



3장 구현을 위한 도구

이장에서 배울 것들

- 어떤 언어, 어떤 개발 환경으로 구현할 것인가.
- 데이터를 다루기 위해 필요한 패키지는 무엇인가.
- 대규모 데이터를 벡터화 연산으로 다루어야 하는 이유는 무엇인가.
- 데이터의 속성들을 더욱 구조적으로 관리할 수 있는 도구는 무엇인가.

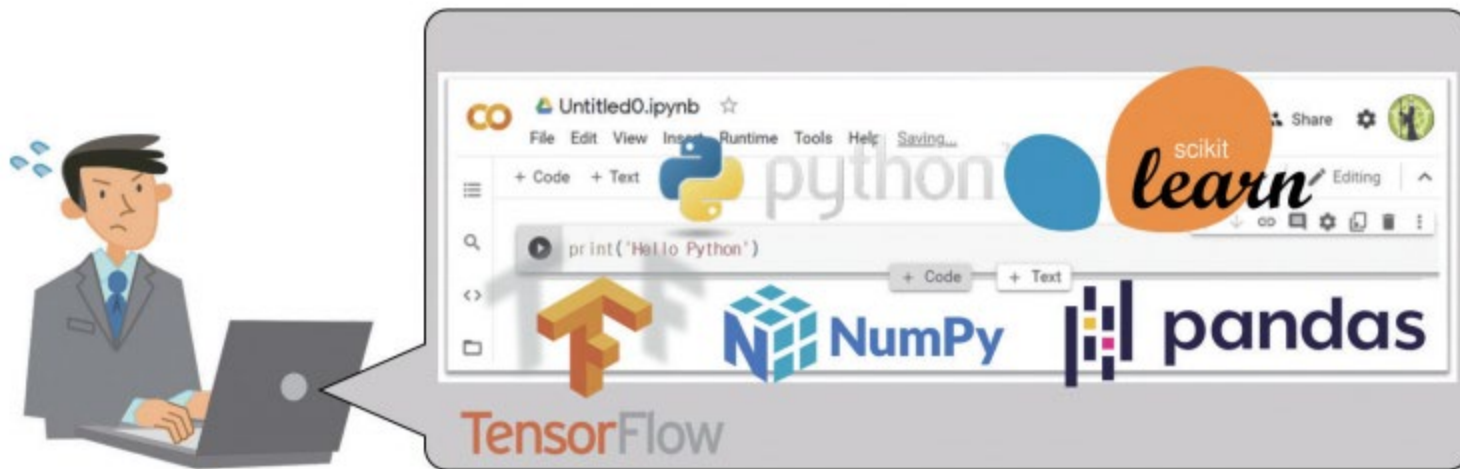
3.1 파이썬

- 파이썬Python은 귀도 반 로섬Guido van Rossum이 1991년에 개발한 대화형 프로그래밍 언어
- 파이썬의 특징 중에 하나는 정수, 부동소수점, 문자와 같은 전통적인 자료형data type 뿐만 아니라, 데이터 묶음을 처리하기에 편리한 리스트, 튜플, 딕셔너리, 집합과 같은 자료형을 기본으로 제공
- 파이썬은 '객체지향 프로그래밍 언어'이며, 파이썬이 다루는 모든 자료형, 함수, 모듈은 객체

- 특정한 클래스에 속한 객체들이 사용할 수 있는 함수들을 해당 클래스의 **메소드** method라고 부른다.
 - 객체 지향 언어는 리스트와 같은 **클래스를 정의**하고, 해당 클래스의 인스턴스 객체를 생성한 뒤, 이들이 메소드를 이용하여 다양한 일을 할 수 있도록 지원하는 언어

```
>>> animals = ['lion', 'tiger', 'cat', 'dog']
>>> animals.sort()           # animals 리스트 내부 문자열을 알파벳 순으로 정렬
>>> animals
['cat', 'dog', 'lion', 'tiger']
>>> animals.append('rabbit')  # animals 리스트에 새 원소를 추가
>>> animals
['cat', 'dog', 'lion', 'tiger', 'rabbit']
>>> animals.reverse()        # animals 리스트를 원래 원소의 역순으로 재배열
>>> animals
['rabbit', 'tiger', 'lion', 'dog', 'cat']
```

- 실습은 **코랩** Colab 환경에서 파이썬으로 이루어지며, **넘파이** NumPy, **판다스** pandas와 같은 데이터 처리 패키지와 **사이킷런** scikit-learn과 **텐서플로우** TensorFlow와 같은 머신러닝 도구를 사용

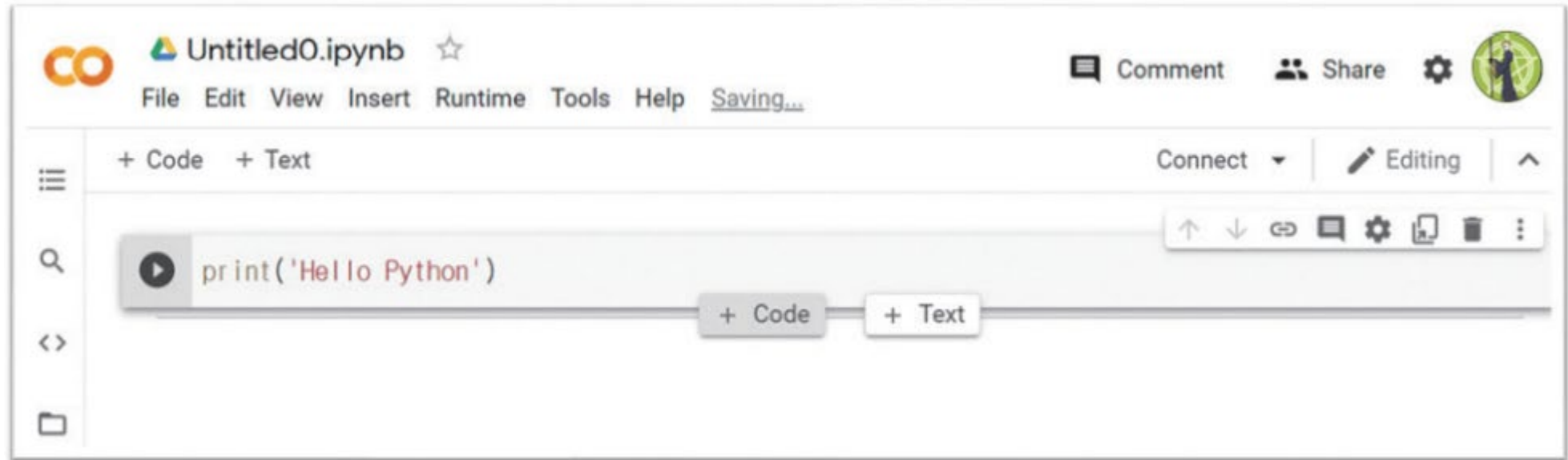


3.2 구글 코래버러토리를 이용한 프로그래밍

- 구글의 [코래버러토리](#) Colaboratory는 클라우드 환경에서 주피터 노트북 기반의 파이썬 개발을 위한 탁월한 환경을 제공
- 주피터 노트북은 웹 환경에서 대화식으로 프로그램을 작성하고 실행할 수 있는 오픈소스 프로젝트
- 줄여서 [코랩](#) colab이라고 부르는 이 서비스는 주피터 노트북을 구글의 서버에서 구동시켜 사용자가 이 서버에 접속하는 방식으로 파이썬을 이용

- 코래버러토리 환경은 다음 웹사이트에 구글계정으로 로그인하여 사용

<https://colab.research.google.com/>



- 코랩 환경은 각자의 컴퓨터가 아니므로 각자가 개인적으로 가지고 있는 데이터를 활용할 수 있는 방법이 필요
- 이를 위해서 구글 드라이브를 이용
- 가장 먼저 해야할 일은 자신의 드라이브를 코랩에서 사용할 수 있도록 **마운트mount**하는 것
- 마운트는 어떤 장치를 컴퓨터 시스템에서 접근할 수 있도록 등록하는 일



```
from google.colab import drive  
drive.mount('/content/drive')
```

Go to this URL in a browser: https://accounts.google.com/*****

Enter your authorization code:

4/3wH*****9UA0ng

Mounted at /content/drive


- 위와 같이 드라이브를 코랩에 마운트시키려고 하면 이 작업에 대한 승인을 요청하는 페이지의 주소가 아래 그림과 같이 나타난다.
- 클릭하면 구글 계정으로 로그인을 하여 드라이브 파일 스트림이 여러분의 드라이브에 접근할 수 있도록 하라는 표시가 나타날 것이다.



- 로그인을 마치면 승인에 필요한 코드가 나타날 것
- 복사하여 구글 코랩에 실행결과 셀의 입력창에 넣으면 된다.

