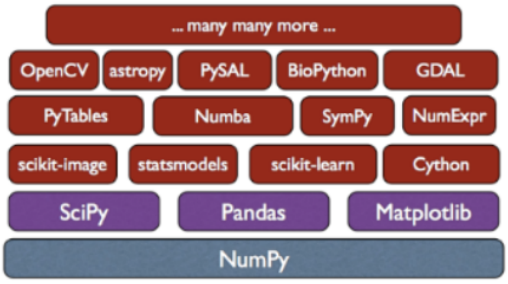
[Numpy 개요]

1. 과학계산 / 데이터분석 파이썬 필수 라이브러리

- NumPy : 대수, 행렬, 통계금융 등 수학 및 과학 계산, 다차원 배열 객체 ndarray (https://numpy.org/)
- Scipy : 과학(계산)수치적분 루틴과 미분방정식 해석기, 방정식근 알고리즘, 표준 연속/이산 확률분포와 다양한 통계 (https://www.scipy.org/)
- Pandas : 구조화된 데이터 처리, DataFrame 객체는 2차원표 또는 행과 열을 나타내는 자료구조 (https://pandas.pydata.org/)
- Matplotlib : 데이터 시각화 라이브러리 (https://matplotlib.org/)



In [ ]:
```
import numpy as np
np.__version__
```

http://taewan.kim/post/numpy_cheat_sheet/

2. Numpy 학습 내용

1. 배열 생성 : 파이썬 시퀀스데이터, 범위지정, 랜덤값, 시간배열생성
2. 배열 연산 : 브로드캐스팅, 산술연산, 비교연산, 집계함수(Aggregate Functions)
3. 배열 조회 : 배열인덱싱(Indexing) / 배열 슬라이싱(Slicing)
4. 배열 값 삽입/수정/삭제/복사
5. 배열 변환 : 전치/요소추가 삭제/분할/변형/결합
6. 함수, 입출력

3. NumPy (Numerical Python) ?

1) 다차원 행렬 자료구조 ndarray 지원 (선형대수 벡터와 행렬 계산에 주로 사용)

- 과학 연산을 위한 파이썬 핵심 라이브러리
- NumPy ("넘파이")는 2005년에 Travis Oliphant가 발표한 수치해석용 Python 패키지
- **다차원의 행렬 자료구조인 ndarray 를 지원하여 벡터와 행렬을 사용하는 선형대수 계산에 주로 사용.**
- NumPy의 행렬 연산은 C로 구현된 내부 반복문을 사용하기 때문에 Python 반복문에 비해 속도가 빠르다.
- NumPy는 Pandas, Scikit-learn, Tensorflow등 데이터 사이언스 분야에서 사용되는 라이브러리들의 토대가 되는 라이브러리.

2) Numpy 필요성

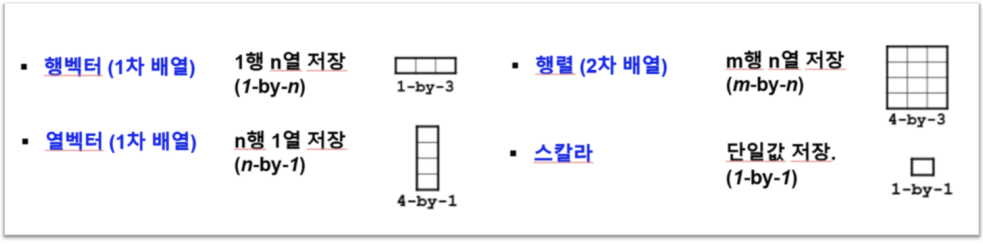
- 크기가 같은 두 개의 숫자형 배열에 대해서 산술 연산(arithmetic operations)을 할 때 'for loops'를 사용하지 않고,
- NumPy ndarray를 사용하면 Vectorization으로 매우 빠르게 batch 연산을 수행할 수 있음.
- 벡터,행렬 연산 가능

