[5주차] 강력한 데이터 계산: NumPy 첫걸음 🔢

안녕하세요! 5주차 수업에 오신 것을 환영합니다.

지난 시간까지 우리는 파이썬의 기본 문법으로 데이터를 다루는 법을 배웠습니다.

하지만 만약 우리가 다룰 데이터가 수백만 개라면 어떻까요?

`for` 반복문만으로는 너무 느리고 비효율적일 겁니다.

오늘은 이 문제를 해결해 줄 파이썬님의 필수 라이브러리, **NumPy**를 소개합니다.

NumPy를 이용해 대규모 숫자 데이터를 얼마나 빠르고 효율적으로 처리할 수 있는지,

그리고 이것이 왜 파이썬님에서 중요한지 직접 확인해 보겠습니다.

🐢 모듈 1: 왜 파이썬 리스트로는 부족할까?

- **상황**
  - 두 학생 그룹의 시험 점수를 각각 5점씩 올려주려 하는 상황
- **파이썬 리스트의 문제점**
  - 리스트에 담긴 모든 숫자에 한 번씩 5를 더하는 간단한 연산이 필가능
  - `[88, 90, 78]` + 5 → **Error!**
    - 리스트와 숫자는 더할 수 없음
- **해결책**
  - `for` 반복문을 사용해서 각 숫자를 하나씩 꺼내 5를 더하고,
  - 새로운 리스트에 다시 담아야 합니다.

- 데이터가 많아질수록 매우 번거롭고 느려집니다!

```
In [2]: # 파이썬 리스트로 점수 관리
scores_list = [80, 95, 72, 100, 88]

# 모든 점수를 5점씩 올리기 위한 과정
new_scores_list = [] # 새로운 점수를 담을 빈 리스트
for score in scores_list:
    # 하나씩 꺼내서 5를 더하고 새 리스트에 추가
    new_scores_list.append(score + 5)

print(f"원본 점수 리스트: {scores_list}")
print(f"5점 올린 후 리스트: {new_scores_list}")
print("\n데이터가 5개만 있어도 이렇게 번거로운데, 100만 개라면...?")
```

```
원본 점수 리스트: [80, 95, 72, 100, 88]
5점 올린 후 리스트: [85, 100, 77, 105, 93]

데이터가 5개만 있어도 이렇게 번거로운데, 100만 개라면...?
```

✨ 모듈 2: NumPy 배열(Array)의 등장

- **NumPy란?**
  - Numerical Python의 줄임말
  - 대규모 숫자 데이터를 빠르고 효율적으로 다룰 수 있게 해주는 라이브러리
- **NumPy 배열 (ndarray)**
  - NumPy에서 사용하는 다차원 배열(리스트)
  - 제약: **같은 종류의 데이터만 저장**
    - 숫자만 있거, 문자만 문자
    - 이 제약이 있는 대신, 속도가 매우 빨릅니다!
- **벡터화 연산 (Vectorized Operation)**
  - 배열의 모든 요소에 대해 반복문 없이 한 번에 연산을 적용할 수 있습니다.
  - 가장 중요한 특징!

```
In [3]: import numpy as np # 관습적으로 np라는 별명으로 불러옵니다.

# 파이썬 리스트를 NumPy 배열로 변환
scores_array = np.array([80, 95, 72, 100, 88])

print(f"NumPy 배열: {scores_array}")
print(f"자료형: {type(scores_array)}")
```

```
NumPy 배열: [ 80  95  72 100  88]
자료형: <class 'numpy.ndarray'>
```

```
In [4]: # NumPy 배열을 이용해 모든 점수 5점 올리기
# 반복문이 전혀 필요 없습니다!

new_scores_array = scores_array + 5

print(f"원본 점수 배열: {scores_array}")
print(f"5점 올린 후 배열: {new_scores_array}")

# 모든 점수에 10% 보너스 연산도 가능합니다.
print(f"10% 보너스 적용: {scores_array * 1.1}")
print(f"모든 점수 2로 나누기: {scores_array / 2}")
```

```
원본 점수 배열: [ 80  95  72 100  88]
5점 올린 후 배열: [ 85 100  77 105  93]

모든 점수 10% 인상: [ 88.  104.5  79.2 110.   96.8]
모든 점수 2로 나누기: [40.  47.5 36.  50.  44. ]
```

