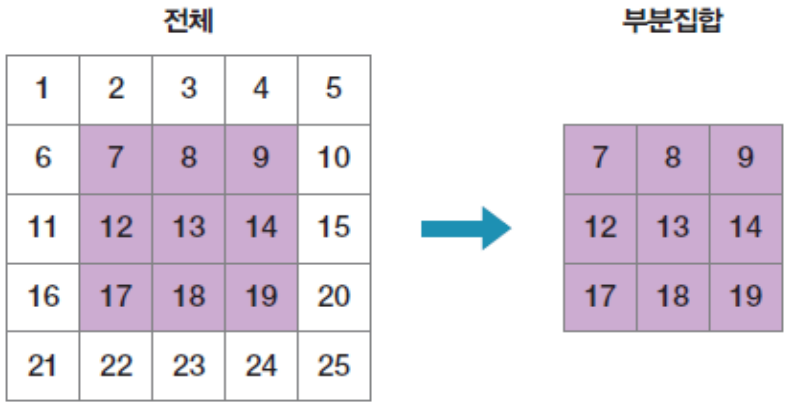
넘파이

내용

- 넘파이 배열과 파이썬 리스트의 차이를 이해한다.
- 넘파이 배열의 장점과 간편성을 학습한다.
- 넘파이의 데이터 형식과 함수들의 사용법을 익힌다.
- 넘파이를 활용한 다양한 프로그램을 작성한다.

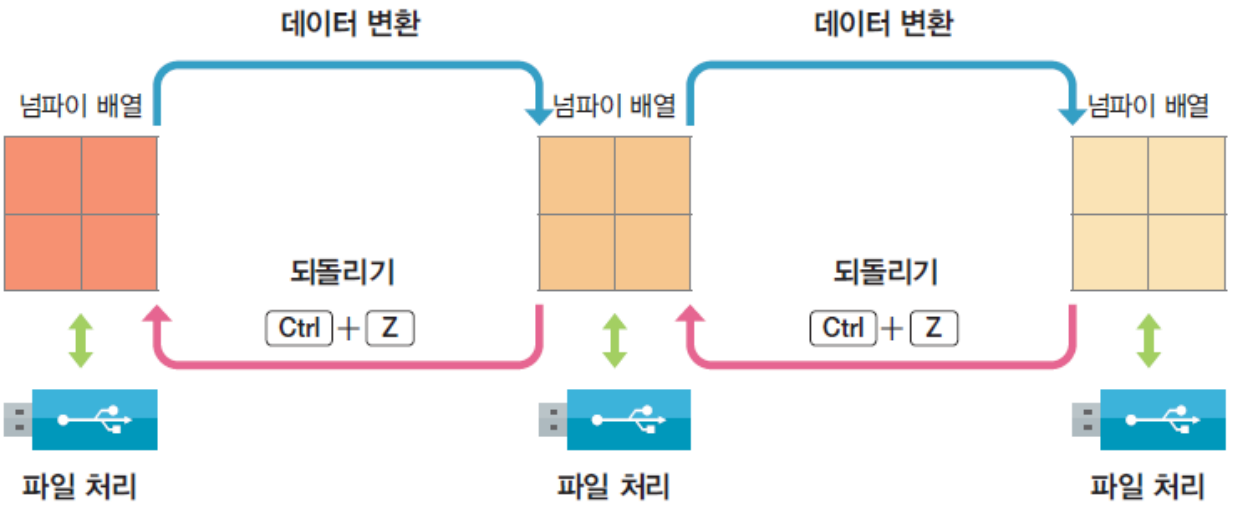
Section 01 이 장에서 만들 프로그램

- **[프로그램 1] 부분집합 추출**
  - 파이썬 리스트와 비교하여 넘파이 배열의 장점을 파악하는 예제



Section 01 이 장에서 만들 프로그램

- **[프로그램 2] 파일 입출력**
  - 넘파이 파일 처리와 함께 Ctrl+Z를 누르면 이전 작업으로 되돌리는 개념을 학습함



Section 02 넘파이 기초

넘파이 개념

- Numerical Python의 약자
- 파이썬에서 배열을 처리할 때 널리 사용되는 유용하고 필수적인 라이브러리
- 특히 배열 계산에 최적화된 방식과 빠른 처리 속도를 자랑함

넘파이 특징

- 처리 속도가 빠름
- 유연한 연산 제공
- 배열의 내용을 한번에 디스크에 저장하고, 다시 불러오는 기능을 제공
- C/C++로 작성한 코드와 연결 가능
- 판다스, OpenCV 등 유명한 라이브러리가 넘파이 기반

Section 02 넘파이 기초

넘파이 개념

- 정수를 저장하기 위해 5×5 크기의 랜덤값으로 초기화된 2차원 리스트를 만들고, 리스트의 모든 값에 100을 더하는 코드

Code10-01.py

```
01    import random
02
03    ## 파이썬 2차원 리스트 생성
04    SIZE = 5
05    pythonList = [ [random.randint(0,255) for _ in range(SIZE)] for _ in range(SIZE)]
06
07    ## 리스트를 출력하기
08    for i in range(SIZE) :
09        for k in range(SIZE) :
10            print("%3d" % pythonList[i][k], end=' ')
11        print()
12    print()
13
14    ## 리스트에 100을 더하기
15    for i in range(SIZE) :
16        for k in range(SIZE) :
17            pythonList[i][k] += 100
```

넘파이 개념

- 정수를 저장하기 위해 5×5 크기의 랜덤값으로 초기화된 2차원 리스트를 만들고, 리스트의 모든 값에 100을 더하는 코드

Code10-01.py

```
18
19    ## 리스트를 출력하기
20    for i in range(SIZE) :
21        for k in range(SIZE) :
22            print("%3d" % pythonList[i][k], end=' ')
23        print()
```

실행 결과

```
  0 110  66 111  65
172  13 253  19  54
 16 248  16   8 159
231 125 131 171 135
182  14 251  88 175

100 210 166 211 165
272 113 353 119 154
116 348 116 108 259
331 225 231 271 235
282 114 351 188 275
```

Section 02 넘파이 기초

- **넘파이 개념**
  - 넘파이 라이브러리를 사용하려면 외부 라이브러리를 설치해야 함

```
pip install numpy
```

넘파이 개념

- Code10-01.py와 동일한 결과를 넘파이를 사용해서 작성

Code10-02.py

```
01    import numpy as np
02
03    ## 넘파이 2차원 배열 생성
04    SIZE = 5
05    numpyAry = np.random.randint(0, 255, size=(SIZE, SIZE))
06
07    ## 배열을 출력하기
08    print(numpyAry)
09    print()
10
11    ## 배열에 100을 더하기
12    numpyAry += 100
13
14    ## 배열을 출력하기
15    print(numpyAry)
```

실행 결과

```
[[  6  31 220 170 241]
 [  5 121 200  42 109]
 [ 50  82 228 215 185]
 [222 104  69  47  73]
 [148 192  80  95 241]]

[[106 131 320 270 341]
 [105 221 300 142 209]
 [150 182 328 315 285]
 [322 204 169 147 173]
 [248 292 180 195 341]]
```

넘파이 개념

SELF STUDY 10-1

Code10-02.py을 참고하여 3×3 크기의 넘파이 배열을 생성하자. 그리고 배열의 모든 값을 2배로 증가시킨 후 평균값을 계산하자. 단, 평균값은 파이썬 리스트처럼 for 문을 사용해서 계산한다.