3) Python list vs. Numpy ndarray 차이점

- **Python list**
  - 여러가지 타입의 원소
  - linked List 구현
  - 메모리 용량이 크고 속도가 느림
  - 벡터화 연산 불가
- **Numpy ndarray**
  - 동일 타입의 원소
  - contiguous memory layout
  - 메모리 최적화, 계산 속도 향상
  - 벡터화 연산 가능

4) 용어 비교 (Array , Tensor)

- 스칼라 (scalar)
- 배열 (array)
  - 벡터(Vector : 1차원 배열)
  - 행렬(Matrix : 2차원 배열)
- 텐서 (Tensor): 텐서는 임의의 차원을 갖고 있는 배열
  - 0차원(스칼라), 1차원(벡터), 2차원(행렬), N차원 고차원의 모든 행렬 포함
- 행(row)과 열(column) (행 x 열)



[NumPy 기초]

- 요소로 이루어진 테이블이라고 볼 수 있으며, 모두 같은 타입이다.
- 넘파이의 배열 클래스는 ndarray라 함.
- 흔히 array(배열)이라고 부른다

- 형상 확인(dim, shape, size, dtype, data)
- 형상 조작(ravel, transpose, reshape, resize ...)

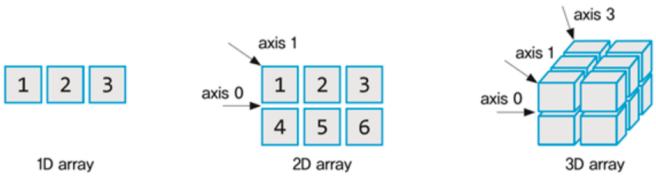
1. ndarray class

1) ndarray (N-Dimension Array type (다차원 배열))

- NumPy의 ndarray는 파이썬에서 사용할 수 있는 배열 구조이다.
- **ndarray 클래스는 NumPy 패키지의 핵심 (numpy.ndarray)**
- ndarray는 같은 종류의 데이터를 담을 수 있는 다차원 배열이며, 모든 원소는 같은 자료형 이어야 한다.
- 모든 배열은 각 차원의 크기를 알려주는 shape라는 튜플과 배열에 저장된 자료형을 알려주는 dtype이라는 객체를 가지고 있다.

2) ndarray는 list나 tuple 같은 시퀀스 자료형으로 부터 생성한다.

- numpy(넘파이) 배열 (n-차원 배열로 ndarrays로 종종 쓰여짐)
- 배열에 쉽게 요소별 연산(elementwise operations)과 행렬대수(matrix algebra)작업을 수행할 수 있다.
- 배열은 n-차원이 될 수 있다.



3) ndarray 객체

- 장점

- 1차 배열은 행벡터나 열벡터 둘 다로 취급할 수 있다
- **`dot(A,v): v는 열벡터, dot(v,A): v는 행벡터로 취급`, 따라서 전치를 복잡하게 수행할 필요가 없다.**
- 요소 간 곱셈이 쉽다 (예: *AB) 사실 모든 산술 연산 (+ - / ** 등등)*이 요소 간 연산
- ndarray는 numpy 의 가장 기본 객체
- 다차원 배열을 쉽게 구현

- 단점

- 행렬간 곱에 obj.dot() 멤버함수를 사용해야 하므로 복잡할 수 있다.
- 세 행렬 A, B, C의 행렬곱은 dot(dot(A,B),C)

4) List vs. Array (데이터 타입(클래스)에 따른 연산, 메소드가 다름)

In [ ]:
```
import numpy as np
ar = np.array([1, 2, 3, 4, 5])

print(ar)
print(type(ar)) # 데이터 타입
```

In [ ]:
```
dir(ar) # 관련 메소드
```

In [ ]:
```
[1,2,3] + [4,5,6]  # 리스트 합
```

In [ ]:
```
np.array([1, 2, 3]) + np.array([4, 5, 6]) # Array 합
```

In [ ]:
```
xl = [1, 2, 3, 4, 5] # 리스트 생성
yl = [6, 7, 8, 9, 10]
```

In [ ]:
```
xl + yl # 리스트 합치기
```

In [ ]:
```
# xl - ylx # 에러
# xl * xl  # 에러
```

In [ ]:
```
xl * 2 # 리스트 중복
```

In [ ]:
```
# xl = [1, 2, 3, 4, 5]  --> [2, 4, 6, 8, 10] 구현 방법 ?

# 산술 연산은 for 문 사용
# arr1*2

ax = []
for i in xl:
    ax.append(2 * i)
ax
```

