



numpy

numpy

❖ numpy

- ✓ Numerical Python의 줄임말로 파이썬에서 과학적 계산을 위한 핵심 라이브러리
- ✓ 고성능 다차원 배열 객체와 이들 배열과 함께 작동하는 도구들을 제공
- ✓ CPython에서만 사용 가능

❖ 배열(ndarray)

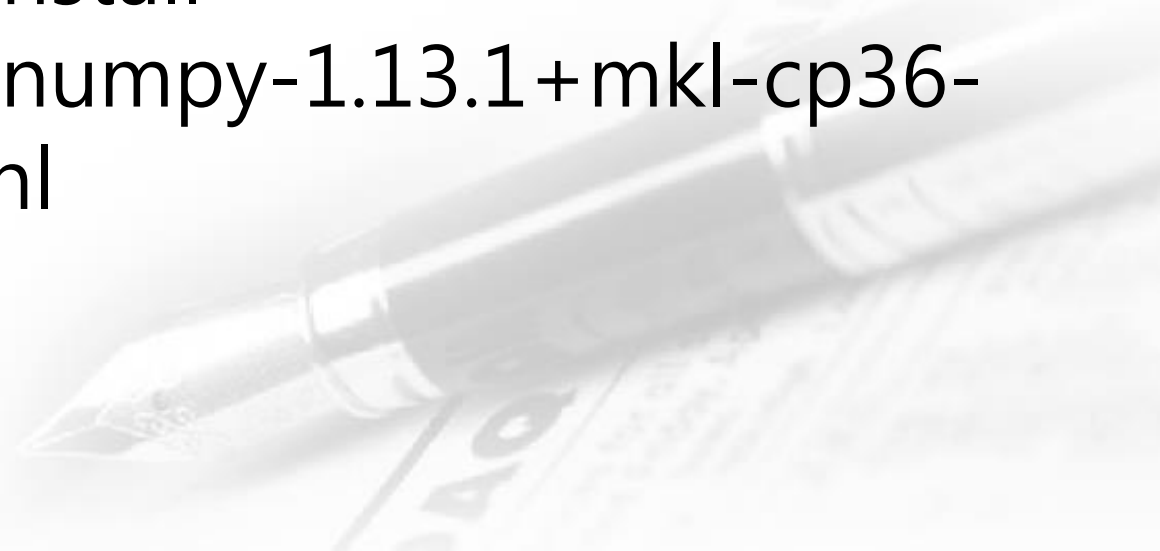
- ✓ list 나 tuple 보다 생성 방법이 다양
- ✓ numpy의 배열은 모두 동일한 자료형의 grid
- ✓ 생성: `numpy.array(컬렉션객체)`
- ✓ `shape` 속성은 각 차원의 크기를 알려주는 정수 튜플
- ✓ 각 데이터의 자료형은 `dtype`으로 확인 가능
- ✓ 배열객체[인덱스]를 이용해서 각각의 데이터 접근 가능
- ✓ `Print` 함수를 이용해서 출력하면 데이터 각각을 순서대로 출력
- ✓ 배열의 크기 변경은 `numpy.ndarray.reshape` 함수로 가능한데 파라미터는 튜플

❖ import

`import numpy` # 넘파이의 모든 객체를 `numpy.obj` 형식으로 불러 사용할 수 있다.
`import numpy as np` # 넘파이의 모든 객체를 `np.obj` 형식으로 불러 사용할 수 있다.
`from numpy import *` # 넘파이의 모든 객체를 내장 함수 (객체) 처럼 사용 가능.

numpy 설치

- ✓ <http://www.lfd.uci.edu/~gohlke/pythonlibs/#numpy>
- ✓ numpy-1.13.1+mkl-cp36-cp36m-win32.whl 다운
- ✓ c:\python\lib 로 copy
- ✓ `python -m pip install [whl 파일의 경로]`
- ❖ `python -m pip install`
`c:\python\lib\numpy-1.13.1+mkl-cp36-cp36m-win32.whl`



numpy

```
import numpy as np
import datetime
li = range(1,10000)
s = datetime.datetime.now()
print("리스트 작업 시작 시간:", s)
for i in li:
    i = i * 10
s = datetime.datetime.now()
print("리스트 작업 종료 시간:", s)
ar = np.arange(1,10000)
s = datetime.datetime.now()
print("ndarray 작업 시작 시간:", s)
ar = ar * 10
s = datetime.datetime.now()
print("ndarray 작업 종료 시간:", s)
```

리스트 작업 시작 시간: 2017-04-12 07:53:34.114300
리스트 작업 종료 시간: 2017-04-12 07:53:34.116300
ndarray 작업 시작 시간: 2017-04-12 07:53:34.116300
ndarray 작업 종료 시간: 2017-04-12 07:53:34.116300



numpy

```
import numpy as np
ar = np.array([1, 2, 3])
print(type(ar))
print (ar.shape)
print (ar[0], ar[1], ar[2])
ar[0] = 5
print (ar)
br = np.array([[1,2,3],[4,5,6]])
print (br.shape)
print (br[0, 0], br[0, 1], br[1, 0])
```

```
<class 'numpy.ndarray'>
(3,)
1 2 3
[5 2 3]
(2, 3)
1 2 4
```



numpy

❖ 배열 생성

- ✓ 배열을 입력하는 가장 기본적인 함수는 `array()`인데 첫 번째 인자로 리스트를 받는데 이것으로 `array`객체를 생성
- ✓ 파이썬의 내장 함수인 `range()`와 유사하게 배열 객체를 반환해주는 `arange()`
- ✓ 시작점과 끝점을 균일 간격으로 나눈 점들을 생성해주는 `linspace()`
- ✓ 생성된 배열은 `reshape()` 함수를 이용하여 행수와 열수를 조절 가능



numpy

```
import numpy as np
ar = np.array([1, 2, 3]) # 1차 배열 (벡터) 생성
print(ar)
ar = np.array([[1, 2, 3], [4, 5, 6]]) # 2x3 크기의 2차원 배열 (행렬) 생성
print(ar);
ar = np.arange(10) # 1차원 배열 [0,1,2, ..., 9] 생성
print(ar);
ar = np.linspace(0, 1, 6) # start, end, num-points
print(ar);
ar = np.linspace(0, 1, 5, endpoint=False)
print(ar);
ar = np.arange(10) # 1차원 배열 [0,1,2, ..., 9] 생성
print(ar);
ar = ar.reshape(2, 5) # 같은 요소를 가지고 2x5 배열로 변형
print(ar);
```

```
[1 2 3]
[[1 2 3]
 [4 5 6]]
[0 1 2 3 4 5 6 7 8 9]
[ 0.  0.2  0.4  0.6  0.8  1. ]
[ 0.  0.2  0.4  0.6  0.8]
[0 1 2 3 4 5 6 7 8 9]
[[0 1 2 3 4]
 [5 6 7 8 9]]
```

numpy

❖ 특수 행렬 생성

- ✓ `zeros()`, `ones()` 라는 함수들은 각각 0과 1로 채워진 배열을 생성하는데 차수를 정수 혹은 튜플로 받습니다.
- ✓ 대각행렬을 생성하기 위한 `eye()` 함수와 `diag()` 함수도 존재
- ✓ `eye()` 함수는 항등행렬을 생성하고 `diag()`는 주어진 정방행렬에서 대각요소만 뽑아내서 벡터를 만들거나 반대로 벡터요소를 대각요소로 하는 정방행렬을 생성
- ✓ 정의 : `eye(N, M=, k=, dtype=)`, M은 열 수, k는 대각 위치(대각일 때가 0), dtype은 데이터형
- ✓ `empty()` 라는 함수는 크기만 지정해 두고 각각의 요소는 초기화 시키지 않고 배열을 생성하는데 이 함수로 생성된 배열의 요소는 가비지 값이 채워져 있습니다.



numpy

```
import numpy as np
b1 = np.ones( 10 ) # 1로 채워진 10 크기의 (1차원) 배열 생성
print(b1)
b2 = np.zeros( (5,5) ) # 0으로 채워진 5x5 크기의 배열 생성
print(b2)
ar = np.eye(2, dtype=int)
print(ar)
ar = np.eye(3, k=1)
print(ar)
ar = np.arange(9).reshape((3,3))
print(ar)
br = np.diag(ar)
print(br)
cr = np.diag(ar,k=1)
print(cr)
dr = np.diag(ar, k=-1)
print(dr)
aar = np.empty( (2,2) ) # 가비지 값으로 채워진 2x2 크기의 배열 생성
print(ar)
```

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

numpy

❖ 난수 생성

- ✓ numpy는 효율적으로 무작위 샘플을 만들 수 있는 numpy.random 모듈을 제공합니다.
- ✓ np.random.normal(size=개수 또는 shape): size를 생략하면 1개의 데이터만 리턴하고 개수를 입력하면 그 개수에 해당하는 데이터를 배열로 리턴하고 shape에 해당하는 배열을 리턴
- ✓ np.random.seed(seed=시드번호): 시드번호를 가지고 난수를 생성
- ✓ np.random.binomial(n, p, size): 이항분포로부터의 무작위 추출 함수로 n은 0부터 나올 수 있는 숫자의 범위로 정수 p는 확률이고 size는 개수



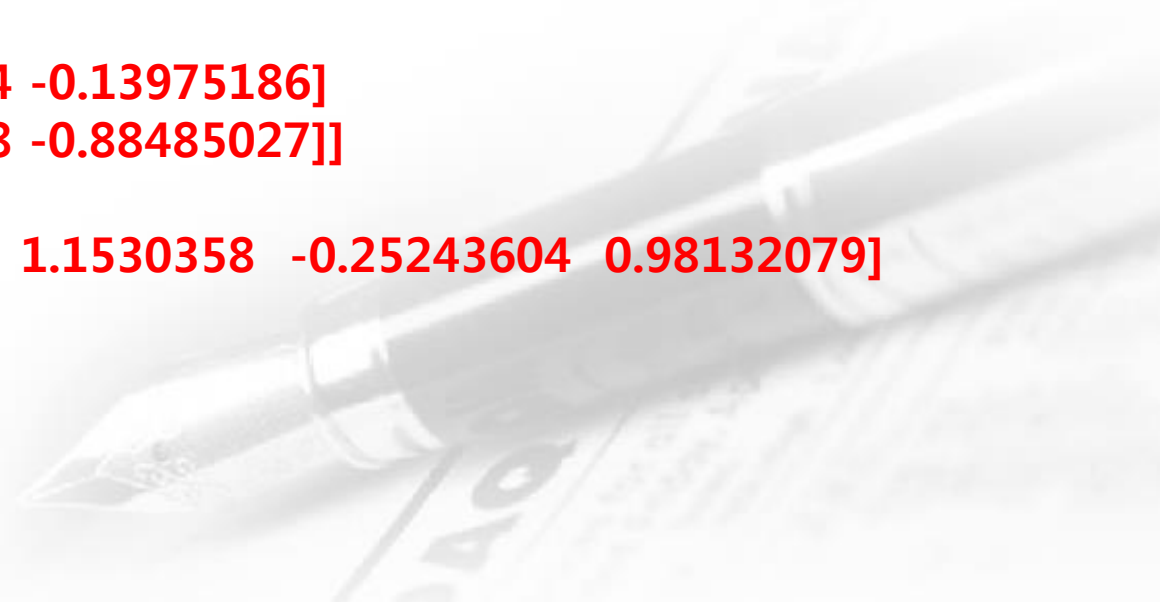
numpy

```
import numpy as np
print(np.random.normal(size=5)) #5개의 난수 생성 - 랜덤
print()
print(np.random.normal(size=(2, 3))) #2행 3열의 난수 생성
print()
np.random.seed(seed=100) #시드 설정
print(np.random.normal(size=5)) #5개의 난수 생성 - 실행할 때마다 동일한 값
```

[-0.24961316 -0.48887513 -0.36998486 0.74862078 -0.27312433]

**[[1.40273795 0.84421724 -0.13975186]
[0.00408551 0.90391068 -0.88485027]]**

[-1.74976547 0.3426804 1.1530358 -0.25243604 0.98132079]



numpy

❖ ndarray의 자료형

- ✓ 객체를 생성할 때 dtype이라는 파라미터를 이용해서 설정 가능하고 dtype이라는 속성을 이용해서 확인 가능 : `np.array([xx, xx], dtype=np.Type)`
- ✓ 서로 다른 데이터 타입의 데이터를 생성할 때 대입하면 자동 형 변환을 해서 배열이 생성됩니다.
- ✓ 정수와 실수가 혼합되어 있다면 실수, 숫자와 문자열이 혼합되어 있다면 문자열로 생성됩니다.
- ✓ `astype` 함수를 이용해서 데이터의 타입을 변경할 수 있습니다.
- ✓ 자료형의 종류

