파이썬으로 배우는
머신러닝의 교과서

Contents

- CHAPTER 02 파이썬 기본
- SECTION 01 사칙 연산
  - 1.1 사칙 연산의 이용
  - 1.2 제곱
- SECTION 02 변수
  - 2.1 변수를 이용한 계산
  - 2.2 변수명의 표기
- SECTION 03 자료형
  - 3.1 자료형의 종류
  - 3.2 type으로 자료형 알아보기
  - 3.3 문자열

Contents

◦ SECTION 04 print 문

- 4.1 print 문의 이용
- 4.2 문자열과 수치를 함께 표시 1
- 4.3 문자열과 수치를 함께 표시 2

◦ SECTION 05 list

- 5.1 list의 이용
- 5.2 2차원 배열
- 5.3 list의 길이
- 5.4 연속된 정수 데이터의 작성

◦ SECTION 06 tuple

- 6.1 tuple의 이용
- 6.2 요소의 참조
- 6.3 길이가 1인 tuple

Contents

- SECTION 07 if 문
  - 7.1 if 문의 사용
  - 7.2 비교 연산자
- SECTION 08 for 문
  - 8.1 for 문의 이용
  - 8.2 enumerate의 이용
- SECTION 09 벡터
  - 9.1 넘파이의 이용
  - 9.2 벡터의 정의
  - 9.3 요소의 참조
  - 9.4 요소의 수정
  - 9.5 연속된 정수 벡터의 생성
  - 9.6 ndarray 형의 주의점

Contents

◦ SECTION 09 벡터

- 9.1 넘파이의 이용
- 9.2 벡터의 정의
- 9.3 요소의 참조
- 9.4 요소의 수정
- 9.5 연속된 정수 벡터의 생성
- 9.6 ndarray 형의 주의점

◦ SECTION 10 행렬

- 10.1 행렬의 정의
- 10.2 행렬의 크기
- 10.3 요소의 참조
- 10.4 요소의 수정
- 10.5 요소가 0과 1인 ndarray 만들기

Contents

- 10.6 요소가 랜덤인 행렬 생성
- 10.7 행렬의 크기 변경

◦ SECTION 11 행렬(ndarray)의 사칙 연산

- 11.1 행렬의 사칙 연산
- 11.2 스칼라 × 행렬
- 11.3 산술 함수
- 11.4 행렬 곱의 계산

◦ SECTION 12 슬라이싱

- 12.1 슬라이싱의 이용

◦ SECTION 13 조건을 만족하는 데이터의 수정

- 13.1 bool 배열 사용

◦ SECTION 14 Help

- 14.1 Help 사용

Contents

◦ SECTION 15 함수

- 15.1 함수의 사용
- 15.2 인수와 반환값

CHAPTER 02 파이썬 기본

머신러닝을 이해하는 데 필요한 최소한의 프로그래밍 지식을 알아본다

SECTION 01 사칙 연산

1.1 사칙 연산의 이용

- 주피터 노트북 셀에 다음 내용을 입력하여 Shift + Enter 키를 누르면 답이 나옴.

```
In  1 + 2
Out 3
```

- 사칙 연산은 다른 대부분의 프로그램 언어와 마찬가지로, +, -, *, /를 사용함.

```
In  (1 + 2 * 3 - 4) / 5
Out 0.6
```

1.2 제곱

- 제곱은 **로 나타냅니다. 예를 들어, 28은 다음과 같음.

```
In  2 ** 8
Out 256
```

SECTION 02 변수

2.1 변수를 이용한 계산

- 알파벳을 사용하여 변수를 나타낼 수 있음.
- 변수에는 값을 저장할 수 있기 때문에 이를 사용하여 계산할 수 있음.

In
```
x = 1
y = 1 / 3
x + y
```

Out
```
1.3333333333333333
```

2.2 변수명의 표기

- 변수명은 Data_1, Data_2와 같이 여러 문자열로 나타낼 수 있음.
- 변수명에는 알파벳, 숫자, 밑줄(_)을 사용할 수 있음. 대문자와 소문자를 구별함.

In
```
Data_1 = 1 / 5
Data_2 = 3 / 5
Data_1 + Data_2
```

Out
```
0.8
```

SECTION 03 자료형

3.1 자료형의 종류

- 파이썬에서 사용할 수 있는 데이터에는 정수와 실수, 문자열 등 다양한 종류가 있고 이들을 자료형이라고 부음.
- 파이썬에서 다룰 수 있는 주요 변수의 형태를 정리하면 다음과 같음.