In [ ]:
```
# numpy 첫번째
import numpy as np

arr1 = np.array(xl) # array 생성
arr1
```

In [ ]:
```
arr2 = np.array(yl) # array 생성
arr2
```

In [ ]:
```
type(xl), type(arr2) # 데이터 타입
```

In [ ]:
```
dir(xl) # 메소드 확인
```

In [ ]:
```
dir(arr2) # 메소드 확인
```

In [ ]:
```
arr1 + arr2 # 배열 합
```

In [ ]:
```
# list for 문 사용없이 x2 구현
arr1 *2 # 배열 곱
```

In [ ]:
```
print(arr1*2)
```

5) 배열 상태 검사 (Inspecting)

- NumPy는 배열의 상태 검사를 위한 방법 제공
- ndarray의 Attribute (속성)
  - arr. shape : 배열의 형상 정보(m x n)
  - arr. ndim : 배열 차원, 축의 개수(Dimension)
  - arr. size : 전체 요소(Element)의 개수
  - arr. dtype : 각 요소(Element)의 타입
  - arr. itemsize : 각 요소(Element)의 타입의 bytes 크기

In [ ]:
```
import numpy as np

# 데모 배열 객체 생성
arr = np.random.random((5,2,3))

# 배열 타입 조회
print(type(arr))

# 배열의 shape 확인
print(arr.shape)

# 배열의 길이
print(len(arr))

# 배열의 차원 수
print(arr.ndim)

# 배열의 요소 수: shape(k, m, n) ==> k*m*n
print(arr.size)

# 배열 타입명 확인
print(arr.dtype)

# 배열 타입명
print(arr.dtype.name)

# 배열 요소를 int로 변환
# 요소의 실제 값이 변환되는 것이 아님
# View의 출력 타입과 연산을 변환하는 것
print(arr.astype(np.int))

# np.float으로 타입을 다시 변환하면 np.int 변환 이전 값으로 모든 원소 값이 복원됨
print(arr.astype(np.float))
```

In [ ]:
```
import numpy as np
a = np.array([[1,2],[3,4],[5,6]]) # 파이썬 리스트를 이용하여 ndarray 객체생성
print(a)
print(a.shape)              # (3,2)
print(a.ndim)               # 2
print(a.size)               # 6
print(a.dtype)              # int64
print(type(a))              # <class 'numpy.ndarray'>
```

In [ ]:
```
a31 = np.array([1,2,3]) # 1차원배열

print(a31)
print(a31.shape)
```

In [ ]:
```
a13 = np.array([[1,2,3]]) # 행벡터(1X3):2차원배열

print(a13)
print(a13.shape)
```

In [ ]:
```
a = np.array([[1],[2],[3]]) # 열벡터(3X1):2차원배열

print(a)
print(a.shape)
```

In [ ]:
```
import numpy as np
a = np.arange(15).reshape(3, 5)

print(a)
print('a.shape:', a.shape)              # 어레이의 shape
print('a.ndim:', a.ndim)                # 어레이의 차원, 축의 개수
print('a.dtype.name:', a.dtype.name)    # 어레이 요소의 타입
print('a.itemsize:', a.itemsize)        # 어레이 요소의 바이트 크기
print('a.size:', a.size)                # 어레이 요소의 총 개수
print('type(a):', type(a))              # 어레이의 타입
```

In [ ]:
```
#  Array 정보 확인을 위한 함수 만들기 pprint()
def pprint(arr):
    print("type:{}".format(type(arr)))
    print("shape: {}, dimension: {}, dtype:{}".format(arr.shape, arr.ndim, a
    print("Array's Data:\n", arr)
```

