# 2026.01.27.화

| | |
|---|---|
| 생성일 | @2026년 1월 27일 오후 12:42 |
| 태그 | numpy |

## 목차



## ▼ 1) Numpy

```
import numpy as np

# 1차원 배열
arr1 = np.array([1, 2, 3, 4, 5])
print(arr1)  # 결과: [1 2 3 4 5]

# 2차원 배열 (행렬)
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2)
# 결과:
# [[1 2 3]
#  [4 5 6]]

# ============================================
import numpy as np

# 0으로 채워진 배열 생성
zeros = np.zeros((3, 4))  # 3행 4열의 0 행렬 생성
print(zeros)

# 1로 채워진 배열 생성
ones = np.ones((2, 3))   # 2행 3열의 1 행렬 생성
print(ones)

# ============================================
import numpy as np

# 특정 범위의 균일한 간격 배열
range_arr = np.arange(0, 10, 2)  # 0에서 10(미만)까지 2 간격으로
print(range_arr)  # [0 2 4 6 8]

# 선형 간격 배열
linear_space = np.linspace(0, 1, 5)  # 0과 1 사이를 5등분
print(linear_space)  # [0.  0.25 0.5  0.75 1.  ]
# 소수점 이하가 0이면 표시 생략됨

# 랜덤 배열
random_arr = np.random.random((2, 2))  # 2x2 균등분포 난수 행렬
print(random_arr)

# ============================================
import numpy as np

# 테스트를 위한 2차원 배열 생성
```

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

# 주요 속성 확인
print(f"배열 차원: {arr.ndim}")       # 결과: 2 (차원 수)
print(f"배열 형태: {arr.shape}")      # 결과: (2, 3) (행, 열)
print(f"배열 크기: {arr.size}")       # 결과: 6 (총 요소 개수)
print(f"요소 데이터 타입: {arr.dtype}") # 결과: int64
print(f"각 요소 바이트 크기: {arr.itemsize}") # 결과: 8 (64비트=8바이트)
print(f"전체 배열 바이트 크기: {arr.nbytes}")  # 결과: 48 (6개 * 8바이트)

# ============================================
import numpy as np

# 1. 전치 (행과 열 바꾸기)
# (앞선 실습에서 생성한 2행 3열의 arr 배열을 사용한다고 가정합니다)
print("원본 배열:")
print(arr)

print("전치 배열 (T):")
print(arr.T)  # 결과: [[1 4], [2 5], [3 6]]
# 계산 과정: (2, 3) 형태의 행렬이 (3, 2) 형태로 바뀜

# 2. 배열 형태 변경 (reshape)
arr1d = np.arange(12)  # [0 1 2 3 4 5 6 7 8 9 10 11] 생성
arr2d = arr1d.reshape(3, 4)  # 1차원 배열을 3행 4열의 2차원 배열로 변경

print("reshape 결과:")
print(arr2d)

# ============================================
# 배열 평탄화 (1차원으로 변환)
print("평탄화 결과 (flatten):")
print(arr2d.flatten())  # 결과: [0 1 2 3 4 5 6 7 8 9 10 11]

# 데이터 타입 변환
arr_float = arr.astype(np.float64)
print(f"타입 변환 후: {arr_float.dtype}")

# 통계 메서드
data = np.array([1, 2, 3, 4, 5])
print(f"합계: {data.sum()}")       # 결과: 15
print(f"평균: {data.mean()}")      # 결과: 3.0
print(f"최소값: {data.min()}")     # 결과: 1
print(f"최대값: {data.max()}")     # 결과: 5
print(f"표준편차: {data.std()}")    # 결과: 1.4142...
print(f"분산: {data.var()}")       # 결과: 2.0
print(f"누적합: {data.cumsum()}")   # 결과: [1 3 6 10 15]

# ============================================
import numpy as np

# 1. 조건 기반 인덱싱 (Boolean Indexing)
# 특정 조건을 만족하는 요소만 추출합니다.
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
print("짝수 요소만 선택:")
print(arr[arr % 2 == 0])  # 결과: [2 4 6 8]

