WED

Indexing

:ndarray 에서 원하는 데이터를 선택

- 기본적인 인덱싱은 list 와 많이 유사
- index라는 용어는 각 행의 이름으로 판단

배열명 [순번]
: 순번에 해당하는 하나의 데이터를 리턴

- 1 차원 배열
: 순번에 해당하는 행을 리턴

- 2 차원 배열
: 순번에 해당하는 행을 리턴

—〉 순번은 기본적으로 0 부터 시작하지만 음수가 가능
음수를 가장 뒤에서 부터 Indexing

배열명 [시작위치 : 종료위치]
: 범위에 해당하는 데이터를 리턴

- 범위설정을 할 때, 종료위치 포함여부를 확인해보는 것이 좋다

특징
1. ' : '을 포함한 상태에서 시작위치를 생략 시 처음부터
종료 위치를 생략 ——〉 ㅊ끝까지

2. 전체를 선택하고자 할 대 [:] 방법을 이용할 수 있다 .

3. 이차원 이상인 경우
[행번호] [열번호] 형태
[행번호 , 열번호] 형태로 Indexing
특정 indexing 생략 시 인덱스는 전체로 간주

———〉 Indexing 한 데이터는 원본데이터의 View(복제한 것이 아님)
Indexing 한 데이터 변경시 원본 데이터도 변경

복제 하려면 ?
Indexing 한 후
.copy() 를 호출해서 복제

```python
import numpy as np

# 1 차원 배열 생성
ar = np.array([1, 2, 3, 4, 5, 6])

# Matrix 라고 부를 때 ---> array 행 과 열의 개수가 같다

# 1차원 배열을 생성
matrix = np.array(range(1, 21))
print("before ----------- ")
print(matrix)
print()

# matrix를 4행 5 열의 2차원 배열로 변환
matrix = matrix.reshape(4, 5)
print("After ----------- ")
print(matrix)
```

```
before -----------
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]

After -----------
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]
```

```python
# Indexing
print("----------------------Indexing ")
# 1 차원 배열에서 하나의 요소를 추출
print("1 차원 배열에서 하나의 요소를 추출")
print(ar[1])
print()

# 2 차원 배열에서 하나의 요소를 추출
print("2 차원 배열에서 하나의 요소를 추출")
print(matrix[1, 2])
print(matrix[1][2])
print()


print("-----------------------------------")
print("2 차원 배열에서 하나의 Index를 대입해서 추출 , 1행 전체")
print(matrix[1])
```

```
--------------------Indexing
1 차원 배열에서 하나의 요소를 추출
2

2 차원 배열에서 하나의 요소를 추출
8
8

---------------------------------
2 차원 배열에서 하나의 Index를 대입해서 추출 , 1행 전체
[ 6  7  8  9 10]
```

```python
# 특정위치 출력
print("특정위치 출력----------------------")
print(f"ar : {ar}")
print()

# 특정 위치 부터 끝까지
print("특정 위치 부터 끝까지")
print(ar[3:])
print()

# 시작 위치 부터 특정 위치 까지
print("시작 위치 부터 특정 위치 까지")
print(ar[:3])
```

```
특정위치 출력----------------------
ar : [1 2 3 4 5 6]

특정 위치 부터 끝까지
[4 5 6]

시작 위치 부터 특정 위치 까지
[1 2 3]
```

```python
# 일반적인 인덱싱 == 데이터의 view
print("일반적인 인덱싱 == 데이터의 view")
print(f"ar : {ar}")
print()

# ar 의 0 번째 데이터를 변경
print("ar 의 0 번째 데이터를 변경")
ar[0] =10000
print(ar)
print(f"ar : {ar}")
print()

# xr = ar 로 지정
print("xr = ar 로 지정")
xr = ar[0:3]
xr[0] = 999999
print(f"ar : {ar}")
print()

# 복제를 하고자 하면 인덱싱한후 copy() 호출
print("복제를 하고자 하면 인덱싱한후 copy() 호출")
br = ar[0:3].copy()
br[0] = 111111
print(f"ar : {ar}")
print(f"br : {br}")
print()
```

```
일반적인 인덱싱 == 데이터의 view
ar : [1 2 3 4 5 6]

ar 의 0 번째 데이터를 변경
[10000     2     3     4     5     6]
ar : [10000     2     3     4     5     6]

xr = ar 로 지정
ar : [999999      2      3      4      5      6]

복제를 하고자 하면 인덱싱한후 copy() 호출
ar : [999999      2      3      4      5      6]
br : [111111      2      3]
```

Fancy Indexing

```python
import numpy as np

ar = np.array([1, 2, 3, 4, 5])
print(f"ar : {ar}")
print()

# 범위 대신 list 를 이용해서 Indexing 하는 것을 Fancy Indexing 이라고 한다
print("범위 대신 list 를 이용해서 Indexing 하는 것을 Fancy Indexing 이라고 한다")
xr = ar[[0, 3, 4]]
print(f"xr = {xr}")
print()

# Fancy Indexing 은 복제를 실행함
print("Fancy Indexing 은 복제를 실행함")
xr[0] = 999999999
print(f"ar : {ar}")
print(f"xr = {xr}")
```

```
ar : [1 2 3 4 5]

범위 대신 list 를 이용해서 Indexing 하는 것을 Fancy Indexing 이라고 한다
xr = [1 4 5]

Fancy Indexing 은 복제를 실행함
ar : [1 2 3 4 5]
xr = [999999999         4         5]
```

데이터 Indexing 할 때 고려사항

: 원본을 가지고 작업을 할 것인지
복제를 해서 작업을 할것인지 시중하게 결정하는 것이 중요

- 원본을 가지고 작업을 하게 되면 뒤로 돌아갈 수 없음
- 여러 작업 처리 시 에러가 발생 혹은 잘못된 지점부터 다시 수행해야 한다

: 권장하지 않음

데이터 분석 분야는

많은 양의 데이터를 외부에서 불러들여서 작업을 수행하는 경우가 많고
데이터 로딩 시간이 길기 때문에 로딩을 다시 해야 하는 상황을 만들지 않는 것이 좋다

