report

Python과 R의 기능 비교



Team) 이순규(Leader) 오준서 임성현

내용

[서론 4](#_Toc93334263)

[본론 5](#_Toc93334264)

[1. empty 5](#_Toc93334265)

[2. sum 7](#_Toc93334266)

[3. mean 9](#_Toc93334267)

[4. zeros 11](#_Toc93334268)

[5. var 14](#_Toc93334269)

[6. min 16](#_Toc93334270)

[7. max 18](#_Toc93334271)

[8. cumsum 20](#_Toc93334272)

[9. random 22](#_Toc93334273)

[10. sqrt 24](#_Toc93334274)

[11. Sort 27](#_Toc93334275)

[12. Append(Array) 28](#_Toc93334276)

[13. delete(numpy) 30](#_Toc93334277)

[14. copy(numpy) 32](#_Toc93334278)

[15. arange(numpy) 34](#_Toc93334279)

[16. read\_csv(pandas) 36](#_Toc93334280)

[17. unique(numpy) 37](#_Toc93334281)

[18. dtype(numpy) 38](#_Toc93334282)

[19. slicing 40](#_Toc93334283)

[20. dataframe(pandas) 41](#_Toc93334284)

[21. dim 42](#_Toc93334285)

[22. size 44](#_Toc93334286)

[23. values 44](#_Toc93334287)

[24. head 46](#_Toc93334288)

[25. tail 47](#_Toc93334289)

[26. shape 48](#_Toc93334290)

[27. T(Transpose) 49](#_Toc93334291)

[28. describe 50](#_Toc93334292)

[29. notnull 52](#_Toc93334293)

[30. get\_dummies() 53](#_Toc93334294)

[결론 54](#_Toc93334295)

[참고자료 55](#_Toc93334296)

# 서론

Numpy와 Pandas 패키지는 데이터 분석에 유용하게 사용되는 패키지 중 하나이다.

아래는 Numpy와 Pandas에 대한 간략한 설명이다.

Numpy:

* 빠르고 효율적인 다차원의 배열인 ndarray 지원
* 배열 또는 배열 간의 수학 연산을 사용하여 요소별 계산을 수행
* 배열을 기반으로 한 데이터 셋을 디스크에 읽고 쓸 수 있는 도구
* 선형 대수 연산, 푸리에 변환, 난수 생성 지원

Pandas:

* NumPy의 배열과 관계형 DB의 유연한 데이터 조작 기능 지원
* 정교한 인덱싱 기능 제공(인덱스 변경, 슬라이싱, 다이스, 집계 등등)

*※ 참고사항*

1~15: numpy

16~30: pandas

# 본론

## empty

**기능)**

numpy : 항목을 초기화하지 않고 지정된 모양과 형식의 새 배열을 생성하는 함수

pandas, R : 데이터 프레임안에 데이터가 비어있는지 확인하는 함수

**비교)**

numpy : 지정된 열과 행의 형식으로 새 배열을 생성한다.

pandas, R : 데이터 프레임안에 데이터가 있는지 확인한다.  
 단, R은 데이터 프레임이 아니고, 일반적인 데이터 일 경우 NA로 표기된다.  
 또한 pandas와 함수 비교를 위해서 데이터 프레임을 생성해야 한다.

**정리)**

numpy를 제외한 두 곳의 쓰임은 같은 것을 볼 수 있었다. 하지만 R과 pandas를 비교할 경우 같은 데이터 프레임을 만들어야 하고, numpy와 같이 일반적인 데이터 형식으로 비교할 경우 NA값이 출력됐다.