정수 자료형: int8, 16, 32, 64, uint8, uint16, uint32, uint64

실수 자료형: float16, 32, 64, 128

복소수 자료형: complex64, 128, 256

boolean: bool

객체: object

문자열: string_

유니코드: unicode_



numpy

```
import numpy as np
ar = np.array([1,2,3])
print("타입 확인:",ar.dtype);
ar = np.array(['12', '2', '3'])
print("타입 확인:",ar.dtype);
ar = np.array([3, 2.9, 4])
print("타입 확인:",ar.dtype);
ar = np.array([3, 2.9, '4'])
print("타입 확인:",ar.dtype);
ar = np.array([3, 2, '4'], dtype=np.int32)
print("타입 확인:",ar.dtype);
ar = ar.astype(np.string_)
print("타입 확인:",ar.dtype);
```

타입 확인: int32
타입 확인: <U2
타입 확인: float64
타입 확인: <U32
타입 확인: int32
타입 확인: |S11



numpy

❖ ndarray의 산술 연산

- ✓ 배열과 숫자 데이터와의 연산은 배열의 모든 요소에 숫자 데이터를 연산한 결과를 리턴합니다.
- ✓ 동일한 크기를 갖는 배열간의 산술 연산은 동일한 위치의 데이터끼리 연산을 수행한 후 결과를 리턴합니다.
- ✓ 동일한 크기를 갖는 배열 간의 비교 연산은 동일한 위치의 데이터를 비교해서 True 또는 False로 리턴해서 배열로 만들어 줍니다 – equal, not_equal, greater, greater_equal, less, less_equal
- ✓ 동일한 크기를 갖는 배열 간의 할당 연산: Add AND ($m += n$), Subtract AND ($m -= n$), Multiply AND ($m *= n$), Divide AND ($m /= n$), Floor Division ($m //= n$), Modulus AND ($m %= n$), Exponent AND ($m **= n$)
- ✓ 배열간의 논리 연산
 - ✓ `np.logical_and(a, b)` : 두 배열의 원소가 모두 '0'이 아니면 True 반환
 - ✓ `np.logical_or(a, b)` : 두 배열의 원소 중 한개라도 '0'이 아니면 True 반환
 - ✓ `np.logical_xor(a, b)` : 두 배열의 원소가 모두 '1'이 아니면 True 반환
- ✓ 소속 여부 판단 연산 : `in`, `not in`
- ✓ 배열 (혹은 리스트)에 특정 객체가 들어있으면 True를 반환, 안들어 있으면 False를 반환

numpy

```
import numpy as np
ar = np.array([1,2,3])
br = np.array([4,5,6])
cr = np.array([[6,7,8], [10,20,30]])
result = ar * 2 #배열의 모든 요소에 2를 곱한 결과
print(result)
result = ar + br; #배열 간의 덧셈: 동일한 위치간의 덧셈을 한 결과
print(result)
```

[2 4 6]
[5 7 9]



numpy

```
import numpy as np
ar = np.array([1,2,3])
br = np.array([4,2,6])
print(np.equal(ar, br))
print(np.not_equal(ar, br))
print(np.greater(ar, br))
print(np.greater_equal(ar, br))
print(np.less(ar, br))
print(np.less_equal(ar, br))
```

```
[False True False]
[ True False  True]
[False False False]
[False True False]
[ True False  True]
[ True  True  True]
```



numpy

❖ ndarray의 산술 연산 - broadcasting

- ✓ 동일하지 않은 크기의 배열끼리의 연산은 브로드캐스팅 연산을 수행하게 됩니다.
- ✓ 브로드캐스팅은 기준 축의 데이터 개수가 동일한 경우에만 수행하며 모든 행에 대해서 연산을 수행합니다.



numpy

```
import numpy as np
a = np.array([10,20,30])
b = np.arange(12).reshape((4, 3))
print(a + b) #b의 각 행에 a의 데이터를 더한 결과
a = np.array([10,20,30]).reshape(3,1)
b = np.arange(12).reshape((3, 4))
print(a + b)
```

```
[[10 21 32]
 [13 24 35]
 [16 27 38]
 [19 30 41]]
[[10 11 12 13]
 [24 25 26 27]
 [38 39 40 41]]
```



numpy

❖ 인덱스

- ✓ 1차원 배열의 인덱스는 list와 유사
- ✓ 인덱스에 음수를 사용하면 뒤에서부터 계산합니다.
- ✓ 2차원 이상의 배열에서 인덱스는 [행번호][열번호]의 형태로 입력해도 되지만 [행번호, 열번호]를 이용할 수 있습니다.
- ✓ 2차원 이상의 배열에서 인덱스를 생략하는 경우에는 생략된 인덱스의 데이터 전체를 의미합니다.
- ✓ 2차원 이상의 배열에서 인덱스를 리스트 형태로 입력하면 리스트에 해당하는 데이터만 리턴합니다.
- ✓ 리스트의 리스트로 대입하는 것도 가능

`[[0,1],[2,3]] => [0,2], [1,3]`

`[[0,1]][[:,[1,2]]] => [0,1],[0,2],[1,1],[1,2]`



numpy

```
import numpy as np
ar = np.array([1,2,3])
print("ar[0]:", ar[0])
print("ar[-1]:", ar[-1])
```

```
ar = np.array([[1,2],[3,4]])
print("ar[0][1]:", ar[0][1])
print("ar[0,1]:", ar[0,1])
print("ar[0]:", ar[0])
```

```
ar[0]: 1
ar[-1]: 3
ar[0][1]: 2
ar[0,1]: 2
ar[0]: [1 2]
```



numpy

```
import numpy as np
ar = np.empty((10, 5)) #가바게 값을 갖는 10행 5열의 배열을 생성
for i in range(10):
    ar[i] = i #각 행의 모든 데이터를 i 값으로 채움
#print(ar)
br = ar[[1,3,5,7]] #1,3,5,7 행만 선택
#print(br)
cr = ar[[0,1], [3,4]] #[0,3],[1,4] 만 선택
#print(cr)
dr = ar[[0,1]][:,[3,4]] #0번 행의 3번째와 4번째 1번행의 3번째와 4번째 데이터 선택
print(dr)
```

[[0. 0.]
 [1. 1.]



numpy

❖ 슬라이싱

- ✓ 슬라이싱을 하면 복제가 되지 않습니다.
- ✓ 복제를 하고자 하면 copy 메서드를 호출해서 리턴받아야 합니다.
- ✓ [시작위치:끝위치]의 형태로 슬라이싱 가능
- ✓ 2차원 이상의 배열의 경우는 [행의 슬라이싱, 열의 슬라이싱]의 형태로 슬라이싱 가능



numpy

```
import numpy as np □  
ar = np.array([1,2,3,4,5])  
br = ar[0:3]  
cr = ar[0:3].copy()  
ar[0]=10  
print(br)  
print(cr)  
br = ar[-2:]  
print(br)
```

```
[10 2 3]  
[1 2 3] □  
[4 5]
```



numpy

❖ 슬라이싱

- ✓ 특정 조건을 만족하는 배열의 모든 열을 선별하기 : ==
- ✓ 특정 조건을 만족하지 않는 배열의 모든 열을 선별하기 : !=, ~(==)
- ✓ 여러 조건을 가지고 작업하기: & (and), | (or)
- ✓ 조건에 맞는 데이터를 선택한 후 = 을 이용해서 특정한 값을 할당할 수 있습니다.



numpy

```
import numpy as np
ar = np.arange(20).reshape(5, 4)
br = np.array(['A', 'B', 'C', 'A', 'C'])
print(br == 'A') #데이터가 A인 경우는 True 그렇지 않으면 False
print(ar[br == 'A']) #True 인 행만 반환
print(ar[br == 'A', 2]) #열을 2번째만 반환
print(ar[br == 'A', 0:2]) #열을 0부터 2앞까지 반환
ar[br == 'A'] = 100
print(ar)
```

```
[ True False False  True False]
[[ 0  1  2  3]
 [12 13 14 15]]
[ 2 14]
[[ 0  1]
 [12 13]]
[[100 100 100 100]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [100 100 100 100]
 [16 17 18 19]]
```

numpy

❖ 슬라이싱

- ✓ 특정 조건을 만족하는 배열의 모든 열을 선별하기 : ==
- ✓ 특정 조건을 만족하지 않는 배열의 모든 열을 선별하기 : !=, ~(==)
- ✓ 여러 조건을 가지고 작업하기: & (and), | (or)
- ✓ 조건에 맞는 데이터를 선택한 후 = 을 이용해서 특정한 값을 할당할 수 있습니다.

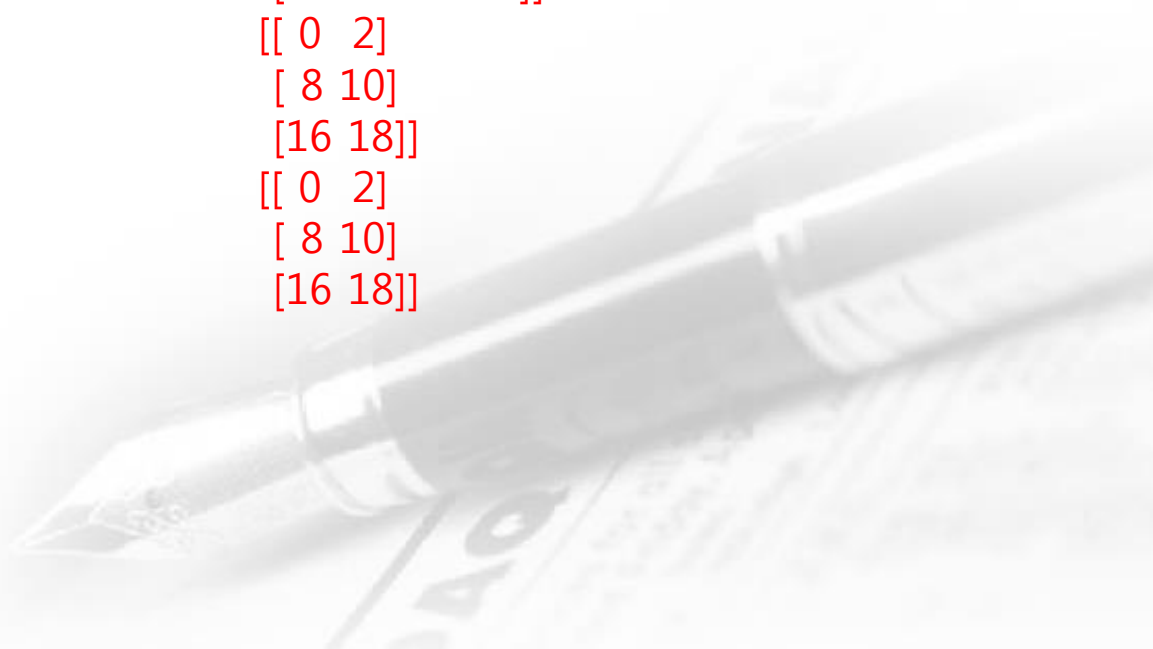
❖ Fancy Indexing

- ✓ 정수 배열을 indexer로 사용해서 다차원 배열로 부터 Indexing
- ✓ Fancy Indexing은 copy를 만듭니다.
- ✓ 특정 순서로 다차원 배열의 행(row)과 열(column)을 Fancy Indexing
 - 특정 순서로 행(row)을 fancy indexing 합니다. 그런 후에 전체 행을 ':'로 선택하고, 특정 칼럼을 순서대로 배열을 사용해서 indexing을 한번 더 해주는 겁니다.
 - np.ix_ 함수를 사용해서 배열1 로 특정 행(row)을 지정, 배열2 로 특정 열(column)을 지정해주는 것입니다.

numpy

```
import numpy as np
ar = np.arange(20).reshape(5, 4)
print(ar)
print(ar[[1,2]])#1행과 2행만 선택하기
print(ar[[-1, -2]])# 뒤에서 2개행 선택하기
print(ar[[0, 2, 4]][:, [0, 2]])
print(ar[np.ix_([0, 2, 4], [0, 2])])
```

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



numpy

❖ 행과 열 변환

- ✓ T라는 속성을 이용하면 행과 열을 변환할 수 있습니다.
- ✓ `transpose()` 메서드를 이용해서도 행과 열을 변환할 수 있습니다.
- ✓ `transpose`는 매개변수가 없으면 행과 열 및 기타 차원의 순서를 반대로 적용하게 되고 매개변수에 축의 순서를 대입하게 되면 축의 순서대로 변환하게 됩니다.



numpy

```
import numpy as np
ar = np.array([[1,2,3], [4,5,6]])
print(ar)
print()
print(ar.T)
```

```
[[1 2 3]
 [4 5 6]]
```

```
[[1 4]
 [2 5]
 [3 6]]
```



numpy

```
import numpy as np
ar = np.array([[[1,2],[3,4],[5,6]],[[71,72],[73,74],[75,76]]])
print()
print(ar.transpose())
print()
print(ar.transpose(2,1,0))
print()
print(ar.transpose(1,0,2))
```

```
[[[ 1 71]
 [ 3 73]
 [ 5 75]]
```

```
[[ 2 72]
 [ 4 74]
 [ 6 76]]]
```

```
[[[ 1 71]
 [ 3 73]
 [ 5 75]]
```

```
[[ 2 72]
 [ 4 74]
 [ 6 76]]]
```

```
[[[ 1  2]
 [71 72]]
```

```
[[ 3  4]
 [73 74]]
```

```
[[ 5  6]
 [75 76]]]
```

numpy

❖ 범용함수(Universal function)