# 2. 배열 연산 메서드 (행렬 곱셈)
# 두 행렬 간의 내적(Dot Product)을 계산합니다.
a = np.array([[1, 2], [3, 4]])
```

```
b = np.array([[5, 6], [7, 8]])

print("행렬 곱셈 (dot):")
print(a.dot(b)) # 결과: # [[19 22] #  [43 50]]

# ============================================
import numpy as np

# 형태가 다른 배열 연산
a = np.array([[1, 2, 3], [4, 5, 6]])  # 2x3 배열
b = np.array([[10], [20]])           # 2x1 배열

# 행렬 덧셈 수행
print(a + b)
# 결과:
# [[11 12 13]
#  [24 25 26]]

# 1. 배열과 스칼라 연산 (기본적인 브로드캐스팅)
arr = np.array([1, 2, 3, 4])
print(arr + 10)  # 결과: [11 12 13 14]

# 2. 다른 형태의 배열 간 브로드캐스팅
a = np.array([[1, 2, 3], [4, 5, 6]])  # 2x3 배열
b = np.array([10, 20, 30])           # 1x3 배열

print(a + b)
# 결과:
# [[11 22 33]
#  [14 25 36]]

# ============================================
import numpy as np

# (이전 단계의 arr = np.arange(12) 배열을 사용한다고 가정합니다)

# 1. 배열 형태 변경 (3x4)
reshaped2 = arr.reshape(3, 4)
print(reshaped2)
# 결과:
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]

# 2. 자동 계산 차원 (-1 사용)
# 전체 요소가 12개일 때, 행을 3으로 지정하고 열에 -1을 주면
# NumPy가 자동으로 12 / 3 = 4열로 계산하여 생성합니다.
reshaped3 = arr.reshape(3, -1)  # 3행, 열은 자동 계산
print(reshaped3)  # 결과: 3x4 배열

# ============================================
import numpy as np

# (앞선 실습의 a, b 배열을 사용한다고 가정합니다)
# a = np.array([[1, 2], [3, 4]])
# b = np.array([[5, 6], [7, 8]])

# 1. 가로 방향으로 합치기 (열 추가)
horizontal = np.concatenate([a, b], axis=1)
print(horizontal)
```

```
# 결과:
# [[1 2 5 6]
#  [3 4 7 8]]

# 2. vstack과 hstack 함수 사용
v_stack = np.vstack([a, b])  # vertical_stack (세로로 쌓기)
h_stack = np.hstack([a, b])  # horizontal_stack (가로로 쌓기)

print(v_stack)  # concatenate와 axis=0 결과가 동일함
print(h_stack)  # concatenate와 axis=1 결과가 동일함

# ============================================
import numpy as np

# 3행 4열의 2차원 배열 생성
arr = np.arange(12).reshape(3, 4)
print("원본 배열:")
print(arr)
# 결과:
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]

# 수평 분할 (행 기준)
# axis=0을 기준으로 3개의 배열로 균등하게 분할합니다.
h_split = np.split(arr, 3, axis=0)  # 3개 배열로 분할

for i, split_arr in enumerate(h_split):
    print(f"분할 {i}:", split_arr)

# 출력 결과:
# 분할 0: [[0 1 2 3]]
# 분할 1: [[4 5 6 7]]
# 분할 2: [[ 8  9 10 11]]

# ============================================
import numpy as np

# 1. 샘플 데이터: 고객 나이
ages = np.array([23, 18, 45, 61, 17, 34, 57, 28, 15, 42])

# 2. 성인(18세 이상) 필터링 (T/F 마스크 생성)
adult_filter = ages >= 18
print("필터링 결과(T/F):")
print(adult_filter)
# 결과: [ True  True  True  True False  True  True  True False  True]

# 3. 성인 데이터만 선택 (마스크를 인덱스로 사용)
adults = ages[adult_filter]
print("성인 데이터만 선택:")
print(adults)  # 결과: [23 18 45 61 34 57 28 42]

# 4. 복합 조건부 필터링 (한 줄로 작성)
# 18세 이상이면서 30세 미만인 '청년층' 추출
young_adults = ages[(ages >= 18) & (ages < 30)]
print("조건부 필터링 (18세 이상 30세 미만):")
print(young_adults)  # 결과: [23 18 28]
```

## ▼ 2) Pandas Series

- 구성 요소: Index + Values
  - 한 가지 타입의 데이터만 담을 수 있습니다 (예: 모두 숫자이거나 모두 문자열).
  - 딕셔너리처럼 인덱스 이름으로 데이터에 접근할 수 있습니다.
  - 통계 계산(평균, 합계 등)이 매우 빠릅니다.

