

```
1 Lab. NumPy 사용하기
2
3 1. Tool
4 1)Microsoft Visual Studio Code
5 2)Jupyter Notebook
6 3)Google Colab
7   #Google Colab 사용방법
8   - 방향키 ↑↓: 셀 간 이동
9   - Enter: 편집모드
10  - Ctrl + Enter: 셀 실행
11  - Shift + Enter: 셀 실행 + 다음 셀 선택
12  - Ctrl + M D: 셀 삭제
13  - Ctrl + M K: 셀 위로 이동
14  - Ctrl + M J: 셀 아래로 이동
15
16
17
18 2. NumPy의 존재 이유
19 1)NumPy는 Pandas, Scikit-learn, Tensorflow등 데이터 사이언스 분야에서 사용되는 라이브러리들의 토대
20   가 되는 라이브러리.
21 2)NumPy 그 자체로는 높은 수준의 데이터 분석 기능을 제공하지 않지만 NumPy를 활용해 데이터를 Python상
22   에서 표현하고 다룰 줄 알아야만 데이터 분석이라는 그 이후 단계로 나아갈 수 있다.
23
24 3. NumPy 장점
25 1)코어 부분이 C로 구현되어 동일한 연산을 하더라도 Python에 비해 속도가 빠름
26 2)라이브러리에 구현되어있는 함수들을 활용해 짧고 간결한 코드 작성 가능
27 3)효율적인 메모리 사용이 가능하도록 구현됨
28 4)ndarray가 list보다 빠른 이유
29   -Image 참조
30   https://image.slidesharecdn.com/numpy20160519-160516164831/95/numpy-8-638.jp
31   g
32 5)Python list가 느린 이유
33   -Python list는 결국 포인터의 배열
34   -경우에 따라서 각각 객체가 메모리 여기저기 흩어져 있음
35   -그러므로 캐시 활용이 어려움
36 6)NumPy ndarray가 빠른 이유
37   -ndarray는 타입을 명시하여 원소의 배열로 데이터를 유지
38   -다차원 데이터도 연속된 메모리 공간이 할당됨
39   -많은 연산이 dimensions과 strides를 잘 활용하면 효율적으로 가능
40   -가령 transpose는 strides를 바꾸는 것으로 거의 공짜
41
42 4. import NumPy
43   import numpy as np
44
45
46
47 5. ndarray 배열 생성
```

```
48 -Refer to https://docs.scipy.org/doc/numpy/reference/routines.array-creation.html
49
50 1)순차적인 객체를 받아 넘겨받은 data가 들어있는 새로운 NumPy 배열을 생성
51 2)array() 함수 이용
52 -Refer to https://cognitiveclass.ai/blog/nested-lists-multidimensional-numpy-arrays/
53 -입력 data(list, tuple, 배열 또는 다른 순차형 data)를 ndarray로 변환
54 -dtype이 명시되지 않은 경우에는 자료형을 추론하여 저장
55 -기본적으로 입력 data는 복사됨.
56
57 data1 = [6,7.5, 8, 0, 1]
58 arr1 = np.array(data1)
59 arr1
60 -----
61 array([6. , 7.5, 8. , 0. , 1. ])
62
63 data2 = [[1,2,3,4], [5,6,7,8]]
64 arr2 = np.array(data2)
65 arr2
66 -----
67 array([[1, 2, 3, 4],
68        [5, 6, 7, 8]])
69
70 arr2.ndim
71 -----
72 2
73
74 arr2.shape
75 -----
76 (2, 4)
77
78 -[1, 2, 3] 배열을 생성.
79 np.array([1, 2, 3])
80
81 -X = [1, 2] 일 때(python list 상태) X를 배열로 변환.
82 X = [1,2]
83 np.array(X)
84
85 # 다른 솔루션
86 np.asarray(X)
87
88 -X = [1, 2] 일 때(python list 상태) X를 'float'형 배열로 변환.
89 X = [1, 2]
90 np.array(X, float)
91
92 # 다른 솔루션
93 np.asarray(X, float)
94
95 # 다른 솔루션
96 np.asfarray(X)
97
98
```

```
99 3)zeros() / zeros_like()
100 -주어진 dtype과 주어진 shape을 가지는 배열을 생성하고 내용을 모두 0으로 초기화한다.
101 -zeros_like는 주어진 배열과 동일한 shape과 dtype을 가지는 배열을 새로 생성하여 내용을 모두 0으로 초기화한다.
102
103 np.zeros(10)
104 -----
105 array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
106
107 np.zeros((3, 6))
108 -----
109 array([[0., 0., 0., 0., 0., 0.],
110        [0., 0., 0., 0., 0., 0.],
111        [0., 0., 0., 0., 0., 0.]])
112
113 arr = np.zeros(10)
114 arr2 = np.zeros_like(arr)
115 print(arr)
116 print(arr2)
117 -----
118 [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]
119 [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]
120
121 #모든 원소가 0으로 채워진 3*2 실수형 다차원 배열을 생성하세요.
122 np.zeros((3,2), float)
123 -----
124 array([[0., 0.],
125        [0., 0.],
126        [0., 0.]])
127 -모든 원소가 0으로 채워진 3*2 실수형 다차원 배열을 생성.
128 np.zeros((3,2), float)
129
130 -X = np.arange(4, dtype=np.int64) 일 때, X와 동일한 shape의 영행렬을 생성.
131 X = np.arange(4, dtype=np.int64)
132 np.zeros_like(x)
133
134
135 4)ones() / ones_like()
136 -주어진 dtype과 주어진 shape을 가지는 배열을 생성하고 내용을 모두 1로 초기화한다.
137 -ones_like는 주어진 배열과 동일한 shape과 dtype을 가지는 배열을 새로 생성하여 내용을 모두 1로 초기화한다.
138
139 np.ones(10)
140 -----
141 array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
142
143 #모든 원소가 1로 채워진 3*2 실수형 다차원 배열을 생성
144 np.ones([3,2], float)
145 -----
146 array([[1., 1.],
147        [1., 1.]])
```

```
148         [1., 1.]))
149
150     -모든 원소가 1로 채워진 3*2 실수형 다차원 배열을 생성.
151     np.ones([3,2], float)
152
153     -X = np.arange(4, dtype=np.int64) 일 때, X와 동일한 shape를 가지고 모든 원소가 1인 다차원 배열
    을 생성.
154     X = np.arange(4, dtype=np.int64)
155     np.ones_like(x)
156
157
158 5)empty() / empty_like()
159     -memory를 할당하여 새로운 배열을 생성하지만 ones나 zeros처럼 값을 초기화하지 않는다.
160     -즉, 0으로 초기화되지 않는 배열을 생성
161
162     np.empty((2,3,2))
163     -----
164     [[[3.19349014e-316, 3.61411312e-316],
165        [3.95252517e-323, 3.61411312e-316],
166        [3.95252517e-323, 3.61411312e-316]],
167
168        [[4.44659081e-323, 0.00000000e+000],
