EURON 1주차 과제

Numpy (Numerical Python)

- 빠른 배열 연산 속도
- 높은 호환성(C/C++)
- Pandas보다 편리성이 떨어진다는 한계

넘파이 ndarray 개요

```
import numpy as np
```

- as np 추가해 약어로 모듈 표현
- 기반 데이터 타입 ndarray → 다차원 배열을 쉽게 생성하고 다양한 연산 수행 가능

```
array1 = np.array([1, 2, 3])
print('array1 type:', type(array1))
print('array2 array 형태:', array1.shape)

array2 = np.array([[1,2,3,],
                   [2,3,4,]])
print('array2 type:', type(array2))
print('array2 array 형태:', array2.shape)

array3 = np.array([[1,2,3,]])
print('array3 type:', type(array3))
print('array3 array 형태:', array3.shape)
```

<Output>

```
array1 type: <class 'numpy.ndarray'>
array2 array 형태: (3,)
array2 type: <class 'numpy.ndarray'>
array2 array 형태: (2, 3)
array3 type: <class 'numpy.ndarray'>
array3 array 형태: (1, 3)
```

- array1은 1차원 array, 3개의 데이터
- array2는 2차원 array, 2 row and 3 column
- array3는 2차원 array, 1 row and 3 column

→ array1과 array3는 같은 데이터값은 같으나 차원이 다름

```
print('array1: {0}차원, array2: {1}차원, array3: {2}차원'.format(array1.ndim, array2.ndim, array3.ndim))
```

<Output>

```
array1: 1차원, array2: 2차원, array3: 2차원
```

- array()의 함수의 인자로 파이썬의 리스트 객체 주로 사용
- 리스트 []는 1차원, 리스트의 리스트 [[]]는 2차원 으로 나타낼 수 있음

ndarray의 데이터 타입

- ndarray내의 데이터값은 숫자, 문자열, 불 값 모두 가능
- 숫자형의 경우 int형, unsigned int형, float형, complex형 모두 가능
- 그러나 한 ndarray 객체 내에서는 같은 타입의 데이터 타입만 가능 (ex. int와 float는 함께 있을 수 없음)
- 데이터 타입 dtype 속성으로 확인 가능

```
list1 = [1, 2, 3]
print(type(list1))
array1 = np.array(list1)
print(type(array1))
print(array1, array1.dtype)
```

<Output>

```
<class 'list'>
<class 'numpy.ndarray'>
[1 2 3] int32
```

- list1은 리스트 자료형이며 숫자 1, 2, 3을 값으로 가짐
- ndarray로 쉽게 변경 가능, 데이터값은 int32형

만약 다른 데이터 유형이 섞여있는 리스트를 ndarray로 변경하면 데이터 크기가 더 큰 데이터 타입으로 형 변형을 일괄 적용

```
list2 = [1, 2, 'test']
array2 = np.array(list2)
print(array2, array2.dtype)

list3 = [1, 2, 3.0]
```

```
array3 = np.array(list3)
print(array3, array3.dtype)
```

<Output>

```
['1' '2' 'test'] <U11
[1. 2. 3.] float64
```

- list2에서 숫자형 값 1, 2가 문자열 값으로 변환
- list3에서 int 1, 2가 float64형으로 변환

ndarray 내 데이터값의 타입 변경은 astype() 메서드를 통해 가능. 메모리를 더 절약해야할 때 보통 이용

```
array_int = np.array([1, 2, 3])
array_float = array_int.astype('float64')
print(array_float, array_float.dtype)

array_int1=array_float.astype('int32')
print(array_int1, array_int1.dtype)

array_float1 = np.array([1.1, 2.1, 3.1])
array_int2 = array_float1.astype('int32')
print(array_int2, array_int.dtype)
```

<Output>

```
[1. 2. 3.] float64
[1 2 3] int32
[1 2 3] int32
```

- int32형 데이터를 float64로 변환
- 다시 float64를 int32로 변경 (float를 int형으로 변경할 때 소수점 이하는 모두 없어짐)

ndarray를 편리하게 생성하기 - arrange, zeors, ones

arrange()

- range()와 유사한 기능
- array를 range()로 표현
- 0부터 함수 인자 값 -1까지의 값을 순차적으로 ndarray의 데이터값으로 변환

```
sequence_array = np.arange(10)
print(sequence_array)
print(sequence_array.dtype, sequence_array.shape)
```

