<[파이썬 데이터 마법사] 제3장: 넘파이 배열 탐험의 끝판왕! 슬라이싱, 조건부 마법, 그리고 배열 생성의 달인!>

선생님: (다양한 크기와 모양의 격자판을 보여주며) 얘들아, 안녕! 지난 시간에 배운 넘파이 배열의 `.shape`과 `.reshape()` 기억나니? 오늘은 이 배열들을 더욱 섬세하게 다루는 슬라이싱 기술과, 조건에 따라 배열의 내용을 바꾸는 마법, 그리고 특별한 배열을 만들어내는 비법들을 배워볼 거야!

<1. 넘파이 배열 조각내기: 2차원 슬라이싱의 달인!>

선생님: 1차원 배열 슬라이싱은 리스트와 비슷했지? 2차원 배열은 마치 여러 층으로 된 케이크 같아서, 가로로도 자르고 세로로도 잘라낼 수 있단다!

```
import numpy as np

# 복습: 1차원 배열 슬라이싱
a_1d = np.array([1, 2, 3, 4, 5])
print("1D 슬라이싱 a_1d[0:3] ->", a_1d[0:3]) # [1 2 3]

# 5x5 모양의 2차원 배열 만들기 (1부터 25까지)
b_range = range(1, 26)
b_np = np.array(b_range)
c_2d = b_np.reshape(5, 5)
print("\n--- 5x5 배열 c_2d ---")
print(c_2d)

# 2차원 배열 슬라이싱 마법!
# 모든 행(:)의, 2번 열(세 번째 열)만 가져와! (케이크를 세로로 싹둑!)
print("\n모든 행의 2번 열 (c_2d[:, 2]):")
print(c_2d[:, 2]) # [ 3  8 13 18 23]

# 마지막 행(-1)의, 2번 열부터 끝까지 가져와!
print("\n마지막 행의 2번 열부터 끝까지 (c_2d[-1, 2:]):")
print(c_2d[-1, 2:]) # [23 24 25]
```

코딩새싹 🌱: 와! `c_2d[:, 2]` 이렇게 하니까 정말 세로로 한 줄만 쏙 빠져나왔어요! 쉼표(,) 앞이 행, 뒤가 열을 선택하는 거군요!

궁금이 💡: 그럼 `c_2d[1:3, 0:2]` 이렇게 하면 1행부터 2행까지, 그리고 0열부터 1열까지의 작은 네모 조각이 나오겠네요?

선생님: 정확해! 그렇게 행과 열의 범위를 지정해서 원하는 부분만 자유자재로 잘라낼 수 있단다.

<2. 배열에 그림 그리기: 반복문으로 값 채우기!>

선생님: 때로는 빈 배열을 만들어두고, 우리가 원하는 규칙에 따라 값을 채워 넣고 싶을 때가 있어. 마치 텅 빈

모눈종이에 점을 찍어 그림을 완성하는 것처럼 말이지!

```
# 5x5 크기의 모든 값이 0인 배열을 먼저 만들고,
a_zeros = np.zeros((5, 5))
print("\n--- 처음엔 0으로 가득 찬 5x5 배열 ---")
print(a_zeros)

# 반복문으로 특별한 규칙에 따라 값을 채워 넣어볼까?
# 바깥 루프는 행(idx1), 안쪽 루프는 열(idx2)을 담당
for idx1, val1 in enumerate(range(0, 25, 5)): # val1은 0, 5, 10, 15, 20
  for idx2, val2 in enumerate(range(1, 10, 2)): # val2는 1, 3, 5, 7, 9
    # a_zeros[idx1, idx2] 위치에 val1 + val2 값을 넣어줘!
    if idx1 < 5 and idx2 < 5 : # 배열 크기를 넘지 않도록!
        a_zeros[idx1, idx2] = val1 + val2

print("\n--- 값을 채워 넣은 후의 5x5 배열 ---")
print(a_zeros)
```

코딩새싹 🌱: enumerate랑 range를 같이 쓰니까 행 번호랑 열 번호, 그리고 계산에 쓸 값까지 한 번에 만들 수 있네요! 복잡한 패턴도 만들 수 있겠어요!

<3. 조건부 마법! np.where()의 변신!>

선생님: np.where(조건)을 쓰면 조건에 맞는 값들의 '위치'를 알려줬었지? 이 np.where()에는 더 강력한 변신 마법이 숨어있단다! 바로 **조건에 따라 다른 값을 채워 넣는 마법**이야!

```
np.where(조건, 조건이_참일_때_값, 조건이_거짓일_때_값)
```

```
b_simple = np.array([3, 4, 1, 7, 2])
print("\n--- np.where()의 조건부 값 할당 ---")
print("원래 배열 b_simple:", b_simple)

# b_simple의 각 원소가 3보다 크거나 같으면 1을, 아니면 0을 넣어 새로운 배열 d를 만들어줘!
d_conditional = np.where(b_simple >= 3, 1, 0)
print("조건에 따라 1 또는 0으로 채운 배열 d_conditional:", d_conditional)
```

궁금이 💡: 와! if-else 문을 배열 전체에 한 번에 적용하는 것 같아요! "3 이상이면 1등급, 아니면 0등급!" 이렇게 점수를 매기는 것 같기도 하고요!