자료형	사용 예	자료형의 의미
int 형	a=1	정수
float 형	a=1.5	실수
str 형	a='learning' 또는 b='abc'	문자열
bool 형	True 또는 False	참과 거짓
list 형	a=[1, 2, 3]	배열
tuple 형	a=(1, 2, 3) 또는 b=(2,)	배열(수정 불가능)
ndarray 형	a=np.array([1, 2, 3])	행렬

[변수의 자료형]

3.2 type으로 자료형 알아보기

- 자료형을 조사하려면 type을 사용함. 다음과 같이 입력하여 결과를 확인할 수 있음.

```
In  type(100)
```

```
Out int
```

```
In  type(100.1)
```

```
Out float
```

- 100은 int 형, 100.1은 float 형으로 취급하는 것을 알 수 있음.

```
In  x = 100
    type(x)
```

```
Out int
```

```
In  x = 100.1
    type (x)
```

```
Out float
```

- int 형 데이터를 변수에 넣으면 그 변수는 자동으로 int 형 변수가 되고, float 형 데이터를 넣으면 float 형 변수가 됨.

3.3 문자열

- 문자열을 다루는 데에는 str 형이 사용됨.
- 작은따옴표(') 또는 큰따옴표(")로 둘러싸면 문자열로 인식함.

In
```
x = 'learning'
type(x)
```

Out
```
str
```

SECTION 04 print 문

4.1 print 문의 이용

- 변수명을 입력해 실행하는 것만으로 그 내용이 표시되지만, 셀의 마지막 행에 적어야만 됨.
- 마지막 행 외에도 다른 변수를 함께 표시할 경우에는 print를 사용함.

In

```
x = 1 / 3
print(x)
y = 2 / 3
print(y)
```

Out

```
0.3333333333333333
0.6666666666666666
```

4.2 문자열과 수치를 함께 표시 1

- 문자열과 함께 수치를 표시하고 싶은 경우에는 다음처럼 표현함.

In
```
print('x= ' + str(x))
```

Out
```
x = 0.3333333333333333
```

4.3 문자열과 수치를 함께 표시 2

- 문자열과 수치를 조합하는 다른 방법으로 format을 사용하면 편리함.

In
```
print('weight = {0} kg'.format(x))
```

Out
```
weight = 0.3333333333333333 kg
```

- 여러 변수를 표시할 경우에는 문자열 내에 {0}, {1}, {2}를 지정합니다
- {수치:.nf}를 입력하면 소수점 이하 n자리까지 표시함.

In
```
x = 1 / 3
y = 1 / 7
z = 1 / 4
print ('weight: {0} kg {1} kg {2} kg'.format(x, y, z))
```

Out
```
weight: 0.3333333333333333 kg, 0.14285714285714285 kg 0.25 kg
```

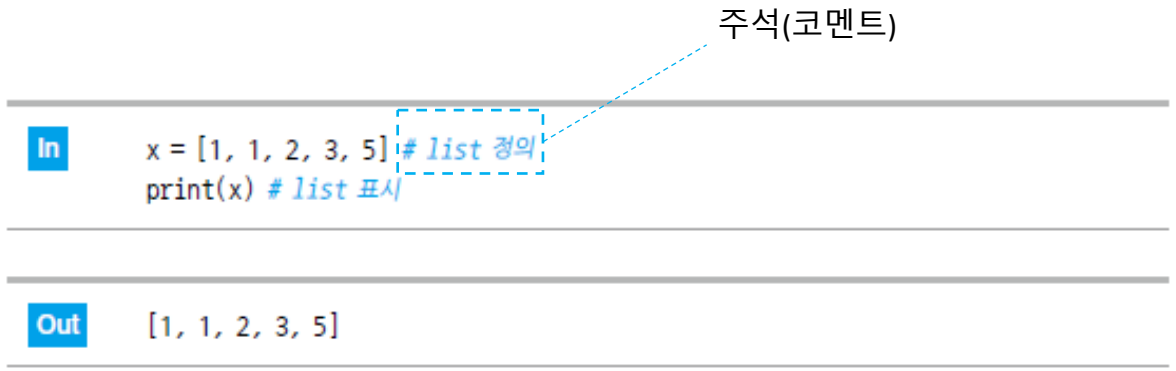
In
```
print('weight: {0:.2f} kg, y={1:.2f} kg, z={2:.2f} kg'.format(x,y,z))
```

