Python의 이해

Python은 컴퓨터 언어로 일반적인 영어 문장과 거의 비슷하여 배우기가 매우 쉽다는 특징을 가지고 있다. 이러한 특성과 더불어 머신러닝(machine learning)과 딥러닝(deep learning)을 위한 함수 및 분석방법에 대한 코드가 공개되어 있고, 대규모의 Python 프로그램 개발자 모임에 힘입어 시간이 갈수록 새롭고 편리한 프로그램 라이브러리가 제공되고 있다.

여기에서 소개되는 Python 언어는 기초통계분석을 위한 Scipy, 통계모형에 기반한 통계분석을 위한 Statsmodels, machine learning을 위한 Scikit-learn, 그리고 딥러닝을 위한 TensorFlow와 Keras를 사용하기 위해 필요한 내용들을 정리한 것이다. 그러므로 전문적인 Python 프로그래머를 위한 Python 사용법은 좀 더 전문적인 교재를 이용하기 바란다.

Python을 사용하기 위해서 먼저 https://www.anaconda.com/distribution 에 들어가서 Anaconda를 설치해야 한다. 설치 후 Anaconda Navigator를 클릭하면 Anaconda dashboard가 나타난다. 우리는 “Jupytor Notebook"과 Spyder"를 사용할 것이다. Jupytor Notebook은 프로그램 cell에 프로그램을 입력하고 실행시키면 즉시 결과를 출력해주는 대화식(interactive) 구조이며, Spyder는 Python 프로그램을 text 형태로 작성하고 Python file을 직접 만드는 형태이다. 일반적으로 처음에는 Jupytor Notebook을 사용하고 Python 프로그램에 익숙해지면 Spyder를 사용한다. Jupyter Notebook과 Spyder는 Anaonda dashboard에 있는 launch 를 클릭하면 바로 사용할 수 있다. Jupyter Notebook은 프로그램 cell에 프로그램을 입력하고 CTRL+ENTER 키를 누르면 결과가 나온다. 새로운 프로그램 cell을 만들기 위해서는 상단부 왼쪽에 있는 + 버튼을 누르면 된다. Spyder는 text editor가 왼쪽에 나타나고 여기에 프로그램을 입력하면 된다. 프로그램 실행은 상단부에 있는 ▶ 표식을 클릭하거나 실행하고자 하는 부분을 dragging한 후 CTRL+ENTER 키를 누르면 실행되어 결과가 오른쪽에 나타난다. 보다 자세한 사항은 VanderPlas (2017)을 참고하거나 Googling을 통해 필요한 사항을 검색하기 바란다.

1. Python 구문(Syntax)

Python은 한 줄에 하나의 명령어를 입력하는 것을 원칙으로 하기 때문에 명령어 끝에 통상적으로 사용하는 세미클론(;)을 사용하지 않는다. 다만, 한 줄에 2개 이상의 명령어가 있으면 ; 으로 구분해준다. 먼저 Jupyter Notebook이나 Spyder에 다음의 명령어를 입력하고 실행하여보자.

```
print('Welcome to Python.')
print('It is fun learning Python.')
print('Welcome to Python'); print('It is fun learning Python.')
```

문자는 반드시 ‘ ’를 이용하여야 하며, ‘ ’와 “ ”는 서로 구분하지 않고 사용한다. 문자를 포함한 정수, 실수 등의 자료타입(data type)은 따로 다루기로 한다. 명령어가 길어서 한 줄에 다 쓸 수 없는 경우에는 ‘\’를 이용한다.

```
sum = 10 + 20 + \
      30 + 40 + \
      30
```

위에서 sum을 실행하는 것과 print(sum)을 실행하는 것은 동일한 결과를 출력한다. 조건문, 순환문(loop), 그리고 함수(function) 등은 하나의 블록(block)으로 구성된다. Python에서 블록의 구성은 4칸의 들여쓰기(indentation)로 구분된다.

```
if 10>3:
    print('This is inside block')
print('This is outside of block')
```

위 조건문은 : 로 시작하고 이 조건문의 블록은 print('This is inside of block')으로 끝나고 있다. 이 조건문 안에 다른 명령어가 있으면 그 명령어는 반드시 4칸의 들여쓰기가 되어 있어야 한다. 만약 더 적거나 많은 들여쓰기를 하면 에러 메시지가 출력된다.

Python에서는 변수(variable), 함수(function), 클래스(class), 그리고 모듈(module)에 이름을 부여한다. 이름 부여(naming)는 A-Z, a-z, 0-9, 그리고 밑줄(underscore)를 이용한다. 변수, 함수, 클래스, 모듈의 정의는 뒤에서 자세하게 다루게 될 것이다.

모듈의 이름은 scipy, sklearn과 같이 모두 소문자를 사용하고, 클래스 이름은 sklearn 안에 logistic regression을 사용하기 위해 만든 LogisticRegressor와 같이

영어 단어의 각 첫 문자를 대문자로 사용한다. 클래스 내부에 일반적으로 정의하는 함수(function) 또는 방법(method)의 이름은 소문자를 사용하며, 변수(variable)의 이름은 person_age와 같이 소문자와 밑줄(underscore)을 이용하여 사용한다. 이름이 한 개 또는 두 개의 밑줄로 시작하는 변수도 있지만 이는 클래스를 설명할 때 다시 다루기로 한다. 이와 같은 이름 부여(naming)는 Python 프로그램에서 관행적으로 사용되는 것으로 이름의 형태에 의해 변수, 함수, 클래스, 모듈 여부를 구별할 수 있으므로 숙지하기 바란다. 이러한 관행으로 인해, 예를 들어, Letter와 letter는 Python에서 완전히 다른 이름으로 인식된다.

2. 변수와 자료형태

변수는 자료를 저장하는 위치로 해석되고 Python에서 변수는 문자, 정수, 실수, 그리고 논리(boolean) 자료를 자유롭게 입력할 수 있다. 자료의 입력은 '='에 의해 간단하게 수행된다.

```
name = 'Mike'
age = 15
score = 102.5
passed = True
```

첫 번째 변수명은 name으로 'Mike'라는 문자형 자료(string)를 저장하고 age는 정수(integer)를, score는 실수(float)를, pass는 논리연산자(boolean)를 저장하고 있다. 문자형 자료는 반드시 따옴표(' ' 또는 " ")를 사용하여야 하며, 논리연산자는 True 또는 False를 할당한다. 변수의 자료타입을 알고 싶은 경우

```
type(age)
```

를 입력하면 자료의 타입이 정수(integer)라는 것을 출력한다.

자료값의 모임으로 정의된 자료의 형태(자료구조)가 있다. list, tuple, 그리고 dictionary 이다.

LIST

list는 여러 개의 자료 값을 한 개의 변수에 정의할 수 있다. [ ]를 이용하여 정의되고 자료값은 ","로 구분한다. 예를 들어,

```
cars = ['Honda', 'Toyota', 2002, 2015]
```

와 같이 cars라는 변수에 list 자료를 저장할 수 있으며, list를 구성하는 자료들은 그 타입이 동일할 필요가 없다. machine learning에서 가장 널리 쓰이는 numpy 모듈의 array는 list와 동일한 형태이지만 list와는 다르게 개별 자료의 타입이 서로 같아야 한다. list는 자료를 모아놓은 것이지만 numpy의 array는 행렬자료인 벡터이기 때문이다.

```
print(len(cars))
```

는 변수 cars에 할당된 list의 길이를 출력하는 명령으로, 4를 결과값으로 출력하게 된다.

```
colors = ['red', 'green', 'white', 'yellow']
nums = list(range(10))
print(nums)
nums_1 = list(range(50,101))
print(nums_1)
```

위 프로그램은 list 자료를 만드는 과정을 보여주고 있다. nums는 0-9의 숫자로 만들어진 list 자료를 위한 변수이고 nums_1은 50-100로 만들어진 list 자료를 위한 변수이다. “list"를 쓰게 되면 list 자료로 만들라는 의미이고, range(a, b)는 a이상 b 미만인 정수를 의미한다.

list는 자료를 편집할 수 있는 기능이 있다. x[a:b]는 리스트 자료 x에서 a부터 시작하여 b-1 까지의 요소(element)를 의미한다. 예를 들어

```
colors_1 = colors[:2]
print(colors_1)
```

을 하면 colors_1=['red', 'green']이 출력된다. Python에서는 list 자료의 위치가 0부터 시작되기 때문에 colors_1은 colors의 자료 중 0-1 까지 두 개의 자료를 갖게 된다. 비슷하게

```
colors_2 = colors[-2:]
```

는 끝에서 2번째 자료부터 마지막 자료까지를 의미하므로 colors_2=['white', 'yellow']가 된다. 또한

```
colors_3 = colors[1:-2]
```

는 colors_3=['green']가 된다. colors의 1번째 열부터 마지막 2번째 자료이전까지를 의미하기 때문이다.

```
colors_4 = colors[1:3]
```

는 colors의 1번째 자료부터 3번째 자료 직전(∴2번째까지) 자료들로 구성된 list가 된다. 즉, colors_4=['green', 'white']가 된다. 마지막으로

```
colors_5 = colors[2]
```

colors_5=['white']가 된다.

행렬은 행과 열로 이루어진 자료형태로 이러한 자료를 2차원 자료라고 한다. list는 이러한 행렬도 표현할 수 있다. python의 열과 행은 반드시 0부터 시작한다는 것을 명심하자. 예를 들어

```
data = [[1, 15, 20, 36], [41, 20, 54, 47], [75, 46,68, 2]]
```

와 같이 list 안의 list로 표현된다. 또한

```
data_1 = data[1]
```

은 data_1=[41, 20, 54, 47]이 되고

```
data_2 = data[1][2]
```

이면 data_2=[54]가 된다. list 변수의 행렬 편집은 numpy 모듈의 array의 행렬 편집과 다르다는 것을 명심해야 한다. 이에 대한 논의는 numpy에서 자세하게 다룰 것이다.

list는 추가하기(appending), 제거하기(remove), 옆에 붙이기(concatenating), 그리고 정렬하기(sorting)와 같은 편집 기능을 가지고 있다. 이는 다음의 프로그램을 실행하면 쉽게 이해할 수 있을 것이다.

```
colors = ['blue', 'white', 'yellow', 'red', 'black']
colors.append('orange')
print(colors)
```

를 실행하면 colors=['blue', 'white', 'yellow', 'red', 'black', 'orange']가 된다. color 변수명에 append() 함수를 사용하여 'orange'가 추가된 것이다.

```
colors.remove('white')
```

를 실행하면 colors=['blue', 'yellow', 'red', 'black', 'orange']가 된다. 바로 직전

에 colors 변수에 'orange'가 추가된 것을 전제로 한 결과이다.

```
colors.sort()
```

를 실행하면 colors=[‘black', 'blue', 'orange', 'red', 'yellow']가 된다. 바로 직전에 colors 변수에 'orange'가 추가된 것을 전제로 한 결과이다.

TUPLE

Tuple은 list와 거의 동일한 역할을 한다. 그러나 가장 큰 차이점은 Tuple은 괄호 없이 또는 ( )를 이용해서 정의한다는 것과, list에서 가능했던 자료의 추가, 제거, 교체 등이 불가능하다는 것이다. 예를 들어,

```
abc_1 = ['a', 'b', 'c', 'd']
abc_2 = ('a', 'b', 'c', 'd')
abc_1[2] = 'p'
abc_2[2] = 'p'
print(abc_1); print(abc_2)
```

를 실행하면 print(abc_1)은 ['a', 'b', 'p', 'd']를 출력하지만 abc_2[2] = 'p'는 에러 메시지를 출력하게 된다.

list에서 사용한 자료 편집 기능은 Tuple에서도 동일하게 적용된다. 예를 들어,

```
print(abc_2[1:4])
```

를 실행하면 ('b', 'c', 'd')를 출력한다.

Tuple의 유용한 기능 중 하나는 짝짓기 기능이다. 예를 들어,

```
name = 'A', 'B', 'C'
type(name)
K_1, K_2, K_3 = name
print(K_1); print(K_2); print(K_3)
```

를 실행하면 차례대로 A, B, C를 출력하게 된다. 이러한 짝짓기 기능은 머신러닝에 자주 등장하므로 기억해 두도록 하자. 또한 다음과 같은 스왑(swap) 기능도 있다.

```
a, b = 11, 22; print(a, b)
a, b = b, a
print(a, b)
```

위 코드를 실행하면 첫 번째 print(a,b)는 11, 22를 출력하고, 두 번째 print(a,b)는 22, 11을 출력한다.

DICTIONARY

마지막 자료형태(data type)는 dictionary이다. Dictionary는 key-value 쌍으로 구성되어 있고, “{ }”를 사용하여 정의되며, key와 value는 ‘:’ 으로 구분한다. 예를 들어,

```
cars = {'name':'kia', 'model':2019, 'color':'white'}
print(cars['name'])
print(cars.keys())
print(cars.values())
print(cars.items())
```

를 실행하면

```
[Out]   kia
        dict_keys(['name', 'model', 'color'])
        dict_values(['kia', 2019, 'white'])
        dict_items([('name', 'kia'), ('model', 2019), ('color', 'white')])
```

를 출력하게 된다.

Dictionary 자료는 추가, 최신화(update), 제거(delete) 기능이 있다.

```
cars['capacity'] = 1500
print(cars)
```

를 실행하면 cars={'name': 'kia', 'model': 2019, 'color': 'white', 'capacity': 1500}가 되며,

```
cars['model'] = 2020
print(cars)
```

를 실행하면 cars={'name': 'kia', 'model': 2020, 'color': 'white', 'capacity': 1500}이 되고,

```
del cars['model']
print(cars)
```

를 실행하면 cars={'name': 'kia', 'color': 'white', 'capacity': 1500} 으로 변경하게 된다.

3. Python의 연산

연산은 산술연산, 비교연산, 할당(assignment) 및 존재(membership)연산이 있다.

산술연산은 더하기(+), 빼기(-), 곱하기(*), 나누기(/), 나머지(%), 제곱승(**) 등이 있다. 예를 들어

```
N1 = 10
N2 = 5
print(N1+N2)
print(N1-N2)
print(N1*N2)
print(N1/N2)
print(N1%N2)
print(N1**N2)
```

의 출력 결과는 차례대로 15, 5, 50, 2, 0, 100,000 이다. N1%N2는 N1을 N2로 나눈 나머지이므로 0이 된다.

논리연산은 and, or, not이 있다. 예를 들어

```
N1 = True
N2 = False
print(N1 and N2)
print(N1 or N2)
print(not(N1 and N2))
```

를 실행해보면 N1 and N2는 모두 True가 아니므로 False를 출력하고, N1 or N2는 둘 중 하나가 True이므로 True를 출력한다. not(N1 and N2)는 not N1 or not N2 이므로 not N1 또는 not N2가 True이면 True이므로 True를 출력한다.

비교연산은 ==, !=, >, <, >=, <= 등이 있다. 여기서 ==는 두 변수가 동일한지를 비교하는 연산이며 !=는 동일하지 않은지를 비교하는 연산이다. 예를 들면

```
N1 = 10
N2 = 5
print(N1==N2)
print(N1!=N2)
print(N1>N2)
print(N1>=N2)
print(N1<N2)
print(N1<=N2)
```

의 출력 결과는 차례대로 False, True, True, True, False, False가 된다.

할당연산은 =, +=, -=, *=, %=, /=, **= 이 있다. 예를 들어

```
N1=10; N2=5
```

일 때, 각 할당연산의 실행 결과는 아래와 같다.

연산(코드)	결과	설명
R=N1+N2	R=15	R에 N1+N2를 할당
N1+=N2	N1=15	N1+N2를 N1에 할당
N1-=N2	N1=5	N1-N2를 N1에 할당
N1/=N2	N1=2	N1/N2를 N1에 할당
N1%=N2	N1=0	N1%N2를 N1에 할당
N1*=N2	N1=50	N1*N2를 N1에 할당
N1**=N2	N1=100,000	N1**N2를 N1에 할당

존재연산은 in, not in이 있으며

```
cars = ['Hyundai', 'kia', 'Audi', 'Benz', 'Honda']
print('Hyundai' in cars)
print('BMW' in cars)
print('BMW' not in cars)
```

의 출력 결과는 차례대로 True, False, True가 된다.

4. 조건문과 반복문

조건문과 반복문은 블록구조를 가지고 있다. 블록은 ':'으로 시작하고 4칸의 들여쓰기(indentation)로 구성된다.

조건문

조건문은 if / else / elif로 블록을 구성하고 제어한다.

```
N1 = 10
N2 = 5
if N1>N2:
    print('N1 is greater than N2')
```

위 프로그램은 N1>N2가 True이면 블록 내의 “N1 is greater than N2"를 출력하라는 의미이다. 이 경우 조건문 N1>N2가 True이므로 블록 내의 print 명령어가 수행되어 ”N1 is greater than N2“가 출력된다.