실행 결과

```
[[211 113 162]
 [138 215  77]
 [215 122  34]]

[[422 226 324]
 [276 430 154]
 [430 244  68]]

평균 값 --> 286.0
```

넘파이 배열

- 넘파이는 주로 1차원, 2차원, 3차원 등의 배열을 처리하기 위한 용도로 만들어짐
- 주로 넘파이 배열을 생성할 때는 다음과 같은 함수들을 많이 사용

표 10-1 자주 사용되는 넘파이 배열 생성 함수

넘파이 함수명	설명
np.array()	파이썬 리스트를 넘파이 배열로 변환한다. 변환 시 데이터형을 지정할 수 있다.
np.arange()	파이썬의 range() 함수와 유사하지만 넘파이 배열용으로 주로 사용된다.
np.ones()	모두 1이 채워진 넘파이 배열을 생성한다.
np.zeros()	모두 0이 채워진 넘파이 배열을 생성한다.
np.empty()	np.ones()나 np.zeros()처럼 배열을 생성하지만 초기화하지 않는다. 즉 쓰레기 값이 들어 있을 수 있다.
np.full()	배열을 생성하면서 파라미터로 넘겨받은 값으로 초기화한다.
np.identity()	대각선은 1로, 나머지는 0으로 채워진 2차원 넘파이 배열을 생성한다. 이를 '단위 행렬'이라 부른다.
np.random.randint()	지정한 범위의 정수 중 임의의 값을 채운 넘파이 배열을 생성한다.
np.random.rand()	0부터 0.9999 범위의 임의의 값을 채운 넘파이 배열을 생성한다.

넘파이 배열

- [표 10-1]의 함수들을 사용해서 넘파이 배열을 생성하는 코드를 작성

Code10-03.py

```
01    import numpy as np
02    import random
03
04    pythonList = [ random.randint(0,255) for _ in range(5)]
05    print('* 파이썬 리스트 --> ', pythonList)
06
07    numpyAry1 = np.array(pythonList)
08    print('* array(pythonList) --> ', numpyAry1)
09
10    numpyAry2 = np.arange(5)
11    print('* arange(5) --> ', numpyAry2)
12    numpyAry3 = np.arange(3, 8)
13    print('* arange(3, 8) --> ', numpyAry3)
14    numpyAry3 = np.arange(0, 100, 20)
15    print('* arange(0, 100, 20) --> ', numpyAry3)
16
```

넘파이 배열

- [표 10-1]의 함수들을 사용해서 넘파이 배열을 생성하는 코드를 작성

Code10-03.py

```
17   numpyAry4 = np.ones(5)
18   print('* ones(5) --> \n', numpyAry4)
19   numpyAry5 = np.ones((3,4))
20   print('* ones((3,4)) )--> \n', numpyAry5)
21
22   numpyAry6 = np.zeros(5)
23   print('* zeros(5)--> ', numpyAry6)
24
25   numpyAry7 = np.empty(6)
26   print('* empty(6)--> ', numpyAry7)
27
28   numpyAry8 = np.full(5, 33)
29   print('* full(5, 33) --> ', numpyAry8)
30
31   numpyAry9 = np.identity(5)
32   print('* identity(5)--> \n', numpyAry9)
```

넘파이 배열

- [표 10-1]의 함수들을 사용해서 넘파이 배열을 생성하는 코드를 작성

실행 결과

```
* 파이썬 리스트 --> [249, 68, 184, 169, 231]
* array(pythonList) --> [249  68 184 169 231]
* arange(5) --> [0 1 2 3 4]
* arange(3, 8) --> [3 4 5 6 7]
* arange(0, 100, 20) --> [ 0 20 40 60 80]
* ones(5) -->
 [1. 1. 1. 1. 1.]
* ones((3,4)) )-->
 [[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
* zeros(5)--> [0. 0. 0. 0. 0.]
* empty(6)--> [6.23042070e-307 3.56043053e-307 1.37961641e-306 2.22518251e-306
  1.33511969e-306 6.23036978e-307]
* full(5, 33) --> [33 33 33 33 33]
* identity(5)-->
 [[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

넘파이 데이터 형식

- 파이썬 리스트와 달리 넘파이 배열은 데이터 형식을 지정
- 넘파이의 데이터 형식을 dtype으로 표현하는데, 정수, 실수, 불리언, 복소수, 문자열 등 다양한 형식이 지원됨
- 정수형도 8bit, 16bit, 32bit, 64bit 등을 지원하는데, int8, int16, int32, int64 등으로 부름
- 부호가 없는 정수형도 지원하는데, 앞에 u를 붙이면 됨

넘파이 데이터 형식

표 10-2 주로 사용되는 넘파이 데이터 형식

데이터 형식(dtype)	설명
int8, int16, int32, int64	부호있는 정수형(Signed Integer). 8bit, 16bit, 32bit, 64bit의 크기를 갖는다. 차례대로 범위는 -128~127, -32678~32767, 약 -21억~21억, 약 -900경~900경의 범위를 갖는다.
uint8, uint16, uint32, uint64	부호없는 정수형(Unsigned Integer). 8bit, 16bit, 32bit, 64bit의 크기를 갖는다. 차례대로 범위는 0~255, 0~65535, 약 0~42억, 약 0~1800경의 범위를 갖는다.
float16, float32, float64, float128	부동 소수점. 16bit, 32bit, 64bit, 128bit의 크기를 갖는다. 소수점 아래 정밀도에 따른 구분으로, float32는 C언어의 float형과, float64는 C언어의 double형과 같다.
bool	True 또는 False를 갖는다.
string_	영문을 기준으로 문자열을 표현한다.
unicode_	한글 등을 기준으로 문자열을 표현한다.

넘파이 데이터 형식

- 넘파이 배열을 생성하면서 데이터 형식(dtype)을 지정할 수 있음
- 이미 지정된 데이터 형식을 astype() 함수로 변경할 수도 있음

Code10-04.py

```
01   import numpy as np
02   import random
03
04   pList1 = [10, 20, 30, 40]
05   pList2 = [10, 20, 30, 40.0]
06
07   numpyAry1 = np.array(pList1)
08   print(numpyAry1, numpyAry1.dtype)
09
10   numpyAry1 = np.array(pList2)
11   print(numpyAry1, numpyAry1.dtype)
12
13   numpyAry2 = np.array(pList1, dtype=np.float32)
14   print(numpyAry2, numpyAry2.dtype)
15
```

넘파이 데이터 형식

- 넘파이 배열을 생성하면서 데이터 형식(dtype)을 지정할 수 있음
- 이미 지정된 데이터 형식을 astype() 함수로 변경할 수도 있음

Code10-04.py

```
16    numpyAry3 = np.arange(5)
17    print(numpyAry3, numpyAry3.dtype)
18
19    numpyAry4 = np.ones(5)
20    print(numpyAry4, numpyAry4.dtype)
21
22    numpyAry5 = np.ones(5, dtype=np.uint8)
23    print(numpyAry5, numpyAry5.dtype)
24
25    numpyAry6 = numpyAry5.astype(np.float16)
26    print(numpyAry6, numpyAry6.dtype)
```

실행 결과

```
[10 20 30 40] int32
[10. 20. 30. 40.] float64
[10. 20. 30. 40.] float32
[0 1 2 3 4] int32
[1. 1. 1. 1. 1.] float64
[1 1 1 1 1] uint8
[1. 1. 1. 1. 1.] float16
```

넘파이 배열 사용 방법

- 배열 크기와 데이터 형식
  - 2×3 크기의 배열을 만듦
  - 배열의 값은 0부터 255 사이의 임의의 값

```
import numpy as np
ary = np.random.randint(0, 255, size=(2,3))
ary
```

실행 결과

```
array([[ 57, 210, 207],
       [208, 143, 116]])
```

넘파이 배열 사용 방법

배열 크기와 데이터 형식

- 배열의 데이터 형식은 dtype 속성으로 알 수 있음
- 배열의 차원은 ndim 속성으로, 배열의 크기는 shape 속성으로 확인할 수 있음

