

NUMPY 라이브러리

목차

- Numpy
- Nddarray 다차원배열
- 배열의 생성
- 모양 변경
- 기본 연산
- 인덱싱과 슬라이싱
- 배열 쌓기
- 배열 분할
- 수학 함수
- 선형 대수
- 랜덤
- 파일 저장 및 로드

[핵심 키워드] : ones, zeros, reshape, full,

[핵심 포인트]

클래스의 추가적인 기능에 대해 알아본다.

- 과학적 컴퓨팅을 위한 기본 패키지이다.
 - 강력한 N차원 배열 객체
 - 정교한 (방송) 기능 - 배열과 스칼라값의 계산
 - C/C++ 및 Fortran 코드 통합을 위한 도구
 - 유용한 선형 대수, 푸리에 변환 및 난수 기능

- 강력한 N차원 배열 객체로 범용적 데이터 처리를 위한 다차원 컨테이너
 - 과학 연산을 위한 파이썬 핵심 라이브러리 중 하나로 빠른 고성능 연산을 위해 C언어로 구현
 - 파이썬의 편의성과 C언어의 연산 능력을 동시에 이용
 - 벡터부터 텐서에 이르기까지 다양한 방법으로 만드는 다차원 배열 제공하며, 선형대수 문제를 쉽게 처리할 수 있음
 - 스칼라 scalar
 - 하나의 값. $a = 10$
 - 벡터 vector
 - 순서가 있는 1차원 배열. $x = [0, 1, 2]$
 - 순서가 없는 배열은 집합 set
 - 행렬 matrix
 - 벡터 m 이 n 개 존재($m \times n$)하는 2차원 배열
 - $1 \times n$ 행 row 벡터 $[[1 \ 2]]$ 와 $m \times 1$ 열 column 벡터 $[[1] \ [2]]$ 는 서로 전치 관계
 - 텐서 tensor
 - 같은 크기의 행렬로 구성된 3차원 이상 배열
 - 인기있는 서드파티 라이브러리들이 NumPy를 기본 자료구조로 사용하거나 호환됨
 - matplotlib, pandas, opencv, pytorch, tensorflow 등

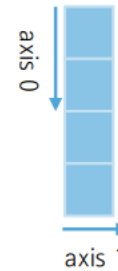
- NumPy의 다차원 배열 객체
 - ndarray의 주요 속성
 - ndarray.ndim: 배열의 축 axis (차원 dimensions) 수
 - ndarray.shape: 배열의 차원으로, 각 차원 배열의 크기를 정수 튜플로 나타냄. shape
 - (4 axis 0), (4 axis 0, 3 axis 1), (2 axis 0, 4 axis 1, 3 axis 2)
 - 4 ≡ (4,) → (1, 4): 1x4 행 벡터
 - (4, 1): 4x1 열 벡터
 - ndarray.size: 배열의 총 요소 수로, 각 차원 배열의 크기를 모두 곱한 값
 - ndarray.dtype: 배열 요소 타입
 - 정수: numpy.int8, numpy.int16, numpy.int32, numpy.int64 (== int, 정수 기본 타입)
 - 실수: numpy.float16, numpy.float32, numpy.float64 (== float, 실수 기본 타입), numpy.float128
 - 기타: numpy.complex, numpy.bool, numpy.str, numpy.object
 - ndarray.itemsize: 바이트 단위 배열 요소의 크기 (요소 타입 크기)
 - ndarray.data: 실제 배열 요소를 포함하는 버퍼로, 이미지나 오디오 같은 바이너리 데이터를 다룰 때 사용

1D array



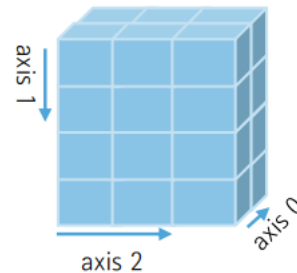
ndim: 1
shape: (4,)
size: 4
dtype: int64
itemsize: 8

