



numpy / scipy

1

numpy



numpy 라이브러리

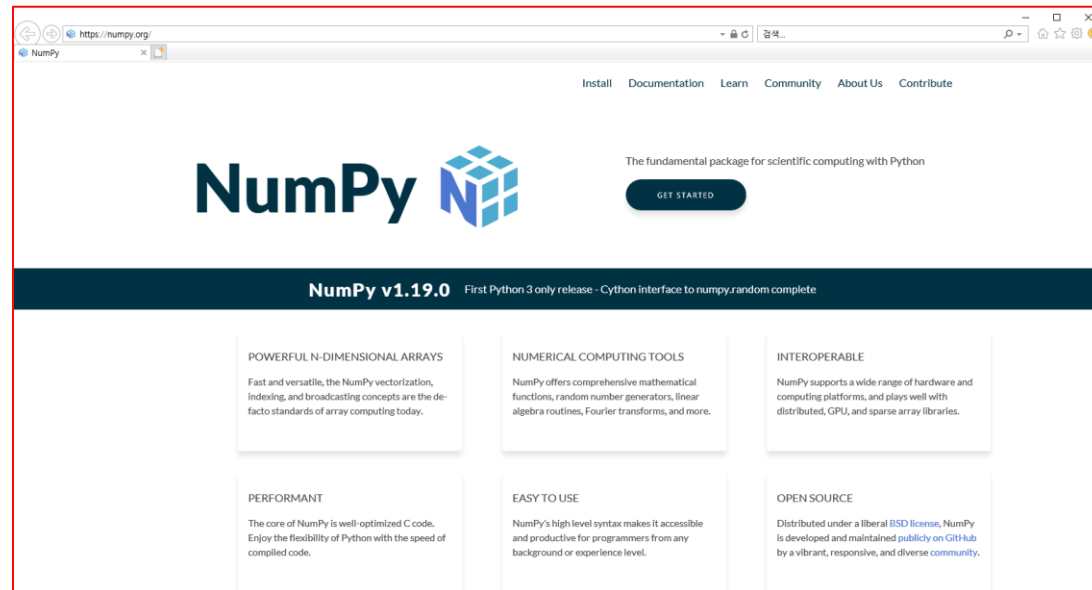


❖ numpy는 데이터 수치분석을 위한 python 패키지

- 행렬의 연산에 좋은 성능을 보임
- 핵심은 ndarray(다차원배열객체)
- 선형 대수, 푸리에(Fourier) 변환, 유사 난수 생성과 같은 유용한 함수들도 제공

❖ numpy의 공식 홈페이지

- <http://www.numpy.org/>



(1) ndarray 객체



❖ ndarray

- n차원 배열객체
- 같은 종류의 데이터만 배열에 담을 수 있음

❖ 용어

- axes : 차원의 번호
- rank : 차원의 개수
- shape : 배열의 차원을 나타내는 Tuple
- (예)

```
[[1.0, 0.0, 0.0],  
 [0.0, 1.0, 2.0]]
```

- rank : 2
- 첫번째 axis는 2행, 두번째 axis는 3열
- shape : (2,3)

ndarray의 속성



❖ *.ndim

- axes의 수 : rank # `a.ndim = len(a.shape)`

❖ *.shape

- 배열의 차원(dimension)을 나타내는 tuple
 - (예) n행, m열 => (n,m)

❖ *.size

- 배열요소의 총 개수

❖ *.dtype

- 자료형(`np.int32`, `np.int16`, `np.float64` 등)

❖ *.itemsize

- 각 요소의 크기(byte)



❖ (예)

```
import numpy as np

a = np.arange(15).reshape(3, 5)
print(a)
print("shape = ", a.shape)
print("ndim = ", a.ndim)
print("datatype = ", a.dtype)
print("itemsize =", a.itemsize)
print("size =", a.size)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
shape = (3, 5)
ndim = 2
datatype = int32
itemsize = 4
size = 15
```

- rank가 2인 배열(2차원 배열)이고...
- shape은 (3, 5)입니다.

(2) ndarray의 생성



❖ 생성방법

- `array(x)` : List x를 이용하여 배열 생성
 - 반드시 List를 초기값으로 주어야 함
- `arange(start, end, step)` : 순차값을 가진 배열 생성
 - 1개의 매개변수 n인 경우 0 ~ n-1 순으로 초기화됨
- `linspace(start, end, num)` : 순차값을 가진 배열을 num개 생성
 - end값 포함, 자동으로 step을 결정
- `zeros(s)` : 주어진 shape s에 따라 0값으로 초기화된 배열 생성
 - shape은 (n,m)과 같이 Tuple로 주어짐
- `ones(s)` : 주어진 shape s에 따라 1값으로 초기화된 배열 생성
- `empty(s)` : 주어진 shape s에 따라 초기화되지 않은 배열 생성
- `full(s, value)` : 주어진 shape s에 따라 value로 초기화된 배열 생성



- `random.randint(start, end)` : start부터 end-1사이의 정수 중에서 1개를 임의로 생성
 - 인수가 1개(end)만 있는 경우에는 0 ~ end-1 사이의 정수 1개 생성
- `random.rand(m, n)` : 균등분포인 m행 n열의 실수를 0 ~ 1사이에서 생성
 - 인수가 1개(m)만 있는 경우에는 m개의 난수 발생
- `random.randn(m, n)` : 가우시안 정규분포(평균이 0이고 분산이 1인)인 m행 n열의 실수를 생성
 - 인수가 1개(m)만 있는 경우에는 m개의 난수 발생
- `fromfunction(func, s)` : 주어진 shape s에 따라 func함수를 이용하여 생성. 단, 각 행과 열의 요소의 값은 index와 동일



❖ 예 : array()

```
import numpy as np

a = np.array([1, 2, 3, 4, 5])
b = np.array([[1, 2, 3, 4],
              [10, 20, 30, 40]])

print(a, "\n")
print(b)
```

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

```
[[ 1  2  3  4]
 [10 20 30 40]]
```



❖ 예 : arange()

```
import numpy as np

a = np.arange(5)
b = np.arange(10, dtype=np.float)
c = np.arange(3.1, 5, 0.25)

print(a, "\n")
print(b, "\n")
print(c)
```

```
[0 1 2 3 4]
```

```
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
```

```
[3.1 3.35 3.6 3.85 4.1 4.35 4.6 4.85]
```



❖ 예 : linspace()

```
import numpy as np  
  
a = np.linspace(0, 0.2, 6)  
  
print(a)
```

```
[0.    0.04 0.08 0.12 0.16 0.2 ]
```



❖ 예 : empty(), ones(), zeros(), full()

```
import numpy as np

a = np.empty((2, 3))
print(a, '\n')
b = np.ones((2, 3))
print(b, '\n')
c = np.zeros((2, 3))
print(c, '\n')
d = np.full((2, 3), np.pi)
print(d)
```

```
[[0.  0.04 0.08]
 [0.12 0.16 0.2 ]]
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
```

```
[[3.14159265 3.14159265 3.14159265]
 [3.14159265 3.14159265 3.14159265]]
```



❖ 예 : randint(), rand() randn()

```
import numpy as np

a = np.random.randint(10)
print(a, '\n')
b = np.random.randint(1, 10)
print(b, '\n')
c = np.random.rand(5)
print(c, '\n')
d = np.random.rand(3, 4)
print(d, '\n')
e = np.random.randn(3, 4)
print(e)
```

```
0
2
[0.5781477  0.31452747 0.00683025 0.93562899
 0.91016152]

[[0.71352295 0.9689287  0.14166002 0.23908673]
 [0.61533556 0.15867572 0.75578179 0.07109531]
 [0.1749837  0.63295633 0.6426747  0.67090928]]

[[-0.06546462 -0.75385209  0.56705616 -1.10309302]
 [ 1.33097749 -1.30430042  0.82065111 -2.95746286]
 [ 0.38655304  0.05706479 -0.06809105  0.27864592]]
```



❖ 예 : fromfunction()

```
import numpy as np

def my_func(x, y):
    return x + y

a = np.fromfunction(my_func, (3, 4))
print(a)
b = np.fromfunction(my_func, (3, 4), dtype=np.int)
print(b)
```

```
[[0. 1. 2. 3.]
 [1. 2. 3. 4.]
 [2. 3. 4. 5.]]
[[0 1 2 3]
 [1 2 3 4]
 [2 3 4 5]]
```

(3) 기본연산



❖ 요소간의 연산

- +, -, *, /, //, %, ** 등
- 배열의 크기는 같아야 함

```
import numpy as np

a = np.array([[20, 30],
              [40, 50]])
b = np.array([[1, 2],
              [3, 4]])

c = a + b
print(c, "\n")
d = a - b
print(d, "\n")
e = a * b    # elementwise product
print(e)
```

```
[[21 32]
 [43 54]]

[[19 28]
 [37 46]]

[[ 20  60]
 [120 200]]
```



❖ 행렬의 곱 : dot(A,B)

- 행렬의 곱셈은 A행렬 열의 크기와 B행렬 행의 크기가 같아야 함

```
import numpy as np

a = np.array([[20, 30],
              [40, 50]])
b = np.array([[1, 2],
              [3, 4]])
c = np.dot(a, b)
print(c)
```

```
[[110 160]
 [190 280]]
```


(4) ndarray의 reshape



❖ 방법

- shape 속성 변경(자기 수정)
- reshape() 사용

```
import numpy as np
```

```
a = np.arange(12)
```

```
print(a, '\n')
```

```
a.shape = (3, 4)
```

```
print(a, '\n')
```

```
b = a.reshape(2, 2, 3)
```

```
print(b)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

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

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



❖ ravel() - 실을 감다...