3.3 구글 드라이브의 내용을 코래버러토리에서 이용하기

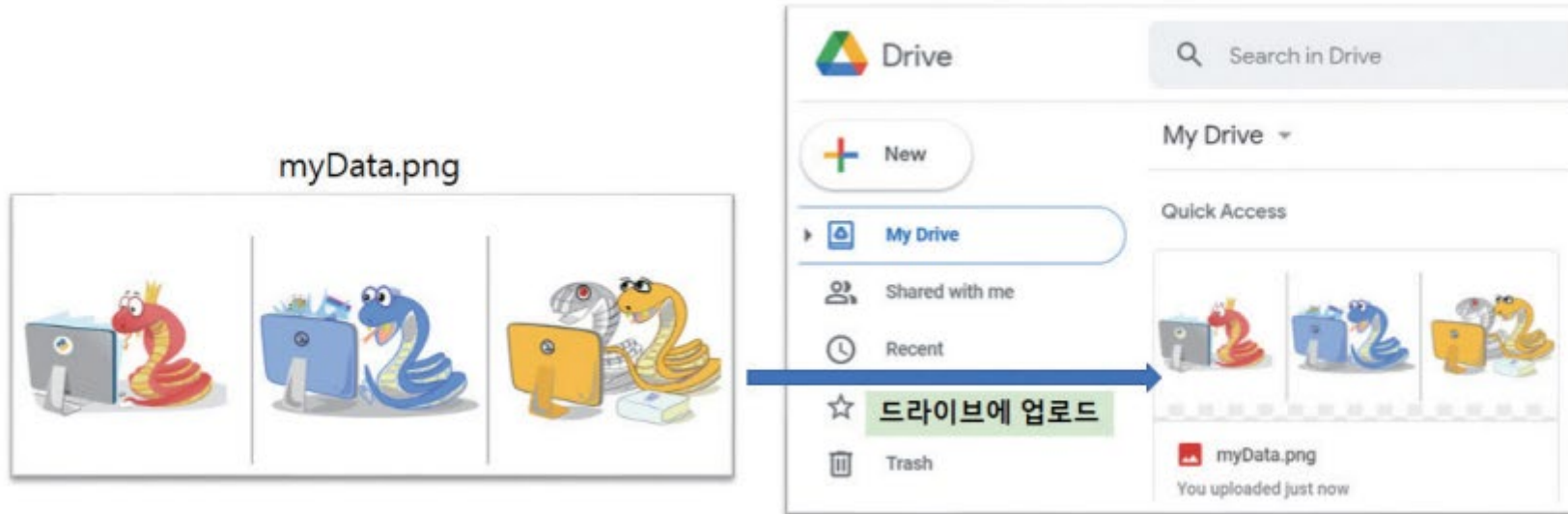
- 구글 코랩에서 현재 작업하고 있는 위치가 어디인지 알고 싶으면 !pwd를 아래와 같이 입력
- !은 코랩의 시스템 명령을 입력하는 기호
- pwd는 "print working directory" 의 약어로 현재 작업 중인 위치를 출력하는 리눅스Linux 명령어(이 책에는 이와 같은 간단한 유닉스 명령이 종종 등장합니다.)



```
!pwd
```

```
/content
```

- 다음과 같이 myData.png 라는 파일을 구글 드라이브 저장소에 업로드
- 저장 장치의 내용을 볼 때 사용하는 리눅스 명령은 ls인데, list를 의미



```
!ls ./drive/MyDrive -la
```

```
total 1247
```

```
-rw----- 1 root root 1226500 Sep  9 05:20 myData.png
```

```
drwx----- 2 root root    4096 Nov 19  2019 Pandas
```

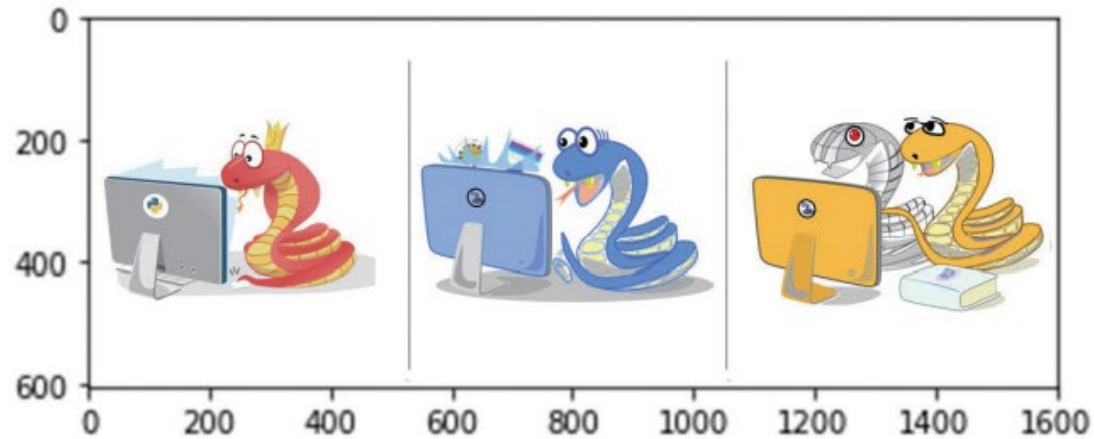
- 이 파일을 읽어서 화면에 그리는 일을 해 보자
 - 맷플롯립이라는 패키지를 활용해 쉽게 할 수 있다.
 - 패키지는 이미 코랩 환경에 설치가 되어 있다.



```
import matplotlib.image as mpimg  
import matplotlib.pyplot as plt
```

```
img = mpimg.imread('./drive/MyDrive/myData.png')  
plt.imshow(img)
```

<matplotlib.image.AxesImage at 0x7fe3d365fda0>



3.4 넘파이는 머신러닝을 위한 데이터 처리의 핵심 도구

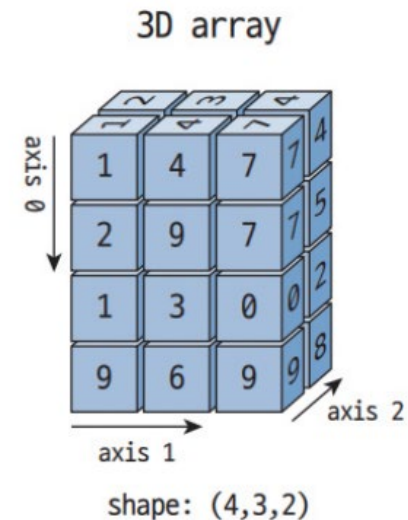
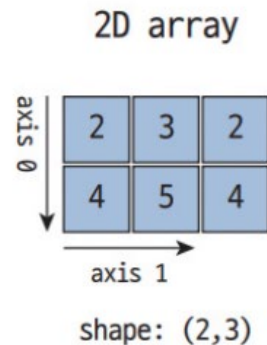
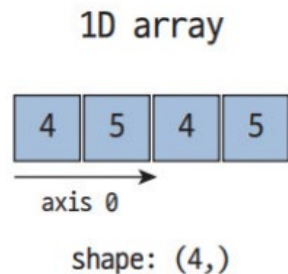
- 데이터를 처리할 때는 리스트와 리스트 간의 다양한 연산이 필요
 - 파이썬 기본 리스트는 이러한 기능이 부족
 - 리스트를 다루는 일은 연산 속도도 빠르지 않다.
- 데이터 과학자들은 넘파이Numpy 배열을 선호
 - 리스트는 두 리스트를 더할 때 이들을 벡터로 간주하지 않고 결합해 버리지만, 넘파이의 배열은 1차원은 벡터, 2차원은 행렬로 간주하고 연산

- 넘파이를 사용할 때는 numpy 모듈을 import
 - 다음과 같이 as 키워드 다음에 나오는 np라는 별칭을 지정하여 사용



```
import numpy as np    # numpy의 별칭으로 np를 지정함
my_array = np.array( [ 1, 2, 3] )
```

- 넘파이의 핵심적인 객체는 다차원 배열
 - 배열의 각 요소는 **인덱스**index라고 불리는 정수들 로 참조
 - 넘파이에서 차원은 **축**axis
 - 아래 그림은 넘파이의 1차원, 2차원, 3 차원 배열의 축과 그 shape



- 넘파이는 성능이 우수한 ndarray 객체를 제공
 - ndarray는 n차원 배열을 의미
- 전통적으로 배열은 동일한 자료형을 가진 데이터를 연속으로 저장
 - ndarray 객체 역시 동일한 자료형의 항목들만 저장
 - 파이썬의 리스트는 동일하지 않은 자료형을 가진 항목들도 저장
- 다차원배열은 다음과 같은 속성을 가지고 있다.



```
a = np.array([1, 2, 3])      # 넘파이 ndarray 객체의 생성
```

```
# a 객체의 형태(shape), 차원, 요소의 자료형, 요소의 크기(byte), 요소의 수  
a.shape, a.ndim, a.dtype, a.itemsize, a.size
```

```
((3,), 1, dtype('int64'), 8, 3)
```

3.5 넘파이 활용의 기본 - 브로드캐스팅, 인덱싱, 슬라이싱

- 넘파이는 데이터를 고속으로 처리하기 위해
 - 요소들을 동일한 자료형으로 제한해 효율적인 접근
 - 벡터화^{vectorization}와 브로드캐스팅^{broadcasting} 기법을 사용

- 어느 회사의 직원 4인 월급이 각각 [240, 260, 220, 255]라고 하고, 이를 넘파이 배열에 아래와 같이 저장하자



```
import numpy as np  
sal = np.array([240, 260, 220, 255])
```