<Output>

```
[0 1 2 3 4 5 6 7 8 9]
int32 (10,)
```

- default 함수 인자는 stop 값
- 0부터 stop 값인 10에서 -1을 더한 9까지의 연속 숫자 값으로 구성된 1차원 ndarray를 만들어줌
- range와 유사하게 start 값도 부여해 0이 아닌 다른 값부터 시작한 연속 값을 부여할 수 있음

zeros()

- 함수 인자로 튜플 형태의 shape 값을 입력하면 모든 값을 0으로 채운 해당 shape를 가진 ndarray를 반환함

```
zero_array = np.zeros((3,2),dtype='int32')
print(zero_array)
print(zero_array.dtype, zero_array.shape)
```

<Output>

```
[[0 0]
 [0 0]
 [0 0]]
int32 (3, 2)
```

ones()

- 함수 인자로 튜플 형태의 shape 값을 입력하면 모든 값을 1로 채운 해당 shape를 가진 ndarray를 반환함
- 함수 인자로 dtype을 정해주지 않으면 default로 float64 형의 데이터로 ndarray를 채움

```
one_array = np.ones((3,2))
print(one_array)
print(one_array.dtype, one_array.shape)
```

<Output>

```
[[1. 1.]
 [1. 1.]
 [1. 1.]]
float64 (3, 2)
```

ndarray의 차원과 크기를 변경하는 reshape()

- reshape() 메서드는 ndarray를 특정 차원 및 크기로 변환
- 변환을 원하는 크기를 함수 인자로 부여

```
array1 = np.arange(10)
print('array1:\n', array1)

array2 = array1.reshape(2,5)
print('array2:\n', array2)

array3 = array1.reshape(5,2)
print('array3:\n', array3)
```

<Output>

```
array1:
 [0 1 2 3 4 5 6 7 8 9]
array2:
 [[0 1 2 3 4]
 [5 6 7 8 9]]
array3:
 [[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
```

- reshape()는 지정된 사이즈로 변경이 불가능하면 오류 발생

```
array1.reshape(4, 3)
```

<Output>

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[18], line 1
----> 1 array1.reshape(4, 3)

ValueError: cannot reshape array of size 10 into shape (4,3)
```

- 인자로 -1 적용 시 원래 ndarray와 호환되는 새로운 shape로 변환해줌

```
array1 = np.arange(10)
print(array1)
array2 = array1.reshape(-1, 5)
print('array2 shape:', array2.shape)
array3 = array1.reshape(5, -1)
print('array3 shape:', array3.shape)
```

<Output>

```
[0 1 2 3 4 5 6 7 8 9]
array2 shape: (2, 5)
array3 shape: (5, 2)
```

- array1은 1차원 ndarray로 0~9까지의 데이터, 총 10개의 데이터를 가짐
- 이를 5개의 column으로 나타내려면 row는 2개가 되어야 함 → array2의 shape
- 5개의 row로 나타내려면 column은 2개가 되어야 함 → array3의 shape

```
array1 = np.arange(10)
array4 = array1.reshape(-1,4)
```

<Output>

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[21], line 2
      1 array1 = np.arange(10)
----> 2 array4 = array1.reshape(-1,4)

ValueError: cannot reshape array of size 10 into shape (4)
```

- 그러나 이 경우, 10개의 데이터값은 4개의 column을 가진 row로 변경할 수 없기 때문에 에러가 발생함

주로 reshape() 에서 -1 인자는 reshape(-1, 1)와 같은 형태로 자주 사용

- 원본 ndarray가 어떤 형태라도 2차원이고, 여러 개의 row를 가지되 반드시 1개의 column을 가진 ndarray로 변환됨을 보장
- 여러개의 넘파이 ndarray를 stack이나 concat으로 결합할 때 각각의 ndarray의 형태를 통일하는데 사용될 수 있음

```
array1 = np.arange(8)
array3d = array1.reshape((2,2,2))
print('array3d:\n', array3d.tolist())

#3차원 ndarray를 2차원 ndarray로 변환
array5 = array3d.reshape(-1,1)
print('array5:\n', array5.tolist())

#1차원 ndarray를 2차원 ndarray로 변환
array6 = array1.reshape(-1,1)
print('array6:\n', array6.tolist())
print('array6 shape:', array6.shape)
```