2D array



ndim: 2
shape: (4, 1)
size: 4
dtype: int64
itemsize: 8

3D array



ndim: 3
shape: (2, 4, 3)
size: 24
dtype: int64
itemsize: 8

- 파이썬 객체 (리스트, 튜플)
 - 2차원 이상은 반드시 열의 개수가 일치해야 함
 - 요소 중에 실수가 하나라도 포함되면 실수 타입
 - 정수 요소의 디폴트 타입은 int64이고 실수 요소의 디폴트 타입은 float64

```
import numpy as np
show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
a1 = np.array([2, 3, 4], dtype=np.int8)
a2 = np.array((1.5, 7, 9, 10))
a3 = np.array([[2, 3, 4], [7, 9, 10]])
show("a1:", a1)
show("a2:", a2)
show("a3:", a3)
```

- 범위 기반
 - `arange()`는 `range()`처럼 주어진 `[start, stop)`에서 균일한 정수 또는 실수 배열 반환
 - `numpy.arange([start]stop, [step,]dtype=None)`
 - `linspace()`는 `arange()`와 유사하나 `step`이 실수이면 오차가 발생할 수 있으므로 개수 사용
 - `numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)`

```
import numpy as np
show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
a1 = np.arange(27+1)
a2 = np.arange(1, 10+1, 0.5)
a3 = np.linspace(1, 10+1, 20)
show("a1:", a1)
show("a2:", a2)
show("a3:", a3)
```


- 특정 값으로 초기화
 - `numpy.zeros(shape, dtype=float)`: 배열 전체를 0으로 초기화
 - `numpy.ones(shape, dtype=None)`: 배열 전체를 1로 초기화. 기본 타입은 float
 - `numpy.full(shape, fill_value, dtype=None)`: 배열 전체를 전달한 값으로 초기화
 - `numpy.eye(N, M=None, k=0, dtype=float)`: $N \times M$ 배열에 대해 대각선이 1이고 나머지는 0으로 초기화
 - M을 생략하면 $M=N$ (항등 또는 단위 행렬 identity matrix), k는 대각선 인덱스로 0은 기준 대각, 양수는 기준 대각 위쪽, 음수는 아래쪽
 - `numpy.diag(a, k=0)`: 단위 행렬의 대각선 값들을 하나의 배열로 반환

```
import numpy as np
show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
a0 = np.zeros(5)
a1 = np.zeros((5,1), dtype=int)
a2 = np.ones((3,4), dtype=int)
a3 = np.full((2,3), -1)
a4 = np.eye(5, k=1)
a5 = np.diag(a4, k=1)
show("a0:", a0);show("a1:", a1);show("a2:", a2)
show("a3:", a3);show("a4:", a4);show("a5:", a5)
```

- 기존 배열 복사
 - `numpy.ones_like(a, dtype=None)`: 1로 채워진 배열
 - `numpy.zeros_like(a, dtype=None)`: 0으로 채워진 배열
 - `Numpy.full_like(a, fill_value, dtype=None)`: 전달한 값으로 채워진 배열

```
import numpy as np

show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')

a = np.linspace(1, 12, 12).reshape((2, 3, 2))

b1 = np.ones_like(a)
b2 = np.zeros_like(a)
b3 = np.full_like(a, -1)

show("a:", a)
show("b1:", b1)
show("b2:", b2)
show("b3:", b3)
```

- 배열은 각 축을 따라 요소의 수로 지정된 모양을 가짐
 - 배열 자체는 변하지 않고 새로운 차원으로 모양만 바꾸므로 size는 같아야 함
 - `numpy.reshape(a, new_shape)`
 - `ndarray.reshape(new_shape)`
 - 모양에 맞춰 배열 크기도 함께 변경하므로 새 모양의 size가 더 크면 나머지 요소는 0으로 채움
 - `ndarray.resize(new_shape)`