- 넘파이는 스칼라 값을 벡터의 각 원소로 전파하여 덧셈
 - 브로드캐스팅이라고 한다.
 - 스칼라 값을 복사하여 같은 차원의 벡터를 만드는 작업이 없어 복사에 의한 속도 저하를 피할 수 있다.



```
sal = sal + 100    # 스칼라값 100은 [100, 100, 100, 100]으로 변환됨  
print(sal)
```

```
[340, 360, 320, 355]
```

- 넘파이 배열에서 파이썬의 리스트와 동일한 방식으로 인덱스와 슬라이싱



```
scores = np.array([58, 72, 93, 94, 89, 78, 99])  
scores[3:]      # 인덱스를 생략하면 마지막 요소까지 슬라이싱
```

```
array([94, 89, 78, 99])
```



```
scores[4:-1]    # 인덱스로 -1을 사용할 경우 78까지 슬라이싱
```

```
array([89, 78])
```

- 넘파이의 2차원 배열은 수학에서의 행렬과 같으며 역행렬이나 행렬식을 구하는 등의 행렬 연산들이 넘파이 배열에 쉽게 적용될 수 있도록 구현



```
np_array = np.array([[1,2,3],  
                     [4,5,6],  
                     [7,8,9]]) # 2차원 배열(넘파이 다차원 배열)
```

```
np_array
```

```
array([[1, 2, 3],  
       [4, 5, 6],  
       [7, 8, 9]])
```

- 2차원 배열도 파이썬의 리스트와 동일한 방식으로 인덱싱을 사용



```
np_array[0, 2]
```

```
3
```

- 넘파이에서 슬라이싱은 넘파이 인덱싱 방식과 결합하여 행렬을 효율적으로 추출



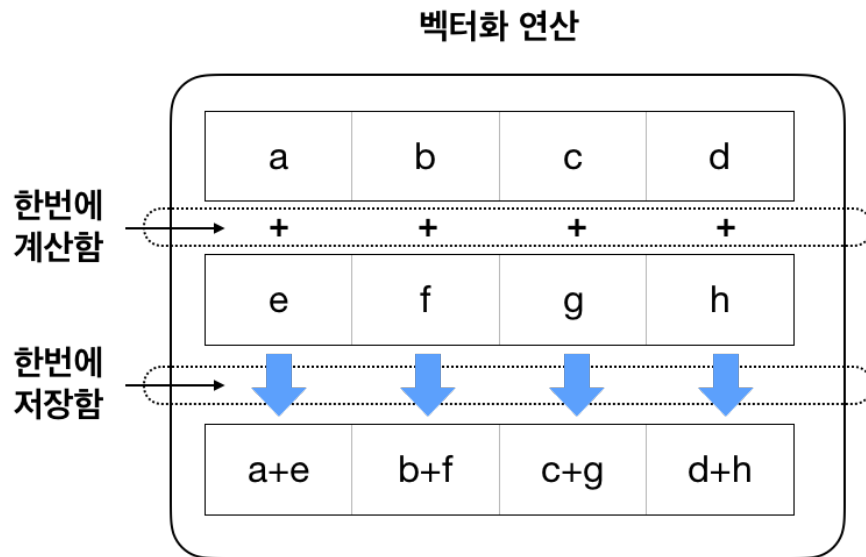
```
np_array = np.array([[1, 2, 3, 4], [5, 6, 7, 8],  
                     [9, 10, 11, 12], [13, 14, 15, 16]])
```

```
np_array[0:2, 2:4]
```

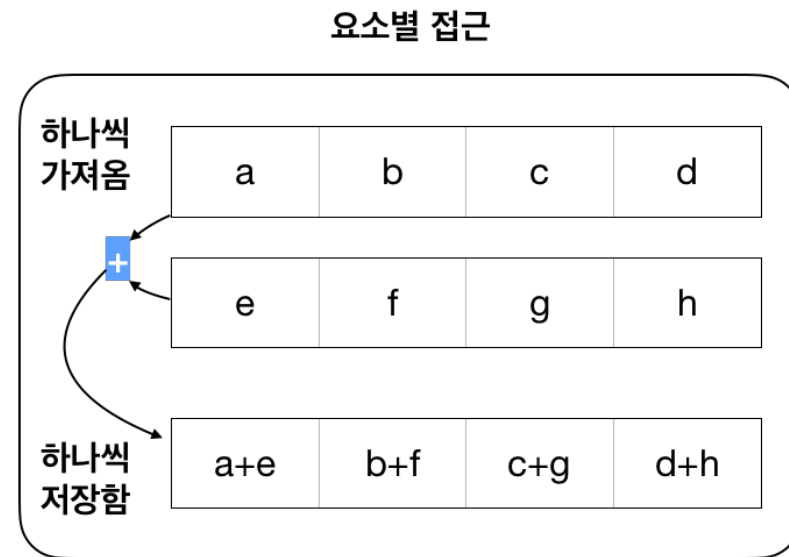
```
array([[3, 4],  
       [7, 8]])
```

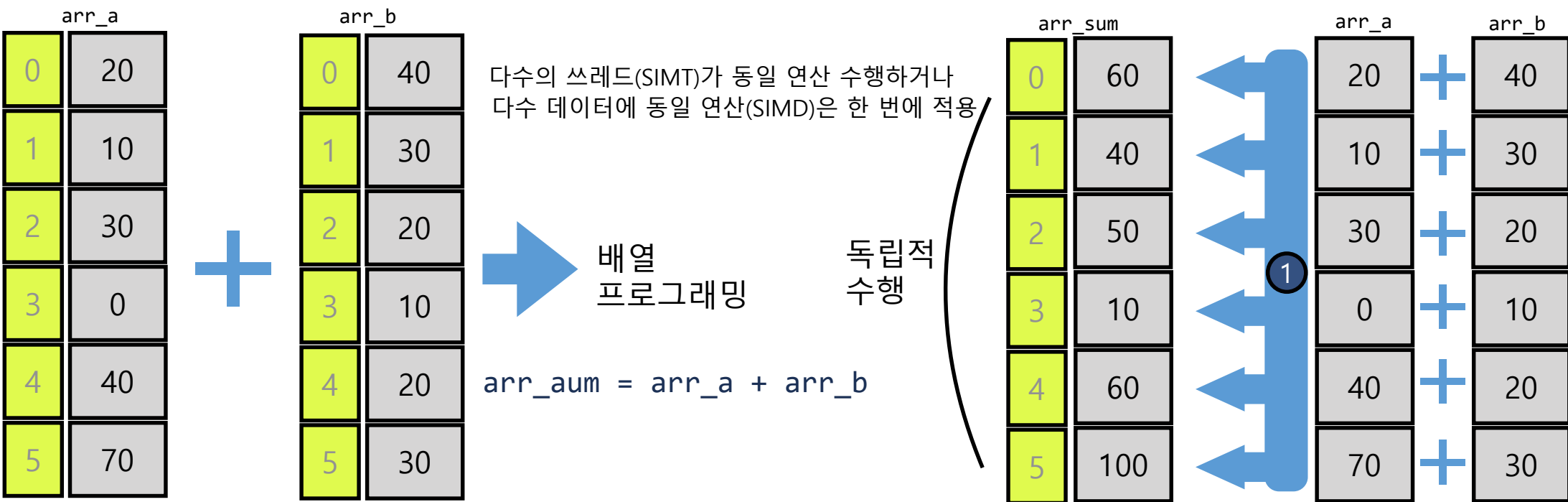
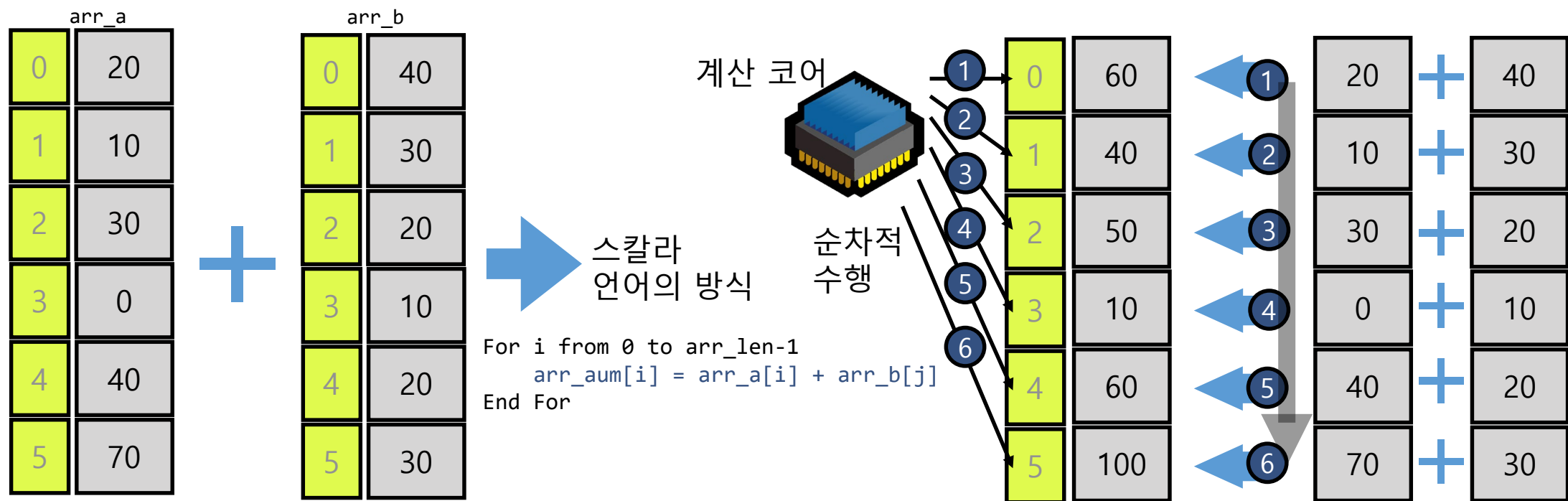
3.6 벡터화 연산 - 넘파이 배열 계산 성능의 핵심

- 넘파이를 이용한 수치 데이터 처리의 가장 큰 장점은 **벡터화** vectorization
- 벡터화 연산은 배열의 원소 각각을 가져와 연산하는 것이 아니라, 하나의 명령이 여러 데이터에 동시에 적용된 방식
 - 배열 프로그래밍 array programming의 핵심 요소



벡터화를 통한 병렬적인 연산이
넘파이의 핵심입니다.