```
a =  
[[[ 0  1]  
 [ 2  3]  
 [ 4  5]  
 [ 6  7]]  
  
 [[ 8  9]  
 [10 11]  
 [12 13]  
 [14 15]]  
  
 [[16 17]  
 [18 19]  
 [20 21]  
 [22 23]]]
```

```
b =  
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
```

```
import numpy as np  
  
a = np.arange(24).reshape(3,4,2)  
print(f"a = \n{a}", '\n')  
  
b = a.ravel()  
print(f"b = \n{b}", '\n')
```

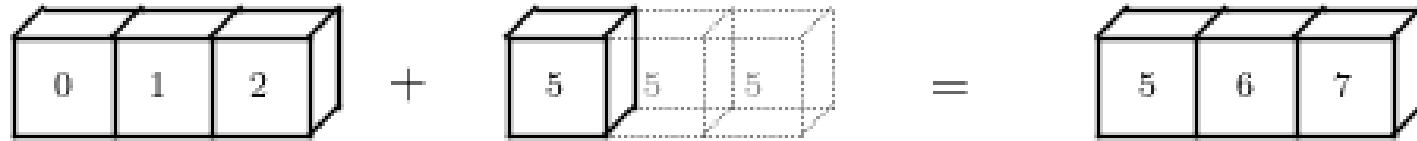
(5) Broadcasting



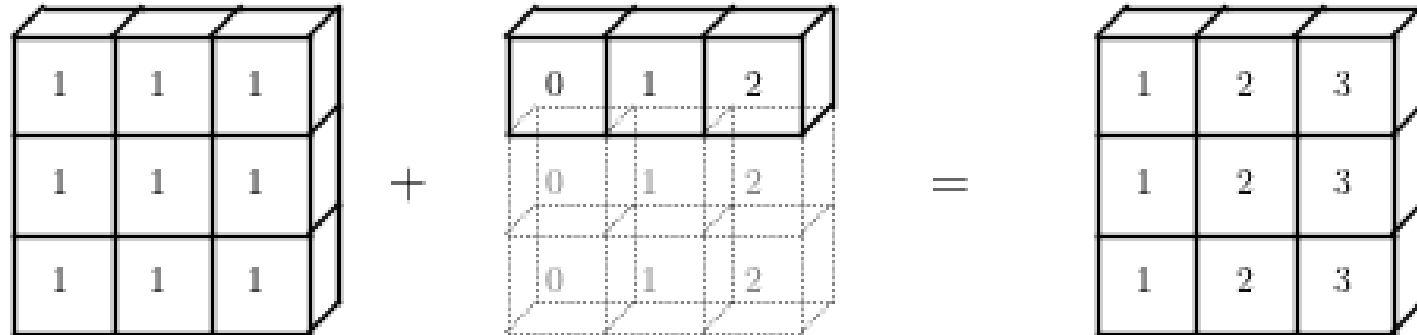
- ❖ 대부분의 연산시 동일한 크기의 ndarray를 기대함
- ❖ 만일, 크기가 일치하지 않고 일정 조건만 충족되면 broadcasting이 적용됨
 - 모양이 다른 배열끼리의 연산도 가능하게 해주며
 - 모양이 부족한 부분은 확장하여 연산을 수행
- ❖ Broadcasting 조건
 - 두 배열 중 하나의 배열이 1차원인 경우
 - 축의 크기가 동일한 경우



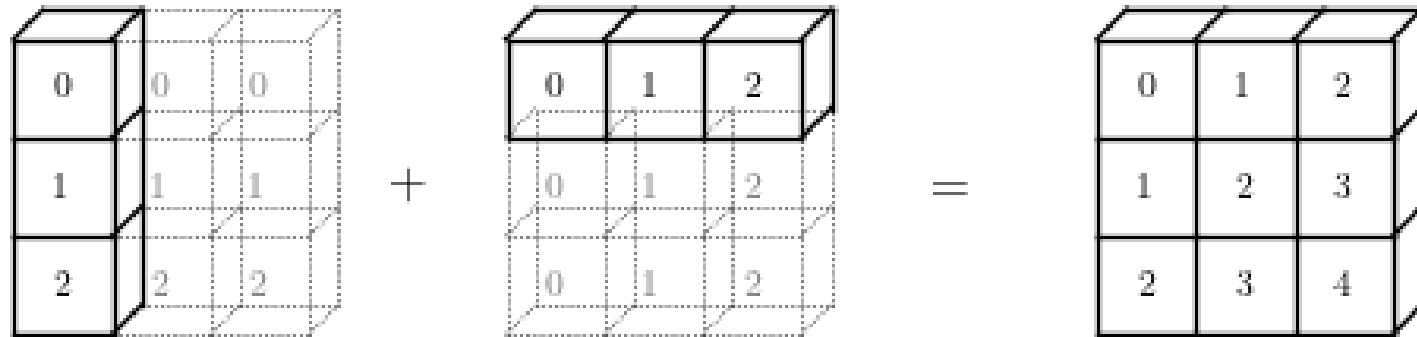
`np.arange(3) + 5`

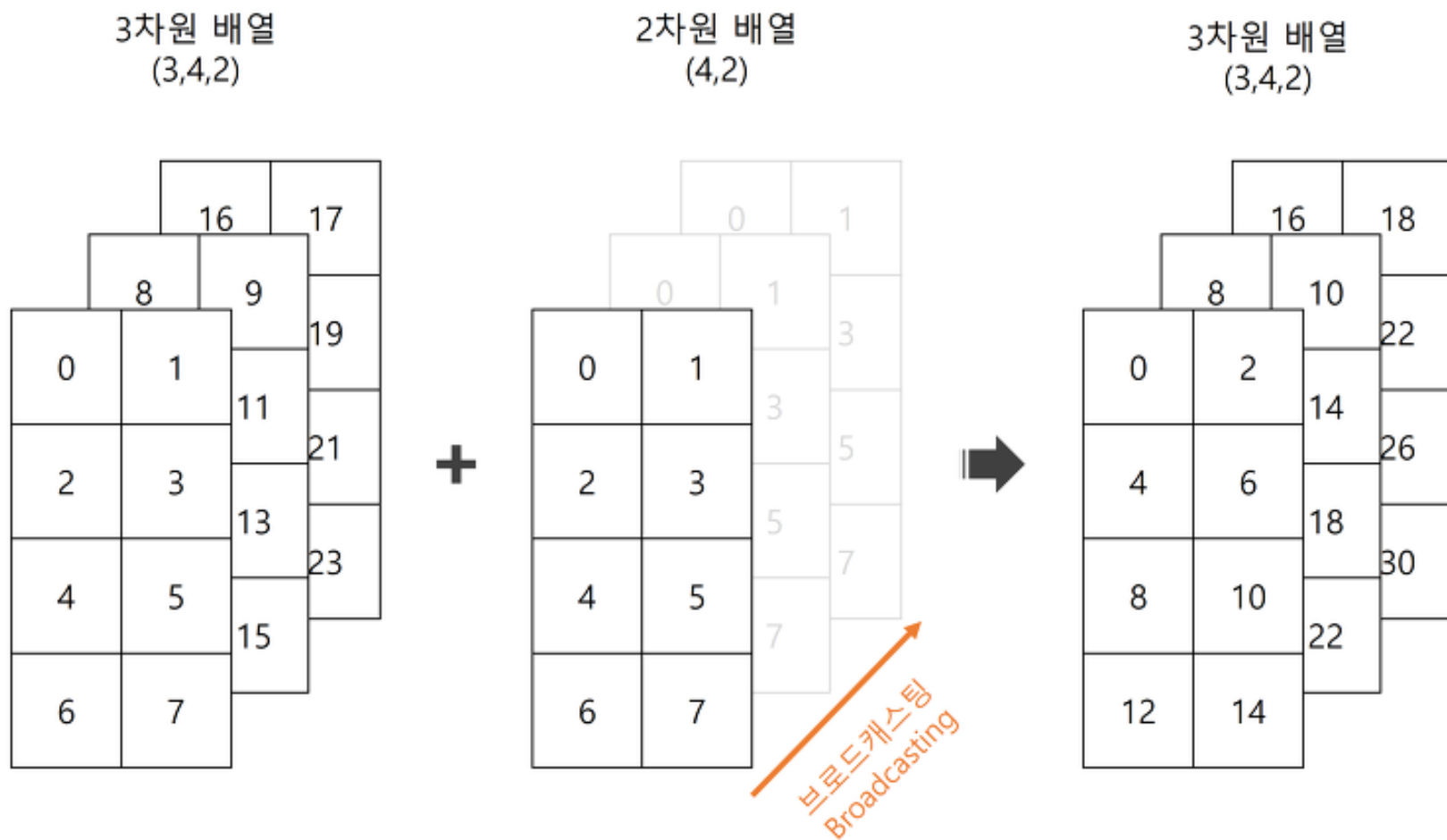


`np.ones((3, 3)) + np.arange(3)`



`np.arange(3).reshape((3, 1)) + np.arange(3)`





출처 : <https://sacko.tistory.com/16>



❖ 예 : 두 배열 중 하나의 배열이 1차원인 경우