- ✓ python에서 제공하는 math module은 실수(real number)에 대해서만 범용함수를 지원하며, cmath module 은 복소수(complex number) 까지 범용함수를 지원합니다.
- ✓ numpy module은 실수, 복소수, 복소수 행렬 (complex matrix)의 원소 간 범용 함수를 모두 지원하므로 사용 범위가 가장 넓어 매우 유용합니다.
- ✓ 배열의 원소간 연산을 위해 numpy의 Universal function 함수는 쓸모가 많습니다.
- ✓ numpy 범용 함수는 몇 개의 배열에 대해 적용이 되는지에 따라 분류
 - 1개의 배열에 적용하는 Unary Universal Functions (ufuncs)
 - 2개의 배열에 대해 적용하는 Binary Universal Functions (ufuncs)



numpy

❖ 기본 통계 함수

- ✓ sum: 배열에 있는 모든 원소의 합을 계산해서 리턴
- ✓ mean, median, std, var: 평균값, 중간값, 표준편차, 분산

```
ar = np.array([1, 2, 4, 5, 5, 7, 9, 10, 13, 18, 21])
```

```
print(np.sum(ar))
```

```
print(np.mean(ar))
```

```
print(np.median(ar))
```

```
print(np.std(ar))
```

```
print(np.var(ar))
```

95

8.63636363636

7.0

6.1388883695

37.6859504132



numpy

❖ 소수 관련 함수

- ✓ `np.around(a)` : 0.5를 기준으로 올림 혹은 내림
- ✓ `np.round_(a, N)` : N 소수점 자릿수까지 반올림
- ✓ `np rint(a)` : 가장 가까운 정수로 올림 혹은 내림
- ✓ `np.fix(a)` : '0' 방향으로 가장 가까운 정수로 올림 혹은 내림
- ✓ `np.ceil(a)` : 각 원소 값보다 크거나 같은 가장 작은 정수 값 (천장 값)으로 올림
- ✓ `np.floor(a)` : 각 원소 값보다 작거나 같은 가장 큰 정수 값 (바닥 값)으로 내림
- ✓ `np.trunc(a)` : 각 원소의 소수점 부분은 잘라버리고 정수 값만 남김



numpy

```
import numpy as np
ar = np.array([-4.62, -2.19, 0, 1.57, 3.40, 4.06])
print(np.around(ar))
print(np.round_(ar, 1))
print(np rint(ar))
print(np.fix(ar))
print(np.ceil(ar))
print(np.floor(ar))
print(np.trunc(ar))
```

```
[-5. -2.  0.  2.  3.  4.]
[-4.6 -2.2  0.  1.6  3.4  4.1]
[-5. -2.  0.  2.  3.  4.]
[-4. -2.  0.  1.  3.  4.]
[-4. -2.  0.  2.  4.  5.]
[-5. -3.  0.  1.  3.  4.]
[-4. -2.  0.  1.  3.  4.]
```



numpy

❖ 배열 통계

- ✓ `np.prod()`: 2차원 배열의 경우 `axis=0` 이면 같은 열(column)의 위*아래 방향으로 배열 원소 간 곱하며, `axis=1` 이면 같은 행(row)의 왼쪽*오른쪽 원소 간 곱을 합니다.
- ✓ `np.sum()`: `keepdims=True` 옵션을 설정하면 1 차원 배열로 배열 원소 간 합을 반환합니다.
- ✓ `np.nanprod()`: NaN (Not a Numbers) 을 '1'(one)로 간주하고 배열 원소 간 곱을 합니다.
- ✓ `np.nansum()`: NaN (Not a Numbers)을 '0'(zero)으로 간주하고 배열 원소 간 더하기를 합니다.
- ✓ `np.cumprod()`: `axis=0` 이면 같은 행(column)의 위에서 아래 방향으로 배열 원소들을 누적(cumulative)으로 곱해 나가며, `axis=1` 이면 같은 열(row)에 있는 배열 원소 간에 왼쪽에서 오른쪽 방향으로 누적으로 곱해 나갑니다.
- ✓ `np.cumsum()`: `axis=0` 이면 같은 행(column)의 위에서 아래 방향으로 배열 원소들을 누적(cumulative)으로 합해 나가며, `axis=1` 이면 같은 열(row)에 있는 배열 원소 간에 왼쪽에서 오른쪽 방향으로 누적으로 합해 나갑니다.
- ✓ `np.diff()`: 배열 원소 간 n차 차분 구하기

numpy

```
import numpy as np
b = np.array([1, 2, 3, 4])
c = np.array([[1, 2], [3, 4]])
print(np.prod(b)) # 1*2*3*4
print(np.prod(c, axis=0)) # [1*3, 2*4]
print(np.prod(c, axis=1)) # [1*2, 3*4]
print(np.sum(b)) # [1+2+3+4]
print(np.sum(b, keepdims=True))
print(np.sum(c, axis=0)) # [1+3, 2+4]
print(np.sum(c, axis=1) ) # [1+2, 3+4]
```

```
24
[3 8]
[ 2 12]
10
[10]
[4 6]
[3 7]
```



numpy

```
import numpy as np
b = np.array([1, 2, 3, 4])
c = np.array([[1, 2], [3, 4]])
print(np.cumprod(b))# [1, 1*2, 1*2*3, 1*2*3*4]
print(np.cumprod(c, axis=0)) # [[1, 2], [1*3, 2*4]]
print(np.cumprod(c, axis=1)) # [[1, 1*2], [3, 3*4]]
print(np.cumsum(b))# [1, 1+2, 1+2+3, 1+2+3+4]
print(np.cumsum(c, axis=0)) # [[1, 2], [1+3, 2+4]]
print(np.cumsum(c, axis=1)) # [[1, 1+2], [3, 3+4]]
```

```
[ 1  2  6 24]
[[1 2]
 [3 8]]
[[ 1  2]
 [ 3 12]]
[ 1  3  6 10]
[[1 2]
 [4 6]]
[[1 3]
 [3 7]]
```



numpy

```
import numpy as np  
g = np.array([1, 2, 4, 10, 13, 20])  
print(np.diff(g)) # [2-1, 4-2, 10-4, 13-10, 20-13]  
print(np.diff(g, n=2)) # [2-1, 6-2, 3-6, 7-3]
```

```
[1 2 6 3 7]  
[ 1 4 -3 4]
```



numpy

❖ 지수 로그 함수

- ✓ `np.exp()` 함수는 밑(base)이 자연상수 e 인 지수함수 로 변환해줍니다.
- ✓ `np.log(x)`, `np.log10(x)`, `np.log2(x)`, `log1p(z)`: 지수함수의 역함수인 로그함수는 밑이 자연상수 e , 혹은 10, 또는 2 이냐에 따라서 `np.log(x)`, `np.log10(x)`, `np.log2(x)` 를 구분해서 사용

❖ 삼각 함수

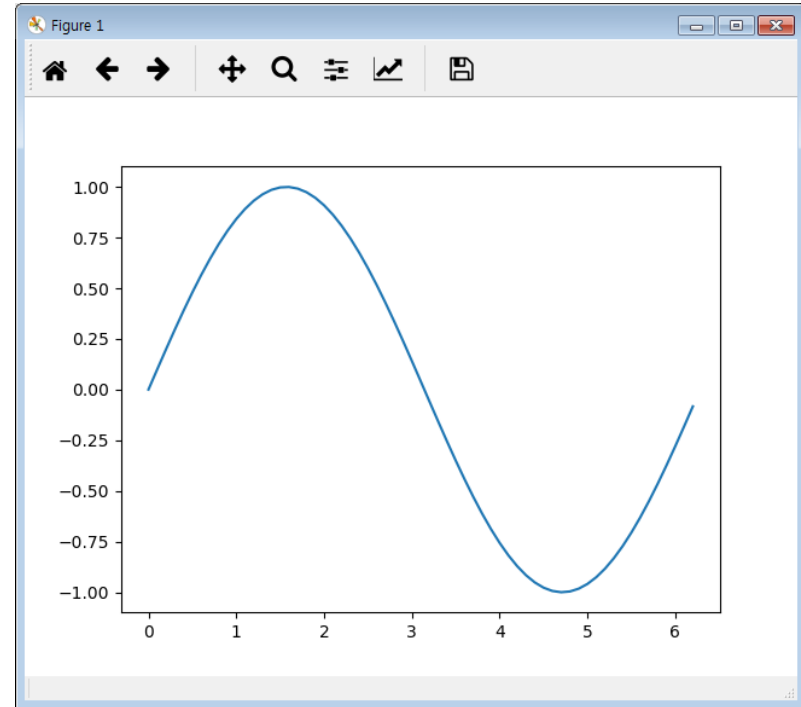
- ✓ `np.sin()`, `np.cos()`, `np.tan()`: 각도는 라디안을 사용하기 때문에 `degree * np.pi/180`으로 연산을 해서 작업을 수행
- ✓ `np.arcsin()`, `np.arccos()`, `np.arctan()`: 역삼각함수



numpy

C:\Users\Wkitcoop\Anaconda3\Scripts
pip install matplotlib

```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(0, 2* np.pi, 0.1)
y = np.sin(x)
plt.plot(x, y)
plt.show()
```



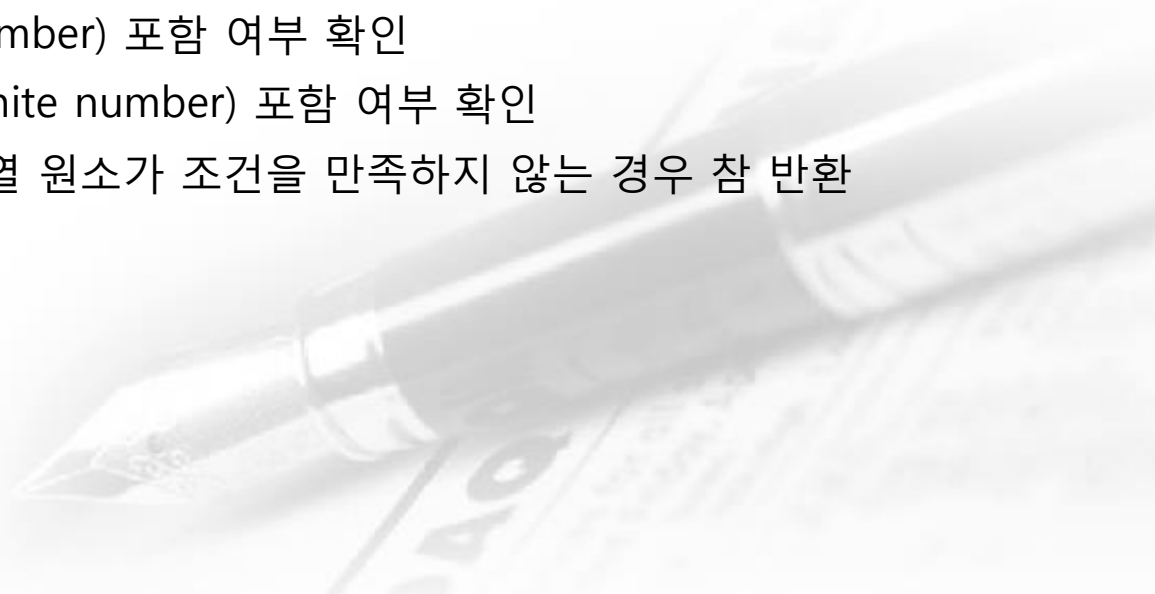
numpy

❖ 숫자 처리 함수

- ✓ `np.abs(x)`, `np.fabs(x)`: 배열 원소의 절대값 (absolute value)
- ✓ `np.sqrt(y)`: 배열 원소의 제곱근 (Square Root)
- ✓ `np.square(y)`: 배열 원소의 제곱값 (square value) 범용 함수
- ✓ `np.modf(z)`: 배열 원소의 정수와 소수점을 구분하여 2개의 배열 반환
- ✓ `np.sign(x)`: 배열 원소의 부호 판별 함수 -> 1 (positive), 0 (zero), -1 (negative)