Out
```
weight: 0.33 kg, y=0.14 kg, z=0.25 kg
```

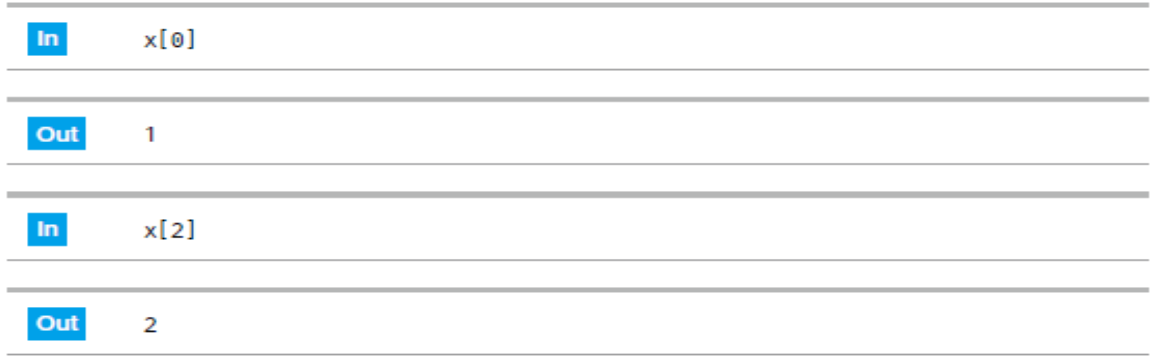
SECTION 05 list

5.1 list의 이용

- 여러 데이터를 하나의 단위로 취급하고 싶은 경우, list (리스트)형을 사용함.
- list는 리스트명[]을 사용하여 나타냅니다.



- 파이썬에서 배열의 요소 번호(인덱스)는 0부터 시작함.



SECTION 05 list

- x는 list 형, x[0]는 int 형인 것을 알 수 있음.
- x는 int 형으로 구성된 list 형이라고 이해할 수 있음. list 형은 str 형으로도 만들 수 있음.

In
```
print(type(x))
print(type(x[0]))
```

Out
```
<class 'list'>
<class 'int'>
```

5.2 2차원 배열

- list 형은 2차원 배열 형태로도 만들 수 있음.
- 3차원 배열, 4차원 배열도 중첩 깊이를 늘려 만들 수 있음.

In
```
a=[[1, 2, 3], [4, 5, 6]]
print(a)
```

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

- list 요소를 수정하려면 리스트명[요소 번호] = 수치로 할 수 있음.

In
```
x=[1, 1, 2, 3, 5]
x[3] = 100
print(x)
```

Out
```
[1, 1, 2, 100, 5]
```

5.3 list의 길이

- list의 길이는 len을 사용해 확인할 수 있음.

```
In    x=[1, 1, 2, 3, 5]
      len(x)
```

```
Out   5
```

5.4 연속된 정수 데이터의 작성

- 5에서 9까지와 같이 연속된 정수 데이터를 만들려면 range(시작숫자, 끝숫자+1)를 사용함.

```
In    y = range(5, 10)
      print(y[0], y[1], y[2], y[3], y[4])
```

```
Out   5 6 7 8 9
```

5.3 list의 길이

- list의 길이는 len을 사용해 확인할 수 있음.

```
In  x=[1, 1, 2, 3, 5]
    len(x)
```

```
Out 5
```

5.4 연속된 정수 데이터의 작성

- 5에서 9까지와 같이 연속된 정수 데이터를 만들려면 range(시작숫자, 끝숫자+1)를 사용함.

```
In  y = range(5, 10)
    print(y[0], y[1], y[2], y[3], y[4])
```

```
Out 5 6 7 8 9
```

SECTION 05 list

• range 형은 list 함수를 사용하여 요소를 수정할 수 있는 list 형으로 변환할 수 있음.

In
```
z=list(range(5, 10))
print(z)
```

Out
```
[5, 6, 7, 8, 9]
```

• 시작숫자를 생략하고 range(끝숫자+1)을 입력하면 0부터 시작되는 수열이 만들어짐.

In
```
list(range(10))
```

Out
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6.1 tuple의 이용

- 배열을 나타내는 형태로 list 형 이외에 tuple (튜플) 형이라는 배열 변수가 있음.
- tuple은 list와는 달리 요소를 수정할 수 없음.
- tuple 형은 **(1, 2, 3)과 같이 괄호()를 사**용하여 배열을 나타냄.