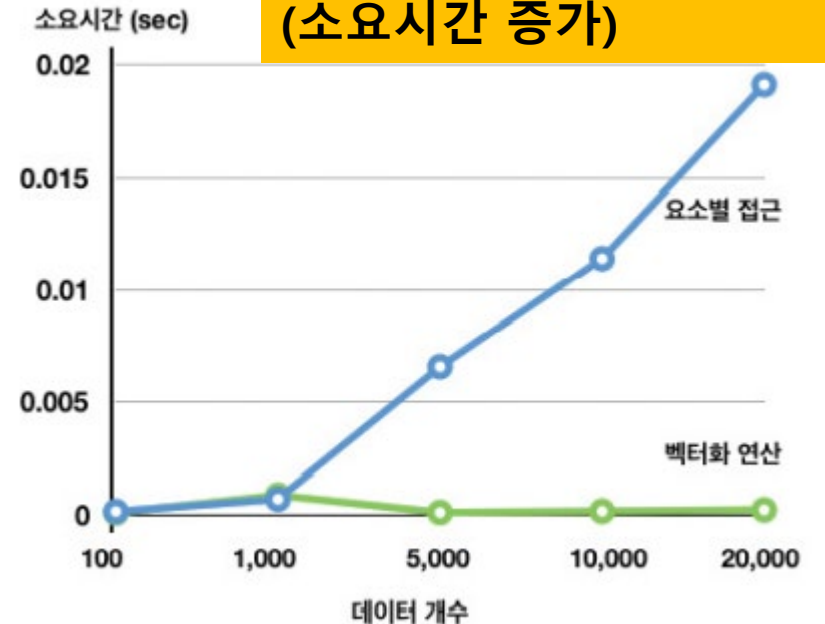
- 벡터화 연산은 큰 작업을 분할하여 다수의 프로세서processor에서 나누어 처리하는 전통적 병렬 처리와 달리 하나의 프로세서 내에서 병렬적 데이터 처리가 묵시적implicit으로 이루어짐.
 - 이러한 방식을 SIMDsingle instruction multiple data 방식
 - 현재 대다수의 CPU가 SIMD 방식을 지원



```
values = np.random.rand(100)
weights = np.random.rand(100)
weighted_values = np.empty(len(values)) # 빈 행렬 생성
for i in range(len(values)):           # 명시적인 작업 지시 : C 스타일
    weighted_values[i] = weights[i] * values[i]
```



```
values = np.random.rand(100)
weights = np.random.rand(100)
weighted_values = values * weights # 묵시적 벡터 요소별 곱셈: Numpy 스타일
```



요소별 접근 연산은 데이터가 증가하면 속도도 느려집니다 (소요시간 증가)

병렬적인 연산(속도 향상)으로 데이터가 증가해도 속도는 느려지지 않습니다.

3.7 논리 인덱싱으로 빠르게 데이터 추려내기

- 논리 인덱싱 `logical indexing`이란 어떤 조건을 주어서 배열에서 원하는 값을 추려내는 것
- 사람들의 나이가 저장된 넘파이 배열 `ages`가 있다고 하자. `ages`에서 20살이 넘는 사람만 고르려고 하면 다음과 같이 조건식을 활용



```
ages = np.array([18, 19, 25, 30, 28])  
y = ages > 20  
y
```

```
array([False, False,  True,  True,  True])
```


- 이러한 방식으로 배열의 특정한 요소에 접근하는 방식을 **논리 인덱싱**

```
ages[ ages > 20 ]  
  
array([25, 30, 28])
```

- 비교 연산자를 사용하여 다음과 같은 짝수 구하기 연산자를 적용

넘파이 배열 np_array

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

np_array %2 == 0

False	True	False	True
False	True	False	True
False	True	False	True
False	True	False	True

np_array [np_array %2 == 0]

2	4	6	8	10	12	14	16
---	---	---	---	----	----	----	----

3.8 판다스 소개

- 넘파이는 2차원 행렬 형태의 데이터를 지원
- 데이터의 속성을 표시하는 행이나 열의 레이블을 가지고 있지 않다. 파이썬의 판다스pandas 패키지를 사용하면 이러한 문제를 해결
 - 시리즈와 데이터프레임이라는 두 가지 데이터 구조를 제공
 - 각 행과 열은 이름이 부여되며, 행의 이름을 **인덱스index**, 열의 이름을 **컬럼스columns**
 - **시리즈series**는 동일 유형의 데이터를 저장하는 1차원 배열



```
import numpy as np
import pandas as pd
series = pd.Series([1, 3, np.nan, 4]) # np.nan은 결측값
series
```

0	1.0
1	3.0
2	NaN
3	4.0

- **데이터프레임** dataframe 은 시리즈 데이터가 여러 개 모여서 2차원적 구조를 갖는 것
- 예를 들면 다음처럼 레이블이 붙은 테이블이 데이터프레임 각 열은 동일한 자료형을 가진 시리즈임을 알 수 있다.

이름	나이	학교	평점
김수안	19	고교	4.35
김수정	23	대학	4.23
박동윤	22	대학	4.25
강이안	19	고교	4.37
강지안	16	중학교	4.25
박동민	25	대학원	4.5



```
name_series = pd.Series(['김수안', '김수정', '박동윤',
                        '강이안', '강지안', '박동민'])
age_series = pd.Series([19, 23, 22, 19, 16, 25])
school_series = pd.Series(['고교', '대학', '대학',
                          '고교', '중학교', '대학원'])
grade_series = pd.Series([4.35, 4.23, 4.25, 4.37, 4.25, 4.5])
print(name_series, age_series, school_series, grade_series)
```

```
0 김수안
1 김수정
2 박동윤
3 강이안
4 강지안
5 박동민
```

dtype: object

```
0 19
1 23
2 22
3 19
4 16
5 25
```

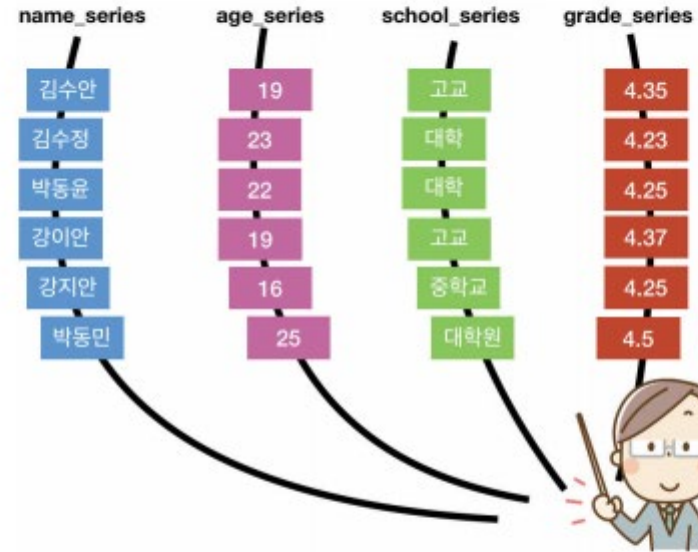
dtype: int64

```
0 고교
1 대학
2 대학
3 고교
4 중학교
5 대학원
```

dtype: object

```
0 4.35
1 4.23
2 4.25
3 4.37
4 4.25
5 4.50
```

dtype: float64



4개의 **시리즈**가 있어요.
이 시리즈들을 모으면
데이터프레임을
만들수 있습니다.