In [ ]:
```
ar = np.arange(15).reshape(3, 5)
pprint(ar)
```

2. Array 생성하기

- 1차원 배열 : 벡터 / 2차원배열: 행렬 / 3차원배열 이상 : 다차원 배열
- Numpy ndarray는 **동일한 데이터 타입만을 가질 수 있음**
- 파이썬 시퀀스 자료형 을 넘파이 배열로 변환 : np.array( ) 사용
- 넘파이 배열 생성 함수 사용 : np.arrange( ), np.ones( ) 등등
- `np.array( ), np.arange( ), np.linspace( ), np.zeros( ), np.ones( ), np.empty( )`

In [ ]:
```
# 리스트 생성
n1 = [1,2,3,4]
n2 = [5,6,7,8]
n21 = [n1,n2]
n21
```

In [ ]:
```
import numpy as np

data1 = [0, 1, 2, 3, 4, 5]
a1 = np.array(data1)
a1
```

In [ ]:
```
a1.dtype
```

In [ ]:
```
data2 = [0.1, 5, 4, 12, 0.5]  # 동일한 데이터타입
a2 = np.array(data2)
a2
```

In [ ]:
```
a2.dtype
```

In [ ]:
```
a = np.array([0.5, 2, 0.01, 8])  # 1 x 3 array ??
a
```

1) 시퀀스 데이터(List)로부터 배열 생성(np.array( ) 메서드 사용)

[비교] 1차배열 vs 행벡터,열벡터(2차배열) :

- 2차원 배열인 벡터 1xN, Nx1, 그리고 N크기의 1차원 배열이 모두 각각 다르다.
- 1차원 배열의 전치는 작동하지 않는다.
- 주의 : ndarray로 벡터를 표현할 때는 반드시 2차 배열을 이용해야 한다.

In [ ]:
```
# ndarray 이용 벡터(vector)를 표현할 때는 2차 배열로 정의해야 함
# 다음 세 가지는 모두 다르다, 이 중 벡터는 두 번째와 세 번째 같이 생성해야 한다.
# 1차 배열은 행벡터나 열벡터 두 가지 모두로 취급

import numpy as np
a1 = np.array([1, 2, 3])              # 크기 (3,)인 1차원 배열 -> (행벡터, 열벡터 두

a2 = np.array([[1, 2, 3]])            # 크기 (1,3)인 2차원 배열 (행벡터)
a3 = np.array([[1], [2], [3]])        # 크기 (3,1)인 2차원 배열 (열벡터)

print(a1)
print(a2)
print(a3)

print("---------------")
print(a1.shape)
print(a1.ndim)

print(a2.shape)
print(a2.ndim)

print(a3.shape)
print(a3.ndim)

# a1.T 는 동작하지 않는다. <-- 1차원 배열

# 반면 a2.T 와 a3.T는 동작한다. <-- 2차원 배열
```

1-D Array : 1차원 배열 (벡터, vector)

In [ ]:
```
# 1차원 배열
arr = np.array([0,1,2])
# arr = np.array(range(3)) # 같은 결과가 나옴

print(type(arr))
print(arr.shape)
print(arr)
```

In [ ]:
```
# 1x3 2차원 배열 (행벡터)
arr = np.array([[0,1,2]]) # 행벡터 (1X3)
print(type(arr))
print(arr.shape)
print(arr)
```

In [ ]:
```
# 3x1 2차원 배열 (열벡터)
arr = np.array([[0],[1],[2]]) # 열벡터(3x1)
print(type(arr))
print(arr.shape)
print(arr)
```

2-D array : 2차원 배열 (행렬, matrix)

In [ ]:
```
# 2x2 Array
np.array([[1,2],[3,4]])
```

In [ ]:
```
b = np.array([[0, 1, 2], [3, 4, 5]])    # 2 x 3 array
b
```

In [ ]:
```
c = np.array([[0, 1], [2, 3], [4, 5]])    # 3 x 2 array
c
```

In [ ]:
```
d = np.array([[1,2,3], [4,5,6], [7,8,9]])  # 3 x 3 array
d
```

In [ ]:
```
# 직관적 3 x 3 array
np.array([[1,2,3],
          [4,5,6],
          [7,8,9]])
```