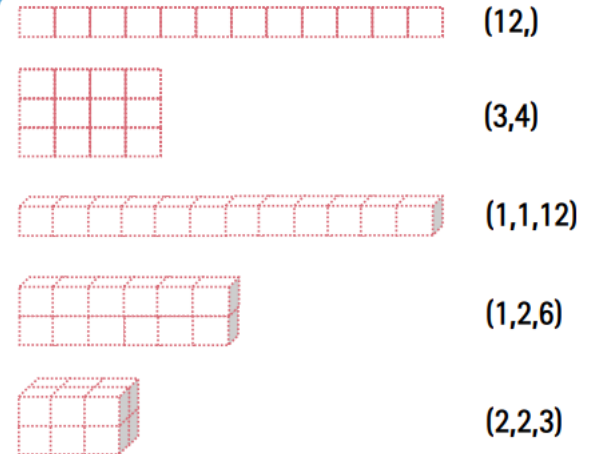
```
import numpy as np
show = lambda m, o : print(m, o.shape,
o.dtype, '\n', o, '\n')

a1 = np.full((6, 3), -1)
a2 = np.reshape(a1, (3, 3, 2))
a3 = np.arange(1, 10 + 1, 0.5).reshape((5,
4))
a4 = np.linspace(1, 10 + 1, 20).reshape((2,
5, 2))
show("a1:", a1)
show("a2:", a2)
show("a3:", a3)
show("a4:", a4)
```

memory (12 elements)



shape



- 산술 연산과 관계 연산

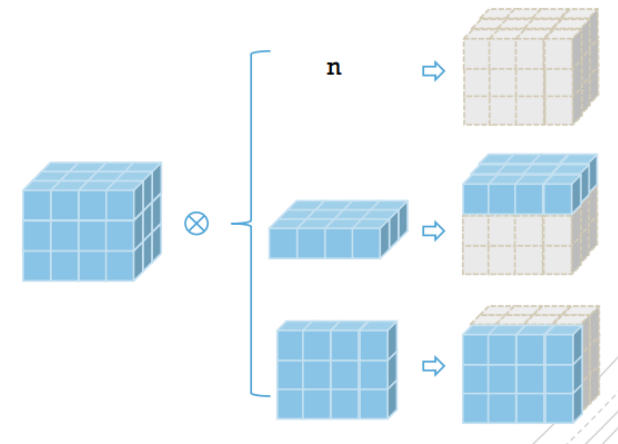
- 산술 연산과 관계 연산은 행렬 요소 사이 1:1로 적용되며 관계 연산은 불 결
 - 피 연산자는 배열 또는 스칼라 값으로 둘 다 배열일 때는 shape와 size가 같아야 함
 - 스칼라 값은 같은 크기 및 모양의 배열로 브로드캐스팅 broadcasting한 후 연산 수행
 - 산술 연산은 연산자 (+, -, *, /, %, **)와 함수(numpy 모듈의 add, subtract, multiply, divide, mod, power) 모두 지원. *는 일반적인 행렬 곱셈이 아님
 - 관계 연산의 결과는 불 배열

```
import numpy as np
show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
a1 = np.array([10, 20, 30, 40, 50, 60])
a2 = np.linspace(5, 6, a1.size)
b1 = a1 - a2
b2 = np.power(b1, 2) # b1 ** [2, 2, 2, 2, 2, 2]
b3 = np.sin(a1) * 10 # np.sin(a1) * [10, 10, 10, 10, 10, 10]
b4 = a1 % a2
b5 = b1 < 25 # b1 < [25, 25, 25, 25, 25, 25]
show("a1:", a1) ; show("a2:", a2)
show("b1:", b1) ; show("b2:", b2) ; show("b3:", b3) ; show("b4:", b4) ;
show("b5:", b5)
```

● 브로드캐스팅

- 스칼라 값이나 벡터를 배열과 연산할 때 이를 배열의 shape와 같도록 확장한 후 기존 데이터 복사
 - 배열과 스칼라 값 사이 연산: 스칼라를 배열의 shape로 확장한 후 스칼라 값 복사
 - 벡터와 벡터 사이 연산: 벡터 N, M에 대해 양쪽 다 배열의 N x M shape로 확장한 후 행 또는 열 단위 요소 복사
 - 배열과 벡터 사이 연산: 배열의 마지막 축 크기와 벡터의 크기가 같을 때, 벡터를 배열의 shape로 확장한 후 벡터 요소 복사
 - 크기가 다른 배열과 배열 사이 연산: 두 배열의 마지막 축부터 차례로 비교해 축의 크기가 같거나 1일 때, 양쪽 배열을 큰 축 기준으로 확장한 후 배열 요소 복사

```
import numpy as np
show = lambda m, o : print(m, o.shape,
o.dtype, '\n', o, '\n')
a1 = np.linspace(1, 12, 12).reshape((3, 4))
a2 = np.arange(1, 5)
a3 = a2.reshape((4, 1))
a4 = a1.reshape((3, 1, 4))
b1 = a1 + a2
b2 = a2 + a3
b3 = a1 + a4
show("a1:", a1) ; show("a2:", a2) ;
show("a3:", a3)
show("b1:", b1) ; show("b2:", b2) ;
show("b3:", b3)
```