🚀 모듈 3: 머신러닝을 위한 NumPy 핵심 기능

1. 배열의 형태 (Shape)

- **Shape이란?**
  - 배열이 몇 개의 행과 열로 이루어져 있는지 알려주는 정보입니다.
  - 머신러닝에서 데이터의 구조를 파악하는 데 가장 기본적이고 중요합니다.
- **1차원 배열:** 벡터(Vector)
  - ex) 학생 5명의 점수
- **2차원 배열:** 행렬(Matrix)
  - ex) 5명 학생의 국어, 영어 2과목 점수
- **3차원 배열:** 텐서(Tensor)
  - ex) (가로, 세로, 컬러) 픽셀로 이루어진 컬러 이미지

```
In [5]: # 1차원 배열 (벡터)
# [국어1, 국어2, 국어3, 국어4, 국어5, 국어6] -> 한 줄로 6개
array_1d = np.array([1, 2, 3, 4, 5, 6])
print(f"1D 배열: {array_1d}")
print(f"1D 배열의 Shape: {array_1d.shape}") # (6,) -> 아이템이 6개인 한 줄

print("-" * 20)

# 2차원 배열 (행렬)
# [[국어1, 영어1],  -> 첫 번째 줄 (행)
#  [국어2, 영어2]]  -> 두 번째 줄 (행)
#  ↑국어1, 영어1] -> 세 번째 줄 (행)
array_2d = np.array([
    [90, 85], # 학생 1
    [75, 92], # 학생 2
    [88, 80]  # 학생 3
])
print(f"2D 배열:\n{array_2d}")
print(f"2D 배열의 Shape: {array_2d.shape}")
# (3, 2) -> 3개의 행, 2개의 열
```

```
1D 배열: [1 2 3 4 5 6]
1D 배열의 Shape: (6,)
--------------------
2D 배열:
[[90 85]
 [75 92]
 [88 80]]
2D 배열의 Shape: (3, 2)
```

2. 배열의 형태 바꾸기 (Reshape)

- `reshape()`
  - 데이터의 내용과 총 개수는 그대로 둔 채,
  - 배열의 형태(깊데기)만 싸물 때 사용합니다.
- **-1의 활용**
  - **"나머지는 네가 알아서 계산해줘"** 라는 의미의 와일드카드입니다.
  - `array.reshape(2, -1)`
    - → "총 6개인데, 2개의 행으로 만들고 싶어. 열은 알아서 계산해줘."
    - → (2, 3)으로 변환
  - `array.reshape(-1, 3)`
    - → "총 6개인데, 3개의 열로 만들고 싶어. 행은 알아서 계산해줘."
    - → (2, 3)으로 변환
  - **주의:** 전체 원소 개수가 나누어 떨어지지 않으면 에러가 발생합니다.
    - ex) 6개 원소를 4개의 행으로 만들 수 없음

```
In [6]: # 1차원 배열 (6개짜 소방)
array_1d = np.array([1, 2, 3, 4, 5, 6])
print(f"원본 1D 배열: {array_1d}")
print(f"원본 1D 배열 Shape: {array_1d.shape}")

# 2행 3열의 2차원 배열로 형태 변경
# [1,2,3,4,5,6] -> [[1,2,3],
#                   [4,5,6]]
array_reshaped = array_1d.reshape(2, 3)
print(f"\n(2, 3)으로 Reshape:\n{array_reshaped}")
print(f"Shape: {array_reshaped.shape}")

# -1을 사용하여 열의 개수를 자동으로 계산
# "6개 원소를 3개씩 묶으로 만들어줘. 몇로 묶어야?" -> 2개씩 딸 필요
array_reshaped_auto = array_1d.reshape(3, -1)
print(f"\n(3, -1)으로 Reshape:\n{array_reshaped_auto}")
print(f"Shape: {array_reshaped_auto.shape}")
```

```
원본 1D 배열: [1 2 3 4 5 6]
원본 1D 배열 Shape: (6,)

(2, 3)으로 Reshape:
[[1 2 3]
 [4 5 6]]
Shape: (2, 3)

(3, -1)으로 Reshape:
[[1 2]
 [3 4]
 [5 6]]
Shape: (3, 2)
```