169         [3.61413842e-316, 3.61413842e-316],
170         [3.61413842e-316, 0.00000000e+000]]]
171
172     #2x2 integers 다차원 배열을 초기화 하지 않은 상태로 생성
173     np.empty([2,2], int)
174     -----
175     [[4607182418800017408, 4607182418800017408],
176        [4607182418800017408,          0]]
177
178     #X = np.array([1,2,3], [4,5,6], np.int32) 일 때, X와 동일한 shape를 가지는 다차원 배열을 초기
    화 하지 않은 상태로 생성하기
179     X = np.array([1,2,3], [4,5,6], np.int32)
180     np.empty_like(X)
181     -----
182     array([[1, 2, 3],
183            [4, 5, 6]], dtype=int32)
184
185     -2*2 integers 다차원 배열을 초기화 하지 않은 상태로 생성
186     np.empty([2,2], int)
187
188     -X = np.array([1,2,3], [4,5,6], np.int32) 일 때, X와 동일한 shape를 가지는 다차원 배열을 초기화
    하지 않은 상태로 생성.
189     X = np.array([1,2,3], [4,5,6], np.int32)
190     np.empty_like(X)
191
192
193 6)arange()
194     -Python의 range() 함수의 배열 version.
195     -내장 range()와 유사하지만 list대신 ndarray를 반환
```

```
196
197     np.arange(15)
198     array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
199
200     #x = np.arange(4, dtype=np.int64) 일 때, X와 동일한 shape를 가지고 모든 원소가 1인 다차원
    배열을 생성하세요.
201     x = np.arange(4, dtype=np.int64)
202     np.ones_like(x)
203     -----
204     array([1, 1, 1, 1])
205
206     #x = np.arange(4, dtype=np.int64) 일 때, X와 동일한 shape의 영행렬을 생성하세요.
207     x = np.arange(4, dtype=np.int64)
208     np.zeros_like(x)
209     -----
210     array([0, 0, 0, 0])
211
212     #2, 4, 6, 8, ..., 100 을 원소로 가지는 배열을 생성하세요.
213     np.arange(2, 101, 2)
214     -----
215     array([ 2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26,
216            28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52,
217            54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78,
218            80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100])
219
220     #3.0 에서 10.0 까지 50개의 원소가 균일하게 분포된 1차원 배열을 생성하세요.
221     np.linspace(3., 10., 50)
222     -----
223     array([ 3.         ,  3.14285714,  3.28571429,  3.42857143,  3.57142857,
224            3.71428571,  3.85714286,  4.         ,  4.14285714,  4.28571429,
225            4.42857143,  4.57142857,  4.71428571,  4.85714286,  5.         ,
226            5.14285714,  5.28571429,  5.42857143,  5.57142857,  5.71428571,
227            5.85714286,  6.         ,  6.14285714,  6.28571429,  6.42857143,
228            6.57142857,  6.71428571,  6.85714286,  7.         ,  7.14285714,
229            7.28571429,  7.42857143,  7.57142857,  7.71428571,  7.85714286,
230            8.         ,  8.14285714,  8.28571429,  8.42857143,  8.57142857,
231            8.71428571,  8.85714286,  9.         ,  9.14285714,  9.28571429,
232            9.42857143,  9.57142857,  9.71428571,  9.85714286, 10.         ])
233
234     -2, 4, 6, 8, ..., 100 을 원소로 가지는 배열을 생성.
235     np.arange(2, 101, 2)
236
237     -3.0 에서 10.0 까지 50개의 원소가 균일하게 분포된 1차원 배열을 생성.
238     np.linspace(3., 10., 50)
239
240
241 7)eye(), identity()
242     -N x N 크기의 단위 행렬(좌상단에서 우하단을 잇는 대각선은 1로 채워지고 나머지는 0으로 채운다)을 생성
243
244     np.identity(3) #크기가 3인 단위행렬
245     -----
```

```
246     array([[1., 0., 0.],
247            [0., 1., 0.],
248            [0., 0., 1.]])
249
250     -크기가 3인 단위행렬을 생성.
251     np.eye(3)
252     np.identity(3)
253
254
255 8)full(), full_like()
256
257     # 모든 원소가 6으로 채워진 2*5 uint형(부호없는 정수) 다차원 배열을 생성하세요.
258     np.full((2, 5), 6, dtype=np.uint)
259     -----
260     array([[6, 6, 6, 6, 6],
261            [6, 6, 6, 6, 6]], dtype=uint64)
262
263     # 이것은 다음과 같이 할 수도 있다.
264     np.ones([2, 5], dtype=np.uint) * 6
265
266     # x = np.arange(4, dtype=np.int64)일 때, Create an array of 6's with the same shape
267     # and type as X.
268     x = np.arange(4, dtype=np.int64)
269     np.full_like(x, 6)
270     -----
271     array([6, 6, 6, 6])
272
273     # 또는
274     np.ones_like(x) * 6
275
276 9)asarray()
277     -입력 data를 ndarray로 변환하지만 입력 data가 이미 ndarray일 경우, 복사가 되지 않는다.
278
279     #[1, 2, 3] 배열을 생성하세요
280     np.array([1, 2, 3])
281     -----
282     array([1, 2, 3])
283
284     # x = [1, 2] 일 때(python list 상태) x를 배열로 변환하세요.
285     x = [1,2]
286     np.array(x)
287     -----
288     array([1, 2])
289
290     # 다른 솔루션
291     np.asarray(x)
292     -----
293     array([1, 2])
294
295
```