```
import numpy as np  
  
a = np.arange(3)  
print(a, '\n')  
b = a + 5  
print(b)
```

`np.arange(3) + 5`



```
[0 1 2]
```

```
[5 6 7]
```



❖ 예 : 축의 크기가 동일한 경우

```
import numpy as np
```

```
a = np.ones((3, 3))
```

```
print(a, '\n')
```

```
b = np.arange(1, 4, dtype=float)
```

```
print(b, '\n')
```

```
c = a + b
```

```
print(c)
```

`np.ones((3, 3)) + np.arange(3)`

1	1	1
1	1	1
1	1	1

+

0	1	2
0	1	2
0	1	2

=

1	2	3
1	2	3
1	2	3

```
[[1. 1. 1.]  
 [1. 1. 1.]  
 [1. 1. 1.]]
```

```
[1. 2. 3.]
```

```
[[2. 3. 4.]  
 [2. 3. 4.]  
 [2. 3. 4.]]
```

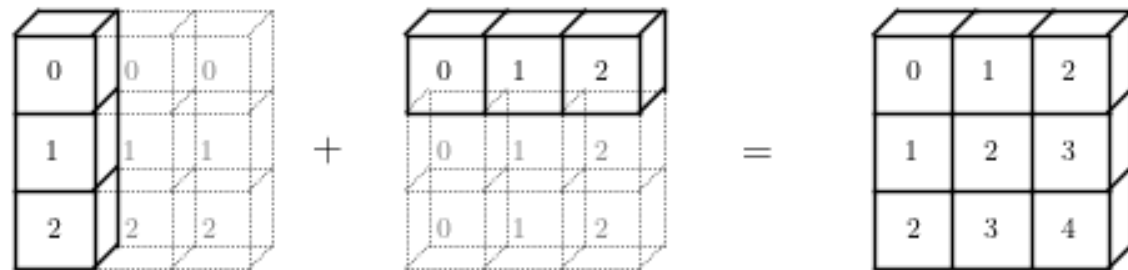


❖ 예 : 축의 크기가 동일한 경우

```
import numpy as np

a = np.arange(3).reshape(3,1)
print(a, '\n')
b = np.arange(3) # (1,3)
print(b, '\n')
c = a + b # (3,3)
print(c)
```

`np.arange(3).reshape((3, 1)) + np.arange(3)`



```
[[0]
 [1]
 [2]]

[0 1 2]

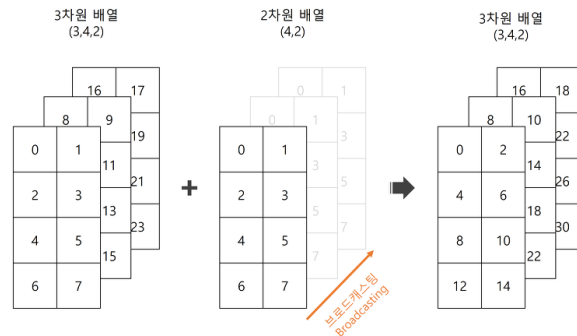
[[0 1 2]
 [1 2 3]
 [2 3 4]]
```




❖ 예 : 행, 열 축의 크기가 동일한 경우

```
import numpy as np

a = np.arange(24).reshape(3,4,2)
print(f"a = \n{a}", '\n')
b = np.arange(8).reshape(4,2)
print(f"b = \n{b}", '\n')
c = a + b                                     # (3,4,2)
print(f"a + b = \n{c}")
```



```
a =
[[[ 0  1]
  [ 2  3]
  [ 4  5]
  [ 6  7]]
```

```
[[ 8  9]
 [10 11]
 [12 13]
 [14 15]]
```

```
[[16 17]
 [18 19]
 [20 21]
 [22 23]]]
```

```
b =
[[0 1]
 [2 3]
 [4 5]
 [6 7]
```

```
a + b =
[[[ 0  2]
  [ 4  6]
  [ 8 10]
  [12 14]]
```

```
[[ 8 10]
 [12 14]
 [16 18]
 [20 22]]
```

```
[[16 18]
 [20 22]
 [24 26]
 [28 30]]]
```



❖ 예 :

```
import numpy as np

a = np.arange(5).reshape(1,1,5)    # (1, 1, 5)
print(a, '\n')
b = a + 5
print(b, '\n')
c = np.arange(5).reshape(5,1)      # (5, 1)
print(c, '\n')
d = a + np.arange(5).reshape(5,1)  # (1, 5, 5)
print(d)
```

```
[[[0 1 2 3 4]]]
```

```
[[[5 6 7 8 9]]]
```

```
[[0]
 [1]
 [2]
 [3]
 [4]]
```

```
[[[0 1 2 3 4]
 [1 2 3 4 5]
 [2 3 4 5 6]
 [3 4 5 6 7]
 [4 5 6 7 8]]]
```

(6) Indexing과 Slicing



❖ Indexing

- python의 인덱싱과 비슷함
- index는 0부터 시작
- index생략시 all을 의미

❖ Slicing

- python의 슬라이싱과 비슷함

❖ indexing/slicing rule

- **[i:j:k]**
 - i:start, j:end(j-1위치), k:step



❖ 1차원 Indexing

```
import numpy as np
```

```
a = np.arange(10) # 0 ~ 9
```

```
print(a[3]) # 3번째 있는 원소
```

```
print(a[:]) # 전체 원소 출력
```

```
print(a[:3]) # 처음부터 3번 원소 앞까지
```

```
print(a[:-1]) # 처음부터 마지막-1 까지
```

```
# 인덱스 2부터 2개씩(step)
```

```
print(a[2::2])
```

```
# 인덱스 7부터 2번 원소 앞까지 -1씩(step)
```

```
print(a[7:2:-1])
```

```
# 마지막부터 처음까지 -1씩(step)
```

```
print(a[::-1])
```

→ 3

→ [0 1 2 3 4 5 6 7 8 9]

→ [0 1 2]

→ [0 1 2 3 4 5 6 7 8]

→ [2 4 6 8]

→ [7 6 5 4 3]

→ [9 8 7 6 5 4 3 2 1 0]



❖ 1차원 Slicing

```
import numpy as np

a = np.arange(10)    # 0 ~ 9
print(a)
s = a[5:8]           # 주의: referencing
print(s)             # [5 6 7]
s[:] = 500           # 전체 수정(broadcasting)
print(s)
print(a)
```

```
[0 1 2 3 4 5 6 7 8 9]
[5 6 7]
[500 500 500]
[ 0  1  2  3  4 500 500 500  8  9]
```



❖ 2차원 Indexing & Slicing

```
import numpy as np

a = np.arange(20).reshape((5,4))
print(a)
print(f"a[2,3] = {a[2,3]}")
print(f"a[0:5, 1] = {a[0:5, 1]}")
print(f"a[:, 1] = {a[:, 1]}")
print(f"a[1:3, :]=\n{a[1:3, :]}")

print(f"a[-1] = {a[-1]}")
print(f"a[:, -1] = {a[:, -1]}")
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]]
a[2,3] = 11
a[0:5, 1] = [ 1  5  9 13 17]
a[:, 1] = [ 1  5  9 13 17]
a[1:3, :]=
[[ 4  5  6  7]
 [ 8  9 10 11]]
a[-1] = [16 17 18 19]
a[:, -1] = [ 3  7 11 15 19]
```



❖ `copy()` : ndarray의 깊은 복사

```
import numpy as np

a = np.arange(0, 3.6, 0.3).reshape((3,4))
print("a = \n", a)
b = a.copy()
print("copy of a = \n", b)
```

```
a =
[[ 0.   0.3  0.6  0.9]
 [ 1.2  1.5  1.8  2.1]
 [ 2.4  2.7  3.   3.3]]
copy of a =
[[ 0.   0.3  0.6  0.9]
 [ 1.2  1.5  1.8  2.1]
 [ 2.4  2.7  3.   3.3]]
```



❖ fancy indexing

```
import numpy as np

a = np.arange(20).reshape((5,4))
print(a, '\n')

b = a[(0,2,3), 2] # fancy indexing
print(b)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]]

[ 2 10 14]
```




❖ boolean indexing

- 하나의 축에만 적용 가능

```
import numpy as np
```

```
a = np.arange(20).reshape((5,4))  
print(a, '\n')
```

```
rows = np.array([True, False, True, True, False])  
b = a[rows, :] # boolean indexing  
print(b)
```

```
[[ 0  1  2  3]  
 [ 4  5  6  7]  
 [ 8  9 10 11]  
 [12 13 14 15]  
 [16 17 18 19]]
```

```
[[ 0  1  2  3]  
 [ 8  9 10 11]  
 [12 13 14 15]]
```



❖ 반복을 이용한 item추출

■ item 추출(1)