<Output>

```
array3d:
 [[[0, 1], [2, 3]], [[4, 5], [6, 7]]]
array5:
 [[0], [1], [2], [3], [4], [5], [6], [7]]
array6:
 [[0], [1], [2], [3], [4], [5], [6], [7]]
array6 shape: (8, 1)
```

- reshape(-1,1)을 이용해 3차원을 2차원으로, 1차원을 2차원으로 변경
- ndarray는 tolist() 메서드를 이용해 리스트 자료형으로 변환 가능

넘파이의 ndarray의 데이터 세트 선택하기 - 인덱싱(Indexing)

1. 특정한 데이터만 추출: 원하는 위치의 인덱스 값을 지정하면 해당 위치의 데이터가 반환됩니다.
2. 슬라이싱(Slicing): 슬라이싱은 연속된 인덱스상의 ndarray를 추출하는 방식입니다. ‘:’ 기호 사이에 시작 인덱스와 종료 인덱스를 표시하면 시작 인덱스에서 종료 인덱스-1 위치에 있는 데이터의 ndarray를 반환합니다. 예를 들어 1:5라고 하면 시작 인덱스 1과 종료 인덱스 4까지에 해당하는 ndarray를 반환합니다.
3. 팬시 인덱싱(Fancy Indexing): 일정한 인덱싱 집합을 리스트 또는 ndarray 형태로 지정해 해당 위치에 있는 데이터의 ndarray를 반환합니다.
4. 불린 인덱싱(Boolean Indexing): 특정 조건에 해당하는지 여부인 Ture/False 값 인덱싱 집합을 기반으로 True에 해당하는 인덱스 위치에 있는 데이터의 ndarray를 반환합니다.

단일 값 추출

```
#1부터 9까지의 1차원 ndarray 생성
array1 = np.arange(start=1, stop=10)
print('array1:', array1)
#index는 0부터 시작하므로 array1[2]는 3번째 index 위치의 데이터값을 의미
value = array1[2]
print('value:', value)
print(type(value))
```

<Output>

```
array1: [1 2 3 4 5 6 7 8 9]
value: 3
<class 'numpy.int32'>
```

- 인덱스는 0부터 시작하므로 array1[2]는 3번째 인덱스 위치의 데이터값 → 3
- array1[2]의 타입은 ndarray 타입이 아닌 데이터값

인덱스에 마이너스 기호 이용 시 맨 뒤에서부터 데이터값 추출

```
print('맨 뒤의 값', array1[-1], '맨 뒤에서 두번째 값:', array1[-2])
```

<Output>

```
맨 뒤의 값 9 맨 뒤에서 두번째 값: 8
```

단일 인덱스 이용해 ndarray 내의 데이터값 수정 시

```
array1[0] = 9
array1[8] = 0
print('array1:', array1)
```

<Output>

```
array1: [9 2 3 4 5 6 7 8 0]
```

다차원 ndarray에서 단일 값 추출

- 1차원과의 차이로 2차원의 경우, 콤마(,)로 분리된 로우와 칼럼 위치의 인덱스를 통해 접근

```
array1d = np.arange(start=1, stop=10)
array2d = array1d.reshape(3,3)
print(array2d)

print('(row=0, col=0) index 가리키는 값:', array2d[0,0])
print('(row=0, col=1) index 가리키는 값:', array2d[0,1])
print('(row=1, col=0) index 가리키는 값:', array2d[1,0])
print('(row=2, col=2) index 가리키는 값:', array2d[2,2])
```

<Output>

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
(row=0, col=0) index 가리키는 값: 1
(row=0, col=1) index 가리키는 값: 2
(row=1, col=0) index 가리키는 값: 4
(row=2, col=2) index 가리키는 값: 9
```

- 실제 넘파이에서는 row와 column을 사용하지 않음

- row = axis 0, column = axis 1
- ex) [row=0, col=1] 은 다음이 옳은 표현임 [axis 0=0, axis 1=1]
- 3차원의 경우 axis 0, axis 1, axis 2로 구분

슬라이싱

- ':' 기호를 이용해 연속한 데이터 슬라이싱해 추출
- 단일 데이터값 추출의 데이터 타입 : 데이터 값 / 슬라이싱, 불린, 팬시 인덱싱의 데이터 타입 : ndarray
- ':' 사이에 시작 인덱스와 종료 인덱스 표시하면 시작 인덱스에서 종료 인덱스-1 위치에 있는 데이터의 ndarray 변환