296

297 6. ndarray의 자료형

298 1)자료형, dtype은 ndarray가 특정 data를 memory에서 해석하기 위해 필요한 정보를 담고 있는 특수한 객체이다.

299

300 arr1 = np.array([1,2,3], dtype=np.float64)

301 arr1.dtype

302 -----

303 dtype('float64')

304

305 arr2 = np.array([1,2,3], dtype=np.int32)

306 arr2.dtype

307 -----

308 dtype('int32')

309

310 2)정수형

311 -정수형의 default data type은 'int64'이다.

312 -진실: default data type은 운영체제에 따라 다르다.

313 -부호없는 정수형의 default data type은 'uint64'이다.

314 -int8 ,unit8(i1, u1)

315 --부호가 있는 8bit(1Byte)와 부호가 없는 8bit 정수형

316 -int16, uint16(i2, u2)

317 --부호가 있는 16bit 정수형과 부호가 없는 16bit 정수형

318 -int32, uint32(i4, u4)

319 --부호가 있는 32bit 정수형과 부호가 없는 32bit 정수형

320 -int64, uint64(i8, u8)

321 --부호가 있는 64bit 정수형과 부호가 없는 64bit 정수형

322

323 intArray = np.array([[1, 2], [3, 4]])

324 intArray

325 intArray.dtype

326

327 3)부호없는 정수형

328 uintArray = np.array([[1, 2], [3, 4]], dtype='uint')

329 uintArray

330 uintArray.dtype

331

332 4)실수형

333 -실수형의 default data type은 'float64'이다.

334

335 floatArray = np.array([[1.1, 2.2], [3.3, 4.4]])

336 floatArray = np.array([[1.1, 2.2], [3.3, 4.4]], dtype='float64')

337 floatArray

338 floatArray.dtype

339

340 -float16(f2)

341 --반정밀도 부동소수점

342 -float32(f4 or f)

343 --단정밀도 부동소수점, C 언어의 float형과 호환

344 -float64(f8 or d)

