
Lecture 1

NumPy 표준 데이터 타입: NumPy 배열

파이썬의 특징

- 본격적인 수업에 앞서 우리가 사용하고 있는 프로그래밍 언어인 파이썬의 특징에 대해 간단히 살펴보자
- 우선 파이썬 문법은 쉽고 직관적이다
- 파이썬은 개발 난이도가 낮으면서도 범용성이 매우 높다
- 전세계에 사용되는 프로그래밍 언어 중 파이썬이 1위를 차지하고 있다
- 파이썬 언어의 사용자 유입이 늘어나게 되면서 유용한 **오픈 소스 라이브러리 (외부 라이브러리)**들이 생겨나게 되고, 이는 더욱 유저풀을 확대시키는 강력한 요인이 되었다
- 라이브러리(library)는 도서관 처럼 원하는 기능들을 찾아서 활용할 수 있게 작성된 프로그램 모음이다
- 실제로 우리가 무심코 사용한 내장 함수들은 모두 파이썬 표준 라이브러리에서 제공되는 것들이다
- **오픈 소스 라이브러리**는 특히 모두가 사용할 수 있도록 외부 사람 혹은 단체에 무료로 공개한 라이브러리이며, 파이썬 언어로 작성한 프로그램에서만 접근 및 활용이 가능한 유용한 라이브러리들이 넘치다보니 개발자들의 입장에서는 파이썬 언어의 사용을 반길 수 밖에 없다
- 대표적인 오픈 소스 라이브러리는 NumPy(수치 및 행렬 계산), Matplotlib(시각화), Pandas(데이터 분석), Scipy(과학·기술 컴퓨팅) 등이 있다

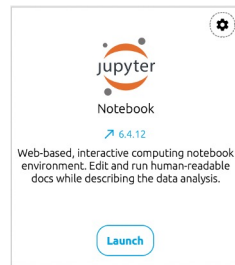
Worldwide, Aug 2023 :

Rank	Change	Language	Share	1-year trend
1		Python	28.04 %	+0.3 %
2		Java	15.78 %	-1.3 %
3		JavaScript	9.27 %	-0.2 %
4		C#	6.77 %	-0.2 %
5		C/C++	6.59 %	+0.4 %

