Python 기초 Chap5: 벡터와 친해지기

Table of contents

5.1 넘파이(NumPy) 벡터 슬라이싱

벡터의 일부를 추출할 때는 대괄호([])를 사용합니다. 대괄호 안에는 추출하려는 요소의 위치나 인덱스를 지정합니다.

```
import numpy as np

# 벡터 슬라이싱 예제, a를 랜덤하게 채움
np.random.seed(42)
a = np.random.randint(1, 21, 10)
print(a)

# 두 번째 값 추출
print(a[1])
```

```
[ 7 20 15 11  8  7 19 11 11  4]
20
```

5.1.1 파이썬 인덱싱 특징

파이썬 인덱싱의 특징은 다음과 같습니다:

- 인덱스는 0부터 시작
- 양의 인덱스는 앞에서부터 세고, 음의 인덱스는 뒤에서부터 셉니다.
- 슬라이싱 구문 `[start:stop:step]`에서 `stop`은 포함되지 않습니다.

예를 들어 `a = [0, 1, 2, 3, 4, 5]`라는 리스트가 있다면:

- `a[0] = 0` (첫 번째 값)
- `a[5] = 5` (마지막 값)
- `a[-1] = 5` (마지막 값, 음수 인덱스 사용)
- `a[1:4] = [1, 2, 3]` (인덱스 1부터 3까지 추출, 4는 미포함)
- `a[::2] = [0, 2, 4]` (처음부터 끝까지, 스텝은 2)

이런 식으로 인덱싱과 슬라이싱을 활용하여 원하는 값들을 추출할 수 있습니다.

```
# 세 번째부터 다섯 번째 값 추출
print(a[2:5])
```

```
[15 11  8]
```

- 주의 할 점: 시작값에 해당하는 포지션은 포함, 마지막 값에 해당하는 포지션은 미포함.

5.1.1.a 파이썬 슬라이싱 연산자

`a[2:5]`에서 `2:5`는 슬라이싱 연산자입니다. 이 연산자는 `start:stop:step` 형태를 가지며, 다음과 같이 동작합니다.

- `start`: 슬라이싱을 시작할 인덱스 위치입니다. 이 인덱스는 포함됩니다.
- `stop`: 슬라이싱을 종료할 인덱스 위치입니다. 이 인덱스는 제외됩니다.
- `step`: 인덱스를 움직일 간격입니다. 기본값은 1입니다.

따라서 `a[2:5]`는 벡터 `a`에서 인덱스 2부터 인덱스 4까지의 값을 추출합니다. 파이썬의 인덱싱이 0부터 시작하므로, 인덱스 2는 실제로 세 번째 값이고, 인덱스 4는 다섯 번째 값입니다. 그리고 인덱스 5는 포함되지 않습니다.

```
# 첫 번째, 세 번째, 다섯 번째 값 추출
print(a[[0, 2, 4]])
```

```
[ 7 15  8]
```

```
# 두 번째 값 제외하고 추출
print(np.delete(a, 1))
```

```
[ 7 15 11  8  7 19 11 11  4]
```

인덱싱 안에 리스트가 들어가도 됩니다. 또한, 인덱싱을 중복해서 선택이 가능합니다.

```
# 인덱싱 중복 선택
print(a)
print(a[[1, 1, 3, 2]])
```

```
[ 7 20 15 11  8  7 19 11 11  4]
[20 20 11 15]
```

대괄호 연산자와 비교 연산자를 사용한 벡터 슬라이싱은 다음과 같이 수행할 수 있습니다.

```
b = a[a > 3]
print(b)
```

```
[ 7 20 15 11  8  7 19 11 11  4]
```

위 예제에서는 벡터 `a`의 값이 3보다 큰 요소만 추출하여 `b`에 할당합니다.