- 시리즈들을 모아 하나의 데이터프레임을 만들 수 있다.
 - 판다스의 DataFrame 클래스를 사용
 - DataFrame을 생성하기 위한 인자로 시리즈들을 나열하면 되는데, 파이썬의 딕셔너리 구조로 열의 이름을 **키**key로, 데이터를 **값**value으로 입력



```
df = pd.DataFrame({'이름': name_series, '나이': age_series,  
                  '학교': school_series, '평점': grade_series})  
print(df)
```

	이름	나이	학교	평점
0	김수안	19	고교	4.35
1	김수정	23	대학	4.23
2	박동윤	22	대학	4.25
3	강이안	19	고교	4.37
4	강지안	16	중학교	4.25
5	박동민	25	대학원	4.50

3.9 판다스로 데이터 읽고 확인하기

- 판다스 모듈은 csv 파일을 읽어들이어서 데이터프레임으로 바꾸는 작업을 간단히 할 수 있는데, 다음과 같이 read_csv 함수를 이용
 - read_csv 함수는 웹 상에 있는 파일도 바로 읽을 수 있다.




```
import pandas as pd

path = 'https://github.com/dknife/ML/raw/main/data/'
file = path+'vehicle_prod.csv'

df = pd.read_csv(file)
```

- CSV 파일은 데이터프레임이 될 수 있도록 각 행이 같은 구조로 되어 있고, 각 열은 동일한 자료형을 가진 시리즈로 되어 있어야 한다.



```
print(df)
```

Unnamed: 0		2007	2008	2009	2010	2011
0	China	7.71	7.95	11.96	15.84	16.33
1	EU	19.02	17.71	15.00	16.70	17.48
2	US	10.47	8.45	5.58	7.60	8.40
3	Japan	10.87	10.83	7.55	9.09	7.88
4	Korea	4.04	3.78	3.45	4.20	4.62
5	Mexico	2.01	2.05	1.50	2.25	2.54

- 위의 표를 살펴보면 각 행에 특정 국가와 연도별 자동차 생산대수 정보를 표시
 - 각 열들은 서로 다른 속성(property)을 나타냄
 - 각 열은 레이블을 가지고 있는데, 여기서는 2007년부터 2011년도의 연도가 레이블
- 데이터프레임에서는 다음과 같이 인덱스(index)와 컬럼스(columns) 객체를 정의하여 사용

csv 파일의 첫 행으로 만들어진 column

	Unnamed: 0	2007	2008	2009	2010	2011
0	China	7.71	7.95	11.96	15.84	16.33
1	EU	19.02	17.71	15	16.7	17.48
2	US	10.47	8.45	5.58	7.6	8.4
3	Japan	10.87	10.83	7.55	9.09	7.88
4	Korea	4.04	3.78	3.45	4.2	4.62
5	Mexico	2.01	2.05	1.5	2.25	2.54

자동생성된 인덱스

- 우리가 첫 행의 첫 열 공간을 비워둔 것은 이 열에 나타나는 국가 이름을 인덱스로 사용하기 위한 것
- 자동으로 0부터 정수를 매겨 인덱스로 쓰지 않고, 국가 이름이 있는 0번 열을 인덱스로 쓰라고 알려주면 좋을 것
 - 판다스의 read_csv() 함수에 index_col이라는 키워드 매개변수에 인자 0을 넘겨주면 첫번째 열이 인덱스로 사용



```
df = pd.read_csv(file, index_col = 0)
print(df)
```

	2007	2008	2009	2010	2011
China	7.71	7.95	11.96	15.84	16.33
EU	19.02	17.71	15.00	16.70	17.48
US	10.47	8.45	5.58	7.60	8.40
Japan	10.87	10.83	7.55	9.09	7.88
Korea	4.04	3.78	3.45	4.20	4.62
Mexico	2.01	2.05	1.50	2.25	2.54

3.10 데이터 시리즈 선택하여 시각화해 보기

- 데이터프레임에서 특정한 열만 선택하는 방법
 - 데이터프레임에서 2007년도 레이블을 가진 열만을 추출하기 위해서는 원하는 컬럼의 레이블 ['2007']을 사용



```
df_my_index = pd.read_csv(file, index_col = 0)
df_no_index = pd.read_csv(file)
print(df_my_index['2007'])
print(df_no_index['2007'])
```

```
China      7.71
EU         19.02
US         10.47
Japan     10.87
Korea      4.04
Mexico     2.01
Name: 2007, dtype: float64
0         7.71
1        19.02
2        10.47
3        10.87
4         4.04
5         2.01
Name: 2007, dtype: float64
```

- 만일 2007년도 와 2008년도, 2009년도의 데이터를 선택하려면 다음과 같이 원하는 열들의 레이블을 리스트로 만들어서 df[]의 인덱스 값으로 전달



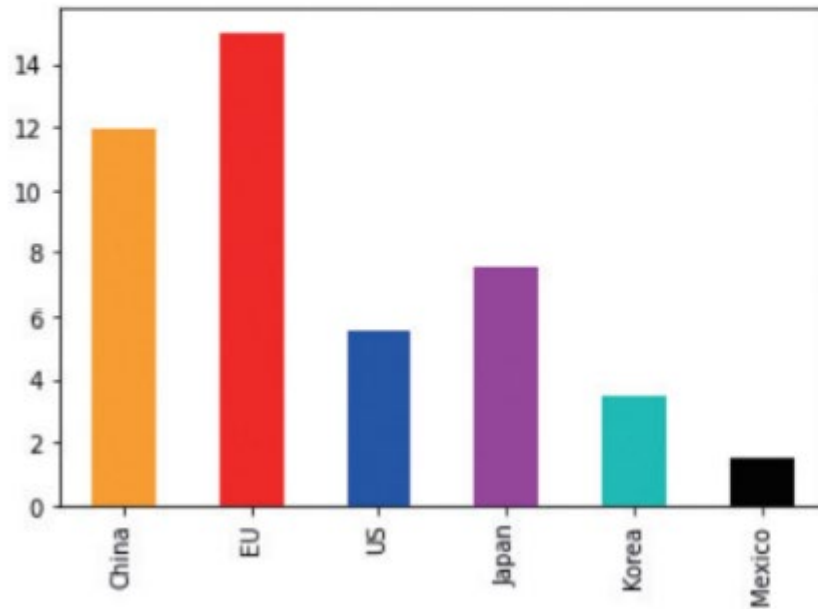
```
print(df[['2007', '2008', '2009']])
```

	2007	2008	2009
China	7.71	7.95	11.96
EU	19.02	17.71	15.00
US	10.47	8.45	5.58
Japan	10.87	10.83	7.55
Korea	4.04	3.78	3.45
Mexico	2.01	2.05	1.50

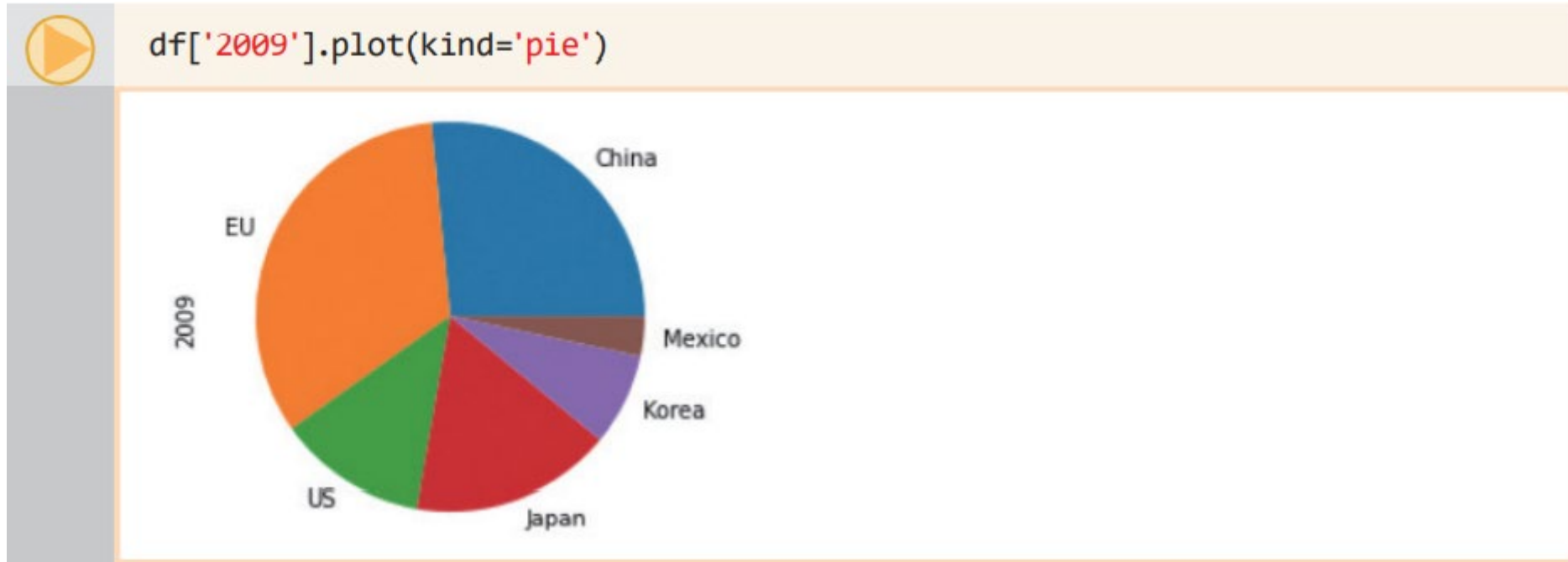
- 우리는 다음과 같은 방법으로 선택된 열을 그래프로 그릴 수 있다.
 - 데이터프레임의 df['2009']을 통해서 2009년도 열을 추출
 - plot() 메소드



```
import pandas as pd
import matplotlib.pyplot as plt
...
df['2009'].plot(kind='bar', color=('orange','r', 'b', 'm', 'c', 'k'))
```