<https://pypl.github.io/PYPL.html>

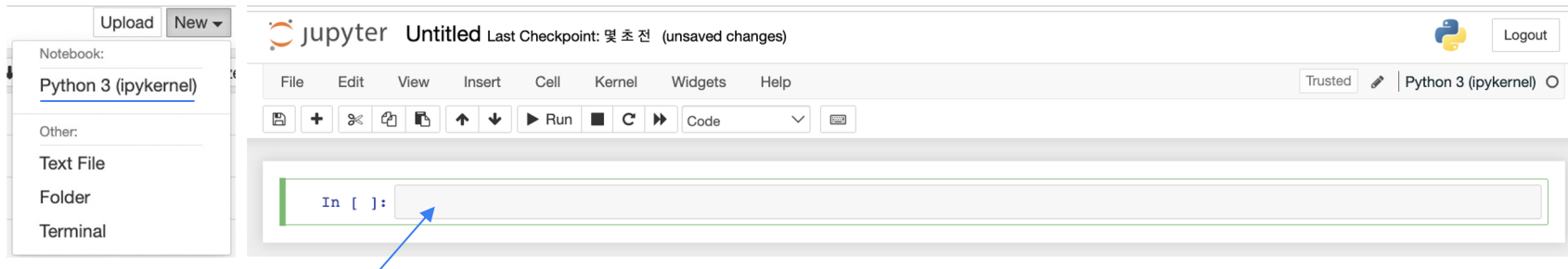
Jupyter Notebook

- 이번 시간부터는 IDE가 아닌 웹 브라우저에서 파이썬 코드를 작성하고 실행할 수 있도록 지원하는 **주피터 노트북(Jupyter Notebook)**을 사용해볼 것이다
- 주피터 노트북은 셀(Cell)단위로 실행 및 결과를 확인할 수 있어 직관적이다
- 많은 개발자들이 사용하는 구글 코랩(Google Colab) 역시 구글에서 제공하는 클라우드 기반의 주피터 노트북 환경이다
- 주피터 노트북을 따로 설치할 수도 있지만 우리는 **아나콘다(Anaconda)**를 각자 개인 PC 환경에 맞추어 설치하도록 하자
- 아나콘다는 주피터 노트북뿐만 아니라 파이썬 컴파일러, 유명 데이터 과학 패키지 등을 한번에 설치 및 관리하여 패키지 관리 및 디플로이를 단순케 해주는 오픈소스 패키지 관리자이다
- 아나콘다의 설치가 완료되었다면 이제 주피터 노트북을 실행하도록 하자



Jupyter Notebook

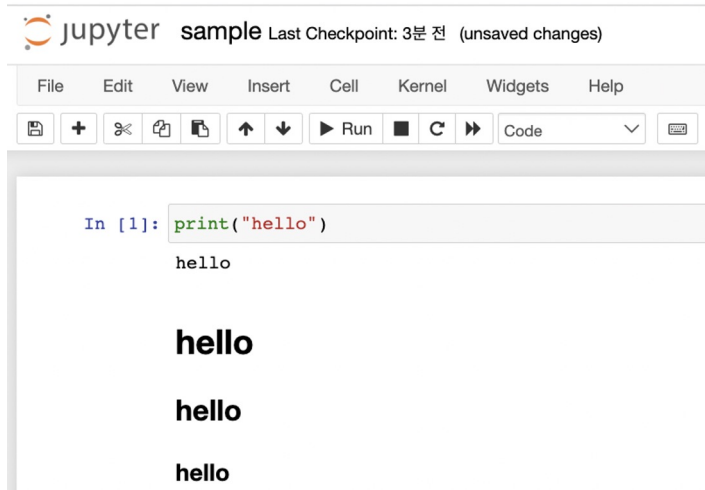
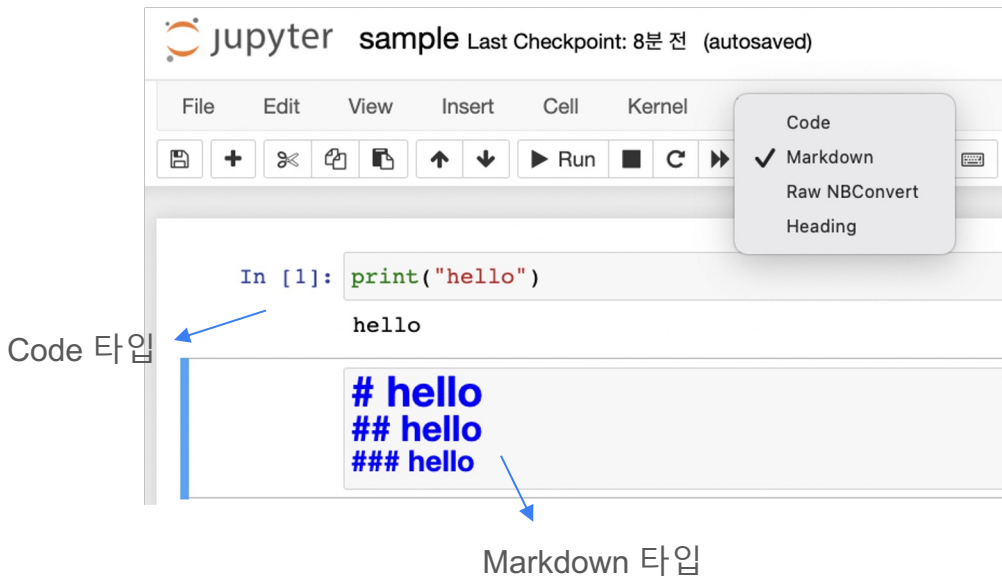
- 주피터 노트북을 실행하면 파일 검색기와 같은 형태를 웹 브라우저에서 만나볼 수 있다
- 원하는 파이썬 개발 경로로 접속한 뒤 오른쪽 상단에 있는 “New” 토글을 클릭한 후 “Notebook:” 아래에 위치한 “Python 3 (ipykernel)” 을 클릭하자
- 현재 우리가 바라보고 있는 생성된 파일의 확장자명은 **.ipynb**이며 “(주피터) 노트북 파일”이라고도 부른다
- 기존까지 사용했던 파이썬 스크립트 파일(.py)과는 다른 형태를 띄고 있음을 알 수 있다