**예시)**

Python 코드

>>> data1 = np.empty((2, 2)) # 2\*2 배열함수 생성 함수

>>> data1

array([[1.46923330e+195, 2.30908182e+251],

[1.99613098e+161, 1.87673448e-152]])

python code pandas)

>>> df\_empty

Empty DataFrame

Columns: [A]

index: []

>>> df\_empty.empty

True

>>> df

A

0 NaN

>>> df.empty

False

>>> df.dropna().empty # NA를 삭제하면 인덱스가 비워진 상태로 출력

True

R code)

> install.packages('plyr') # empty함수를 쓰기 위한 패키지 설치

> library(plyr)

> # numpy 비교

> empty(data1)

[1] NA

> empty(data2)

[1] FALSE

> # pandas 비교

> empty(df)

[1] FALSE

## sum

**기능)**

: 각 데이터 안에 있는 원소들의 합 값을 출력한다.

**정리)**

R에서는 NA 값의 처리가 필요하다는 것으로 보아 자동적으로 NA값이 처리되는 numpy와 pandas의 sum함수 쓰임이 우수하다.

**예시)**

python code numpy)

>>> np.sum(data1) # 데이터 안에 들어있는 모든 원소의 합 추출 함수

10.5

>>> np.sum(data2)

36

python code pandas)

>>> df

one two

a 1.40 NaN

b 7.10 -4.5

c NaN NaN

d 0.75 -1.3

>>> df.sum()

one 9.25

two -5.80

dtype: float64

>>> df.sum('index')

one 9.25

two -5.80

dtype: float64

>>> df.sum('columns') # nan = 0으로 자동변환

a 1.40

b 2.60

c 0.00

d -0.55

dtype: float64

R code)

> # numpy 비교

> sum(data1)

[1] 10.5

> sum(data2)

[1] 36

> # pandas 비교

> sum(df$own, na.rm = T)

[1] 9.25

> sum(df$two, na.rm = T)

[1] -5.8

> rowSums(df, na.rm = T)

a b c d

1.40 2.60 0.00 -0.55

## mean

**기능)**

: 각 데이터 안에 있는 원소들의 평균 값을 출력한다.

**비교)**

R에서는 NA값의 처리가 필요하다. 따라서, 자동적으로 NA값이 처리되는 numpy와 pandas에서의 mean()함수 쓰임이 우수하다고 볼 수 있다.

**예시)**

python code numpy)

>>> np.mean(data1) # 데이터 안에 있는 모든 원소의 평균

2.625

>>> np.mean(data2)

4.5

python code pandas)

>>> df.mean()

one 3.083333

two -2.900000

dtype: float64

>>> df.mean('index')

one 3.083333

two -2.900000

dtype: float64

>>> df.mean('columns')

a 1.400

b 1.300

c NaN

d -0.275

dtype: float64

R code)

> # numpy 비교

> mean(data1)

[1] 2.625

> mean(data2)

[1] 4.5

> # pandas 비교

> mean(df$own, na.rm = T)

[1] 3.083333

> mean(df$two, na.rm = T)

[1] -2.9

> rowMeans(df, na.rm = T)

a b c d

1.400 1.300 NaN -0.275

## zeros

**기능)**

: 모든 데이터 값을 0으로 채운 배열을 생성한다.

**비교)**

numpy : 지정한 행/열/차원의 모양으로 값을 0으로 채워 생성한다.

pandas, R : 매트릭스 형태로 영행렬을 생성하며, 3차원 이상의 배열 생성은 불가하다.

**정리)**

R 에서는 3차원 이상의 영행렬을 생성하려면 2차원으로 생성한 뒤 다시 배열을 생성해야 한다. 그러나, numpy에서는 바로 3차원 영행렬 생성이 가능하므로 함수의 쓰임은 numpy가 우수하다.  
 또한, pandas의 데이터 프레임은 3차원 구조를 지원하지 않기 때문에 3차원으로 변형 시 데이터 프레임 구조가 달라질 수 있으므로 주의해야 한다.

**예시)**

python code numpy)

>>> np.zeros(10) # 주어진 길이나 모양에 0이라는 값으로 배열 생성

array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

>>> np.zeros((3, 6))

array([[0., 0., 0., 0., 0., 0.],

[0., 0., 0., 0., 0., 0.],

[0., 0., 0., 0., 0., 0.]])