클라이언트 Application 개발

클라이언트의 환경을 고민 —> 복제를 하게되면 메모리를 많이 사용
메모리 부족으로 인해서 Application이 제대로 동작하지 않음

Server Application 개발

복제를 많이함 — > 동기화 문제

Transaction 문제

여러 번 복제를 했을 때 변경된 내용을 어떤 방식으로 적용할 것이냐 하는 문제

Boolean Indexing

: index 에 bool 배열을 적용하면 결과가 True 인 데이터만 Return

bool 배열 끼리 연산을 하고자 하면

> and , or 가 아닌
> & 나 | , ^ 를 사용해야 한다.

and, or

: 데이터 1개를 가지고 연산을 수행
—> list , tuple 같은 집단 자료형을 가지고 연산
집단 자료형의 데이터를 하나의 bool 로 변환해서 연산
: 조건 논리 연산자

& , | , ^

: 집단 자료형을 가지고 연산을 하면
각 요소들을 가지고 연산을 해서 다시 집단 자료형으로 return
: 비트 논리 연산자

```
import numpy as np

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

# True 위치의 데이터만 골라서 복제를 해서 리턴
print("True 위치의 데이터만 골라서 복제를 해서 리턴")
xr = ar[[True, False, True, False, True, False]]
print(xr)
print()

# xr[0] 의 데이터를 변경해도 ar 에 영향이 없다
xr[0] = 1000
print(f'ar : {ar}')
print(f'xr : {xr}')

print("And 나 Or 연산자 ==================================")
cas1 = [True, False, True, False, True, False]
cas2 = [True, False, True, False, False, True]

br = ar[cas1 and cas2]
# 이거 뒤에오는 cas2 값을 덮어 씌우는 것 같음
print(f'br : {br}')
```

```
True 위치의 데이터만 골라서 복제를 해서 리턴
[1 3 5]


ar : [1 2 3 4 5 6]
xr : [1000    3    5]
And 나 Or 연산자 ==================================
br : [1 3 6]
```

numpy 연산

- 산술 연산
: 차원이 동일하고 데이터 개수가 동일한 ndarray 끼리 산술연산이 가능
차원이 서로 다른 데이터끼리도 연산이 가능 ——〉 broadcast 연산을 수행
차원이 큰 쪽의 데이터를
작은 쪽의 모든 데이터와 연산을 수행해서
결과를 큰쪽으로 생성해서 리턴

- scala data(1개) : 산술 연산이 가능
모든 요소를 scala data 와 연산해서 결과를 배열로 리턴

```
import numpy as np

x = 100
ar1 = np.array([1, 2, 3])
ar2 = np.array([4, 5, 6])

matrix1 = np.array(range(0, 6, 1)).reshape(2, 3)
matrix2 = np.array(range(0, 6, 1)).reshape(3, 2)

# 동일한 크기의 배열끼리 연산
print("동일한 크기의 배열끼리 연산")
print(ar1 + ar2)
print()

# 2개의 배열은 2차원인 것은 같지만 구조가 다르기 때문에 연산 불가
print("2개의 배열은 2차원인 것은 같지만 구조가 다르기 때문에 연산 불가")
print(matrix2 + matrix1)
print()
```


```
Traceback (most recent call last):
  File "/Users/mac/PycharmProjects/pythonProject/class/0209/_4_ndarray_Oper.py", line 17, in <module>
    print(matrix2 + matrix1)
ValueError: operands could not be broadcast together with shapes (3,2) (2,3)
동일한 크기의 배열끼리 연산
[5 7 9]
```

```python
# scala Data 와 배열의 연산
# 배열의 각 요소에 scala data를 연산을 한 후 결과를 배열로 리턴
print("배열의 각 요소에 scala data를 연산을 한 후 결과를 배열로 리턴")
print(x+ar1)
print()

# 차원이 다른 배열끼리의 연산
# 1 차원 배열의 요소 개수 와 2 차원 배열의 행을 구성하는 열의 개수가 같아서 수행
print("1 차원 배열의 요소 개수 와 2 차원 배열의 행을 구성하는 열의 개수가 같아서 수행")
print(ar1 + matrix1)
print()

# 차원이 다른 배열끼리의 연산 - 2
# 1 차원 배열의 요소 개수 와 2 차원 배열의 행을 구성하는 열의 개수가 달라서 Error
print("1 차원 배열의 요소 개수 와 2 차원 배열의 행을 구성하는 열의 개수가 달라서 Error")
print(ar1 + matrix2)
print()
```

ar1 : 1차원 3열

ar2 : 2차원 3열

```
Traceback (most recent call last):
  File "/Users/mac/PycharmProjects/pythonProject/class/0209/_4_ndarray_Oper.py", line 35, in <module>
    print(ar1 + matrix2)
ValueError: operands could not be broadcast together with shapes (3,) (3,2)
동일한 크기의 배열끼리 연산
[5 7 9]

배열의 각 요소에 scala data를 연산을 한 후 결과를 배열로 리턴
[101 102 103]

1 차원 배열의 요소 개수 와 2 차원 배열의 행을 구성하는 열의 개수가 같아서 수행
[[1 3 5]
 [4 6 8]]

1 차원 배열의 요소 개수 와 2 차원 배열의 행을 구성하는 열의 개수가 달라서 Error
```

차원이 작은쪽 데이터 우선함 Return : 차원이 큰쪽 값으로 Return

차원이 다른 데이터끼리 연산을 수행할 때 전체 데이터를 순회하면서 수행하는 것을 BroadCast 연산이라고 한다 .

numpy 의 연산

: 눈에 보이지는 않지만 Vector 화 된 연산을 수행한다

반복문을 사용하지 않고 같은 위치의 데이터끼리 연산을 수행하는 것
(반복문을 사용할 때 보다 빠르게 연산을 수행)

numpy 의 데이터 가공

python의 집단 자료형에서는
데이터를 가공하고자 할 때
1. 반복문을 사용하거나
2. map 함수를 이용

— numpy에서는 vectorize 함수를 이용해서 벡터화된 함수를 적용하는 것이 가능

```python
import numpy as np

ar1 = np.array([100, 200, 300])

# 한줄 짜리 함수를 생성
def add_func(i):
    return i + 100

# 람다를 이용해서 작성
add_lamda = lambda i: i + 100

# vector 화 된 함수 생성
vectorized_add_lamda = np.vectorize(add_lamda)

# vector 화 된 함수를 이용한 데이터 가공
print(vectorized_add_lamda(ar1))
```