- 주피터 노트북은 **셀(Cell)**단위로 실행 및 결과를 확인할 수 있어 직관적이다
- In []: 옆으로 검색창처럼 생긴 것이 셀이며 **Shift+Enter** 키를 누르면 셀 실행 및 신규 셀을 생성한다

Jupyter Notebook

- 가장 상단의 “Untitled”를 클릭하면 현재 파일 명의 이름을 바꿀 수 있다
- 셀을 선택한 후 “Code” 토글을 누른 뒤 각 셀 타입을 변경할 수도 있다



NumPy

- 넘파이(NumPy)는 대표적인 파이썬 수치 및 행렬 계산용 외부 라이브러리이다
- 외부 라이브러리가기에 개발 중에 넘파이에서 제공하는 기능을 사용하고 싶으면 사용하기 전에 코드 상에서 임포트(import)해야한다

In [1]: `1 !pip install numpy # NumPy 라이브러리가 설치되어 있지 않은 경우에만 필요`
`2 import numpy as np # 개발자들 십중팔구 넘파이는 np라는 별칭(alias)을 사용`

- 1995년 Numeric이라는 이름으로 출시한 후, 2006년 NumPy로 명칭을 바꾸어 꾸준히 개선되었다
- 특히 복잡한 행렬 계산(선형대수)과 관련된 기능을 많이 제공하여 범용적으로 사용된다



NumPy

- 파이썬 모듈(라이브러리, 패키지)에는 버전을 확인할 수 있는 `__version__` 속성을 지니도록 권장하고 있으며 넘파이 역시 이를 갖고 있다

In [2]: 1 print(np.__version__) # 넘파이 버전 확인

1.25.2

- 넘파이에서는 다차원 행렬 계산을 위해 **ndarray** (그리고 **matrix**) 자료형을 자체적으로 가지고 있다

In [3]: 1 my_first_array = np.array([0, 0.5, 1]) # 리스트로부터
 ndarray 객체 생성 (float와 int의 혼합)
 2 print(my_first_array)
 3 print(type(my_first_array))
 4 print(type(my_first_array[0])) # float64로의 변환
 5 print(my_first_array.dtype) # 전체 원소의 자료형

[0. 0.5 1.]
 <class 'numpy.ndarray'>
 <class 'numpy.float64'>
 float64

- 넘파이에서는 자체적인 자료형을 가지고 있는데 float64, int64가 예시이다
- ndarray 객체 생성 시 원소들이 전부 숫자이지만 하나라도 float값이 들어간다면 원소 전체가 float64 자료형으로 변환됨을 확인할 수 있다

ndarray(넘파이 배열)

- ndarray 객체는 배열로 간주하면 되며(행렬과 유사) 관련 연산들을 지원한다
 - 배열의 덧셈: 덧셈 연산자 +, 배열의 뺄셈: 뺄셈 연산자 -
 - 배열 원소별(element-wise) 곱셈: 곱셈 연산자 *
 - 배열 원소별 나눗셈, 몫 연산, 나머지 연산: 나눗셈 연산자 /, 몫 연산자 //, 나머지 연산자 %

```
In [4]: 1 my_second_array = np.array((1, 1, 1)) # 튜플로부터
        2 ndarray 객체 생성 (int만으로 구성)
        3 print(my_second_array)
        4 print(type(my_second_array))
        5 print(my_first_array + my_second_array) # 배열 덧셈
        6 print(my_first_array - my_second_array) # 배열 뺄셈
        7 print(my_first_array * my_second_array) # 배열 원소별 곱셈
        8 print(my_first_array / my_second_array) # 배열 원소별 나눗셈
        9 print(my_first_array // my_second_array) # 배열 원소별 몫 연산
       10 print(my_first_array % my_second_array) # 배열 원소별 나머지 연산
```



```
[1 1 1]
<class 'numpy.ndarray'>
<class 'numpy.int64'>
[1. 1.5 2.]
[-1. -0.5 0.]
[0. 0.5 1.]
[0. 0.5 1.]
[0. 0. 1.]
[0. 0.5 0.]
```