>>> np.zeros((2, 3, 2))

array([[[0., 0.],

[0., 0.],

[0., 0.]],

[[0., 0.],

[0., 0.],

[0., 0.]]])

python code pandas)

>>> a = pd.DataFrame(np.zeros(10))

>>> b = pd.DataFrame(np.zeros((3, 6)))

>>> c = pd.DataFrame(np.zeros(2,3,2)) # 3차원 부터는 출력의 지원이 안된다.

>>> Traceback (most recent call last):

>>> File "<input>", line 1, in <module>

>>> TypeError: Cannot interpret '3' as a data type

>>> a

0

0 0.0

1 0.0

2 0.0

3 0.0

4 0.0

5 0.0

6 0.0

7 0.0

8 0.0

9 0.0

>>> b

0 1 2 3 4 5

0 0.0 0.0 0.0 0.0 0.0 0.0

1 0.0 0.0 0.0 0.0 0.0 0.0

2 0.0 0.0 0.0 0.0 0.0 0.0

R code)

> install.packages('phonTools') # zeros함수를 쓰기위한 패키지 설치

> library(phonTools)

> zeros(10)

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

> zeros(3, 6)

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

[1,] 0 0 0 0 0 0

[2,] 0 0 0 0 0 0

[3,] 0 0 0 0 0 0

> zeros(2, 3, 2) # 3차원 지원 출력이 안됨, 에러발생

Error in zeros(2, 3, 2) : unused argument (2)

## var

**기능)**

: 각 데이터 안에 있는 원소들의 분산 값을 출력한다.

**비교)**

R에서 분산을 구하기 위해선 NA 값의 처리가 필요하다. 반면, 자동적으로 NA값이 처리되는 numpy와 pandas는 R보다 함수 쓰임이 우수하다고 볼 수 있다.

**예시)**

python code numpy)

>>> np.var(data1, ddof=1) # 데이터 안에 있는 원소들의 분산 값 / ddof=1 (표준편차를 계산할 때, n-1로 나누라는 의미)

1.2291666666666667

>>> np.var(data2, ddof=1) # https://www.abbreviationfinder.org/ko/acronyms/ddof.html#aim

6.0

python code pandas)

>>> df.var()

one 12.205833

two 5.120000

dtype: float64

>>> df.var('index')

one 12.205833

two 5.120000

dtype: float64

>>> df.var('columns')

a NaN

b 67.28000

c NaN

d 2.10125

dtype: float64

R code)

> # numpy 비교

> var(data1)

[1] 1.229167

> var(data2)

[1] 6

> # pandas 비교

> var(df$own, na.rm = T)

[1] 12.20583

> var(df$two, na.rm = T)

[1] 5.12

> apply(df, 1, var, na.rm = TRUE)

a b c d

NA 67.28000 NA 2.10125

## min

**기능)**

: 각 데이터 안에 있는 원소들 중 최소 값을 출력한다.

**비교)**

R에서는 NA 값의 처리가 필요하다는 것으로 보아 자동적으로 NA값이 처리되는 numpy와 pandas에서의 min 함수 쓰임이 우수하다.

**예시)**

python code numpy)

>>> np.min(data1)

1.5

>>> np.min(data2)

1

python code pandas)

>>> df.min()

one 0.75

two -4.50

dtype: float64

>>> df.min('index')

one 0.75

two -4.50

dtype: float64

>>> df.min('columns')

a 1.4

b -4.5

c NaN

d -1.3

dtype: float64

R code)

> # numpy 비교

> min(data1) # 데이터 안에 있는 원소중 제일 작은 값

[1] 1.5

> min(data2)

[1] 1

> # pandas 비교

> min(df$own, na.rm = T)

[1] 0.75

> min(df$two, na.rm = T)

[1] -4.5

> apply(df, 1, min, na.rm = TRUE)

a b c d

1.4 -4.5 Inf -1.3

## max

**기능)**

: 각 데이터 안에 있는 원소들 중 최대 값을 출력한다.

**비교)**

R 에서는 NA 값의 처리가 필요하다는 것으로 보아 자동적으로 NA값이 처리되는 numpy와 pandas에서의 max 함수 쓰임이 우수하다.