```
import numpy as np

a = np.arange(24).reshape(2, 3, 4)
print(f"a =\n{a}")

for item in a:
    print(f"item =\n{item}")
```

```
a =
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]

item =
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

item =
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```



❖ 반복을 이용한 item추출

- item 추출(2)

```
import numpy as np

a = np.arange(24).reshape(2, 3, 4)
print(f"a =\n{a}")

for i in range(len(a)):
    print(f"a[{i}] =\n{a[i]}")
```

```
a =
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]

a[0] =
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

a[1] =
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```



❖ 반복을 이용한 item추출

- item 추출(3)
 - flat를 이용하여 모든 item을 추출

```
import numpy as np

a = np.arange(24).reshape(2, 3, 4)
print(f"a =\n{a}")

for item in a.flat:
    print(item, end=' ')
```

```
a =
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
0 1 2 3 4 5 6 7 8 9 10 11 12 13
14 15 16 17 18 19 20 21 22 23
```

(7) 배열 쌓기와 분리하기



❖ 배열 쌓기

- `vstack()` : 수직으로 배열 연결
- `hstack()` : 수평으로 배열 연결
- 주의: 수직 또는 수평의 크기가 같아야 함

❖ 배열 분리하기

- `vsplit()` : 수직으로 배열 분리
- `hsplit()` : 수평으로 배열 분리
- 주의: 분리시에 수직 또는 수평으로 동일한 크기로 분리될 수 있어야 함



❖ vstack()

```
import numpy as np

a = np.arange(8, dtype=float).reshape(2, 4)
b = np.full((3, 4), 2.0)
c = np.ones((2, 4))
print(f"a =\n{a}")
print(f"b =\n{b}")
print(f"c =\n{c}")

d = np.vstack((a, b, c)) # tuple
print(f"d =\n{d}")
```

```
a =
[[0. 1. 2. 3.]
 [4. 5. 6. 7.]]
b =
[[2. 2. 2. 2.]
 [2. 2. 2. 2.]
 [2. 2. 2. 2.]]
c =
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]]
d =
[[0. 1. 2. 3.]
 [4. 5. 6. 7.]
 [2. 2. 2. 2.]
 [2. 2. 2. 2.]
 [2. 2. 2. 2.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```



❖ hstack()

```
import numpy as np

a = np.arange(8, dtype=float).reshape(2, 4)
b = np.full((3, 4), 2.0)
c = np.ones((2, 4))
print(f"a =\n{a}")
print(f"b =\n{b}")
print(f"c =\n{c}")

d = np.hstack((a, c)) # tuple
print(f"d =\n{d}")
```

- b는 차수가 달라 연결할 수 없음.

```
a =
[[0. 1. 2. 3.]
 [4. 5. 6. 7.]]
b =
[[2. 2. 2. 2.]
 [2. 2. 2. 2.]
 [2. 2. 2. 2.]]
c =
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]]
d =
[[0. 1. 2. 3. 1. 1. 1. 1.]
 [4. 5. 6. 7. 1. 1. 1. 1.]]
```



❖ vsplit()

```
import numpy as np

a = np.arange(24, dtype=float).reshape(6, 4)
print(f"a =\n{a}")

v1, v2 = np.vsplit(a, 2)
print(f"v1 =\n{v1}")
print(f"v2 =\n{v2}")
```

```
a =
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]
 [12. 13. 14. 15.]
 [16. 17. 18. 19.]
 [20. 21. 22. 23.]]

v1 =
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]


v2 =
[[12. 13. 14. 15.]
 [16. 17. 18. 19.]
 [20. 21. 22. 23.]]
```


❖ `hsplit()`

```
import numpy as np

a = np.arange(24, dtype=float).reshape(6, 4)
print(f"a =\n{a}")

h1, h2 = np.hsplit(a, 2)
print(f"h1 =\n{h1}")
print(f"h2 =\n{h2}")
```



```
a =
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]
 [12. 13. 14. 15.]
 [16. 17. 18. 19.]
 [20. 21. 22. 23.]]

h1 =
[[ 0.  1.]
 [ 4.  5.]
 [ 8.  9.]
 [12. 13.]
 [16. 17.]
 [20. 21.]]

h2 =
[[ 2.  3.]
 [ 6.  7.]
 [10. 11.]
 [14. 15.]
 [18. 19.]
 [22. 23.]]
```

(8) 선형대수 관련 연산



❖ 행렬의 곱셈

- `np.dot(A, B)` - 앞에서 다루었음

❖ 단위행렬 만들기

- `np.eye(n)` - $n \times n$ 의 단위행렬

❖ 전치행렬

- `np.transpose(a, s)`

❖ 대각원소와 대각합

- `np.diag()`, `np.trace()`

❖ 역행렬과 행렬식

- `np.linalg.inv()`, `np.linalg.det()`

❖ 고유값과 고유벡터

- `np.linalg.eig()`



❖ 단위행렬 만들기

- `np.eye(n)` : $n \times n$ 의 단위행렬 생성
- 행렬과 그 행렬의 역행렬을 곱하면 단위행렬이 됨
- `np.linalg.inv(a)` : a 의 역행렬

```
import numpy as np
import numpy.linalg as npl

a = np.eye(3)
print("a = \n", a)

b = np.array([[1,3], [2,4]])
print("b = \n", b)
print("inverse matrix of b = \n", npl.inv(b))
```

```
a =
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
b =
[[1 3]
 [2 4]]
inverse matrix of b =
[[-2.  1.5]
 [ 1. -0.5]]
```



❖ 전치행렬

■ a.T

```
import numpy as np

a = np.arange(12).reshape(3, 4)
b = np.arange(24).reshape(2, 4, 3)
print(a, '\n')
print(a.T, '\n')
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
[[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
```

- 3차원 행렬에 적용하면
 - 0, 1, 2번 축이 2, 1, 0번 축으로 전치됨

❖ 전치행렬

- `np.transpose(a[, s])`

```
import numpy as np

a = np.arange(24).reshape(2, 4, 3)
print(a, '\n')
print(np.transpose(b, (2, 0, 1))) # (3, 2, 4)
```

- a의 shape이 (2, 4, 3)이고
- a의 축번호는 (0, 1, 2)임. transpose에 의해
- a의 축번호가 (2, 0, 1)로 바뀜.
즉, **열->면**, **면->행**, **행->열**로 바뀌게 됨
- 따라서 shape (3, 2, 4)로 바뀌게 됨

```
[[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

```
[[12 13 14]
 [15 16 17]
 [18 19 20]
 [21 22 23]]]
```

```
[[[ 0  3  6  9]
 [12 15 18 21]]
```

```
[[ 1  4  7 10]
 [13 16 19 22]]
```

```
[[ 2  5  8 11]
 [14 17 20 23]]]
```



❖ 대각원소와 대각합

- `np.diag()`
- `np.trace()`

```
import numpy as np

a = np.arange(1, 10).reshape(3, 3)
print(a, '\n')
b = np.diag(a)
print(b, '\n')
c = np.trace(a)
print(c)
```

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

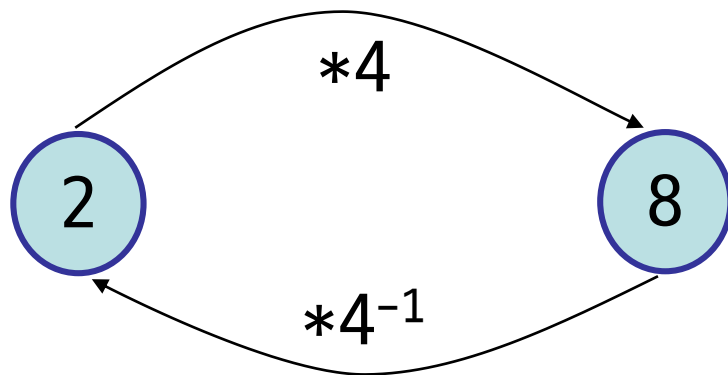
```
[1 5 9]
```