```
import pandas as pd

# 리스트로 시리즈 만들기
prices = pd.Series([1500, 3000, 2500],
                                        index=['사과', '바나나', '포도'],
                                        name='과일가격')
print(prices)

# ================================================
import pandas as pd

# 기본 Series 생성
s = pd.Series([1, 3, 5, 7, 9])

print(s)
# 출력 결과:
# 0    1
# 1    3
# 2    5
# 3    7
# 4    9
# dtype: int64

# ================================================
import pandas as pd

# 인덱스 지정하여 Series 생성
s2 = pd.Series([10, 20, 30, 40], index=[`a`, `b`, `c`, `d`,])

print(s2)
# 출력 결과:
# 0    10
# 1    20
# 2    30
# 3    40
# dtype: int64

# ================================================
import pandas as pd

# 딕셔너리로 Series 생성
population = {
    'Seoul': 9776,
    'Busan': 3429,
    'Incheon': 2947,
    'Daegu': 2465
}

pop_series = pd.Series(population)

print(pop_series)
# 출력 결과:
# Seoul      9776
```

```
# Busan      3429
# Incheon    2947
# Daegu      2465
# dtype: int64

# ================================================
s = pd.Series([10, 20, 30, 40, 50],
                             index=['a', 'b', 'c', 'd', 'e'])

# 기본 속성
print("값 배열:", s.values) # [10, 20, 30, 40, 50]
print("인덱스:", s.index) # index=['a', 'b', 'c', 'd', 'e']

# 기본 통계 메소드
print("평균:", s.mean()) # 30.0
print("합계:", s.sum()) # 15e
print("최솟값:", s.min()) # 10
print("최댓값:", s.max()) 3 50

# 데이터 접근
print("'c' 인덱스의 값:", s['c']) # 30
print("'c' 인덱스의 값:", s['[a', 'c', 'e']]) # a 10, c 30, e 50

# ================================================
# 조건부 필터링
print("30보다 큰 값:", s[s > 30])

# 데이터 변환
print("제곱근:", s.apply(np.sqrt))

# apply : 함수를 적용하여 각 요소에 대한 연산을 수행
print("2배 값:", s * 2)

# ================================================
# 결측치 확인 및 처리
s2 = pd.Series([10, np.nan, 30, np.nan, 50],
                                index=['a', 'b', 'c', 'd', 'e'])
print("결측치 여부:", s2.isna())  # 결측치 여부를 불리언으로
print("결측치 제외:", s2.dropna()) # 결측치가 있는 항목 제거
print("결측치 0으로 채우기:", s2.fillna(0)) # 결측치를 0 으로 대체
```

## ▼ 3) Pandas Dataframe

- 구성 요소: Index (행) + Columns (열) + Values (값)
  - 각 열(Column)은 서로 다른 데이터 타입을 가질 수 있습니다 (이름은 문자열, 나이는 숫자 등).
  - 행과 열 방향 모두 인덱싱이 가능합니다.
  - 데이터 분석에서 가장 많이 사용되는 형태이며, 머신러닝 모델의 입력값으로 주로 쓰입니다.

```
import pandas as pd

data = {
    '이름': ['철수', '영희', '민수', '지영'],
    '점수': [90, 85, 95, 80],
    '지역': ['서울', '부산', '대구', '제주']
}
# 인덱스를 숫자가 아닌 'A', 'B', 'C', 'D'로 설정해 보겠습니다.
df = pd.DataFrame(data, index=['A', 'B', 'C', 'D'])

# 1. 특정 행 추출
```

```
print(df.loc['A'])          # 인덱스가 'A'인 행 (Series 반환)