❖ 논리 함수

- ✓ `np.isnan(x)`: NaN (Not a Number) 포함 여부 확인
- ✓ `np.isfinite(x)`: 유한수 (finite number) 포함 여부 확인
- ✓ `np.isinf(x)`: 배열에 무한수 (infinite number) 포함 여부 확인
- ✓ `np.logical_not(condition)`: 배열 원소가 조건을 만족하지 않는 경우 참 반환



numpy

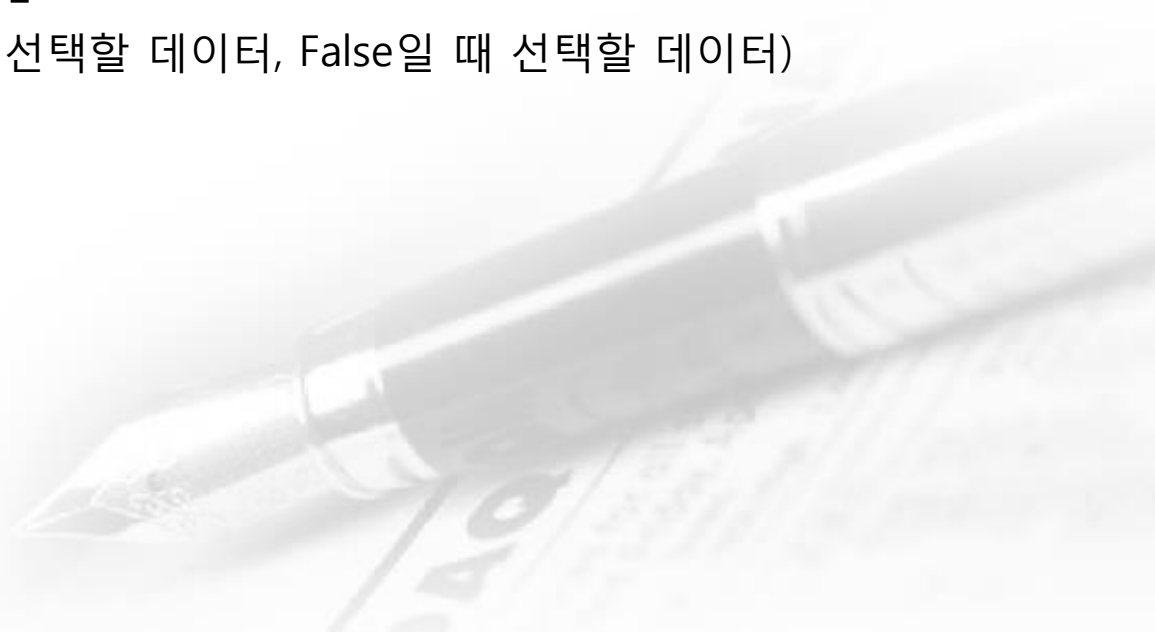
```
import numpy as np
b = np.array([10, 1, 2, 3, 4])
print(np.logical_not( b <= 2 ))
c = b[np.logical_not( b <= 2 )]
print(c)
```

```
[ True False False  True  True]
[10  3  4]
```



numpy

- ❖ 이항 함수: 배열 2개를 가지고 작업하는 함수
 - ✓ add, subtract, multiply, divide, floor_divide
 - ✓ power
 - ✓ maximum, fmax, minimum, fmin
 - ✓ mod
 - ✓ copysign
 - ✓ greater, greater_equal, less, less_equal, equal, not_equal
 - ✓ logical_and, logical_or, logical_xor
- ❖ where(boolean 배열, True일 때 선택할 데이터, False일 때 선택할 데이터)



numpy

```
import numpy as np
ar = np.array([1, 2, 3, 4])
br = np.array([5, 6, 7, 8])
print(ar + br)
cond = [True, False, False, True]
print(np.where(cond, ar, br))
```

```
[ 6  8 10 12]
[ 1  6  7  4]
```



numpy

- ❖ 이항 함수: 배열 2개를 가지고 작업하는 함수
 - ✓ `numpy.concatenate`: 여러 개의 배열을 한 개로 합치는 함수로 X축과 Y축 방향으로 합치는 두 가지 방법이 있으며 `axis` 라는 파라미터를 통해 제어함.
 - `axis = 0`: Y축 (세로 방향) 으로 설정
 - `axis = 1`: X축 (가로 방향) 으로 설정



numpy

```
import numpy as np
ar = [1,2,3,4]
br = [5,6,7,8]
cr = np.concatenate((ar, br))
print(cr)
ar = np.array([[1, 2], [3, 4]])
br = np.array([[5, 6], [7, 8]])
cr = np.concatenate((ar, br), axis = 0)
print(cr)
cr = np.concatenate((ar, br), axis = 1)
print(cr)
```

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



numpy

❖ numpy.split

- ✓ 배열을 여러 개의 크기로 나누어주는 함수
- ✓ 나누는 방법은 X축, 그리고 Y축을 기준을 나누는 두 가지의 방법이 있으며 numpy.concatenate의 axis와 동일하게 작동
- ✓ 또한 두 번째 파라미터에 숫자 N을 넣으면 배열을 N개의 동일한 크기의 배열들로 나누고 리스트를 넣는다면 리스트 안의 숫자들 번째 인덱스에서 배열을 나눈다.

❖ ndarray.sort()

- ✓ 배열의 데이터를 정렬
- ✓ numpy.sort()는 정렬한 결과를 리턴
- ✓ axis 매개변수는 정렬할 축 번호



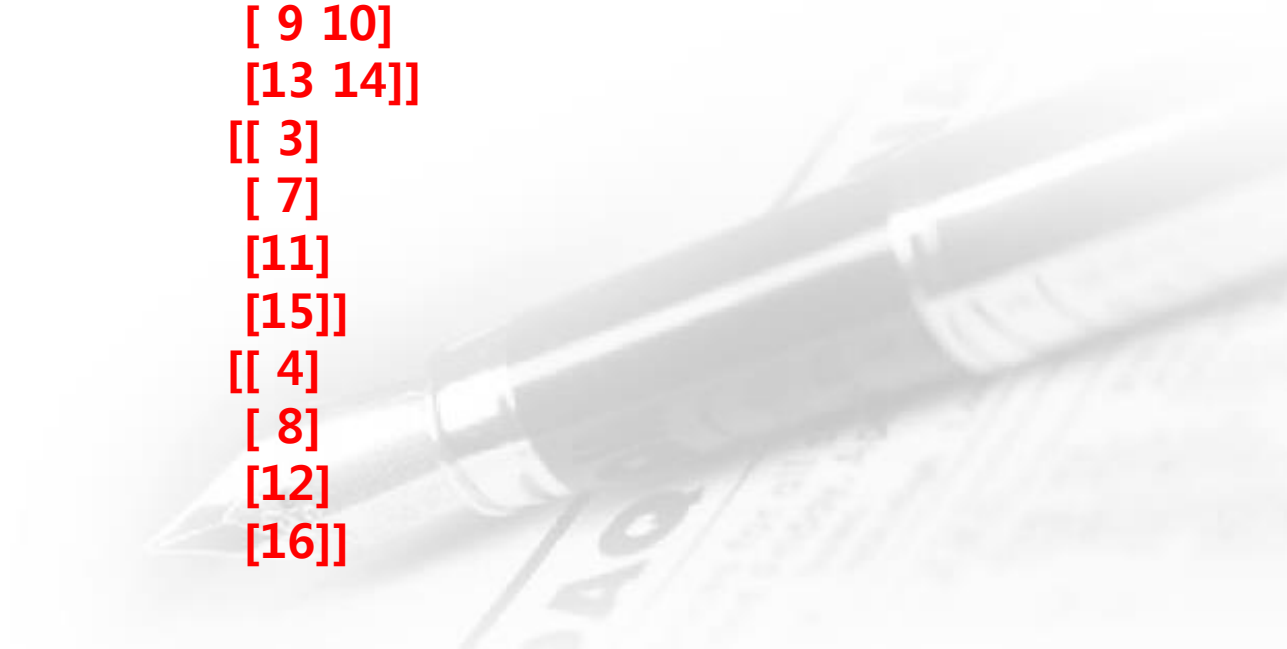
numpy

```
import numpy as np
ar = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]])
print(ar)
slice_Y_equal_size = np.split(ar, 2, axis = 0) #x축 방향으로 2개로 나눔
print(slice_Y_equal_size[0])
print(slice_Y_equal_size[1])
slice_X_different_sizes = np.split(ar, [2, 3], axis = 1) #2번째 앞까지 나누고 3번째 앞까지 나누고 나머지
print(slice_X_different_sizes[0])
print(slice_X_different_sizes[1])
print(slice_X_different_sizes[2])
```



numpy

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



numpy

```
import numpy as np  
ar = np.array([90, 40, 30, 78])  
ar.sort()  
print(ar)  
br = np.sort(ar)  
print(br)
```

```
[30 40 78 90]  
[30 40 78 90]  
[[30 40 27]  
 [90 78 50]]
```

```
ar = np.array([[90, 40, 50], [30, 78, 27]])  
ar.sort(axis=0)  
print(ar)
```



numpy

```
import numpy as np
ar = np.array([90, 40, 30, 78])
ar.sort()
print(ar[0:int(0.5 * len(ar))]) #하위 50%
print(ar[int(0.5 * len(ar))-1:]) #상위 50%
```

[30]

[90]



numpy

❖ 집합 관련 함수

- ✓ `unique()`: 중복을 제거
- ✓ `intersect1d()`: 교집합
- ✓ `union1d()`: 합집합
- ✓ `in1d()`: 데이터의 존재 여부를 boolean 배열로 리턴
- ✓ `setdiff1d()`: 차집합
- ✓ `setxor1d()`: 한쪽에만 있는 데이터의 집합



numpy

```
import numpy as np
ar = np.array([90, 40, 30, 78, 30])
print(np.unique(ar))
br = np.array([30, 45, 76, 90])
print(np.intersect1d(ar, br))
print(np.union1d(ar, br))
print(np.in1d(ar, br))
print(np.setdiff1d(ar, br))
print(np.setxor1d(ar, br))
```

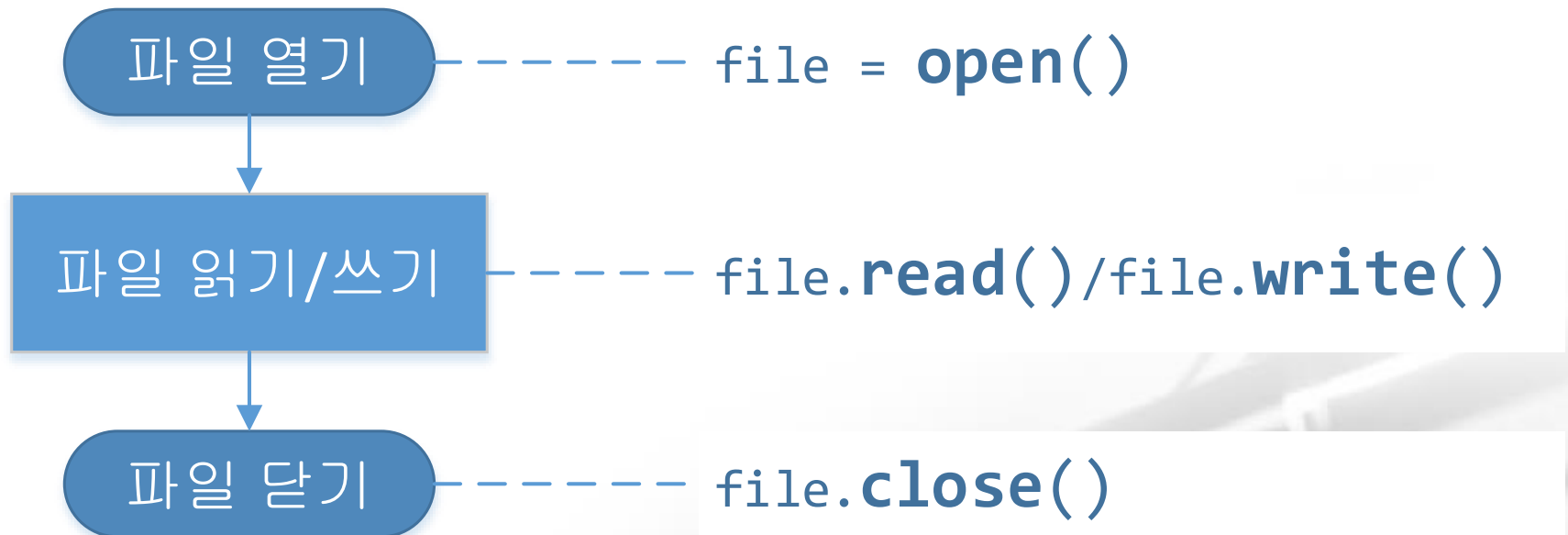
```
[30 40 78 90]
[30 90]
[30 40 45 76
 78 90]
[ True False
 True False  True]
[40 78]
[40 45 76 78]
```



파일 처리

파일처리

- ❖ 어플리케이션이 운영체제에게 파일 처리를 API 함수를 통해 요청하면, 운영체제가 요청한 업무를 수행해주고 그 결과를 어플리케이션에게 돌려줌.
- ❖ 어플리케이션이 파일 처리 업무를 의뢰하는 과정과 각 과정에서 사용되는 파이썬 함수



파일처리

❖ open() 함수

✓ 파일 쓰기

- 파일에 기록하는 경우에는 2개나 3개의 매개변수를 대입
- 첫번째 매개변수는 파일 경로이고 두번째는 'w' 이고 세번째는 인코딩 방식입니다.
- 인코딩 방식의 경우 생략이 가능한데 생략을 하게되면 시스템의 기본적인 문자 인코딩으로 저장 되게 됩니다.
- open()으로 연 파일 객체에 기록을 할 때는 write 메소드를 이용해서 기록할 내용을 전달하면 파일에 기록이 됩니다.
- \n이 포함된 문자열의 컬렉션의 경우에는 writelines를 이용해서 줄 단위 기록이 가능
- \n이 포함된 문자열의 컬렉션의 경우에는 join을 이용하면 write 메소드로도 줄 단위 기록이 가능하고 \n이 없는 경우에도 가능

✓ 파일 읽기

- 파일 경로만 입력하면 읽기 모드로 열리게 됩니다.
- 두번째 매개변수로 'r'을 전달해도 읽기 모드가 되고 생략하고 인코딩 방식을 대입해도 됩니다.
- 전체 데이터를 하나의 문자열 읽고자 하는 경우에는 read()를 이용하면 전체를 읽어 옵니다.
- 줄단위로 읽을 때는 파일 객체의 반복자나 readline() 또는 파일 전체를 줄 단위로 끊어서 리스트에 저장하는 readlines 메소드를 이용해서 줄 단위로 읽을 수 있습니다.
- read(정수)를 이용하면 정수만큼의 바이트를 읽어옵니다.

