스크립트를 작성해두고 cronjob 등을 통해 주기적으로 실행하면 됩니다.

파일 생성 전용 라이브러리이므로, 로직 상 “Excel 데이터를 읽어 처리한 뒤, 일부 수정만 해서 다시 저장한다”는 종류의 시나리오에는 적합하지 않는다. 그러나 “Python의 다른 도구나 DB, 혹은 CSV 파일로부터 읽어온 결과를 예쁘게 꾸민 Excel 보고서 형태로 만들고 싶다”는 요구 사항이라면, xlsxwriter만큼 간편하고 강력한 대안을 찾기 어려울 것이다.

### openpyxl[[7]](#footnote-7)

openpyxl은 **XLSX 파일의 읽기와 쓰기**를 모두 지원하는 대표적인 라이브러리이다. Python에서 Excel 파일을 다룰 때 가장 널리 쓰이는 도구 중 하나이기도 하며, 실제로 pandas의 read\_excel이나 to\_excel 함수도 내부적으로 openpyxl을 이용해 Excel 파일을 처리하는 경우가 많다.

openpyxl이 가진 가장 큰 장점은 “이미 존재하는 엑셀 파일을 **열어서** 특정 부분을 **수정**한 뒤 다시 **저장**할 수 있다는 점”이다. 예를 들어, 사내에서 배포된 템플릿 양식에 매주 새로운 데이터를 추가해야 하는 상황을 가정해보자. 이때 openpyxl을 사용하면, 템플릿 파일을 로드해 필요한 셀에 값을 써넣고, 차트나 수식, 서식 등을 유지한 상태로 업데이트된 파일을 저장할 수 있다. 차트, 피벗 테이블, 조건부 서식과 같이 Excel에서 흔히 쓰이는 요소들도 어느 정도까지는 openpyxl에서 직접 조작이 가능하다.

다만, **모든 기능을 100% 다 지원**한다고 말하기는 어렵습니다. Excel에서 제공하는 일부 고급 기능이나 복잡한 VBA 매크로 등은 수정이 제한적일 수 있다. 또한 라이브러리 자체가 방대한 기능을 제공하다 보니, 속도나 메모리 사용 면에서 다소 무거울 수 있다. 그럼에도 불구하고 “Excel을 외부 프로그램에서 직접 열어볼 수 없는 서버 환경”에서 파일 내용을 가공해야 하거나, “배포된 양식을 유지하면서 데이터만 갱신”해야 하는 업무에서는 openpyxl이 사실상 표준으로 자리 잡은 상태다. 정리하면, openpyxl은 **Excel 파일을 원본 그대로 열어서 필요한 조작을 하고 다시 저장**해야 하는 시나리오에 최적화된 선택지다. 읽기와 쓰기를 모두 지원하기 때문에, 엑셀 기반의 다양한 프로세스(예: 통합, 분할, 필터링)를 파이썬 코드 한 곳에서 쉽게 제어할 수 있다.

### Excel 관련 라이브러리 선택 기준

이처럼 xlsx2csv, xlsxwriter, openpyxl은 저마다 다른 목적과 강점을 지니고 있다. 우선 **xlsx2csv**는 XLSX 파일을 간단히 CSV로 변환하기 위한 “읽기 전용” 변환 도구로서, 매일 쏟아지는 Excel 파일을 빠르게 전처리해야 하는 자동화 파이프라인에 적합하다. 반면 **xlsxwriter**는 오로지 파일 “생성”에 최적화된 라이브러리라, “보고서 작성 자동화”나 “Excel 문서를 예쁘게 꾸민 뒤 완성된 형태로 만들기”에 뛰어난 강점을 발휘한다. 마지막으로, **openpyxl**은 읽기와 쓰기가 모두 가능하므로, “이미 존재하는 XLSX 파일을 열어서 내용만 갱신”하거나 “파이썬에서 Excel 서식을 유지하면서 데이터를 조금씩 가공하고 싶다”는 상황에 알맞다.