2) 범위를 지정해 배열 생성

함수	설명	사용법
arange(start, stop, step, ..)	데이터 기준으로 step 간격으로 데이터를 생성한 후 배열을 지정	arange(0, 1, 5)
linspace(start, stop, n, ..)	요소 기준으로 n개만큼 균일하게 배열 생성	linspace(0, 1, 5)
logspace(start, stop, n, ..)	로그스케일로 n개만큼 균일하게 배열 생성	logspace(1, 10, 5)

np.arrange([start,] stop [, step]) : step 간격

- 연속적인 숫자들을 만들어내기 위해 NumPy는 파이썬의 range()와 유사한 함수인 np.arange( ) 를 제공

In [ ]:
```
np.arange(0, 10, 2)
```

In [ ]:
```
np.arange(1, 10)
```

In [ ]:
```
np.arange(5)
```

In [ ]:
```
# 4 X 3 행렬
np.arange(12).reshape(4,3)
```

In [ ]:
```
# m x n 행렬의 형태 확인
b1 = np.arange(12).reshape(4,3)
b1.shape
```

In [ ]:
```
b2 = np.arange(5)
print(b2)
b2.shape
```

In [ ]:
```
np.arange(0, 10, 5)
```

np.linspace(start, stop[, num]) : 요소 갯수

- 범위의 시작과 끝, 데이터의 개수를 지정해 배열 생성
- 부동소수점 정밀도의 제한 때문에 일반적으로 요소의 개수를 예측하기 어렵습니다.
- 증가폭 (step) 대신 요소의 개수를 인수로 입력하는 np.linspace() 함수를 사용하는 것이 좋음

In [ ]:
```
np.linspace(1, 10, 10)
```

In [ ]:
```
np.linspace(0, np.pi, 20 )
```

3) 특별한 형태의 배열 생성

함수	설명	사용법
zeros((s), ..)	지정(s) 배열 생성 후 값을 모두 0으로 초기화	zeros((3,4))
ones((s), ..)	지정(s) 배열 생성 후 값을 모두 1로 초기화	ones((3,4))
full((s), n, ..)	지정(s) 배열 생성 후 값을 모두 n으로 초기화	full((3,4))
empty((s), ..)	지정(s) 배열 생성, 초기화 없음	empty((3,4))
eye(s, ..)	지정(s) 배열 생성 후 단위행렬(대각선이 1) 생성	eye(4)
like(a, ..)	지정한 배열(a)와 같은 배열 생성 후 초기화	zeros_like(a), ...

In [ ]:
```
print(np.ones(5))
```

In [ ]:
```
print(np.ones(5).shape) # 1차원 배열
```

In [ ]:
```
print(np.ones((3,5)))
```

In [ ]:
```
print(np.ones((3,5))*5) # 전체 5
```

In [ ]:
```
np.zeros(10)
```

In [ ]:
```
np.zeros((3,4))
```

In [ ]:
```
# 단위 행렬 생성
np.eye(3)
```

4) 난수 배열의 생성

- random.randint: 균일 분포 정수 난수
- random.rand: 0~1사이 균일 분포
- random.randn: 정규 분포

함수	설명
random.rand(s)	0부터 1까지 균등하게 무작위 추출하여 s 배열 생성 ex)random.rand(3,4)
random.randint(start, end, (s), ..)	지정된 수 범위에서 정수를 균등하게 추출하여 s 배열 생성 ex) random.randint(1,10, (3,3))
random.randn(s)	정규분포로 수를 추출하여 s 배열 생성 ex)random.randn(5,5)
random.normal(M, SD, (s), ..)	평균(M), 표준편차(SD)를 가지는 정규분포에서 추출하여 배열 생성 ex)random.normal(10,1, (4,4))