5.1.2 논리 연산자와 조건문

대괄호 안에 연산자를 조합하여 원하는 값을 추출할 수도 있습니다.

```
b = a[(a > 2) & (a < 9)]
print(b)
```

```
[7 8 7 4]
```

5.1.3 알아두면 쓸데있는 연산자들

- ==, !=

```
print(a[a == 8])
print(a[a != 8])
```

```
[8]
[ 7 20 15 11  7 19 11 11  4]
```

- % (나머지), // (몫)

나머지 연산자(%)를 사용하여 벡터 슬라이싱을 수행할 수도 있습니다. 나머지 연산자를 사용하여 특정 패턴의 값만 추출할 수 있습니다.

```
b = a[np.arange(1, 11) % 2 == 1]
print(b)
```

```
[ 7 15  8 19 11]
```

위 예제에서는 `np.arange(1, 11)`을 사용하여 인덱스 벡터를 생성합니다. 이후, `%` 연산자를 사용하여 인덱스 벡터를 2로 나눈 나머지가 1인 요소만 추출합니다. 이를 통해 벡터 `a`에서 홀수 번째 요소만 추출할 수 있습니다.

다른 예로, 3으로 나눈 나머지가 0인 요소만 추출하는 코드는 다음과 같습니다.

```
b = a[a % 3 == 0]
print(b)
```

```
[15]
```

- & (AND)

```

```

```
x = np.array([True, True, False])
y = np.array([True, False, False])
print(x & y)
```

```
[ True False False]
```

- | (OR)

```
print(x | y)
```

```
[ True  True False]
```

5.1.3.a 조건문 혼합하기

위에서 배운 논리 연산자 내용을 떠올리면서 다음 코드의 결과값을 해석해보자.

```
import numpy as np
a = np.array([1, 2, 3, 4, 16, 17, 18])  # 예시 배열

result = a[(a == 4) & (a > 15)]
print(result)
```

```
[]
```

`a[(a == 4) & (a > 15)]`는 배열 a에서 값이 4인 원소 중에서 15보다 큰 원소들을 선택하는 코드입니다. &는 논리 연산자 AND를 나타내며, 두 개의 조건이 모두 만족하는 원소를 선택합니다. 따라서 이 코드에서는 값이 4인 원소와 15보다 큰 원소 두 가지 조건을 모두 만족하는 원소는 없으므로, 빈 배열을 반환합니다.

```
import numpy as np
a = np.array([1, 2, 3, 4, 16, 17, 18])  # 예시 배열

result = a[(a == 4) | (a > 15)]
print(result)
```

```
[ 4 16 17 18]
```

`a[(a == 4) | (a > 15)]`는 배열 a에서 값이 4인 원소 또는 15보다 큰 원소를 선택하는 코드입니다. |는 논리 연산자 OR를 나타내며, 두 개의 조건 중 하나 이상을 만족하는 원소를 선택합니다. 따라서 이 코드에서는 값이 4인 원소와 15보다 큰 원소 중 하나 이상의 조건을 만족하는 숫자들을 선택합니다.

5.1.4 필터링을 이용한 벡터 변경

앞에서 배운 필터링을 이용하면, 벡터에 대한 조건문을 사용하여 벡터의 일부 값을 변경할 수 있습니다.

```
import numpy as np
a = np.array([5, 10, 15, 20, 25, 30])  # 예시 배열

a[a >= 10] = 10
a
```

```
array([ 5, 10, 10, 10, 10, 10])
```

`a[a >= 10]`는 `a` 벡터에서 10 이상인 원소들을 선택한 것을 의미합니다. 이 선택된 원소들에 대해서 10이 할당되면서, `a` 벡터에서 10 이상인 값들은 모두 10으로 변경됩니다. 이후에 `a`를 출력하면, `a` 벡터에서 10 이하인 값들은 그대로 유지되고, 10 이상인 값들은 모두 10으로 변경되어 있음을 확인할 수 있습니다.