다음은 다양한 형태의 조건문 예제 및 이에 대한 설명이다.

예제 코드	설명
N1 = 10 N2 = 20 N3 = 30 if N2<N1 or N3>N2: print('N2<N1 or N3>N2')	논리연산자를 사용. N3>N2가 True이므로 print 명령어를 수행

if N2>N1:
  if N2>N3:
  print('N2>N1 and N3>N2')
	조건문 안에 조건문이 있는 형태.
	if N2>N1 and N2>N3:와 동일한 조건문.
	조건문이 False이므로 print 명령어를 수행하지 않음.

if N1>N2:
  print('N1>N2')
else:
  print('N2>N1')
	if / else 형태로, else도 블록을 형성함.
	N2>N1이 False이므로 print('N2>N1')을 실행.

다음은 if / elif / else의 조건문 예를 보여주고 있다.

```
if N1>N2:
    print('N1>N2')
elif N2>N3:
    print('N2>N3')
elif N3>N2:
    print('N3>N2')
else:
    print('None of the conditions are true.')
```

위 프로그램은 if, elif, else가 각각의 블록을 구성하고 있으며, N3>N2가 True이므로 print('N3>N2')를 실행하게 된다. 참고로, elif 안에 if나 elif문을 추가적으로 넣을 수 있다. 이를 내포된(nested) elif라고 한다. 물론 이 경우에도 if / elif 각각이 블록 구조를 가지게 된다.

반복문

반복문에는 "for" 반복문과 "while" 반복문이 있다. "for" 반복문은 주어진 자료의 모임(즉, list, tuple, dictionary)의 각 원소에 대해 반복을 실행하고, "while" 반복문은 조건이 만족될 때까지 반복을 실행한다.

for 반복문의 형태는

```
for i in [List]:
    statement 1
    statement 2
```

이다. 예를 들면

```
cars = ['AB', 'CD', 'EF', 'GH']
for car in cars:
    print(car)
```

을 실행하면 'AB', 'CD', 'EF', 'GH'를 차례대로 한 줄에 하나씩 출력 한다.

```
for i in range(10):
    print(i)

for i in range(50, 101):
    print(i)
```

는 0-9를 첫 번째 for 반복문에서 print하고, 두 번째 for 반복문에서는 50-100을 한 줄에 하나씩 출력한다.

```
for c in 'Hello world':
    print(c)
```

를 실행하면 H e l l o w o r l d를 한 줄에 한 칸씩 (o와 w 사이의 빈 칸도) 출력한다.

“while” 반복문의 형태는

```
while(expression=True):
    statement 1
    statement 2
```

이다. 예를 들면

```
i = 1
while i<11:
    print(i)
    i += 1
```

은 i를 1씩 증가시키면서 1-10을 출력한다.

다음 프로그램은 구구단 9단을 출력한다.

```
i = 1
while i<10:
    print('9x' + str(i) + '=' + str(i*9))
    i = i+1
```

위 프로그램의 print() 안에서는 옆에 붙이기(concatenating)를 위한 '+'와 정수형 자료를 문자형 자료로 바꾸는 str()을 이용하고 있다. 예를 들어 i=3 일 때에는 $9\times3=27$을 출력한다

"for"와 "while" 반복문은 명령어 break, continue 및 함수 enumerate(), zip()을 유용하게 사용할 수 있다.

break는 반복문을 빠져나올 때 사용하는 명령어이다.

```
for i in range(1,11):
    if i>5:
        break
    print(i)
```

위 프로그램은 i=1~5일 때까지 i를 1씩 증가시키면서 출력하고, i가 5보다 큰 값인 경우 break가 실행되어 for 반복문을 빠져나온다.

continue는 반복문 전체를 빠져 나오는 것이 아니라 해당 반복에서 명령문 실행을 건너뛸 때 사용된다. 반복문 내에서 continue를 만나면 그 아래 명령어들을 수행하지 않고 다음 반복으로 건너뛴다. 아래 프로그램에서 i는 짝수인 경우만 출력된다.

```
for i in range(1,11):
    if (i%2 != 0):
        continue
    print(i)
```

enumerate()는 List내 자료값에 대응하는 순서값이 필요할 때 사용한다. 예를 들어

```
player = ['John', 'Taylor', 'Ronado', 'Messi']
player_1 = list(enumerate(player,1))
print(player_1)
```

는

```
[Out]    [(1, 'John'), (2, 'Taylor'), (3, 'Ronado'), (4, 'Messi')]
```

를 출력한다. enumerate(player, 1)에서 1은 순서값의 시작을 1로 하라는 옵션이다. 출력된 결과를 보면 첫 번째 자료값 'John'의 순서값이 1임을 확인할 수 있다. enumerate()는 tuple로 자료를 반환함을 알 수 있다.

```
for num, p in enumerate(player,1):
    print('Player number ' + str(num) + ' name:', p)
```

위 프로그램에서 num과 P는 enumerate(player)가 반환하는 tuple의 1번째 원소(순서값)와 2번째 원소(자료값)을 각각 나타낸다.

zip()은 list, tuple, dictionary에 의한 자료의 모임을 짝짓기 해주는 기능을 가지고 있다. 예를 들어

```
cars = ['Sonata', 'Toyota', 'Ford', 'Benz', 'Kia']
nations = ['Korea', 'Japan', 'America', 'Germany']
for c, n in zip(cars, nations):
    print('%s is made in %s' %(c,n))
```

는 실행 결과로

```
[Out]    Sonata is made in Korea
         Toyota is made in Japan
         Ford is made in America
         Benz is made in Germany
```

를 출력한다. zip()은 cars의 원소와 nations의 원소를 tuple 형태로 1:1로 짝을 지어줌을 확인할 수 있다. 짝짓기의 대상이 되는 자료 모임들의 길이가 서로 다른 경우에는 길이가 작은 것 기준으로 짝짓기를 수행한다. 여기에서는 cars의 길이보다 nations의 길이가 더 작으므로 nations의 길이 기준으로 짝짓기라 수행되었고, cars의 'Kia'는 무시되었다.

위 프로그램에서 print() 안에서 사용한 %s는 문자형 변수를 따옴표 안으로 불러서 사용하기 위한 형식이고, 따옴표 밖의 %(c, n)과 함께 사용된다. 따옴표 안의 두 개의 %s 중 첫 번째는 %(c, n)의 c에 대응하고, 두 번째 %s는 %(c, n)의 n에 대응한

다. 참고로, 출력하려는 변수가 정수형일 때에는 %d를 사용하고, 실수형인 때에는 예를 들어 소수점 이하 셋 째 자리까지 표시하는 실수인 경우, %.3f를 사용한다.

```
N = 10
a = 2.543
b = 'car'
print('A typical example of data types is %d, %.2f, and %s.' %(N, a, b))
```

```
[Out]    A typical example of data types is 10, 2.54, and car.
```

5. 객체지향성 프로그램(object-oriented programming)

Python은 객체지향성 프로그램이다. 여러 개의 속성(attribution), 함수(function), 방법(method)을 하나의 클래스(class)로 정의하고, 이 클래스를 객체화한 후, 이 객체(object)를 통해 클래스 내에 정의된 속성, 함수, 방법을 불러서 사용하는 프로그램을 객체지향성 프로그램이라고 한다.

5.1 함수(function)

함수는 다음과 같이 "def"를 이용하여 정의된다.

```
def function_name():
    code line 1
    code line 2
```

아래와 같이 정의된 가장 간단한 형태로 출발하여보자.

```
def welcome():
    print('Welcome to Python')
```

이 함수를 부르기 위해서는 단순히

```
welcome()
```

을 하면 welcome()을 실행하여 "Welcome to Python"을 출력한다. 아주 약간 더 복잡한 경우를 살펴보자.

```
def p_inf(name, age, gender):
    print ('Person name: '+name)
    print ('Person age: ', age)
    print ('Person gender: '+gender)
    print ('----------------------')

p_inf('James', 20, 'Male')
p_inf('Suzan', 19, 'Female')
```

위 프로그램의 실행 결과는

```
[Out]   Person name: James
        Person age:  20
        Person gender: Male
        ----------------------
        Person name: Suzan
        Person age:  19
        Person gender: Female
        ----------------------
```

이다. 여기에서 함수 p_inf() 안에 있는 name, age, gender를 함수의 모수(parameter)라고 한다. 모수를 입력받아서 이를 바탕으로 산출된 새로운 값을 반환(return)하는 형태로 함수를 정의할 수도 있다. name과 gender는 문자형 변수이므로 +name, +gender 또는 “, name” 또는 “, gender”로 쓸 수 있다. age는 정수형 변수로”, age“만 가능하다.

```
def summ(num1, num2):
    result = num1 + num2
    return result

test = summ(20, 30)
print(test)
```

위 프로그램의 실행 결과는 50으로, 함수 summ() 내에서 계산되어 result 값이 반환되어 변수 test에 저장된 것이다.

```
def update_1(newlist):
    newlist.append([20, 25, 30])
    return

uplist = [5, 10, 15]
update_1(uplist)
print(uplist)
```

위 프로그램은 함수 update_1()의 매개변수 newlist는 단순히 reference 기능을 하고 있음을 보여주고 있다. uplist = [5, 10, 15] 이지만 update_1(uplist) 실행에 의해 변수 uplist의 값이 함수 update_1() 안에서 [5, 10, 15, [20, 25, 30]]으로 바뀌어서 반환되므로, print(uplist)는 [5, 10, 15, [20, 25, 30]]을 출력하게 된다.

함수를 정의할 때 def를 사용하지 않는 방법이 있다. 아주 간단한 함수를 처리하기 위한 것으로, 아래와 같이 “lambda" 명령어를 사용한다.

```
result = lambda n1, n2, n3: n1+n2+n3
result(1, 2, 3)
```

위 프로그램의 출력 결과는 6이다. 즉, lambda의 모수는 n1, n2, n3이고, n1+n2+n3을 반환해주는 함수이다.

5.2 클래스(Class)의 구조요소와 역할

클래스의 구성요소를 잘 이해하게 되면 Python의 scipy, statsmodels, sklearn과 같은 라이브러리(library)를 사용할 수 있는 기초 지식을 습득한다고 할 수 있다. 클래스는 속성(attribute)과 실제적으로 함수인 방법(method)으로 구성되어 있다.

```
class Player:
    name = 'John'
    age = 22
    gender = 'male'

    def stand(self):
        print("he stands")

    def run(self):
        print("he runs")
```

Player라고 이름 붙여진 클래스는 3개의 속성과 2개의 방법으로 구성되어 있다.

def 밖에 있는 name, age, gender는 Player의 속성이고, stand()와 run()은 Player의 방법이며, 방법의 첫 모수 self는 디폴트이다. 방법의 첫 모수 self가 의미하는 바는 stand()와 run()은 class Player에 소속된 방법(또는 함수)이라는 것이다. 이제 클래스 Player를 어떻게 객체화하고 사용하는지 살펴보자.

```
male_1 = Player()
player_name = male_1.name
print(player_name)
male_1.run()
```

클래스 Player를 male_1으로 객체화하였고, 객체화된 male_1에 의해 속성인 name을 불러내었다. print(player_name)의 결과는 "John"이다. 세 번째 줄의 명령어는 male_1에 의해 run() 함수(방법)을 불러들이고 이로 인해 "he runs"가 출력된다.

속성은 클래스 속성과 사례 속성(instance attribute)으로 나눌 수 있다. 클래스 속성은 클래스 객체들 간에 공유할 수 있는 속성이고, 사례 속성은 해당 객체에만 적용되는 속성을 말한다. 다음의 예제는 클래스 속성과 사례 속성을 보여주고 있다.

```
class Player:
    person_count = 0  # 클래스 속성

    def infor(self, name, age, gender):
        self.name = name  # 사례 속성
        self.age = age
        self.gender = gender
        Player.person_count +=1
        print('Infor for player ' + self.name + ' has been stored.')
```

여기에서 person_count는 클래스 속성이고, self.name, self.age, self.gender는 사례 속성이다. 클래스 속성은 Player.person_count와 같이 클래스 이름을 통해 불러내고, 사례 속성은 self를 이용하여 호출한다. 이 두 속성의 차이를 보이기 위해 다음을 실행하여 보자.

```
player1 = Player()
player1.infor('John', 22, 'male')
print('Player count: ' + str(player1.person_count))

player2 = Player()
player2.infor('Messi', 30, 'Messi')
print('Player count: ' + str(player2.person_count))
```

위 프로그램에서 볼 수 있듯이 Player 클래스는 player1과 player2로 두 번 객체화 되었다. player1.infor()로 Player 클래스 내의 함수를 호출하였고, “Infor for player John has been stored”가 출력되었다. 세 번째 줄의 print()의 결과로 “Player count: 1”이 출력되고, Player2.infor()로 “Infor for player Messi has been stored”가 출력된다. 그러므로 infor() 함수 내의 self.name은 객체별로 각각 "John" 또는"Messi"로 바뀌게 됨을 알 수 있다. 즉, 사례속성(instance attribute)은 객체 간에 연계되지 않고 독립적으로 적용된다는 것을 알 수 있다. 그러나 맨 마지막 줄의 print() 명령어의 실행 결과는 "Player count: 2"가 되어 값이 증가되어 있음을 확인할 수 있다. 즉, 클래스 속성(class attribute)은 player1 객체와 player2 객체 간에 연계되어 있음을 알 수 있다.

클래스를 호출하면 자동으로 사례속성을 만들어 내는 특수 방법이 있다.

```
class Food:
    def __init__(self, name, expiry_year, expiry_month):  # 생성자
        self.name = name
        self.expiry_year = expiry_year
        self.expiry_month = expiry_month

        print("initialized instance varibles.")

    def expiry_date(self):
        print('The expiration date is ' + str(self.expiry_month) + \
              '/' + str(self.expiry_year))

food_1 = Food('Pizza', 2018, 12)  # 클래스를 호출하는 동시에 속성에 값이 할당됨
food_1.expiry_date()
```

클래스 Food를 food_1으로 객체화하였는데, 가장 큰 특징은 Food() 대신에 클래스 Food의 모수 3개를 직접 지정하고 있음을 알 수 있다. 이는 앞과 뒤에 2개의 밑줄

(underscore)로 만들어진 __init__() 덕분에 가능하다. 이 특수함수(또는 방법)는 클래스가 호출될 때 자동으로 실행되며, 주로 사례변수를 구동(initiate)시키는 역할을 한다. 그러므로 __init__()에서 지정된 name, expiry_year, expiry_month를 클래스 호출 시 Food('pizza', 2019, 12)와 같이 직접 모수를 지정할 수 있으며, 위 프로그램의 결과로

```
[Out]
        initialized instance variables
        The expiration date is 12/2019
```

를 출력하게 된다.

앞의 프로그램에서 food_2 = Food('Bread', 2019, 20)으로 입력하면 food_2.expiry_date()를 실행하면 “The expiration date is 20/2019”가 출력된다. 하지만 20개월은 존재하지 않으므로, 이를 클래스 내에서 제어할 필요가 있다. 이를 위해

```
class Food:
    def __init__(self, name, expiry_year, expiry_month):
        self.name = name
        self.expiry_year = expiry_year
        self.expiry_month = expiry_month

        print("initialized instance varibles.")

    @property
    def expiry_month(self):
        return self.__expiry_month

    @expiry_month.setter
    def expiry_month(self, expiry_month):
        if expiry_month<1:
            self.__expiry_month = 1
        elif expiry_month>12:
            self.__expiry_month = 12
        else:
            self.__expiry_month = expiry_month

    def expiry_date(self):
        print('The expiration data is ' + str(self.expiry_month) + \
              '/' + str(self.expiry_year))
```

와 같이 클래스 Food를 다시 정의하고 아래 프로그램을 실행하면

```
food_2 = Food('Bread', 2019, 20)
food_2.expiry_date()
```

실행결과는

```
[Out]    initialized instance varibles.
         The expiration data is 12/2019
```

가 된다. 그 이유는 @property와 @expiry_month_setter 때문이다. 논리적 오류를 방지하기 위해 속성(attribute)에 속성의 성질을 @property로 부여할 수 있다. @property에 소속된 방법의 이름은 제어하려는 속성의 이름과 동일해야 한다. 이 예제에서는 expiry_month이므로 @property의 방법은 def expiry_month()가 되고, 사례속성(instance attribute)은 self.__expiry_month로 지정한다. expiry_month 앞에 밑줄이 두 개 있는 변수를 개인변수(private variable)이라고 한다. 이 변수는

클래스 내에서만 사용할 수 있는 변수이다. 반면, 밑줄이 없는 self.name과 같은 속성은 공공변수(public variable)라고 하며, foof_2.name과 같이 클래스 밖에서 사용할 수 있다. 예를 들어

```
print(food_2.name)
```

의 실행 결과 “Bread”를 출력하게 된다. 클래스 내에 밑줄 한 개로 시작하는 변수가 있는데, 이 역시 클래스 내에서만 사용하는 변수라고 생각하면 된다.