Vector X

Vector화

numpy 연산
= Vector화된연산

Try

```
/Users/mac/Pych
[200 300 400]
```

배열에서 크기 비교 한 동일성 확인 연산 가능

BroadCast 연산 가능

배열 간에는 연산자 대신 함수를 이용하는 것이 가능
: equal, not_equal, greater, greater_equal , less, less_equal

```python
import numpy as np

ar1 = np.array([1, 10, 3])
ar2 = np.array([2, 3, 4])

# 각 요소를 가지고 연산을 해서 결과를 배열로 리턴
print("각 요소를 가지고 연산을 해서 결과를 배열로 리턴")
print(ar1 > ar2)
print()

# 함수를 이용
print("함수를 이용")
print(np.greater(ar1, ar2))
print()

# broadcast 연산을 이용해서 특정 조건에 맞는 데이터만 추출
print("broadcast 연산을 이용해서 특정 조건에 맞는 데이터만 추출")
print(f'ar1 : {ar1}')
print(f'ar2 : {ar2}')
print()
print("ar2[ar2 > 3]")
print(ar2[ar2 > 3])
print()
print("ar2[ar1 > 3]")
print(ar2[ar1 > 3])
print()

# ----> 이 원리를 데이터 전처리에 많이 사용
# outlier(이상치) 추출에 많이 사용

# 1 보다 크고 10 보다 작은 데이터 추출
print("보다 크고 10 보다 작은 데이터 추출")
print(ar1[(ar1 > 1) & (ar1 < 10)])
```

조건에 만족하는 index를 출력

```
각 요소를 가지고 연산을 해서 결과를 배열로 리턴
[False  True False]

함수를 이용
[False  True False]

broadcast 연산을 이용해서 특정 조건에 맞는 데이터만 추출
ar1 : [ 1 10  3]
ar2 : [2 3 4]

ar2[ar2 > 3]
[4]

ar2[ar1 > 3]
[3]

보다 크고 10 보다 작은 데이터 추출
[3]
```

배열 간의 논리 연산
: & , | , ^ 연산 가능
- logical_and , logical_or, logical_xor

in연산자
: 특정 데이터가 배열에 포함되어 있는지 확인하는 연산에 사용

배열 간의 할당 연산 가증
: += , -= 연산 가능

배열의 전치 - 축을 변경

: 2차원 배열의 경우 T 라는 속성을 이용해서
행과 열을 변경한 데이터를 리턴 받을 수 있다

transpose 라는 함수를 이용해서 전치 할 수 있는데
이 경우는 변경할
행 과 열의 순번을 순서대로 대입

전치.

2 차원 배열은 거의 사용하지 않고 3 차원 이상의 배열에서 사용

사용하는 경우
1. 배열의 전치는 데이터를 가져올 때 구조의 방향이 다를 때
2. 데이터 증강시 간접적으로 사용

```python
import numpy as np

# 0 에서 14 까지 요소가 들어있는 5, 3 matrix 생성
print("0 에서 14 까지 요소가 들어있는 5, 3 matrix 생성")
matrix = np.arange(15).reshape(5, 3)
print(matrix)
print()

# 전치 --> 방향을 바꿈 (5,3) 에서 (3,5)
print("전치 --> 방향을 바꿈 (5,3) 에서 (3,5)")
print(matrix.T)
print()

# 0 번과 1 번 축을 변경 ---> 전치를 해준 것과 값이 같다
print("0 번과 1 번 축을 변경 ---> 전치를 해준 것과 값이 같다======")
print("----- before")
print(matrix)
print("----- after")
print(matrix.transpose(1, 0))
print()


# 3 차원 만들기 // 딥러닝 할 때 차원 변환을 많이 한다
# -1 의 의미를 알아 두어야 한다 //  마지막 차원의 개수가 -1 일때 알아서 분할하여 생성
# flattern도 알아두어야 한다  // reshape 와 다르게 새로운 배열을 생성해서 리턴하는 것이다
print("3 차원 만들기 // 딥러닝 할 때 차원 변환을 많이 한다=======")
matrix = np.arange(30).reshape(5, 3, -1)
print(matrix)
print()

# 3 차원 이상일 때는 순서를 직접 설정
print("before -----------")
print(matrix)
print("after -----------")
print(matrix.transpose(0, 2, 1))
```

5행 3열

데이터개수는
생성할 Matrix 수 X 차원수 X (열의수)

높이, 차원, 열
순서대로 0, 1, 2
인데
0 2 1
로 바뀌어서
차원의 수와 열의수가
바뀜

그냥하면
알맞는 열의수를
자동으로 생성

```
0 에서 14 까지 요소가 들어있는 5, 3 matrix 생성
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]]

전치 --> 방향을 바꿈 (5,3) 에서 (3,5)
[[ 0  3  6  9 12]
 [ 1  4  7 10 13]
 [ 2  5  8 11 14]]

0 번과 1 번 축을 변경 ---> 전치를 해준 것과 값이 같다======
----- before
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]]
----- after
[[ 0  3  6  9 12]
 [ 1  4  7 10 13]
 [ 2  5  8 11 14]]
```

5행 3열

3행 5열

```
3 차원 만들기 // 딥러닝 할 때 차원 변환을 많이 한다=======
[[[ 0  1]
  [ 2  3]
  [ 4  5]]

 [[ 6  7]
  [ 8  9]
  [10 11]]

 [[12 13]
  [14 15]
  [16 17]]

 [[18 19]
  [20 21]
  [22 23]]

 [[24 25]
  [26 27]
  [28 29]]]
```

```
before -----------
[[[ 0  1]
  [ 2  3]
  [ 4  5]]

 [[ 6  7]
  [ 8  9]
  [10 11]]

 [[12 13]
  [14 15]
  [16 17]]

 [[18 19]
  [20 21]
  [22 23]]

 [[24 25]
  [26 27]
  [28 29]]]
```

```
after -----------
[[[ 0  2  4]
  [ 1  3  5]]

 [[ 6  8 10]
  [ 7  9 11]]

 [[12 14 16]
  [13 15 17]]

 [[18 20 22]
  [19 21 23]]

 [[24 26 28]
  [25 27 29]]]
```

Random

: numpy.random

- sampling (표본 추출) 할 때 중요

seed (seed = 시드 번호)
: seed 설정 , 설정하지 않으면 seed 가 현재 시간을 기준으로 Random 으로 설정

Random 설정 시 어떤 데이터가 return 될 지 예측하기 어려움

* machine Learning 에서 seed 를 고정값으로 설정하는 이유 ?