In
```
a=(1, 2, 3)
print(a)
```

Out
```
(1, 2, 3)
```

6.2 요소의 참조

- tuple 형에서 요소를 참조하려면 list 형과 같은 방식을 사용함.
- 참조는 괄호() 대신 list 형처럼 대괄호[]를 사용하는 것에 주의.

```
In    a[1]
```

```
Out   2
```

- tuple 형도 type( )으로 알 수 있음.

```
In    type(a)
```

```
Out   tuple
```

6.3 길이가 1인 tuple

- 길이가 1인 tuple은 (1,)과 같이 쉼표(,)를 붙임.
- (1, 2)는 tuple이지만, (1)은 tuple이 아님.

In
```
a = (1)
type(a)
```

Out
```
int
```

In
```
a = (1,)
type(a)
```

Out
```
tuple
```

SECTION 07 if 문

7.1 if 문의 사용

- 프로그램을 조건에 따라 나누어 실행시키려면 if 문을 사용함.
- 파이썬에서는 들여쓰기에 중요한 의미가 있으므로 주의

In

```
x = 11
if x > 10:
    print('x is ') # ... (A1)
    print('       larger than 10.') # ... (A2)
else:
    print('x is smaller than 11') # ... (B1)
```

Out

```
x is
larger than 10.
```

- 첫 행에서 x 값으로 11을 대입하고 있으므로, if의 x > 10이라는 조건이 참(True). 4칸 오른쪽으로 들여쓰기 된 A1, A2행이 모두 실행.

7.2 비교 연산자

- ' >'를 비교 연산자라고 부름.
- if 문에서 사용하는 비교 연산자를 정리하면 다음과 같으며, 연산의 결과는 모두 bool 형이 됨.

비교 연산자	내용
a == b	a와 b가 같다
a > b	a가 b보다 크다
a >= b	a가 b 이상이다
a < b	a가 b보다 작다
a <= b	a가 b 이하이다
a != b	a와 b는 다르다

- 여러 조건을 사용하려면, and(그리고)와 or(또는)를 사용함.

In
```
x = 15
if 10 <= x and x <= 20:
    print('x is between 10 and 20.')
```

Out
```
x is between 10 and 20.
```

SECTION 08 for 문

8.1 for 문의 이용

- 연산을 반복하려면 for 문을 사용함.

In
```
for i in [1, 2, 3]:
    print(i)
```

Out
```
1
2
3
```

8.2 enumerate의 이용

- enumerate을 사용하여 앞에서 구현한 기능을 좀 더 우아하게 기술할 수 있음.

In
```
num = [2, 4, 6, 8, 10]
for i, n in enumerate(num):
    num[i] = n * 2
print(num)
```

Out
```
[4, 8, 12, 16, 20]
```

SECTION 09 벡터

9.1 넘파이의 이용

- 파이썬으로 벡터나 행렬을 나타내려면 넘파이(NumPy)라는 라이브러리를 통해 기능을 확장.
- import로 간단히 가져올 수 있음.

```
In    import numpy as np
```

9.2 벡터의 정의

- 벡터(1차원 배열)은 np.array(list 형)으로 정의함.
- type(x)를 입력하면 x가 numpy.ndarray 형인 것을 알 수 있음.

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

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

```
In    type(x)

Out   numpy.ndarray
```

9.3 요소의 참조

- 하나의 요소를 참조하려면 list 형과 마찬가지로 대괄호 [ ]를 사용함.

```
In    x[0]
```

```
Out   1
```

9. 4 요소의 수정

- 요소를 수정하려면 x[수정할 요소 번호] = 수치를 사용함.

```
In    x[0] = 100
      print(x)
```

```
Out   [100 2 3]
```

9.5 연속된 정수 벡터의 생성

- np.arange(n)으로 요소의 값이 1씩 증가하는 벡터 배열을 만들 수 있음.

```
In    print(np.arange(10))
```

```
Out   [0 1 2 3 4 5 6 7 8 9]
```

9.6 ndarray 형의 주의점

- ndarray 형의 내용을 복사하려면 일반변수처럼 b = a를 사용하는 것이 아니라 b = a.copy( )를 사용해야 됨.

```
In    a = np.array([1, 1])
      b = a.copy()
      print('a =' + str(a))
      print('b =' + str(b))
      b[0] = 100
      print('b =' + str(b))
      print('a =' + str(a))
```

```
Out   a =[1 1]
      b =[1 1]
      b =[100 1]
      a =[1 1]
```

SECTION 10 행렬

10.1 행렬의 정의

- ndarray의 2차원 배열로 다음과 같이 행렬을 정의할 수 있음.