# 2. 특정 행과 열 추출
print(df.loc['B', '점수'])  # 'B' 행의 '점수' 값 (85)

# 3. 슬라이싱 (중요: 끝점인 'C'를 포함합니다!)
print(df.loc['A':'C', '이름':'점수'])
# A, B, C 행의 이름과 점수 컬럼 출력

# 1. 첫 번째 행 추출
print(df.iloc[0])           # 0번 행 (철수 데이터)

# 2. 특정 위치 추출
print(df.iloc[1, 1])        # 1번 행, 1번 열 (85)

# 3. 슬라이싱 (중요: 끝점인 3을 포함하지 않습니다!)
print(df.iloc[0:3, 0:2])
# 0, 1, 2번 행의 0, 1번 열 출력 (A, B, C행 / 이름, 점수열)

# 기본 속성 확인
print("크기(행, 열):", df.shape)     # 결과: (5, 4)
print("열 이름:", df.columns)        # 결과: Index(['Name', 'Age', 'City', 'Salary'], dtype='object')
print("행 인덱스:", df.index)        # 결과: RangeIndex(start=0, stop=5, step=1)
print("데이터 타입:", df.dtypes)      # 각 열의 데이터 타입 출력

# 데이터 확인
print("처음 2행\n:", df.head(2))        # 처음 2행 출력
print("마지막 2행\n:", df.tail(2))       # 마지막 2행 출력
print("기본 통계량\n:", df.describe())     # 수치형 열의 통계 요약

# ================================================
# 데이터 접근
# 1. 단일 열 선택 (Series 형태로 반환)
print("'Age' 열:\n", df['Age'])

# 2. 여러 열 선택 (리스트를 사용해 DataFrame 형태로 반환)
print("여러 열 선택:\n", df[['Name', 'Salary']])

# 3. 위치 기반 인덱싱으로 특정 행 선택 (0번부터 2번 행까지)
print("첫 3행:\n", df.iloc[0:3])

# 4. 조건부 선택 (불리언 인덱싱)
# 'Age'가 30보다 큰 행들만 필터링
print("조건부 선택:\n", df[df['Age'] > 30])

# ================================================
# 데이터 수정
df['Age'] = df['Age'] + 1                      # 모든 나이에 1 추가
print("모든 나이에 1 추가:\n", df)

# 새 열 추가
df['Country'] = ['USA', 'France', 'Germany', 'UK', 'Japan']
print("새 열 추가:\n", df)

# 새 행 추가
df.loc[5] = ['Charlie', 29, 'Sydney', 70000, 'Australia']
print("새 행 추가:\n", df)

# 열 및 행 삭제
df.drop('Country', axis=1, inplace=True)       # 열 삭제
```

```
df.drop(5, axis=0, inplace=True)              # 행 삭제

print("열 삭제:\n", df)
print("행 삭제:\n", df)

# ================================================
import pandas as pd

# 1. 샘플 DataFrame 생성 - 직원 정보를 담은 테이블 형태의 데이터
df = pd.DataFrame({
    'Name': ['John', 'Anna', 'Peter', 'Linda', 'Bob'],          # 직원 이름 (문자열)
    'Age': [28, 24, 35, 32, 43],                               # 나이 (정수)
    'City': ['New York', 'Paris', 'Berlin', 'London', 'Tokyo'],     # 거주 도시 (문자열)
    'Salary': [50000, 65000, 75000, 85000, 60000],              # 급여 (정수)
    'Department': ['IT', 'HR', 'IT', 'Finance', 'Marketing']    # 부서 (문자열)
})

# 2. 열 선택 - 단일 열을 선택하면 Series 객체가 반환됨
print("단일 열 선택 (Series 반환):")
print(df['Name'])  # df['열이름'] 형식으로 특정 열 선택, pandas Series로 반환

# 여러 열 선택 - 리스트 형태로 여러 열 이름을 전달하면 DataFrame이 반환됨
print("\n여러 열 선택 (DataFrame 반환):")
print(df[['Name', 'Salary']])  # df[['열1', '열2', ...]] 형식, pandas DataFrame으로 반환

# 행 선택 (위치 기반) - iloc는 정수 인덱스를 사용한 위치 기반 선택
print("\n처음 3행 선택:")
print(df.iloc[0:3])  # iloc[시작:끝] - 0번째부터 2번째까지 (끝 인덱스 제외)
                     # iloc는 integer location의 줄임말