```
import numpy as np
ary = np.random.randint(0, 255, size=(2,3))
ary.dtype
ary.ndim
ary.shape
```

실행 결과

```
int32
2
(2, 3)
```

넘파이 배열 사용 방법

- 산술 연산과 브로드캐스팅
  - 배열에 숫자 100을 빼자 모든 배열 항목 하나하나에 100을 뺀 효과가 나타남
  - 배열끼리 더하면 각 자릿수끼리 더해짐
  - 이를 브로드캐스팅(Broadcasting)이라고 함

```
import numpy as np
ary = np.random.randint(0, 255, size=(2,3))
ary = ary - 100
ary
ary = ary + ary
ary
```

실행 결과

```
array([[-43, 110, 107],
       [108,  43,  16]])
array([[-86, 220, 214],
       [216,  86,  32]])
```

넘파이 배열 사용 방법

산술 연산과 브로드캐스팅

모든 값에 10을 더하는 브로드캐스팅

```
import numpy as np
ary1 = np.array( [[1, 2], [3, 4]] )
ary1 + 10
```

실행 결과

```
array([[11, 12],
       [13, 14]])
```

넘파이 배열 사용 방법

- 산술 연산과 브로드캐스팅
  - 원래 배열과 숫자는 연산이 될 수 없지만, 넘파이가 다음과 같이 숫자를 동일한 크기의 배열로 만든 후에 연산을 함

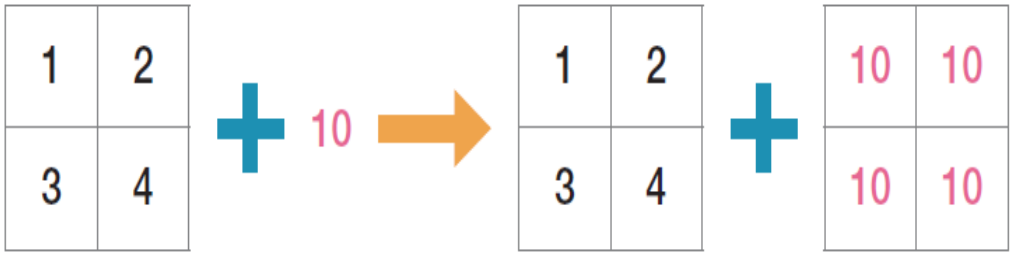


그림 10-2 숫자의 브로드캐스팅

넘파이 배열 사용 방법

산술 연산과 브로드캐스팅

배열의 크기가 다른 경우의 연산

```
import numpy as np
ary1 = np.array( [[1, 2], [3, 4]] )
ary2 = np.array( [ 100, 200] )
ary1 + ary2
```

실행 결과

```
array([[101, 202],
       [103, 204]])
```

넘파이 배열 사용 방법

- 산술 연산과 브로드캐스팅
  - 앞 배열의 크기 2×2에 맞춰서 뒤쪽 배열도 2×2 크기로 확장된 후 연산을 진행함

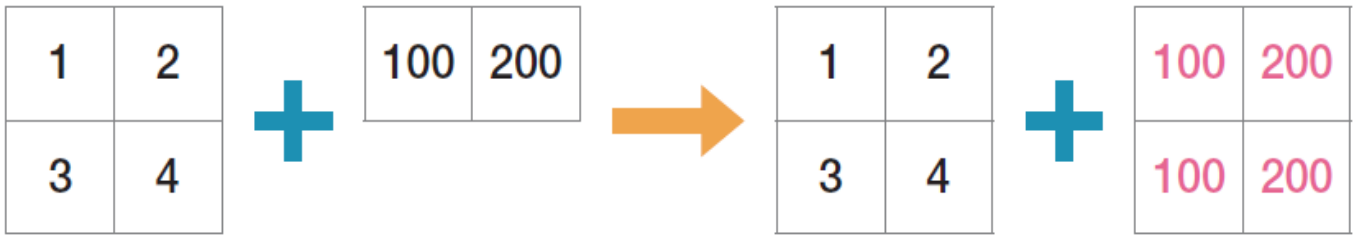


그림 10-3 배열의 브로드캐스팅

넘파이 배열 사용 방법

산술 연산과 브로드캐스팅

- 배열을 숫자와 나누거나 제곱해도 됨

```
import numpy as np
ary = np.random.randint(0, 255, size=(2,3))
ary = 1/ary
ary
ary = ary ** 5
ary
```

실행 결과

```
array([[-0.01162791,  0.00454545,  0.0046729 ],
       [ 0.00462963,  0.01162791,  0.03125   ]])
array([[-2.12572825e-10,  1.94037913e-12,  2.22808181e-12],
       [ 2.12682249e-12,  2.12572825e-10,  2.98023224e-08]])
```

넘파이 배열 사용 방법

첨자와 슬라이싱

- 파이썬의 리스트와 마찬가지로 인덱스를 통해서 처리할 수 있음

```
import numpy as np
ary = np.arange(10)
ary
ary[0] = 100
ary
```

실행 결과

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
array([100,   1,   2,   3,   4,   5,   6,   7,   8,   9])
```

넘파이 배열 사용 방법

- 첨자와 슬라이싱
  - 배열명[첨자]를 사용하여 값에 접근할 수 있음
  - 배열의 부분집합은 배열명[시작:끝+1] 형식으로 파이썬 리스트와 마찬가지로 조작 가능
  - 첨자를 –1, -2, … 방식으로 사용하면 마지막 위치부터 접근, 배열명[:]은 배열 자체를 의미

```
import numpy as np
ary = np.arange(10)
ary[0] = 100
ary[2:5]
ary[-1]
ary[-2]
ary[:]
```

실행 결과

```
array([2, 3, 4])
9
8
array([100,   1,   2,   3,   4,   5,   6,   7,   8,   9]
```

넘파이 배열 사용 방법

첨자와 슬라이싱

- 슬라이싱 방식으로 부분집합을 먼저 추출한 후 그 값을 변경하는 코드

```
import numpy as np
ary = np.arange(10)
ary[0] = 100
ary
ary_sub1 = ary[3:7]
ary_sub1
ary_sub1[:] = 77
ary_sub1
ary
```

실행 결과

```
array([100,   1,   2,   3,   4,   5,   6,   7,   8,   9])
array([3, 4, 5, 6])
array([77, 77, 77, 77])
array([100,   1,   2,  77,  77,  77,  77,   7,   8,   9])
```

넘파이 배열 사용 방법

- 첨자와 슬라이싱
  - 실행 결과에 따르면 배열의 부분집합을 추출했을 때 부분집합에 새로운 값이 복사되는 것이 아니라, 부분집합이 원래 배열과 메모리를 공유하는 것을 알 수 있음
  - 따라서 부분집합의 값을 변경했는데 원래 배열의 값이 변경되었음

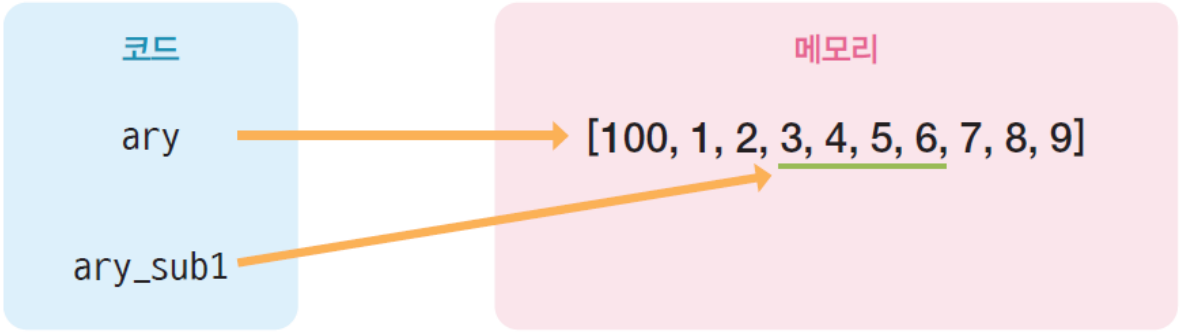


그림 10-4 원래 배열을 공유하는 부분집합 1

- **넘파이 배열 사용 방법**
  - 첨자와 슬라이싱
    - ary_sub1을 모두 77로 변경하면 [그림 10-5]와 같이 변경되기 때문에 ary를 출력하면 77이 보이는 것