파일처리

❖ 파일 열기 모드: open() 함수의 두번째 매개변수

| 문자 | 의미 |
|-----|---|
| 'r' | 읽기용으로 열기 (기본값) |
| 'w' | 쓰기용으로 열기. 이미 같은 경로에 파일이 존재하면 파일내용을 비움. |
| 'x' | 배타적 생성모드로 열기. 파일이 존재하면 IOError 예외 일으킴. |
| 'a' | 쓰기용으로 열기. 단, 'w'와는 달리 이미 같은 경로에 파일이 존재하는 경우 기존 내용에 덧붙이기를 함. |
| 'b' | 바이너리 모드 |
| 't' | 텍스트 모드 (기본값) |
| '+' | 읽기/쓰기용으로 파일 읽기 |



파일처리

❖ 파일의 경로 설정

✓ 절대 경로

- 파일의 경로를 루트부터 직접 기재하는 방식
- 리눅스나 매킨토시는 디렉토리 구분 기호로 /를 사용하고 윈도우의 경우에는 \를 사용
- 절대 경로를 사용할 때는 윈도우에서는 [\\로](#) 설정해야 합니다.

✓ 상대경로

- 파일의 경로를 현재 위치로부터의 상대적인 경로로 설정하는 방식
- /를 이용해서 디렉토리를 구분
- ./는 현재 디렉토리로 생략해도 현재 디렉토리가 됩니다.
- ../는 상위 디렉토리가 됩니다.



파일 쓰기

- ❖ 프로그램을 실행하고 현재 작업 디렉토리 확인

```
file = open('test.txt', 'w')
#파일 객체 정보 출력
print(file)
#데이터를 한번에 기록
file.write('Hello File')
file.write('\n\n')
#데이터를 줄 단위로 기록
lines = ['안녕하세요', '반갑습니다.', '파이썬입니다.']
file.write('\n'.join(lines))
'''
\n이 포함된 경우는
file.write('').join(lines)
file.writelines(lines)
가능
'''

file.close()
```

Hello File

안녕하세요
반갑습니다.
파이썬입니다.

파일 읽기

```
file = open('test.txt', 'r')
#한꺼번에 전부 읽기
content = file.read()
print(content)
file.close()
print("=====")
#줄단위로 읽기
file = open('test.txt', 'r')
for line in file:
    print(line)
file.close()
print("=====")
file = open('test.txt', 'r')
lines = file.readlines()
print(lines)
file.close()
```

Hello File

안녕하세요
반갑습니다.
파이썬입니다.

=====
Hello File

안녕하세요

반갑습니다.

파이썬입니다.

=====

['Hello File\n', '\n', '안녕하세요\n', '반갑
습니다.\n', '파이썬입니다.']

파일 읽기(csv)

❖test.csv

park,kim,choi

```
file = open('test.csv', 'r')
#한꺼번에 전부 읽기
content = file.read()
file.close()
print("=====")
ar = content.split(',')
li = list(ar)
for imsi in li:
    print(imsi)
```

park
kim
choi



바이너리 파일

- ❖ 바이너리 파일은 바이트 단위로 데이터를 읽고 쓰는 것
- ❖ 바이너리 파일은 문자열을 기록할 수 없고 byte 단위로만 기록해야 합니다.
- ❖ 문자열의 경우는 encode 함수를 이용해서 byte로 변형 할 수 있고 바이트를 문자열로 변환할 때는 decode 함수를 이용하면 됩니다.

```
f = open('test.bin', 'wb')  
f.write("안녕하세요".encode())  
f.close()  
f = open('test.bin', 'rb')  
byteAr = f.read()  
#print(byteAr)  
print(byteAr.decode())
```

문자열을 바이트 단위로 기록했으므로 읽을 때는 문자열로 변환해서 읽어야 합니다.

Serializable

❖객체를 파일에 저장하는 것

- 피클링 모듈이나 DBM 관련 모듈을 이용
- 피클링 모듈은 임의의 파이썬 객체를 저장하는 가장 일반화된 모듈
- 파일에 내용을 기록하는 경우
 - ✓`pickle.dump(출력할 객체, 파일객체)`
- 파일에서 내용을 읽어오는 경우
 - ✓`pickle.load(파일객체)` => 객체를 1개씩 읽기
 - ✓`Pickle.loads(파일객체)` => 객체 전체를 바이트 단위로 읽기



Serializable

```
class Dto:
    def setNum(self, num):
        self.num = num
    def setName(self, name):
        self.name = name
    def getNum(self):
        return self.num
    def getName(self):
        return self.name
    def toString(self):
        return "{번호:" + str(self.num) + ",이름:" +
self.name + "}"
```



Serializable

```
data1 = Dto()
data1.setNum(1)
data1.setName("park")
data2 = Dto()
data2.setNum(2)
data2.setName("kim")
li = [data1, data2]
import pickle
with open('test.txt', 'wb') as f:
    pickle.dump(li, f)

with open('test.txt', 'rb') as f:
    result = pickle.load(f)
    for temp in result:
        print(temp.toString())
```

{번호:1,이름:park}
{번호:2,이름:kim}

파일 압축

❖ zip 파일 압축은 zipfile 모듈 사용

- ✓ ZipFile 이라는 함수로 압축 객체를 생성하고 write 함수를 이용해서 하나씩 압축
- ✓ 압축 해제는 압축 파일을 가지고 ZipFile을 만든 후 extractall()을 호출하면 됩니다.

❖ tar 파일 압축은 tarfile 모듈 사용

- ✓ open 함수를 이용해서 압축 객체를 만든 후 add 함수를 이용해서 파일을 추가
- ✓ 압축 해제는 압축 파일 이름을 가지고 open 함수로 객체를 생성한 후 extractall()을 호출



파일 압축

```
import zipfile
filelist = ["c:\\test.txt"]
with zipfile.ZipFile('test.zip', 'w', compression=zipfile.ZIP_BZIP2) as myzip:
    for temp in filelist:
        myzip.write(temp)
zipfile.ZipFile('test.zip').extractall()
```

```
import tarfile
filelist = ["c:\\test.txt"]
with tarfile.open('test.tar.gz', 'w:gz') as mytar:
    for temp in filelist:
        mytar.add(temp)
tarfile.open('test.tar.gz').extractall()
```



파일 압축

Apache Web Server 에서 유효한 페이지 접근 횟수 출력

```
pageviews = 0

with open('로그파일 경로', 'r') as f:
    logs = f.readlines()
    for log in logs:
        log = log.split()
        status = log[8]
        if status == '200':
            pageviews += 1
print('총 페이지뷰: [%d]' %pageviews)
```



파일 압축

Apache Web Server 에서 ip 수 출력

```
visit_ip = []

with open('로그파일 경로', 'r') as f:
    logs = f.readlines()
    for log in logs:
        log = log.split()
        ip = log[0]
        if ip not in visit_ip:
            visit_ip.append(ip)

print('고유 방문자수: [%d]' %len(visit_ip))
```




파일 압축

Apache Web Server 에서 전체 트래픽 출력

```
total_service = 0

with open('로그파일 경로', 'r') as f:
    logs = f.readlines()
    for log in logs:
        log = log.split()
        servicebyte = log[9]
        if servicebyte.isdigit():
            total_service += int(servicebyte)
total_service /= 1024
print('총 서비스 용량: %dKB' %total_service)
```



파일 압축

Apache Web Server 에서 ip 별 트래픽 출력

```
services = {}

with open('로그파일경로', 'r') as f:
    logs = f.readlines()
    for log in logs:
        log = log.split()
        ip = log[0]
        servicebyte = log[9]
        if servicebyte.isdigit():
            servicebyte = int(servicebyte)
        else:
            servicebyte = 0

        if ip not in services:
            services[ip] = servicebyte
        else:
            services[ip] += servicebyte
```



파일 압축

```
ret = sorted(services.items(), key=lambda x: x[1], reverse=True)
```

```
print('사용자IP - 서비스용량')  
for ip, b in ret:  
    print('[%s] - [%d]' %(ip, b))
```



numpy에서의 저장과 열기

- ❖ `numpy.save(파일경로, 데이터)`: raw 데이터의 형태로 저장하는데 확장자는 `.npy`인데 입력하지 않으면 자동으로 삽입합니다.
- ❖ 읽을 때는 `load(파일경로)`: 저장할 객체 그대로 리턴
- ❖ `savez('파일명', 키=배열, 키=배열)`: 디셔너리 형식으로 데이터를 저장하는데 이 때는 확장자가 `npz`
- ❖ 위의 경우는 디셔너리 형식으로 저장되어 있으므로 데이터를 전부 읽은 후 ['키']를 이용해서 읽어야 합니다.
- ❖ `loadtxt('파일경로', delimiter='구분자')`를 이용해서 csv 형식의 파일도 읽어 낼 수 있습니다.



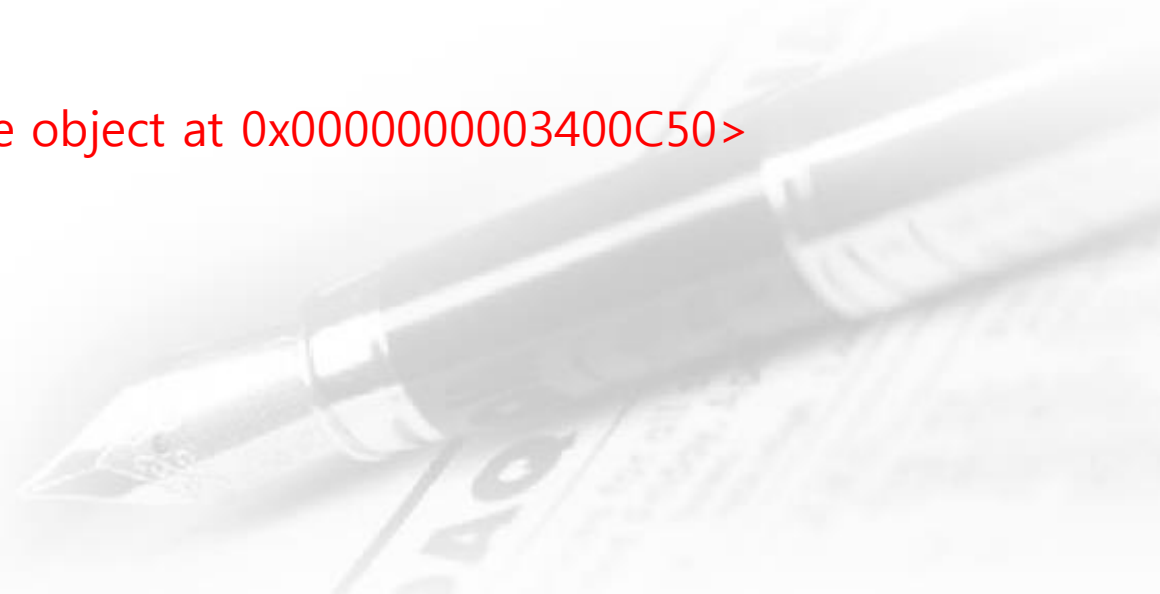
numpy에서의 저장과 열기

```
import numpy as np
ar = [100, 300, 200]
np.save('ar', ar)
br = np.load('ar.npy')
print(br)
cr = ar+br
np.savez('dic', a=ar, b=br, c=cr)
result = np.load('dic.npz')
print(result)
print(result['c'])
```