결국 어떤 라이브러리를 선택할지는 “Excel 파일을 어떻게 다루고 싶은가?”라는 질문에 달려 있다. 순수 변환 목적이라면 xlsx2csv, 새로운 문서를 생성하는 보고서 업무에는 xlsxwriter, 기존 파일 편집에는 openpyxl을 고려하면 된다. 필요하다면 두 가지 이상의 라이브러리를 결합하여 대규모 파일을 CSV로 변환한 뒤, 가공을 마치고 다시 멋진 Excel 보고서로 만드는 식으로 파이프라인을 구성할 수도 있을 것이다.

### Polars DataFrame에서 Excel로 내보내기

아래 코드는 **Polars** 라이브러리를 이용해 CSV 파일을 읽고, 그 결과를 **Excel(xlsx) 파일**로 저장하는 예시이다. Polars에서 있는 write\_excel() 함수를 사용하였다.[[8]](#footnote-8) 코드의 결과값, 마지막 줄에서 반환되는 <xlsxwriter.workbook.Workbook at 0x134cd9ce180>는 **xlsxwriter** 라이브러리가 내부적으로 생성한 Workbook 객체를 가리키며, 메모리 주소가 함께 표시된 것이다. 즉, xlsxwriter 라이브러리가 활용된 것임을 확인할 수 있다.

본 코드에서는 매개변수가 2개가 사용되었다. 먼저 매개변수 worksheet='Sheet1'는 **엑셀 워크시트 이름**을 지정하는 것이다. 두번째 매개변수 header\_format={'bold': True}는 엑셀의 **열 헤더를 볼드체**로 설정하겠다는 의미이다. 이는 내부적으로 **xlsxwriter** 라이브러리를 통해 적용된다.

|  |
| --- |
| df\_read = pl.read\_csv('data/ch06/input/customer\_shopping\_data.csv')  df\_read.write\_excel(  'data/ch06/output/customer\_shopping\_data.xlsx',  worksheet='Sheet1',  header\_format={'bold': True}  )  [결과]  <xlsxwriter.workbook.Workbook at 0x134cd9ce180> |

### Excel 파일 읽기

내보낸 파일을 불러오도록 한다. **Polars** 라이브러리를 사용하여 기존에 생성된 Excel 파일을 다시 읽어오는 예시이다. Polars에 있는 read\_excel() 함수를 사용한다.[[9]](#footnote-9) 특히 **sheet\_name**, **engine**, **read\_options** 등 다양한 옵션을 통해 어떤 시트에서 어떻게 데이터를 파싱할지 지정할 수 있다. 주요 매개변수는 engine='openpyxl' 인데, default는 calamine 라이브러리를 활용하지만, 경우에 따라서는 'openpyxl', ‘xlsx2csv’ 등을 사용할 수 있다. 자세한 것은 공식문서의 내용을 참고하기를 바란다. read\_options={'try\_parse\_dates': True}는 읽기 과정에서 **날짜로 해석 가능한 값**을 **자동으로 날짜 타입**으로 파싱하려고 시도한다는 의미다. 이 옵션을 활성화하면, "2023-01-01"처럼 날짜 형태로 보이는 문자열 컬럼을 Polars가 적절히 날짜형 혹은 날짜·시간형 필드로 변환할 수 있다.