In [ ]:
```
np.random.rand(2)
```

In [ ]:
```
np.random.rand(2,3)
```

In [ ]:
```
np.random.rand(2,3,4)
```

In [ ]:
```
np.random.randint(10, size=(3, 4))
```

In [ ]:
```
np.random.randint(1, 30, 3)
```

5) 배열의 데이터 타입 변환

In [ ]:
```
np.array(['1.5', '0.62', '2', '3.14', '3.141592'])
```

In [ ]:
```
str_a1 = np.array(['1.567', '0.123', '5.123', '9', '8'])
num_a1 = str_a1.astype(float)
num_a1
```

In [ ]:
```
str_a1.dtype
```

In [ ]:
```
num_a1.dtype
```

In [ ]:
```
str_a2 = np.array(['1', '3', '5', '7', '9'])
num_a2 = str_a2.astype(int)
num_a2
```

In [ ]:
```
str_a2.dtype
```

In [ ]:
```
num_a2.dtype
```

In [ ]:
```
num_f1 = np.array([10, 21, 0.549, 4.75, 5.98])
num_i1 = num_f1.astype(int)
num_i1
```

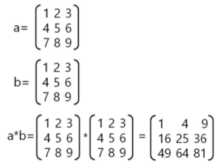
In [ ]:
```
num_f1.dtype
```

In [ ]:
```
num_i1.dtype
```

3. 배열의 연산

- NumPy의 배열 연산은 벡터화(vectorized) 연산을 사용
- 일반적으로 NumPy의 범용 함수(universal functions)를 통해 구현
- 배열 요소에 대한 반복적인 계산을 효율적으로 수행

함수	설명	함수	설명
a+b / add(a,b)	a,b를 더한다.	sqrt(a)	a의 제곱근을 구한다.
a-b / substract(a,b)	a에서 b를 감한다.	sin(a)	sin(a)를 구한다(cos, tan).
a/b / divide(a,b)	a를 b로 나눈다.	log(a)	log(a)를 구한다.
a*b / multiply(a,b)	a를 b로 곱한다.	dot(a,b)	벡터의 내적을 구한다.
exp(b)	b의 지수를 구한다.	a==b	a와 b를 비교한다(<,> 등)



행렬 예 :

- 행렬의 덧셈과 뺄셈은 대응하는 원소끼리 즉 같은 위치에 있는 원소끼리 가능 (행렬의 크기가 서로 같은 경우)
- 예 : 행렬덧셈

$$\begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} + \begin{pmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{pmatrix} = \begin{pmatrix} x_{11}+y_{11} & x_{12}+y_{12} \\ x_{21}+y_{21} & x_{22}+y_{22} \end{pmatrix} \quad (1)$$

1) 기본 연산

In [ ]:
```
arr1 = np.array([10, 20, 30, 40])
arr2 = np.array([1, 2, 3, 4])
```

In [ ]:
```
arr1 + arr2
```

In [ ]:
```
arr1 - arr2
```

In [ ]:
```
arr2 * 2
```

In [ ]:
```
arr2 ** 2
```

In [ ]:
```
arr1 * arr2
```

In [ ]:
```
arr1 / arr2
```

In [ ]:
```
arr1 / (arr2 ** 2)
```

In [ ]:
```
arr1 > 20
```

2) 행렬 연산

- 연산자 : * , np.dot( ), np.multiply( )
- 요소간곱 : np,dot(A,B), A.dot(B)
- 행렬곱 : np.dot(A,B), np.dot(A,B)

행렬 연산	사용 예
행렬곱(matrix product)	A.dot(B), 혹은 np.dot(A,B)
전치행렬(transpose matrix)	A.transpose(), 혹은 np.transpose(A)
역행렬(inverse matrix)	np.linalg.inv(A)
행렬식(determinant)	np.linalg.det(A)