@property에 의해 property가 만들어지면 다음은 @expiry_month.setter로 expiry_month를 구체적으로 어떻게 제어하는지를 규정하게 된다.

클래스의 또 하나의 중요한 기능은 자식 클래스를 가질 수 있다는 것이다. 이는 프로그램 개발자 입장에서 매우 유용한 기능이다. 이미 개발된 클래스를 부모 클래스로 연계하여 새로운 기능을 추가할 수 있기 때문이다. 자식 클래스는 한 개 이상의 부모 클래스를 가질 수 있고, 부모 클래스의 함수(또는 방법)와 속성을 바로 사용할 수 있다. 예를 들어

```
class Parent A:
    def defA:
        statement

class Parent B:
    def defB:
        statement

class Child(Parent A, Parent B)
    def defC:
        ststement

inherit = Child()
inherit.defA()
inherit.defB()
inherit.defC()
```

은 Child라는 자식 클래스는 Parent A와 Parent B라는 부모 클래스와 연계되어 있음을 보여준다. Child 클래스는 inherit으로 객체화되었고, inherit.defA()와 inherit.defB()와 같이 Parent A와 Parent B의 방법을 직접 호출할 수 있다. 또한, 부모 클래스에 있는 사례속성(instance attribute)도 사용할 수 있다. 예를 들어

```
class Parent:
    def __init__(self, aa, bb):

class Child(Parent):
    def __init__(self, aa, bb, cc):
        Parent.__init__(self, aa, bb)
        self.cc = cc
```

는 자식 클래스인 Child에서 부모 클래스의 속성 aa와 bb를 Parent.__init__(self, aa, bb)를 이용하여 불러서 사용하고, 추가적인 속성 cc를 정의하고 있다.

6. 넘파이(numpy)

넘파이(numpy)는 다음 장에서 설명할 판다스(pandas)와 더불어 분석자료를 생성하고 편집, 가공, 그리고 연산을 위한 라이브러리이다. 판다스는 넘파이 위에 만들어졌으므로 (넘파이는 판다스의 부모 클래스 역할을 한다) 넘파이의 이해는 판다스의 이해와 직결된다. 넘파이는(판다스와 비교하여) 주로 함수의 기능을 하기 때문에 “np.함수()”의 형태로 사용된다. 여기에서 np는 numpy의 약자이다. 반면 판다스는 주로 클래스의 기능을 하기 때문에 객체를 만들고 ‘객체.함수()’의 형태로 주로 사용된다. 물론 넘파이도 객체화하여 사용할 수 있으나, 속도가 느리고 Python 자체의 내장함수와 충돌이 일어날 가능성이 있기 때문에 일반적으로 numpy의 객체화는 사용하지 않는다. 넘파이는 딥러닝에서 자료편집과 연산에 사용되는 라이브러리이다.

넘파이의 사용은 아래와 같이 넘파이를 불러옴으로써 시작할 수 있다.

```
import numpy as np
```

여기에서 np는 numpy의 약자이며 일반적으로 Python에서는 이 약자를 사용한다. 넘파이는 Python의 list로부터 넘파이 array를 만든다. 예를 들어

```
np.array([1, 5, 7, 6])
```

는 array([1, 5, 7, 6])을 출력하고

```
np.array([1.25, 1, 5, 7])
```

은 array([1.25, 1. , 5. , 7.])를 출력하게 된다. 넘파이는 하나의 통일된 자료타입(type)를 사용하기 때문에 정수(integer)인 1, 5, 7은 각각 1.0, 5.0, 7.0과 같이 실수(float)형으로 자동으로 변경하여 출력한다.

다음은 넘파이를 이용하여 1차원, 2차원, 3차원 자료를 만들어내는 유용한 함수의 예제들이다. 딥러닝에서는 자료의 차원을 텐서(tensor)라고 말하며 1차원을 1텐서, 2차원을 2텐서, 3차원자료를 3텐서라고 한다. 그러므로 하나의 숫자는 0텐서라고 말하며 4차원 자료는 4텐서 자료라고 말한다. 5개의 0으로 구성된 1차원 자료는

```
np.zeros(5, dtype=int)
```

```
[Out]    array([0, 0, 0, 0, 0])
```

이고, 실수 1로 구성된 2×3 행렬은

```
np.ones([2,3], dtype=float)
```

```
[Out]    array([[1., 1., 1.],
                [1., 1., 1.]])
```

이다. 리스트 안에 리스트가 구성요소로 있기 때문에 이를 2차원(two-dimensional) array라고 한다. 이와 같은 0 array나 1으로 구성된 행렬은 딥러닝에 있는 모수의 초기치로 사용된다.

0에서 10 사이의 정수를 2 간격으로 자료를 만들고 싶을 때(10은 포함되지 않음)에는

```
np.arange(0, 10, 2)
```

```
[Out]    array([0, 2, 4, 6, 8])
```

와 같이 할 수 있다. np.linspace()는 그림을 그릴 때 x축이나 y축의 값을 정의할 때 유용하게 쓰이는 함수로,

```
np.linspace(0, 1, 5)
```

```
[Out]    array([0. , 0.25, 0.5 , 0.75, 1.  ])
```

와 같이 사용할 수 있다.

통계학에서 특정 분포를 따르는 확률변수의 값을 생성(generating) 할 때,

```
np.random.seed(12)
```

으로 지정하면 반복된 확률변수 값을 동일하게 생성하고 싶을 때 사용하는 것으로, 12 이외의 다른 값을 지정하여도 된다.

```
np.random.random((3,3))
```

은 Uniform(0,1)으로부터 3×3 행렬을 생성하는 함수이다.

```
np.random.normal(0, 1, (3,3))
```

은 평균이 0이고 분산이 1인 정규분포로부터 3×3 행렬을 생성하는 함수이며,

```
np.random.randint(0, 10, (3,3))
```

은 0-9 범위의 정수로부터 3×3 행렬을 구성하고자 할 때 사용하는 함수이다.

다른 확률분포로부터의 값 생성은 np.random? 또는 np.random?? 를 Jupiter Notebook이나 spyder 프로그램창에 입력하고 실행시키면, χ^2, F, $Poisson$ 등 원하는 확률변수 생성을 위한 함수를 찾을 수 있다. 이러한 "?"와 "??"는 Python의 어느 속성 및 함수에나 적용되므로 유용하게 사용할 수 있다. 특히, Jupiter Notebook에서 np.random에서 np.r를 입력하거나 또는 np.r + Tab 키를 numpy 내의 r로 시작하는 모든 함수를 출력하여 주고, np.*an*를 입력하면 numpy 내의 중간에 an을 가진 모든 함수를 출력하여 주므로, 사용하고자 하는 함수가 정확하게 기억나지 않을 때 유용하게 사용할 수 있다. 프로그램 창에서 np. 를 쓰면 넘파이 함수가 팝업으로 나타나고 필요한 함수를 클릭하여 선택할 수 있다.

```
rg = np.random.RandomState(40)
```

위 프로그램은 np.random.seed(40)과 동일한 역할을 하지만, np.random.RandomState 클래스를 rg로 객체화하였다는 차이점이 있다. 이러한 객체화를 통해 rg.random, rg.normal, rg.randint로 Uniform(0,1), 정규분포 등 확률변수를 각각 생성할 수 있다.

넘파이에 의해 만들어진 array 자료는 객체화하여 자료의 속성을 알 수 있다.

```
import numpy as np
np.random.seed(1)
x = np.random.randint(10, size=(2,3,4))
x
```

```
x.ndim
```

```
x.shape
```

```
x.size
```

실행 결과는 차례로

```
[Out]   array([[[5, 8, 9, 5],
                [0, 0, 1, 7],
                [6, 9, 2, 4]],

               [[5, 2, 4, 2],
                [4, 7, 7, 9],
                [1, 7, 0, 6]]])
[Out]   3
[Out]   (2, 3, 4)
[Out]   24
```

이다. 여기서 3은 차원을, 24는 3차원 행렬을 구성하는 원소의 총 개수를 의미한다.

넘파이 array의 접근과 편집은 1차원 자료의 경우 제2장에서 설명한 list 자료와 거의 동일하다. 예를 들어

```
x = np.array([0,3,5,2,4,6])
```

에서 x[1]은 3을 의미하고 x[1:3]은 array([3,5]), x[1:-2]는 array([3, 5, 2])가 된다.

x[::2]는 array([0, 5, 4])를 출력한다. 이러한 기능은 list 자료형식에는 존재하지 않는 자료 접근방법이다. 넘파이 array의 편집(slicing)은 기본적으로

x[start:stop:step]

의 형식을 따른다. 이는 start 요소부터 stop-1 요소까지를 의미하고 step에 지정한 수만큼 스텝을 건너뛰라는 명령어이며, 생략 시 start=0, stop=맨 마지막 자료, 그리고 step=0이다. x[:2]는 array([0, 3])이다. x[::-1]는 array([6, 4, 2, 5, 3, 0])으로 x[::-1]에서 맨 마지막의 "-"는 역순(reverse)의 의미를 갖는다. 그러므로 x[5::-2]는 array([6, 2, 3])이 된다. 역순으로 5번 자료부터 투스텝씩 자료를 구성하라는 의미이기 때문이다.

이러한 규칙은 자료가 2차원, 3차원인 경우에도 동일하게 적용된다. 즉, x가 2차원 자료일 때는

x[start:stop:step, start:stop:step]

이고, 3차원 자료일 때는

x[start:stop:step, start:stop:step, start:stop:step]

을 이용하여 자료를 편집(slicing)하게 된다. 예를 들어

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

일 때,

```
x[:2, :2]
```

은 0-1행과 0-1열을 의미하므로

```
[Out]    array([[1, 2],
                [4, 5]])
```

가 된다. 또한 x[:, 0]은 첫 번째 열을, x[0, :]은 첫 번째 행을 의미한다, 특히 x[0]은 x[0,:]과 같다. x[1,2]는 6이다. list 자료형태에서는 x[1][2]으로 한 것과 다름을 알 수 있다. 한편,

```
x[0, 0] = 99
```

로 놓으면 x는

```
[Out]    array([[99, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
```

와 같이 변경된다.

만약

```
x1 = x[:2, :2]
```

으로 하면 x1은

```
[Out]    array([[99,  2],
                [ 4,  5]])
```

가 되고,

```
x1[1, 1] = 99
```

로 하면

```
[Out]    array([[99,  2],
                [ 4, 99]])
```

가 된다. 그런데 print(x)를 실행하면 x가

```
[Out]    [[99  2  3]
          [ 4 99  6]
          [ 7  8  9]]
```

로 변경되어 있음을 알 수 있다. 그 이유는 x1은 단지 x의 객체이므로 객체를 변경하

면 x도 바로 변경될 수 밖에 없기 때문이다. 이를 해결하기 위해서는 copy() 함수를 사용하면 된다. 즉,

```
x1_copy = x[:2, :2].copy()
```

으로 정의하면 x1_copy를 변경하여도 x에는 반영되지 않는다.

6.1 넘파이(numpy)의 차원변경, 분할, 덧붙이기

넘파이의 중요한 기능 중에 하나는 reshape() 함수를 이용하여 차원을 쉽게 조절할 수 있다는 것이다.

```
x = np.array([1,2,3])
x.reshape((1,3))
```

의 실행 결과는

```
[Out]    array([[1, 2, 3]])
```

이 된다. 원래의 1차원 자료 [1, 2, 3]을 reshape((1,3))을 이용하여 2차원 행벡터로 변경시킨 것을 알 수 있다. x[np.newaxis, :]도 같은 기능을 한다. x.reshape((3,1))과 x[:, np.newaxis]는 x를 2차원 열벡터로 변경하게 된다.

```
x = np.random.randint(10, size=(100,3,4))
x.shape[1]
```

를 실행하면 0-9 범위의 정수로 만들어진 100×3×4 size의 3차원 자료가 생성된다. 대부분의 이미지 자료는 3차원으로 구현된다. x.shape[1]은 “3”을 출력하여 x 변수의 두 번째 shape의 크기를 나타낸다. 첫 번째 index는 관측치의 크기, 두 번째와 세 번째는 pixel 자료로서, 일반적으로 이 자료를 2차원으로 전환하여 실제 분석을 수행하게 된다.

```
x1 = x.reshape((100,12))
x2 = x.reshape((-1,12))
```

x의 두 번째와 세 번째 index의 총 크기가 3×4=12이므로, x를 100×12인 자료로 변환한다. x1은 100개의 관측치를 가지고 12개의 변수로 구성된 자료로 해석할 수 있다. x2는 x1과 동일한 2차원 자료이다. 3차원 자료인 x의 총 자료의 수는 $100 \times 3 \times 4 = 1200$이므로 이를 2차원으로 변환할 때 열의 수를 12로 지정하면 “-1”로 표시된 행의 수를 자동으로 100으로 계산해 주는 유용한 기능이다. 동일하게

x.reshape((100,-1))도 x2와 동일한 결과를 출력한다.

한 개의 자료를 두 개 이상으로 나누는 것은 머신러닝에서 일반적으로 이용된다. 모형적합의 일반화를 위해, 자료를 학습데이터(training data)와 시험데이터(testing data)로 나누어 모형을 학습데이터에 적합시키고 이 적합된 모형을 시험데이터에 적용한다. 학습된 모형이 시험데이터에서도 잘 작동하면 이 모형은 성능이 좋다고 한다. 다음의 간단한 예제는 하나의 자료를 2개 이상으로 나누는 넘파이 함수이다.

```
x = [1, 2, 3, 4, 5, 6, 7, 8]
x1, x2, x3 = np.split(x, [2,6])
print(x1, x2, x3)
```

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

위 프로그램은 index를 [0, 2), [2, 6), [6, 마지막 index]로 분류한 결과를 보여주고 있다.

```
x = np.random.random((4,4))
x1, x2 = np.vsplit(x, [2])
print(x1)
print(x2)
```

를 실행하면 x1은 x의 0행과 1행으로 이루어진 2×4 행렬이고, x2는 x의 2행과 3행으로 구성된 2×4 행렬이다.

```
[Out]    [[0.10003841 0.22572587 0.35833974 0.14195943]
          [0.78481057 0.3724655  0.06184169 0.72403182]]
         [[0.34972937 0.48880047 0.70703175 0.54284659]
          [0.24979053 0.75282184 0.42721264 0.94107892]]
```

또한,

```
y1, y2 = np.hsplit(x, [2])
```

를 실행하면 y는 x의 0열과 1열로 구성된 4×2 행렬 y1과, x의 0열과 1열로 구성된 4×2 행렬 y2로 분해(split)된다. vsplit은 vertical축을 자른다는 의미고 hsplit은 horizontal축으로 나눈다는 의미이다. python의 명령어는 사람이 행하는 것으로 주로 목적어(즉 ~을, ~으로)의 의미로 쓰이기 때문이다.

두 개 이상의 자료소스가 있을 때 분석을 위해 하나의 자료로 합칠 필요가 있다. 이를 위해 사용하는 넘파이 함수는 np.concatenate()와 np.hstack(), 그리고

np.vstack()이 있다. 다음은 몇 가지 간단한 예제와 실행 결과를 보여주고 있다.

```
x = np.array([1,2])
y = np.array([2,3])
z = np.array([4,5])
np.concatenate([x, y, z])
```

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

```
xx = np.array([[1,2,3], [4,5,6]])
yy = np.array([[2,4,6], [8,7,6]])
np.concatenate([xx, yy])
```

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

```
np.concatenate([xx, yy], axis=1)
```

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

```
zz = np.array([55,66,77])
np.vstack([zz,xx])
```

```
[Out]    array([[55, 66, 77],
                [ 1,  2,  3],
                [ 4,  5,  6]])
```

```
ww = np.array([[100],[200]])  # 열벡터
np.hstack([ww,xx])
```

```
[Out]    array([[100,   1,   2,   3],
                [200,   4,   5,   6]])
```

위 예제 프로그램들에서 볼 수 있듯이, np.concatenate() 보다는 np.hstack() 또는 np.vstack()이 보다 명료한 자료붙이기 방법이라고 할 수 있다.

