Numpy Array 데이터 조작

.reshape()

- 배열을 다른 shape으로 바꿔줌, 요소만 안 사라진다면 얼마든지 변형 가능

```
# (2, 3) 형태의 2차원 배열 만들기
a = np.array([[1, 2, 3],
              [4, 5, 6]])
# (3, 2) 형태의 2차원 배열로 Reshape
b = a.reshape(3, 2)
print(b)
'''
[[1 2]
 [3 4]
 [5 6]]
'''
# 1차원 배열로 Reshape
c = a.reshape(6)
print(c)
'''
[1 2 3 4 5 6]
'''

# -1로 지정할 경우, 해당 행이나 열은 자동으로 계산해줌
# reshape(m, -1), reshape(-1, n) 형태로 지정하여 Reshape 가능
print(a.reshape(1, -1), end='\n\n')
'''
[[1 2 3 4 5 6]]
'''
print(a.reshape(2, -1), end='\n\n')
'''
[[1 2 3]
 [4 5 6]]
'''
print(a.reshape(3, -1), end='\n\n')
'''
[[1 2]
 [3 4]
 [5 6]]
'''

# 4행이나 5행으로 설정할 경우, 열과 갯수가 맞지 않아서 오류가 발생함
#print(a.reshape(4, -1))
#print(a.reshape(5, -1))

print(a.reshape(6, -1))
'''
[[1]
 [2]
```

```
 [3]
 [4]
 [5]
 [6]]
'''
```

인덱싱

- 1차원 배열은 리스트와 방법이 같으므로 설명을 생략합니다.
- **배열[행, 열]** 형태로 특정 위치의 요소를 조회합니다.
- **배열[[행1,행2,..], :]** 또는 **배열[[행1,행2,..]]** 형태로 특정 행을 조회합니다.
- **배열[:, [열1,열2,...]]** 형태로 특정 열을 조회합니다.
- **배열[[행1,행2,...], [열1,열2,...]]** 형태로 특정 행의 특정 열을 조회합니다.

```
# (3, 3) 형태의 2차원 배열 만들기
a = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
# 요소 조회
print(a[0, 1]) # 2
print(a[0][1]) # 2
# 행 조회
print(a[[0, 1]])
print(a[[0, 1], :])
'''
[[1 2 3]
 [4 5 6]]
'''
# 열 조회
print(a[:, [0, 1]])
'''
[[1 2]
 [4 5]
 [7 8]]
'''
# 행, 열 조회

# 두 번째 행 두 번째 열의 요소 조회
print(a[[1], [1]]) # [5]
# 세 번째 행 두 번째 열의 요소 조회
print(a[[2], [1]]) # [8]

# [0, 0], [1, 1], [2, 2] 조회 -> [행][열]로 작성
print(a[[0, 1, 2], [0, 1, 2]])
'''
[1 5 9]
'''
```

슬라이싱

- **배열[행1:행N,열1:열N]** 형태로 지정해 그 위치의 요소를 조회합니다.
- 조회 결과는 **2차원 배열**이 됩니다.
- 마지막 **범위 값은 대상에 포함되지 않습니다.**
- 즉, **배열[1:M, 2:N]**이라면 1 ~ M-1행, 2 ~ N-1열이 조회 대상이 됩니다.

```
# (3, 3) 형태의 2차원 배열 만들기
a = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
# 첫 번째 ~ 두 번째 행 조회
print(a[0:2])
print(a[0:2, :])
'''
[[1 2 3]
 [4 5 6]]
'''
# 첫 번째 ~ 두 번째 열을 조회
print(a[:, 0:2])
'''
[[1 2]
 [4 5]
 [7 8]]
'''
# 첫 번째 행, 첫 번째 ~ 두 번째 열 조회
print(a[0, 0:2])
'''
[1 2]
'''
# 결과를 2차원으로 받고 싶을 때
print(a[[0], 0:2])
print(a[0:1, 0:2])
'''
[[1 2]]
'''
# 첫 번째 ~ 세 번째 행, 두 번째 ~ 세 번째 열 조회
print(a[0:3, 1:3])
'''
[[2 3]
 [5 6]
 [8 9]]
'''
# 두 번째 ~ 끝 행, 두 번째 ~ 끝 열 조회
print(a[1:, 1:])
'''
[[5 6]
 [8 9]]
'''
# [0, 1] 위치부터 [1,2] 위치까지 요소 조회
print(a[0:2, 1:3])
print(a[0:2, 1:])
'''
[[2 3]
```

```
 [5 6]]
'''
```