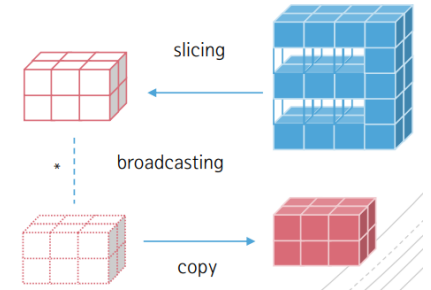


- 리스트처럼 NumPy 배열도 인덱스로 요소에 접근하고 원하는 부분만 잘라내는 슬라이싱 가능
 - 리스트와 다른점은 슬라이싱은 원본 배열의 데이터를 참조하는 새로운 배열이므로 슬라이싱된 배열을 수정하면 원본 배열도 함께 수정됨
 - 1차원 배열의 슬라이싱은 `[[start]:[end]:[step]]`
 - 인덱스 `i`가 `start`일 때 `i < end` 동안 `i += step` 단위로 이동하며 요소 접근. `i`가 음수면 마지막 요소부터 역순으로 접근
 - `start`를 생략하면 0, `end`를 생략하면 -1(마지막 요소), `step`을 생략하면 1

```
import numpy as np

show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
a1 = np.arange(1, 10+1)**2
a2 = a1[2:9]
show("a1", a1) ; show("a2", a2)
a1[3] = a1[1] + a1[2]
a2[:5:2] = 10_1000
for i in range(len(a1)):
    print(a1[(i+1)*-1], end=', ')
print()
```

- 다차원 배열은 축당 하나의 인덱스를 가질 수 있으며 인덱스는 쉼표로 구분 ➡ [x, y]
 - 축 수보다 적은 인덱스를 제공하면 누락된 인덱스는 슬라이스로 간주
 - 점(...)은 완전한 인덱싱에 필요한 만큼의 콜론을 나타냄
 - 5차원 배열 x에 대해
 - $x[1, 2, \dots] == x[1, 2, :, :, :]$
 - $x[\dots, 3] == x[:, :, :, :, 3]$
 - $x[4, \dots, 5, :] = x[4, :, :, 5, :]$
 - 다차원 배열에 대한 반복은 첫 번째 축에 대해 수행
 - 반복자인 flat 속성을 이용하면 배열의 모든 요소에 접근할 수 있음
- 슬라이싱할 때 추가 연산을 통해 브로드캐스팅이 발생하면 사본 복사



```
import numpy as np

show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')

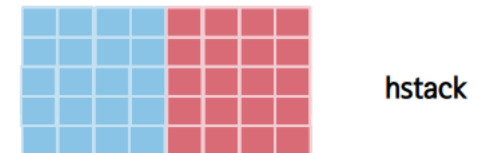
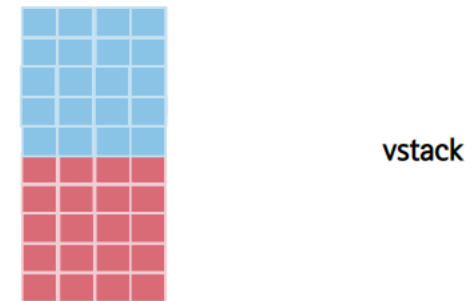
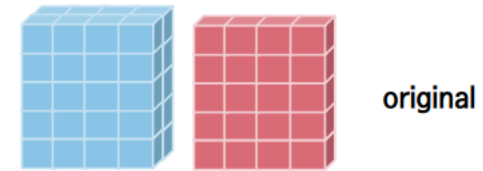
a1 = np.fromfunction(lambda x, y, z: x + y + z, (2, 5, 4), dtype=int)
a2 = a1[:, 1::2, :3] * 10 # a2는 새로운 배열

a2[1, ...] = -1

show("a1", a1)
for array in a2.flat:
    print("out:\n", array) # print(element, end=', ')
```