6.2 넘파이(numpy) 연산

연산의 가장 기본적인 개념은 브로드케스팅(broadcasting)이다. 브로드케스팅의 개

념을 다음의 예제를 통해 살펴볼 수 있다.

```
x = np.array([1, 2, 3, 4])
print(x+3)
print(x-3)
print(x*3)
print(x/3)
print(x**3)
print(-(0.5*x+1)**2)
```

```
[Out]   [4 5 6 7]
        [-2 -1  0  1]
        [ 3  6  9 12]
        [0.33333333 0.66666667 1.         1.33333333]
        [ 1  8 27 64]
        [-2.25 -4.   -6.25 -9.  ]
```

넘파이 aray에 3을 더하면 이 덧셈은 element-wise하게 수행된다. 즉, x+3 = [1+3, 2+3, 3+3, 4+3]이 된다. 다른 연산도 동일하게 적용되고, 이를 브로드케스팅 연산이라고 한다. 추가적으로 e^x는 np.exp(x), 3^x는 np.power(3,x), $\ln(x)$는 np.log(x), $\log_2(x)$는 np.log2(x), 그리고 $\log_{10}(x)$는 np.log10(x)를 사용하여 브로드케스팅으로 계산할 수 있다. 행렬의 경우도 동일한 연산법칙이 적용된다.

```
M = np.ones((3, 3))
M1 = M + 1
M1
```

```
[Out]   array([[2., 2., 2.],
               [2., 2., 2.],
               [2., 2., 2.]])
```

```
a = np.array([1, 2, 3])
M2 = M + a
print(M2)
```

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

위 프로그램에서 array a가 row by row로 덧셈이 실행되었음을 알 수 있다.

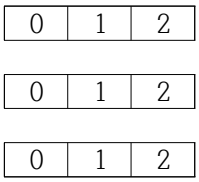
좀 더 재미있는 브로드케스팅 연산은 다음과 같다.

```
a = np.arange(3)
b = np.arange(3)[:, np.newaxis]
print(a)
print(b)
```

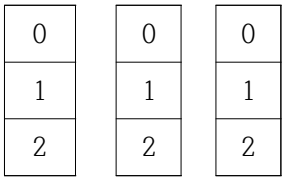
```
[Out]   [0 1 2]
        [[0]
         [1]
         [2]]
```

```
a+b
```

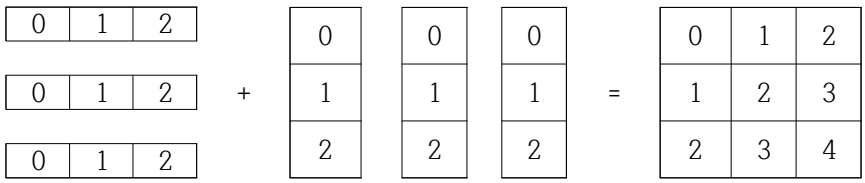
```
[Out]   array([[0, 1, 2],
               [1, 2, 3],
               [2, 3, 4]])
```



위 프로그램에서 a+b는 a = [0, 1, 2]가 가 되고 b = [[0], [1], [2]]가



이 되어, 이들 두 확장자가 합쳐지면



으로 브로드케스팅 연산이 이루어짐을 알 수 있다.

통계분석의 첫 단계는 자료의 특성을 파악하기 위한 기초통계량의 산출이다. 대표적인 기초통계량은 평균, 중위수, 표준편차, 최소값, 최대값, 분위수 등이다. 넘파이는 이들 기초통계량을 쉽게 계산할 수 있는 함수를 제공한다.

```
x = np.random.randint(10, size=(3,4))
print(x)
```

```
[Out]    [[2 7 6 7]
          [3 0 3 3]
          [3 1 2 0]]
```

참고로, 위 프로그램의 실행결과는 매 실행마다 달라진다.

```
np.sum(x)
np.min(x, axis=1)
np.mean(x, axis=0)
np.std(x, axis=0)
```

np.sum(x)의 결과는 54로 x 행렬 원소들의 총 합계를 나타내고, np.min(x, axis=1)의 결과는 [2,2,1]로 각 행의 최소값을 보여주고 있다. 여기에서 axis=1은 열을 의미하고 “열에 걸쳐(across columns)” 구하게 되므로 각 행의 최소값을 구하게 된다. np.mean(x, axis=0)에서 axis=0은 “across rows"이므로 각 열의 평균을 구하게 되어 [2.67, 2.67, 3.67, 3.33]이며, 표준편차는 [0.47, 3.09, 1.70, 2.87] 이다.

```
x_std = (x - np.mean(x, axis=0))/np.std(x, axis=0)
np.sum(x_std, axis=0)
```

위 프로그램을 실행하면 x_std는 각 열 별로 표준화하였기 때문에 np.sum(x_std, axis=0)의 결과는 [0, 0, 0, 0]이 된다.

기초통계량을 계산하기 위한 넘파이 함수를 정리하면, np.sum(), np.prod(), np.mean(), np.std(), np.var(), np.min(), np.max(), np.argmin(), np.argmax(), np.median(), np.percentile()이 있다. np.prod()는 행렬 요소들의 곱을 의미하고, np.var()는 분산, np.argmax는 최대값의 index를 찾는 함수이다.

넘파이에서 결측값은 NaN으로 표기되는데, 이를 제외하고 기초통계량을 계산하는 함수는 “np.nan+함수”의 형태를 가진다. 예를 들어 np.nansum(), np.nanmean(), np.nanmax()와 같이 사용하면 결측치를 제외하고 기초통계량을 구하게 된다. 결측치 문제는 pandas에서 구체적으로 논의하게 될 것이다.

6.3 논리 인덱싱(Boolean indexing)과 순서 정렬(Sorting)

논리 인덱싱은 분류(classification) 등 기계학습에서 데이터 정리 및 추출을 위한 유용한 기법이다. 먼저 다음과 같은 예제부터 출발하여보자.

```
x = np.array([[1,2,3,4], [5,6,7,8]])
x >= 5
```

결과는

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

가 된다.

```
np.sum(x>=5)
np.sum(x>=5, axis=1)
```

위 프로그램을 실행하면 결과는 4와 array([0, 4])가 된다. 논리적 연산에서 True는 1, False는 0 이므로, np.sum(x>=5)는 4가 되고, np.sum(x>=5, axis=1)은 행별로 5 보다 크거나 같은 요소의 개수를 출력하게 된다.

이상치 또는 오류를 찾는데 유용한

```
np.any(x>100)
np.all(x<7, axis=0)
np.all(x==100)
```

이 있다. 실행 결과는 순서대로

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

가 출력된다.

```
x = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12], [13,14,15,16]])
y = np.array([1,0,1,0])
x[y==1, :]
```

을 실행하면 array([[1, 2, 3, 4], [9, 10, 11, 12]])을 출력한다. 즉, 0 번째, 2 번째 행으로만 이루어진 행렬이 구성된다. 유사하게, 1 번째, 3 번째 열로 구성된 행렬은 x[:,y==0]이 될 것이다.

지금까지 살펴본 논리적 인덱싱에 더하여 넘파이 인덱싱의 뛰어난 성능을 추가적으로 살펴보자.

```
x = np.array([11, 22, 33, 44, 55, 66, 77, 88, 99])
ind1 = np.array([2, 3, 8])
ind2 = np.array([[2,1,3], [5,2,7]])
```

으로 array 자료를 만든 후,

```
x1 = x[ind1]
x2 = x[ind2]
print(x1); print(x2)
```

를 실행하면

```
[Out]    [33 44 99]
         [[33 22 44]
          [66 33 88]]
```

가 출력된다. ind1에 의해 자료의 위치로 x1을 만들고, reshape()을 사용하지 않고도 1차원 자료 x로부터 2차원 자료 x2를 만들어낼 수 있다.

인덱싱에도 브로드케스팅 용법을 사용할 수 있다.

```
x = np.arange(12).reshape((3,4))
x
```

```
[Out]    array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11]])
```

이고

```
y1 = np.array([1, 2, 2])
y2 = np.array([2, 0, 1])
x[y1, y2]
```

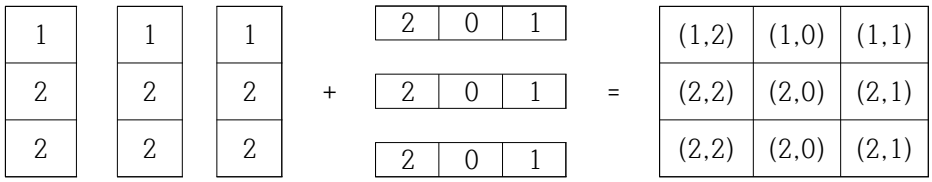
에서 x[y1, y2] 은 x의 (1,2), (2,0), (2,1) 요소이므로, 실행 결과는 array([6, 8, 9])가 된다.

```
x[y1[:,np.newaxis], y2]
```

는

```
[Out]    array([[ 6,  4,  5],
                [10,  8,  9],
                [10,  8,  9]])
```

을 출력한다. y1[:,np.newaxis]는 array([[1], [2], [2]])으로 열벡터이고 y2=array([2, 0, 1]) 이므로, 브로드캐스팅 용법에 따라 인덱싱이



이 되므로, 3×3 행렬로 shape도 달라지고 구성요소도 조절 가능하다.

넘파이의 순서 정렬 함수는 np.sort()이다.

```
x = np.array([2, 4, 3, 1, 5])
np.sort(x)
```

실행 결과는

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

가 된다. 또한

```
ind = np.argsort(x)
print(ind)
```

를 실행하면 각 요소의 위치 index를 array([3,0,2,1,4])를 출력한다.

2차원 자료에서는 np.sort(x, axis=0) 또는 np.sort(x, axis=1)로 지정하여 열 별 또는 행 별로 각각 sorting을 할 수 있다.

7. 판다스(Pandas)

판다스는 넘파이(Numpy)를 기초로 만들어진 라이브러리이다. 이러한 이유로 판다스는 넘파이가 가지고 있는 대부분의 특성을 가지고 있다. 넘파이와 비교하여 판다스의 가장 큰 차이점은 행렬자료에 행과 열의 이름을 부여할 수 있다는 것이다. 판다스 자료는 행렬이 행과 열의 이름을 가지고 있으므로 자료의 편집, 인덱싱(indexing), 통합, 분류에 유연한 특성을 가지고 있다. 또 하나의 큰 특징은 넘파이는 동일한 타입(type)의 데이터만 허용되지만, 판다스는 다양한 타입을 동시에 사용할 수 있다. 판다스는 1차원 자료에 이용하는 클래스 Series와 2차원 자료에 사용하는 클래스 DataFrame이 있다.

```
import pandas as pd    #판다스 Pakage를 불러오는 명령어
X1=pd.Series([0.1,0.2,0.3,0.4], index=['a','b','c','d'])
print(X1)
```

```
Out[1]:  a    0.1
         b    0.2
         c    0.3
         d    0.4
          dtype: float64
```

판다스는 위와 같이 Index에 이름을 부여할 수 있으며, 생략하면 index가 0,1,2,3으로 부여된다. Series가 클래스이므로 위와 같이 X1으로 객체화하여 사용하게 된다.

```
X1.values
X1.index
```

```
Out[1]:  array([0.1, 0.2, 0.3, 0.4])
Out[2]:  Index(['a', 'b', 'c', 'd'], dtype='object')
```

시행결과는 array([0.1, 0.2, 0.3, 0.4])가 되어 넘파이 array와 동일하게 되고, X1.index는 index(['a', 'b', 'c', 'd'], dtype='object')를 출력하게 된다.

```
X1['a':'c']
X1[0:3]
```

는 모두 동일한 결과를 출력한다.

```
Out[1]:  a    0.1
         b    0.2
         c    0.3
         dtype: float64
```

X1['a':'c']는 부여된 index로 자료를 잘라냈고(slicing), X1[0:3]은 0~2 번째 자료까지 잘라낸 것을 알 수 있다. X1['a':'c']에서 'a':'c'를 구체적 index 라고 하고 0:3은 행 0,1,2를 의미하는 의미상 index 라고 한다.

pd.Series는 Dictionary 형식 데이터를 1차원 자료로 환원할 수 있다.

```
population1={'서울':9700000,
             '부산':4500000,
             '인천':4000000,
             '광주':2000000,
             '대구':2500000}
popu=pd.Series(population1)
print(popu)
```

```
Out[1]: 서울     9700000
        부산     4500000
        인천     4000000
        광주     2000000
        대구     2500000
            dtype: int64
```

으로 dictionary 형식 데이터에서 key가 Series의 index로 변한 것을 알 수 있다.

```
X2=pd.DataFrame([[1,2,3],[4,5,6],[7,8,9]],columns=['a','b','c'],index=[1,2,3])
```

```
Out[1]:    a  b  c
        1  1  2  3
        2  4  5  6
        3  7  8  9
```

2차원 자료구성을 위한 DataFrame은 클래스이므로 X2로 객체화하였고 행의 이름은 [1,2,3]으로 열의 이름은 [a,b,c]로 부여하고 있다.

앞의 예제에서 population1을

```
pd.DataFrame(popu,columns=['pop'])
```

```
Out[1]:          pop
        서울  9700000
        부산  4500000
        인천  4000000
        광주  2000000
        대구  2500000
```

을 출력하게 된다 이는 pd.Series로 만들어진 1차원 array를 2차원 행렬자료로 변환시킬 뿐만 아니라 열 벡터의 열 이름을 부여할 수 있다.

```
ind1={'서울':95,
      '부산':80,
      '인천':85,
      '광주':75,
      '대구':80}
ind=pd.Series(ind1)
y=pd.DataFrame({'pop':popu,'자립도(%)':ind})
print(y)
```

```
Out[1]:          pop  자립도(%)
        서울  9700000       95
        부산  4500000       80
        인천  4000000       85
        광주  2000000       75
        대구  2500000       80
```

을 출력한다. 즉 pd.Series로 만든 2개의 1차원자료 popu과 ind를 dictionary형태로 자료화한 후 DataFrame을 적용하면 공통의 index인 서울,부산,인천,광주,대구를 index로 하고 dictionary 데이터 형태의 key인 ‘pop’와 ‘자립도(%)’을 열이름으로 하는 2차원 자료를 생성할 수 있다.

판다스 자료의 편집(slicing)은 행과 열 이름이 있으므로 매우 명료하게 사용할 수 있다. 예를 들어 열의 이름과 행의 이름을 그대로 사용하여 data를 살펴볼 수 있다.

```
y['자립도(%)']
y['부산':'광주']
```

```
Out[1]: 서울    95
        부산    80
        인천    85
        광주    75
        대구    80
        Name: 자립도(%), dtype: int64

Out[2]:          pop  자립도(%)
        부산  4500000       80
        인천  4000000       85
        광주  2000000       75
```

pd.DataFrame의 편집은 Numpy의 slicing도 사용할 수 있다. 다음의 예를 고려하여 보자.

```
import numpy as np #numpy package를 불러오는 명령어
W=pd.DataFrame(np.arange(12).reshape(3,4), columns=['a','b','c','d'])
print(W)
```

```
Out[1]:    a  b   c   d
        0  0  1   2   3
        1  4  5   6   7
        2  8  9  10  11
```

자료의 편집에 사용하는 loc와 iloc의 차이에 대하여 알아보자.

```
W.loc[0:1,:]
W.iloc[0:1,:]
W.iloc[0:2][‘c’]
```

```
Out[1]:      a   b   c   d
         0   0   1   2   3
         1   4   5   6   7

Out[2]:      a   b   c   d
         0   0   1   2   3
Out[3] 0  2
       1  6
       Name: c, dtype: int64
```

속성(attribute) loc는 구체적인 자리수를, iloc는 의미상의 자리수를 의미하며, loc속성은[a:b]는 a에서 b자리 수까지를, iloc속성은 [a:b]는 a에서 b-1 자리 수까지를 의미한다. 여기에서 유의해야 할 것은 W[1:3]과 같이 행만 slicing하면 오류없이 작동하지만 W[1:3,‘a’]와 같이 행과 열의 slicing은 작동하지 않는다는 점이다. 반드시 W.loc[1:2,‘a’] 또는 W.iloc[1:3,0]으로 해야만 slicing이 작동하는 점을 유의해야 한다. DataFrame이나 Series에서는 항상 loc 또는 iloc 속성을 이용해서 slicing한다고 생각하면 오류없이 slicing을 할 수 있을 것이다.

판다스 데이터에 행과 열의 이름이 있기 때문에 2개 이상의 자료를 연산할 때 각자의 행과 열의 이름을 그대로 유지하는 특성을 가지고 있다. 예를 들어 아래와 같은 행렬의 연산을 살펴보자.

```
D1=pd.DataFrame([[1,2],[3,4]],columns=['a','b'])
print(D1)
D2=pd.DataFrame([[11,22,33],[44,55,66],[77,88,99]],columns=['a','b','c'])
print(D2)
D1+D2
```

```
Out[1]:    a  b
        0  1  2
        1  3  4

Out[2]:    a   b   c
        0  11  22  33
        1  44  55  66
        2  77  88  99

Out[3]:     a     b    c
        0  12.0  24.0  NaN
        1  47.0  59.0  NaN
        2   NaN   NaN  NaN
```

D1에 행 index 2가 없고, 열 index c가 없으므로 행렬의 덧셈연산은 NaN(not a number의 약자이며 결측치를 의미한다)으로 표기된다. 이러한 연산은 두 개의 자료 크기가 다르더라도 에러메시지 없이 실행된다는 점에서 매우 유용한 기능이다. 물론 D1-D2, D1/D2, D1*D2, D2*D1, D2%D1도 D1+D2와 동일한 형태의 3X3행렬을 출력하므로 직접 실행하여 보길 바란다. 지금까지 살펴본 행렬의 연산은 element-wise하게 적용된 예이다. 실제 행렬연산을 하기 위해서는 np,matmul()를 사용하면 된다.