조건 조회

- **조건에 맞는 요소를 선택**하는 방식이며, **불리안 방식(boolean)**이라고 부릅니다.
- 조회 결과는 **1차원 배열**이 됩니다.

```
# 2차원 배열 만들기
score= np.array([[78, 91, 84, 89, 93, 65],
                 [82, 87, 96, 79, 91, 73]])
# np.array에 조건문을 작성하면 결과를 boolean 값으로 반환
print(score >= 90)
'''
[[False,  True, False, False,  True, False],
 [False, False,  True, False,  True, False]])
'''
# 요소 중에서 90 이상인 것만 조회
print(score[score >= 90])
'''
[91 93 96 91]
'''
# 검색조건을 변수로 선언해서 사용 가능
condition = score >= 90
print(score[condition])
'''
[91 93 96 91]
'''
# 모든 요소 중에서 90 이상 95 미만인 것만 조회
print(score[(score >= 90) & (score <= 95)])
'''
[91 93 91]
'''
```

배열 연산

```
# 두 개의 (2, 2) 형태의 2차원 배열 만들기
x = np.array([[1, 2], [3, 4]])
y = np.array([[5, 6], [7, 8]])

# ndarray 생성
arr = np.array([[1, 2, 3], [4, 5, 6]])

# 벡터화 연산 (Element-wise operation)
print(x + 10)
'''
[[11, 12],
 [13, 14]]
'''
```

```
# 배열 더하기
print(x + y)
print(np.add(x, y))
'''
[[ 6  8]
 [10 12]]
'''
# 배열 빼기
print(x - y)
print(np.subtract(x, y))
'''
[[-4 -4]
 [-4 -4]]
'''
# 배열 곱하기
print(x * y)
print(np.multiply(x, y))
'''
[[ 5 12]
 [21 32]]
'''
# 배열 나누기
print(x / y)
print(np.divide(x, y))
'''
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
'''
# 배열 y 제곱
print(x ** y)
print(np.power(x, y))
'''
[[    1    64]
 [ 2187 65536]]
'''
# 행렬 곱
print(x @ y)
print(np.dot(x, y))
'''
[[19, 22],
 [43, 50]]
'''
```

Numpy Array에서 자주 사용되는 함수들

- `np.sum()`, 혹은 `array.sum()`
  - `axis = 0` : 열 기준 집계
  - `axis = 1` : 행 기준 집계
  - 생략하면 : 전체 집계
- 동일한 형태로 사용 가능한 함수 : `np.max()`, `np.min()`, `np.mean()`, `np.std()`

```
# array를 생성합니다.
a = np.array([[1,5,7],[2,3,8]])

# 전체 집계
print(np.sum(a)) # 26

# 열기준 집계
print(np.sum(a, axis = 0)) # [ 3  8 15]

# 행기준 집계
print(np.sum(a, axis = 1)) # [13 13]
```

- `np.argmax()`, `np.argmin()` : 인덱스 값으로 반환해줌

```
# 전체 중에서 가장 큰 값의 인덱스
print(np.argmax(a)) # 5

# 행 방향 최대값의 인덱스
print(np.argmax(a, axis = 0)) # [1 0 1]

# 열 방향 최대값의 인덱스
print(np.argmax(a, axis = 1)) # [2 2]
```

- `np.where(조건문, True 값, False 값)` : 조건문에 True면 2번째 인수로, False면 3번째 인수로 변환

```
# 선언
a = np.array([1,3,2,7])

# 조건 1
print(np.where(a > 2, 1, 0))   # [0 1 0 1]

# 조건 2
print(np.where(a > 3, -1, 5))  # [ 5  5  5 -1]

# 원래 값 유지하고 싶다면 배열 변수를 사용
print(np.where(a > 2, a, 0))   # [0 3 0 7]

# 마찬가지로 변수를 이용한 연산 값도 반환 가능
print(np.where(a > 2, a, a-1)) # [0 3 1 7]
```