서로 다른 배열 쌓기

- 서로 다른 축을 따라 여러 배열을 수평 또는 수직으로 쌓은 새로운 배열 생성
 - `vstack(tup)`, `hstack(tub)`: 세로(행 방향), 가로(열 방향) 순서로 쌓음
 - `tup`: 배열 시퀀스로 배열 모양은 첫 번째 축을 제외하고 모두 동일해야 함. 1차원 배열은 길이가 같아야 함
 - `concatenate((a1, a2, ...), axis=0, ...)`: 기존 축을 따라 배열 시퀀스 결합
 - `a1, a2, ...`: 배열 시퀀스로 첫 번째 축의 차원을 제외하고 동일한 모양을 가져야 함
 - `axis`: 배열이 결합될 축. `None`으로 설정하면 1차원으로 결합



```
import numpy as np

show = lambda m, o : print(m, o.shape, o.dtype, '\n',
o, '\n')

a1 = np.fromfunction(lambda x, y, z: x + y + z, (2, 5,
4), dtype=int)
a2 = np.arange(1, (1*5*4)+1).reshape((1, 5, 4)) * 10

a3 = np.vstack((a1[0, ...], a2[0, ...]))
a4 = np.hstack((a1[1, ...], a2[0, ...]))
a5 = np.concatenate((a1, a2), 0)

show("a1", a1) ; show("a2", a2)
show("a3", a3) ; show("a4", a4) ; show("a5", a5)
```


- 하나의 배열은 여러 개의 작은 배열로 분할될 수 있음
 - `vsplit(ary, indices_or_sections)`: 배열을 세로로(행 방향) 첫 번째 축을 따라 여러 하위 배열로 분할
 - `ary`는 배열, `indices_or_sections`는 분할 개수 또는 분할 위치 지정 스퀀스 또는 배열
 - 반환은 튜플 배열
 - `hsplit(ary, indices_or_sections)`: 배열을 가로로(열 방향) 두 번째 축을 따라 여러 하위 배열로 분할

```
import numpy as np

show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')

a = np.arange(24).reshape(3, 2, 4)

b1, b2, b3 = np.vsplit(a, 3)
b4, b5 = np.vsplit(a, np.array([1]))

c1, c2 = np.hsplit(a, 2)

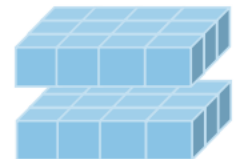
show("a", a)
show("b1", b1) ; show("b2", b2) ; show("b3", b3) ; show("b4", b4) ; show("b5", b5)
show("c1", c1) ; show("c2", c2)
```



original



vsplit (axis=0)



hsplit (axis=1)

- 배열에 대한 연산 결과로 만들어지는 새로운 배열은 기존 배열의 참조 또는 복사
 - 대입문에 의한 단순 할당과 함수의 인자 전달은 객체 참조
 - 얇은 복사: view() 메소드나 슬라이싱은 동일한 데이터를 새로운 관점에서 보여주는 새로운 배열 객체를 만듦
 - 깊은 복사: copy() 메소드는 배열과 해당 데이터의 전체 사본을 만듦

```
import numpy as np

show = lambda m, o : print(m, o.shape,
                           o.dtype, '\n', o, '\n')

def foo(n):
    return id(n)

a = np.arange(20).reshape(4, 5)
b = a
c = a.view()
print(b is a, id(a) == foo(a))
print(c is a, c.flags.owndata)
# .flags.owndata 데이터 유무
```

```
d = c.reshape((2, 10))
show("d", d) ; show("a", a)
s = a[: 1:3]
s[:] = -1
show("a", a)
e = a.copy()
print(e is a)
d[1, 2] = 2_000_000
show("e", e) ; show("a", a)
m = np.arange(1_000_000)
n = m[:100].copy()
del m
```