In [ ]:
```
A = np.array([0, 1, 2, 3]).reshape(2,2)
A
```

In [ ]:
```
B = np.array([3, 2, 0, 1]).reshape(2,2)
B
```

In [ ]:
```
# 요소곱
A*B
```

In [ ]:
```
# 요소곱
np.multiply(A,B)
```

In [ ]:
```
# 행렬곱
A.dot(B) # np.dot(A,B)
```

In [ ]:
```
np.dot(A,B)
```

4. 배열 인덱싱

- 인덱싱(Indexing): 배열의 위치나 조건을 지정해 배열의 원소를 선택
- ndarray 클래스로 구현한 다차원 행렬의 원소 하나 하나는 다음과 같이 콤마(comma ,)를 사용하여 접근할 수 있다.
- 콤마로 구분된 차원을 축(axis)이라고도 한다. 플롯의 x축과 y축을 떠올리면 될 것이다.

In [ ]:
```
a1 = np.array([0, 10, 20, 30, 40, 50])
a1
```

In [ ]:
```
a1[0]
```

In [ ]:
```
a1[4]
```

In [ ]:
```
a1[5] = 70
a1
```

In [ ]:
```
a1[[1,3,4]]
```

In [ ]:
```
a2 = np.arange(10, 100, 10).reshape(3,3)
a2
```

In [ ]:
```
a2[0, 2]  # 첫번째 행의 세번째 열
```

In [ ]:
```
a2[2, 2] = 95
a2
```

In [ ]:
```
a2[1] # 특정 행 전체 지정
```

In [ ]:
```
a2[1] = np.array([45, 55, 65])
a2
```

In [ ]:
```
a2[1] = [47, 57, 67]
a2
```

In [ ]:
```
a2[0,2]
```

In [ ]:
```
a2[0,1]
```

불리안(Boolean) 방식 행렬 인덱싱 : 배열명[조건]

- True인 원소만 선택
- 인덱스의 크기가 행렬의 크기와 같아야 한다.

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

In [ ]:
```
a[(a % 2) == 0]
```

In [ ]:
```
aa = np.array([[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]]) # 2차원 이상의 인덱스인 경우
aa[aa % 2 == 0]
```

5. 배열 슬라이싱(Slicing)

- 범위를 지정해 배열의 원소를 선택
- ndarray 클래스 구현한 다차원 행렬의 원소 중 복수 개를 접근하려면 --> 슬라이싱(slicing)과 comma(,)를 함께 사용

배열의 슬라이싱 : 배열[시작위치:끝위치]

In [ ]:
```
import numpy as np
b1 = np.array([0, 10, 20, 30, 40, 50])
b1[1:4]
```

In [ ]:
```
b1[:3]
```

In [ ]:
```
b1[2:]
```

In [ ]:
```
b1[2:5] = np.array([25, 35, 45])
b1
```

In [ ]:
```
b1[3:6] = 60
b1
```

2차원 배열의 슬라이싱: 배열[행시작위치:행끝위치, 열시작위치:열끝위치]

In [ ]:
```
import numpy as np
arr = np.arange(10, 100, 10).reshape(3,3)
arr
```

In [ ]:
```
arr[1:3, 1:3]
```

In [ ]:
```
arr[2]
```

In [ ]:
```
arr[1][0:2]
```

In [ ]:
```
arr[2, :]
```

In [ ]:
```
arr[2:, :]
```

In [ ]:
```
arr[:, :2]
```

In [ ]:
```
arr[1, :2]
```

In [ ]:
```
arr[1:2, :2]
```

In [ ]:
```
arr[:3, 1:]
```

In [ ]:
```
arr[0:2, 1:3] = np.array([[25, 35], [55, 65]])
arr
```

6. 함수

유니버설 함수 (ufunc) : ndarray 안에 있는 데이터 원소 별로 연산을 수행하는 함수

In [ ]:
```
import numpy as np
a = np.arange(1, 6)

print(a)
print(np.sqrt(a))
print(np.square(a))
print(np.exp(a))
print(np.log(a))
print(np.sin(a))
print(np.tanh(a))
```