★ `np.array` 가 지원하는 메소드는 `np.array` 데이터에 직접적으로 메소드 호출이 가능하다.

```
# 선언
a = [1,2,3]            # 리스트
b = (1,2,3)            # 튜플
c = np.array([1,2,3])  # 배열
```

```
# 평균 구하기 : 함수
print(np.mean(a))      # 2.0
print(np.mean(b))      # 2.0
print(np.mean(c))      # 2.0

# 평균 구하기 : 메서드 방식
print(a.mean()) # AttributeError: 'list' object has no attribute 'mean'
print(b.mean()) # AttributeError: 'tuple' object has no attribute 'mean'

print(c.mean()) # 2.0
```

Pandas DataFrame 데이터 조작

적용 가능한 상황

- **변수 선택**: 분석에 필요한 변수(열)만 선택하거나, 불필요한 변수를 제거할 때.
- **데이터 클리닝**: 특정 조건을 만족하는 이상치를 제거하거나, 잘못 입력된 값을 수정할 때.
- **파생 변수 생성**: 기존 변수들을 조합하여 새로운 의미를 갖는 변수(열)를 만들 때. (e.g., '키'와 '몸무게'로 'BMI'를 계산)
- **데이터 정규화**: 범주형 변수를 수치형으로 변환하거나(e.g., '남성'/'여성' -> 0/1), 수치형 변수의 단위를 변경할 때.
- **사용자 정의 함수 적용**: Pandas에서 기본으로 제공하지 않는 복잡한 로직을 데이터의 각 부분에 적용해야 할 때.

예제 데이터

- Kaggle의 타이타닉 데이터

```
import pandas as pd
data = pd.read_csv("Titanic-Dataset.csv")
```

DataFrame 데이터 조회

- 1차원(시리즈)로 조회하기

```
# Ticket 열 조회(시리즈)
data['Ticket']
# data.Ticket   ← 메소드인지 column 값인지 구분이 안가서 선호하지 않는 방식
```

- 2차원(데이터프레임)으로 조회

```
# Ticket 열 조회(데이터프레임)
data[['Ticket']] # column 이름을 '리스트'로 입력 한 것이다!
```

```
# 여러 열 한번에 조회 : Age, Cabin, Embarked열만 조회
data[['Age', 'Cabin', 'Embarked']]
```

조건부 조회

- 하나 이상의 조건을 만족하는 데이터 행을 불리언 인덱싱(Boolean Indexing)을 통해 추출
- 여러 조건을 결합할 때는 `&` (AND), `|` (OR) 연산자를 사용하며, 각 조건은 반드시 괄호 `()`로 묶어야 함 (e.g., `(df['A'] > 1) & (df['B'] < 10)`)

```
# 단일 조건 필터링
print(data[data['Survived'] == 0])

# 다중 조건 필터링
print(data[(data['Survived'] == 0) & (data['Sex'] == 'male')])

# isin을 사용한 필터링
print(data[data['Embarked'].isin(['C', 'Q'])])
```

- `.loc[행 조건, 열 이름]`
    - **df.loc[조건]** 형태로 조건을 지정해 조건에 만족하는 데이터만 조회 가능

1. 단일 조건 조회

```
# DistanceFromHome 열 값이 10 보다 큰 행 조회
data.loc[data[data['Survived'] == 0]
```

2. 여러 조건 조회

- []안에 조건을 여러개 연결할 때 `and`와 `or` 대신에 `&`와 `|`를 사용해야 함
- 각 조건들은 **(조건1) & (조건2)** 형태로 **괄호**로 묶어야 함 (`.loc[]` 안 쓸 때와 동일)

```
# and로 여러 조건 연결
data.loc[(data['Survived'] == 0) & (data['Sex'] == 'male')]
# or 조건 |
data.loc[(data['Survived'] == 0) | (data['Sex'] == 'male')]
```

필터링

- `isin([값1, 값2,…, 값n])` : 값1 또는 값2 또는…값n인 데이터만 조회, 매개변수는 무조건 리스트 형태

```
# 1이나 4인 값 나열
data.loc[data['Age'].isin([1,4])]
# 위 구문은 아래 or 조건으로 작성한 것과 동일하다
data.loc[(data['Age'] == 1) | (data['Age'] == 4)]
```

- `between(값1, 값2)` : 값1 ~ 값2까지 범위 안의 데이터만 조회
    - inclusive = True (기본값, 양쪽 끝 포함), False (양쪽 끝 제외)

```
# 범위 지정
data.loc[data['Age'].between(25, 30)]
# 위 구문은 아래 and 조건으로 작성한 것과 동일하다
data.loc[(data['Age'] >= 25) & (data['Age'] <= 30)]
```