삼각 함수

- `numpy.pi`: π 상수
- `numpy.sin(x)`, `numpy.cos(x)`, `numpy.tan(x)` : x 에 대해 sine, cosine, tangent 계산.
 x 는 라디안 값 또는 배열
- `numpy.arcsin(x)`, `numpy.arccos(x)`, `numpy.arctan(x)`: x 에 대해 sine, cosine, tangent의 역 함수 계산
- `numpy.radians(x)`: x 에 대해 디그리를 라디안으로 변환. $degree(\pi/180)$
- `numpy.degrees(x)`: x 에 대해 라디안을 디그리로 변환. $radian(180/\pi)$

```
import numpy as np

show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')

print(np.pi)
print(np.sin(30 * np.pi / 180), np.cos(3 * np.pi / 2), end='\n\n')
a1 = np.degrees([np.pi/4, 3*np.pi/4, 5*np.pi/4, 7*np.pi/4])
a2 = np.radians(a1)
a3 = np.sin(a1 * np.pi / 180)
a4 = np.cos(a2)
show("a1", a1) ; show("a2", a2) ; show("a3", a3) ; show("a4", a4)
```

● 지수와 로그

- `numpy.e` : 지수(자연 로그 밑) 상수(약 2.718281...)
- `numpy.exp(x)`: x 에 대해 $y = e^x$ 인 지수(자연 로그 역) 계산
- `numpy.log(x)`: x 에 대해 밑이 e 인 자연 로그(지수 함수 역) 계산
 - 밑이 10인 상용로그는 `numpy.log10(x)`

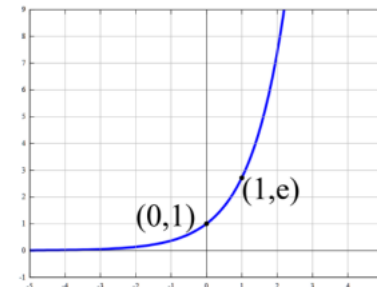
```
import numpy as np

show = lambda m, o : print(m, o.shape,
o.dtype, '\n', o, '\n')

d1 = np.exp(1)
a1 = np.exp([i for i in range(1, 10 + 1, 2)])
d2 = np.log(np.e)
a2 = np.log(a1)
print("d1:", d1)
print("d2:", d2)

show("a1", a1)
show("a2", a2)
```

x	$(1+x)^{\frac{1}{x}}$	x	$(1+x)^{\frac{1}{x}}$
0.1	2.59374...	-0.1	2.86797...
0.01	2.70481...	-0.01	2.73199...
0.001	2.71692...	-0.001	2.71964...
0.0001	2.71814...	-0.0001	2.71841...
0.00001	2.71826...	-0.00001	2.71829...
⋮	⋮	⋮	⋮



- 기타 함수

- numpy.abs(x), numpy.absolute(x): x에 대해 절대값 계산
 - abs()는 정수 한정
- numpy.ceil(x): x에 대해 x보다 작은 정수 중 가장 큰 값 계산
- numpy.floor(x): x에 대해 x보다 큰 정수 중 가장 작은 값 계산
- numpy.sqrt(x): x에 대해 제곱근 계산

```
import numpy as np

show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')

a1 = np.array([-2.5, 1, -1.7, 5.1, 4.0, -0.2, 6.8])
a2 = np.absolute(a1)
a3 = np.ceil(a1)
a4 = np.floor(a1)
a5 = np.sqrt(a2)
show("a1", a1)
show("a2", a2)
show("a3", a3)
show("a4", a4)
show("a5", a5)
```

- 전치 행렬 transposed matrix

- 행과 열을 서로 맞바꾼 행렬로 numpy 모듈이나 ndarray 객체의 transpose() 또는 T 프로퍼티 사용

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \Rightarrow A^T = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

```
import numpy as np

show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')

a1 = np.array([1, 2, 3, 4])
a2 = np.array([[1, 2, 3], [4, 5, 6]])
a3 = np.linspace(10, 120, 12).reshape((3, 2, 2))
b1 = a1.T
b2 = a2.transpose()
b3 = np.transpose(a3)

show("a1:", a1) ; show("a2:", a2) ; show("a3:", a3)
show("b1:", b1) ; show("b2:", b2) ; show("b3:", b3)
```