ndarray(넘파이 배열)

- ndarray 객체는 배열로 간주하면 되며 관련 연산들을 지원한다
 - 배열의 곱셈 (내적): @ 혹은 dot 메소드

```
In [5]: 1 A = np.array([[1, 2], [3, 4]]) # (2,2) shape의 배열
        2 B = np.array([[1, 1], [2, 2]])
        3 print(A, "A.shape:", A.shape, "A.ndim:", A.ndim)
        4 print("A.size:", A.size, "len(A):", len(A), "\n")
        5 print(B, "\n")
        6 print(A*B, "\n") # 배열 원소별 곱셈
        7 print(A@B) # 배열의 곱 (1)
        8 print(A.dot(B)) # 배열의 곱 (2)
        9 print(np.dot(A, B)) # 배열의 곱 (3)
```

잠깐! shape은 구체적인 배열의 모양을 알려주고 ndim은 총 차원 수를 알려준다 (len(A.shape) == A.ndim) size는 배열 내의 모든 원소 개수를 알려주고 배열에서 사용하는 len 함수는 가장 바깥 (혹은 첫) 차원에서의 값을 나타낸다 (len(A) == A.shape[0])

```
[[1 2]
 [3 4]] A.shape: (2, 2) A.ndim: 2
A.size: 4 len(A): 2
```

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

```
[[1 2]
 [6 8]]
```

```
[[ 5  5]
 [11 11]]
```

```
[[ 5  5]
 [11 11]]
```


```
[[ 5  5]
 [11 11]]
```

NumPy의 메소드

- 넘파이에서 제공하는 여러 메소드 (= 클래스 함수) 중 몇 개를 소개한다
- **arange**: range 내장 함수와 거의 동일한 기능을 제공한다; 지정된 범위에 따라 ndarray 객체를 생성한다

In [6]:

```
1 print(np.arange(5)) # 0부터(디폴트) 5 이전까지 1 간격(디폴트)
2 print(np.arange(1,5)) # 1부터 5 이전까지 1 간격(디폴트)
3 print(np.arange(1,5,2)) # 1부터 5 이전까지 2 간격
```




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

- **linspace**: 특정 범위의 수를 균등하게 나누고자 할 때 사용한다;
np.linspace(start, stop, count) 이처럼 세 개의 파라미터를 받으며 'start 부터 stop까지 count개 구간으로 나눈다'를 의미한다

In [7]:

```
1 print(np.linspace(1, 50), len(np.linspace(1,50))) # 1부터 50까지 총 50개 구간(디폴트)
2 print(np.linspace(1, 10, 5)) # 1부터 10까지 총 5개의 구간
```



```
[ 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. 30. 31. 32. 33. 34. 35. 36.
 37. 38. 39. 40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50.] 50
[ 1.   3.25  5.5   7.75 10. ]
```