# 행 선택 (레이블 기반) - loc는 레이블을 사용한 선택 (기본 인덱스는 정수)
print("\n인덱스 1, 3, 4 행 선택:")
print(df.loc[[1, 3, 4]])  # loc[인덱스_리스트] - 특정 인덱스들을 리스트로 전달
                          # loc는 Label location의 줄임말

# 특정 행과 열 동시 선택 - loc를 사용하여 행과 열을 모두 지정
print("\n첫 2행의 'Name'과 'Age' 열:")

print(df.loc[0:1, ['Name', 'Age']])
# loc[행_범위, 열_리스트] 형식으로 접근
# 💡 참고: loc는 슬라이싱 시 끝 인덱스(1)를 포함하여 0, 1행을 모두 가져옵니다.

# 조건부 필터링 - 불리언 인덱싱을 사용하여 조건에 맞는 행만 선택
print("\n30세 이상인 직원:")

print(df[df['Age'] >= 30]) # df['Age'] >= 30은 각 행에 대해 True/False를 가진 Series를 생성합니다.
# 이 불리언 Series를 다시 df[]에 전달하면 True인 행만 필터링되어 출력됩니다.

# 다중 조건 필터링 - 여러 조건을 논리 연산자로 결합
print("\nIT 부서의 30세 이상 직원:")
print(df[(df['Age'] >= 30) & (df['Department'] == 'IT')])
# (조건1) & (조건2) - AND 연산자, 두 조건 모두 True인 행만 선택
# 주의: Python의 'and' 대신 '&' 사용, 각 조건을 괄호로 묶어야 함
# 다른 논리 연산자: | (OR), ~ (NOT)

# 값 존재 여부 필터링 - isin() 메소드로 특정 값들 중 하나와 일치하는지 확인
print("\n도쿄나 런던에 사는 직원:")
print(df[df['City'].isin(['Tokyo', 'London'])])
# isin([값1, 값2, ...]) - 리스트에 포함된 값 중 하나와 일치하는 행 선택
# SQL의 IN 연산자와 유사한 기능
```

```
# 반대 조건은 ~df['City'].isin(['Tokyo', 'London'])로 사용 가능

# =====================================================
import pandas as pd

# 1. 샘플 데이터프레임 생성
df = pd.DataFrame({
    'Department': ['IT', 'HR', 'IT', 'Finance', 'HR', 'IT'],
    'Employee': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Frank'],
    'Salary': [75000, 65000, 80000, 90000, 60000, 78000],
    'Age': [28, 35, 32, 45, 30, 29],
    'Year': [2021, 2022, 2021, 2022, 2021, 2022]
})

# 2. 기본 groupby 연산
# 'Department' 열을 기준으로 데이터를 그룹화합니다.
dept_groups = df.groupby('Department')

print("부서별 평균 급여:")
# 그룹화된 객체에서 'Salary' 열만 선택하여 평균(mean)을 구합니다.
print(dept_groups['Salary'].mean())

# 기본 groupby 연산
# groupby()는 데이터를 특정 기준으로 그룹화하여 각 그룹별로 집계 연산을 수행할 수 있게 해줍니다.
# SQL의 GROUP BY와 동일한 개념입니다.

# 'Department' 열을 기준으로 그룹화
dept_groups = df.groupby('Department')

print("부서별 평균 급여:")
# 각 부서별로 급여의 평균을 계산
print(dept_groups['Salary'].mean())

# 결과 예시:
# Finance    90000.0
# HR         62500.0
# IT         77666.666667

# 다중 열 기준 그룹화
# 여러 열을 기준으로 그룹화할 때는 리스트 형태로 전달합니다.
# 이는 계층적 그룹화(hierarchical grouping)를 만들어냅니다.

# 'Year'와 'Department' 두 개의 열을 기준으로 그룹화
year_dept_groups = df.groupby(['Year', 'Department'])  # 연도와 부서 두 기준으로 그룹화

print("\n연도별, 부서별 평균 급여:")
# 그룹화된 데이터에서 'Salary'의 평균 계산
print(year_dept_groups['Salary'].mean())