5.1.5 조건을 만족하는 위치 탐색 `np.where()`

벡터에 대한 조건문과 `np.where()` 함수를 사용하여, 조건을 만족하는 원소의 위치를 선택할 수 있습니다. `a < 7`은 `a` 벡터에서 7보다 작은 원소들에 대해 논리값 `True`를, 7 이상인 원소들에 대해 논리값 `False`를 반환합니다. `a < 7`의 결과는 다음과 같습니다.

```
import numpy as np
a = np.array([1, 5, 7, 8, 10])  # 예시 배열

result = a < 7
result
```

```
array([ True,  True, False, False, False])
```

`np.where()` 함수를 이용하여 논리값이 `True`인 원소의 위치를 선택합니다. 따라서 `np.where(a < 7)`은 `a` 벡터에서 7보다 작은 원소들의 위치를 반환하게 됩니다.

```
import numpy as np
a = np.array([1, 5, 7, 8, 10])  # 예시 배열

result = np.where(a < 7)
result
```

```
(array([0, 1], dtype=int64),)
```

이렇게 np.where() 함수를 이용하면 선택된 원소의 위치를 반환할 수 있으며, 이를 활용하여 다양한 계산을 수행할 수 있습니다.

5.1.6 벡터 함수 사용하기

파이썬에서도 다양한 벡터 함수를 제공합니다. 이러한 함수를 사용하면 벡터의 합계, 평균, 중앙값, 표준편차 등을 계산할 수 있습니다.

```
import numpy as np

# 벡터 함수 사용하기 예제
a = np.array([1, 2, 3, 4, 5])
sum_a = np.sum(a)          # 합계 계산
mean_a = np.mean(a)        # 평균 계산
median_a = np.median(a)    # 중앙값 계산
std_a = np.std(a, ddof=1)  # 표준편차 계산

sum_a, mean_a, median_a, std_a
```

```
(15, 3.0, 3.0, 1.5811388300841898)
```

5.1.7 빈 칸을 나타내는 방법

5.1.7.a 데이터가 정의 되지 않은 np.nan

np.nan는 정의 되지 않은 값(not a number)을 나타냅니다. np.nan를 벡터에 추가하면 해당 위치에는 nan라는 이상치가 들어갑니다. nan는 실제로 값을 가지고 있지 않지만, 벡터의 길이나 타입을 유지하기 위해 존재하는 것입니다.

- nan: not a number

```
import numpy as np

a = np.array([20, np.nan, 13, 24, 309])
a
```

```
array([ 20.,  nan,  13.,  24., 309.])
```

벡터 안에 nan가 들어있는 경우, 계산 값이 nan로 나오게 됩니다. 이유는 숫자에 nan를 더하면 nan가 되기 때문입니다. 이러한 것을 방지하기 위해서, 많은 함수들에는 nan 무시 옵션이 들어있습니다.

```
np.mean(a)
```

```
nan
```

- nan 무시 옵션

```
np.nanmean(a)  # nan 무시 함수
```

```
91.5
```

5.1.7.b 값이 없음을 나타내는 None

None은 아무런 값도 없는 상태를 나타냅니다.

- nan와 None의 차이

5.1.7.b.a None

- 의미: None은 값이 없음을 나타내는 특수한 상수입니다.
- 타입: NoneType 타입입니다.
- 사용처: 주로 변수 초기화, 함수 반환 값, 조건문, 기본 인자값 등에 사용됩니다.
- 비교: None과의 비교는 is 연산자를 사용하여 이루어집니다.

```
my_variable = None
if my_variable is None:
    print("변수에 값이 없습니다.")
```

```
변수에 값이 없습니다.
```

- 수치연산 불가

```
None + 1
```

```
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

5.1.7.b.b np.nan

- 의미: np.nan은 “Not a Number"의 약자로, 수치 연산에서 정의되지 않은 값이나 잘못된 값을 나타냅니다.
- 타입: float 타입입니다. numpy 라이브러리에서 제공하는 상수입니다.
- 사용처: 주로 데이터 분석에서 결측값(missing value)을 나타내기 위해 사용됩니다.
- 비교: np.nan과의 비교는 직접적으로 == 연산자를 사용할 수 없고, numpy의 np.isnan() 함수를 사용해야 합니다.