NumPy의 메소드

- 넘파이에서 제공하는 여러 메소드 중 몇 개를 소개한다
- **zeros**: shape에 맞춰서 float64 타입의 0으로 채워진 ndarray 객체 생성
- **zeros_like**: 주어진 시퀀스 객체와 동일한 shape으로 int64 타입의 0으로 채워진 ndarray 객체 생성

```
In [8]: 1 print(np.zeros(5)) # float64 자료형으로 0을 채움
        2 print(np.zeros(5).dtype)
        3 C = np.array([[1,2],[3,4],[5,6]])
        4 print(np.zeros_like(C)) # 특정 시퀀스 객체와 같은
          shape를 가지되 0으로 채우기
        5 print(np.zeros_like(C).dtype) # zeros와는 다르게
          int64 자료형으로 0을 채움
        6 d = [1,2]
        7 print(np.zeros_like(d))
        8 e = (1,2,3)
        9 print(np.zeros_like(e))
```



```
[0. 0. 0. 0. 0.]
float64
[[0 0]
 [0 0]
 [0 0]]
int64
[0 0]
[0 0 0]
```

NumPy의 메소드

- 넘파이에서 제공하는 여러 메소드 중 몇 개를 소개한다
- **ones**: shape에 맞춰서 float64 타입의 1로 채워진 ndarray 객체 생성
- **ones_like**: 주어진 시퀀스 객체와 동일한 shape으로 int64 타입의 1로 채워진 ndarray 객체 생성

```
In [9]: 1 print(np.ones(5)) # float64 자료형으로 1을 채움
        2 print(np.ones(5).dtype)
        3 C = np.array([[1,2],[3,4],[5,6]])
        4 print(np.ones_like(C)) # 특정 시퀀스 객체와 같은
        shape를 가지되 1로 채우기
        5 print(np.ones_like(C).dtype) # zeros와는 다르게
        int64 자료형으로 1을 채움
        6 d = [1,2]
        7 print(np.ones_like(d))
        8 e = (1,2,3)
        9 print(np.ones_like(e))
```



```
[1. 1. 1. 1. 1.]
float64
[[1 1]
 [1 1]
 [1 1]]
int64
[1 1]
[1 1 1]
```

NumPy의 메소드

- 넘파이에서 제공하는 여러 메소드 중 몇 개를 소개한다
- **full**: 주어진 shape와 값에 맞춰서 채워진 ndarray 객체 생성 (zeros_like, ones_like의 일반화된 버전)

```
In [10]: 1 print(np.full((2,3), 3))
          2 print(np.full((2,3), 3).dtype)
          3 print(np.full(shape=(2,3), fill_value=3.))
          4 print(np.full((2,3), 3.).dtype)
```



```
[[3 3 3]
 [3 3 3]]
int64
[[3. 3. 3.]
 [3. 3. 3.]]
float64
```

- **eye**: 주어진 shape에 맞춰서 Identical matrix(단위행렬) 객체 생성 (Identical의 'I' 발음에 본따서 명칭)

```
In [11]: 1 print(np.eye(3)) # (3,3)으로 단위행렬 객체 생성
          2 print(np.eye(3)*3) # (3,3)으로 값이 3인 대각행렬 생성
```



```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[[3. 0. 0.]
 [0. 3. 0.]
 [0. 0. 3.]]
```

NumPy의 메소드

- 넘파이에서 제공하는 여러 메소드 중 몇 개를 소개한다
- **random.randn**: 표준정규분포로부터 주어진 shape에 맞춰 난수를 채운 ndarray 객체 생성

```
In [12]: 1 print(np.random.randn(3))  
         2 print(np.random.randn(3,3))  
         3 print(np.random.randn(2,2,2))
```




```
[-0.22963385  1.89041293 -  
 1.7878039 ]  
[[-2.13048263 -0.51521275  
  0.54960021]  
 [ 0.41027972 -0.11498224  
  0.92683174]  
 [ 0.62963986 -0.51164382 -  
  0.13280488]]
```

잠깐! 표준정규분포란 평균이 0이고 표준편차가 1인 정규분포를 의미합니다

NumPy의 메소드

- ndarray 객체는 **reshape** 속성 혹은 **reshape** 메소드를 통해 shape을 변경할 수 있다

```
In [13]: 1 my_array = np.arange(10)
          2 my_new_array = my_array.reshape(2,5) # reshape 속성
          3 my_new_array2 = np.reshape(my_array, (2,-1))
            # reshape 메소드; -1은 인터프리터가 자동으로 연산
          4 print(my_new_array)
          5 print(my_new_array2)
```