345 --배정밀도 부동소수점, C 언어의 double형과 Python의 float 객체와 호환

```
346     -float128(f16 or g)
347         --확장 정밀도 부동소수점
348
349 5)복소수형
350     complexArray = np.array([1+1j, 2+2j, 3+3j, 4+4j, 5+5j])
351     complexArray
352     complexArray.dtype
353
354     -complex64, complex128, complex256(c8, c16, c32)
355         --각각 2개의 32, 64, 128bit 부동소수점형을 가지는 복소수
356
357 6)bool
358     -True, False 값을 저장하는 boolean형
359
360     boolArray = np.array([True, False, True, True, False])
361     boolArray
362     boolArray.dtype
363
364 7)object(O)
365     -Python 객체형
366
367 8)string_(S)
368     -고정 길이 문자열형(각 글자는 1Byte).
369     -길이가 10인 글자열의 dtype은 S10이 된다.
370
371 9)unicode_(U)
372     -고정 길이 unicode(platform에 따라 글자별 byte 수는 다르다).
373     -string_형과 같은 형식을 쓴다(예:U10)
374
375 10)dtype 변환하기
376     -데이터가 정수로 입력되더라도 data type을 실수형으로 명시한다면 실수형으로 자동 형변환이 일어난다.
377
378     floatArray2 = np.array([[1, 2], [3, 4]], dtype='float64')
379     floatArray2
380     floatArray2.dtype
381
382     -정수형에서 실수형으로의 형변환 과정은 데이터 손실이 일어나지 않아 문제될 부분이 없지만 반대의 경우에는 문
383     제가 발생할 수 있다.
384
385     intArray2 = np.array([[1.1, 2.2], [3.3, 4.4]], dtype='int')
386     intArray2
387
388     -x = [1, 2] 일 때(python list 상태) x를 float형 배열로 변환하기.
389     x = [1, 2]
390     np.array(x, float)
391     -----
392     array([1., 2.])
393
394     # 다른 솔루션
395     np.asarray(x, float)
396     -----
```



```
396     array([1., 2.])
397
398     # 다른 솔루션
399     np.asfarray(x)
400     -----
401     array([1., 2.])
402
403 11)astype()으로 변환하기
404
405     arr = np.array([1,2,3,4,5])
406     arr.dtype
407     -----
408     dtype('int64')
409
410     float_arr = arr.astype(np.float64)
411     float_arr.dtype
412     -----
413     dtype('float64')
414
415
416     arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
417     arr
418     -----
419     array([ 3.7, -1.2, -2.6,  0.5, 12.9, 10.1])
420
421     arr.astype(np.int32)  # 소수점 아랫자리를 버려진다.
422     -----
423     array([ 3, -1, -2,  0, 12, 10], dtype=int32)
424
425     -숫자 형식의 문자열을 담고 있는 배열이 있다면 astype을 사용하여 숫자로 변환 가능
426     numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
427     numeric_strings.astype(float)
428     -----
429     array([ 1.25, -9.6 , 42.  ])
430
431     -형 변환이 실패하면 TypeError 예외 발생
432     -원래 np.float64라고 해야 하는데, float라고 입력해도 변환가능
433
434     int_array = np.arange(10)
435     calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)
436     int_array.astype(calibers.dtype)
437     -----
438     array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
439
440     -축약 code도 사용 가능
441
442     empty_uint32 = np.empty(8, dtype='u4')
443     empty_uint32
444     -----
445     array([      0, 1072693248,      0, 1072693248,      0,
446            1072693248,      0,      0], dtype=uint32)
```

```
447
448
449
450 7. 배열 다루기 루틴
451 1)X가 10 x 10 x 3의 다차원 배열일때, X의 두번째 차원이 150인 2차원 배열이 되도록 reshape하기.
452     X = np.ones([10, 10, 3])
453     X.reshape(2,150)
454
455 2)X가 [[1, 2, 3], [4, 5, 6]]일 때, [1 4 2 5 3 6]로 변환.
456     X = np.array([[1, 2, 3], [4, 5, 6]])
457     np.ravel(X)
458     X.flatten()
459
460 3)X가 [[1, 2, 3], [4, 5, 6]]일 때, flatten한 후 5번째 원소를 가져오기.
461     X = np.array([[1, 2, 3], [4, 5, 6]])
462     out = X.flatten()
463     out[4]
464
465 4)다음 X와 Y를 연결해서 [[1, 2, 3, 7, 8, 9], [4, 5, 6, 10, 11, 12]] 를 만들기.
466     X= [[ 1 2 3], [ 4 5 6]]
467     Y = [[ 7 8 9],[10 11 12]]
468
469     X = np.array([[1, 2, 3], [4, 5, 6]])
470     Y = np.array([[7, 8, 9], [10, 11, 12]])
471     np.concatenate((X, Y), 1)
472
473 5)다음의 X와 Y를 연결해서 [[ 1 2 3], [ 4 5 6], [ 7 8 9], [10 11 12]]를 만들기.
474     X = [[ 1 2 3], [ 4 5 6]]
475     Y = [[ 7 8 9], [10 11 12]]
476
477     X = np.array([[1, 2, 3], [4, 5, 6]])
478     Y = np.array([[7, 8, 9], [10, 11, 12]])
479     np.concatenate((X, Y), 0)
480
481 6)X가 [0, 1, 2]일 때, [0, 0, 1, 1, 2, 2]를 생성.
482     X = np.array([0, 1, 2])
483     np.repeat(X, 2)
484
485 7)X가 [0, 0, 0, 1, 2, 3, 0, 2, 1, 0]일 때, 앞, 뒤의 0을 제거.
486     X = np.array((0, 0, 0, 1, 2, 3, 0, 2, 1, 0))
487     np.trim_zeros(X)
488
489
490
491 8. 배열과 Scala간의 연산
492 1)for 반복문을 사용하지 않고 data를 일괄처리 가능 --> vector화
493 2)같은 크기의 배열 간 산술연산은 배열의 각 요소 단위로 적용된다.
494
495     arr = np.array([[1., 2., 3.], [4., 5., 6.]])
496     arr
497     -----
```

```
498     array([[1., 2., 3.],
499            [4., 5., 6.]])
500
501     arr * arr
502     -----
503     array([[ 1.,  4.,  9.],
504            [16., 25., 36.]])
505
506     arr - arr
507     -----
508     array([[0., 0., 0.],
509            [0., 0., 0.]])
510
511     1 / arr
512     -----
513     array([[1.      , 0.5      , 0.33333333],
514            [0.25     , 0.2      , 0.16666667]])
515
516
517     arr ** 0.5
518     -----
519     array([[1.      , 1.41421356, 1.73205081],
520            [2.      , 2.23606798, 2.44948974]])
```

524 9. Indexing과 Slicing

525 1)Slicing

- 526 -Python list와 유사하게, Numpy 배열도 슬라이싱이 가능하다.
- 527 -Numpy 배열은 다차원인 경우가 많기에, 각 차원별로 어떻게 슬라이스할건지 명확히 해야 한다
- 528 -Numpy 배열을 슬라이싱하면, 연속된 값을 가져오기에 결과로 얻어지는 배열은 언제나 원본 배열의 부분 배열이다.

```
529
530     # shape가 (3, 4)인 2차원 배열 생성
531     a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
532
```

- 533 -다차원배열은 아래와 같은 방법으로 슬라이싱 한다.

```
534     a[ :, : ]
```

- 535 -a[행 슬라이싱 시작:행 슬라이싱 끝, 열 슬라이싱 시작:열 슬라이싱 끝]

```
536
```

- 537 -시작값부터 끝값전까지 슬라이싱 된다.

```
538     a[0:2, 0:4]
```

- 539 #0행부터 2행 전까지(= 1행 까지), 0열부터 4열 전까지(= 3열 까지) 슬라이스되었다.

```
540
```

- 541 -시작값이 0인 경우 생략 가능하며 끝 값이 shape의 값과 동일한 경우 생략 가능하다.

- 542 -a[0:2, 0:4]은 a[:2, :]로 표시할 수 있다.

```
543
```

```
544     a[:, 2, :]
```

- 545 -위 코드는 열을 슬라이싱 하지 않는 코드이며 이런 경우 열 부분은 전체 생략 가능하다.