[100 300 200]

<numpy.lib.npyio.NpzFile object at 0x0000000003400C50>

[200 600 400]



numpy에서의 저장과 열기

```
import numpy as np
ar = np.arange(30)
np.savetxt('test.csv', ar)
br = np.loadtxt('test.csv', delimiter='\\t')
print(br)
```

```
[ 0.  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.]
```



Pandas

pandas

- ❖ 효과적인 데이터 분석을 위한 고수준의 자료구조와 데이터 분석 도구를 제공합니다. Pandas의 Series는 1차원 데이터를 다루는 데 효과적인 자료구조이며, DataFrame은 행과 열로 구성된 2차원 데이터를 다루는 데 효과적인 자료구조입니다.
- ❖ Series: 리스트와 비슷하고 어떤 면에서는 파이썬의 딕셔너리와 닮은 자료구조
 - ✓ 리스트를 가지고 생성할 수 있는데 기본적으로는 리스트처럼 정수 인덱스를 이용해서 순서대로 저장합니다.
 - ✓ values 속성을 호출하면 데이터의 배열이 리턴됩니다.
 - ✓ index 속성을 호출하면 인덱스의 배열이 리턴됩니다.
 - ✓ 각각의 데이터는 [인덱스]를 이용해서 접근이 가능합니다.
 - ✓ index 파라미터를 이용해서 파라미터를 직접 대입이 가능



pandas

```
from pandas import Series, DataFrame
price = Series([4000, 3000, 3500, 2000])
print(price)
print(price.index)
print(price.values)
print('=====')
good = Series([4000, 3000, 3500, 2000],
index=['apple', 'mellon', 'orange', 'kiwi'])
print(good)
print(good[0]) #순번을 이용해서 데이터 접근
print(good['apple']) #직접 입력한 인덱스를 이용해서
데이터 접근
```

```
0    4000
1    3000
2    3500
3    2000
dtype: int64
RangeIndex(start=0,
stop=4, step=1)
[4000 3000 3500 2000]
=====
=
apple    4000
mellon   3000
orange   3500
kiwi     2000
dtype: int64
4000
4000
```

pandas

❖ Series

- ✓ 인덱스를 리스트의 형태로 대입하면 인덱스에 해당하는 데이터를 리턴
- ✓ 인덱스로 조건을 입력하면 조건에 맞는 데이터만 리턴
- ✓ 정수 데이터와 산술 연산 가능
- ✓ numpy의 함수 사용 가능
- ✓ 연산을 할 때는 values의 값을 가지고 연산
- ✓ dict 객체로 생성 가능



pandas

```
from pandas import Series, DataFrame
import numpy as np
import pandas as pd
good = Series([4000, 3000, 3500, 2000],
index=['apple', 'apple', 'orange', 'kiwi'])
print(good[good>3000])
print("=====")
print(good+100)
print("=====")
print(np.sum(good))
print("=====")
keys = ['apple', 'apple', 'orange', 'kiwi'] #dict는 동일한
키를 저장할 수 없음
values = (4000, 3000, 3500, 2000)
dic = dict(zip(keys, values))
good = Series(dic)
print(good)
```

```
apple    4000
orange   3500
dtype: int64
```

```
=====
apple    4100
apple    3100
orange   3600
kiwi     2100
dtype: int64
```

```
=====
12500
=====
```

```
apple    3000
kiwi     2000
orange   3500
dtype: int64
```


pandas

❖ Series


- ✓ `pandas.isnull(Series객체)`: 데이터가 없는 경우는 True 있는 경우는 False로 value를 생성해서 Series 객체로 리턴
- ✓ `pandas.isnotnull(Series객체)`: 데이터가 있는 경우는 True 없는 경우는 False로 value를 생성해서 Series 객체로 리턴
- ✓ 매개변수 없이 Series객체가 호출해도 동일한 결과
- ✓ Series끼리의 산술 연산은 인덱스가 동일한 데이터끼리 연산
- ✓ 한쪽에만 존재하거나 None 데이터와의 연산은 NaN
- ✓ `name` 속성을 이용해서 데이터에 이름 부여 가능
- ✓ `index.name` 속성을 이용해서 인덱스에 이름 부여 가능



pandas

```
from pandas import Series, DataFrame
import numpy as np
import pandas as pd
good1 = Series([4000, 3500, None, 2000],
index=['apple', 'mango','orange', 'kiwi'])
good2 = Series([3000, 3000, 3500, 2000],
index=['apple', 'banana','mango', 'kiwi'])
print(pd.isnull(good1))
print(good1+good2)
```

```
apple    False
mango    False
orange    True
kiwi     False
dtype: bool
apple    7000.0
banana   NaN
kiwi     4000.0
mango    7000.0
orange   NaN
dtype: float64
```



pandas

❖ DataFrame

✓ DataFrame은 여러 개의 칼럼(Column)으로 구성된 2차원 형태의 자료구조

| 일자별 주가 | | 일봉차트 | | | | | | 자세히▶ |
|----------|--------|--------|--------|--------|------|--------|---------|------|
| 일자 | 시가 | 고가 | 저가 | 종가 | 전일비 | 등락률 | 거래량 | |
| 16,02,29 | 11,650 | 12,100 | 11,600 | 11,900 | ▲300 | +2,59% | 225,844 | |
| 16,02,26 | 11,100 | 11,800 | 11,050 | 11,600 | ▲600 | +5,45% | 385,241 | |
| 16,02,25 | 11,200 | 11,200 | 10,900 | 11,000 | ▼100 | -0,90% | 161,214 | |
| 16,02,24 | 11,100 | 11,100 | 10,950 | 11,100 | ▲50 | +0,45% | 77,201 | |
| 16,02,23 | 11,000 | 11,150 | 10,900 | 11,050 | ▲100 | +0,91% | 113,131 | |
| 16,02,22 | 10,950 | 11,050 | 10,850 | 10,950 | ▼100 | -0,90% | 138,387 | |
| 16,02,19 | 10,950 | 11,100 | 10,800 | 11,050 | -0 | 0,00% | 76,105 | |
| 16,02,18 | 11,050 | 11,200 | 10,950 | 11,050 | ▲250 | +2,31% | 83,611 | |
| 16,02,17 | 11,150 | 11,300 | 10,800 | 10,800 | ▼350 | -3,14% | 189,480 | |
| 16,02,16 | 10,950 | 11,200 | 10,850 | 11,150 | ▲300 | +2,76% | 133,359 | |

pandas

❖ DataFrame

- ✓ 디셔너리의 배열과 유사
- ✓ 일반적으로 디셔너리를 이용해서 생성합니다.
- ✓ 각 키에 리스트가 할당된 디셔너리를 변환할 수 있습니다.
- ✓ 디셔너리의 키는 정렬되서 배치됩니다.
- ✓ 정렬순서를 변경하고자 하면 columns 매개변수에 순서를 리스트로 대입하면 됩니다.
- ✓ DataFrame의 각 컬럼의 데이터는 사전처럼['컬럼이름'] 또는 .컬럼이름으로 접근하면 Series 객체로 리턴됩니다.
- ✓ 특정 행에 접근하기 위해서는 ix[인덱스]를 이용하면 되는데 Series 객체로 데이터는 리턴



pandas

❖ DataFrame

✓ 생성자에서 사용 가능한 입력 데이터

- 2차원 ndarray
- 리스트, 튜플, dict, Series의 dict
- dict, Series의 list
- 리스트, 튜플의 리스트



pandas

```
from pandas import Series, DataFrame

items = {'code': [1,2,3,4,5,6],
         'name': ['apple','watermelon','oriental melon', 'banana', 'lemon', 'mango'],
         'manufacture': ['korea', 'korea', 'korea', 'philippines', 'korea', 'taiwan'],
         'price': [1500, 15000, 1000, 500, 1500, 700]}

data = DataFrame(items)
print(data)
```

| | code | manufacture | name | price |
|---|------|-------------|----------------|-------|
| 0 | 1 | korea | apple | 1500 |
| 1 | 2 | korea | watermelon | 15000 |
| 2 | 3 | korea | oriental melon | 1000 |
| 3 | 4 | philippines | banana | 500 |
| 4 | 5 | korea | lemon | 1500 |
| 5 | 6 | taiwan | mango | 700 |

A fountain pen is shown in the background, resting on a document that contains a table. The table has columns for 'code', 'manufacture', 'name', and 'price', with data rows corresponding to the pandas DataFrame output shown in the foreground. The pen is a silver-colored fountain pen with a black grip section.

pandas

```
from pandas import Series, DataFrame
```

```
items = {'code': [1,2,3,4,5,6],  
         'name': ['apple','watermelon','oriental melon', 'banana', 'lemon', 'mango'],  
         'manufacture': ['korea', 'korea', 'korea','philippines','korea', 'taiwan'],  
         'price':[1500, 15000,1000,500,1500,700]}
```

```
data = DataFrame(items, columns=['code', 'name', 'manufacture', 'price'])
```

```
print(data['name']) #name 컬럼의 모든 값을 가져오기
```

```
print('=====')
```

```
print(data.ix[0]) #0번째 데이터를 가져오기
```

```
print('=====')
```

```
print(data['name'][0])#0번째 데이터의 name 값 가져오기
```



pandas

```
0      apple
1  watermelon
2  oriental melon
3      banana
4      lemon
5      mango
Name: name, dtype: object
```

```
=====
```

```
code      1
name      apple
manufacture korea
price     1500
Name: 0, dtype: object
```

```
=====
```

```
apple
```



pandas

❖ DataFrame

- ✓ 기본적으로 인덱스는 0부터 시작하는 숫자이지만 index 속성을 이용해서 생성할 때 index 지정이 가능하며 나중에 지정도 가능
- ✓ index 속성은 index의 값들을 리턴하고 values는 데이터의 모임을 2차원 배열의 형태로 리턴합니다.
- ✓ 특정 컬럼에 데이터를 삽입하거나 변경하는 것 가능
- ✓ 특정 컬럼을 삭제할 때는 del 프레임객체['컬럼이름']
- ✓ 컬럼의 데이터 전체를 변경할 때는 Series 객체 또는 리스트 및 튜플을 이용해서 가능한데 리스트나 튜플의 길이가 DataFrame의 행 길이와 동일한 크기 이어야 합니다.
- ✓ Series 객체를 대입할 때는 index에 해당하는 데이터가 수정되는데 없는 index에는 NaN 값이 대입됩니다.
- ✓ 기존 객체에 없는 컬럼의 이름을 이용해서 대입하면 컬럼이 추가됩니다.
- ✓ T 속성을 이용해서 index와 column을 변경한 객체를 리턴받을 수 있습니다.

pandas

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np

items = {'code': [1,2,3,4,5,6],
        'name': ['apple','watermelon','oriental melon', 'banana', 'lemon', 'mango'],
        'manufacture': ['korea', 'korea', 'korea','philippines','korea', 'taiwan'],
        'price':[1500, 15000,1000,500,1500,700]}

data = DataFrame(items, columns=['code', 'name', 'manufacture', 'price'])
print(data.index)
data.index = np.arange(1,7,1)
print(data.index)
data.price = [1500, 10000, 500, 1200, 300, 5000]
print(data)
data.price = Series([3000, 20000, 300, 1000 ], index=[1,2,3,4])
print(data)
```

pandas