3. 축(Axis)을 따른 연산

- **축(Axis)이란?**
  - 다차원 배열에서 **연산을 수행할 방향**을 의미하는 것입니다.
- **머신속으로 그리기! (2차원 기준)**
  - **`axis=0` : 세로 방향(↓)**
    - 각 **열(column)**에 대해 연산합니다.
    - 여러 행들을 하나의 행으로 꾹 눌러 압축하는 모습을 상상하세요.
  - **`axis=1` : 가로 방향(→)**
    - 각 **행(row)**에 대해 연산합니다.
    - 여러 열들을 하나의 열로 꾹 눌러 압축하는 모습을 상상하세요.
- `axis`를 지정하지 않으면, 배열의 모든 원소에 대해 연산합니다.

```
In [8]: # 3명 학생의 국어, 영어 점수
scores_2d = np.array([
    [90, 85], # 학생 1
    [75, 92], # 학생 2
    [88, 80]  # 학생 3
])
print(f"원본 점수 배열 (Shape: {scores_2d.shape}):\n{scores_2d}")

# 1. 전체 합계 / 평균
print(f"\n전체 총점: {scores_2d.sum()}")
print(f"전체 평균: {scores_2d.mean():.2f}")

# 2. axis=0 (세로 방향 ↓) 연산
# 각 '분'의 합계 or 평균 -> 과목별 통계
print(f"\n--- axis=0 (과목별 통계) ---")

print(f"과목별 총점: {scores_2d.sum(axis=0)}")
# [90+75+88, 85+92+80]

print(f"과목별 평균: {scores_2d.mean(axis=0)}")
# [avg(90,75,88), avg(85,92,80)]

print(f"과목별 최고점: {scores_2d.max(axis=0)}")
# [max(90,75,88), max(85,92,80)]

# 3. axis=1 (가로 방향 →) 연산
# 각 '분'의 합계 or 평균 -> 학생별 통계
print(f"\n--- axis=1 (학생별 통계) ---")

print(f"학생별 총점: {scores_2d.sum(axis=1)}")
# [90+85, 75+92, 88+80]

print(f"학생별 평균: {scores_2d.mean(axis=1)}")
# [avg(90,85), avg(75,92), avg(88,80)]

print(f"학생별 최고점: {scores_2d.max(axis=1)}")
# [max(90,85), max(75,92), max(88,80)]
```

```
원본 점수 배열 (Shape: (3, 2)):
[[90 85]
 [75 92]
 [88 80]]

전체 총점: 510
전체 평균: 85.00

--- axis=0 (과목별 통계) ---
과목별 총점: [253 257]
과목별 평균: [84.33333333 85.66666667]
과목별 최고점: [90 92]

--- axis=1 (학생별 통계) ---
학생별 총점: [175 167 168]
학생별 평균: [87.5 83.5 84. ]
학생별 최고점: [90 92 88]
```

4. 유용한 배열 생성 함수

- `np.arange(n)`: 0부터 n-1까지의 숫자를 가진 배열 생성
- `np.zeros(shape)`: 모든 값이 0인 배열 생성 (초기화에 유용)
- `np.ones(shape)`: 모든 값이 1인 배열 생성

```
In [9]: # 0부터 9까지의 숫자를 가진 배열
range_array = np.arange(10)
print(f"np.arange(10):\n{range_array}")

# (2, 3) 형태로 모든 값이 0인 배열
zeros_array = np.zeros((2, 3))
print(f"\nnp.zeros((2, 3)):\n{zeros_array}")

# (3, 2) 형태로 모든 값이 1인 배열
ones_array = np.ones((3, 2))
print(f"\nnp.ones((3, 2)):\n{ones_array}")
```

```
np.arange(10):
[0 1 2 3 4 5 6 7 8 9]

np.zeros((2, 3)):
[[0. 0. 0.]
 [0. 0. 0.]]

np.ones((3, 2)):
[[1. 1.]
 [1. 1.]
 [1. 1.]]
```