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

- 성립될 수 없는 shape으로 모양을 변형하려고 하면 에러가 발생한다

```
In [14]: 1 my_array = np.arange(10)
          2 my_new_array2 = my_array.reshape(3,-1)
          -----
```



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

NumPy의 메소드

- ndarray 객체는 **T 속성** 혹은 **transpose 메소드**를 통해 치환(transpose)할 수 있다

```
In [15]: 1 print(my_new_array) # (2,5) shape
          2 my_new_array = my_new_array.T
          3 print(my_new_array) # (5,2) shape
          4 my_new_array = np.transpose(my_new_array)
          5 print(my_new_array) # (2,5) shape
          6 my_3d_array = np.arange(8).reshape(2,2,2)
          7 print(my_3d_array)
          8 my_3d_array = my_3d_array.T # 2D가 아닐 경우에는?
          -> 항상 처음과 마지막 axis간의 치환이 이뤄진다
          9 print(my_3d_array)
```



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

 [[4 5]
   [6 7]]]
[[[0 4]
   [2 6]]

 [[1 5]
   [3 7]]]
```


NumPy의 메소드

- **swapaxes** 메소드는 특정 축들간의 변경도 가능하다
(transpose의 일반화된 버전)

```
In [16]: 1 my_3d_array = np.arange(8).reshape(2,2,2)
          2 print(my_3d_array)
          3 my_3d_array = np.swapaxes(my_3d_array, 0 , 1)
            # 0번째와 1번째 축간의 교환
          4 print(my_3d_array)
```



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

  [[4 5]
   [6 7]]]

[[[0 1]
   [4 5]]

  [[2 3]
   [6 7]]]
```

NumPy의 메소드

- **ndarray의 결합과 분리**와 관련된 메소드들 몇 가지에 대해서 알아보자
- 결합: `hstack`, `vstack`, `stack`, `concatenate`
- 분리: `split`
- **hstack**: horizontal stack; 넘파이 배열을 수평으로 쌓는다

```
In [17]: 1 count_to_ten = np.arange(1,11)
          2 count_to_twenty = np.hstack([count_to_ten, count_to_ten+10])
          3 print(count_to_twenty)
```

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

- **vstack**: vertical stack; 넘파이 배열을 수직으로 쌓는다

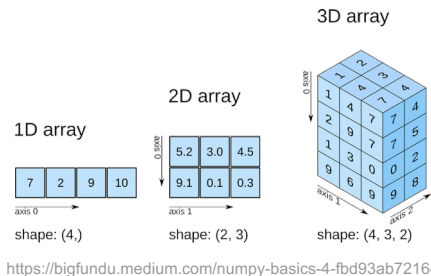
```
In [18]: 1 count_to_twenty = np.vstack([count_to_ten, count_to_ten+10])
          2 print(count_to_twenty)
```

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

NumPy의 메소드

- **stack**: hstack과 vstack의 일반화 버전; 새로운 축을 생성하여 그 방향으로 넘파이 배열을 결합한다

```
In [19]: 1 a = np.arange(1,5).reshape(2,2)
          2 print(a, "ndim:", a.ndim, "shape:", a.shape)
          3 print(np.stack([a, a+4, a+8]), np.stack([a, a+4, a+8]).shape) # axis = 0
          4 print(np.stack([a, a+4, a+8], axis=1), np.stack([a, a+4, a+8], axis=1).shape)
          5 print(np.stack([a, a+4, a+8], axis=2), np.stack([a, a+4, a+8], axis=2).shape)
```



```
[[1 2]
 [3 4]] ndim: 2 shape: (2, 2)
```

```
[[[ 1  2]
   [ 3  4]]
 [[ 5  6]
   [ 7  8]]
 [[ 9 10]
   [11 12]]] (3, 2, 2)
```

```
[[[ 1  2]
   [ 5  6]
   [ 9 10]]
 [[ 3  4]
   [ 7  8]
   [11 12]]] (2, 3, 2)
```

```
[[[ 1  5  9]
   [ 2  6 10]]
 [[ 3  7 11]
   [ 4  8 12]]] (2, 2, 3)
```