```
15
```

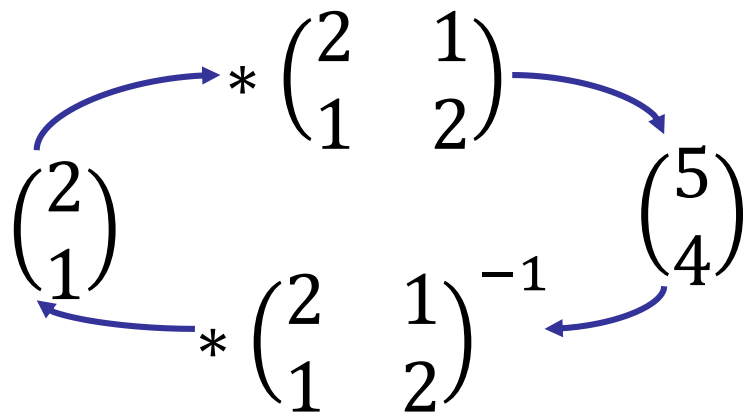


❖ 역행렬 (A^{-1} : inverse matrix)

- 행렬 A 와 곱하면 단위행렬 I 가 나오는 행렬 A^{-1} 를 역행렬이라고 함
- $A^{-1}A = AA^{-1} = I$



$$4^{-1} * 4 = 1$$



$$\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}^{-1} * \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I$$



❖ 역행렬 (A^{-1} : inverse matrix)

- 정방행렬에 대해서만 정의됨
- 역행렬이 없으면 특이행렬 (singular matrix)

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad A^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

- (예)

$$A = \begin{pmatrix} 2 & 1 \\ 6 & 4 \end{pmatrix}$$

$$A^{-1} = \frac{1}{2 \cdot 4 - 1 \cdot 6} \begin{pmatrix} 4 & -1 \\ -6 & 2 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 4 & -1 \\ -6 & 2 \end{pmatrix} = \begin{pmatrix} 2 & -0.5 \\ -3 & 1 \end{pmatrix}$$



❖ Determinant(행렬식)

- 어떤 행렬의 역행렬 존재여부에 대한 판별값

- det의 값이 0이면 역행렬 없음

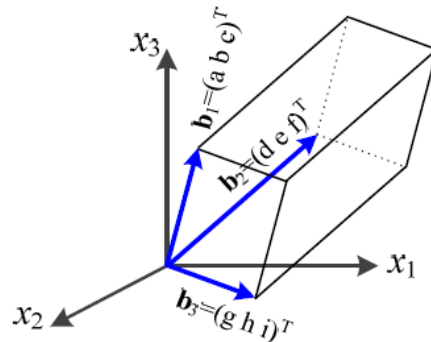
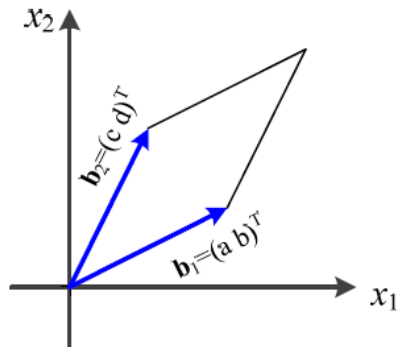
- 정방행렬에 대해서만 정의됨

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad \det(A) = ad - bc$$

- 기하학적 의미

- 2차원 : 2개의 행벡터가 이루는 평행사변형의 넓이

- 3차원 : 3개의 행벡터가 이루는 평행사각기둥의 부피





❖ 역행렬과 행렬식

- `linalg.inv()`
- `linalg.det()`

```
import numpy as np

a = np.array([[ 1,  2,  3],
               [ 5,  7, 11],
               [21, 29, 31]])

print(a, '\n')
b = np.linalg.inv(a)
print(b, '\n')
c = np.linalg.det(a)
print(c)
```

```
[[ 1  2  3]
 [ 5  7 11]
 [21 29 31]]

[[-2.31818182  0.56818182  0.02272727]
 [ 1.72727273 -0.72727273  0.09090909]
 [-0.04545455  0.29545455 -0.06818182]]

43.99999999999997
```

참고: 고유벡터와 고유값



❖ $A\mathbf{v} = \lambda\mathbf{v}$

- \mathbf{v} : 고유벡터(eigen vector: 위 식을 만족하는 0이 아닌 벡터)
- λ : 고유값(eigen value)
 - $m \times m$ 행렬은 최대 m 개의 고유값과 고유벡터를 가질 수 있음
 - 모든 고유벡터는 서로 직교(orthogonal). 길이는 고유값에 따름

❖ (예) $A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$

$$\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = 3 \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$\lambda = 3, \mathbf{v} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \end{pmatrix} = 1 \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

$$\lambda = 1, \mathbf{v} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

참고: 고유벡터와 고유값의 기하학적 의미



❖ 반지름 1인 원에 있는 4개의 벡터

$$\blacksquare x_1 = \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix}, x_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, x_3 = \begin{pmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix}, x_4 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

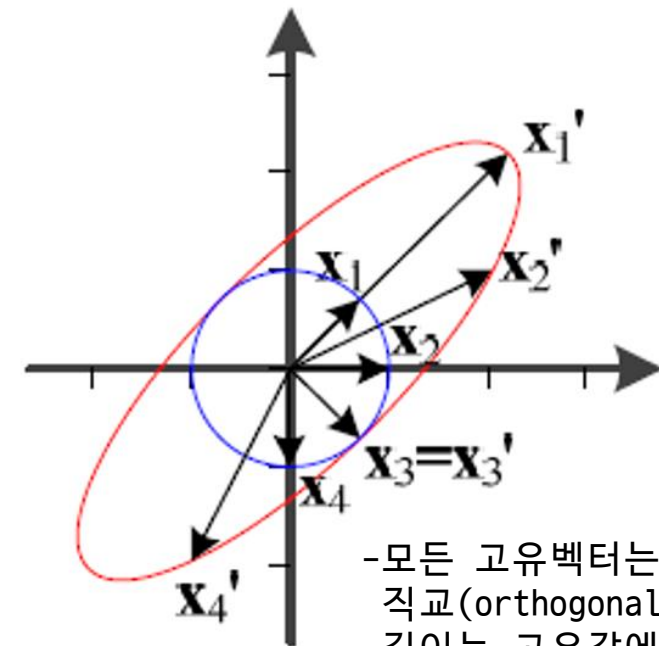
❖ $A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$ 로 변환

$$\blacksquare x'_1 = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix} = 3 \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix}$$

$$\blacksquare x'_2 = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

$$\blacksquare x'_3 = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix} = 1 \begin{pmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix}$$

$$\blacksquare x'_4 = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 0 \\ -1 \end{pmatrix} = \begin{pmatrix} -1 \\ -2 \end{pmatrix}$$



-모든 고유벡터는 서로
직교(orthogonal)
-길이는 고유값에 따름

❖ 파란색 원이 빨간색 타원으로 바뀌더라도
방향이 바뀌지 않는 것은 고유벡터인 x_1 과 x_3 뿐이다.



❖ 고유값과 고유벡터

- `linalg.eig(a)`

```
import numpy as np
```

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

```
print(f"a =\n{a}")
```

```
eigenvalues, eigenvectors = np.linalg.eig(a)
```

```
print(f"eigenvalues =\n{eigenvalues}")
```

```
print(f"eigenvectors =\n{eigenvectors}")
```

```
a =  
[[4 2]  
 [3 5]]  
eigenvalues =  
[2. 7.]  
eigenvectors =  
[[-0.70710678 -0.5547002 ]  
 [ 0.70710678 -0.83205029]]
```

2

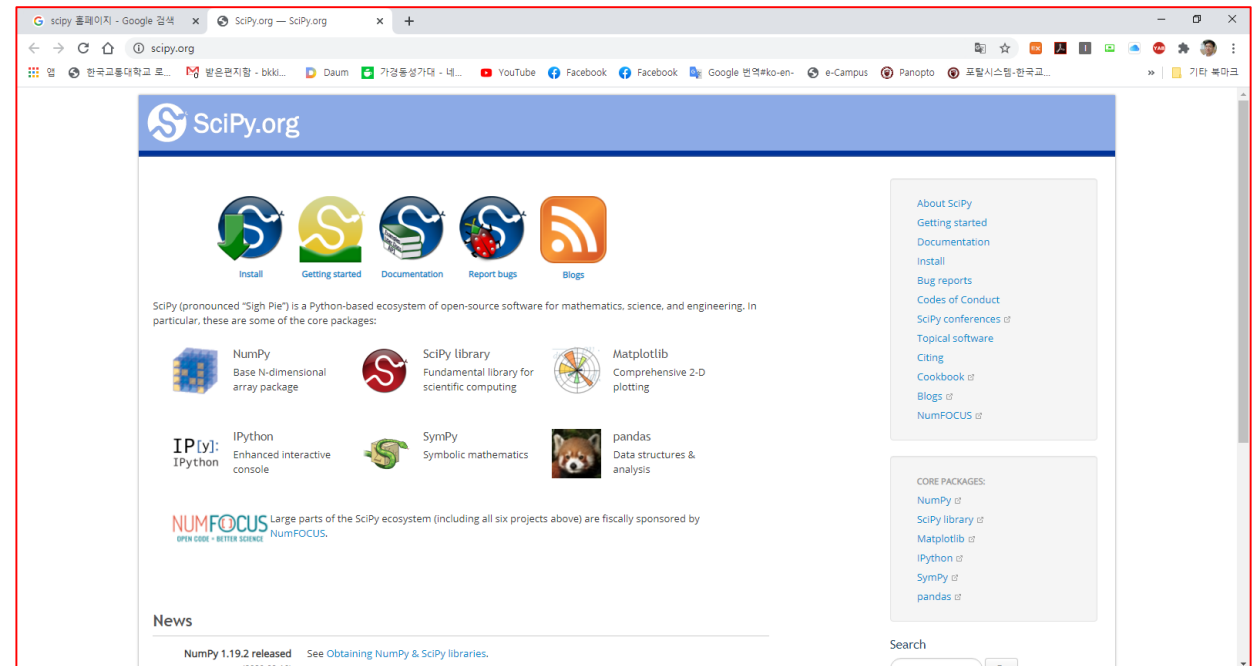
scipy



scipy 라이브러리



- ❖ 과학기술계산을 위한 Python 라이브러리
- ❖ numpy, matplotlib, pandas 등과 연계되어 사용됨
 - numpy의 상위 라이브러리로 이해해도 됨
- ❖ 설치
 - Anaconda 사용자는 별도의 설치 필요 없음
 - cmd창에서 설치방법
 - `C:\> pip install scipy`
- ❖ scipy 홈페이지
 - <https://www.scipy.org/>



가능한 알고리즘과 패키지



❖ 주요 서브패키지

- Linear algebra (`scipy.linalg`)
- Statistical functions (`scipy.stats`)
- Optimization and root finding (`scipy.optimize`)
- Integration and ODEs (`scipy.integrate`)
- Interpolation (`scipy.interpolate`)
- Sparse matrices (`scipy.sparse`)
- Sparse linear algebra (`scipy.sparse.linalg`)
- Spatial algorithms and data structures (`scipy.spatial`)
- Special functions (`scipy.special`)
- Discrete Fourier transforms (`scipy.fftpack`)



❖ **sciPy는 기본적으로 numpy의 ndarray를 기본 자료형으로 사용**

❖ **주요 연산**

- 선형대수(역함수, 행렬식, 고유값, 고유벡터 등)
- 연립방정식의 해
- 비선형방정식의 해
- 함수의 최소값
- 수치적분
- 선형회귀
- 비선형회귀



❖ 역함수, 행렬식