- 546 -행을 슬라이싱 하지 않는다고 하더라도, 행부분을 생략할 순 없다.

```
547
```

```
548     a[:2]
549     -슬라이싱에 익숙해지기 전까진 생략하지 말고 명시적으로 코드를 작성해주는것이 실수를 방지하는 방법이다.
550
551
552 2)Indexing
553     -Indexing을 통해 원소에 접근할 수 있다.
554     -두 가지 표현법이 있다.
555         # 선호하는 방식
556         a[0, 0]
557         a[0][0]
558
559     -Numpy 배열을 슬라이싱하면, 연속된 값을 가져오기에 결과로 얻어지는 배열은 언제나 원본 배열의 부분 배열
    이다.
560     -그러나 인덱싱을 한다면, 연속하지 않은 값을 가져올 수 있으니 원본과 다른 배열을 만들 수 있다.
561
562         # 0행, 2행만 인덱싱
563         a[[0, 2], ]
564
565         # 0열, 1열, 3열만 인덱싱
566         a[:, [0,1,3]]
567
568     arr2d = np.array([[1,2,3], [4,5,6], [7,8,9]])
569     arr2d[2]
570     -----
571     array([7, 8, 9])
572
573     #아래의 2개의 표현을 같다.
574     arr2d[0][2]
575     -----
576     3
577
578     arr2d[0, 2]
579     -----
580     3
581
582
583 3)차원
584     -정수 Indexing과 Slicing을 혼합해서 사용하면 낮은 차원의 배열이 생성되지만,
585     -Slicing만 사용하면 원본 배열과 동일한 차원의 배열이 생성된다.
586     -0번째 행을 인덱싱하는 경우와 슬라이싱 하는 경우를 비교해보자.
587
588     a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
589     print(a, a.shape, a.ndim)
590
591     # 슬라이싱만 사용
592     slicedRow = a[0:1, :]
593     print(slicedRow, slicedRow.shape, slicedRow.ndim)
594
595     # 아래와 동일한 코드
596     # 인덱싱&슬라이싱 혼합 사용
597     indexedRow = a[0]
```

```
598     print(indexedRow, indexedRow.shape, indexedRow.ndim)
599
600     # 위와 동일한 코드
601     # 인덱싱&슬라이싱 혼합 사용
602     indexedRow2 = a[0, :]
603     print(indexedRow2, indexedRow2.shape, indexedRow2.ndim)
604
605     -행이 아닌 열의 경우에도 마찬가지이다.
606
607     # 슬라이싱만 사용
608     slicedCol = a[:, 0:1]
609     print(slicedCol, slicedCol.shape, slicedCol.ndim)
610
611     # 인덱싱&슬라이싱 혼합 사용
612     indexedCol = a[:, 0]
613     print(indexedCol, indexedCol.shape, indexedCol.ndim)
614
615     -다차원배열간 연산에서 차원이 달라 문제가 발생하는 경우가 종종 있다.
616     -특히 인덱싱을 써야 하는 상황이 아니라면, 인덱싱보다 슬라이싱을 추천.
617
618
619 4)Python의 list와 다른 점
620     -배열의 slice은 원본 배열의 view이다.
621     -즉, data는 값복사되는 것이 아니라 그대로 원본 배열에 반영된다.
622     -NumPy는 대용량 data 처리를 염두에 두고 설계되었기 때문에, 만약 NumPy가 data의 값복사를 남발한다
        면 성능과 memory문제에 직면할 것이기 때문이다.
623     -Indexing은 값을 복사한다.
624     -복사된 값을 변경해도 원본의 값은 변하지 않는다.
625     -Slicing된 배열은 원본 배열과 같은 데이터를 참조한다.
626     -만일 값복사를 하려면 arr[5:8].copy()를 사용해서 명시적으로 배열을 복사하면 된다.
627
628     # 인덱싱해서 b에 대입
629     b = a[0, 0]
630
631     # b값 수정
632     b = 100
633
634     print("a[0, 0]: {}".format(a[0, 0]))
635     print("b: {}".format(b))
636
637     -Slicing 배열은 원본 배열과 같은 데이터를 참조한다.
638     -즉 Slicing된 배열을 수정하면 원본 배열 역시 수정된다.
639
640     # a를 슬라이스하여 c 생성
641     a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
642     c = a[1:3, 1:3]
643     print(c)
644
645     # c[0, 0]은 a[1, 1]과 같은 데이터.
646     c[0, 0] = 100
647
```

```
648     print(c)
649     print('-----')
650     print(a)
651
652
653     arr[5:8] = 12
654     arr
655     -----
656     array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
657
658     arr_slice = arr[5:8]
659     arr_slice[1] = 12345
660     arr
661     -----
662     array([  0,   1,   2,   3,   4, 64, 12345,  64,   8,   9])
663
664     arr_slice[:] = 64
665     arr
666     -----
667     array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
668
669
```

5) 다차원 배열에서 마지막 index를 생략하면 반환되는 객체는 상위 차원의 **data**를 포함하고 있는, 한 차원 낮은 ndarray가 된다.

-즉, arr3d가 2 x 2 x 3크기의 배열이라면 arr3d[0]은 2 x 3 크기의 배열이다.

```
671
672
673     arr3d = np.array([[[1,2,3], [4,5,6]], [[7,8,9], [10,11,12]]])
674     arr3d
675     -----
676     array([[[ 1,  2,  3],
677             [ 4,  5,  6]],
678            [[ 7,  8,  9],
679             [10, 11, 12]]])
680
681
682     arr3d[0]
683     -----
684     array([[1, 2, 3],
685            [4, 5, 6]])
686
687     -arr3d[0]에는 scala값과 배열 모두 대입할 수 있다.
688
689     old_values = arr3d[0].copy()
690     arr3d[0] = 42
691     arr3d
692     -----
693     array([[[42, 42, 42],
694             [42, 42, 42]],
695            [[ 7,  8,  9],
696             [10, 11, 12]]])
697
```

```
698
699
700     arr3d[0] = old_values
701     arr3d
702     -----
703     array([[[ 1,  2,  3],
704            [ 4,  5,  6]],
705           [[ 7,  8,  9],
706            [10, 11, 12]]])
707
708
709     arr3d[1, 0]
710     -----
711     array([7, 8, 9])
712
713
714
```