이 코드는 **Sheet1** 시트의 데이터를 **날짜 컬럼을 자동 변환**하도록 설정한 뒤 **Polars DataFrame**에 로드한다. 그 결과 df\_read\_excel에는 파싱된 데이터가 담기게 되며, 이후 Polars의 다양한 메서드(집계, 필터링, 조인 등)를 활용해 데이터를 분석할 수 있다.

|  |
| --- |
| df\_read\_excel = pl.read\_excel(  'data/ch06/output/customer\_shopping\_data.xlsx',  sheet\_name='Sheet1',  engine='openpyxl', # xlsx2csv, calamine  read\_options={'try\_parse\_dates' : True}  )  df\_read\_excel.head(1)  [결과] |

텍스트, 폰트, 라인, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

# Chapter 1. 책을 출간하면서

## 책을 집필하면서 느끼는 소회

## 무엇을 배울 수 있는가

# Chapter 2. 개발환경설정

본 교재의 독자는 기본적으로 Pandas를 활용할 수 있다고 가정하므로, 별도의 Python 개발환경 설정 설명은 생략한다.[[10]](#footnote-10) 다만, 원활한 실습을 위해 Google Colab에서도 실행할 수 있도록 코드를 구성하였다. Google Colab은 브라우저 기반의 Jupyter Notebook 환경으로, 별도의 설치 없이 Python을 실행할 수 있는 장점이 있다. 또한, 실행 환경이 클라우드에 존재하므로 로컬 컴퓨터의 환경 설정과 무관하게 일관된 실행 결과를 얻을 수 있다.

## Google Colab

### Colab에서 패키지 설치 및 업그레이드

본 교재에서 활용하는 주요 패키지는 **Pandas**와 함께 **Polars**이다. Polars는 Pandas보다 속도가 빠르고 메모리 사용량이 적어 대량의 데이터를 처리하는 데 유리하다. Colab에서 최신 버전을 사용하려면 아래 명령어를 실행한 후 세션을 다시 시작해야 한다.Google Colab이나Jupyter Notebook에서 라이브러리를 설치 할 때는 ‘!’를 앞에 명시적으로 붙여줘야 한다.위 명령어를 실행한 후, [런타임] → [세션 다시 시작] 을 선택하면 변경된 버전이 적용된다.

|  |
| --- |
| !pip install Polars==1.22 --quiet |

### Colab에서 패키지 버전 확인

설치된 패키지의 버전을 확인하려면 다음 명령어를 사용하면 된다. 2025년 2월 기준, Google Colab의 기본 Polars 버전은 **1.9.0**이다. 최신 버전의 변경 사항이 필요할 경우 공식 문서를 참고하면 된다.

|  |
| --- |
| import polars as pl  pl.\_\_version\_\_  [결과]  ‘1.22.0’ |

## 주요 라이브러리

본 교재에서 활용한 주요 라이브러리는 다음과 같다.

실습파일명 : requirements.txt

|  |
| --- |
| Polars==1.22.0  pandas==2.2.3  numpy==2.0.2  matplotlib==3.9.4  seaborn==0.13.2  scikit-learn==1.6.1  graphviz==0.20.3  statsmodels==0.14.4  jupyterlab==4.3.5  pyarrow==19.0.0  deltalake==0.24.0  xlsx2csv==0.8.4  xlsxwriter==3.2.2  openpyxl==3.1.5  xlrd==2.0.1 |

데이터 분석을 수행할 때 사용되는 라이브러리는 매우 다양하다. 본 교재에서는 **데이터 처리, 분석, 시각화, 머신러닝**을 포함한 다양한 작업을 수행할 수 있도록 필요한 패키지를 정리하였다. 특히, 로컬 환경에서도 동일한 코드를 실행할 수 있도록 주요 라이브러리의 버전 정보를 명시하였다. 각 라이브러리에 대한 설명은 다음 표에서 대략적으로 확인 할 수 있다.