```
import numpy as np
```

```
value = np.nan
if np.isnan(value):
    print("값이 NaN입니다.")
```

```
값이 NaN입니다.
```

- 수치연산 가능

```
np.nan + 1
```

```
nan
```

5.1.8 빈 칸을 제거하는 방법

빈 칸을 제거하는 방법은 다음과 같습니다. `np.isnan()` 함수는 벡터 `a`의 원소가 `nan`인지를 아닌지를 알려주는 함수입니다. `nan`인 경우 `True`를 반환하고, 그렇지 않은 경우 `False`를 반환합니다.

따라서, 이러한 논리 벡터를 사용하여 벡터 필터링을 하게 되면, 다음과 같이 `nan`이 생략된 벡터를 얻을 수 있습니다.

```
a_filtered = a[~np.isnan(a)]
a_filtered
```

```
array([ 20.,  13.,  24., 309.])
```

5.1.9 벡터 합치기

5.1.9.a 벡터는 숫자만 묶을 수 있는 것이 아니다.

벡터는 같은 타입의 정보 (숫자, 문자)를 묶어놓은 것입니다. 즉, 숫자면 숫자, 문자면 문자이기만 하면, 묶을 수 있습니다. 다음은 문자열로 이루어진 벡터입니다.

```
import numpy as np

str_vec = np.array(["사과", "배", "수박", "참외"])
str_vec
str_vec[[0, 2]]
```

```
array(['사과', '수박'], dtype='<U2')
```

그렇다면, 문자와 숫자를 섞어서 벡터를 만든다면 어떨까요?

```
import numpy as np

mix_vec = np.array(["사과", 12, "수박", "참외"], dtype=str)
mix_vec
```

```
array(['사과', '12', '수박', '참외'], dtype='<U2')
```

결과를 살펴보면, 파이썬은 자동으로 통일할 수 있는 타입(문자) 정보로 바꿔서, 벡터로 저장하는 것을 관찰할 수 있습니다.

5.1.10 여러 벡터들을 묶어보자

여러 개의 벡터들을 하나로 묶을 수 있는 방법이 세 가지 존재합니다. 첫 번째 방법은 `np.concatenate()` 함수를 사용하는 것입니다. 앞에서 정의한 str_vec와 mix_vec을 묶어 하나의 벡터로 만들려면 다음과 같이 할 수 있습니다.

```
combined_vec = np.concatenate((str_vec, mix_vec))
combined_vec
```

```
array(['사과', '배', '수박', '참외', '사과', '12', '수박', '참외'], dtype='<U2')
```

5.1.10.a `np.column_stack()`와 `np.row_stack()`

`np.column_stack()` 함수는 벡터들을 세로로 붙여줍니다.

```
col_stacked = np.column_stack((np.arange(1, 5), np.arange(12, 16)))
col_stacked
```

```
array([[ 1, 12],
       [ 2, 13],
       [ 3, 14],
       [ 4, 15]])
```

`np.row_stack()` 함수는 벡터들을 가로로 쌓아줍니다.

```
row_stacked = np.row_stack((np.arange(1, 5), np.arange(12, 16)))
row_stacked
```

```
array([[ 1,  2,  3,  4],
       [12, 13, 14, 15]])
```

5.1.10.b 길이가 다른 벡터 합치기

만약 앞의 방법으로 합칠 때 길이가 다른 벡터를 합치게 되면 어떤 일이 벌어질까요? 다음의 코드를 보겠습니다.

```
uneven_stacked = np.column_stack((np.arange(1, 5), np.arange(12, 18)))
uneven_stacked
```

```
ValueError: all the input array dimensions for the concatenation axis must match exactly, but along dimension 0, the array at index 0 has size 4 and the array at index 1 has size 6
```

보시는 바와 같이 길이가 다른 경우 쌓을 수 없는 것을 확인 할 수 있습니다.