- 조건을 만족하는 행의 일부 열만 조회 : **df.loc[조건, ['열 이름1', '열 이름2',...]]**

```
# 조건에 맞는 하나의 열 조회
data.loc[data['Fare'] >= 100, ['Age']]
# 조건에 맞는 여러 열 조회
data.loc[data['Fare'] >= 100, ['Age', 'Sex', 'Embarked']]

# 열 이름만 지정하고 모든 행을 가져오려면, 아래와 같이 작성
data.loc[ : , ['Age', 'Sex', 'Embarked']]
```

`.iloc`, `.at`

- `.loc`는 레이블(이름) 기반, `.iloc`는 정수 위치 기반 → 명확히 구분해서 사용 필요
    - 연속되지 않은 여러 행/열을 선택할 때는 리스트 `[]` 사용
- `.at`은 단일 값 선택 시 더 빠름

```
print(data.iloc[0:2]) # 0, 1번 위치의 행 (2는 포함 안 됨)
print(data.at[0, 'Age']) # print(data.loc[0, 'Age'])
```

DataFrame 데이터 집계 (`.groupby()`)

- dataframe.groupby('집계기준변수', as_index =)['집계대상변수'].집계함수
    - **집계기준변수:** ~별에 해당되는 변수 혹은 리스트. 범주형변수(예: 월 별, 지역 별 등)
    - **집계대상변수:** 집계함수로 집계할 변수 혹은 리스트. (예: 매출액 합계)
    - **as_index=True**를 설정(기본값)하면 집계 기준이 되는 열이 인덱스 열이 됩니다.

```
# 집계 결과가 data 열만 가지니 '시리즈'로 반환
# Pclass 별 Age 평균 --> 시리즈
data.groupby('Pclass', as_index=True)['Age'].mean()

# 대괄호로 한번 더 감싸면 열이 여럿이라서 '데이터프레임'으로 반환
# Pclass 별 Age 평균 --> 데이터프레임
data.groupby('Pclass', as_index=True)[['Age']].mean()

# 'as_index=False'를 설정하면, 행 번호를 기반으로 한 정수 값이 인덱스로 설정
```

```
# Pclass 별 Age 평균 --> 데이터프레임
data.groupby('Pclass', as_index=False)[['Age']].mean()
```

★ 집계 결과는 새로운 데이터프레임으로 선언하여 사용하는 경우가 많다!

- 여러 열 집계 : 집계 대상 열을 리스트로 지정하면 됨
  - sum() 앞에 아무 열도 지정하지 않으면 **기준열 이외의 모든 열에 대한 집계** 수행
    → 숫자형 변수만 집계되도록 명시적으로 지정할 필요가 있음

```
# 여러 열 집계 방법
data.groupby('Pclass', as_index=False)[['Age','Fare']].mean()
# 전부 sum하는 함수 형태
data.groupby('Pclass', as_index=False).sum()
```

- **by=['feature1', 'feature2']** 과 같이 집계 기준 열을 여럿 설정 가능

```
# 'Pclass', 'Sex'별 나머지 열들 평균 조회
data.groupby(['Pclass', 'Sex'], as_index=False)[['Age','Fare']].mean()
```

- df.groupby()**.agg(['함수1','함수2', ...])** : 여러 함수 한번에 사용 (as_index=False 작동 안함)

```
# min, max, mean 함수 3가지 한번에 적용
data.groupby('Pclass')[['Age']].agg(['min','max','mean'])

# 각 열마다 다른 집계를 한 번에 수행 가능
data.groupby('Pclass', as_index=False).agg({'Fare':'mean', 'Age':'min'})
```

DataFrame 변경

.astype

- 분석에 적합한 형태로 열의 데이터의 타입(Dtype)을 변환
- 변환하려는 열에 부적절한 값(e.g., 숫자로 바꿀 열에 문자 포함)이나 결측치(NaN)가 있으면 에러가 발생 가능
- **변환 가능한 타입**
  - 불 자료형 : bool
  - 정수 : int32, int64
  - 소수 : float32, float64
  - 범주형 : category
  - 문자 : object
  - 시간 : datetime64[ns], timedelta64[ns]

```