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

```
a=np.array([[1,2,3],[4,5,6]])
b=np.array([[1,2],[3,4],[5,6]])
c=np.matmul(a,b)
print(c)
d=c.T
print(d)
```

```
[[22 28]
 [49 64]]
[[22 49]
 [28 64]]
```

a는 2×3 행렬이고 b는 3×2 행렬이므로 행렬 c는 2×2 행렬이 되면 c.T는 행렬 c의 전치행렬(transpose)이다.

7.1 결측 자료의 처리

자료의 결측은 실제 자료에서 가장 빈번하게 일어나며 관측치 전체를 의미하기 보다는 관측치를 구성하는 변수 중 일부가 결측되는 경우를 의미한다. 결측 자료의 처리 방법 중, 가장 쉬운 방법은 결측 자료를 없애고 분석에 사용하는 것이다. 그러나 이는 관측치를 구성하는 변수가 많을 경우 자료의 손실이 매우 크다. 빅데이터의 경우 자료 중 결측치가 존재하는지를 파악하는 것도 쉽지 않을 뿐만 아니라 존재한다면 몇 변의 관측치에서 그리고 어떤 변수에서 결측이 발생하는지를 파악해야 하고 이 결측치를 처리하는 방법을 고민해야 할 것이다. 판다스는 이러한 작업을 비교적 손쉽게 해결할 수 있다.

isnull()은 결측여부를, notnull() 역시 isnull()의 반대 개념으로 결측 여부를, dropna()는 결측치가 있는 관측치를 버리라는 함수이고, fillna()는 결측치를 특정한 값으로 채우라는 함수이다. 판다스에서는 결측치를 np.nan 또는 None으로 표기한다. 간단한 예를 들어 결측치의 처리를 알아보자.

```
N=pd.Series([1,np.nan,'missing',None])
N.isnull()
```

```
Out[1]:  0    False
         1     True
         2    False
         3     True
          dtype: bool
```

이미 언급하였지만 판다스는 정수, 실수, 문자, None등 다양한 타입(type)의 자료를 수용하고 있음을 알 수 있고, N.isnull()은 논리형 자료를 출력해 주고, 1 번째와 3 번째 자료에서 결측이 발생하였음을 알 수 있다. 결측이 없는 자료만 추출해 내고 싶을 경우 아래와 같이 notnull()을 이용하면 된다. 또한 특정 숫자로 NaN을 대체하고 싶을 경우 fillna()를 이용하면 된다.

```
N[N.notnull()]
N.fillna(0)
```

```
Out[1]:      0            1
             2      missing
               dtype: object

Out[2]:      0            1
             1            0
             2      missing
             3            0
               dtype: object
```

추가로 아래와 같이 새로운 data를 정의하고 결측치에 대한 처리에 관해 좀 더 알아보자.

```
M=pd.DataFrame([[1,None,2,None],[4,5,6,None],[None,7,8,None]],index=list('123'),col
umns=list('abcd'))
print(M)
```

```
Out[1]:       a     b     c     d
         1  1.0   NaN     2  None
         2  4.0   5.0     6  None
         3  NaN   7.0     8  None
```

index=list('123')은 행의 이름을 1,2,3으로 차례대로 부여하고, columns =list('abcd')는 열의 이름을 차례대로 a,b,c,d로 부여하게 된다.

아래의 dropna를 살펴보면 M.dropna(axis='columns',how='all')의 경우 열을 기준으로 모두 NaN이면 열을 제거하라는 명령어이다. 또한 M.dropna(axis='rows',thresh=3)의 경우 3개 이상의 non-missing인 행만 남기라는 의미이다.

```
M.dropna(axis='columns',how='all')
M.dropna(axis='rows',thresh=3)
```

```
Out[1]:       a     b     c
         1  1.0   NaN     2
         2  4.0   5.0     6
         3  NaN   7.0     8

Out[2]:     a     b    c     d
        2  4.0   5.0   6  None
```

2차원 data에서 결측치를 채우는 방법에는 아래와 같은 두 가지를 살펴볼 수 있다. M.fillna(method='bfill', axis=1)의 경우는 각 행 별로(axis=1 이므로 across

columns의 의미이다) NaN의 바로 뒤 열값으로 채우라는 명령어가 된다. M.fillna(method='ffill',axis=1) 의 경우 'ffill'의 옵션이 각 행별로 NaN의 바로 전 열의 값으로 결측치를 채우라는 의미이다.

```
M.fillna(method='bfill',axis=1)
M.fillna(method='ffill',axis=1)
```

```
Out[1]:     a     b     c    d
       1  1.0  2.0  2.0  NaN
       2  4.0  5.0  6.0  NaN
       3  7.0  7.0  8.0  NaN

Out[2]:      a      b      c      d
         1  1.0    1.0    2.0    2.0
         2  4.0    5.0    6.0    6.0
         3  NaN    7.0    8.0    8.0
```

7.2 논리적 인덱스

판다스에서 논리적 인덱스는 결측치가 있는 관측치를 찾아내거나, 조건문을 부여하여 조건문을 만족하는 관측치나 변수를 찾아내고 이를 바탕으로 새로운 자료를 만드는 데 매우 유용하다. 이미 언급한 대로 DataFrame 클래스에서는 slicing을 위해 loc를 사용할 것을 권고한 바 있다.

다음의 자료를 고려하여 보자.

```
x=pd.DataFrame([[1,None,2,None],[3,4,5,None],[None,6,7,None]],index=list('123'),colu
mns=list('abcd'))
print(x)
```

```
Out[1]:      a      b      c      d
         1  1.0    NaN    2   None
         2  3.0    4.0    5   None
         3  NaN    6.0    7   None
```

또한 일부 원소만 값을 변경하기 위하여 아래와 같은 방법으로 loc를 이용하여 원하는 원소만 지정할 수 있다.

```
x.loc['2','d']=99
print(x)
```

```
x[pd.isnull(x['b'])]
```

```
Out[1]:      a      b      c      d
         1  1.0    NaN    2    None
         2  3.0    4.0    5     99
         3  NaN    6.0    7    None
Out[2]:      a      b      c      d
         1  1.0    NaN    2    None
```

Out[2]의 경우 x[조건문]은 조건문을 만족하는 행을 출력하는 명령어 이다. (x.loc[조건문]도 동일하게 작동한다). 그러므로 x['b']에 의해 열의 이름이 'b'인 열의 결측치가 있는 행으로 데이터를 구성하게 되므로 'b'의 1 행만 NaN이 있으므로 위의 결과와 같이 '1'행의 자료만 출력하게 된다. 아래에서 좀 더 복잡한 조건문을 사용하는 연습하여 보자.

```
x[pd.isnull(x).any(axis=1)]['b']
x.loc[pd.isnull(x).any(axis=1),'b']
x[pd.isnull(x).any(axis=1),'b']   #error만 발생
```

```
Out[1]: 1    NaN
        3    6.0
          Name: b, dtype: float64
```

pd.isnull(x).any(axis=1)은 axis=1에 의해 x의 각행에서 하나라도 결측치가 있는 행을 찾아내서 (만약 pd.isnull(x)만 사용하면 x 행렬 전체에서 결측치가 하나라도 있으면 x의 모든 행을 선택하게 됨.) 이중 ['b']에 의해 'b'열을 추출하라는 명령어이다. 이는 2차원 list자료 형태에서 x[i][j]는 x list행렬에서 i번째 행과 j번째 열 자료를 지칭하는 성질을 그대로 가지고 온 것이다. 물론 x.loc[pd.isnull(x).any(axis=1),'b']도 위와 동일한 결과를 산출한다. 그러나 주의할 점은 x.loc 대신 x만 쓴 x[pd.isnull(x).any(axis=1),'b']은 실행되지 않으므로 주의해야 한다.

또 다른 논리 조건문을 고려하기 위하여 다음을 살펴보자. 다음과 같이 DataFrame를 생성한 후 조건문을 사용하여 보면

```
K=pd.DataFrame(np.arange(12).reshape(3, 4),columns=list('abcd'))   #data 생성
print(K)
K.loc[K['b']>4]['c']
K.loc[K['b']>4,'c']
K.loc[K['b']>4,['b','d']]
```

```
Out[1]:    a  b   c   d
        0  0  1   2   3
        1  4  5   6   7
        2  8  9  10  11
```

```
Out[2]:    1    6
        2   10
        Name: c, dtype: int32
```

```
Out[3]    b    d
        1  5    7
        2  9    11
```

Out[2]의 경우 K의 'b'열이 4보다 큰 행(두 번째와 세 번째행)을 대상으로 'c'열의 값을 의미하므로 위와 같은 결과를 나타낸다. K.loc[K['b']>4,'c']도 동일한 결과를 산출하는 것을 알 수 있다. Out[3]의 경우 'b'열이 4보다 큰 행을 대상으로 열'b'와 'd'를 의미하므로 위와 같은 결과를 보여주고 있다.

그러나 list 데이터 형식의 index는 오직 하나의 열만 지정해야 하므로 ['b','d']와 같은 형식의 열 지정은 적용될 수 없다. 따라서 K.loc[K['b']>4]['b','d']와 같은 표현은 작동하지 않는다.

```
P=pd.DataFrame(np.arange(6).reshape(2,  3),index=list('ab'),columns=list('123'))
#data의 설정
print(P)
```

```
P.stack()
```

```
Out[1]:    1  2  3
        a  0  1  2
        b  3  4  5
```

```
Out[2]:  a  1   0
            2   1
            3   2
         b  1   3
            2   4
            3   5
                dtype: int32
```

P라는 DataFrame을 설정한 후 P.stack은 행 index와 열 index를 합친 후 다중 index를 만들어 자료를 재정리하는 기능을 가지고 있다. 또한 unstack을 이용하여 위의 과정을 거꾸로 실행할 수 있다. 즉, P.stack().unstack()은 원래의 P로 환원한

다.

7.3 데이터 붙이기

2개 이상의 데이터가 있을 때 데이터를 합치는 방법에는 데이터를 밑으로 붙이는 방법과 옆으로 붙이는 방법이 있다. 일반적인 분석 데이터의 경우 열에 변수가, 행에 case 또는 sample이 배열되는 경우가 많다. 이에 데이터를 밑으로 붙인다는 것은 데이터의 크기를 증가시킨다(case를 증가시킨다)는 의미이고, 옆으로 붙인다는 의미는 분석에 사용되는 변수가 늘어난다는 것을 의미한다.

먼저 데이터의 크기를 증가시키는 밑으로 붙이기 과정을 살펴보자.

```
Q1=pd.DataFrame([[1,2,3],[4,5,6]],columns=list('abc'))
Q2=pd.DataFrame([[11,22,33],[44,55,66]],columns=list('bcd'))
print(Q1)
print(Q2)
pd.concat([Q1,Q2])
```

```
Out[1]: (Q1)
   a  b  c
0  1  2  3
1  4  5  6
```

```
Out[2]:(Q2)
    b   c   d
0  11  22  33
1  44  55  66
```

```
     a   b   c     d
0  1.0   2   3   NaN
1  4.0   5   6   NaN
0  NaN  11  22  33.0
1  NaN  44  55  66.0
```

밑으로 붙이기는 concat([,]) 함수를 사용한다. 위 예제에서 볼 수 있듯이 Q1은 열 'd'를 가지고 있지 않으므로 결측 처리되고 Q2는 'a'가 없으므로 결측 처리되고 있음을 알 수 있다. index가 '0 1','0 1'로 Q1과 Q2의 index를 그대로 옮겨 놓고 있다. 이 index가 반복되지 않고 0 1 2 3과 같이 순서 있는 index를 원하면, ignore_index=True를 추가하여 사용하면 된다.

```
pd.concat([Q1,Q2], ignore_index=True)
```

```
Out[1]:
     a   b   c     d
0  1.0   2   3   NaN
1  4.0   5   6   NaN
2  NaN  11  22  33.0
3  NaN  44  55  66.0
```

또한 Out[2]와 같이 Q1과 Q2가 공통으로 가지고 있는 열 'b'와 'c'로만 구성된 자료의 아래로 붙이기를 원하면 아래와 같이 join='inner'를 사용하면 된다.

```
pd.concat([Q1,Q2], join='inner')
```

```
Out[1]:    b   c
       0   2   3
       1   5   6
       0  11  22
       1  44  55
```

이제는 데이터의 분석 변수를 추가할 때 많이 사용하게 되는 옆으로 붙이기는 고려하여 보자.

```
R1=pd.DataFrame([[1,2,3],[4,5,6]],columns=list('abc'))
R2=pd.DataFrame([[11,22],[33,44]],columns=list('de'))
print(R1)
print(R2)
R3=pd.concat([R1,R2],axis=1)
print(R3)
```

```
Out[1]: (R1)   a  b  c
           0   1  2  3
           1   4  5  6
```

```
Out[2]:(R2)    d   e
           0  11  22
           1  33  44
```

```
Out[3]:(R3)
          a  b  c   d   e
       0  1  2  3  11  22
       1  4  5  6  33  44
```

즉, concat([,])에 axis=1을 추가하면 옆으로 붙이기가 된다. 그러나 2개의 데이터가 공통의 정보를 가진 열 이름이 있을 경우, 예를 들어 두 개의 데이터에 ID라는 공통된 변수가 있을 경우 이를 기준으로 옆으로 데이터를 붙이기를 해야 할 필요가 있을 것이다.

```
S1=pd.DataFrame([[1,'aa',3],[4,'bb',5]],columns=list('abc'))
S2=pd.DataFrame([[11,'bb'],[22,'aa']],columns=list('db'))
print(S1)
print(S2)
S3=pd.merge(S1,S2,on='b')
print(S3)
```

```
Out[1]: (S1)   a   b  c
           0   1  aa  3
           1   4  bb  5
```

```
Out[2]:(S2)    d   b
           0  11  bb
           1  22  aa
```

```
Out[3]:(S3)
          a   b  c   d
       0  1  aa  3  22
       1  4  bb  5  11
```

위의 예제는 두 데이터 열'b'에 공통의 ID "aa","bb"가 있다. 이를 기준으로 옆으로 붙이기는 merge( )에 on='b'을 추가하면 데이터 S3가 생성되게 된다.

그런데 다음과 같은 데이터를 살펴보자. S11에는 'aa'가 두 번, S21에는 'bb'가 두 번 나타나는 데이터이다.

```
S11=pd.DataFrame([[1,'aa',3],[4,'bb',5],[6,'aa',7]],columns=list('abc'))
S21=pd.DataFrame([[11,'bb'],[22,'aa'],[33,'bb']],columns=list('db'))
print(S11)
print(S21)
S31=pd.merge(S11,S21,on='b')
print(S31)
```

Out[1]: (S11)

	a	b	c
0	1	aa	3
1	4	bb	5
2	6	aa	7

Out[2]:(S21)

	d	b
0	11	bb
1	22	aa
2	33	bb

Out[3]:(S31)

	a	b	c	d
0	1	aa	3	22
1	6	aa	7	22
2	4	bb	5	11
3	4	bb	5	33

S11에 'aa'가 두 번 나타나므로 이에 대응되는 S21에 있는 'aa'에 대응하는 열 d의 값 22가 S31에 두 번 반복되고, 한편 S21에 'bb'가 두 번 있으므로 이에 대응되는 S11에 있는 'bb'의 열 a와 c의 값이 S31에 두 번 반복되어 나타나고 있다. 이러한 정렬은 자료 정리에 매우 유용한 기능이다. 예를 들어 S11이 진료 기록이면 동일인이 두 번 이상 진료를 받은 데이터를 표현한 것이고, S21은 몇 년간의 건강검진기록이라면 이 역시 동일인이 두 번 이상 나타날 것이므로, 두 데이터를 통합할 때 본래 가지고 있는 정보의 손실이 없어야하기 때문에 이러한 자료통합은 매우 유용하다.

```
W12=pd.DataFrame([['aa',3],['bb',5],['aa',7]],columns=list('ab'))
W22=pd.DataFrame([[11,'bb'],[22,'aa']],columns=list('ba'))
print(W12)
print(W22)
W32=pd.merge(S12,S22,on='a')
print(W32)
```

Out[1]: (W12)

	a	b
0	aa	3
1	bb	5
2	aa	7

Out[2]:(W22)

	b	a
0	11	bb
1	22	aa

Out[3]:(W32)

	a	b_x	b_y
0	aa	3	22
1	aa	7	22
2	bb	5	11

위 예제에서는 W12와 W22가 모두 같은 열 이름(또는 변수명)을 가지고 있고 a열의 ID로 옆붙이기를 한 결과이다. 예를 들어, 동일인이 반복해서 건강검진을 받을 때 a

열은 ID를 b열은 검사항목을 나타내는 패널 데이터의 경우에 발생하는 자료 유형이다. merge( )함수는 데이터셋 W32와 같이 자동으로 b_x,b_y를 분리해서 표현해 주고 있다.