**예시)**

python code numpy)

>>> np.max(data1)

4.0

>>> np.max(data2)

8

python code pandas)

>>> df.max()

one 7.1

two -1.3

dtype: float64

>>> df.max('index')

one 7.1

two -1.3

dtype: float64

>>> df.max('columns')

a 1.40

b 7.10

c NaN

d 0.75

dtype: float64

R code)

> # numpy 비교

> max(data1) # 데이터 안에 있는 원소중 제일 큰 값

[1] 4

> max(data2)

[1] 8

> # pandas 비교

> max(df$own, na.rm = T)

[1] 7.1

> max(df$two, na.rm = T)

[1] -1.3

> apply(df, 1, max, na.rm = TRUE)

a b c d

1.40 7.10 -Inf 0.75

## cumsum

**기능)**

: 각 데이터 안에 있는 원소들의 값을 누적치 합의 값으로 출력한다.

**비교)**

R에서는 NA 값의 처리가 필요했으나, numpy와 pandas는 자동적으로 NA값을 처리해주기 때문에 사용성 측면에서는 R보다 우수하다고 볼 수 있다.

**예시)**

python code numpy)

>>> np.cumsum(data1) # 데이터 안에 있는 원소들의 누적 합의 추출 함수 (sum과는 별개)

array([ 1.5, 5.5, 7.5, 10.5])

>>> np.cumsum(data2)

array([ 1, 3, 6, 10, 15, 21, 28, 36], dtype=int32)

python code pandas)

>>> df

one two

a 1.40 NaN

b 7.10 -4.5

c NaN NaN

d 0.75 -1.3

>>> df.cumsum('index')

one two

a 1.40 NaN

b 8.50 -4.5

c NaN NaN

d 9.25 -5.8

>>> df.cumsum('columns')

one two

a 1.40 NaN

b 7.10 2.60

c NaN NaN

d 0.75 -0.55

R code)

> # numpy 비교

> cumsum(data1) # 데이터 안에 있는 원소들의 누적 합

[1] 1.5 5.5 7.5 10.5

> cumsum(data2)

[1] 1 3 6 10 15 21 28 36

> # pandas 비교 na값을 전처리 해야함 그래야 비교가능

> is.na(df)

own two

a FALSE TRUE

b FALSE FALSE

c TRUE TRUE

d FALSE FALSE

> df1 <- na.omit(df)

> cumsum(df1$own)

[1] 7.10 7.85

> cumsum(df1$two)

[1] -4.5 -5.8

> apply(df1, 1, cumsum)

b d

own 7.1 0.75

two 2.6 -0.55

## random

**기능)**

: 난수를 생성한다.

**비교)**

R에서의 난수 생성은 1차원으로만 가능하다. 반면, numpy와 pandas에서는 다차원 난수의 생성이 가능하므로, numpy와 pandas에서의 함수 쓰임이 더 우수하다고 볼 수 있다.

**예시)**

python code numpy)

>>> np.sqrt(data1)

array([1.22474487, 2. , 1.41421356, 1.73205081])

>>> np.sqrt(data2)

array([[1. , 1.41421356, 1.73205081, 2. ],

[2.23606798, 2.44948974, 2.64575131, 2.82842712]])

R code)

> # numpy 비교

> sqrt(data1)

[1] 1.224745 2.000000 1.414214 1.732051

> sqrt(data2)

, , 1

[,1] [,2] [,3] [,4]

[1,] 1 1.414214 1.732051 2

, , 2

[,1] [,2] [,3] [,4]

[1,] 2.236068 2.44949 2.645751 2.828427

> # pandas 비교

> sqrt(df$own)

[1] 1.1832160 2.6645825 NA 0.8660254

> sqrt(df$two)

[1] NA NaN NA NaN

Warning message:

In sqrt(df$two) : NaNs produced

> apply(df, 1, sqrt)

a b c d

own 1.183216 2.664583 NA 0.8660254

two NA NaN NA NaN

## sqrt

**기능)**

: 각 데이터 안에 있는 원소들의 값을 제곱근의 값으로 출력한다.