```
RangeIndex(start=0, stop=6, step=1)
```

```
Int64Index([1, 2, 3, 4, 5, 6], dtype='int64')
```

| | code | name | manufacture | price |
|---|------|----------------|-------------|-------|
| 1 | 1 | apple | korea | 1500 |
| 2 | 2 | watermelon | korea | 10000 |
| 3 | 3 | oriental melon | korea | 500 |
| 4 | 4 | banana | philippines | 1200 |
| 5 | 5 | lemon | korea | 300 |
| 6 | 6 | mango | taiwan | 5000 |

| | code | name | manufacture | price |
|---|------|----------------|-------------|---------|
| 1 | 1 | apple | korea | 3000.0 |
| 2 | 2 | watermelon | korea | 20000.0 |
| 3 | 3 | oriental melon | korea | 300.0 |
| 4 | 4 | banana | philippines | 1000.0 |
| 5 | 5 | lemon | korea | NaN |
| 6 | 6 | mango | taiwan | NaN |

pandas

❖ DataFrame

- ✓ 중첩 dict 도 DataFrame으로 생성 가능
- ✓ 이 때 외부에 있는 키가 컬럼의 이름이 되고 내부에 있는 키가 인덱스가 됩니다.
- ✓ 인덱스는 index 속성을 이용해서 변경이 가능합니다.
- ✓ index 나 columns의 name 속성을 이용해서 index나 column 들에 이름을 부여하는 것이 가능합니다.



pandas

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np
```

```
items = {'1':{'name':'apple', 'manufacture':'korea', 'price':1500},
        '2':{'name': 'watermelon', 'manufacture': 'korea', 'price': 15000},
        '3':{'name': 'oriental melon', 'manufacture': 'korea', 'price': 1000},
        '4':{'name': 'banana', 'manufacture': 'philippines', 'price': 500},
        '5':{'name': 'lemon', 'manufacture': 'korea', 'price': 1500},
        '6':{'name': 'mango', 'manufacture': 'korea', 'price': 700}}
```

```
data = DataFrame(items)
print(data)
data = data.T
print(data)
```



pandas

| 1 | 2 | 3 | 4 | 5 | 6 | | | |
|-------------|-------|------------|----------|-------|-------------------|--------|-------|-------|
| manufacture | korea | | korea | | korea philippines | korea | korea | |
| name | apple | watermelon | oriental | melon | | banana | lemon | mango |
| price | 1500 | 15000 | | 1000 | | 500 | 1500 | 700 |

| | manufacture | | name | price |
|---|-------------|--|----------------|-------|
| 1 | korea | | apple | 1500 |
| 2 | korea | | watermelon | 15000 |
| 3 | korea | | oriental melon | 1000 |
| 4 | philippines | | banana | 500 |
| 5 | korea | | lemon | 1500 |
| 6 | korea | | mango | 700 |



pandas

❖ Series, DataFrame의 인덱스 재구성

- ✓ `reindex` 속성을 이용해서 `index`를 재배치하거나 추가하거나 삭제하는 것이 가능
- ✓ 인덱스 자체의 값을 변경하는 것은 안됩니다.
- ✓ 인덱스를 변경할 때 `fill_value` 매개변수에 값을 대입하면 누락된 인덱스에는 값을 대입됩니다.
- ✓ `method` 매개변수에 `ffill` 또는 `bfill`을 대입하면 이전 데이터나 뒤의 데이터를 대입합니다
- ✓ DataFrame의 경우는 `index` 와 `columns` 매개변수를 이용해서 `index`와 `column` 을 재배치 할 수 있습니다.



pandas

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np

items = {'1':{'name':'apple', 'manufacture':'korea', 'price':1500},
        '2':{'name': 'watermelon', 'manufacture': 'korea', 'price': 15000},
        '3':{'name': 'oriental melon', 'manufacture': 'korea', 'price': 1000},
        '4':{'name': 'banana', 'manufacture': 'philippines', 'price': 500},
        '5':{'name': 'lemon', 'manufacture': 'korea', 'price': 1500},
        '6':{'name': 'mango', 'manufacture': 'korea', 'price': 700}}

data = DataFrame(items)
data = data.T
data = data.reindex(['1','2','3','4','5','7']) # 재 인덱싱 - 6번은 없어지고 7번은 추가되는데
7번의 데이터는 NaN
print(data);
print("=====")
```


pandas

```
data = data.reindex(['1','2','3','4','5','7'], fill_value=0) # 재 인덱싱 - 6번은 없어지고 7번  
은 추가되는데 모든 값은 0  
print(data);  
print("=====  
data = data.reindex(['1','2','3','4','5','7'], method='ffill', limit=2) # 재 인덱싱 - 6번은 없  
어지고 7번은 추가되는데 값은 이전데이터와 동일  
#print(data);
```



pandas

❖ Series, DataFrame의 데이터 삭제

- ✓ 인덱스 이름을 이용해서 행을 삭제할 수 있는 이 때는 drop 메서드에 인덱스 또는 인덱스의 리스트를 넘겨주면 됩니다.
- ✓ 열을 삭제할 때는 열이름 또는 열이름의 리스트를 넘겨주소 axis 파라미터에 1을 대입해 주면 됩니다.



pandas

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np

items = {'1':{'name':'apple', 'manufacture':'korea', 'price':1500},
        '2':{'name': 'watermelon', 'manufacture': 'korea', 'price': 15000},
        '3':{'name': 'oriental melon', 'manufacture': 'korea', 'price': 1000},
        '4':{'name': 'banana', 'manufacture': 'philippines', 'price': 500},
        '5':{'name': 'lemon', 'manufacture': 'korea', 'price': 1500},
        '6':{'name': 'mango', 'manufacture': 'korea', 'price': 700}}

data = DataFrame(items)
data = data.T
data = data.drop("1") #인덱스가 1인 행 삭제
print(data)
print("=====")
data = data.drop(["3","5"]) #인덱스가 3, 5 인 데이터 삭제
print(data)
print("=====")
data = data.drop("price", axis=1) #price 컬럼 삭제
print(data)
```

pandas

| | manufacture | name | price |
|---|-------------|----------------|-------|
| 2 | korea | watermelon | 15000 |
| 3 | korea | oriental melon | 1000 |
| 4 | philippines | banana | 500 |
| 5 | korea | lemon | 1500 |
| 6 | korea | mango | 700 |

=====

| | manufacture | name | price |
|---|-------------|------------|-------|
| 2 | korea | watermelon | 15000 |
| 4 | philippines | banana | 500 |
| 6 | korea | mango | 700 |

=====

| | manufacture | name |
|---|-------------|------------|
| 2 | korea | watermelon |
| 4 | philippines | banana |
| 6 | korea | mango |



pandas

- ❖ Series, DataFrame의 색인 및 선택 또는 필터링
 - ✓ 색인에 컬럼 이름을 대입해서 조회가능
 - ✓ 컬럼 이름의 범위를 대입해서 데이터를 컬럼 단위로 추출 가능
 - ✓ 컬럼 이름을 이용한 조건을 대입해서 컬럼 단위 추출 가능
 - ✓ ix[행번호 또는 인덱스]를 대입하면 행번호 또는 인덱스에 해당하는 데이터를 리턴합니다.
 - ✓ ix[행번호 또는 인덱스, [컬럼이름]]를 대입하면 행번호 또는 인덱스에 해당하는 데이터의 컬럼 값만 리턴합니다.
 - ✓ 행번호 또는 인덱스 자리에 컬럼을 이용한 조건 입력 가능



pandas

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np
```

```
items = {'1':{'name':'apple', 'manufacture':'korea', 'price':1500},
        '2': {'name': 'watermelon', 'manufacture': 'korea', 'price': 15000},
        '3': {'name': 'oriental melon', 'manufacture': 'korea', 'price': 1000},
        '4': {'name': 'banana', 'manufacture': 'philippines', 'price': 500},
        '5': {'name': 'lemon', 'manufacture': 'korea', 'price': 1500},
        '6': {'name': 'mango', 'manufacture': 'korea', 'price': 700}}
```

```
data = DataFrame(items)
print(data['1']) #key가 1번인 데이터
print('=====')
print(data['1':'3']) #key가 1:3번인 데이터
print('=====')
print(data[['1','3']]) #key가 1,3번인 데이터
```

pandas

```
manufacture  korea  
name         apple  
price        1500  
Name: 1, dtype: object
```

```
=====
```

```
Empty DataFrame  
Columns: [1, 2, 3, 4, 5, 6]  
Index: []
```

```
=====
```

| | 1 | 3 |
|-------------|-------|----------------|
| manufacture | korea | korea |
| name | apple | oriental melon |
| price | 1500 | 1000 |



pandas

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np

items = {'1':{'name':'apple', 'manufacture':'korea', 'price':1500},
        '2':{'name': 'watermelon', 'manufacture': 'korea', 'price': 15000},
        '3':{'name': 'oriental melon', 'manufacture': 'korea', 'price': 1000},
        '4':{'name': 'banana', 'manufacture': 'philippines', 'price': 500},
        '5':{'name': 'lemon', 'manufacture': 'korea', 'price': 1500},
        '6':{'name': 'mango', 'manufacture': 'korea', 'price': 700}}

data = DataFrame(items)
data = data.T
print(data[0:3]) #0-3행까지 추출
print('=====')
print(data['price'] > 1000) #price 컬럼의 값이 1000이 넘는지 확인
print('=====')
print(data[data['price'] > 1000]) #price 컬럼의 값이 1000이 넘는 행 만 출력
```


pandas

```
manufacture      name price
1    korea      apple  1500
2    korea  watermelon 15000
3    korea  oriental melon 1000
```

=====

```
1    True
2    True
3    False
4    False
5    True
6    False
```

Name: price, dtype: bool

=====

```
manufacture      name price
1    korea      apple  1500
2    korea  watermelon 15000
5    korea      lemon  1500
```



pandas

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np
items = {'1':{'name':'apple', 'manufacture':'korea', 'price':1500},
        '2': {'name': 'watermelon', 'manufacture': 'korea', 'price': 15000},
        '3': {'name': 'oriental melon', 'manufacture': 'korea', 'price': 1000},
        '4': {'name': 'banana', 'manufacture': 'philippines', 'price': 500},
        '5': {'name': 'lemon', 'manufacture': 'korea', 'price': 1500},
        '6': {'name': 'mango', 'manufacture': 'korea', 'price': 700}}
data = DataFrame(items)
data = data.T
print(data.ix[0])#0번행의 데이터를 출력
print('=====')
print(data.ix[0, ['name']])#0번행의 name 데이터를 출력
print('=====')
print(data.ix[0, ['name', 'price']])#0번행의 name과 price 데이터를 출력
print('=====')
print(data.ix[:3, ['name', 'price']])#3번행 까지의 name과 price 데이터를 출력
print('=====')
print(data.ix[data.price>1000, ['name', 'price']])#price가 1000이 넘는 데이터의 name과
price 출력 name과 price 데이터를 출력
```

pandas

manufacture korea

name apple

price 1500

Name: 1, dtype: object

=====

name apple

Name: 1, dtype: object

=====

name apple

price 1500

Name: 1, dtype: object

=====

name price

1 apple 1500

2 watermelon 15000

3 oriental melon 1000

=====

name price

1 apple 1500

2 watermelon 15000

5 lemon 1500



pandas

❖ DataFrame의 연산

- ✓ 산술 연산은 동일한 인덱스 값을 찾아서 연산을 수행합니다.
- ✓ 어느 한쪽에만 존재하는 인덱스의 연산은 NaN이 됩니다.
- ✓ 산술연산을 add, sub, div, mul 메서드를 이용해서 수행할 수 있는데 이 때 fill_value를 이용해서 한쪽에만 존재하는 인덱스에 기본 값을 삽입할 수 있습니다.
- ✓ Series와의 산술 연산 가능
 - Series의 인덱스를 DataFrame의 컬럼 이름과 매핑해서 연산을 수행하는데 동일한 값이 없으면 NaN
 - DataFrame의 모든 행에 대해서 브로드캐스팅 연산
 - 행단위로 연산을 하고자 하는 경우는 add, sub, div, mul 메서드를 호출해서 첫번째 매개변수로 Series 객체를 대입하고 axis에 0을 대입



pandas

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np

items1 = {'1':{'price':1500},
          '2':{'price': 15000},
          '3':{'price': 1000}}
items2 = {'1':{'price':1500},
          '2':{'price': 15000},
          '4':{'price': 1000}}

data1 = DataFrame(items1)
data1 = data1.T

data2 = DataFrame(items2)
data2 = data2.T

print(data1 + data2)
print("=====")
print(data1.add(data2, fill_value=0))
```

pandas

price

```
1  3000.0
2 30000.0
3    NaN
4    NaN
```

=====

price

```
1  3000.0
2 30000.0
3  1000.0
4  1000.0
```



pandas

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np

items1 = {'1':{'price':1500},
          '2': {'price': 15000},
          '3': {'price': 1000}}
items2 = Series([300, 200, 100], index=["1", "2", "4"])

data1 = DataFrame(items1)
print(data1+items2)
print('=====')
data1 = data1.T
print(data1.add(items2, axis=0))
```



pandas

```
1      2  3  4  
price 1800.0 15200.0 NaN NaN
```

```
=====
```

```
      price  
1  1800.0  
2 15200.0  
3     NaN  
4     NaN
```



pandas

❖ DataFrame에 함수 적용

- ✓ apply 메서드를 이용하면 행이나 열 단위로 함수를 적용할 수 있습니다.
- ✓ apply()의 첫번째 매개변수는 함수이며 axis의 값을 생략하면 컬럼 단위로 함수를 수행하고 1을 대입하면 행 단위로 함수를 수행합니다.
- ✓ 데이터의 각각에 함수를 적용하고자 하는 경우는 applymap을 이용
- ✓ Series에 적용하고자 하는 경우는 map을 이용



pandas

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np

def func(x):
    return x.sum()
#f = lambda x: x.sum()

items = {'apple':{'count':10,'price':1500},
        'banana': {'count':5, 'price': 15000},
        'melon': { 'count':7,'price': 1000},
        'kiwi': {'count':20,'price': 500},
        'mango': {'count':30,'price': 1500},
        'orange': { 'count':4,'price': 700}}

data = DataFrame(items)
data = data.T
print(data.apply(func))
print("=====")
print(data.apply(func, axis=1))
```

pandas

```
count    76  
price    20200  
dtype: int64  
=====
```

| | |
|--------|-------|
| apple | 1510 |
| banana | 15005 |
| kiwi | 520 |
| mango | 1530 |
| melon | 1007 |
| orange | 704 |

```
dtype: int64
```



pandas

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np

def func(x):
    return x+10
#f = lambda x: x+10

items = {'apple':{'count':10,'price':1500},
        'banana': {'count':5, 'price': 15000},
        'melon': { 'count':7,'price': 1000},
        'kiwi': {'count':20,'price': 500},
        'mango': {'count':30,'price': 1500},
        'orange': { 'count':4,'price': 700}}

data = DataFrame(items)
data = data.T
print(data.applymap(func))
print("=====")
print(data["count"].map(func))
```

pandas