- kind='pie'로 지정하고, color 키워드 인자 부분을 삭제 파이 차트가 그려짐
 - 파이 차트에서는 색상을 지정하지 않는다는 점에 유의



3.11 편리한 데이터 다루기 - 슬라이싱과 열 데이터 추가


- 처음 5행만 얻으려면 head()를 사용
- 마지막 5행만을 얻으려면 tail()을 사용
- 만일 head()와 tail()에 정수를 인자로 넘겨주면 그 정수 만큼의 행



df.head(3)

	2007	2008	2009	2010	2011
China	7.71	7.95	11.96	15.84	16.33
EU	19.02	17.71	15.00	16.70	17.48
US	10.47	8.45	5.58	7.60	8.40

- 일반적인 방법은 다음과 같이 슬라이싱 표기법을 사용하는 것



```
df[2:6]
```

	2007	2008	2009	2010	2011
US	10.47	8.45	5.58	7.60	8.40
Japan	10.87	10.83	7.55	9.09	7.88
Korea	4.04	3.78	3.45	4.20	4.62
Mexico	2.01	2.05	1.50	2.25	2.54

- 만약 'US', 'Korea'와 같은 행의 인덱스를 사용하여 행을 선택하려면 loc[인덱스]를 사용



```
df.loc['Korea']
```

2007	4.04
2008	3.78
2009	3.45
2010	4.20
2011	4.62

Name: Korea, dtype: float64

- 열 선택과 행 선택을 결합하는 방법
 - 예를 들어서 2011년도의 한국과 중국의 자동차 생산 대수를 추출하고자 할 경우에는 `df['2011']` 과 같이 2011년도 열을 추출 한 다음에 다음과 같은 방법을 사용

```
df['2011'][[0, 4]]
```

China	16.33
Korea	4.62

Name: 2011, dtype: float64

- 데이터프레임에서 특정한 요소 하나만을 선택하려면 간단하게 `loc()` 함수에 행과 열의 레이블을 써주면 된다.

```
df.loc['Korea', '2011']
```

4.62

- 판다스를 이용하면 기존에 존재하는 열의 정보를 토대로 새로운 열을 생성할 수 있음
- 우리의 데이터프레임에 2007년부터 2011년도까지의 차량생산 대수의 합을 나타내는 'total'이라는 이름의 열을 생성



```
df['total'] = df.sum(axis = 1)  
print(df)
```

	2007	2008	2009	2010	2011	total
China	7.71	7.95	11.96	15.84	16.33	59.79
EU	19.02	17.71	15.00	16.70	17.48	85.91
US	10.47	8.45	5.58	7.60	8.40	40.50
Japan	10.87	10.83	7.55	9.09	7.88	46.22
Korea	4.04	3.78	3.45	4.20	4.62	20.09
Mexico	2.01	2.05	1.50	2.25	2.54	10.35

3.12 판다스를 이용한 데이터 분석

- 데이터프레임이 저장한 데이터를 분석하려면 describe() 함수를 사용
 - 이 파일에는 한글이 포함되어 있으므로 encoding='CP949'를 통해 한글 인코딩 방식을 알려줌
 - head(3) 메소드로 첫 3 항목과 weather.shape 속성값도 출력



```
import pandas as pd
```

```
path = 'https://github.com/dknife/ML/raw/main/data/'  
weather_file = path + 'weather.csv'
```

```
weather = pd.read_csv(weather_file, index_col = 0, encoding='CP949')  
print(weather.head(3))  
print('weather 데이터의 shape :', weather.shape)
```

	평균기온	최대풍속	평균풍속
일시			
2010-08-01	28.7	8.3	3.4
2010-08-02	25.2	8.7	3.8
2010-08-03	22.1	6.3	2.9

weather 데이터의 shape : (3653, 3)

- 이제 데이터가 담고 있는 내용을 간략히 분석하기 위해 describe() 함수

	평균기온	최대풍속	평균풍속
count	3653.000000	3649.000000	3647.000000
mean	12.942102	7.911099	3.936441
std	8.538507	3.029862	1.888473
min	-9.000000	2.000000	0.200000
25%	5.400000	5.700000	2.500000
50%	13.800000	7.600000	3.600000
75%	20.100000	9.700000	5.000000
max	31.300000	26.000000	14.900000

- describe() 함수를 통해서 각 속성값들의 개수^{count}, 평균값^{mean}, 표준편차^{std}, 최소값^{min}, 최대값^{max} 등을 쉽게 알 수 있음

- mean(), std() 함수를 통해 특정한 분석 값만 볼 수도 있음



```
print('평균 분석 -----')  
print(weather.mean())  
print('표준편차 분석 -----')  
print(weather.std())
```

```
평균 분석 -----  
평균기온      12.942102  
최대풍속      7.911099  
평균풍속      3.936441  
dtype: float64  
표준편차 분석 -----  
평균기온      8.538507  
최대풍속      3.029862  
평균풍속      1.888473  
dtype: float64
```

- 최대풍속 열과 평균풍속 열의 최대값을 max()로 조회할 수 있음
 - min(), count(), mean(), sum() 등과 같은 다른 함수도 사용할 수 있음

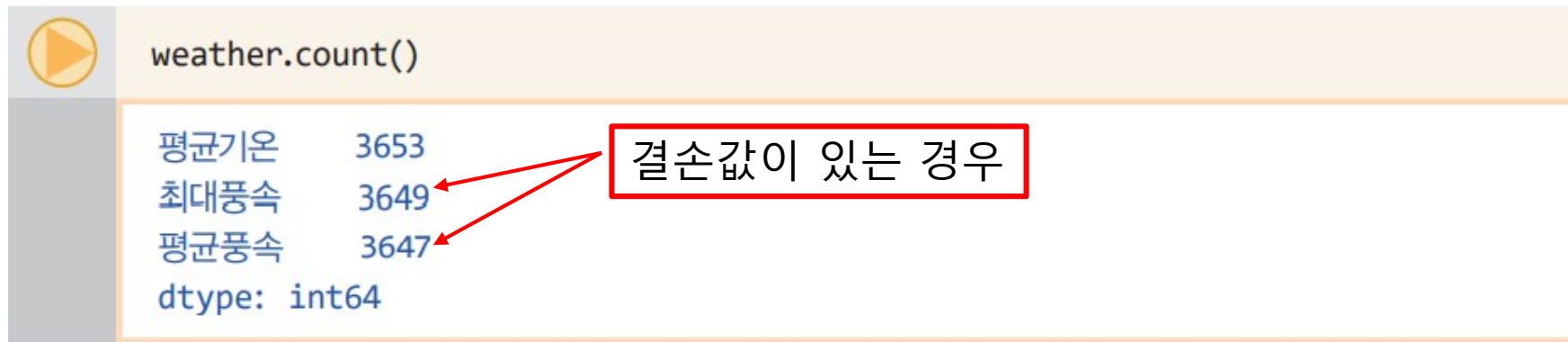


```
weather[['최대풍속', '평균풍속']].max()
```

```
최대풍속    26.0  
평균풍속    14.9  
dtype: float64
```

3.13 데이터 정제와 결손값의 처리

- 데이터과학과 머신러닝에서 사용하는 실제 데이터는 대부분 완벽하지 않고 상당한 수의 결손 값을 가지고 있음
 - 데이터를 처리하기 전에 반드시 거쳐야 하는 절차가 데이터 정제이다.



weather.count()	
평균기온	3653
최대풍속	3649
평균풍속	3647
dtype: int64	

- 어떤 이유로 인해 데이터에 값이 입력되지 못하여 그 항목이 비어 있을 수 있는데 이를 결손값^{missing} data
 - '최대풍속' 열에는 '평균기온'에 비해 4개의 결손값이 더 있는 것

- 판다스에서는 결손값을 Not a Number를 의미하는 NaN으로 나타냄(혹은 NA로 표기함).
- 판다스는 결손값을 탐지하고 수정하는 함수를 제공
- 데이터에 결손값이 있는지를 **불리언**^{boolean} 값으로 반환하는 함수는 isna()



```
missing_data = weather[ weather['평균풍속'].isna() ]  
print(missing_data)
```

	평균기온	최대풍속	평균풍속
일시			
2012-02-11	-0.7	NaN	NaN
2012-02-12	0.4	NaN	NaN
2012-02-13	4.0	NaN	NaN
2015-03-22	10.1	11.6	NaN
2015-04-01	7.3	12.1	NaN
2019-04-18	15.7	11.7	NaN

결손값표시 : NaN

- 결손값을 다루는 방법은 결손값을 가진 행이나 열을 삭제하는 것
 - 판다스에서 `dropna()` 함수를 이용하여 삭제.

축이 0이면 결손데이터를 포함한 행을 삭제하고
축이 1이면 결손데이터를 포함한 열을 삭제한다.