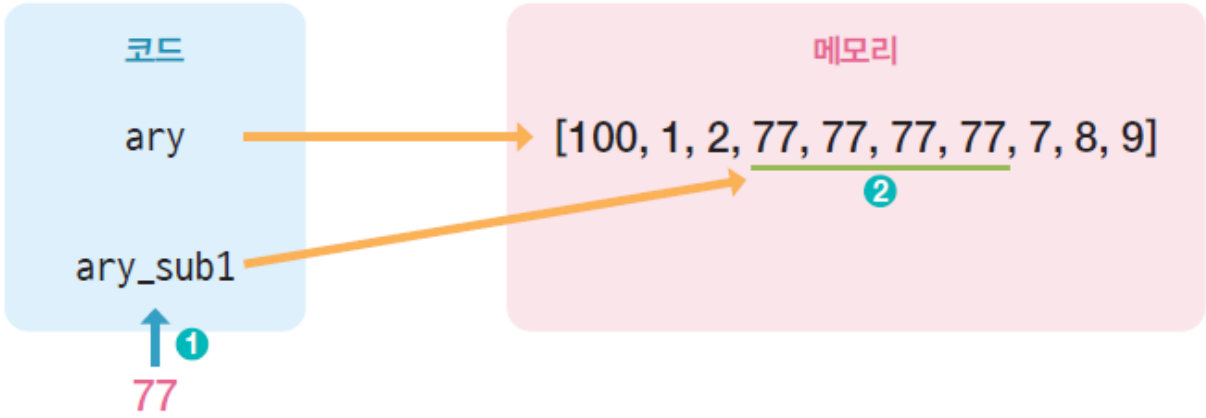


그림 10-5 원래 배열을 공유하는 부분집합 2

넘파이 배열 사용 방법

첨자와 슬라이싱

- 완전히 분리된 부분집합을 만들고 싶다면 부분집합을 만들 때 복사하는 방식을 사용해야 함

```
import numpy as np
ary = np.arange(10)
ary_sub2 = ary[3:7].copy()
ary_sub2
ary_sub2[:] = 88
ary_sub2
ary
```

실행 결과

```
array([3, 4, 5, 6])
array([88, 88, 88, 88])
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

넘파이 배열 사용 방법

첨자와 슬라이싱

- 이전 코드의 세 번째 행은 [그림 10-6]과 같이 새로운 공간에 복사되어 ary_sub2가 만들어짐

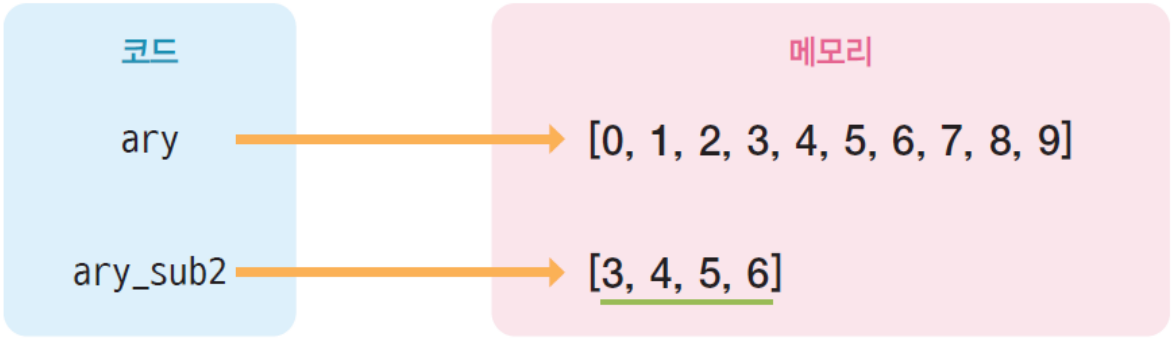


그림 10-6 부분집합을 복사해서 만든 경우 1

- **넘파이 배열 사용 방법**
  - 첨자와 슬라이싱
    - [그림 10-7]과 같이 ary_sub2를 변경해도 원 배열인 ary는 변경되지 않았음

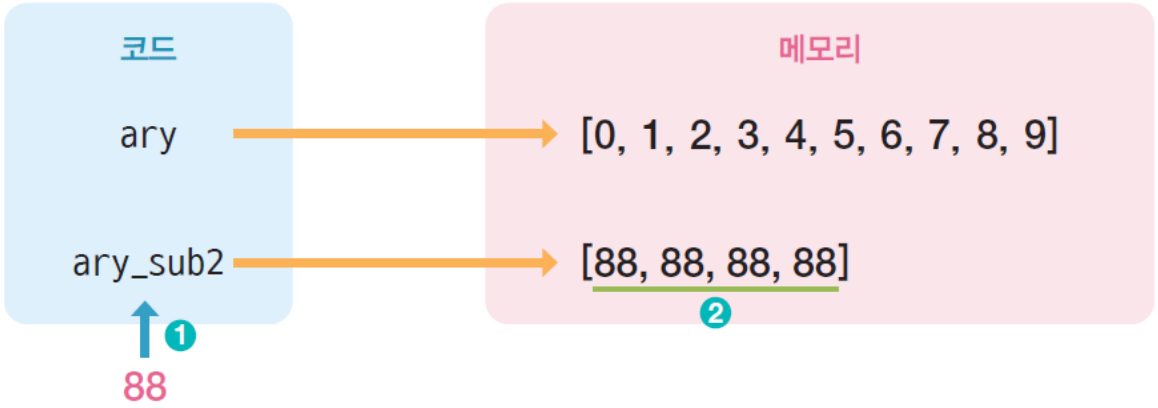


그림 10-7 부분집합을 복사해서 만든 경우 2

넘파이 배열 사용 방법

- 파이썬 리스트와 넘파이 배열의 슬라이싱 비교
  - 파이썬 리스트에서 2차원 배열 일부를 추출하기 위해서는 다소 복잡한 과정을 거침
  - [그림 10-8]과 같이 2차원 리스트 일부를 추출하는 코드를 작성

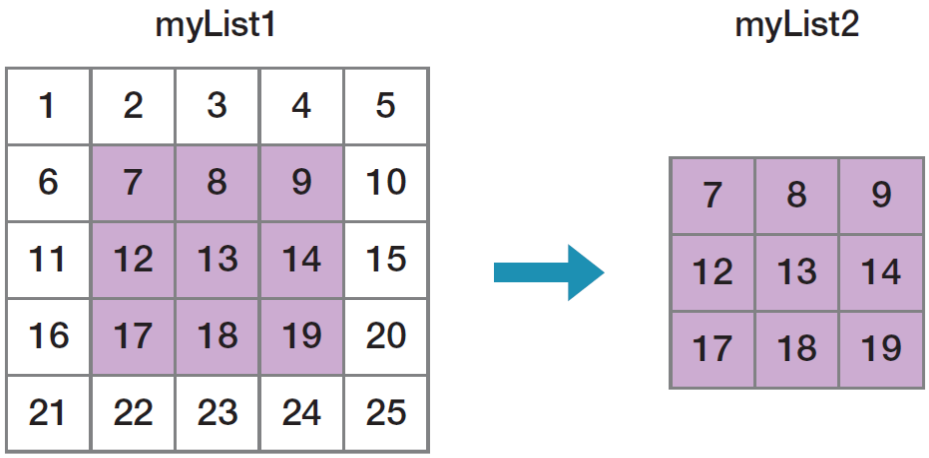


그림 10-8 2차원 리스트 추출

넘파이 배열 사용 방법

- 파이썬 리스트와 넘파이 배열의 슬라이싱 비교

Code10-05.py