**비교)**

R과 pandas에서는 NA 값의 처리가 필요하다. 반면, numpy에서는 자동적으로 NA값이 처리되기 때문에 좀 더 간편하게 제곱근을 구할 수 있다.

**예시)**

python code

numpy)

>>> np.sqrt(data1)

array([1.22474487, 2. , 1.41421356, 1.73205081])

>>> np.sqrt(data2)

array([[1. , 1.41421356, 1.73205081, 2. ],

[2.23606798, 2.44948974, 2.64575131, 2.82842712]])

python code pandas)

>>> df.dropna(axis=0)

one two

b 7.10 -4.5

d 0.75 -1.3

>>> df.dropna(axis=1)

Empty DataFrame

Columns: []

Index: [a, b, c, d]

>>> df.transform('sqrt')

one two

a 1.183216 NaN

b 2.664583 NaN

c NaN NaN

d 0.866025 NaN

pandas)

>>> df.dropna(axis=0)

one two

b 7.10 -4.5

d 0.75 -1.3

>>> df.dropna(axis=1)

Empty DataFrame

Columns: []

Index: [a, b, c, d]

>>> df.transform('sqrt')

one two

a 1.183216 NaN

b 2.664583 NaN

c NaN NaN

d 0.866025 NaN

R code)

> # numpy 비교

> sqrt(data1)

[1] 1.224745 2.000000 1.414214 1.732051

> sqrt(data2)

, , 1

[,1] [,2] [,3] [,4]

[1,] 1 1.414214 1.732051 2

, , 2

[,1] [,2] [,3] [,4]

[1,] 2.236068 2.44949 2.645751 2.828427

> # pandas 비교

> sqrt(df$own)

[1] 1.1832160 2.6645825 NA 0.8660254

> sqrt(df$two)

[1] NA NaN NA NaN

Warning message:

In sqrt(df$two) : NaNs produced

> apply(df, 1, sqrt)

a b c d

own 1.183216 2.664583 NA 0.8660254

two NA NaN NA NaN

## Sort

**비교)**  
**Numpy**: axis, kind, order 옵션이 있다.  
 - Axis : 기본값은 -1이며, 오름차순으로 배열을 정렬한다.  
 - kind : quicksort, mergesort, heapsort, stable 옵션이 있으며 정렬 알고리즘을 선택한다.  
 - order : 배열의 필드가 정의된 경우 먼저 비교할 필드를 지정하는데 사용한다.

**Pandas**: 데이터프레임으로 변환하고 기본 내장함수인 sort\_values 함수를 적용한다.

**R**: R에서는 기본 내장함수인 sort를 사용한다.

**정리)**  
파이썬에서는 정렬 알고리즘 선택이 가능하기 때문에 좀더 세세한 커스텀이 가능했으나, 자료형에 영향을 많이 받아서 다루기가 어려웠고, R에서는 자료형에 영향없이 자동으로 정렬이 가능했다.

**예시)**  
Python Code

Numpy)

sortData1 = np.sort(data1)

sortData1

array([1.5, 2. , 3. , 4. ])

R code

data1

## [1] 1.5 4.0 2.0 3.0

x11 <- sort(data1)  
x11

## [1] 1.5 2.0 3.0 4.0

## Append(Array)

**비교)**

**Numpy:** 내장 함수인 append 와는 다르게 numpy로 만든 배열은 차원의 수가 맞지 않으면 이어 붙일 수 없다.

**Pandas**: 데이터 프레임에 대한 결합이므로 여기서는 다루지 않음.

**R**: 다차원 배열을 append 할 경우 1차원으로 바뀌며 순차적으로 append를 실행한다. 2차원 이상의 배열을 유지하려면 배열을 새로 생성하면 된다.

**정리)**  
공통적으로 Python과 R 모두 컬럼의 개수가 다른 배열을 이어 붙이기 위해서는 우선적으로 컬럼 수를 맞춰주어야 한다.  
R 에서는 Python과 같은 append() 를 이용할 경우 다차원의 배열이 1차원으로 변한다. 따라서, 차원을 유지하려면 배열을 새로 생성해서 비교적 간단하게 해결할 수 있다.