715 10. Boolean 배열 Indexing

- 716 1) Boolean 배열 Indexing을 통해 배열 속 요소를 취사선택할 수 있다.
717 2) Boolean 배열 Indexing은 특정 조건을 만족하게 하는 요소만 선택하고자 할 때 자주 사용된다.

```
718
719     a = np.array([[1,2], [3, 4], [5, 6]])
720     print(a)
721
722     bool_idx = (a > 2) # 2보다 큰 a의 요소를 찾는다.
723     # 이 코드는 a와 shape가 같고 불리언 자료형을 요소로 하는 numpy 배열을 반환한다.
724     # bool_idx의 각 요소는 동일한 위치에 있는 a의 요소가 2보다 큰지를 말해준다.
725
726     print(bool_idx)    # 출력 "[[False False]
727                       #      [ True  True]
728                       #      [ True  True]]"
```

- 730 3) Boolean 배열 Indexing을 통해 bool_idx에서 참 값을 가지는 요소로 구성되는 rank 1인 배열을 구성할 수 있다.

```
731
732     print(a[bool_idx]) # 출력 "[3 4 5 6]"
733
734     # 위에서 한 모든것을 한 문장으로 할 수 있다.
735
736     print(a[a > 2])    # 출력 "[3 4 5 6]"
737
738
739
```

740 11. 정수 배열 Indexing

```
741     a = np.array([[1,2], [3, 4], [5, 6]])
742     print(a)
743
744     1)정수 배열 인덱싱의 예.
745     # 반환되는 배열의 shape는 (3,)
746     print(a[[0, 1, 2], [0, 1, 0]]) # 출력 "[1 4 5]"
747
```

```
748     # 위에서 본 정수 배열 인덱싱 예제는 다음과 동일하다.
749     print(np.array([a[0, 0], a[1, 1], a[2, 0]])) # 출력 "[1 4 5]"
750
751     # 정수 배열 인덱싱을 사용할 때 원본 배열의 같은 요소를 재사용할 수 있다.
752     print(a[[0, 0], [1, 1]]) # 출력 "[2 2]"
753
754     # 위 예제는 다음과 동일하다.
755     print(np.array([a[0, 1], a[0, 1]])) # 출력 "[2 2]"
756
757
758
759 12. 전치(Transpose)
760     1)종종 배열의 모양을 바꾸거나 데이터를 처리해야 할 때가 있다.
761     2)가장 간단한 예는 행렬의 주 대각선을 기준으로 대칭되는 요소끼리 뒤바꾸는 것이다.
762     3)이를 전치라고 하며 행렬을 전치하기 위해선, 간단하게 배열 객체의 'T' 속성을 사용하면 된다.
763
764     x = np.array([[1,2], [3,4]])
765     print(x)
766
767     print(x.T)
768
769     4)차원이 1인 배열을 전치할 경우 아무 일도 일어나지 않는다.
770
771     v = np.array([1,2,3])
772     print(v) # 출력 "[1 2 3]"
773     print(v.T) # 출력 "[1 2 3]"
774
775
776
777 13. Shape 변경
778     1)numpy.reshape
779     -Gives a new shape to an array without changing its data
780
781     np.arange(6)
782     np.arange(6).reshape((3, 2))
783     a = np.arange(6).reshape((3, 2))
784     a
785     np.reshape(a, (2,3))
786
787     # 2차원
788     np.reshape(a, (1,6))
789
790     # 1차원
791     np.reshape(a, 6)
792
793
794     2)numpy.ravel()
795     -Return a contiguous flattened array.
796
797     np.ravel(a)
798     a.ravel()
```



```
799
800 3)numpy.ndarray.flatten()
801     -Return a copy of the array collapsed into one dimension.
802
803     #np.flatten(a) # Numpy 모듈 함수가 아님
804     a.flatten()    # ndarray 객체의 메소드로만 사용 가능
805
806 4)ravel()과 flatten() 차이점
807     - flatten은 객체의 메소드로만 사용 가능
808     - ravel()은 뷰를 반환, flatten은 복사본을 반환
809
810 5)numpy.concatenate
811     -연결
812     -Refer to
813     https://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html#joining-arrays
814
815     a = np.array([[1, 2], [3, 4]])
816     b = np.array([[5, 6]])
817     np.concatenate((a, b), axis=0)
818     a
819     a[0:1].T
820     np.concatenate((a, b.T), axis=1)
821
822
823 14. 다차원 배열 연산
824     1)기본적인 수학함수는 배열의 각 요소별로 동작하며 연산자를 통해 동작하거나 numpy 함수모듈을 통해 동작한다.
825     2)다차원 배열간 연산시, shape가 맞아야 연산이 이루어진다.
826     3)요소별 합, 차, 곱, 나눗셈의 경우 shape가 일치해야 한다.
827     4)dot의 경우 앞 배열의 열과 뒤 배열의 행의 크기가 일치해야 한다.
828     5)Refer to https://docs.scipy.org/doc/numpy/reference/routines.math.html
829
830     x = np.array([[1., 2.], [3., 4.]])
831     y = np.array([[5., 6.], [7., 8.]])
832
833 6)요소별 합
834     x = [[ 6.0  8.0]
835     y = [10.0 12.0]]
836     print(x + y)
837     print(np.add(x, y))
838
839 7)요소별 차
840     x = [[-4.0 -4.0]
841     y = [-4.0 -4.0]]
842     print(x - y)
843     print(np.subtract(x, y))
844
845 8)요소별 곱
846     x = [[ 5.0 12.0]
```

```
847     y = [21.0 32.0]]
848     print(x * y)
849     print(np.multiply(x, y))
850
851 9)요소별 나눗셈
852     x = [[ 0.2      0.33333333]
853     y = [ 0.42857143  0.5      ]]
854     print(x / y)
855     print(np.divide(x, y))
856
857 10)요소별 제곱근
858     x = [[ 1.      1.41421356], [ 1.73205081  2.      ]]
859     print(np.sqrt(x))
860
861 11)Numpy에선 벡터의 내적, 벡터와 행렬의 곱, 행렬곱을 위해 `*` 대신 `dot` 함수를 사용한다.
862 12)dot은 Numpy 모듈 함수로서도 배열 객체의 메소드로서도 이용 가능한 함수이다
863
864     x = np.array([[1,2],[3,4]])
865     y = np.array([[5,6],[7,8]])
866
867     v = np.array([9,10])
868     w = np.array([11, 12])
869
870     # 벡터의 내적; 둘 다 결과는 219
871     print(v.dot(w))
872     print(np.dot(v, w))
873
874 13)행렬과 벡터의 곱
875     #둘 다 결과는 dimension 1인 배열 [29 67]
876     print(x.dot(v))
877     print(np.dot(x, v))
878
879 14)행렬곱
880     #둘 다 결과는 dimension 2인 배열
881     x = [[19 22]
882     y = [43 50]]
883     print(x.dot(y))
884     print(np.dot(x, y))
885
886
887
888 25. NumPy 고급 기능
889 1)Broadcasting은 Numpy에서 shape가 다른 배열 간에도 산술 연산이 가능하게 하는 메커니즘이다.
890 2)종종 작은 배열과 큰 배열이 있을 때, 큰 배열을 대상으로 작은 배열을 여러 번 연산하고자 할 때가 있다.
891 3)예를 들어, 행렬의 각 행에 상수 벡터를 더하는 걸 생각해보자.
892 4)이는 다음과 같은 방식으로 처리될 수 있습니다.
893
894     # 행렬 x의 각 행에 벡터 v를 더한 뒤,
895     # 그 결과를 행렬 y에 저장하고자 한다.
896     x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
897     x = "[[1 2 3]
```

```
898         [4 5 6]
899         [7 8 9]
900         [10 11 12]]]"
901
902     v = np.array([1, 0, 1])
903
904     x
905     v
906     x.shape
907     v.shape
908
909     y = np.empty_like(x) # x와 동일한 shape를 가지며 비어있는 행렬 생성
910     y
911
912     # 명시적 반복문을 통해 행렬 x의 각 행에 벡터 v를 더하는 방법
913     for i in range(4):
914         y[i, :] = x[i, :] + v
915
916     # 이제 y는 다음과 같다.
917     # [[ 2  2  4]
918     #   [ 5  5  7]
919     #   [ 8  8 10]
920     #  [11 11 13]]
921     print(y)
922
923     5)x가 매우 큰 행렬이라면, 파이썬의 반복문을 이용한 위 코드는 매우 느려질 수 있다.
924     6)벡터 v를 행렬 x의 각 행에 더하는 것은 v를 여러 개 복사해 수직으로 쌓은 행렬 vv를 만들고 이 vv를 x에 더하
    는것과 동일하다.
925     7)이 과정을 아래처럼 구현할 수 있습니다:
926
927     # 벡터 v를 행렬 x의 각 행에 더한 뒤,
928     # 그 결과를 행렬 y에 저장하고자 한다.
929     x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
930     v = np.array([1, 0, 1])
931
932     vv = np.tile(v, (4, 1)) # v의 복사본 4개를 위로 차곡차곡 쌓은 것이 vv
933
934     vv.shape
935
936     print(vv)          # 출력 "[[1 0 1]
937                        #      [1 0 1]
938                        #      [1 0 1]
939                        #      [1 0 1]]]"
940
941     y = x + vv # x와 vv의 요소별 합
942     print(y)    # 출력 "[[ 2  2  4
943                  #      [ 5  5  7]
944                  #      [ 8  8 10]
945                  #      [11 11 13]]]"
946
947     8)Numpy 브로드캐스팅을 이용한다면 이렇게 v의 복사본을 여러 개 만들지 않아도 동일한 연산을 할 수 있다.
```