```
01  import random
02  SIZE = 5    # 원본 크기
03  startRow,startCol = 1, 1 # 새로운 리스트의 시작 위치
04  nSIZE = 3   # 새로운 리스트의 크기
05
06  ## 파이썬 2차원 리스트 생성
07  value = 1
08  myList1 = []
09  for _ in range(SIZE) :
10      tmpList= []
11      for _ in range(SIZE) :
12          tmpList.append(value)
13          value += 1
14      myList1.append(tmpList)
15
16  ## 파이썬 2차원 리스트의 출력
17  for i in range(SIZE) :
18      [ print("%3d" % myList1[i][k], end=' ') for k in range(SIZE) ]
19      print()
```

넘파이 배열 사용 방법

- 파이썬 리스트와 넘파이 배열의 슬라이싱 비교

Code10-05.py

```
20    print()
21
22    ## 파이썬 2차원 리스트의 슬라이싱
23    myList2 = []
24    for i in range(startRow, startRow+nSIZE) :
25        tmpList = []
26        for k in range(startCol, startCol+nSIZE) :
27            tmpList.append(myList1[i][k])
28        myList2.append(tmpList)
29
30    ## 파이썬 2차원 리스트의 출력
31    for i in range(nSIZE) :
32        [ print("%3d" % myList2[i][k], end=' ') for k in range(nSIZE) ]
33        print()
34    print()
```

실행 결과

```
  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

  7   8   9
 12  13  14
 17  18  19
```

넘파이 배열 사용 방법

- 파이썬 리스트와 넘파이 배열의 슬라이싱 비교
  - 넘파이 배열의 슬라이싱 기능을 사용하면 훨씬 간단함
  - 1차원 배열을 2차원 배열로 만드는 reshape()의 기능을 사용

```
import numpy as np
ary1 = np.arange(1,26,1)
ary2 = ary1.reshape(5,5)
ary2
```

실행 결과

```
array([[ 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]])
```

넘파이 배열 사용 방법

- 파이썬 리스트와 넘파이 배열의 슬라이싱 비교
  - '1차원배열.reshape(행,열)'을 사용하면 1차원 배열이 간단히 2차원 배열로 변경됨
  - 2차원 배열도 '2차원배열.reshape(크기)'를 사용하면 1차원 배열로 변경됨

```
import numpy as np
ary1 = np.arange(1,26,1)
ary2 = ary1.reshape(5,5)
ary3 = ary2.reshape(25)
ary3
```

실행 결과

```
array([ 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])
```

- **넘파이 배열 사용 방법**
  - 2차원 넘파이 배열의 슬라이싱
    - 다양한 2차원 넘파이 배열의 슬라이싱 방법

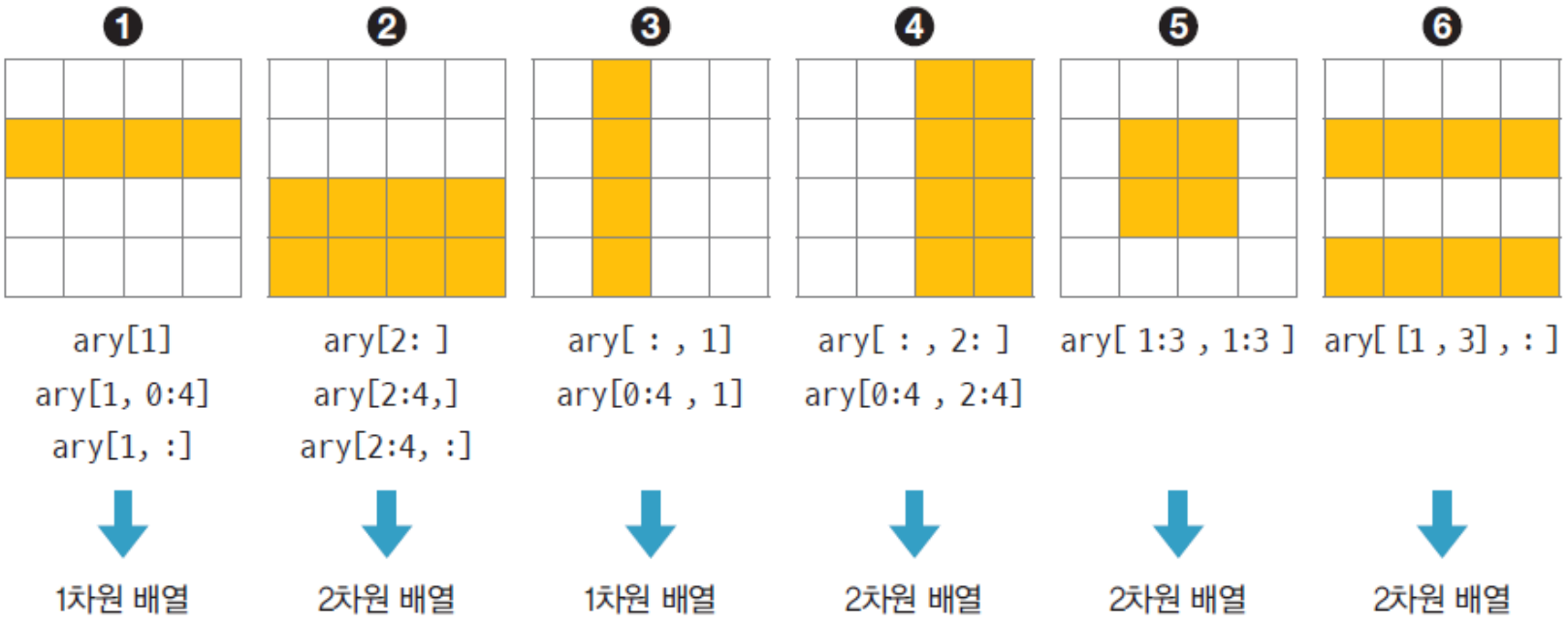


그림 10-9 2차원 넘파이 배열의 슬라이싱

넘파이 배열 사용 방법

- 2차원 넘파이 배열의 슬라이싱
  - [그림 10-9]의 4×4 크기의 2차원 배열에서 행을 추출하는 다양한 방법
  - 먼저 간단히 4×4 크기의 배열을 준비

```
import numpy as np
ary = np.arange(1,17)
ary = ary.reshape(4,4)
ary
```

실행 결과

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

넘파이 배열 사용 방법

- 2차원 넘파이 배열의 슬라이싱
  - **1개 행을 추출할 경우**
    - 2차원 넘파이 배열에서 슬라이싱을 할 때는 원칙적으로 [행, 열] 형식을 사용
    - 하지만 행만 표시한다면 열은 모든 열인 ':' 또는 '0:열개수'를 생략한 것으로 간주
    - 2차원 배열에서 행 1개만 추출했으므로 결과는 1차원 배열

```
import numpy as np
ary = np.arange(1,17)
ary = ary.reshape(4,4)
ary[1]
ary[1, 0:4]
ary[1, :]
```

실행 결과

```
array([5, 6, 7, 8])
array([5, 6, 7, 8])
array([5, 6, 7, 8])
```

넘파이 배열 사용 방법

- 2차원 넘파이 배열의 슬라이싱
  - **인접한 여러 개 행을 추출하는 경우**
    - 여러 개 행을 추출할 때는 행의 범위를 설정
    - 열 부분은 생략하거나 모든 열로 지정할 수 있음
    - 결과는 2차원 배열이 됨