# 결과는 MultiIndex로 표시됩니다: (2021, HR), (2021, IT), (2022, Finance) 등

# describe()는 각 그룹별로 기본 통계량(개수, 평균, 표준편차, 최소값, 25%, 50%, 75%, 최대값)을 제공합니다.
print("\n부서별 급여 통계 요약:")

# 그룹화된 객체(dept_groups)의 특정 열('Salary')에 대해 종합 통계량 산출
print(dept_groups['Salary'].describe())

# 결과로는 각 부서별로 count, mean, std, min, 25%, 50%, 75%, max 값이 표 형태로 표시됩니다.

# agg 메소드로 다양한 집계 함수 적용
```

```
# agg()는 aggregate의 줄임말로, 하나 이상의 집계 함수를 동시에 적용할 수 있습니다.

print("\n여러 집계 함수 적용:")
# 그룹화된 객체에서 'Salary' 열에 대해 개수, 평균, 합계, 표준편차, 최솟값, 최댓값을 한 번에 계산
print(dept_groups['Salary'].agg(['count', 'mean', 'sum', 'std', 'min', 'max']))

# agg와 describe 차이점:
# - agg: 사용자가 원하는 집계 함수들을 직접 선택하여 적용
# - describe: 미리 정의된 통계 요약 정보를 제공 (count, mean, std, min, 25%, 50%, 75%, max)

# 열마다 다른 집계 함수 적용
# 딕셔너리 형태로 각 열에 대해 서로 다른 집계 함수들을 적용할 수 있습니다.

print("\n열별 다른 집계 함수:")
print(dept_groups.agg({
    'Salary': ['mean', 'max'],          # 급여는 평균과 최대값만 산출
    'Age': ['mean', 'min', 'max']      # 나이는 평균, 최소값, 최대값 산출
}))

# 결과는 MultiIndex 컬럼으로 표시됩니다: (Salary, mean), (Salary, max), (Age, mean) 등

# transform 메소드 - 그룹 통계를 원본 데이터 크기로 변환
# transform()은 그룹별 집계 결과를 원본 DataFrame과 같은 크기로 확장하여 반환합니다.
# 이를 통해 각 행에 해당 그룹의 통계값을 추가할 수 있습니다.

# 'Salary'의 부서별 평균을 구해 원본 데이터와 같은 형태로 반환하고 새 열에 저장
df['Dept_Avg_Salary'] = dept_groups['Salary'].transform('mean')

print("\n각 직원의 급여와 부서 평균 급여:")
print(df[['Employee', 'Department', 'Salary', 'Dept_Avg_Salary']])

# 예: Alice(IT)의 경우 개인 급여는 75000이지만, IT 부서 평균 급여 77666.67이 추가됩니다.

# filter 메소드 - 그룹 조건에 따라 필터링
# filter()는 그룹 전체가 특정 조건을 만족하는지 확인하여,
# 조건을 만족하는 그룹의 모든 행을 반환합니다.
# Lambda 함수를 사용하여 각 그룹(x)에 대한 조건을 정의합니다.

# 평균 급여가 70,000원 초과인 '부서'의 모든 데이터를 추출
high_salary_depts = dept_groups.filter(lambda x: x['Salary'].mean() > 70000)

print("\n평균 급여가 70000 이상인 부서의 모든 직원:")
print(high_salary_depts)

# 결과 분석:
# IT 부서(평균 77666.67)와 Finance 부서(평균 90000)는 조건을 만족하므로
# 해당 부서 소속 직원 전원의 데이터가 반환됩니다.
# 반면 HR 부서(평균 62500)는 조건에 미달하여 결과에서 제외됩니다.

# get_group 메소드 - 특정 그룹 선택
# get_group()을 사용하여 특정 그룹의 데이터만 추출할 수 있습니다.

print("\nIT 부서 직원:")
# groupby 객체(dept_groups)에서 'IT' 그룹에 속한 행들만 가져옵니다.
print(dept_groups.get_group('IT'))

# 결과: IT 부서에 속한 모든 직원(Alice, Charlie, Frank)의 정보가 반환됩니다.
```