948 9)아래는 브로드캐스팅을 이용한 예시 코드입니다.

949

```
950 import numpy as np
```

951

```
952 # 벡터 v를 행렬 x의 각 행에 더한 뒤,
```

```
953 # 그 결과를 행렬 y에 저장하고자 한다.
```

```
954 x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
```

```
955 v = np.array([1, 0, 1])
```

```
956 y = x + v # 브로드캐스팅을 이용하여 v를 x의 각 행에 더하기
```

```
957 print(y) # 출력 "[[ 2  2  4]
```

```
958          #      [ 5  5  7]
```

```
959          #      [ 8  8 10]
```

```
960          #      [11 11 13]]"
```

961

962 10)x의 shape가 (4, 3)이고 v의 shape가 (3,)라도 브로드캐스팅으로 인해 $y = x + v$ 는 문제없이 수행된다.

963 11)이때 `v`는 `v`의 복사본이 차곡차곡 쌓인 shape (4, 3)처럼 간주되어 `x`와 동일한 shape가 되며 이들 간의 요소별 덧셈연산이 y에 저장된다.

964 12)두 배열의 브로드캐스팅은 아래의 규칙을 따른다.

965 -두 배열이 동일한 dimension을 가지고 있지 않다면, 낮은 dimension의 1차원 배열이 높은 dimension 배열의 shape로 간주한다.

966 -특정 차원에서 두 배열이 동일한 크기를 갖거나, 두 배열 중 하나의 크기가 1이라면 그 두 배열은 특정 차원에서 compatible하다고 여겨진다.

967 -두 행렬이 모든 차원에서 compatible하다면, 브로드캐스팅이 가능하다.

968 -브로드캐스팅이 이뤄지면, 각 배열 shape의 요소별 최소공배수로 이루어진 shape가 두 배열의 shape로 간주한다.

969 -차원에 상관없이 크기가 1인 배열과 1보다 큰 배열이 있을 때, 크기가 1인 배열은 자신의 차원 수만큼 복사되어 쌓인 것처럼 간주한다.

970

971 13)브로드캐스팅을 지원하는 함수를 universal functions라고 한다.