```
import numpy as np

# 길이가 다른 벡터
vec1 = np.arange(1, 5)
vec2 = np.arange(12, 18)
vec1 = np.resize(vec1, len(vec2))
vec1
```

```
array([1, 2, 3, 4, 1, 2])
```

`np.resize()` 함수를 사용하면 길이를 강제로 맞춰주고, 값을 앞에서부터 채워줍니다.

```
# 두 벡터를 세로로 쌓기
uneven_stacked = np.column_stack((vec1, vec2))
uneven_stacked
```

```
array([[ 1, 12],
       [ 2, 13],
       [ 3, 14],
       [ 4, 15],
       [ 1, 16],
       [ 2, 17]])
```

5.1.11 연습 문제

5.1.11.a 연습 문제 1

주어진 벡터의 각 요소에 5를 더한 새로운 벡터를 생성하세요.

```

```

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

```
array([1, 2, 3, 4, 5])
```

5.1.11.b 연습 문제 2

주어진 벡터의 홀수 번째 요소만 추출하여 새로운 벡터를 생성하세요.

```
a = np.array([12, 21, 35, 48, 5])
a
```

```
array([12, 21, 35, 48,  5])
```

5.1.11.c 연습 문제 3

주어진 벡터에서 최대값을 찾으세요.

```
a = np.array([1, 22, 93, 64, 54])
a
```

```
array([ 1, 22, 93, 64, 54])
```

5.1.11.d 연습 문제 4

주어진 벡터에서 중복된 값을 제거한 새로운 벡터를 생성하세요.

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

```
array([1, 2, 3, 2, 4, 5, 4, 6])
```

5.1.11.e 연습 문제 5

주어진 두 벡터의 요소를 번갈아 가면서 합쳐서 새로운 벡터를 생성하세요.

```
a = np.array([21, 31, 58])
b = np.array([24, 44, 67])
a
b
```

```
array([24, 44, 67])
```

5.1.11.f 연습 문제 6

다음 a 벡터의 마지막 값은 제외한 두 벡터 a와 b를 더한 결과를 구하세요.

```
a = np.array([1, 2, 3, 4, 5])
b = np.array([6, 7, 8, 9])
```

5.1.12 해답

5.1.12.a 연습 문제 1

주어진 벡터의 각 요소에 5를 더한 새로운 벡터를 생성하세요.

```
a = np.array([1, 2, 3, 4, 5])
a + 5
```

```
array([ 6,  7,  8,  9, 10])
```

5.1.12.b 연습 문제 2

주어진 벡터의 홀수 번째 요소만 추출하여 새로운 벡터를 생성하세요.

```
a = np.array([12, 21, 35, 48, 5])
a[::2]
```

```
array([12, 35,  5])
```

5.1.12.c 연습 문제 3

주어진 벡터에서 최대값을 찾으세요.

```
a = np.array([1, 22, 93, 64, 54])
np.max(a)
```

```
93
```

5.1.12.d 연습 문제 4

주어진 벡터에서 중복된 값을 제거한 새로운 벡터를 생성하세요.

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

```
array([1, 2, 3, 4, 5, 6])
```

5.1.12.e 연습 문제 5

주어진 두 벡터의 요소를 번갈아 가면서 합쳐서 새로운 벡터를 생성하세요.

```
a = np.array([21, 31, 58])
b = np.array([24, 44, 67])
c = np.empty(a.size + b.size, dtype=a.dtype)
c[0::2] = a
c[1::2] = b
c
```

```
array([21, 24, 31, 44, 58, 67])
```

5.1.12.f 연습 문제 6

다음 `a` 벡터의 마지막 값은 제외한 두 벡터 `a`와 `b`를 더한 결과를 구하세요.

```
a = np.array([1, 2, 3, 4, 5])
b = np.array([6, 7, 8, 9])
c = a[:-1] + b
c
```

```
array([ 7,  9, 11, 13])
```