`inplace`가 `True`이면 원본 데이터에서 결손데이터를 삭제하고
`False`인 경우는 원본은 그대로 두고 고쳐진 데이터프레임 반환

`pandas.DataFrame.dropna(axis=0, how='any', inplace=False)`

`how`의 값이 `'any'`이면 결손 데이터가 하나라도 포함되면 제거 대상이 되고,
`'all'`이면 `axis` 인자에 따라서 행 혹은 열 전체가 결손 데이터이어야 제거한다.



- 결손값을 다루는 또 하나의 방법은 결손값을 다른 값으로 교체하는 것
- fillna() 함수를 사용하면 특정한 값을 다른 값으로 교체
 - 결손값을 단순히 0으로 채우게 되면 데이터의 개수가 작을 경우 전체 데이터의 대표값인 평균 값을 왜곡
 - 0대신에 데이터의 평균값으로 채우는 것이 더 나을 것



```
weather.fillna( weather['평균풍속'].mean(), inplace = True)  
print(weather.loc['2012-02-11'])
```

```
평균기온    -0.7  
최대풍속     3.936441  
평균풍속     3.936441  
Name: 2012-02-11, dtype: float64
```

3.14 그룹핑과 필터링


- 데이터 분석을 할 때에는 특정한 값에 기반하여 데이터를 그룹으로 묶는 일이 많다.
 - `groupby()`라는 함수를 사용



```
...  
weather = pd.read_csv(weather_file, encoding='CP949')  
...  
weather['month'] = pd.DatetimeIndex(weather['일시']).month  
means = weather.groupby('month').mean()  
print(means)
```

	평균기온	최대풍속	평균풍속
month			
1	1.598387	8.158065	3.757419
2	2.136396	8.225357	3.946786
3	6.250323	8.871935	4.390291
4	11.064667	9.305017	4.622483
5	16.564194	8.548710	4.219355
6	19.616667	6.945667	3.461000
7	23.328387	7.322581	3.877419
8	24.748710	6.853226	3.596129
9	20.323667	6.896333	3.661667
10	15.383871	7.766774	3.961613
11	9.889667	8.013333	3.930667
12	3.753548	8.045484	3.817097

- 판다스는 특정한 조건을 주어서 데이터프레임의 값을 필터링할 수 있음



```
means['평균풍속'] >= 4.0
```

month	
1	False
2	False
3	True
4	True
5	True
6	False
7	False
8	False
9	False
10	False
11	False
12	False

Name: 평균풍속, dtype: bool

- 특정 조건을 만족하는 데이터만을 추출하는 방식



```
means[ means['평균풍속'] >= 4.0 ]
```

부울리언 인덱싱 혹은
논리 인덱싱이라고 해요,

	평균기온	최대풍속	평균풍속
month			
3	6.250323	8.871935	4.390291
4	11.064667	9.305017	4.622483
5	16.564194	8.548710	4.219355

데이터프레임 df

	Col1	Col2	Col3
Index			
i1	2	A	B
i2	3	A	C
i3	5	D	A
i4	7	G	Y
i5	8	C	Q
i6	10	K	D

df[Col1 % 2 == 0]

index	
i1	True
i2	False
i3	False
i4	False
i5	True
i6	True

df[df[Col1 % 2 == 0]]

	Col1	Col2	Col3
Index			
i1	2	A	B

	Col1	Col2	Col3
Index			
i5	8	C	Q
i6	10	K	D



	Col1	Col2	Col3
Index			
i1	2	A	B
i5	8	C	Q
i6	10	K	D

3.15 데이터 구조의 변경: pivot과 concat

- 데이터 분석을 하다 보면 테이블에서 행과 열의 위치를 바꾼다거나, 어떤 기준을 정해서 집계를 해서 테이블 구조를 변경해야 하는 경우
 - 판다스는 이런 경우를 위해 테이블 구조를 변경하는 함수들을 제공.
 - 대표적인 함수가 pivot() 함수

- item은 상품의 이름이고 type은 상품의 재질, price는 상품의 가격
 - 동일한 상품을 서로 다른 행에 중복하여 두고 있어 자료를 찾기가 쉽지 않음

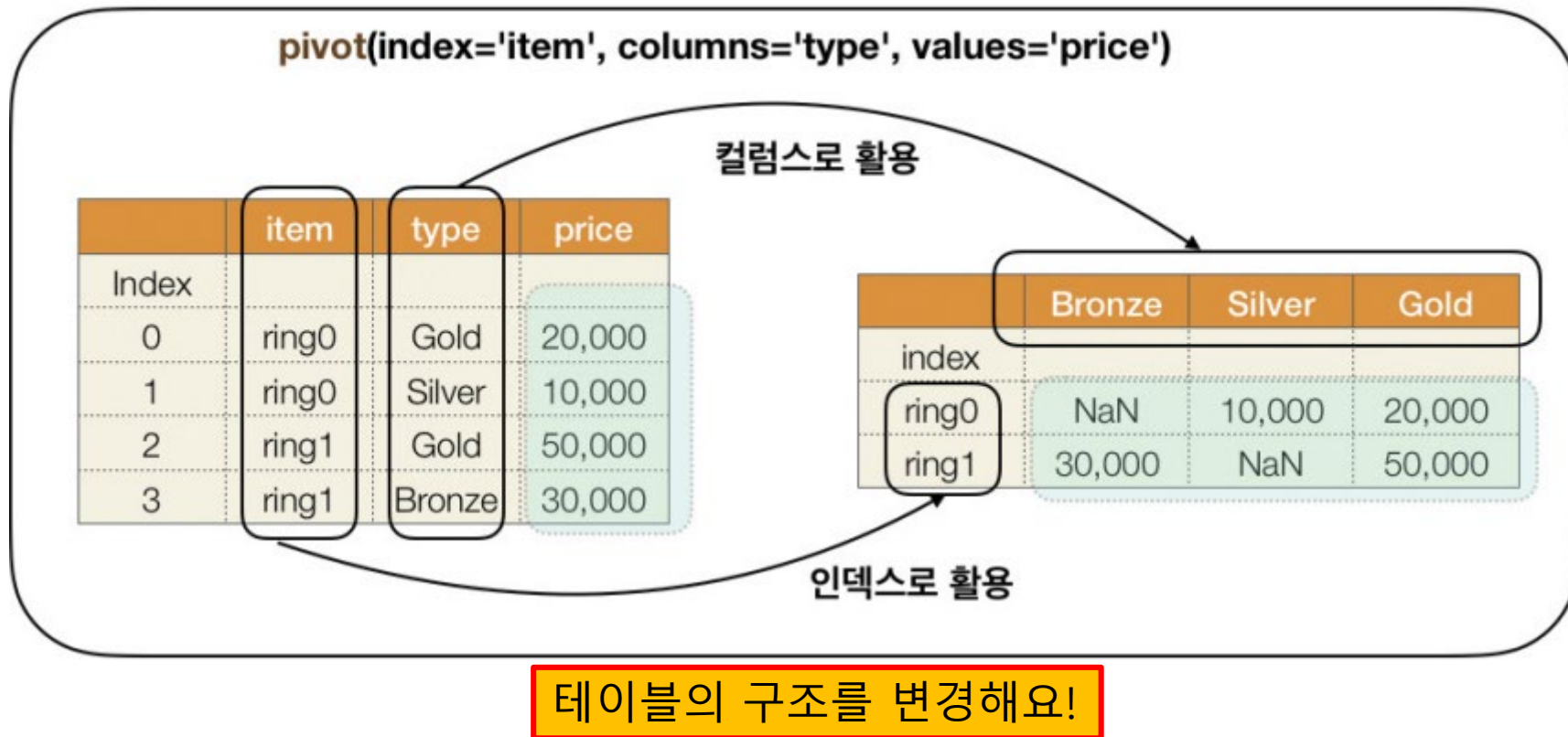


```
import pandas as pd
df_1 = pd.DataFrame({'item' : ['ring0', 'ring0', 'ring1', 'ring1'],
                     'type' : ['Gold', 'Silver', 'Gold', 'Bronze'],
                     'price': [20000, 10000, 50000, 30000]})
```

df_1

	item	type	price
0	ring0	Gold	20000
1	ring0	Silver	10000
2	ring1	Gold	50000
3	ring1	Bronze	30000