```
count price  
apple    20  1510  
banana   15 15010  
kiwi     30   510  
mango    40  1510  
melon    17  1010  
orange   14   710
```

=====

```
apple    20  
banana   15  
kiwi     30  
mango    40  
melon    17  
orange   14
```

```
Name: count, dtype: int64
```



pandas

❖ DataFrame의 정렬과 순위

✓ 정렬

- `sort_index()`를 호출하면 인덱스가 정렬하며 `axis`에 1을 대입하면 컬럼의 이름이 정렬
- 기본은 오름차순 정렬이며 내림차순 정렬을 하고자 하는 경우에는 `ascending`의 값을 `False`로 대입
- Series 객체의 정렬은 `sort_values()`
- DataFrame에서 특정 컬럼을 기준으로 정렬을 하고자 하면 `sort_values` 메서드에 `by` 매개변수로 컬럼의 이름이나 컬럼의 이름 리스트를 대입하면 됩니다.
- 데이터가 `NaN` 인 경우는 정렬을 하는 경우 가장 마지막에 위치

✓ 순위

- 데이터의 순위는 `rank()`를 이용하는데 기본적으로는 오름차순으로 순위를 설정
- `ascending`의 값을 `False`로 대입하면 내림차순 순위
- `axis` 매개변수를 이용하면 축을 설정할 수 있으며 기본적으로는 컬럼 단위이며 `axis`에 1을 대입하면 행 단위 순위
- 동점은 순위의 평균을 출력하는데 `method` 매개변수에 `max`를 대입하면 큰 순위를 출력하고 `min`을 대입하면 작은 순위 `first`를 대입하면 먼저 등장한 데이터가 작은 순위를 갖습니다

pandas

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np

items = {'apple':{'count':10,'price':1500},
        'banana': {'count':5, 'price': 15000},
        'melon': { 'count':7,'price': 1000},
        'kiwi': {'count':20,'price': 500},
        'mango': {'count':30,'price': 1500},
        'orange': { 'count':4,'price': 700}}
data = DataFrame(items)
data = data.T
print(data.sort_values(ascending=[False,True], by=["price", "count"]))
```



pandas

| | count | price |
|--------|-------|-------|
| banana | 5 | 15000 |
| apple | 10 | 1500 |
| mango | 30 | 1500 |
| melon | 7 | 1000 |
| orange | 4 | 700 |
| kiwi | 20 | 500 |



pandas

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np
items = {'apple':{'count':10,'price':1500},
        'banana': {'count':5, 'price': 15000},
        'melon': { 'count':7,'price': 1000},
        'kiwi': {'count':20,'price': 500},
        'mango': {'count':30,'price': 1500},
        'orange': { 'count':4,'price': 700}}
data = DataFrame(items)
data = data.T
print(data.rank())
print("=====")
print(data.rank(ascending=False, method='min'))
print("=====")
```



pandas

```
count price  
apple 4.0 4.5  
banana 2.0 6.0  
kiwi 5.0 1.0  
mango 6.0 4.5  
melon 3.0 3.0  
orange 1.0 2.0
```

=====

```
count price  
apple 3.0 2.0  
banana 5.0 1.0  
kiwi 2.0 6.0  
mango 1.0 2.0  
melon 4.0 4.0  
orange 6.0 5.0
```

=====



pandas

❖ 통계 메서드

- axis는 계산 방향으로 0은 행 단위이고 1은 열 단위
- skipna는 NaN값이 있는 경우 제외여부로 True로 설정하면 제외하고 False이면 포함
- count, min, max, sum, mean, median, var, std
- argmin(최소값 위치), argmax, idxmin(최소값 색인), idxmax, quantile(분위수)
- describe(요약)
- cumsum(누적합), cummin, cummax, cumprod
- diff(산술 차)
- pct_change
- unique(): 동일한 값을 제외한 배열 리턴 - Series에만 사용
- value_counts(): 도수를 리턴하는데 기본적으로 내림차순 정렬을 수행하며 sort 속성에 False를 대입하면 정렬하지 않습니다. - Series에만 사용

pandas

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np
items = {'apple':{'count':10,'price':1500},
        'banana': {'count':5, 'price': 15000},
        'melon': { 'count':7,'price': 1000},
        'kiwi': {'count':20,'price': 500},
        'mango': {'count':30,'price': 1500},
        'orange': { 'count':4,'price': 700}}
data = DataFrame(items)
data = data.T
print(data.describe())
print("=====")
```



pandas

| | count | price |
|-------|-----------|--------------|
| count | 6.000000 | 6.000000 |
| mean | 12.666667 | 3366.666667 |
| std | 10.269697 | 5713.726163 |
| min | 4.000000 | 500.000000 |
| 25% | 5.500000 | 775.000000 |
| 50% | 8.500000 | 1250.000000 |
| 75% | 17.500000 | 1500.000000 |
| max | 30.000000 | 15000.000000 |



pandas

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np
stocks = {'2017-02-19': {'다음': 50300, '네이버': 51100},
          '2017-02-22': {'다음': 50300, '네이버': 50800},
          '2016-02-23': {'다음': 50800, '네이버': 53000}}
data = DataFrame(stocks)
data = data.T
print(data.diff())
print("=====")
print(data.pct_change())
print("=====")
```



pandas

네이버 다음

| | | |
|------------|---------|--------|
| 2016-02-23 | NaN | NaN |
| 2017-02-19 | -1900.0 | -500.0 |
| 2017-02-22 | -300.0 | 0.0 |

=====

네이버 다음

| | | |
|------------|-----------|-----------|
| 2016-02-23 | NaN | NaN |
| 2017-02-19 | -0.035849 | -0.009843 |
| 2017-02-22 | -0.005871 | 0.000000 |

=====



pandas

❖ DataFrame의 NaN 처리

- `isnull()`: NaN 이나 None 인 True 그렇지 않은 경우는 False 리턴
- `notnull()`: `isnull()`의 반대
- `dropna()`: NaN 인 값을 소유한 행 제외하는데 `how` 매개변수에 all을 대입하면 컬럼의 모든 값이 NaN 인 경우만 제외하며 `thresh` 매개변수에 정수를 대입하면 그 정수 값이 상의 값을 소유한 컬럼만 리턴
- `fillna()`: NaN을 소유한 데이터의 값을 설정할 때 사용하는 메서드로 특정한 값으로 변경할 수 있고 `method` 매개변수를 이용해서 이전 값이나 이후 값으로 채울 수 있으며 `limit`를 이용해서 채울 개수를 지정할 수 있습니다.



pandas

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np
stocks = {'2017-02-19':{'다음':50300,'네이버': 51100, "넥슨":None, "NC":None},
"2017-02-22":{'다음':50300, '네이버': 50800, "넥슨":35000, "NC":None},
'2017-02-23':{'다음':50800,'네이버': 53000, "넥슨":37000, "NC":8000}}
data = DataFrame(stocks)
data = data.T
print(data.dropna())
print("=====")
print(data.dropna(how='all'))
print("=====")
print(data.diff().fillna(0))
print("=====")
```



pandas

```
NC    네이버    넥슨    다음
2017-02-23  8000.0  53000.0  37000.0  50800.0
```

```
=====
```

```
      NC    네이버    넥슨    다음
2017-02-19   NaN  51100.0   NaN  50300.0
2017-02-22   NaN  50800.0  35000.0  50300.0
2017-02-23  8000.0  53000.0  37000.0  50800.0
```

```
=====
```

```
      NC    네이버    넥슨    다음
2017-02-19  0.0    0.0    0.0    0.0
2017-02-22  0.0  -300.0    0.0    0.0
2017-02-23  0.0  2200.0  2000.0  500.0
```

```
=====
```



pandas

❖ DataFrame 상관관계

- `corr()`을 이용하면 상관관계를 알아볼 수 있습니다.
- `cov()`는 공분산
- Series 사이의 상관관계나 공분산을 알고자 할 때는 Series 객체를 매개변수로 넘겨주면 됩니다.
- DataFrame이 호출하면 모든 상관관계나 공분산을 리턴
- DataFrame에서 `corrwith` 메서드를 이용하면 Series 나 DataFrame 과의 상관관계를 리턴합니다.



pandas

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np
stocks = {'2017-02-19':{'다음':50300,"네이버": 51100, "넥슨":32000, "NC":4000},
"2017-02-22":{'다음':50300, '네이버': 50800, "넥슨":35000, "NC":6500},
'2017-02-23':{'다음':50800,'네이버': 50700, "넥슨":37000, "NC":8000}}
data = DataFrame(stocks)
data = data.T
d = data.pct_change().fillna(0)
print(d)
print('=====')
print('넥슨과 네이버', d.넥슨.corr(d.네이버))
print('=====')
print('넥슨과 NC',d.넥슨.corr(d.NC))
print('=====')
print(d.corr())
print('=====')
print(d.corrwith(d.NC))
```

pandas

NC 네이버 넥슨 다음

```
2017-02-19 0.000000 0.000000 0.000000 0.000000
2017-02-22 0.625000 -0.005871 0.093750 0.000000
2017-02-23 0.230769 -0.001969 0.057143 0.00994
```

=====

넥슨과 네이버 -0.951188402706

=====

넥슨과 NC 0.962244003854

=====

NC 네이버 넥슨 다음

```
NC 1.000000 -0.999276 0.962244 -0.149307
네이버 -0.999276 1.000000 -0.951188 0.186829
넥슨 0.962244 -0.951188 1.000000 0.125468
다음 -0.149307 0.186829 0.125468 1.000000
```

=====

```
NC 1.000000
네이버 -0.999276
넥슨 0.962244
다음 -0.149307
```

dtype: float64

pandas

❖ DataFrame 계층적 색인

- index나 컬럼이 2 level 이상으로 이루어진 경우
- 그룹화 연산을 할 때 유용
- 컬럼의 값들을 가져오는 방법은 이전과 동일
- 집계함수를 이용할 때 level에 인덱스나 컬럼의 이름을 대입하고 axis에 축 방향을 대입하면 그 레벨에 맞는 집계를 구하는 것이 가능



pandas

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np
li = [50300, 51100, 32000, 4000, 50300, 50800, 35000, 6500, 50800,
50700, 37000, 8000, 51800, 50500, 37500, 8200]
ar = np.array(li)
ar = ar.reshape(4,4)
stocks = DataFrame(ar, index=['다음', '네이버', '넥슨', 'NC'],
                    columns=[['3월', '3월', '4월', '4월'], ['11일', '12일', '11일', '12일']])
print(stocks)
print("=====")
print(stocks['3월']) #3월의 데이터 가져오기
print("=====")
print(stocks['3월']['12일'])#3월 11일의 데이터 가져오기
print("=====")
print(stocks.ix['다음'])#다음의 데이터 가져오기
```

pandas

3월 4월

11일 12일 11일 12일

다음 50300 51100 32000 4000

네이버 50300 50800 35000 6500

넥슨 50800 50700 37000 8000

NC 51800 50500 37500 8200

=====

11일 12일

다음 50300 51100

네이버 50300 50800

넥슨 50800 50700

NC 51800 50500

=====

다음 51100

네이버 50800

넥슨 50700

NC 50500

Name: 12일, dtype: int32

=====

3월 11일 50300

12일 51100

4월 11일 32000

12일 4000



pandas

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np
li = [50300, 51100, 32000, 4000, 50300, 50800, 35000, 6500, 50800,
50700, 37000, 8000, 51800, 50500, 37500, 8200]
ar = np.array(li)
ar = ar.reshape(4,4)
stocks = DataFrame(ar, index=['다음', '네이버', '넥슨', 'NC'],
                    columns=[['3월', '3월', '4월', '4월'], ['11일', '12일', '11일', '12일']])

print("=====")
print(stocks)
stocks.columns.names=['월', '일']
print(stocks.sum(level='월', axis=1))#월 별 합계
print("=====")
print(stocks.sum(level='일', axis=1))#일 별 합계
print("=====")
```

pandas

=====

3월 4월

11일 12일 11일 12일

다음 50300 51100 32000 4000

네이버 50300 50800 35000 6500

넥슨 50800 50700 37000 8000

NC 51800 50500 37500 8200

월 3월 4월

다음 101400 36000

네이버 101100 41500

넥슨 101500 45000

NC 102300 45700

=====

일 11일 12일

다음 82300 55100

네이버 85300 57300

넥슨 87800 58700

NC 89300 58700

=====