<4. 정상에서 만나요! 최댓값과 그 위치 찾기!>

선생님: 넓은 산(2차원 배열)에서 가장 높은 봉우리(최댓값)를 찾고, 그 봉우리가 어디쯤(인덱스)에 있는지 알아내는 것도 넘파이에겐 식은 죽 먹기지!

```
x_2d = np.array(range(1, 26)).reshape(5, 5) # 다시 5x5 배열!
print("\n--- 최댓값과 그 위치 찾기 ---")
```

```
print("탐험할 2D 배열 x_2d:\n", x_2d)

max_x = np.max(x_2d) # 가장 높은 봉우리 값 찾기!
print(f"가장 높은 봉우리의 높이는? {max_x}")

# 그 봉우리는 어디에 있을까? np.where() 탐정 출동!
idx_max_x = np.where(x_2d == max_x)
print("최댓값의 위치 정보:", idx_max_x)
# 결과: (array([4]), array([4])) -> (행 인덱스들이 담긴 배열, 열 인덱스들이 담긴 배열)

# 첫 번째 찾은 최댓값의 위치 (0부터 세니까 4행, 4열이 맞지!)
row_idx = idx_max_x[0][0]
col_idx = idx_max_x[1][0]
print(f"최댓값 {max_x}의 정확한 위치는 ({row_idx}행, {col_idx}열) 입니다.")
```

코딩새싹 🌱: np.where()가 2차원 배열에 쓰이니까 결과가 튜플 안에 배열 두 개로 나오네요! 첫 번째 배열은 행 번호들, 두 번째 배열은 열 번호들이군요! 신기하다!

<5. 합계의 달인: sum() vs np.sum() 그리고 np.arange()>

선생님: 리스트나 배열의 합계를 구할 때 파이썬 기본 sum() 함수도 쓸 수 있지만, 넘파이에게는 np.sum()이라는 전용 합계 마법도 있단다!

```
a_list = [2, 3, 8, 1]
print("\n--- 합계 구하기 ---")
print("파이썬 sum()으로 리스트 합계:", sum(a_list))

b_np_sum = np.array(a_list)
print("파이썬 sum()으로 넘파이 배열 합계:", sum(b_np_sum))
print("넘파이 np.sum()으로 넘파이 배열 합계:", np.sum(b_np_sum))
```

궁금이 💡: 둘 다 결과는 똑같은데, 왜 np.sum()이 따로 있는 거예요?

선생님: 좋은 질문이야! 간단한 1차원 배열은 결과가 같지만, np.sum()은 넘파이 배열에 최적화되어 있어서 아주 큰 배열일 경우 더 빠를 수 있고, 나중에 배우겠지만 다차원 배열에서 특정 행이나 열만의 합계를 구하는 등 더 강력한 기능을 가지고 있단다!

선생님: 그리고 우리가 range()로 숫자 순서를 만들었듯이, 넘파이에는 **np.arange()** 라는 친구가 있어서 처음부터 넘파이 배열 형태로 숫자 순서를 만들어준단다!

```
# 0부터 9까지 넘파이 배열 만들기
print("np.arange(10) ->", np.arange(10))
# 10부터 19까지 넘파이 배열 만들고, 거기에 10씩 더하기! (벡터 연산!)
print("np.arange(10, 20) + 10 ->", np.arange(10, 20) + 10)
```

<6. 한 줄 마법, 람다 함수 복습!>

선생님: (작은 삼각형 모양 종이를 보여주며) 삼각형 넓이 구하는 공식 기억나니? 밑변 * 높이 / 2 였지? 이걸 간단한 함수로 만들 때, def 대신 lambda를 쓸 수도 있었지!

```
# def tri_area(a, h):
#   return a * h / 2

triangle_area_lambda = lambda base, height: base * height / 2
print("\n람다 함수로 구한 밑변 2, 높이 4인 삼각형의 넓이:", triangle_area_lambda(2, 4))
```

선생님: (마법사 모자를 벗으며) 자, 오늘 우리는 넘파이 배열을 자유자재로 자르고, 모양을 바꾸고, 조건에 따라 값을 채우고, 특별한 값과 그 위치를 찾아내고, 심지어 넘파이 스타일로 숫자 순서까지 만들어내는 등 정말 많은 고급 마법들을 배웠어! 이제 여러분은 넘파이라는 강력한 데이터 마법 지팡이를 더욱 능숙하게 다룰 수 있게 되었을 거야! 정말 대단한 발전이야! 🥳