```
[[ 1  2  3]
 [ 5  7 11]
 [21 29 31]]
```

det(a) = 44.0

inv(a) =

```
[[ -2.31818182  0.56818182  0.02272727]
 [  1.72727273 -0.72727273  0.09090909]
 [ -0.04545455  0.29545455 -0.06818182]]
```

a * inv(a) =

```
[[ 1.00000000e+00  0.00000000e+00 -2.77555756e-17]
 [-1.33226763e-15  1.00000000e+00 -1.11022302e-16]
 [-4.21884749e-15  1.77635684e-15  1.00000000e+00]]
```

```
import numpy as np
from scipy.linalg import *

a = np.array([[ 1,  2,  3],
               [ 5,  7, 11],
               [21, 29, 31]])

print(a, '\n')
print(f"det(a) = {det(a)}")
print(f"inv(a) =\n{inv(a)}")
print(f"a * inv(a) =\n{np.dot(a, inv(a))}")
```



❖ 고유값과 고유벡터

- `linalg.eig(a)`

```
import numpy as np
from scipy.linalg import *
```

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

```
print(f"a =\n{a}")
```

```
eigenvalues, eigenvectors = eig(a)
print(f"eigenvalues =\n{eigenvalues}")
print(f"eigenvectors =\n{eigenvectors}")
```

```
a =
[[4 2]
 [3 5]]
eigenvalues =
[2.+0.j 7.+0.j]
eigenvectors =
[[-0.70710678 -0.5547002 ]
 [ 0.70710678 -0.83205029]]
```



❖ 연립방정식 풀기

$$\begin{aligned} 3x + 2y &= 2 \\ x - y &= 4 \\ 5y + z &= -1 \end{aligned}$$

$$\begin{pmatrix} 3 & 2 & 0 \\ 1 & -1 & 0 \\ 0 & 5 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 4 \\ -1 \end{pmatrix}$$

```
import numpy as np
from scipy.linalg import *

A = np.array([[3, 2, 0], [1, -1, 0], [0, 5, 1]])
B = np.array([2, 4, -1]) # 열벡터
s = solve(A, B)
print(f"(x, y, z) = {s}")
print(f"A * s = {np.dot(A, s)}")
```

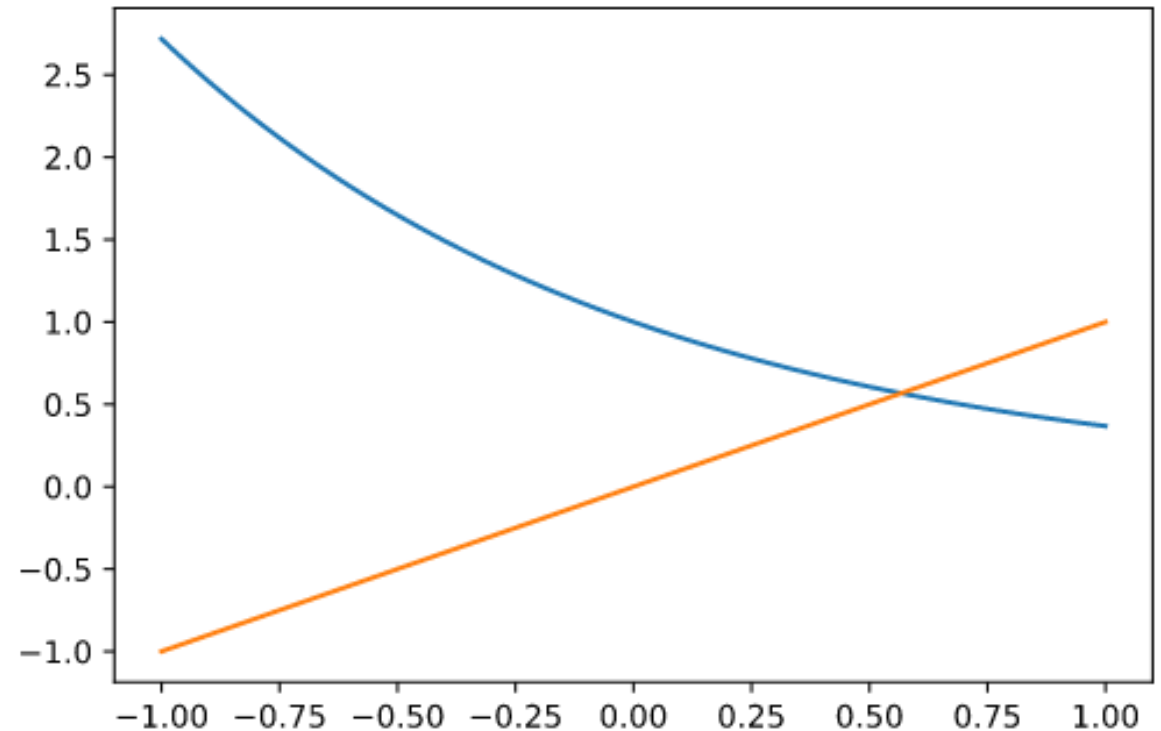
```
(x, y, z) = [ 2. -2.  9.]
A * s = [ 2.  4. -1.]
```



❖ 비선형방정식의 해

- (예) 방정식 $x = e^{-x}$
 - matplotlib을 이용하여 plotting

```
import matplotlib.pyplot as plt  
  
x = np.linspace(-1,1,101)  
  
plt.plot(x, np.exp(-x))  
plt.plot(x, x)  
plt.show()
```



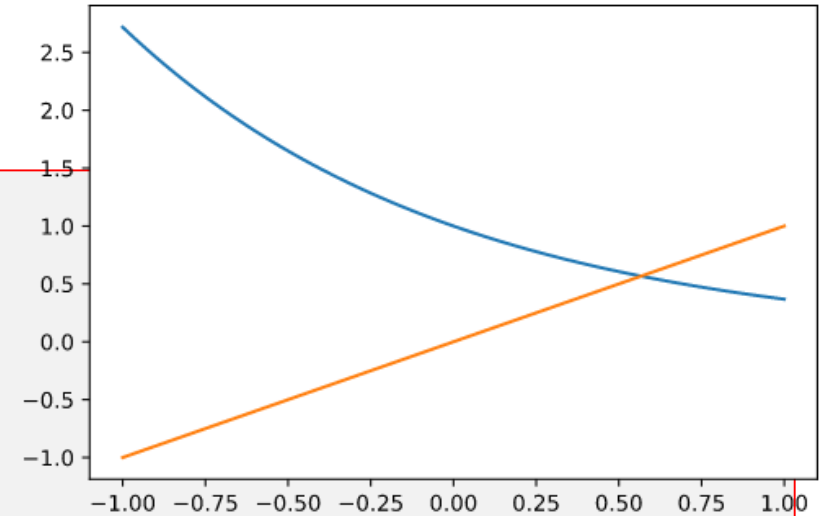


- (예)방정식 $x = e^{-x}$