- 행렬 곱셈

- 두 행렬에 대한 곱셈은 연산자 @ 또는 numpy 모듈이나 ndarray 객체의 dot() 사용
 - 첫 번째 배열 행의 요소 수와 두 번째 배열 열의 요소 수는 같아야 함

$$\begin{array}{l} ax + by + cz = p \\ dx + ey + fz = q \\ gx + hy + iz = r \end{array} \iff \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

```
import numpy as np

show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')

a1 = np.array([[2, -1, 5], [-5, 2, 2], [2, 1, 3]])

a2 = np.array([[0, 1, 0], [1, 0, 1], [1, 1, 0]])
a3 = np.array([1, -1, 1])
a4 = np.array([1, 0, 1]).reshape((3,1))

b1 = np.dot(a1, a2)
b2 = a1.dot(a3)
b3 = a1 @ a4

show("a1:", a1) ; show("a2:", a2) ; show("a3:", a3) ; show("a4:", a4)
show("b1:", b1) ; show("b2:", b2) ; show("b3:", b3)
```

● 역 행렬과 방정식 해

- 역 행렬은 $(n \times n)$ 정방 행렬에 대한 곱셈 결과가 항등원인 단위행렬을 만드는 행렬

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad A^{-1} = \begin{bmatrix} x & y \\ u & v \end{bmatrix} \quad \Rightarrow \quad A A^{-1} = E \quad \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x & y \\ u & v \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

- `numpy.linalg.inv(a)`: 배열의 역 행렬 계산
- `numpy.linalg.solve(a, b)`: 연립방정식 해 계산

$$\begin{array}{l} 2x + 3y = 4 \\ 5x + 6y = 7 \end{array} \quad \Rightarrow \quad \begin{bmatrix} 2 & 3 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \end{bmatrix}$$

```
import numpy as np

show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')

a1 = np.random.randint(10, size=(3, 3))
a2 = np.linalg.inv(a1)
a3 = np.array([[2, 3], [5, 6]])
a4 = np.array([4, 7])
a5 = np.linalg.solve(a3, a4)
show("a1", a1) ; show("a2", a2)
show("a3", a3) ; show("a4", a4) ; show("a5", a5)
```


- 연립 방정식에 대한 A, B 배열에 대해 $A^{-1} \cdot B$ 계산

$$\begin{array}{l} 2x + 3y = 4 \\ 5x + 6y = 7 \end{array} \Rightarrow \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ 5 & 6 \end{bmatrix}^{-1} \begin{bmatrix} 4 \\ 7 \end{bmatrix}$$

```
import numpy as np

show = lambda m, o : print(m,
o.shape, o.dtype, '\n', o, '\n')

A = np.array([[2, 3], [5, 6]])
B = np.array([4, 7])
C = np.linalg.inv (A)
D = np.dot(C, B)
show("solve", D)
```

- 균등 분포 uniform distribution : 확률 밀도 $\mathcal{P} x = 1/(b - a)$ 에 대해 값이나 요소를 $[a, b)$ 에서 균등 배치
 - numpy.random.random(size=None): [0.0, 1.0) 범위 실수
 - size는 생략하거나 스칼라 값, 2차원 이상은 shape로 반환은 스칼라 값 또는 배열
 - numpy.random.uniform(low=0.0, high=1.0, size=None): [low, high) 범위 실수 타입
 - numpy.random.randint(low, high=None, size=None, dtype=int): [low, high) 범위 정수 타입
 - high를 생략하면 low = 0, high = low

```
import numpy as np

show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
a1 = np.random.random(6)
a2 = np.random.random((2, 3))
a3 = np.random.uniform(1.0, 2.0, (2, 3))
d = np.random.randint(10)
a4 = np.random.randint(1, 10, (2, 3))
show("a1", a1) ; show("a2", a2) ; show("a3", a3)
print("d:", d) ; show("a4", a4)
```

- `numpy.random.shuffle(x)`: 시퀀스 `x`의 내용을 섞어서 수정. 다차원 배열은 첫 번째 축 기준
- `numpy.random.choice(a, size=None, replace=True, p=None)`: 주어진 1차원 배열에서 무작위 샘플 생성
 - `replace`는 선택 값 재 사용 유무, `p`는 `a` 항목 선택 확률
- `numpy.seed(seed=None)`: 의사 난수 제너레이터에서 사용할 시드 값 설정

```
import numpy as np

show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
np.random.seed(1)
a1 = np.arange(9)
np.random.shuffle(a1)
a2 = np.arange(3*3).reshape((3, 3))
np.random.shuffle(a2)
a3 = np.random.choice([4, 2, 6, 1], 3, False, [0.4, 0.2, 0.3, 0.1])
show("a1", a1)
show("a2", a2)
show("a3", a3)
```