여러 모델을 비교하거나
하나의 모델을 가지고 여러 개의 샘플을 만들어서 테스트 하고자 할 때
동일한 Sampling 하기 위해서

* machine Learning 에서는 42 라는 seed 를 선호
——〉 아무 의미 없음

random, sample , random_sample
: 0.0 ~ 1.0 사이의 난수

randint()
: 범위 내의 정수리턴

rand()
: 정규 분포에서 표본 추ㄹ출

normal()
: 가우시안 분포

Permutation
: 정수 하나를 받아서 0 에서 정수 이전까지의 배열을 생성하여 랜덤하고 배치하고 return

Shuffle
: Permutation 과 유사한 추출

———〉 비복원 추출

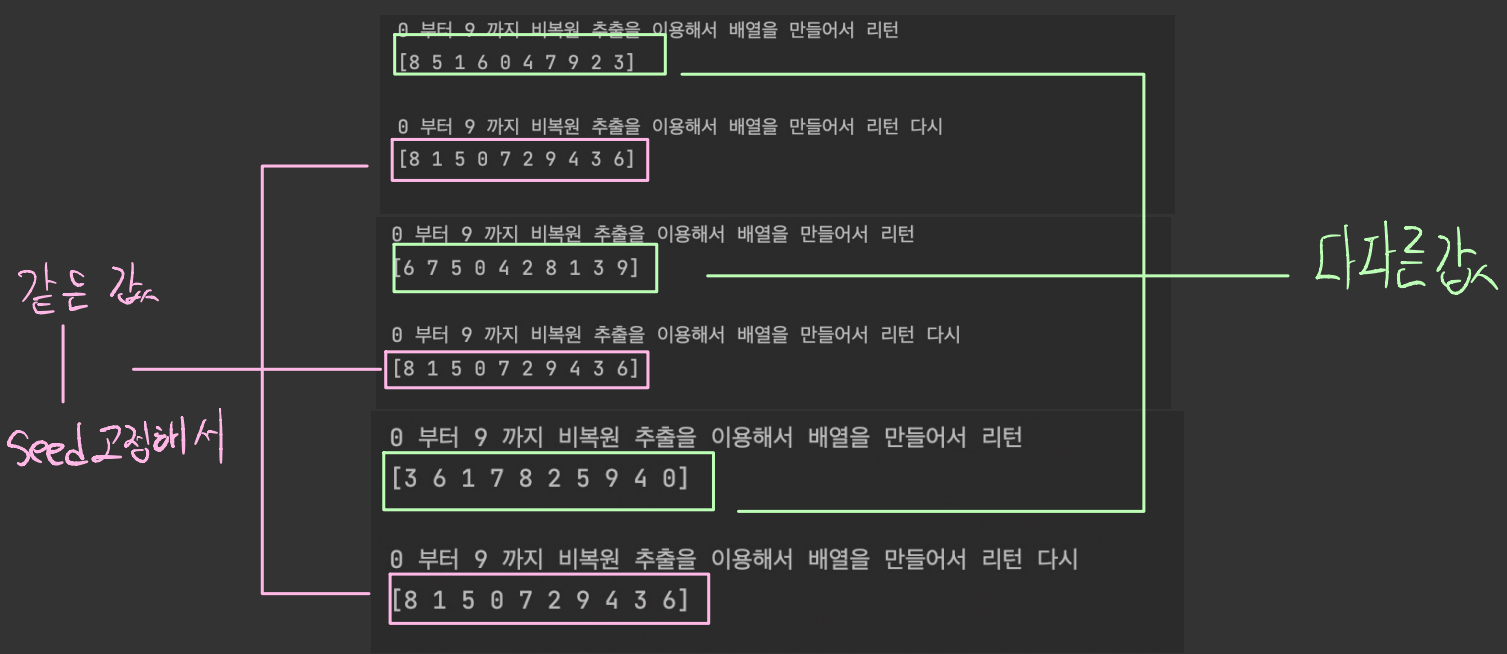
choice()
: 배열에서 복원 추출을 이용해서 데이터를 생성한 후 return

* 비복원 추출 과 복원 추출

1. 비복원 추출 :
   데이터 모임에서 하나 추출
   추출된 데이터를 데이터 모임에서 제거한 후 샘플 추출

2. 복원 추출 :
   데이터 모임에서 하나의 샘플을 추출
   데이터의 모임에서 추출된 데이터를 제거하지 않고 다음 sample 을 추출
   (동일한 데이터가 여러번 추출될 수 있다)

```python
import numpy as np

# 0 부터 9 까지 비복원 추출을 이용해서 배열을 만들어서 리턴
print("0 부터 9 까지 비복원 추출을 이용해서 배열을 만들어서 리턴")
print(np.random.permutation(10))
print()


# seed를 고정
np.random.seed(42)

# 0 부터 9 까지 비복원 추출을 이용해서 배열을 만들어서 리턴 다시
print("0 부터 9 까지 비복원 추출을 이용해서 배열을 만들어서 리턴 다시 ")
print(np.random.permutation(10))
print()

# 비복원 추출 : 같은 값이 나오지 않음
```