```
import math
import numpy as np
from scipy.optimize import fsolve
```

```
def f(x):
    return x - math.exp(-x)           #  $x - \exp(-x) = 0$ 
```

```
print(fsolve(f, 0.5))
```



```
[0.56714329]
```



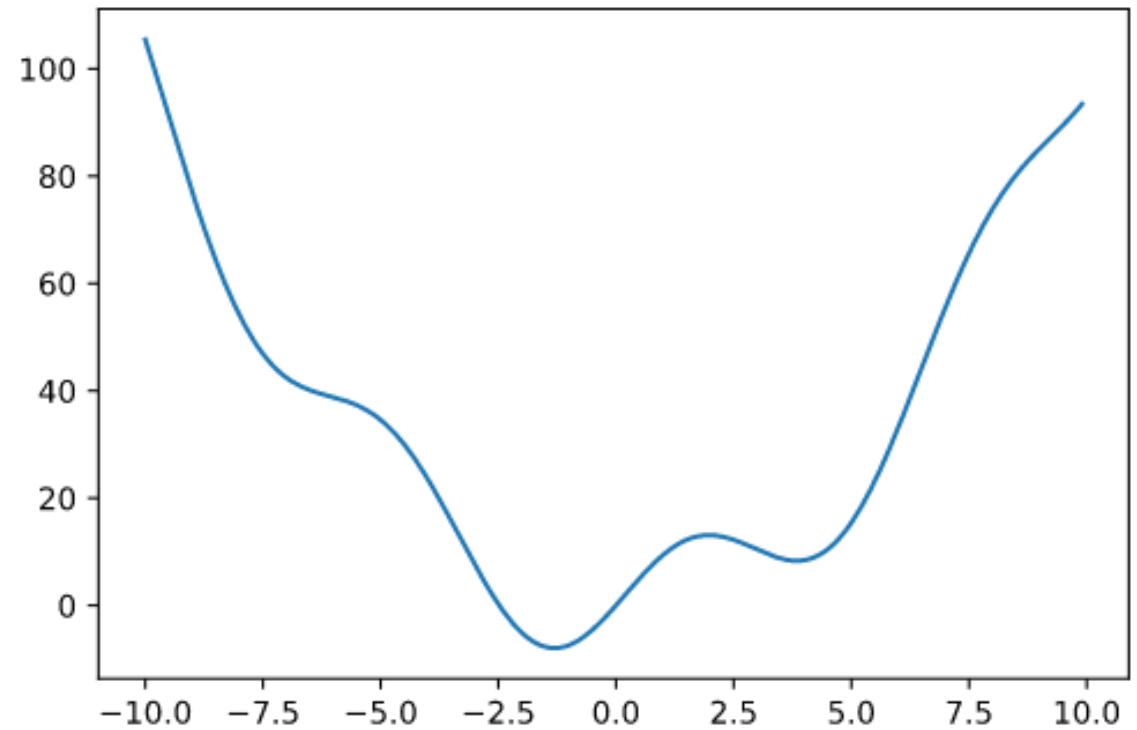
❖ 함수의 최소값

- (예) $f(x) = x^2 + 10 \sin(x)$
 - matplotlib을 이용하여 plotting

```
import matplotlib.pyplot as plt

def f(x):
    return x**2 + 10*np.sin(x)

x = np.arange(-10, 10, 0.1)
plt.plot(x, f(x))
plt.show()
```



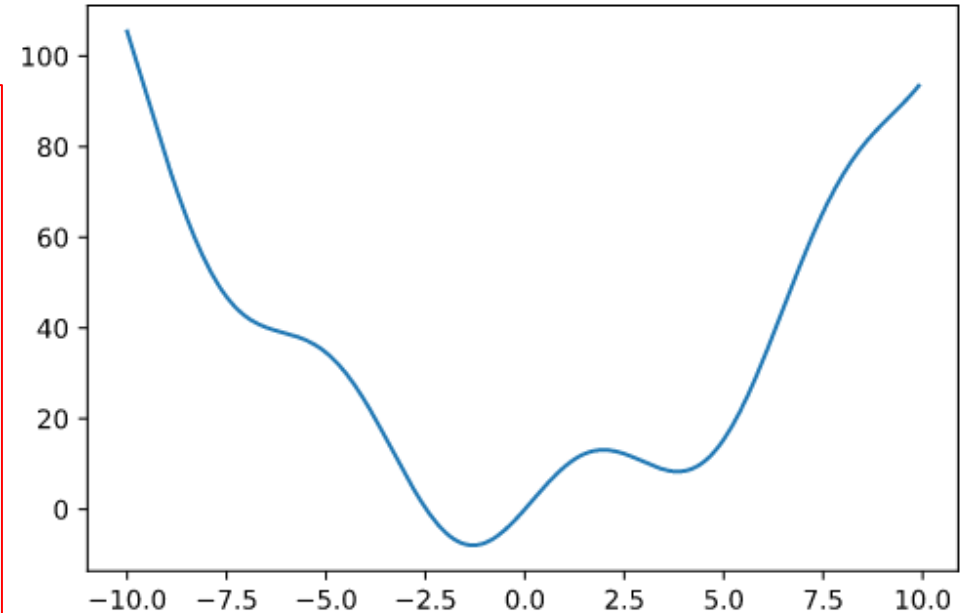


- (예) $f(x) = x^2 + 10 \sin(x)$

```
import numpy as np
from scipy.optimize import fmin_bfgs

def f(x):
    return x**2 + 10*np.sin(x)

print(fmin_bfgs(f, 0)) # 0에서 시작해서 탐색
print(fmin_bfgs(f, 6)) # 6에서 시작해서 탐색
```



```
Optimization terminated successfully.
      Current function value: -7.945823
      Iterations: 5
      Function evaluations: 18
      Gradient evaluations: 6
[-1.30644012]
```

```
Optimization terminated successfully.
      Current function value: 8.315586
      Iterations: 7
      Function evaluations: 24
      Gradient evaluations: 8
[3.83746709]
```




❖ 수치적분

■ (예)

$$y = \int_0^1 x^2 dx \qquad y = \int_0^{\infty} e^{-x} dx$$

```
import numpy as np
from scipy.integrate import quad

f1 = lambda x: x**2
f2 = lambda x: np.exp(-x)
print(quad(f1, 0, 1))          # return (적분값, 추정오차)
print(quad(f2, 0, np.inf))
```

```
(0.33333333333333337, 3.700743415417189e-15)
(1.0000000000000002, 5.842606703608969e-11)
```

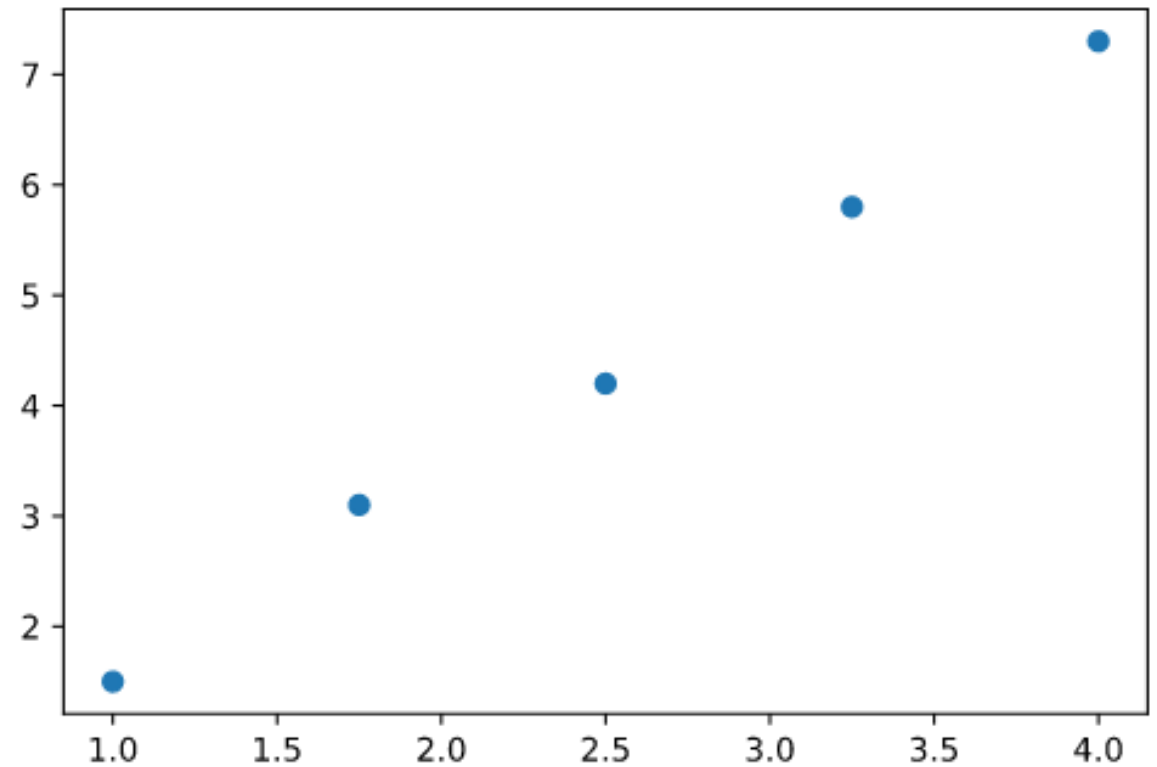


❖ 선형회귀

- matplotlib을 이용하여 plotting

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(1, 4, 5)
y = [1.5, 3.1, 4.2, 5.8, 7.3]
plt.plot(x, y, 'o')
plt.show()
```



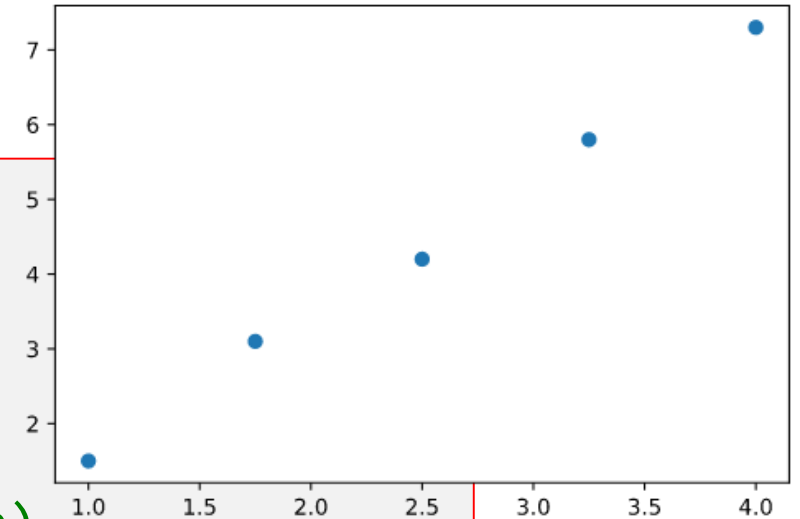


■ 기울기와 절편 구하기

```
import numpy as np
from scipy.stats import linregress

x = np.linspace(0, 4, 5)
y = [1.5, 3.1, 4.2, 5.8, 7.3]
# 기울기(slope), 절편(intercept), 상관계수(r_value)
# 예측 불확실성(p_value), 표준편차(std_err)
slope, intercept, r_value, p_value, std_err = linregress(x, y)

print(f"slope = {slope:.3f}")
print(f"intercept = {intercept:.3f}")
```



```
slope = 1.430
intercept = 1.520
```