**예시)**

Python Code

x=np.array(data1)

y=np.array(data2)

print(x.shape)

(4,)

print(y.shape)

(2, 4)

append1 = np.append(x, y.reshape(1, 2), axis=0) **#오류발생!**

append2 = np.append(y, x.reshape(1, 4), axis=0)

append2

array([[1. , 2. , 3. , 4. ],

[5. , 6. , 7. , 8. ],

[1.5, 4. , 2. , 3. ]])

R code

data1 <- array(c(1.5, 4, 2, 3))  
data2 <- array(c(1:8),c(1,4,2))  
data1

## [1] 1.5 4.0 2.0 3.0

data2

## , , 1  
##   
## [,1] [,2] [,3] [,4]  
## [1,] 1 2 3 4  
##   
## , , 2  
##   
## [,1] [,2] [,3] [,4]  
## [1,] 5 6 7 8

append1 <- append(data1,data2)  
append2 <- array(c(c(data2,data2)),c(1,4,4))  
append1

## [1] 1.5 4.0 2.0 3.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 #1차원으로 변함!

append2

## , , 1  
##   
## [,1] [,2] [,3] [,4]  
## [1,] 1 2 3 4  
##   
## , , 2  
##   
## [,1] [,2] [,3] [,4]  
## [1,] 5 6 7 8  
##   
## , , 3  
##   
## [,1] [,2] [,3] [,4]  
## [1,] 1 2 3 4  
##   
## , , 4  
##   
## [,1] [,2] [,3] [,4]  
## [1,] 5 6 7 8

## delete(numpy)

**비교)**

**Numpy:** 특정 요소, 행/열 등을 삭제할 수 있었다.

**R:** 3차원 배열에서 컬럼 삭제 시 행과 열이 바뀌어 불편한 점이 있었다.

**정리)**

delete 함수에서는 파이썬이 R보다 조금 더 직관적으로 동작했다.

**예시)**

Python Code

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

arr

array([[ 1, 2, 3, 4],

[ 5, 6, 7, 8],

[ 9, 10, 11, 12]])

np.delete(arr, 1, 0)

array([[ 1, 2, 3, 4],

[ 9, 10, 11, 12]])

np.delete(arr, np.s\_[::2], 1)

array([[ 2, 4],

[ 6, 8],

[10, 12]])

np.delete(arr, [1,3,5], None)

array([ 1, 3, 5, 7, 8, 9, 10, 11, 12])

R Code

data1 <- array(c(1.5, 4, 2, 3))  
data2 <- array(c(1:8),c(1,4,2))  
  
data2

## , , 1  
##   
## [,1] [,2] [,3] [,4]  
## [1,] 1 2 3 4  
##   
## , , 2  
##   
## [,1] [,2] [,3] [,4]  
## [1,] 5 6 7 8

data2[,,-1]

## [1] 5 6 7 8

t(data2[,-c(2,4),])

## [,1] [,2]  
## [1,] 1 3  
## [2,] 5 7

data2[!data2 %in% c(1,3,5)]

## [1] 2 4 6 7 8

## copy(numpy)

**비교)**

**numpy:** 얕은 복사이기 때문에 x의 데이터 내용이 data2로 바뀌어도 기존에 복사했던 data1의 데이터를 갖고 있는다.

**R:** copy()를 사용하기 위해서는 data.table 패키지를 설치해야 한다.

**정리)**

둘다 얕은 복사와 깊은 복사의 문법 차이는 있었으나 기능상 큰 차이는 없었다.