In
```
x = np.array([[1, 2, 3], [4, 5, 6]])
print(x)
```

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

10.2 행렬의 크기

- 행렬(배열)의 크기는 ndarray변수명.shape 명령으로 알 수 있음.

In
```
x = np.array([[1, 2, 3], [4, 5, 6]])
x.shape
```

Out
```
(2, 3)
```

10.3 요소의 참조

- 요소를 참조하려면 다음과 같이 차원마다 ','로 구분하여 나타냄.

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

Out
```
6
```

10.4 요소의 수정

- 행과 열의 인덱스가 0에서 시작. 다음과 같이 기술하여 요소를 수정함.

In
```
x = np.array([[1, 2, 3], [4, 5, 6]])
x[1, 2] = 100
print(x)
```

Out
```
[[  1   2   3]
 [  4   5 100]]
```

10.5 요소가 0과 1인 ndarray 만들기

- 모든 요소가 0인 ndarray는 np.zeros(size)로 만들 수 있음.

```
In  print(np.zeros(10))

Out [ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

- size=(2, 10)을 사용하면 모든 요소가 0인 2 × 10 크기의 행렬이 생성됨.

```
In  print(np.zeros((2, 10)))

Out [[ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
     [ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

- 요소를 0이 아니라 1로 하려면 np.ones(size)를 사용함.

```
In  print(np.ones((2, 10)))

Out [[ 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
     [ 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]]
```

10.6 요소가 랜덤인 행렬 생성

- 요소가 랜덤(임의)인 행렬을 생성하는 경우에는 np.random.rand(size)를 사용함.
- 각 요소는 0에서 1사이의 균일한 분포를 보이며 행렬 형태로 생성함.
- np.random.randn(size)를 사용하면 평균 0 분산 1의 가우스 분포로 난수를 생성할 수 있음.

In
```
np.random.rand(2, 3)
```

Out
```
array([[ 0.61172168, 0.20792486, 0.95905162],
       [ 0.86475323, 0.18373685, 0.55318816]])
```

10.7 행렬의 크기 변경

- 행렬의 크기를 변경하는 경우 변수명.reshape(n, m)를 사용함.

In
```
a = np.arange(10)
print(a)
```

Out
```
[0 1 2 3 4 5 6 7 8 9]
```

SECTION 11 행렬(ndarray)의 사칙 연산

11.1 행렬의 사칙 연산

- 사칙 연산 +, -, *, /는 해당되는 요소 전체에 적용됨.

In
```
x = np.array([[4, 4, 4], [8, 8, 8]])
y = np.array([[1, 1, 1], [2, 2, 2]])
print(x + y)
```

Out
```
[[ 5 5 5]
 [10 10 10]]
```

11.2 스칼라 × 행렬

- 스칼라를 행렬에 곱하면 다음과 같이 모든 요소에 적용됨.

In
```
x = np.array([[4, 4, 4], [8, 8, 8]])
print(10 * x)
```

Out
```
[[40 40 40]
 [80 80 80]]
```

SECTION 11 행렬(ndarray)의 사칙 연산

11.3 산술 함수

- 넘파이에는 다양한 수학 함수가 준비됨. exp(x) 함수는 다음과 같이 계산됨.

In
```
x = np.array([[4, 4, 4], [8, 8, 8]])
print(np.exp(x))
```

Out
```
[[  54.59815003   54.59815003   54.59815003]
 [2980.95798704 2980.95798704 2980.95798704]]
```

11.4 행렬 곱의 계산

- 요소별로 계산하지 않는 행렬 곱은 변수명1.dot(변수명2)로 계산할 수 있음.

In
```
v = np.array([[1, 2, 3], [4, 5, 6]])
w = np.array([[1, 1], [2, 2], [3, 3]])
print(v.dot(w))
```

Out
```
[[14 14]
 [32 32]]
```

SECTION 12 슬라이싱

12.1 슬라이싱의 이용

- list와 ndarray에서 요소를 한 번에 나타낼 때 슬라이스라는 편리한 방법을 사용할 수 있음.
- 슬라이스는 ':'을 사용하여 나타냄.

In
```
x = np.arange(10)
print(x)
print(x[:5])
```

Out
```
[0 1 2 3 4 5 6 7 8 9]
[0 1 2 3 4]
```

- 변수명[n:] 을 입력하면 n에서 마지막 요소까지 참조됨.

In
```
print(x[5:])
```

Out
```
[5 6 7 8 9]
```

SECTION 12 슬라이싱

- 변수명[n1:n2:dn]을 입력하면 n1에서 n2-1의 요소까지 dn 간격으로 참조됨.