```
import numpy as np
ary = np.arange(1,17)
ary = ary.reshape(4,4)
ary[2: ]
ary[2:4,]
ary[2:4, :]
```

실행 결과

```
array([[ 9, 10, 11, 12],
       [13, 14, 15, 16]])
array([[ 9, 10, 11, 12],
       [13, 14, 15, 16]])
array([[ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

넘파이 배열 사용 방법

- 2차원 넘파이 배열의 슬라이싱
  - **한 개 열을 추출하는 경우**
    - 한 개 열을 추출할 때는 [행, 열]에서 행 부분은 ':' 또는 '0:행개수'로 지정
    - 단 [, 열] 방식으로 앞부분을 비워 놓으면 오류가 발생

```
import numpy as np
ary = np.arange(1,17)
ary = ary.reshape(4,4)
ary[ : , 1]
ary[ 0:4 , 1]
ary [ , 1]     # 오류
```

실행 결과

```
array([ 2,  6, 10, 14])
array([ 2,  6, 10, 14])
```

- **넘파이 배열 사용 방법**
- 2차원 넘파이 배열의 슬라이싱
  - **인접한 여러 개 열을 추출하는 경우**
    - 여러 개 열을 추출할 때는 열의 범위를 설정
    - 행 부분은 모든 열로 지정

```
import numpy as np
ary = np.arange(1,17)
ary = ary.reshape(4,4)
ary[ : , 2:]
ary[0:4, 2:4]
```

실행 결과

```
array([[ 3,  4],
       [ 7,  8],
       [11, 12],
       [15, 16]])
array([[ 3,  4],
       [ 7,  8],
       [11, 12],
       [15, 16]])
```

넘파이 배열 사용 방법

- 2차원 넘파이 배열의 슬라이싱
  - **중간 부분을 추출하는 경우**
    - 행 및 열의 범위를 설정하면 되며 결과는 2차원 배열이 됨

```
import numpy as np
ary = np.arange(1,17)
ary = ary.reshape(4,4)
ary [ 1:3 , 1:3]
```

실행 결과

```
array([[ 6,  7],
       [10, 11]])
```

넘파이 배열 사용 방법

- 2차원 넘파이 배열의 슬라이싱
  - **인접하지 않은 행 또는 열을 추출하는 경우**
    - [[행, 열]] 방식으로 추출
    - 행 및 열을 1개씩만 써도 되고, 범위를 지정해도 됨

```
import numpy as np
ary = np.arange(1,17)
ary = ary.reshape(4,4)
ary[ [ 1 , 3], : ]
ary[ : , [1, 3] ]
```

실행 결과

```
array([[ 5,  6,  7,  8],
       [13, 14, 15, 16]])
array([[ 2,  4],
       [ 6,  8],
       [10, 12],
       [14, 16]])
```

넘파이 배열 사용 방법

- 2차원 넘파이 배열 연결 및 축 전환
  - 2차원 넘파이 배열을 연결할 때는 np.concatenate((배열1, 배열2)) 함수를 사용
  - 옵션 axis를 통해서 0은 세로로, 1을 가로로 연결

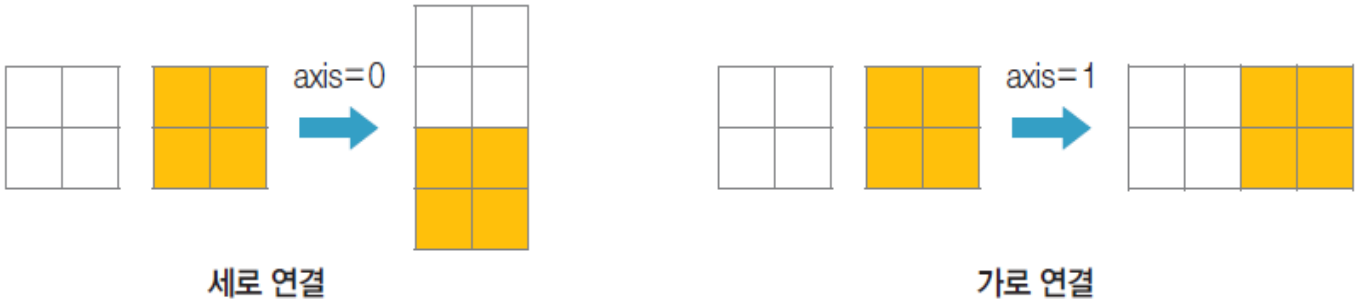


그림 10-10 넘파이 배열의 연결

- **넘파이 배열 사용 방법**
- 2차원 넘파이 배열 연결 및 축 전환

```
ary1 = np.zeros( (2, 2), dtype=np.int8)
ary2 = np.ones( (2, 2), dtype=np.int8)
ary1
ary2
np.concatenate( (ary1, ary2 ) , axis = 0)
np.concatenate( (ary1, ary2 ) , axis = 1)
```

실행 결과

```
array([[0, 0],
       [0, 0]], dtype=int8)
array([[1, 1],
       [1, 1]], dtype=int8)
array([[0, 0],
       [0, 0],
       [1, 1],
       [1, 1]], dtype=int8)
array([[0, 0, 1, 1],
       [0, 0, 1, 1]], dtype=int8)
```

- **넘파이 배열 사용 방법**
- 2차원 넘파이 배열 연결 및 축 전환
  - 넘파이 배열의 행과 열을 바꾸려면 transpose(1, 0) 또는 T 속성을 사용
  - transpose (1, 0)에서 1은 1번 축을, 0은 0번 축을 가리키는데, 두 축을 바꾸기 때문에 행과 열이 바뀜
  - transpose(0, 1)은 축이 바뀌지 않으므로 그대로 출력됨

- **넘파이 배열 사용 방법**
- 2차원 넘파이 배열 연결 및 축 전환

```
import numpy as np
ary = np.array( [ [ 1, 2, 3] , [ 4, 5, 6 ] ] )
ary
ary.transpose (1, 0)
ary.T
ary.transpose (0, 1)
```

실행 결과

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

- **[프로그램 1] 완성**
  - 넘파이 배열로 Code10-05.py와 동일한 결과를 내는 코드

Code10-06.py

```
01    import numpy as np
02    SIZE = 5     # 원본 크기
03    startRow,startCol = 1, 1 # 새로운 리스트의 시작 위치
04    nSIZE = 3    # 새로운 리스트의 크기
05
06    ## 넘파이 2차원 배열 생성
07    value = 1
08    myAry1 = np.arange(value, value+(SIZE*SIZE), 1)
09    myAry1 = myAry1.reshape(SIZE, SIZE)
10
11    ## 넘파이 2차원 배열 리스트의 출력
…     ~~ Code10-05.py의 17~20행과 동일. 단 myAry1을 출력. ~~
16
17    ## 넘파이 2차원 배열의 슬라이싱
18    myAry2 = myAry1[startRow:startRow+nSIZE, startCol: startCol+nSIZE].copy()
19
20    ## 넘파이 2차원 배열의 출력
…     ~~ Code10-05.py의 31~34행과 동일. 단 myAry2를 출력. ~~
```

- [프로그램 1] 완성

SELF STUDY 10-2

Code10-06.py을 참고하여 2차원 배열의 크기가 8×8, 12×12, 16×16, 20×20 중 하나가 랜덤하게 선택된 후, 중앙부터 절반 크기의 배열로 만들자. 예로 8×8은 4×4(행4~7, 열4~7)로 생성한다.

힌트 SIZE = np.random.choice([8, 12, 16, 20])으로 크기를 랜덤하게 구한다.