- 정규 분포 normal distribution
 - 가우스 분포의 확률 밀도 \mathcal{P} 에 대해
 - 평균에서 최대치이며, 표준 편차와 함께 확산되며 증가함
 - 멀리 떨어져 있는 것보다 평균에 가까운 샘플을 반환할 확률이 높음
 - `numpy.random.normal(loc=0.0, scale=1.0, size=None)`: 정규 분포
 - `loc`은 분포의 평균(중앙), `scale`은 양의 값으로 표준 편차(확산 또는 너비), `size`는 스칼라 또는 shape
 - `numpy.random.standard_normal(size=None)`: 표준 정규 분포 standard normal (평균=0, 표준편차=1). 무작위 샘플은 N

```
import numpy as np
show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
mu, sigma = 0, 0.1
a1 = np.random.normal(mu, sigma, 50)
a2 = np.random.normal(4, 1.7, size=(3, 4)) # N(4, 2.89)
a3 = np.random.standard_normal(50)
a4 = 4 + 1.7 * np.random.standard_normal((3,4)) # N(4, 2.89)
show("a1:", a1)
print("mean and standard deviation:", abs(mu - np.mean(a1)), sigma - np.std(a1, ddof=1),
end="\n\n")
show("a2:", a2) ; show("a3", a3) ; show("a4", a4)
```

- 데이터의 재 사용을 위해 ndarray의 배열 요소를 파일로 저장하거나 파일로부터 데이터를 읽어 ndarray 객체로 변환
 - 바이너리 형식으로 확장자는 .npy
 - `numpy.save(fname, arr)` : 1개 배열을 파일로 저장
 - file은 확장자 생략 가능
 - `numpy.savez(fname, *args, **kwargs)` : n개의 배열을 파일로 저장
 - kwargs는 배열 이름을 키, 배열을 값으로 나열한 키워드 인자
 - `numpy.load(fname)`: .npy 파일에서 배열 로드. 압축된 바이너리라면 압축 해제
 - 파일명은 확장자까지 모두 포함해야 하며, 반환 객체는 ndarray와 호환되는 `numpy.lib.npyio.NpzFile`
 - `numpy.lib.npyio.NpzFile.close()`: 파일 닫기
 - 파일을 닫으면 더 이상 배열 접근 불가
 - 압축된 바이너리 파일
 - `numpy.savez_compressed(fname, *args, **kwargs)`: 압축 후 저장
 - 텍스트 파일(엑셀 등과 호환) 형식
 - `numpy.savetxt(fname, x)`: 파일 저장
 - x는 1차원 또는 1차원 배열
 - `numpy.loadtxt(fname)`: 파일 로드

```
import numpy as np
show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
a1 = np.arange(360+1)
a2 = np.sin(a1 * np.pi / 180)
show("a1", a1)
show("a2", a2)
np.savez_compressed("sin_sample", x=a1, y=a2)
a3 = np.load("sin_sample.npz")
print((a3['x'] == a1).all()) # a3['x'] == a1의 결과는 불 배열이며, all()은
전체 요소가 같은지 검사
print((a3['y'] == a2).all())
a3.close()
```

- 간단히 Life라고도 알려진 Game of Life는 1970년 영국의 수학자 John Horton Conway가 고안한 셀룰러 오토마톤입니다.
- 공간은 사각형 셀의 무한한 2차원 직교 격자
- 셀은 수평, 수직 또는 대각선으로 직접 인접한 셀인 8개의 이웃과 상호 작용
 - 살아있는 이웃이 2개 미만인 살아있는 세포는 인구 부족으로 의해 사망
 - 3개 이상의 살아있는 이웃이 있는 살아있는 세포는 인구과밀로 사망
 - 2~3개의 살아있는 이웃이 있는 살아있는 세포는 다음 세대까지 생존
 - 정확히 3개의 살아있는 이웃이 있는 공간은 세포 생성