유니버설 이항 함수 : 2개의 배열을 인자로 취해서 단을 배열을 반환하는 함수

In [ ]:
```
import numpy as np
a1 = np.array([2,3,4])
a2 = np.array([1,5,2])
print(np.maximum(a1, a2))
print(np.minimum(a1, a2))
```

통계 함수 : 배열 전체 혹은 배열에서 한 축에 따르는 자료에 대한 통계를 계산하는 함수

함수	설명
sum	배열 전체 혹은 특정 축에 대한 모든 원소의 합
mean	배열 전체 혹은 특정 축에 대한 모든 원소의 평균
min, max	최소 값, 최대 값
argmin, argmax	최소 원소의 색인 값, 최대 원소의 색인 값

In [ ]:
```
import numpy as np
a = np.array([[6,2,4,8,2,4],[-7,3,4,-1,9,0]])
print(a)
```

```
print(np.sum(a))
print(np.mean(a))
print(np.min(a))
print(np.max(a))
print(np.argmin(a))
print(np.argmax(a))
```

In [ ]:
```
# 일부 통계 함수는 선택적으로 axis 인자를 받아 해당 axis에 대한 통계를 계산하고 한 차수 낮은 배
import numpy as np
a = np.array([[6,2,4,8,2,4],[-7,3,4,-1,9,0]])
print(a)
print(np.sum(a, axis=0))
print(np.mean(a, axis=0))
print(np.argmax(a, axis=0))
print(np.argmax(a, axis=1))
```

7. 배열 변환

- 배열 변환 방법 : 전치/shape변환/요소추가삭제/분할/변형/결합

전치(Transpose)

- Tranpose는 행렬의 인덱스가 바뀌는 변환
- 이 속성은 전치된 행렬을 반환
- A.transpose( ), np.transpose(A), A.T

In [ ]:
```
import numpy as np
A1 = np.array([1, 2, 3])       # 크기 (3,)인 1차원 배열 -> (행벡터, 열벡터 두 가지 모
A2 = np.array([[1, 2, 3]])     # 크기 (1,3)인 2차원 배열 (행벡터)
A3 = np.array([[1], [2], [3]]) # 크기 (3,1)인 2차원 배열 (열벡터)

print(A1)
print(A2)
print(A3)
print('-----------')

print(A1.T) # 전치 안됨
print(A2.T) # 전치 print(a2.transpose()), print(np.transpose(a2)
print(A3.T) # 전치
```

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

In [ ]:
```
# 전치행렬
A.transpose()
```

In [ ]:
```
# 전치행렬
np.transpose(A)
```

In [ ]:
```
# 전치행렬
A.T
```

Reshape

- Shape 변경
  - . ravel : 배열 shape을 1차원 배열로 만드는 메서드
  - . reshape : 데이터 변경없이 지정된 shape 변환하는 메서드
- numpy.ravel()
  - 배열을 1차원 배열로 반환하는 메서드
  - numpy.ndarray 배열 객체의 View를 반환
  - ravel 메서드가 반환하는 배열의 요소를 수정하면 원본 배열 요소에도 반영됨
- numpy.reshape()
  - 실제 데이터를 변경하는 것은 아님.
  - 배열 객체의 shape 정보만을 수정.

In [ ]:
```
import numpy as np

a = np.arange(6)                         # 1d array
print(a)

b = np.arange(12).reshape(4,3)           # 2d array
print(b)

c = np.arange(24).reshape(2,3,4)         # 3d array
print(c)
```

In [ ]:
```
import numpy as np
data = np.array([1, 2, 3, 4, 5, 6])
data
```

In [ ]:
```
data.reshape(2, 3)
```

In [ ]:
```
data.reshape(3, 2)
```

In [ ]:
```
data.reshape(3,2).transpose()
```

In [ ]:
```
data = np.array([1, 2, 3, 4, 5, 6])
print(data.transpose())## ????  array([1, 2, 3, 4, 5, 6])
```

In [ ]:
```
data1 = np.array([[1, 2, 3, 4, 5, 6]]) ## ????
data1.transpose()
```

In [ ]:
```
np.arange(0,10).reshape(2,5)
```

---------- END ----------