```

```
# Survived의 dtype을 category 타입으로 변경
# 변경 내용 적용을 원할 경우, 어디에 적용할지 선언
data['Survived'] = data['Survived'].astype('category')

# 서로 다른 dtype 지정을 원할 경우, dict 형식으로 입력
data = data.astype({'Survived':'category', 'Age':'object'})
```

- `pd.to_numeric`: 문자열이나 object 타입 Series를 숫자형(int, float) 으로 변환.
- `errors='coerce'` 옵션을 통해 변환 불가능한 값을 강제로 `NaN`으로 만들어주어, 데이터 클리닝 시 유용
- **파라미터**
  - `errors` : 변환 불가능한 값 처리 방법
    - 'raise' (기본값): 에러 발생
    - 'coerce': 변환 불가능한 값은 NaN 처리
    - 'ignore': 변환 실패 시 원본 그대로 반환
  - `downcast` : 'integer', 'signed', 'unsigned', 'float' → 가능한 경우 더 작은 dtype으로 변환

```
# Survived를 다시 수치형으로 변경
data['Survived'] = pd.to_numeric(data['Survived'])
# Age는 `errors='coerce'` 적용해서 변경
data['Age'] = pd.to_numeric(data['Age'], errors='coerce')
```

- `pd.to_datetime`: 날짜/시간 문자열을 datetime64[ns] 로 변환.
- **파라미터**
  - `errors`:
    - 'raise': 변환 실패 시 에러 발생 (기본값)
    - 'coerce': 변환 실패 값은 NaT 처리
    - 'ignore': 변환 실패 시 원본 반환
  - `dayfirst`: True일 경우 "01/02/2020" → 2월 1일로 해석 (기본은 False → 1월 2일)
  - `format`: datetime 형식 지정 (예: '%Y-%m-%d')
  - `utc`: True → UTC 시간으로 변환
  - `infer_datetime_format`: True → pandas가 포맷을 추측해서 성능 향상
- `pd.to_timedelta`: 문자열/숫자를 timedelta64[ns] 로 변환. (시간 차이를 나타냄)
- **파라미터**
  - `unit`: 'd', 'h', 'm', 's', 'ms', 'us', 'ns'
    - 숫자 입력 시 어떤 단위로 해석할지 지정
  - `errors`:
    - 'raise': 변환 실패 시 에러 발생 (기본값)
    - 'coerce': 변환 실패 시 NaT 처리
    - 'ignore': 변환 실패 시 원본 반환

column명 변경

- `.rename()` : 일부 열 이름 변경 하기

```
# columns 에 dictionary 데이터로 입력, inplace=True로 작성 시 데이터에 변경사항 바로 적용
data.rename(columns={'Sex' : 'Gender'}, inplace=True)
```

- `.columns` : 모든 열 이름을 변경할 때 사용할 수 있음

```
# 변경을 원치 않다면 기존 열 이름 그대로 작성
# 숫자가 서로 다를 경우, Length Mismatch 에러 발생
data.columns = ['번호','생존','등급','이름','성별','나이','형제/자매,배우자 수','부모/자녀 수', '티켓', '요금', '객실', '탑승항']
```

column 추가

- 열 추가 방법 : 없는 column 명으로 넣고자 하는 데이터를 작성하면 자동으로 추가됨

```
# 데이터 새로 로드(초기화) 후 진행
data['diff'] = data['SibSp']-data['Parch']
```

- `insert()` 메소드 사용 시 원하는 위치에 column 추가 가능
  - column의 위치를 크게 신경 안 쓰는게 좋음 (사용할 일이 없음)

```
# 첫번째 인수는 index 값(위치), 넣고자하는 column명, 넣고자하는 데이터 순으로 인수 작성
data.insert(1, 'diff', data['SibSp']-data['Parch'])
```

데이터 삭제

- `.drop()` : 데이터 삭제를 위한 메소드
  - axis = 0 : 행 삭제 (default)
  - axis = 1 : 열 삭제
  - inplace=False : 삭제한 데이터 반환만, 기존 데이터에 반영 안함 (default)
  - inplace=True : 기존 데이터에도 삭제한 내용을 반영함

★ 삭제는 잘못하면 되돌릴 수가 없는만큼, 데이터를 미리 copy 하고 진행하는 것이 좋다!

```
# data를 복사합니다.
data2 = data.copy()

# 열 하나 삭제
data2.drop('diff', axis=1, inplace=True)
```

```
# 열 두 개 삭제
data2.drop(['SibSp','Parch'], axis=1, inplace=True)
```