**예시)**

Python Code

x = data1  
y = np.copy(data1)

x == data1

True

y == data1

array([ True, True, True, True])

x = data2

x == data1

False

y == data1

array([ True, True, True, True])

R Code

x = data1

y = copy(x)

x == data1

y == data1

x = data2

x == data1 # 변수내용이 data2로 바뀌자 오류발생

y == data1

## arange(numpy)

**비교)**

**numpy:** 내장 함수인 range와 동일한 기능이지만, 결과는 list가 아닌 ndarray로 생성한다는 차이가 있다.

**R:** R에서 range는 단지 최대, 최소값만 나타내며, 파이썬처럼 정해진 규칙대로 배열을 생성하려면 seq()를 사용한다.

**정리)**

Python과 R 에서의 range()는 이름은 같으나 서로 다른 기능을 하는 함수이고, 동일한 기능을 보여주는 seq와 비교해봤을 때 큰 차이는 느끼지 못했다.

**예시)**

Python Code

>>> np.arange(3)

array([0, 1, 2])

>>> np.arange(3.0)

array([0., 1., 2.])

>>> np.arange(3,7)

array([3, 4, 5, 6])

>>> np.arange(3,7,2)

array([3, 5])

R Code

range(1,3,6)

## [1] 1 6

range(1.0:3.0,0.1)

## [1] 0.1 3.0

seq(1,3,by=1)

## [1] 1 2 3

seq(1.0,3.0,by=1.0) # 소수단위는 생성하지 않는다.

## [1] 1 2 3

seq(3.0,6.5) # by옵션 미설정시 기본값은 1이다.

## [1] 3 4 5 6

seq(3,6,2)

## [1] 3 5

## read\_csv(pandas)

**기능)**

: 외부의 csv 파일을 불러온다.

**비교)**

25mb 분량의 csv파일을 다운로드 했을 때, Python의 pandas 패키지가 R보다 더 빠르게 작동했다.

## unique(numpy)

**기능)**

: 배열 내 중복된 요소를 제거한다.

**정리)**

numpy에서는 숫자형이 아닌 자료에 unique()를 사용하면 데이터 타입이 따로 표기가 되어 나왔고, R에서는 자료형에 상관없이 중복요소를 제거했다.

**예시)**

Python Code

names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

np.unique(names)

array(['Bob', 'Joe', 'Will'], dtype='<U4')

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

np.unique(ints)

array([1, 2, 3, 4])

R Code

names = c('Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe')  
unique(names)

## [1] "Bob" "Joe" "Will"

ints = c(3, 3, 3, 2, 2, 1, 1, 4, 4)  
unique(ints)

## [1] 3 2 1 4

mix <- c(c(3, 3, 3, 2, 2, 1, 1, 4, 4),c('Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'))  
unique(mix)

## [1] "3" "2" "1" "4" "Bob" "Joe" "Will"

## dtype(numpy)

**기능)**

: 데이터의 타입을 알려준다.

**정리)**

자료형의 표현방식에 큰 차이가 있었다.

**예시)**

npdata1 = np.array(data1)

npdata1.dtype

dtype('float64')

npdata2 = np.array(data2)

npdata2.dtype

dtype('int32')

R Code

data1 <- c(1.5, 4, 2, 3)   
data1

## [1] 1.5 4.0 2.0 3.0

vec1 <- c(1:8)  
data2 <- array(vec1, c(1, 4, 2))  
data2

## , , 1  
##   
## [,1] [,2] [,3] [,4]  
## [1,] 1 2 3 4  
##   
## , , 2  
##   
## [,1] [,2] [,3] [,4]  
## [1,] 5 6 7 8

mode(data1)

## [1] "numeric"

mode(data2)

## [1] "numeric"

typeof(data1)

## [1] "double"

typeof(data2)

## [1] "integer"

## slicing

**기능)**

: 배열 내에서 특정 요소를 추출한다.

**비교)**

Python과 R 둘 다 슬라이싱하는 방법 및 기능이 모두 동일했다. 그러나 R에서는 차원이 맞지 않거나 슬라이싱 범위를 구체적으로 적지 않으면 에러가 발생했다. Python에서는 arrayp[1 : ] 와 같이 슬라이싱의 end지점을 지정하지 않으면 끝까지 자동으로 출력해준다는 점에서 사용성이 좀 더 우수했다.