972 14)Refer to <https://docs.scipy.org/doc/numpy/reference/ufuncs.html>

973 15)다음은 브로드캐스팅을 응용한 예시들이다.

974

```
975 import numpy as np
```

976

```
977 v = np.array([1,2,3]) # v의 shape는 (3,)
```

```
978 w = np.array([4,5]) # w의 shape는 (2,)
```

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

```
980               [4,5,6]])
```

981

```
982 print(x)
```

983

```
984 # 벡터를 행렬의 각 행에 더하기
```

```
985 # x는 shape가 (2, 3)이고 v는 shape가 (3,)이므로 이 둘을 브로드캐스팅하면 shape가 (2, 3)인 아래와 같은 행렬이 나온다.
```

```
986 # [[2 4 6]
```

```
987 #  [5 7 9]]
```

```
988 print(x + v)
```

989

```
990 x.T
```

```
991 w
```

```
992 x.T + w
```

```
993     (x.T + w).T
994
995     # 벡터를 행렬의 각 행에 더하기
996     # x는 shape가 (2, 3)이고 w는 shape가 (2,)이다.
997     # x의 전치행렬은 shape가 (3,2)이며 이는 w와 브로드캐스팅이 가능하고 결과로 shape가 (3,2)인 행렬이
    생긴다.
998     # 이 행렬을 전치하면 shape가 (2,3)인 행렬이 나오며
999     # 이는 행렬 x의 각 열에 벡터 w를 더한 결과와 동일하다.
1000    # [[ 5  6  7]
1001    #   [ 9 10 11]]
1002    print((x.T + w).T)
1003
1004    w
1005    np.reshape(w, (2, 1))
1006
1007    # 다른 방법은 w를 shape가 (2,1)인 열벡터로 변환하는 것이다.
1008    # 그런 다음 이를 바로 x에 브로드캐스팅해 더하면
1009    # 동일한 결과가 나온다.
1010    print(x + np.reshape(w, (2, 1)))
1011
1012    16)브로드캐스팅은 보통 코드를 간결하고 빠르게 한다.
1013    17)그래서 권장한다.
1014
1015
1016
1017    26. 파일에서 데이터를 입력 받아 다차원 배열 생성하기
1018    1)np.genfromtxt()을 이용하여 파일에 저장된 데이터를 입력받아 다차원 배열을 생성할 수 있다.
1019    2)하지만 사용할 일은 많지 않다. 그 이유는,
1020    -NumPy ndarray는 동일한 데이터타입만을 가질 수 있다.
1021    -대부분 데이터 파일에는 하나의 데이터 타입만 있는게 아니라 정수, 실수, 문자열이 섞여 있다.
1022    3)파일에서 데이터를 읽어올때 NumPy의 genfromtxt()보다는 Pandas의 read_csv()나 read_excel()을
    주로 사용한다.
1023
1024    #파일 다운로드 받기
1025    !wget -O 'mnist_train_super_small.csv'
    https://docs.google.com/spreadsheets/d/1IUBHgDIG5EVozxfSF84X5jbcyynjnWSK8\_weiQApt2M/export?format=csv
1026
1027    import numpy as np
1028    fromCSVArray = np.genfromtxt('mnist_train_super_small.csv', delimiter=',')
1029    fromCSVArray
1030
1031
1032
1033    27. Google Drive에서 파일 가져오기
1034    1)Google Drive Mount
1035
1036    from google.colab import drive
1037    drive.mount('/content/drive')
1038
1039    # Google Drive내에 있는 파일 경로 지정
```

```
1040 fromCSVArray = np.genfromtxt('/content/drive/My
Drive/talk-on-seminar-numpy/resources/mnist_train_super_small.csv', delimiter=',')
1041 fromCSVArray
1042
1043
1044
1045 28. Statistics
1046 1)Refer to https://docs.scipy.org/doc/numpy/reference/routines.statistics.html
1047 2)numpy.amin
1048 -Return the minimum of an array or minimum along an axis.
1049
1050 A = np.arange(4).reshape((2,2))
1051 np.amin(A, 0)
1052 np.amin(A, axis=0)
1053
1054 np.amin(A, 1) # np.amin(A, axis=1)
1055 np.amin(A)
1056
1057 3)numpy.amax
1058 -Return the maximum of an array or maximum along an axis.
1059
1060 A = np.arange(4).reshape((2,2))
1061 np.amax(A, 0) # np.amax(A, axis=0)
1062
1063 np.amax(A, 1) # np.amax(A, axis=1)
1064 np.amax(A, 1).shape
1065 np.amax(A)
1066
1067 4)numpy.ptp
1068 -Range of values (maximum - minimum) along an axis.
1069 -The name of the function comes from the acronym for 'peak to peak'.
1070
1071 A = np.arange(4).reshape((2,2))
1072 np.ptp(A, 0)
1073 np.ptp(A, 1)
1074 np.ptp(A)
1075
1076 5)numpy.median
1077 -Compute the median along the specified axis.
1078
1079 A = np.array([[10, 7, 4], [3, 2, 1]])
1080 np.median(A, 0)
1081 np.median(A, 1)
1082 np.median(A)
1083
1084 6)numpy.mean
1085 -Compute the arithmetic mean along the specified axis.
1086 -가중평균을 구하려면 numpy.average를 사용해야 한다.
1087
1088 A = np.array([[1, 2], [3, 4]])
1089 np.mean(A, 0)
```

```
1090     np.mean(A, 1)
1091     np.mean(A)
1092
1093 7)numpy.var
1094     -Compute the variance along the specified axis.
1095
1096     A = np.array([[1, 2], [3, 4]])
1097     np.var(A, 0)
1098     np.var(A, 1)
1099     np.var(A)
1100
1101 8)numpy.std
1102     -Compute the standard deviation along the specified axis.
1103
1104     A = np.array([[1, 2], [3, 4]])
1105     np.std(A, 0)
1106     np.std(A, 1)
1107     np.std(A)
```