다음은 두 개의 데이터 셋이 공통의 ID가 있으나 ID를 나타내는 열 이름(또는 변수명)이 다를 때 옆 붙이기 예를 보여주고 있다.

```
U1=pd.DataFrame([[1,'aa',3],[4,'bb',5]],columns=list('abc'))
U2=pd.DataFrame([[11,'bb'],[22,'aa']],columns=list('de'))
print(U1)
print(U2)
U3=pd.merge(U1,U2,left_on='b',right_on='e')
print(U3)
```

```
Out[1]: (U1)   a   b   c
           0   1   aa  3
           1   4   bb  5
```

```
Out[2]:(U2)      d   e
              0  11  bb
              1  22  aa
```

```
Out[3]:(U3)
   a   b   c   d   e
0  1   aa  3   22  aa
1  4   bb  5   11  bb
```

U1의 b열과 U2의 e열이 ID를 가지고 있지만 열 이름이 다르므로(즉, 변수명이 다르므로)왼쪽에 있는(U1) 데이터의 b(left_on='b')'와 오른쪽에 있는 데이터(U2)의 e(right_on='e')를 지정해 주어 U1과 U2를 옆붙이기 할 수 있다. 그런데 U3에는 동일한 정보를 가진 b와 e열이 있다. 이중 하나를 제거해 주어야 할 필요가 있다.

```
U4=pd.merge(U1,U2,left_on='b',right_on='e').drop('e', axis=1)
print(U4)
```

```
Out[1]: (U4)
   a   b   c   d
0  1   aa  3   22
1  4   bb  5   11
```

위와 같이 drop('e',axis=1)을 추가하여 불필요한 'e'열을 제거하고 있다. 이때 열이라는 의미의 axis=1은 반드시 포함해야 한다.

```
V1=pd.DataFrame([[1,'aa',3],[4,'bb',5]],columns=list('abc'))
V2=pd.DataFrame([[11,'cc'],[22,'aa']],columns=list('db'))
print(V1)
print(V2)
V3=pd.merge(V1,V2,how='inner')
print(V3)
```

```
Out[1]: (V1)   a   b  c
          0  1  aa  3
          1  4  bb  5
```

```
Out[2]:(V2)     d   b
          0  11  cc
          1  22  aa
```

```
Out[3]:(V3)
     a   b  c   d
0  1  aa  3  22
```

V1과 V2가 공통적으로 'b'열을 가지고 있으므로 자동으로 'b'열을 기준으로 옆붙이기를 한다. 그러나 how='inner'를 지정하여 'b'열에 공통으로 가지고 있는 ID 'aa'가 있는 행만으로 구성된다.

```
V4=pd.merge(V1,V2,how='outer')
print(V4)
V5=pd.merge(V1,V2,how='left')
print(V5)
```

```
Out[1]: (V4)      a    b    c     d
            0  1.0   aa  3.0  22.0
            1  4.0   bb  5.0   NaN
            2  NaN   cc  NaN  11.0
```

```
Out[2]:(V5)     a   b  c     d
            0  1  aa  3  22.0
            1  4  bb  5   NaN
```

위의 결과를 보면 how='outer'는 V1과 V2의 공통 열 'b'를 기준으로 합집합에 해당 된다. 그러므로 how='inner'는 V1과 V2의 교집합으로 해석할 수 있다. how='left'은 V1을 기준으로 'b'열에 있는 ID가 있는 V2의 행만 옆 붙이기를 한다.

```
rng=np.random.RandomState(1)
dset=pd.DataFrame({'ID':['A','B','C','A','B','C'],
                   'x1':range(6),
                   'x2':rng.randint(0,10,6)})
print(dset)
```

Out[1]: (dset)

	ID	x1	x2
0	A	0	5
1	B	1	8
2	C	2	9
3	A	3	5
4	B	4	0
5	C	5	0

DataFrame에서 dictionary 데이터의 key에 list데이터가 연결되면 key가 열 이름이 된다. dset의 열이름이 ID, x1, x2가 된 이유이다. 그러나 key에 하나의 값만 연결되면 key가 행 이름이 된다는 것을 주의하자. ID행은 A,B,C가 모두 2개씩 있으므로 ID별로 x1 과 x2의 기초 통계량, 예를 들어 평균,중앙값,표준편차,합계,최소값,최대값등을 구하고자 할 때 groupby( ) 함수를 사용하면 쉽게 구할 수 있다.

```
dset.groupby('ID').mean()
dset.groupby('ID').std()
```

Out[1]: (mean)

ID	x1	x2
A	1.5	5.0
B	2.5	4.0
C	3.5	4.5

Out[2]:(std)

ID	x1	x2
A	2.12132	0.000000
B	2.12132	5.656854
C	2.12132	6.363961

또한 여러 가지 통계량을 한꺼번에 구하고 싶은 경우 아래와 같이 사용 가능하다.

```
dset.groupby('ID').aggregate([min,max,np.mean,np.median,np.std])
```

Out[1]:

	x1					x2				
	min	max	mean	median	std	min	max	mean	median	std
ID										
A	0	3	1.5	1.5	2.12132	5	5	5.0	5.0	0.000000
B	1	4	2.5	2.5	2.12132	0	8	4.0	4.0	5.656854
C	2	5	3.5	3.5	2.12132	0	9	4.5	4.5	6.363961

위와 같이 원하는 통계량을 지정하여 구할 수도 있으나 pandas에서는 아래와 같이 기본적이 기초 통계량을 제공하는 describe()함수를 제공하고 있다.

```
dset.groupby('ID')['x1'].describe()
```

Out[1]:

ID	count	mean	std	min	25%	50%	75%	max
A	2.0	1.5	2.12132	0.0	0.75	1.5	2.25	3.0
B	2.0	2.5	2.12132	1.0	1.75	2.5	3.25	4.0
C	2.0	3.5	2.12132	2.0	2.75	3.5	4.25	5.0

위 프로그램은 'ID' 그룹별로 'x1'에 대한 기초 통계량을 산출해 주고 있다.

8. 그림그리기

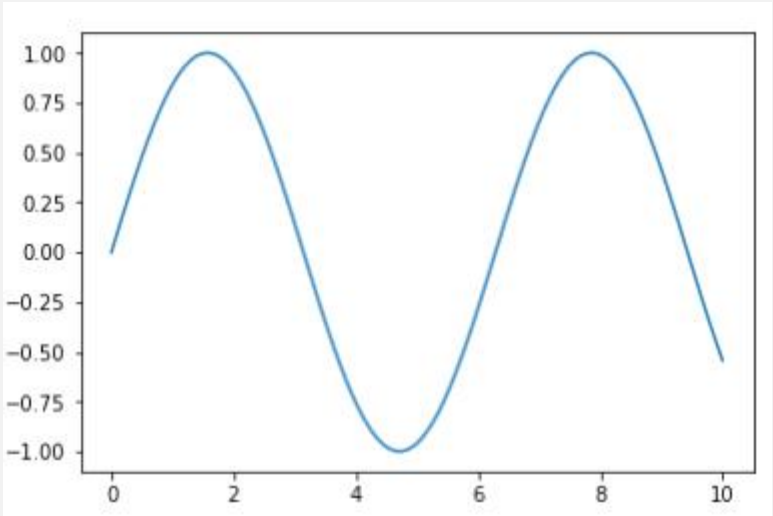
그림그리기는 자료 분석에서 매우 중요하다. 변수에 대한 Plot, Histogram, box-plot 등을 통해 변수 선택, 변수간의 관계, 변수의 분포 등 자료의 특성을 자료 분석이전에 파악해야하기 때문이다.

Python에서 그림그리기의 기본은 matplotlib라는 라이브러리에 의해 실행된다. 특히 matplotlib내의 pyplot을 주로 사용하여 자료의 특성을 파악할 수 있다.

가장 간단한 예로부터 출발하여 보자.

```
import matplotlib.pyplot as plt
import numpy as np
x=np.linspace(0,10,100)
plt.plot(x, np.sin(x))
plt.show()
```

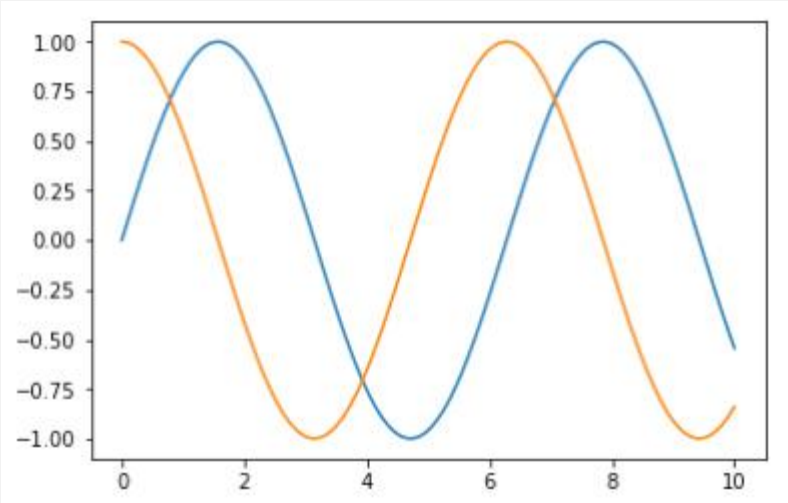
Out[1]:



Python의 그림그리기는 matplotlib.pyplot을 호출하여 시작할 수 있고 그림의 끝은 plt.show()로 끝나게 된다.(Jupyter notebook을 사용할 때 그림의 결과를 보기 위해 matplotlib.pyplot 이전에 %matplotlib inline을 입력하라고 일부 책에서는 소개하고 있으나 최근 버전에서는 필요가 없다. plt.show()는 없어도 되지만 이를 입력하지 않으면 불필요한 그림 정보를 출력하므로 습관적으로 입력해 주는 것이 좋다.)

```
plt.plot(x,np.sin(x))
plt.plot(x,np.cos(x))
plt.show()
```

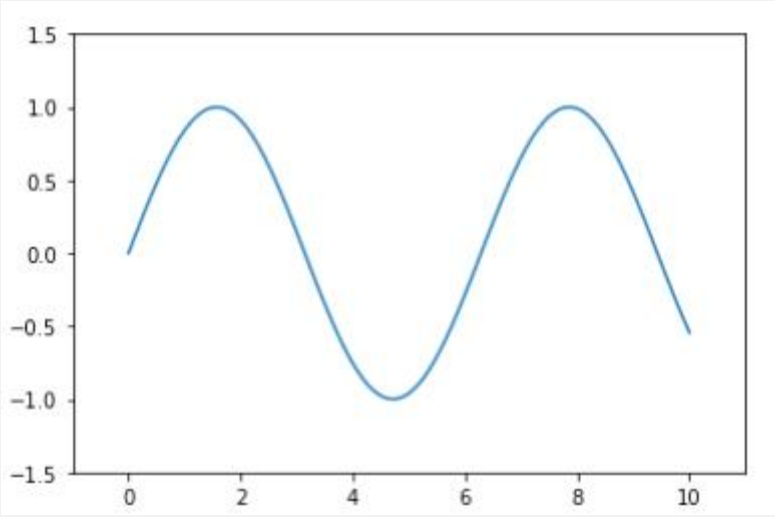
Out[1]:



위와 같이 하나의 틀(이를 axis라고 한다.)속에 두 개의 그림을 넣을 수도 있다. x축과 y축의 범위도 아래와 같이 조정할 수 있다.

```
plt.plot(x,np.sin(x))
plt.xlim(-1,11)
plt.ylim(-1.5,1.5)
plt.show()
```

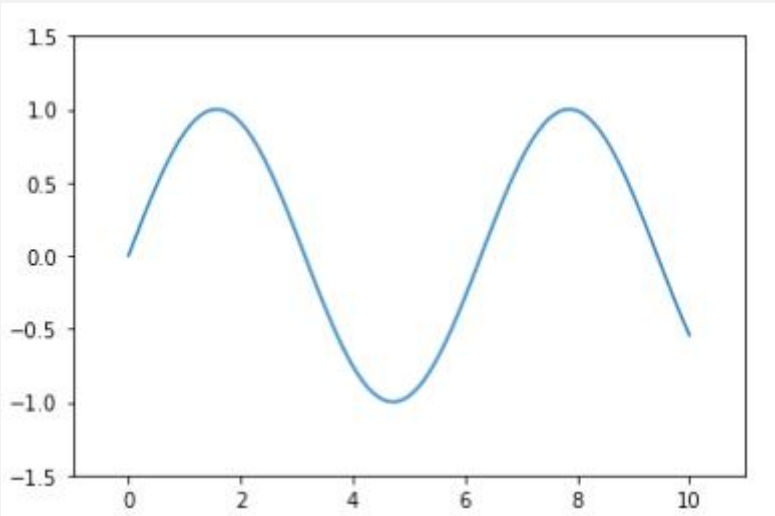
Out[1]:



위에서 사용한 xlim과 ylim 대신에 axis[xmin,xmax,ymin,ymax] 함수를 이용할 수 있다.

```
plt.plot(x,np.sin(x))
plt.axis([-1,11,-1.5,1.5])
plt.show()
```

Out[1]:

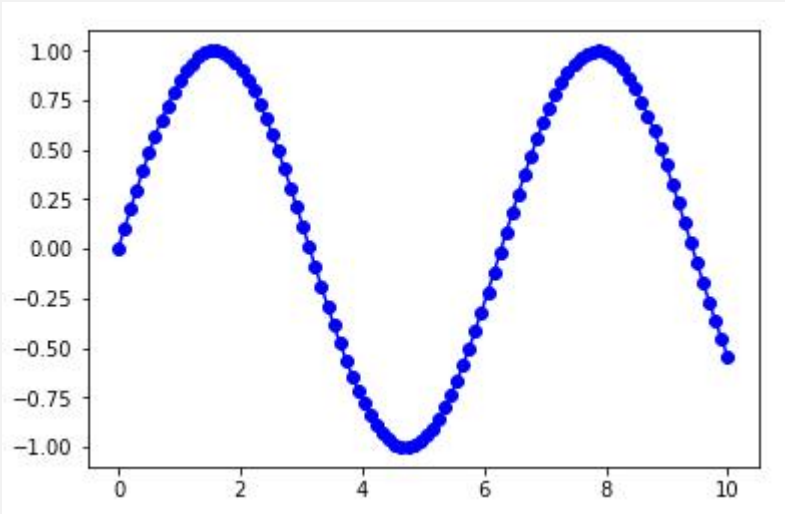


플롯의 색, 선의 종류, 자료점 표시방법 등을 부여할 수도 있다. 플롯의 색은 rgb(red/green/blue)와 cmyk(cyan/megenta/yellow/black)을 쓸 수 있고, 선의 종류는 -(solid), --(dash), -·(dash dot), :(dot)으로 구분할 수 있고 자료점 표시 방법은 아래 표와 같다. [출처: https://matplotlib.org/index.html]

marker	symbol	description
"."		point
","		pixel
"o"		circle
"v"		triangle_down
"^"		triangle_up
"<"		triangle_left
">"		triangle_right
"1"		tri_down
"2"		tri_up
"3"		tri_left
"4"		tri_right
"8"		octagon
"s"		square
"p"		pentagon
"P"		plus (filled)
"*"		star
"h"		hexagon1
"H"		hexagon2
"+"		plus
"x"		x
"X"		x (filled)
"D"		diamond
"d"		thin_diamond

```
plt.plot(x,np.sin(x),'b-o')
plt.show()
```

Out[1]:

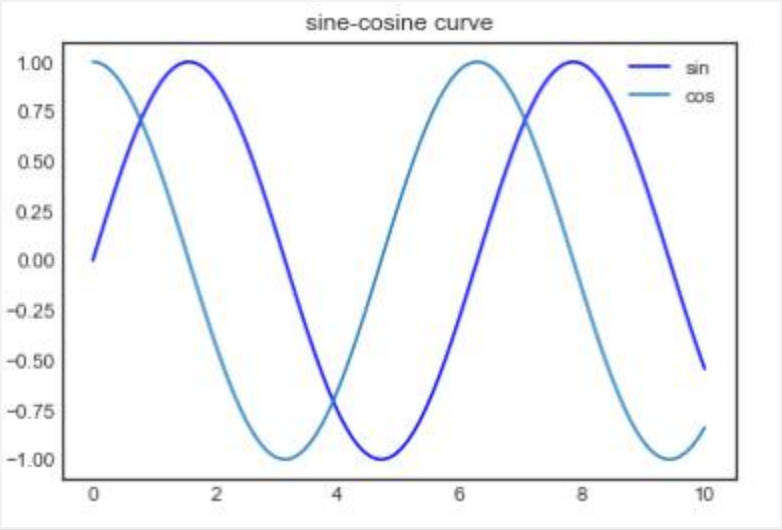


plt.plot(x,np.sin(x),'b-o')의 'b-o'에 의해 sine그래프의 형태가 달라졌음을 알 수 있다.'b-o' 내의 차례는 어떤 차례가 와도 인식하고 3개중 어느 것을 생략해도 인식하므로 필요에 따라 선택하여 사용할 수 있다.