범용 함수

: numpy 에서 사용 가능한 함수

〈 기본 통계 함수〉

1. sum
: 합계

2. mean
: 평균

axis : 중심선

3. median
: 중간값

4. max , min
: 최대 , 최소

5. std , var
: 표준 편차와 분산
——〉 ddof(자유도 : 데이터가 결정되면 자동으로 결정되는 데이터의 개수)
옵션을 1 로 설정하면 비편향 표준 편차 와 분산

6. percentile
: 배열 과 백분위 값을 받아서 배열에서 백분위에 해당하는 데이터를 리턴

- 2 차원 이상인 경우 옵션이 없으면 전체를 1로 간주하고 작업을 수행
- axis 옵션을 이용해서 0 이나 1 을 설정하면 행, 열 단위로 작업을 수행
——〉 keepdims 옵션을 True 로 설정하면 배열과 동일한 차원으로 결과를 리턴

```
1 차원 배열 생성 ===========

1 차원 배열인 ar 을  2차원으로 변경

 1차원 배열인 ar : [1 2 3 4 5 6 7 8]

 2차원 배열인 br : [[1 2]
 [3 4]
 [5 6]
 [7 8]]

===============
 ar 의 합계 : 36

데이터 개수로 나눈 값 : 수학에서 사용 ================+++=============
표준 편차( 자유도 적용 안함 ) : 2.29128784747792

(데이터 개수 -1) 로 나눈 값 : 통계학에서 많이 사용 ================+++=============
표준 편차( 자유도 1 적용 ) : 2.449489742783178

3 사분위 수 구하기 ================+++=============
3 사분위 수 구하기 : 6.25

2차원배열 ======================
최대 값 : 8

행 단위 (axis = 0)최대 값 : [7 8]
열 단위 (axis = 1)최대 값 : [2 4 6 8]
결과를 원래 데이터의 차원인 2 차원으로 만들어서 리턴 ======================
최대 값 :
[[2]
 [4]
 [6]
 [8]]
```

```python
import numpy as np

# 1 차원 배열 생성
print("1 차원 배열 생성 ===========")
# 데이터 개수 8 개
ar = np.array([1, 2, 3, 4, 5, 6, 7, 8])
print()

print("1 차원 배열인 ar 을  2차원으로 변경 ")
br = ar.reshape(4, 2)
print()

print(f" 1차원 배열인 ar : {ar}")
print()
print(f" 2차원 배열인 br : {br}")
print()

print("================")
# 합계
print(f" ar 의 합계 : {np.sum(ar)}")
print()

# 데이터 개수로 나눈 값 : 수학에서 사용
print('데이터 개수로 나눈 값 : 수학에서 사용 ================+++=============')
print(f"표준 편차( 자유도 적용 안함 ) : {np.std(ar)}")
print()
# (데이터 개수 -1) 로 나눈 값 : 통계학에서 많이 사용
print('(데이터 개수 -1) 로 나눈 값 : 통계학에서 많이 사용 ================+++=============')
print(f"표준 편차( 자유도 1 적용 ) : {np.std(ar, ddof=1)}")
print()

# 3 사분위 수 구하기
print("3 사분위 수 구하기 ================+++============= ")
print(f"3 사분위 수 구하기 : {np.percentile(ar, 75)}")
print()

# 2차원 배열
print("2차원배열 ======================")
print(f"최대 값 : {np.max(br)}")
print()
# 행과 열 단위로 최대 값 구하기
print(f"행 단위 (axis = 0)최대 값 : {np.max(br, axis=0)}")
print(f"열 단위 (axis = 1)최대 값 : {np.max(br, axis=1)}")

# 결과를 원래 데이터의 차원인 2 차원으로 만들어서 리턴
print("결과를 원래 데이터의 차원인 2 차원으로 만들어서 리턴 ====================== ")
print(f"최대 값 : \n{np.max(br, axis=1, keepdims=True)}")
```

이건 어디서?

$(n+1)\times\frac{3}{4} = \frac{9\times3}{4} = 6.25$

원래데이터 차원으로 출력

배열 통계 함수

1. argmin, argmzx
: 최소값과 최대 값을 가진 인덱스

2 . nanprod
: 배열 원소 간의 곱
None(Null)
NaN(숫자가 아닌 데이터) 인 데이터는 1 로 간주

* NaN (Not a Number)

3. cumprod()
: 누적 곱

4. cumsum()
: 누적 합

5. diff()
: 정수를 대입하면 n차 차분을 구해준다
기본값은 1.

- 2차원 이상
axis 옵션을 사용
: 행방향 , 열방향으로 연산 가능
keepdims 옵션을 사용하는 것이 가능

keep dimension

8의 index 값

```
import numpy as np

# 1차원 배열 생성
print(" 1차원 배열 생성 ")
ar = np.array([1, 2, 3, 4, 5, 6, 7, 8])
print()

# 최대값 구하기
print("최대값 구하기")
print(f"Max : {np.max(ar)}")
print(f"최대값 위치 : {np.argmax(ar)}")
print()

# 배열에 None 이 있을 때
print("배열에 None 이 있을 때")
br = np.array([1, 2, 3, 4, 5, 6, 7, None])
print(br.dtype)
```

```
1차원 배열 생성

최대값 구하기
Max : 8
최대값 위치 : 7

배열에 None 이 있을 때
object
```

```python
# 누적합
print("누적합 : array 0 부터 계속해서 합산하여 출력")
print(f"누적합 : {np.cumsum(ar)}")
print()

# 원본 데이터 개수 보다 차분을 할때 마다 1 개씩 결과가 줄어든다
print("차분 구하기 ==========")
print(f"1차 차분 : {np.diff(ar)}")
print(f"n차 차분 (n= 2) :{np.diff(ar, n=2)}")
print(f"n차 차분 (n= 3) :{np.diff(ar, n=3)}")
print(f"n차 차분 (n= 4) :{np.diff(ar, n=4)}")
```

```
누적합 : [ 1  3  6 10 15 21 28 36]

1차 차분 구하기
1차 차분 : [1 1 1 1 1 1 1]
n차 차분 (n= 2) :[0 0 0 0 0 0]
n차 차분 (n= 3) :[0 0 0 0 0]
n차 차분 (n= 4) :[0 0 0 0]
```