|  |  |
| --- | --- |
| 라이브러리 | 주요기능 |
| Polars | 대용량 데이터 처리를 위한 고속 데이터프레임 연산 |
| pandas | 전통적인 데이터 분석 및 테이블 형태 데이터 조작 |
| numpy | 벡터 및 행렬 연산을 위한 과학 연산 라이브러리 |
| matplotlib | 기본적인 데이터 시각화 도구 |
| seaborn | 통계적 데이터 시각화 |
| scikit-learn | 머신러닝 모델 구축 및 평가 |
| graphviz | 그래프 및 네트워크 시각화 |
| statsmodels | 통계적 분석 및 회귀 모델링 |
| jupyterlab | Jupyter Notebook의 확장형 개발 환경 |
| pyarrow | Apache Arrow 기반의 고속 데이터 처리 |
| deltalake | Delta Lake 지원 (데이터 레이크 관리) |
| xlsx2csv | XLSX 파일을 CSV로 변환 |
| xlsxwriter | Excel 파일 생성 및 편집 |
| openpyxl | Excel 파일 읽기/쓰기 |
| xlrd | XLS 파일 읽기 지원 |

각 개인 Local에서 개발환경을 설정 할 때는 프로젝트 경로에서 다음과 같이 명령어를 실행하면 된다.

|  |
| --- |
| pip install -r requirements |

### Local에서의 패키지 버전 확인

주요 라이브러리의 설치 버전을 확인하고 싶다면 다음과 같은 코드를 작성하면 확인할 수 있다.

|  |
| --- |
| import pkg\_resources  # requirements.txt 파일에 명시된 패키지들의 설치 여부와 버전을 확인하는 코드  # 각 패키지에 대해 현재 환경에 설치된 버전을 출력하며, 설치되지 않은 경우 메시지 표시  packages = [  'Polars', 'pandas', 'numpy', 'matplotlib', 'seaborn',  'scikit-learn', 'graphviz', 'statsmodels', 'jupyterlab',  'pyarrow', 'deltalake', 'xlsx2csv', 'xlsxwriter', 'openpyxl', 'xlrd'  ]  # 각 패키지를 순회하면서 버전 정보 확인  for package in packages:  try:  # pkg\_resources를 사용하여 설치된 패키지의 버전 정보 조회  version = pkg\_resources.get\_distribution(package).version  print(f"{package}: {version}")  except pkg\_resources.DistributionNotFound:  # 패키지가 설치되지 않은 경우 예외 처리  print(f"{package}: 설치되지 않음")  [결과]  Polars: 1.22.0  pandas: 2.2.3  numpy: 2.0.2  matplotlib: 3.9.4  seaborn: 0.13.2  scikit-learn: 1.6.1  graphviz: 0.20.3  statsmodels: 0.14.4  jupyterlab: 4.3.5  pyarrow: 19.0.0  deltalake: 0.24.0  xlsx2csv: 0.8.4  xlsxwriter: 3.2.2  openpyxl: 3.1.5  xlrd: 2.0.1 |

1. *Selectors : https://docs.pola.rs/api/python/stable/reference/selectors.html* [↑](#footnote-ref-1)
2. polars.Expr.map\_elements : https://docs.pola.rs/api/python/dev/reference/expressions/api/polars.Expr.map\_elements.html [↑](#footnote-ref-2)
3. Credit Card Fraud Detection : https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud [↑](#footnote-ref-3)
4. Delta Lake : https://delta-io.github.io/delta-rs/ [↑](#footnote-ref-4)
5. https://github.com/dilshod/xlsx2csv [↑](#footnote-ref-5)
6. https://github.com/jmcnamara/XlsxWriter [↑](#footnote-ref-6)
7. https://openpyxl.readthedocs.io/en/stable/ [↑](#footnote-ref-7)
8. write\_excel() 함수에 대한 구체적인 설명 : https://docs.pola.rs/api/python/stable/reference/api/polars.DataFrame.write\_excel.html [↑](#footnote-ref-8)
9. read\_excel() 함수에 대한 구체적인 설명 : https://docs.pola.rs/api/python/dev/reference/api/polars.read\_excel.html [↑](#footnote-ref-9)
10. 개발환경설정이 궁금하다면 다음 필자의 유투브에 접속해서 확인한다. https://youtu.be/BqkwJ5zCMb0?si=ylqt185WSSPJ-x9q [↑](#footnote-ref-10)