Plot의 제목과 범례를 아래와 같이 설정할 수 있다.

```
plt.style.use('seaborn-white')
plt.plot(x,np.sin(x),'b', label='sin')
plt.plot(x,np.cos(x), label='cos')
plt.title('sine-cosine curve')
plt.legend()
plt.show()
```

Out[1]:

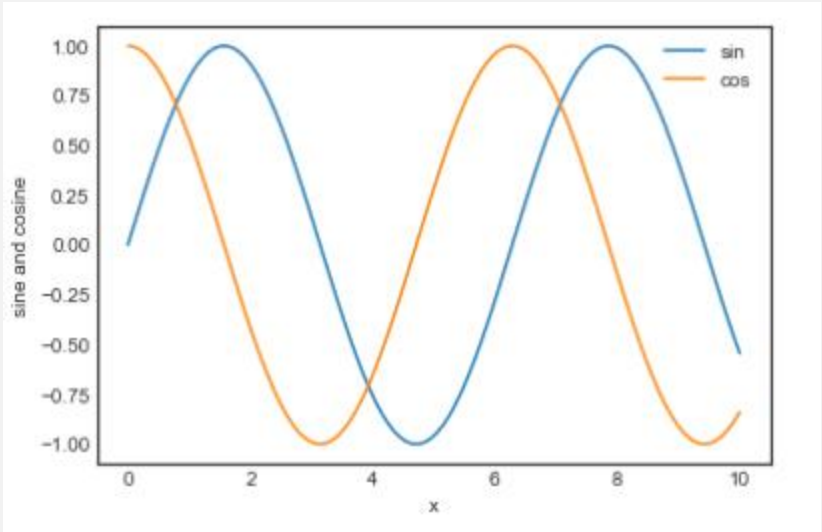


** Matplotlib에 설정되어 있는 style에 관련해서는 다음의 링크를 참고하길 바란다. https://tonysyu.github.io/raw_content/matplotlib-style-gallery/gallery.html

plt.plot내에 label=' '로 이름을 부여하고 plt.legend( ) 함수를 불러오면 범례를 표시할 수 있으며 plt.title('title name')을 불러내어 그림의 제목을 붙일 수 있다. plt.legend()에 의한 범례는 loc='upper right','upper left','lower right',''lower left','lower center'등을 사용하여 위치를 지정할 수 있으며, 'best'로 조정할 경우 가장 적절한 위치를 찾아서 범례를 보일 수 있다.

```
plt.plot(x,np.sin(x),label='sin')
plt.plot(x,np.cos(x),label='cos')
plt.legend(loc='upper right')
plt.xlabel('x')
plt.ylabel('sine and cosine')
plt.show()
```

Out[1]:

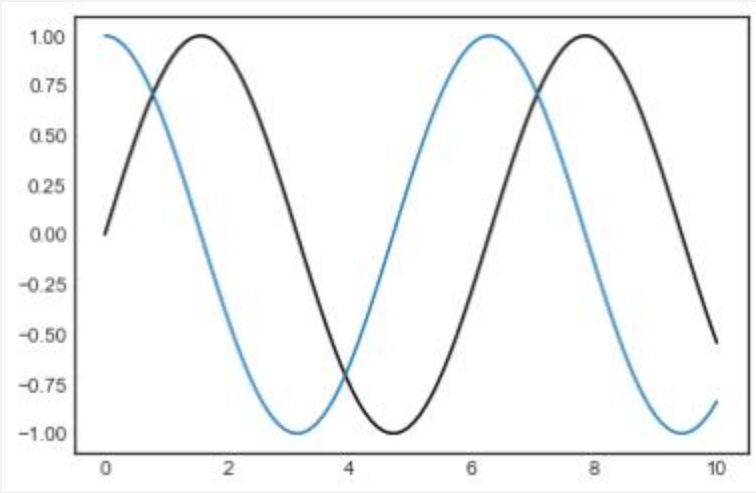


plt.xlabel('x')는 x축에 이름을 부여하고 plt.ylabel('sine and cosine')은 y축에 이름을 부여하고 있다.

matplotlib.pyplot에서 그림을 그리는 또 다른 방법은 아래와 같이 틀을 객체화하여 앞에서 논의한 다양한 그림을 그려볼 수 있다.

```
import matplotlib.pyplot as plt
import numpy as np
ax=plt.axes()
ax.plot(x,np.sin(x),'k-')
ax.plot(x,np.cos(x))
plt.show()
```

Out[1]:

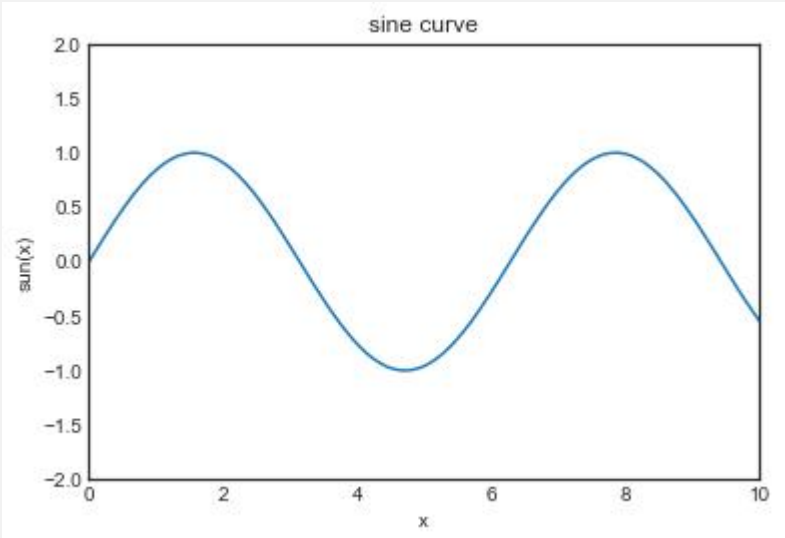


위와 같이 틀(axes)를 ax로 객체화하였고 plt.plot을 ax.plot으로(같은 방법으로

plt.legend()를 ax.legend()로) 바꿔서 사용한 것을 볼 수 있다. 그러나 주의할 점은 plt.xlabel()은 ax.set_xlabel()으로, plt.ylabel()은 ax.set_ylabel()으로, plt.xlim()은 ax.set_xlim으로, plt.ylim()은 ax.set_ylim()으로, 그리고 plt.title()는 ax.set_title()로 고쳐야 한다. 이러한 과정은 ax.set() 함수 하나로 다음과 같이 해결할 수 있다.

```
ax=plt.axes()
ax.plot(x,np.sin(x))
ax.set(xlim=(0,10),ylim=(-2,2),xlabel='x',ylabel='sun(x)',title='sine curve')
plt.show()
```

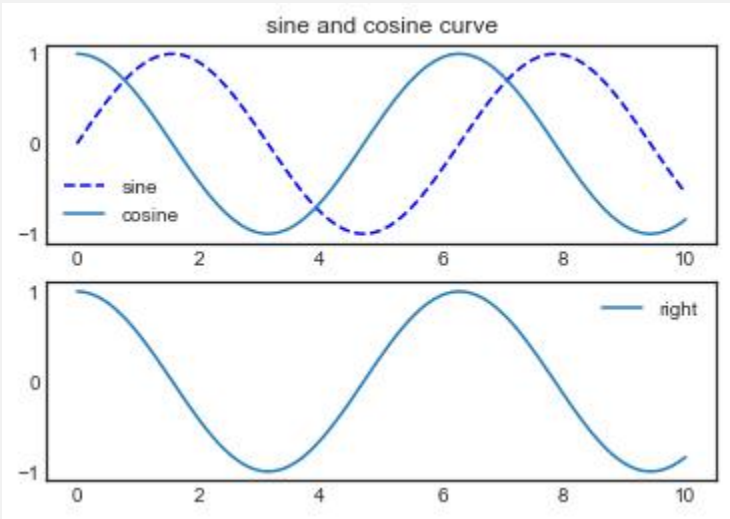
Out[1]:



'figure'는 2개 이상의 틀(axes)로 구성되어 있다. 이는 2개 이상의 플롯을 하나의 그림으로 묶어서 출력하고자 할 때 사용된다. 이를 위해서 먼저 figure() 함수를 호출하고 subplot 함수에 의해 틀을 구성한다. subplot(n,m,i)은 행이 n개이고 열이 m개인 플롯으로 구성되어 있다는 의미이고 i는 i번째 plot을 의미한다.

```
import matplotlib.pyplot as plt
import numpy as np
x=np.linspace(0,10,100)
plt.figure()
plt.subplot(2,1,1)
plt.plot(x,np.sin(x),'b--',label='sine')
plt.plot(x,np.cos(x),label='cosine')
plt.title('sine and cosine curve')
plt.legend(loc='best')
plt.subplot(2,1,2)
plt.plot(x,np.cos(x),label='right')
plt.legend(loc='best')
plt.show()
```

Out[1]:

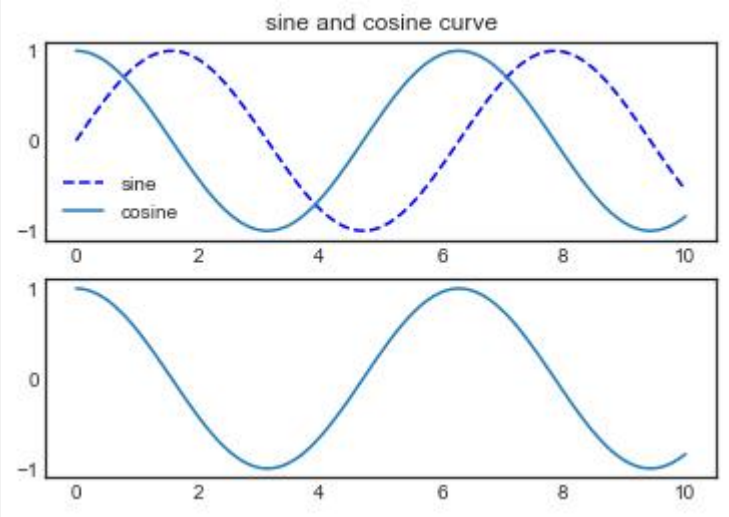


subplot(2,1,1)이므로 2행 1열로 구성된 그림으로 그중 첫 번째 그림을 의미하며 plot.subplot(2,1,1) 아래에서 plt.subplot(2,1,2)이전까지가 subplot(2,1,1)들에 있는 그림의 옵션들이다. plt.figure()대신 plt.figure(figsize=18.5)로 그림의 크기를 조절할 수 있다.

아래는 틀 객체화를 통해서 동일한 결과를 그리는 프로그램이다.

```
fig, ax = plt.subplots(2,1)
ax[0].plot(x,np.sin(x),'b--',label='sine')
ax[0].plot(x,np.cos(x),label='cosine')
ax[0].set_title('sine and cosine curve')
ax[0].legend(loc='best')
ax[1].plot(x,np.cos(x))
plt.show()
```

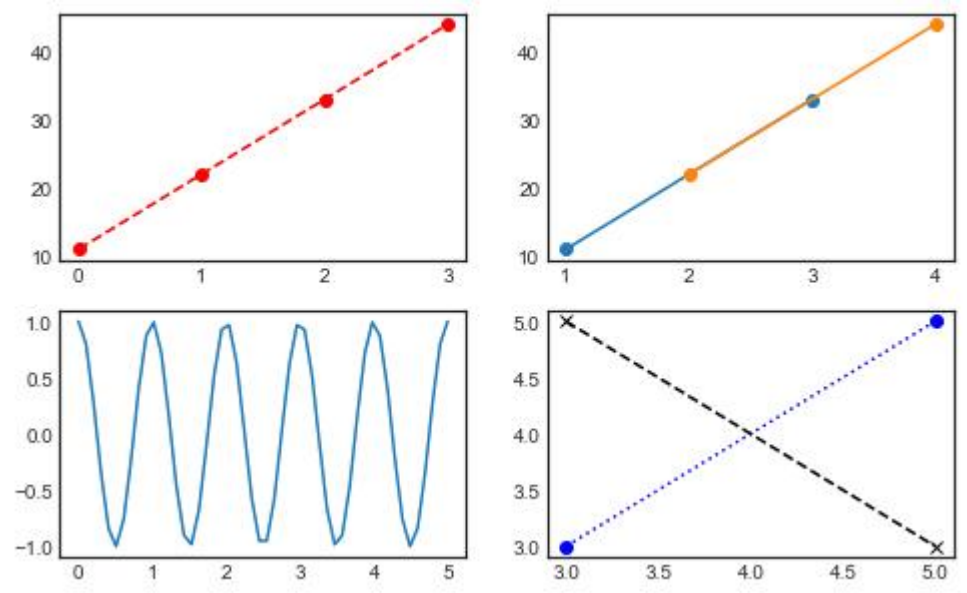
Out[1]:



틀 표현은 ax[0], ax[1]에 의해 첫 번째 틀과 두 번째 틀을 표현하고 있다. 하나의 figure에 2×2의 틀이 있을 때 틀 객체화를 통한 4개의 플롯은 다음과 같다.

```
fig, ax = plt.subplots(2,2,figsize=(8,5))
ax[0][0].plot([11,22,33,44],'ro--')
ax[0][1].plot([[1,2],[3,4]],[[11,22],[33,44]],'o-')
ax[1][0].plot(np.linspace(0,5),np.cos(2*np.pi*np.linspace(0,5)))
ax[1][1].plot([3,5],[3,5],'bo:')
ax[1][1].plot([3,5],[5,3],'kx--')
plt.show()
```

Out[1]:



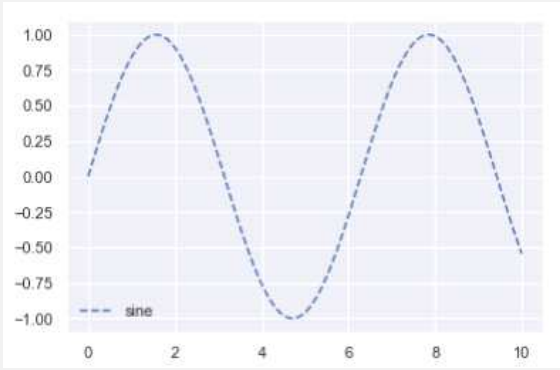
행과 열을 나타내는 ax[i][j]로 plot의 위치를 지정하고 있고 ax[0][0]에서 plot([11,22,33,44])와 같이 오직 하나의 list 데이터만 있을 때 이 값은 y축의 값을 의미하고 대응하는 x축의 값은 [0,1,2,3]을 자동으로 부여한다. ax[0][1]은 2차원 자료가 입력값으로 구성되어 있다. x축의 첫 번째 행과 y축의 첫 번째 행에 대응되게 자료 쌍을 구성하고 x축의 두 번째 행과 y축의 두 번째 행이 자료 쌍을 이루어 (1,11), (2,22), (3,33), (4,44)의 4개의 자료 점을 구성한다.

Matplotlib는 이외에도 Histogram, box-plot, 등고선 지도, 3차원 그림 등 다양한 그림을 그릴 수 있는 좋은 도구이다. 좀더 자세한 내용은 다음의 링크 : https://matplotlib.org/index.html를 참고하길 바란다.

통계적 자료탐색에 특화되고 matplotlib보다 고급 명령어가 seaborn plot이다. 물론 지금까지 논의한 그림그리기를 seaborn style로도 그릴 수 있다.

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
x=np.linspace(0,10,100)
plt.plot(x,np.sin(x),'b--', label='sine')
plt.legend(loc='best')
plt.show()
```

Out[1]:

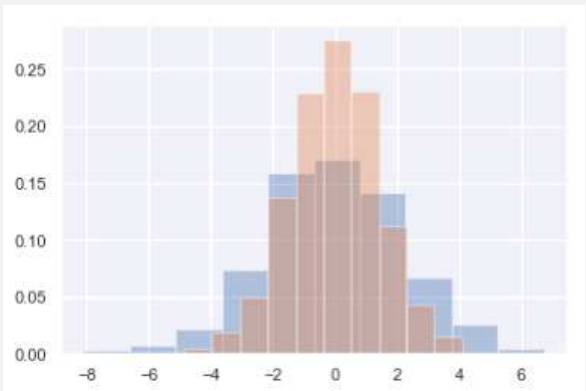


sns.set() 함수를 입력해서 seaborn style의 플롯을 그릴 수 있게 한다.

이변량 정규분포를 따르는 두 개의 변수 x, y가 평균이 모두 0이고, x의 분산이 5, y의 분산2, 그리고 x와 y의 공분산이 1일 때, 1000개의 자료를 추출하여 seaborn을 사용하여 보자.