소수 관련 함수

function

- arround

- round_

- rint

- fix

- ceil

- floor

- trunc

===> 정수를 변환하거나 특정 소수점 자리에서 올림 ,반올림 , 버림 해주는 함수
(통계 에서 많이 사용하고 AI 에서는 사용하지 않음)

숫자 처리 함수

abs

sqrt

squre

modf
: 정수와 소수를 분리해서 배열로 반환

sign
: 부호

논리 함수

isnan()
: NaN 포함 여부 확인

isfinite()
: 유한수 포함 여부

isinf()
: 무한수 포함 여부

logical_not(조건)
: 조건을 만족하지 않으면 True 조건을 만족하면 False를 리턴

where

where(bool 배열 , True 일 때 데이터 , False 일 때 데이터)
—— 조건에 따라 데이터를 선택하도록 한다 .

```
import numpy as np

# 1 차원 배열 생성
ar = np.arange(4)
br = np.array([10 ,20, 30, 40])
print(ar)
print(br)
print()

# condition 생성
condition = np.array([True, False, True, False])

# where 사용하기
# True 일때 ar 값이 출력
# False 일때는 br 값이 출력
print('where 사용하기 ====================== ')
print(np.where(condition, ar, br))
```

True 일때 출력

bool 배열

false 일 때 출력

```
[0 1 2 3]
[10 20 30 40]

where 사용하기 ======================
[ 0 20  2 40]
```

집합 관련 함수

unique()

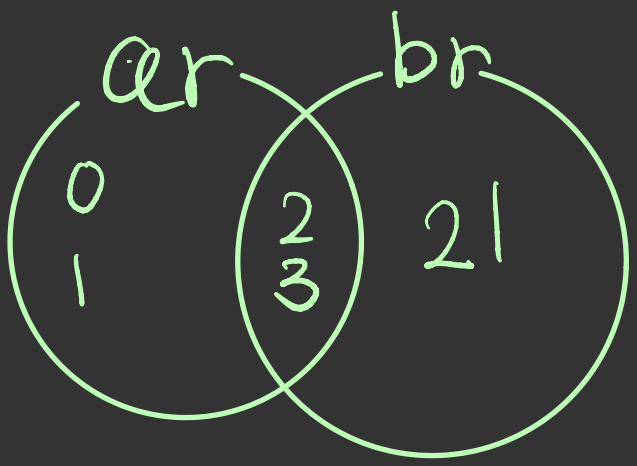
intersect1d()
: 교집합

union1d()
: 합집합

in1d()
: 데이터의 존재 여부를 bool 배열로 리턴

setdiff1d()
: 차집합

setxor1d()
: 한쪽에만 있는 데이터의 집합 (합집합에서 교집합을 제외한 데이터)




```
import numpy as np

# 배열 생성하기
ar = np.arange(4)
br = np.array([21, 2, 3, 3, 3, 3, 2])

# 배열이 잘 생성되었는지 확인
print("배열이 잘 생성되었는지 확인====================================")
print(ar)
print(br)
print()

# 중복제거 Unique
print(f" Unique( 중복 제거 ) :{np.unique(br)} ")
print()

# 차집합
print(f" setdiff1d(차집합--> ar 에서 교집합을 제거 ) : {np.setdiff1d(ar, br)} ")
print(f" setdiff1d(차집합--> br 에서 교집합을 제거 ) : {np.setdiff1d(br, ar)} ")
print()

# ar, br 중 한쪽에만 존재하는 data
print(f" setxor1d : {np.setxor1d(ar, br)}")
print()
```

```
배열이 잘 생성되었는지 확인====================================
[0 1 2 3]
[21  2  3  3  3  3  2]

 Unique( 중복 제거 ) :[ 2  3 21]

 setdiff1d(차집합--> ar 에서 교집합을 제거 ) : [0 1]
 setdiff1d(차집합--> br 에서 교집합을 제거 ) : [21]

 setxor1d : [ 0  1 21]
```

Sort

특징

1. axis 옵션이 있어서
   행, 열 단위 정렬도 가능

2. kind에 정렬이 있어야 가능

```
import numpy as np
help(np.sort)
```

```
kind : {'quicksort', 'mergesort', 'heapsort', 'stable'}, optional
```

```
import numpy as np
help(np.sort)

# array 생성
ar = np.array([99, 2, 13, 1000, 9, 28932])

# 정렬 전 = ar
print(" before ======")
print(ar)
print()

# 정렬 하고 나서 --> br = np.sort(ar)
br = np.sort(ar)
print(" after =======")
print(br)
print()

# 내림차순
# 역순으로 접근하고자 하는 경우 list의 경우 reverse 함수를 이용하지만
# ndarray는 인덱싱을 이용
print(" Descending ======")
br = np.sort(ar)[::-1]
print(br)
print()


# 정렬 알고리즘 설정
# 결과는 같지만 알고리즘이 mergesort로 바뀜
# 정렬 공부 시 selection, bubble , insertion, quick, merge, heap 정렬을 해두는 것이 좋다
print("MergeSort ========")
br = np.sort(ar, kind="mergesort")[::-1]
print(br)
print()
```

```
 before ======
[   99     2    13  1000     9 28932]

 after ======
[    2     9    13    99  1000 28932]

 Descending ======
[28932  1000    99    13     9     2]

MergeSort ========
[28932  1000    99    13     9     2]
```

배열 분할

: split

- 첫번째 parameter
: 분할할 배열

- 두번째 parameter
: 분할할개수

* axis 는 2차원일때 분할하는 방향
두번째 parameter 에 List 대입 시
list 에 작성된 데이터의 개수만큼 분할

```
import numpy as np

# 2 차원 배열 생성
ar = np.array(range(10, 161, 10)).reshape(4,-1)
print(ar)
print()
# 기본적으로 행방향으로 분할
print("행방향으로 2개 분할 ")
ar_split = np.split(ar, 2)
print(ar_split)
print()

# 열방향으로 분할
ar_split = np.split(ar, 2, axis=1)
print("열방향으로 2개 분할")
print(ar_split)
print()
```

```
[[ 10  20  30  40]
 [ 50  60  70  80]
 [ 90 100 110 120]
 [130 140 150 160]]

행방향으로 2개 분할
[array([[10, 20, 30, 40],
       [50, 60, 70, 80]]), array([[ 90, 100, 110, 120],
       [130, 140, 150, 160]])]

열방향으로 2개 분할
[array([[ 10,  20],
       [ 50,  60],
       [ 90, 100],
       [130, 140]]), array([[ 30,  40],
       [ 70,  80],
       [110, 120],
       [150, 160]])]
```