NumPy의 메소드

- **concatenate**: stack과는 다르게 기존에 있던 축을 활용하여 배열을 연장하는 방식

```
In [20]: 1 a = np.arange(1,5).reshape(2,2)
          2 print(a, "ndim:", a.ndim, "shape:", a.shape)
          3 print(np.concatenate([a, a+4, a+8]), np.concatenate([a, a+4, a+8]).shape)
          4 print(np.concatenate([a, a+4, a+8], axis=1), np.concatenate([a, a+4, a+8], axis=1).shape)
```

```
[[1 2]
 [3 4]] ndim: 2 shape: (2, 2)
```

```
[[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12]] (6, 2)
```

```
[[ 1  2  5  6  9 10]
 [ 3  4  7  8 11 12]] (2, 6)
```

```
5 print(np.concatenate([a, a+4, a+8], axis=2)) # 축 오류 발생 → AxisError: axis 2 is out of bounds for array of dimension 2
```

NumPy의 메소드

- **split**: split은 넘파이 배열을 분리하는 메소드로 `np.split(ary, indices_or_sections, axis)` 방식을 취한다
- 첫번째 인자인 `ary`는 원본 넘파이 배열이고 세번째 인자인 `axis`는 분리하는 축 방향을 의미한다 (0이 디폴트)
- `indices_or_sections`에 해당하는 인수가 **정수형 N**이라면 **분리되는 배열이 N개의 동일한 값을 가진다** (나눌 수 없다면 오류 발생) 혹은 **정렬된(sorted) 1D 꼴**이라면 배열의 요소값을 기준으로 값을 나눈다

```
In [21]: 1 x = np.arange(6)
          2 print(np.split(x,2)) # 2등분
          3 print(np.split(x,3)) # 3등분
          4 print(np.split(x,4)) # 6개 원소를 4등분할 수 없음 (오류)
```

→

```
[array([0, 1, 2]), array([3, 4, 5])
 array([0, 1]), array([2, 3]), array([4, 5])]
```

→

```
ValueError: array split does not result in an equal division
```

```
In [22]: 1 x = np.arange(6)
          2 print(np.split(x,(2,3))) # 인덱스 2와 3을 기준으로 분리
          3 print(np.split(x,[2,4,5,9])) # 인덱스 2, 4, 5, 그리고 9를 기준으로 분리
```

```
[array([0, 1]), array([2]), array([3, 4, 5])
 array([0, 1]), array([2, 3]), array([4]), array([5]), array([], dtype=int64)]
```

NumPy의 메소드

- **split**: split은 넘파이 배열을 분리하는 메소드로 `np.split(ary, indices_or_sections, axis)` 방식을 취한다
- 첫번째 인자인 `ary`는 원본 넘파이 배열이고 세번째 인자인 `axis`는 분리하는 축 방향을 의미한다 (0이 디폴트)
- `indices_or_sections`에 해당하는 인수가 **정수형 N**이라면 **분리되는 배열이 N개의 동일한 값을 가진다** (나눌 수 없다면 오류 발생) 혹은 **정렬된(sorted) 1D꼴**이라면 배열의 요소값을 기준으로 값을 나눈다

```
In [23]: 1 x = np.arange(6).reshape(2,3)
          2 print(x)
          3 print(np.split(x, 2)) # axis=0
          4 print(np.split(x, 3, axis=1))
```



```
[[0 1 2]
 [3 4 5]]
array([[0, 1, 2]], array([[3, 4, 5]])]
array([[0],
       [3]]), array([[1],
       [4]]), array([[2],
       [5]])]
```

```
5 print(np.split(x, 2, axis=1)) # 3개 원소를 2등분 불가능 (오류)
```

→ **ValueError: array split does not result in an equal division**