실행 결과

```
 1  2  3  4  5  6  7  8
 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32
33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48
49 50 51 52 53 54 55 56
57 58 59 60 61 62 63 64

37 38 39 40
45 46 47 48
53 54 55 56
61 62 63 64
```

수식 계산 함수

단항 수식 함수

- 넘파이 배열 1개를 연산

```
import numpy as np
ary = np.array( [ [ -1, -2 ] , [ 3, 4 ] ] )
ary
np.abs(ary)
np.sign(ary)
np.sqrt(ary)   # 경고 발생
```

실행 결과

```
array([[-1, -2],
       [ 3,  4]])
array([[1, 2],
       [3, 4]])
array([[-1, -1],
       [ 1,  1]])
array([[       nan,      nan],
       [1.73205081, 2.      ]])
```

수식 계산 함수

단항 수식 함수

- np.abs()는 모두 양수를 반환하고, sign()의 경우 양수는 0을, 음수는 1을 반환
- np.sqrt()는 루트값을 반환하는데, 음수의 루트값은 nan(Not a Number)로 표시됨

표 10-3 주로 사용되는 단항 수식 함수

넘파이 함수이름	설명
np.square( )	각 항목의 제곱값을 계산한다.
np.log( ), np.log2( ), np.log10( )	각 항목의 로그값을 계산한다.
np.ceil( ), np.floor( )	각 항목의 소수를 올림, 내림한다.
np.rint( )	각 항목을 반올림한다.
np.cos( ), np.sin( ), np.tan( )	삼각함수를 계산한다.
np.logical_not( )	각 항목의 부정(not) 값을 계산한다.

수식 계산 함수

이항 수식 함수

- 넘파이 배열 2개를 연산
- 더하기(add), 빼기(subtract), 곱하기(multiply) 등이 가능함

```
import numpy as np
ary1 = np.array( [ [ 1, 2 ] , [ 3, 4 ] ] )
ary2 = np.array( [ [ 5, 6 ] , [ 7, 8 ] ] )
np.add(ary1, ary2)
ary1 + ary2
```

실행 결과

```
array([[ 6,  8],
       [10, 12]])
```

- **수식 계산 함수**
  - **이항 수식 함수**
    - np.add(배열1, 배열2)는 배열1+배열2와 결과가 동일함
    - 따라서, 함수를 사용하는 것과 직접 연산자를 사용하는 것은 차이가 없음
    - np.maximum(배열1, 배열2)는 두 배열 중 큰 값만 추출해서 새 배열을 반환

수식 계산 함수

이항 수식 함수

```
import numpy as np
ary1 = np.random.randint(0, 255, size=(3, 3))
ary1
ary2= np.random.randint(0, 255, size=(3, 3))
ary2
np.maximum(ary1, ary2)
```

실행 결과

```
array([[244, 211, 138],
       [ 29, 237, 104],
       [ 73, 238,  68]])
array([[  5, 203, 179],
       [ 85, 250,  90],
       [175, 181, 192]])
array([[244, 211, 179],
       [ 85, 250, 104],
       [175, 238, 192]])
```

수식 계산 함수

이항 수식 함수

- np.greater(배열1, 배열2)는 '배열1 > 배열2'이 경우엔 True, 그렇지 않으면 False로 구성된 새 배열을 반환

```
import numpy as np
ary1 = np.random.randint(0, 255, size=(3, 3))
ary2= np.random.randint(0, 255, size=(3, 3))
np.greater(ary1, ary2)
```

실행 결과

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

수식 계산 함수

이항 수식 함수

표 10-4 주로 사용되는 이항 수식 함수

함수이름	설명
np.subtract(배열1, 배열2)	배열1에서 배열2를 뺀다. '배열1-배열2'와 동일하다.
np.multiply(배열1, 배열2)	두 배열을 곱한다. '배열1*배열2'와 동일하다.
np.divide(배열1, 배열2)	배열1을 배열2로 나눈다. '배열1/배열2'와 동일하다.
np.power(배열1, 배열2)	배열1을 배열2로 제곱한다. '배열1** 배열2'와 동일하다.
np.minimum(배열1, 배열2)	두 배열 중 작은 값만 추출해서 새 배열을 구한다.
np.mod(배열1, 배열2)	배열1을 배열2로 나눈 나머지 값을 구한다.
np.copysign(배열1, 배열2)	배열1의 부호를 배열2의 부호로 바꾼다.
np.greater_equal, np.less, np.less_equal, np.equal, np.not_equal(배열1, 배열2)	배열1과 배열2를 비교한다. >=, <, <=, ==, !=과 동일하다.

- 기타 넘파이 기능
  - 조건식 표현
    - 넘파이는 파이썬의 if 문처럼 배열을 통째로 조건식으로 표현할 수 있음

```
import numpy as np
ary1 = np.random.randint(0, 255, size=(3, 3))
ary1
ary2= np.random.randint(0, 255, size=(3, 3))
ary2
condAry = np.random.choice([True, False], size=(3, 3))
condAry
np.where(condAry, ary1, ary2)
```

실행 결과

```
array([[212,  19, 121],
       [178, 154,  30],
       [129, 189,  23]])
array([[185, 168, 127],
       [217,  93, 145],
       [231, 189, 183]])
array([[ True, False, False],
       [ True,  True, False],
       [ True,  True, False]])
array([[212, 168, 127],
       [178, 154, 145],
       [129, 189, 183]])
```

기타 넘파이 기능

조건식 표현

- 다음 예시의 np.where(배열 < 0, 0, 배열)은 배열 항목의 값이 음수라면 0으로, 그렇지 않으면 원래 값을 반환

```
import numpy as np
ary = np.random.randint(-128, 127, size=(3, 3))
ary
np.where(ary < 0, 0, ary)
```

실행 결과

```
array([[ 71, -127,  -65],
       [ 20,  -94,   42],
       [-122,   39,  -17]])
array([[71,  0,  0],
       [20,  0, 42],
       [ 0, 39,  0]])
```

- 기타 넘파이 기능
  - 통계 함수
    - 전체 배열의 합(sum), 평균(mean)의 통계치도 구할 수 있음
    - 1차원 배열의 합과 평균을 구하는 간단한 예시

```
import numpy as np
ary1 = np.random.randint(0, 255, size=10)
ary1
ary1.sum()
ary1.mean()
```

실행 결과

```
array([243,   5, 162, 192, 114, 254, 228, 130,   1, 203])
1532
153.2
```

기타 넘파이 기능

통계 함수

- 2차원 배열은 전체의 평균 및 합도 연산할 수 있으며 가로 또는 세로별 평균 및 덧셈 연산도 가능함
- 다음 예시에서 axis=0은 세로의 합을, axis=1은 가로의 합을 구하는 것

```
import numpy as np
ary2 = np.random.randint(0, 255, size=(3, 3))
ary2
ary2.sum()
ary2.sum(axis=0)
ary2.sum(axis=1)
```

실행 결과

```
array([[176, 155,  85],
       [ 77,  14,  47],
       [  8,  57,  81]])
700
array([261, 226, 213])
array([416, 138, 146])
```

- 기타 넘파이 기능
  - 통계 함수
    - 합이나 평균을 구할 때 조건식을 함께 사용할 수 있음
    - 다음 예시는 128 이상인 항목의 개수를 세는 코드

```
import numpy as np
ary2 = np.random.randint(0, 255, size=(3, 3))
ary2
(ary2 > 128).sum()
```

실행 결과

```
array([[176, 155,  85],
       [ 77,  14,  47],
       [  8,  57,  81]])
2
```

- 기타 넘파이 기능
  - 통계 함수
    - 배열에서 min()으로 최솟값을, max()로 최댓값을 찾을 수 있음

```
import numpy as np
ary2 = np.random.randint(0, 255, size=(3, 3))
ary2
ary2.min()
ary2.max()
```

실행 결과

```
array([[176, 155,  85],
       [ 77,  14,  47],
       [  8,  57,  81]])
8
176
```