배열 합치기

: 통계 분석을 할 때 많이 사용

concatenates

: 합치고자 하는 배열들을 tuple의 형태로 전달
axis 속성을 이용하면 행 방향이나 , 열 방향으로 합칠 수 있다.

```python
import numpy as np

# 배열 2개 생성
ar = np.arange(4)
ar1 = np.arange(4, 9)

# concatenate 사용시 tuple 의 형태로 전달
arR = np.concatenate((ar, ar1))
print(arR)
print("result : [0 1 2 3 4 5 6 7] ")
print()

# 1 차원 배열은 axis 가 없다
print("1 차원 배열은 axis 가 없다")
errorR = np.concatenate((ar, ar1), axis=1)
print(errorR)
```

```
Traceback (most recent call last):
  File "/Users/mac/PycharmProjects/pythonProject/class/0209/_15_ArrayMerge.py", line 15, in <module>
    errorR = np.concatenate((ar, ar1), axis=1)
  File "<__array_function__ internals>", line 180, in concatenate
numpy.AxisError: axis 1 is out of bounds for array of dimension 1
[0 1 2 3 4 5 6 7]
result : [0 1 2 3 4 5 6 7]
```

```python
# 2차원 배열을 생성
m1 = ar.reshape(2, 2)
m2 = ar1.reshape(2, 2)

# 2차원배열 합치기
print("2차원배열 합치기 -----")
mr = np.concatenate((m1, m2))
print(mr)
print("axis 적용시켜서 방향에 따라 합침 ")
mr = np.concatenate((m1, m2), axis=1)
print(mr)
print()
```

$m_1 = \begin{matrix} 0 & 1 \\ 2 & 3 \end{matrix}$

$m_2 = \begin{matrix} 4 & 5 \\ 6 & 7 \end{matrix}$

$axis:X = \begin{matrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{matrix}$

$axis:0 = \begin{matrix} 0 & 1 & 4 & 5 \\ 2 & 3 & 6 & 7 \end{matrix}$

```
2차원배열 합치기 -----
[[0 1]
 [2 3]
 [4 5]
 [6 7]]
axis 적용시켜서 방향에 따라 합침
[[0 1 4 5]
 [2 3 6 7]]
```

hstack
: concatenate 에서 axis = 1을 적용한 것과 동일한 효과

horizon

vstack
: concatenate 에서 axis를 생략한 것과 동일한 효과

Vertical

```
# hstack , vstack - concatenate 에서
# axis 를 1을 설정하느냐 생략하느냐의 차이
```

```python
import numpy as np

ar = np.arange(4)
br = np.arange(4, 17, 4)
print(f'ar : {ar}')
print(f'br : {br}')
print()

m1 = ar.reshape(2, 2)
m2 = br.reshape(2, 2)

# Vertical Stack
print("Vertical Stack---")
print(np.vstack((m1, m2)))
print()
print("Horizon ------------------------------------")
print("Horizon Stack ----")
print(np.hstack((m1, m2)))
print()
```

```
ar : [0 1 2 3]
br : [ 4  8 12 16]
Vertical Stack---
[[ 0  1]
 [ 2  3]
 [ 4  8]
 [12 16]]

Horizon ------------------------------------
Horizon Stack ----
[[ 0  1  4  8]
 [ 2  3 12 16]]
```

dstack
: 2개의 동일한 위치의 데이터끼리 묶어서 결합하는 함수

```python
import numpy as np

ar = np.arange(4)
br = np.arange(4, 17, 4)
print(f'ar : {ar}')
print(f'br : {br}')
print()

m1 = ar.reshape(2, 2)
m2 = br.reshape(2, 2)

print(f"Dstack : {np.dstack((m1, m2))}")
```

```
ar : [0 1 2 3]
br : [ 4  8 12 16]

Dstack : [[[ 0  4]
  [ 1  8]]

 [[ 2 12]
  [ 3 16]]]
```

stack

: dstack 과 유사

r_ ——————————— Row

: hstack 함수처럼 사용하지 않고 [ ] 안에 인덱스를 설정 하듯이 배열을 나열

c_ ——————————— Column

: vstack 함수처럼 사용하지 않고 [ ] 안에 인덱스를 설정 하듯이 배열을 나열

Index 형태로 설정했는데 어떤 작업을 수행하도록 해주는 것을 Indexer 라 한다.

C# 안에서 [ ] 안에 이름 작성

—〉

이름에 해당하는 Property 가 호출되는데 이것을 Indexer 라고 한다 .

```python
# r_, c_ : indexer ---> 기능은 hstack , vstack 과 유사
import numpy as np

ar = np.arange(4)
br = np.arange(4, 17, 4)
print(f'ar : {ar}')
print(f'br : {br}')
print()

m1 = ar.reshape(2, 2)
m2 = br.reshape(2, 2)


print(f'hstack : {np.hstack((m1, m2))}')
print("============")
print(f'np.c_ : {np.c_[m1, m2]}')
```

```
ar : [0 1 2 3]
br : [ 4  8 12 16]

hstack : [[ 0  1  4  8]
 [ 2  3 12 16]]
============
np.c_ : [[ 0  1  4  8]
 [ 2  3 12 16]]
```

title

: 하나의 배열을 여러 번 반복해서 결합

배열 과 반복 횟수를 parameter로 대입

데이터 분석 과정

기획

—〉 데이터 수집

—〉 데이터 탐색

—〉 데이터 전처리

—〉 모델 생성 (분석)

—〉검증

—〉 배포 (보고서 작성)