데이터 변경

★ 값 변경 또한 복구가 어려울 수 있는만큼, 데이터를 미리 copy 해두고 하는 것도 좋다!

```
# data를 복사
data2 = data.copy()

# diff의 값을 모두 0로 변경
data2['diff'] = 0

# 조건부 변경 : Fare 의 값이 8보다 작은 경우, 0로 변경
data2.loc[data2['Fare'] < 8, 'Fare'] = 0

# 조건부 변경 : Age가 30보다 많으면 1, 아니면 0으로 변경
data2['Age'] = np.where(data2['Age'] > 30, 1, 0)
```

- `.apply()`: DataFrame의 행 또는 열 전체에 함수 적용
- 일반적으로 벡터화된 연산(Numpy/Pandas 기본 연산)이 `apply`보다 훨씬 빠르므로, 가능한 한 벡터화 연산을 먼저 고려해야 함

```
# apply: 열(axis=0) 또는 행(axis=1) 단위로 함수 적용
def get_stats(x):
  return pd.Series([x.min(), x.max(), x.mean()], index=['min', 'max', 'mean'])

# Age, Fare 열에 대해 통계량 계산
print(data[['Age', 'Fare']].apply(get_stats))

# apply with lambda (행 단위 연산)
data['diff'] = data.apply(lambda row: row['SibSp'] - row['Parch'], axis=1)
print(data)
```

- `.map()`: 인자로 dictionary { } 값을 넣어서 key에 해당하는 데이터를 value 값으로 변경함
  - 주로 범주형 데이터를 다룰 때 사용

```
# male -> 1, female -> 0
data['Sex'] = data['Sex'].map({'male': 1, 'female': 0})
```

- `pd.cut()` : 숫자형 변수를 범주형 변수로 변환하는 함수
  - default는 범위를 균등하게 나누는 것이고, 값의 개수는 균등하지 않음
  - `labels` 파라미터를 이용하여 구간별 원하는 명칭 적용 가능
  - `bins` 파라미터를 이용하여 원하는 구간 나누기 가능

```
# Age 열을 3 등분으로 분할
age_group = pd.cut(data['Age'], 3)
age_group.value_counts()
'''
(26.947, 53.473]    345
(0.34, 26.947]      319
(53.473, 80.0]       50
'''

# Age 열을 3 등분으로 분할후 a,b,c로 이름 붙이기
age_group = pd.cut(data['Age'], 3, labels = ['a','b','c'])
age_group.value_counts()
'''
b    345
a    319
c     50
'''

# 원하는 구간대로 나누고, 이름 붙이기
# 'young'  : =< 30
# 'junior' : 30 <   =< 40
# 'senior' : 40 <
age_group = pd.cut(data['Age'], bins =[0, 30, 40, 100] , labels =
['young','junior','senior'])
age_group.value_counts()
'''
young     409
junior    155
senior    150
'''
```

One-hot encoding (가변수화)

- `pd.get_dummies`: 범주형 변수를 가변수화시켜준다.

```
import pandas as pd

df = pd.DataFrame({
    "Sex": ["male", "female", "female", "male"],
    "Embarked": ["C", "Q", "S", "S"]
})

# 범주형 변수 가변수화
df_dummies = pd.get_dummies(df, columns=["Sex", "Embarked"])
print(df_dummies)
'''
   Sex_female  Sex_male  Embarked_C  Embarked_Q  Embarked_S
0           0         1           1           0           0
1           1         0           0           1           0
2           1         0           0           0           1
```

```
3           0         1           0           0           1
'''
```

- 번외로 `sklearn.preprocessing.OneHotEncoder` 사용 시, `inverse_transform` 메서드를 통한 빠른 복원이 가능하다.
    - 출력된 결과물을 복원시켜야할 경우 사용 가능

```
from sklearn.preprocessing import OneHotEncoder
import numpy as np

X = np.array([["male"], ["female"], ["female"], ["male"]])
encoder = OneHotEncoder(sparse=False)
X_encoded = encoder.fit_transform(X)

print(X_encoded)
'''
[[0. 1.]
 [1. 0.]
 [1. 0.]
 [0. 1.]]
'''

X_restored = encoder.inverse_transform(X_encoded)
print(X_restored)
'''
[['male']
 ['female']
 ['female']
 ['male']]
'''
```