```
array1 = np.arange(start=1, stop=10)
array3 = array1[0:3]
print(array3)
print(type(array3))
```

<Output>

```
[1 2 3]
<class 'numpy.ndarray'>
```

슬라이싱 기호인 ':' 사이의 시작, 종료 인덱스는 생략 가능

1. ':' 기호 앞에 시작 인덱스 생략시 맨 처음 인덱스인 0으로 간주
2. ':' 기호 뒤에 종료 인덱스 생략시 맨 마지막 인덱스로 간주
3. ':' 기호 앞/뒤에 시작/종료 인덱스 생락시 맨 처음/마지막 인덱스로 간주

```
array1 = np.arange(start=1, stop=10)
array4 = array1[:3]
print(array4)

array5 = array1[3:]
print(array5)

array6 = array1[:]
print(array6)
```

<Output>

```
[1 2 3]
[4 5 6 7 8 9]
[1 2 3 4 5 6 7 8 9]
```

2차원 ndarray의 경우 콤마(,)로 로우와 칼럼 인덱스를 지칭

```
array1d = np.arange(start=1, stop=10)
array2d = array1d.reshape(3,3)
print('array2d:\n', array2d)

print('array2d[0:2, 0:2]\n', array2d[0:2, 0:2])
print('array2d[1:3, 0:3]\n', array2d[1:3, 0:3])
print('array2d[1:3, :]\n', array2d[1:3, :])
print('array2d[:, :]\n', array2d[:, :])
print('array2d[:2, 1:]\n', array2d[:2, 1:])
print('array2d[:2, 0]\n', array2d[:2, 0])
```

<Output>

```
array2d:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
array2d[0:2, 0:2]
 [[1 2]
 [4 5]]
array2d[1:3, 0:3]
 [[4 5 6]
 [7 8 9]]
array2d[1:3, :]
 [[4 5 6]
 [7 8 9]]
array2d[:, :]
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
array2d[:2, 1:]
 [[2 3]
 [5 6]]
array2d[:2, 0]
 [1 4]
```

팬시 인덱싱

팬시 인덱싱(Fancy Indexing)은 리스트나 ndarray로 인덱스 집합을 지정하면 해당 위치의 인덱스에 해당하는 ndarray를 반환하는 인덱싱 방식

```
array1d = np.arange(start=1, stop=10)
array2d = array1d.reshape(3, 3)

array3 = array2d[[0,1], 2]
print('array2d[[0, 1], 2] =>',array3.tolist())

array4 = array2d[[0, 1], 0:2]
print('array2d[[0, 1], 0:2]=>', array4.tolist())

array5 = array2d[[0, 1]]
print('array2d[[0,1]]=>', array5.tolist())
```

<Output>

```
array2d[[0, 1], 2] => [3, 6]
array2d[[0, 1], 0:2]=> [[1, 2], [4, 5]]
array2d[[0,1]]=> [[1, 2, 3], [4, 5, 6]]
```

불린 인덱싱

- 불린 인덱싱(Boolean indexing)은 조건 필터링과 검색을 동시에 할 수 있기 때문에 매우 자주 사용되는 인덱싱 방식
- 불린 인덱싱 이용 시 for loops/if else 문보다 훨씬 간단하게 이를 구현할 수 있음

```
array1d = np.arange(start=1, stop=10)
#[]안에 array1d > 5 Boolean indexing을 적용
array3 = array1d[array1d > 5]
print('array1d > 5 불린 인덱싱 결과 값 :', array3)
```

<Output>

```
array1d > 5 불린 인덱싱 결과 값 : [6 7 8 9]
```

위와 동일한 불린 ndarray를 만들고 이를 array1d[]내에 인덱스로 입력하면 동일한 데이터 세트가 반환됨

```
boolean_indexs = np.array([False, False, False, False, False, True, True, True, True ])
array3 = array1d[boolean_indexs]
print('불린 인덱스로 필터링 결과 :', array3)
```

<Output>

```
불린 인덱스로 필터링 결과 : [6 7 8 9]
```

다음과 같이 직접 인덱스 집합을 만들어 대입한 것과 동일

```
indexs = np.array([5, 6, 7, 8])
array4 = array1d[indexs]
print('일반 인덱스로 필터링 결과 :', array4)
```

<Output>

```
일반 인덱스로 필터링 결과 : [6 7 8 9]
```