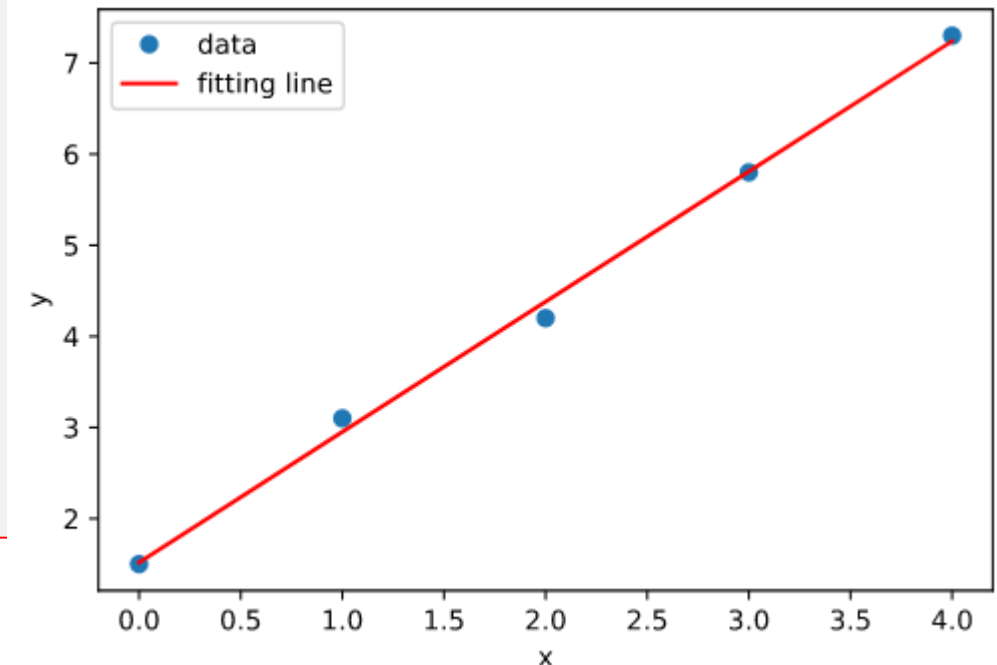
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import linregress
```

```
x = np.linspace(0, 4, 5)
y = [1.5, 3.1, 4.2, 5.8, 7.3]
slope, intercept, r_value, p_value, std_err = linregress(x, y)
print(f"slope = {slope:.3f}")
print(f"intercept = {intercept:.3f}")
```

```
yfit = slope * x + intercept
```

```
plt.plot(x, y, 'o', label='data')
plt.plot(x, yfit, 'r', label='fitting line')
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

slope = 1.430
intercept = 1.520

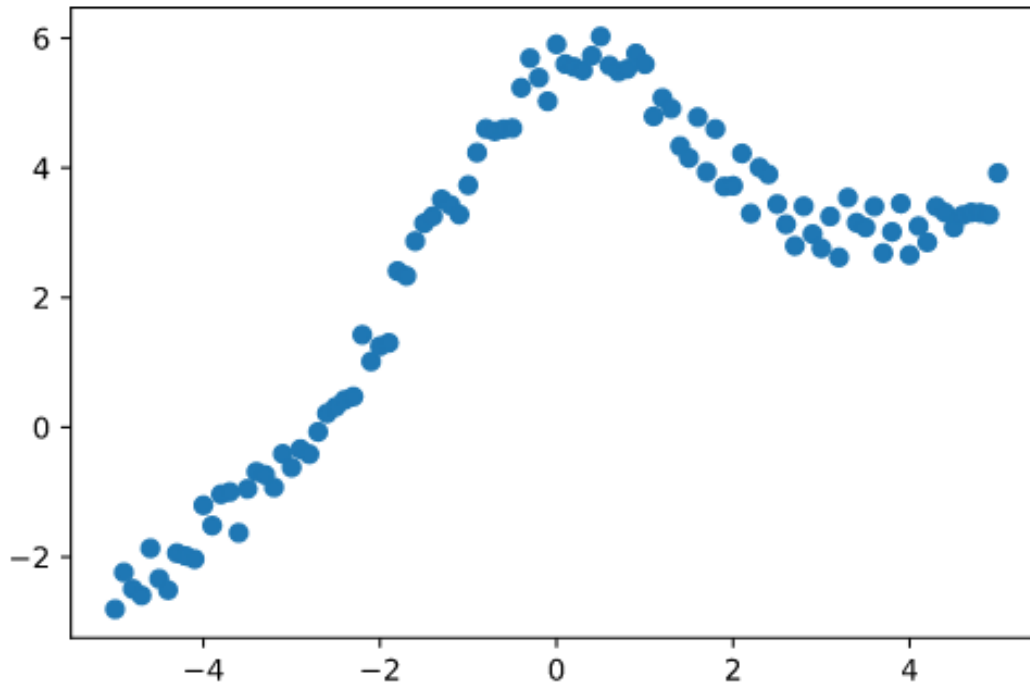




❖ 비선형회귀

- 문제: 다음과 같은 데이터에 대해, Levenberg-Marquardt 알고리즘을 이용하여 최적의 곡선을 구한다.
- fitting함수 :

$$y = ax + b e^{-cx^2}$$



```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-5, 5, 101)
y = 0.6 * x + 5 * np.exp(-0.2 * x**2)
    + np.random.rand(101)
plt.plot(x, y, 'o')
plt.show()
```



```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def func(x, a, b, c):
    return a * x + b * np.exp(-c * x**2 )

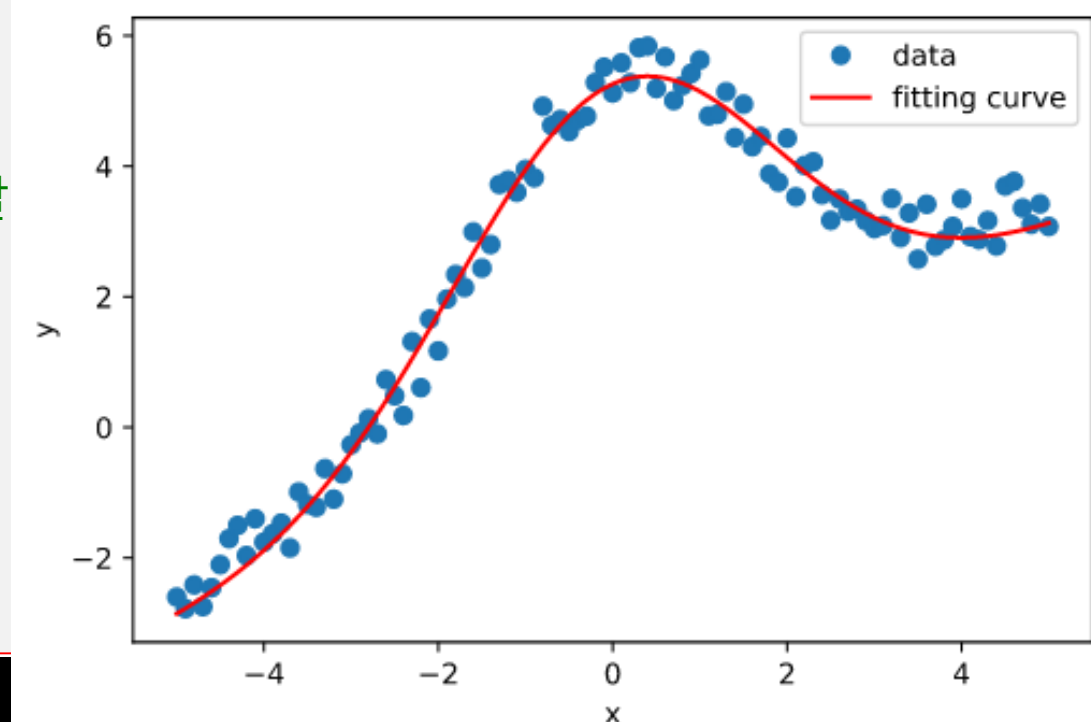
x = np.linspace(-5, 5, 101)
y = 0.6*x + 5* np.exp(-0.2* x**2) + np.random.rand(101)
```

```
popt, pcov = curve_fit(func, x, y)
print(popt) # 파라미터 (a, b, c), pcov는 공분산

yfit = func(x, *popt) # popt를 함수의 a,b,c로 전달
```

```
plt.plot(x, y, 'o', label='data')
plt.plot(x, yfit, 'r', label='fitting curve')
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

```
[0.59856002  5.26066393  0.14591535]
```



MEMO





감사합니다!