```
import pandas as pd
bn=np.random.multivariate_normal([0,0],[[5,1],[1,2]],size=1000)
bn=pd.DataFrame(bn, columns=['x','y'])
for i in 'xy':
    plt.hist(bn[i], density=True, alpha=0.4)
```

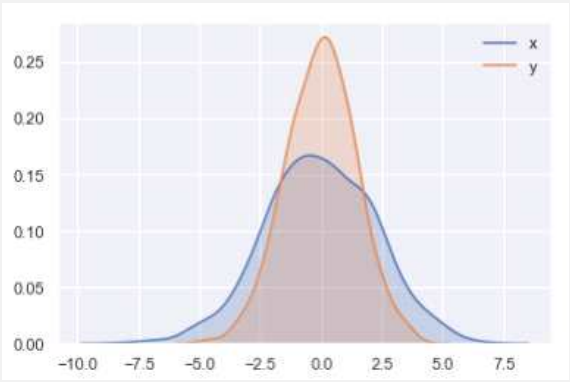
Out[1]:



위의 결과는 matplotlib에 의한 히스토그램이다. density=True는 y축의 값이 백분율을 나타내고 alpha는 그림의 투명도를 나타낸다. 1은 불투명, 0은 완전투명으로 아무 그림이 나타나지 않는다.

```
for i in 'xy':
    sns.kdeplot(bn[i],shade=True)
```

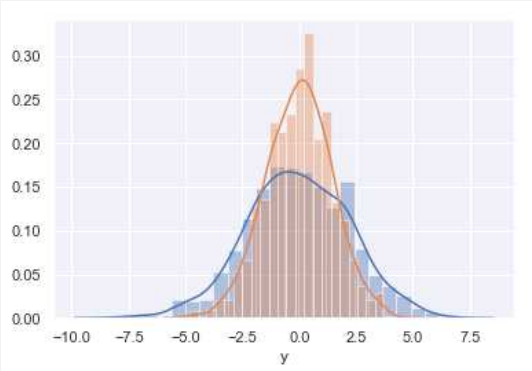
Out[1]:



kdeplot은 히스토그램을 이용한 분포 함수의 추정치로 kernel density estimation (kde)을 이용한 것이다. shade=True는 분포함수 내부에 그림자 표시를 하라는 명령어 이다. sns는 히스트 그램과 kde 분포함수 추정치를 다음과 같이 displot()함수로 구현할 수 있다.

```
sns.distplot(bn['x'])
sns.distplot(bn['y'])
```

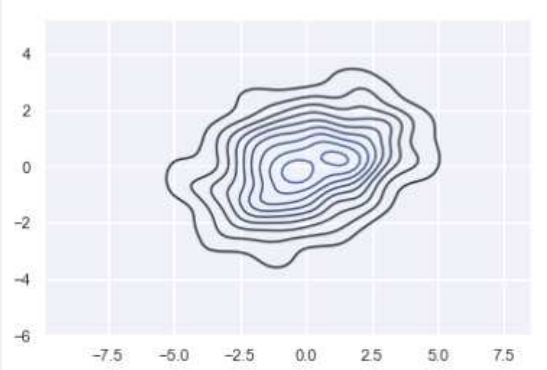
Out[1]:



2차원의 kernel density 추정치도 다음과 같이 간단하게 구할 수 있다.

```
sns.kdeplot(bn)
```

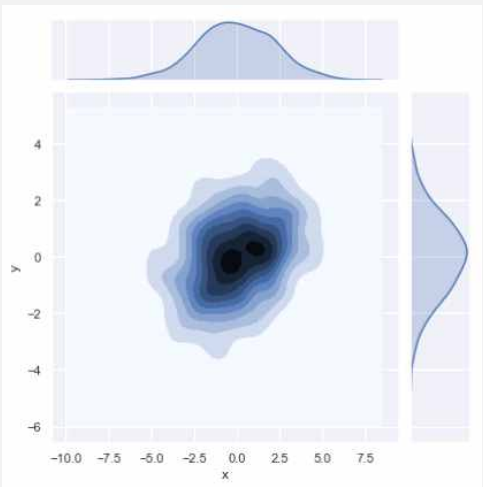
Out[1]:



2차원 Kernel density 추정치와 더불어 주변 분포도 sns.jointplot 함수를 이용하여 그릴 수 있다.

```
sns.jointplot(bn)
```

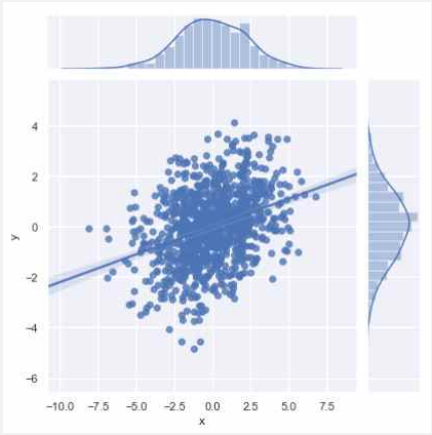
Out[1]:



sns.jointplot(x='x',y='y',data=bn, kind='kde')도 위와 동일한 결과를 출력한다. 다음은 kind='reg'로 지정하여 회귀선과 주변 분포를 동시에 그려주는 프로그램이다.

```
sns.jointplot(x='x',y='y',data=bn,kind='reg')
```

Out[1]:



다음은 seaborn이 제공하는 4종류의 붓꽃 iris 데이터이다.

```
iris=sns.load_dataset('iris')
iris.head()
```

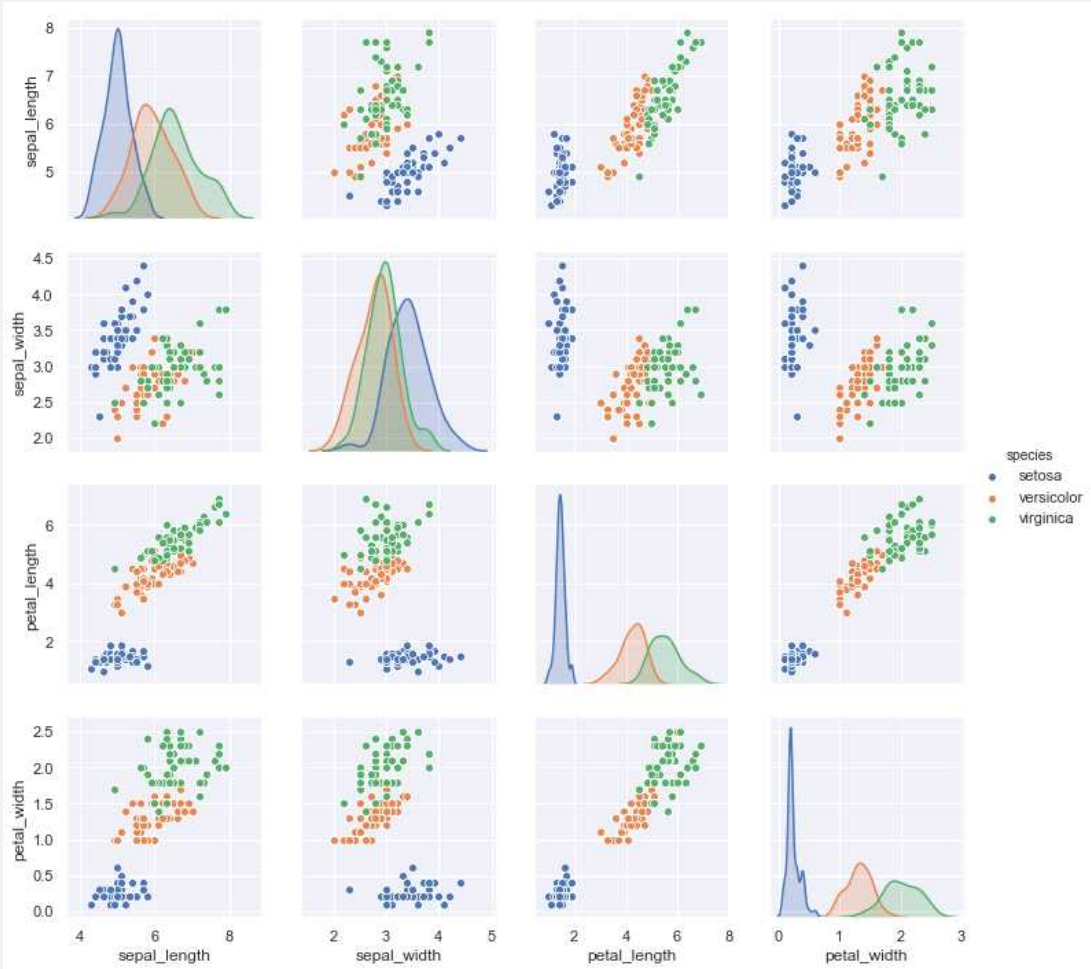
Out[1]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

위의 결과와 같이 species열만 문자 데이터이고 다른 변수들은 실수 값이다.

```
sns.pairplot(iris, hue='species')
```

Out[1]:



실수값에 대한 각 pair에 대해 산포도(scatter plot)을 그려주고 대각 행렬은 히스토그램을 출력한다. hue=문자값 데이터를 가진 변수를 (이 데이터의 경우, species)지정하여 주면 히스토그램과 산포도를 색깔로 3가지 species를 구별하여 준다. 위 그림에 의해 어떤 변수 또는 변수 조합이 세 종류의 붓꽃을 잘 구별하여 주는지를 알 수 있다.

Box-plot은 관심있는 그룹들의 분포를 비교하고 이상치가 존재하는지를 점검하는데 이를 위해 다음의 자료를 seaborn으로부터 호출하여 사용하자.

```
tips=sns.load_dataset('tips')
tips.head()
```

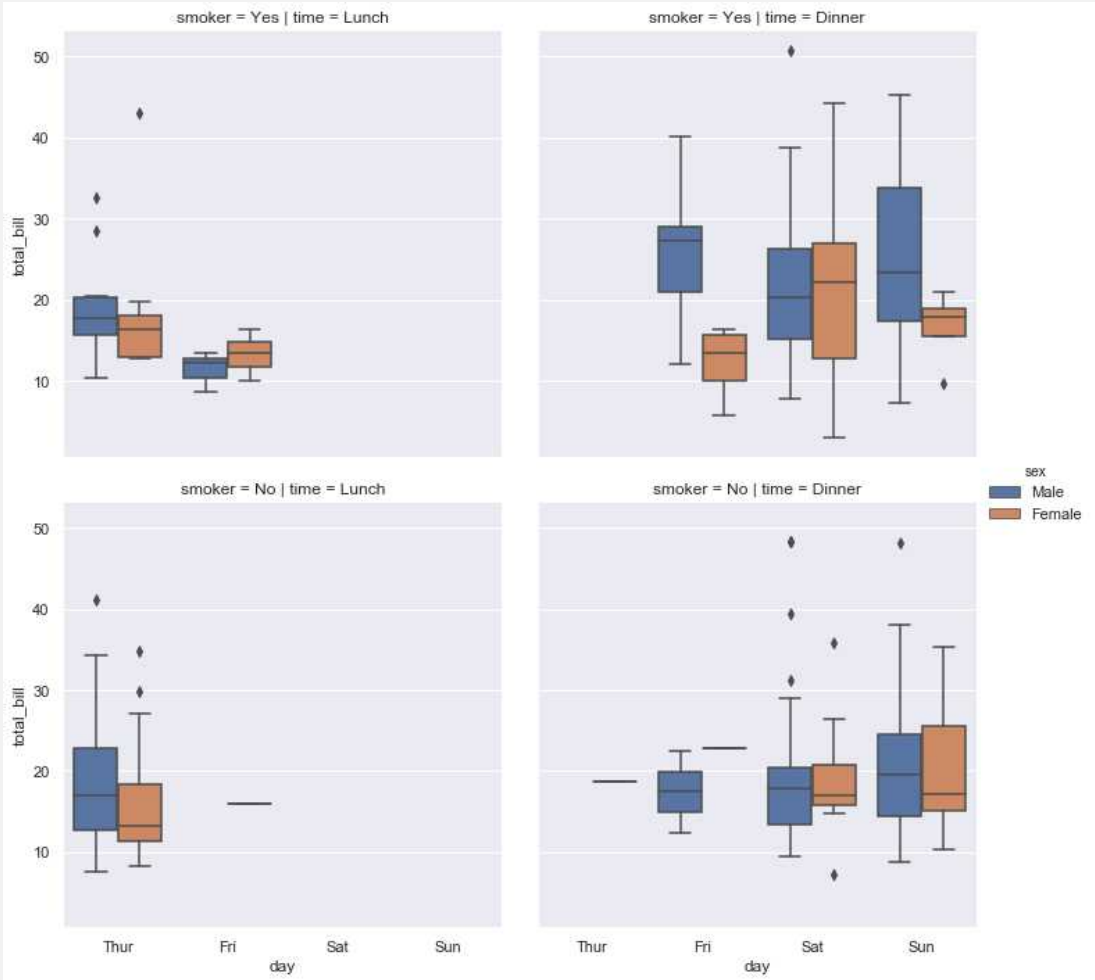
Out[1]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

total_bill, tips, size는 실수 값을 갖는 변수이고 sex, smoker, day, time은 모두 범주형(categorical)자료이다. day별로 성별, total_bill을 비교하고자 하는데 smoker와 time 부분그룹(즉,(smoker,Lunch), (smoker,Dinner), (NonSmoker,Lunch), (Nonsmoker,Dinner) 로 네 개의 부분 그룹)으로 나누어 비교하고자 할 때 아래와 같이 비교하면 된다.

```
sns.factorplot(x='day',y='total_bill',hue='sex',data=tips,
                    row='smoker',col='time', kind='box')
```

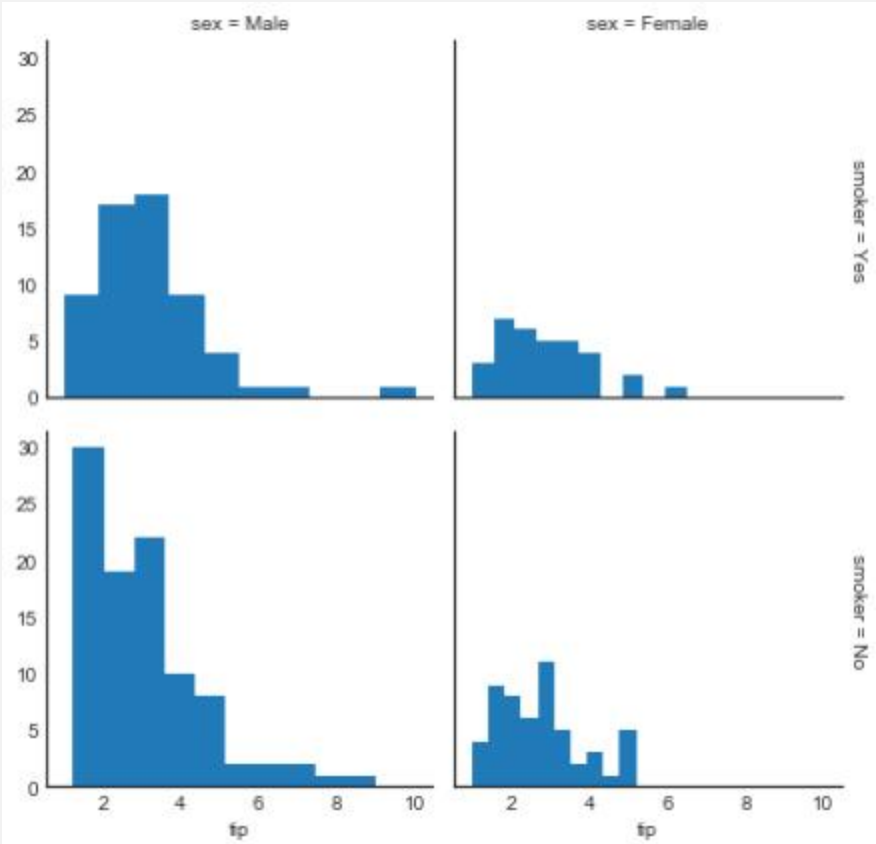
Out[1]:



히스트그램도 부분그룹 내에서 그릴 수 있다. 예를 들어, smoker×sex 그룹별로 tip의 분포를 알고 싶을 경우 아래와 같이 살펴볼 수 있다.

```
grid=sns.FacetGrid(data=tips, row='smoker', col='sex', margin_titles=True)
grid.map(plt.hist, 'tip')
```

Out[1]:



margin_titles=True을 부여함으로써 그림의 열은 sex=Male 또는 Female로 표시하고 행은 smoker=yes 또는 No으로 표시해 준다.