- 기타 넘파이 기능
  - 정렬
    - np.sort() 함수는 배열을 정렬
    - 기본적으로 오름차순으로 정렬하는데, 내림차순으로 정렬하려면 정렬결과에 [::-1]을 붙이면 됨

```
import numpy as np
ary = np.random.randint(0, 255, size=10)
ary
np.sort(ary)
np.sort(ary)[::-1]
```

실행 결과

```
array([ 97, 172, 132,  63, 153, 111, 197,  60, 130,  84])
array([ 60,  63,  84,  97, 111, 130, 132, 153, 172, 197])
array([197, 172, 153, 132, 130, 111,  97,  84,  63,  60])
```

- 기타 넘파이 기능
  - 중복값 제거와 공통값 추출
    - 중복된 값을 제거하고 하나씩만 남기려면 np.unique() 함수를 사용
    - 중복이 제거된 항목들이 정렬되어 반환됨

```
import numpy as np
ary = np.random.randint(0, 9, size=10)
ary
np.unique(ary)
```

실행 결과

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

- 기타 넘파이 기능
  - 중복값 제거와 공통값 추출
    - 두 배열 중에서 공통된 항목만 추출하려면 np.intersect1d(배열1, 배열2) 함수를 사용
    - 공통된 항목들이 정렬되어 반환됨

```
import numpy as np
ary1 = np.random.randint(0, 9, size=10)
ary1
ary2 = np.random.randint(0, 9, size=10)
ary2
np.intersect1d(ary1,ary2)
```

실행 결과

```
array([4, 0, 0, 7, 3, 4, 1, 8, 3, 5])
array([8, 5, 0, 1, 5, 3, 4, 8, 5, 3])
array([0, 1, 3, 4, 5, 8])
```

- **[프로그램 2] 완성**
  - 대부분의 소프트웨어에서 작업을 진행하다가, Ctrl+Z를 누르면 직전 값으로 되돌리는 과정을 단순화해서 구현한 프로그램

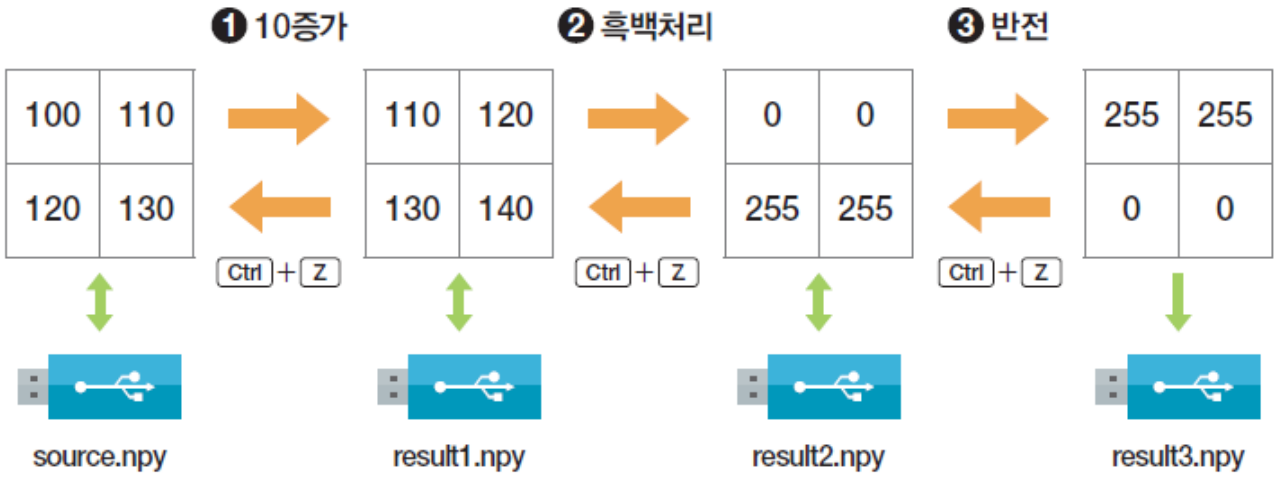


그림 10-11 파일 입출력 흐름도

- [프로그램 2] 완성

Code10-07.py

```
01    import numpy as np
02    SIZE = 5 # 원본 크기
03
04    ## 넘파이 2차원 배열 생성
05    imageAry = np.random.randint(0, 255, size=(SIZE, SIZE))
06    print('### 1. 원본 ###')
07    print(imageAry)
08    np.save('source', imageAry)
09
10    ## (1) 10 증가후 저장
11    imageAry += 10
12    print('### 2. 10 증가 ###')
13    print(imageAry)
14    np.save('result1', imageAry)
15
16    ## (2) 흑백 처리후 저장
17    imageAry = np.where( imageAry<128, 0, 255)
18    print('### 3. 흑백 처리 ###')
19    print(imageAry)
20    np.save('result2', imageAry)
21
```

- [프로그램 2] 완성

Code10-07.py

```
22  ## (3) 반전 처리후 저장
23  imageAry = 255 - imageAry
24  print('### 4. 반전 처리 ###')
25  print(imageAry)
26  np.save('result3', imageAry)
27
28  ## 복구1 ##
29  imageAry = np.load('result2.npy')
30  print('### 복구1 : result2.npy ###')
31  print(imageAry)
32
33  ## 복구2 ##
34  imageAry = np.load('result1.npy')
35  print('### 복구2 : result1.npy ###')
36  print(imageAry)
37
38  ## 복구3 ##
39  imageAry = np.load('source.npy')
40  print('### 복구3(원본) : source.npy ###')
41  print(imageAry)
```

- [프로그램 2] 완성

실행 결과

```
### 1. 원본 ###
[[106  11 207 101  20]
 [ 30   4 166 251  29]
 [205  56 141 116  72]
 [ 59  87 227 167 175]
 [ 24 237  54 194  57]]

### 2. 10 증가 ###
[[116  21 217 111  30]
 [ 40  14 176 261  39]
 [215  66 151 126  82]
 [ 69  97 237 177 185]
 [ 34 247  64 204  67]]

### 3. 흑백 처리 ###
[[  0   0 255   0   0]
 [  0   0 255 255   0]
 [255   0 255   0   0]
 [  0   0 255 255 255]
 [  0 255   0 255   0]]

### 4. 반전 처리 ###
[[255 255   0 255 255]
 [255 255   0   0 255]
 [  0 255   0 255 255]
 [255 255   0   0   0]
 [255   0 255   0 255]]

### 복구1 : result2.npy ###
[[  0   0 255   0   0]
 [  0   0 255 255   0]
 [255   0 255   0   0]
 [  0   0 255 255 255]
 [  0 255   0 255   0]]

### 복구2 : result1.npy ###
[[116  21 217 111  30]
 [ 40  14 176 261  39]
 [215  66 151 126  82]
 [ 69  97 237 177 185]
 [ 34 247  64 204  67]]

### 복구3(원본) : source.npy ###
[[106  11 207 101  20]
 [ 30   4 166 251  29]
 [205  56 141 116  72]
 [ 59  87 227 167 175]
 [ 24 237  54 194  57]]
```

- [프로그램 2] 완성

SELF STUDY 10-3

0부터 255 사이의 값을 갖는 5,000×5,000 크기의 임의의 넘파이 배열을 생성하자. 그리고 np.save( ), np.savez( ), np.savez_compressed( ) 등 3가지 저장 파일의 크기를 비교해 보자.

힌트 파일의 크기는 os.path.getsize(파일명)으로 구한다.

실행 결과

```
### 1. np.save()로 저장한 파일 크기 --> 100000128
### 2. np.savez()로 저장한 파일 크기 --> 100000264
### 3. np.savez_compressed()로 저장한 파일 크기 --> 38328301
```