수집 ~ 검증

까지는 나선형태로 여러번 반복

각각의 과정은 오랜 시간이 걸리는 경우가 많고
특히 전처리는 1가지 작업이 아니고 여러가지 작업인 경우가 많다

부분적인 작업이 완료되면
그때 까지 사용한 데이터를 메모리가 아닌
다른곳에 저장해두는 것이 중요

——〉 어떤 작업이 잘못 되었을 때 부분 완료된 데이터를 가지고
작업을 이어나갈 수 있다 .

memory 에 저장해두는 것
kernel restart 하면 모두 소멸

numpy 배열 저장과 읽어 오기

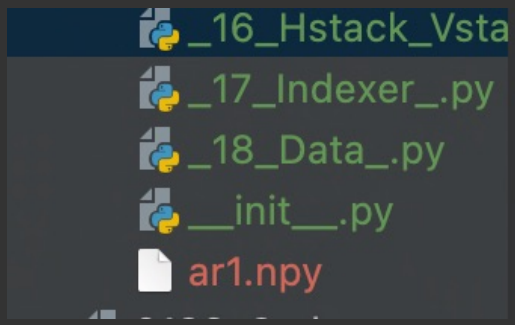
1. 저장
numpy. save('파일 경로' , 배열)

2. 확장자
: npy를 많이 사용한다 .

3 . 읽어오기
: numpy.load(' 파일 경로 ')

4. 여러개의 데이터를 저장하고 싶을때
: numpy. savez('파일경로 ' , 이름 = 데이터, 이름 = 데이터 …)
—> 배열의 배열로 저장해둔다
이때 이름이 배열의 Index 가 된다.

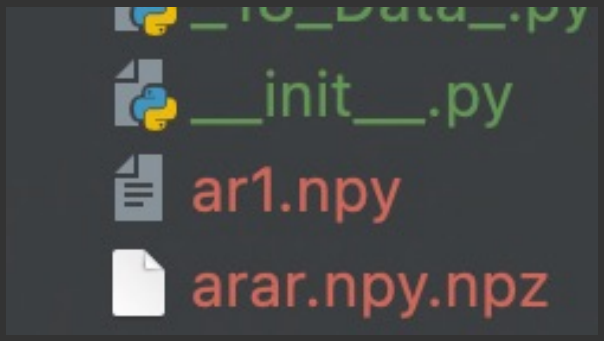
```python
# 데이터 저장과 일기 --> 중요!! , 확장자 확인

import numpy as np

# 배열을 생성
ar = np.arange(1, 5)
br = np.arange(4, 17, 4)
# 배열 확인
print(f'ar = {ar}')
print(f'br = {br}')
print()

# 하나의 데이터 저장 시
print("하나의 데이터 저장 --------- ")
np.save('./ar1.npy', ar)
```




```
ar = [1 2 3 4]
br = [ 4  8 12 16]

하나의 데이터 저장 ---------
```

```python
# 읽어오기
print("저장된 npy 데이터 읽어오기 ")
xr = np.load('./ar1.npy')
print(xr)
print()

# 데이터 여러개 저장하기
print("데이터 여러개 저장하기=====")
np.savez('./arar.npy', a=ar, b=br)
print()

# 배열을 여러개 저장한 파일 읽어오기
print("배열을 여러개 저장한 파일 읽어오기 ----- 1")
xr = np.load('./arar.npy.npz')
# 여러개 저장시 압축되어 있어서 데이터를 바로 읽을 수가 없음
print()

print("배열을 여러개 저장한 파일 읽어오기 ----- 2")
# 여러 개를 저장한 경우에는 이름이 인덱스가 된다.
print(xr['b'])
```

```
저장된 npy 데이터 읽어오기
[1 2 3 4]

데이터 여러개 저장하기=====

배열을 여러개 저장한 파일 읽어오기 ----- 1

배열을 여러개 저장한 파일 읽어오기 ----- 2
[ 4  8 12 16]
```



** Pandas

pandas
: Series 와 DataFrame 이라는 자료구조를 제공해주는 Package

1. Series
: 하나의 열에 해당하는 자료구조

2. DataFrame
: Series 를 옆으로 연결한 테이블 구조의 자료

특징
1. Series 는 **모든 요소의 Data Type 이 일치**

2. DataFrame 은 각 **Series(열)의 자료형이 달라도 된다.**

3. Numpy 에 비해서 **다양한 소스로부터 데이터를 읽을 수 있는 방법을 제공**
(데이터 수집 시에는 Numpy 를 사용하기 어렵다 !!)
——〉**그래서 Data 수집에는 Pandas 를 많이 사용**

4. 데이터 전처리에 관련된 함수도 많이 제공

5. 시각화 기능도 일부분 제공
: 탐색적 분석에서 pandas 만을 사용하더라도 가능하다

설치
기본 python package 가 아님으로 설치를 해야한다.
(anaconda 에는 내장되어 있다)

python -m pip list : 이걸로 확인!!

```
(venv) (base) mac@SungMak pythonProject % pip install pandas
Collecting pandas
  Downloading pandas-1.4.0-cp39-cp39-macosx_10_9_x86_64.whl (11.5 MB)
     |████████████████████████████████| 11.5 MB 537 kB/s
Requirement already satisfied: numpy>=1.18.5 in ./venv/lib/python3.9/site-packages (from pandas) (1.22.1)
Collecting pytz>=2020.1
  Downloading pytz-2021.3-py2.py3-none-any.whl (503 kB)
     |████████████████████████████████| 503 kB 50.8 MB/s
Requirement already satisfied: python-dateutil>=2.8.1 in ./venv/lib/python3.9/site-packages (from pandas) (2.8.2)
Requirement already satisfied: six>=1.5 in ./venv/lib/python3.9/site-packages (from python-dateutil>=2.8.1->pandas) (1.16.0)
Installing collected packages: pytz, pandas
Successfully installed pandas-1.4.0 pytz-2021.3
WARNING: You are using pip version 21.1.2; however, version 22.0.3 is available.
You should consider upgrading via the '/Users/mac/PycharmProjects/pythonProject/venv/bin/python -m pip install --upgrade pip' command.
(venv) (base) mac@SungMak pythonProject %
```