- index = 'item'으로 하여 item 열을 인덱스로 사용하고 columns='type'으로 열을 변경함



- 일반적으로 데이터들은 작은 테이블로 나누어져 있는 경우
 - 데이터를 하나로 합치는 방법 가운데 하나인 concat() 함수



```
df_1 = pd.DataFrame( {'A' : ['a10', 'a11', 'a12'],  
                      'B' : ['b10', 'b11', 'b12'],  
                      'C' : ['c10', 'c11', 'c12']} ,  
                      index = ['가', '나', '다'] )  
  
df_2 = pd.DataFrame( {'B' : ['b23', 'b24', 'b25'],  
                      'C' : ['c23', 'c24', 'c25'],  
                      'D' : ['d23', 'd24', 'd25']} ,  
                      index = ['다', '라', '마'] )
```

df_1

	A	B	C
가	a10	b10	c10
나	a11	b11	c11
다	a12	b12	c12

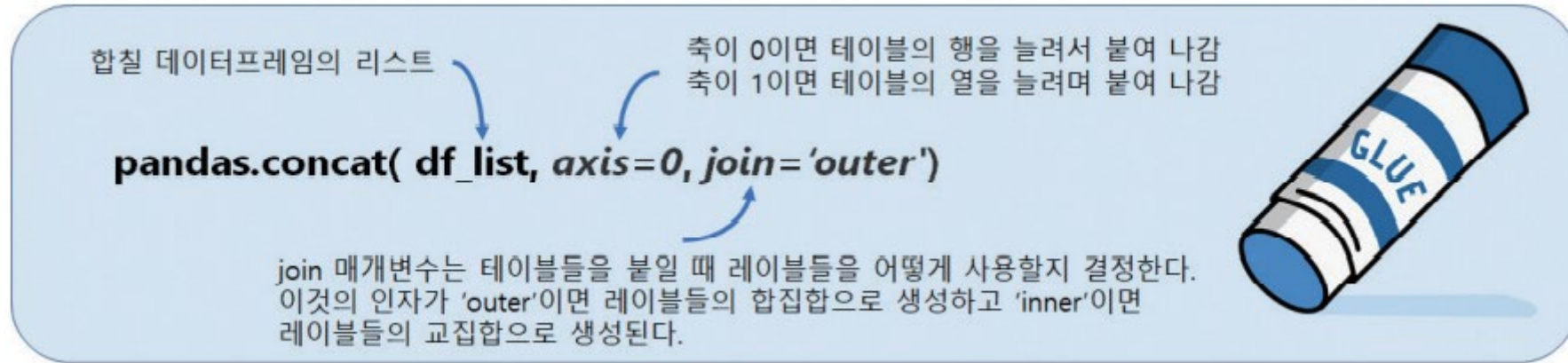
인덱스

df_2

	B	C	D
다	b23	c23	d23
라	b24	c24	d24
마	b25	c25	d25

인덱스

- 결합을 위해 `concat()` 함수를 사용
 - 첫 인자로 데이터프레임의 리스트
 - 두 개의 중요한 키워드 매개변수 `axis`와 `join`의 의미가 그림에 설명



- 디폴트 축 0을 사용할 경우 행을 늘려 데이터 테이블을 붙여 나감
 - 인덱스 '다'는 두 데이터 프레임에 각각 존재했는데, 별도의 행으로 합쳐지는 것을 확인할 수 있음
 - 열은 'outer'를 조인 방식으로 사용, inner는 교집합, outer는 합집합을 생성



```
df_3 = pd.concat( [df_1, df_2])  
print(df_3)
```

	A	B	C	D
가	a10	b10	c10	NaN
나	a11	b11	c11	NaN
다	a12	b12	c12	NaN
다	NaN	b23	c23	d23
라	NaN	b24	c24	d24
마	NaN	b25	c25	d25

3.16 데이터의 병합: merge

- 데이터베이스는 **조인**join이라는 연산을 지원
- 조인 연산과 같은 방식의 데이터 병합을 지원하는 판다스 함수가 merge() 함수

DataFrame.merge(right, how='inner', on=None)

right: 현재의 데이터프레임과 결합할 데이터프레임

how: 결합의 방식 'left', 'right', 'inner', 'outer' 가능

on: 조인 연산을 수행하기 위해 사용할 레이블 (두 데이터프레임 모두에 존재해야 함)

- merge를 적용
 - 조인연산을 'B' 레이블을 가진 열의 데이터를 키로 사용하게 하였다.

df_1				df_2			
	A	B	C		B	C	D
가	a10	b10	c10	다	b23	c23	d23
나	a11	b11	c11	라	b24	c24	d24
다	a12	b12	c12	마	b25	c25	d25

인덱스

인덱스



```
print('left outer \n' , df_1.merge(df_2, how='left', on='B' ) )
print('right outer \n' ,df_1.merge(df_2, how='right', on='B' ) )
print('full outer \n' ,df_1.merge(df_2, how='outer', on='B' ) )
print('inner \n' ,df_1.merge(df_2, how='inner', on='B' ) )
```

left outer

	A	B	C_x	C_y	D
0	a10	b10	c10	NaN	NaN
1	a11	b11	c11	NaN	NaN
2	a12	b12	c12	NaN	NaN

right outer

	A	B	C_x	C_y	D
0	NaN	b23	NaN	c23	d23
1	NaN	b24	NaN	c24	d24
2	NaN	b25	NaN	c25	d25

full outer

	A	B	C_x	C_y	D
0	a10	b10	c10	NaN	NaN
1	a11	b11	c11	NaN	NaN
2	a12	b12	c12	NaN	NaN
3	NaN	b23	NaN	c23	d23
4	NaN	b24	NaN	c24	d24
5	NaN	b25	NaN	c25	d25

inner

Empty DataFrame

- 'outer' 방식이 기본
- 'left'는 'outer' 조인의 결과에서 왼쪽 데이터프레임에 존재하는 키를 가진 것만 뽑아내면 되고, 'right'는 오른쪽 테이블에서 발견되는 키를 가진 것만 뽑으면 됨
- inner는 왼쪽과 오른쪽 테이블 모두에서 발견되는 키를 가져야 추출되는데, 여기에서는 공백
- 'B' 뿐만이 아니라 'C' 레이블도 두 테이블에 동시에 나타남
 - 왼쪽 테이블에서 가져온 것은 'C_x', 오른쪽에서 가져온 것은 'C_y'로 이름을 새로 붙여 테이블을 만듦.



```
df_3 = df_1.merge(df_2, how='outer', on='B')  
print(df_3)
```

	A	B	C_x	C_y	D
0	a10	b10	c10	NaN	NaN
1	a11	b11	c11	NaN	NaN
2	a12	b12	c12	NaN	NaN
3	NaN	b23	NaN	c23	d23
4	NaN	b24	NaN	c24	d24
5	NaN	b25	NaN	c25	d25

- 인덱스를 모두 사용하여 병합의 키가 되도록 하는 방식
- 'outer' 방식이므로 두 인덱스의 합집합이 새로운 인덱스가 됨
- 'inner'라면 교집합으로 '다' 행만 남게 된다.



```
df_3 = df_1.merge(df_2, how = 'outer',  
                  left_index = True, right_index = True )  
print(df_3)
```

	A	B_x	C_x	B_y	C_y	D
가	a10	b10	c10	NaN	NaN	NaN
나	a11	b11	c11	NaN	NaN	NaN
다	a12	b12	c12	b23	c23	d23
라	NaN	NaN	NaN	b24	c24	d24
마	NaN	NaN	NaN	b25	c25	d25

핵심 정리

- 파이썬은 데이터 처리와 분석에 매우 적합한 프로그래밍 언어이다.
- 파이썬은 데이터를 다루기에 편리한 **리스트**와 같은 자료형을 기본으로 제공하며, **슬라이싱**을 통해 효율적으로 데이터 부분집합을 다룰 수 있다.
- 파이썬은 **객체지향 프로그래밍 언어**이며, 객체들이 가진 다양한 메소드를 활용하여 효율적으로 프로그래밍을 할 수 있다.
- 효과적인 데이터 처리를 위해서는 다양한 패키지를 활용할 수 있어야 하는데, **구글 코래버러토리는** 웹 상에서 이러한 패키지들을 미리 설정해 둔 **쥬피터 노트북** 개발 환경을 제공한다.
- **구글 드라이브와 연동**하면 구글 코래버러토리 환경에서도 각자의 데이터에 활용할 수 있다.
- **넘파이**는 **수치 데이터**를 다루는 데에 있어 가장 대표적인 패키지로 데이터 과학 뿐만 아니라 여러 과학분야의 정보 처리에 활용되고 있다.
- 넘파이의 핵심적 자료 구조는 **다차원 배열**로, **모든 원소가 동일한 자료형**을 갖는다는 제약조건을 통해 파이썬의 리스트에 비해 높은 성능으로 데이터 처리가 가능하다.

핵심 정리

- 넘파이는 배열 데이터 처리를 위해 **브로드캐스팅**과 **벡터화 연산**을 지원한다. 이것은 넘파이가 대규모 데이터에 대해 강력한 성능을 보일 수 있는 이유이다.
- 넘파이 배열에 대해서도 파이썬의 **리스트처럼 인덱싱**과 **슬라이싱**을 할 수 있다.
- 넘파이 배열은 쉼표를 사용하는 **넘파이 배열 스타일의 인덱싱**을 지원하며, 이를 이용한 슬라이싱을 통해 행렬이나 고차원 텐서의 일부분을 효과적으로 추출할 수 있다.
- 판다스는 데이터를 **시리즈**와 **데이터프레임**이라는 구조로 관리하며, 데이터 시리즈에 레이블을 부여하여 넘파이 배열보다 편리하게 데이터를 관리할 수 있게 한다.
- 판다스는 **데이터의 가시화**도 잘 지원하며, 이것은 맷플롯립을 통해 구현되어 있다.
- 판다스는 데이터의 **슬라이싱**이나 추가 등을 잘 지원하며 **논리 인덱싱**도 잘 동작한다.
- 데이터를 **그룹핑**하고 나뉘어진 데이터를 **병합**하는 일도 판다스를 통해 쉽게 할 수 있다.