In
```
print(x[3:8:2])
```

Out
```
[3 5 7]
```

- 슬라이스는 1차원 이상의 list나 ndarray로도 사용할 수 있음.

In
```
y = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(y)
print(y[:2, 1:2])
```

Out
```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[2]
 [5]]
```

SECTION 13 조건을 만족하는 데이터의 수정

13.1 bool 배열 사용

- 넘파이를 통해 행렬 데이터에서 특정 조건을 만족하는 것을 추출하여 쉽게 수정할 수 있음.

In
```
x = np.array([1, 1, 2, 3, 5, 8, 13])
x > 3
```

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

[x를 정의하고 x > 3을 입력하면 결과가 True 또는 False로 bool 형식의 배열이 반환 됨]

In
```
x[x > 3]
```

Out
```
array([ 5, 8, 13])
```

[bool 배열의 요소를 참조하면, True 요소만 출력됨]

In
```
x[x > 3] = 999
print(x)
```

Out
```
[ 1 1 2 3 999 999 999]
```

[x > 3을 충족하는 요소만 999로 바꾸려면 다음과 같이 입력함]

SECTION 14 Help

14.1 Help 사용

- 함수의 설명을 확인하는 help(함수명)
- 함수의 다양한 사용법을 help 명령으로 확인할 수 있음.

In
```
help(np.random.randint)
```

Out
```
Help on built-in function randint:
randint(...) method of mtrand.RandomState instance
    randint(low, high=None, size=None, dtype='l')
        Return random integers from 'low' (inclusive) to 'high' (exclusive).
          ( …중략… )
```

SECTION 15 함수

15.1 함수의 사용

- 프로그램의 일부를 함수로 정리할 수 있음.
- 자주 사용하는 반복된 코드는 함수로 만드는 것 이 좋음.
- 함수는 def 함수명(): 으로 시작하여, 함수의 내용은 들여쓰기로 정의 함.

In

```
def my_func1():
    print('Hi!')
#함수 my_func1()의 정의는 여기까지
my_func1() # 함수를 실행
```

```
Hi!
```

In

```
def my_func2(a, b):
    c = a + b
    return c


my_func2(1, 2)
```

Out

```
3
```

15.2 인수와 반환값

- 함수에 전달할 변수를 인수라고 함.
- 함수의 출력은 반환값이라고 함.

In
```
def my_func3(D):
    m = np.mean(D)
    s = np.std(D)
    return m, s
```

- 여러 반환값을 받는 방법은 data_mean, data_std = my_func3(data)와 같이 ','로 구분 기술함.

In
```
data=np.random.randn(100)
data_mean, data_std = my_func3(data)
print('mean:{0:3.2f}, std:{1:3.2f}'.format(data_mean, data_std))
```

Out
```
mean:0.10, std:1.04
```

SECTION 16 파일 저장

16.1 하나의 ndarray 형을 저장

- 하나의 ndarray 형을 파일에 저장하려면 np.save ('파일명.npy', 변수명)을 사용함.
- 파일의 확장자는 .npy 임.(난수이므로 실행할 때마다 다른 결과가 저장됩니다).
- 데이터를로드하려면(읽으려면) np.load('파일명.npy')를 사용함.

In

```
data = np.random.randn(5)
print(data)
np.save('datafile.npy', data)    # 세이브
data = []                        # 데이터 삭제
print(data)
data = np.load('datafile.npy')   # 로드
print(data)
```

Out

```
[ 0.04283863 0.11556549 -0.12882679 1.03572699 1.2465202 ]
[]
[ 0.04283863 0.11556549 -0.12882679 1.03572699 1.2465202 ]
```

SECTION 16 파일 저장

16.2 여러 ndarray 형을 저장

- 여러 ndarray 형을 저장하려면 np.savez('파일명.npz', 변수명1 = 변수명1, 변수명2 =변수명2, ...)을 사용함.

In

```
data1 = np.array([1, 2, 3])
data2 = np.array([10, 20, 30])
np.savez('datafile2.npz', data1=data1, data2=data2) # 세이브
data1 = [] # 데이터 삭제
data2 = []
outfile = np.load('datafile2.npz') # 로드
print(outfile.files) # 저장된 데이터 표시
data1 = outfile['data1'] # data1 꺼내기
data2 = outfile['data2'] # data2 꺼내기
print(data1)
print(data2)
```

Out

```
['data1', 'data2']
[1 2 3]
[10 20 30]
```