Python Code

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5])

[2 3 4 5]

R Code

# 19. slicing  
arr <- array(c(1, 2, 3, 4, 5, 6, 7))  
print(arr[1:5])

## [1] 1 2 3 4 5

## dataframe(pandas)

**기능)**

: 데이터 프레임을 생성한다.

**비교)**

**pandas**: 형식: pd.DataFrame(data, index, columns, dtype, copy)  
ndarray, series, map, lists, dict, 상수 및 다른 DataFrame까지 변수의 형태로 가질 수 있다.

**R**: 형식: DF <- data.frame(vector1, vector2, matrix1, ....)  
하나의 열을 추출할 땐 벡터형태로 추출하고, 두 개 이상의 열을 추출할 땐 데이터 프레임 형태로 추출한다.

**정리)**

여러 자료형을 하나로 묶을 수 있는 2차원의 데이터 구조란 점은 동일했다. 그러나, Python의 pandas 데이터 프레임은 멀티 인덱싱 또는 멀티 컬럼 설정이 가능했다는 부분에서 차이가 있었다.

## dim

**기능)**

: 데이터의 차원을 보여주는 기능이다.

**비교)**

Pandas : R과는 다르게 데이터의 차원자체만을 보여줄 수 있는 기능이 있다.

R : 차원만을 보여주는 함수가 없어 dim()함수를 이용하여 확인할 수 있다.

**예시)**

Python 코드

# 공통 데이터 생성

data = {**'a'** : [1,2,3], **'b'** : [4,5,6]}  
c = pd.Series([1,2,pd.NA,4], index=(1,2,3,4))  
print(c); print(data)

df = pd.DataFrame(data, index=(1,2,3))  
print(df)

# ndim ; 데이터의 차원을 보여준다.

print(df.ndim)

## 2

R 코드

# 공통 데이터 생성

a = c(1,2,3)

b = c(4,5,6)

df = data.frame(a, b)

df

c = c(1,2,NA,4)

# pandas 비교

# dim ; 데이터의 차원만 따로 보여주는 기능을 찾지 못해 dim으로 대체

dim(df)

## [1] 3 2

## size

**기능)**

: 데이터의 원소의 개수를 알려주는 기능이다.

**비교)**

Pandas: 데이터의 원소의 개수를 간단한 메서드를 통해 알아볼 수 있다.

R: nrow() \* ncol()의 함수를 이용하여 행과 열을 곱하거나, table() 함수를 이용하여 데이터 원소를 확인할 수 있다.

**예시)**

Python 코드

# size ; 데이터 원소의 수

print(df.size)

## 6

R 코드

# pandas 비교

# nrow() \* ncol() ; 전체 데이터의 원소의 수 or table() 함수 사용

nrow(df) \* ncol(df)

## [1] 6

## values

**기능)**

: 데이터의 구조를 array 구조로 변경하는 기능이다.

**비교)**

Pandas: R과는 비교적 쉽게 데이터 구조를 변경할 수 있는 장점이 있다.

R: array() 함수를 데이터 프레임 객체에 사용할 때 데이터 프레임을 구성하는 컬럼이나 벡터등을 이용하여 구조를 변경해야하는 불편함이 있다.

**예시)**

Python 코드

# values

print(df.values)

## [[1 4]

[2 5]

[3 6]]

R 코드

# pandas 비교

# array() ; array 구조로 변경

# array(c(a,b), dim = c(3,2,1))

## [,1] [,2]

[1,] 1 4

[2,] 2 5

[3,] 3 6

## head

**기능)**

: 데이터의 앞의 5개의 자료를 보여준다.

**비교)**

Pandas: 데이터의 앞의 5개의 자료를 보여준다.

R: head() 함수는 Pandas.head()와는 다르게 데이터의 앞의 6개의 자료를 보여준다.

**예시)**

Python 코드

# head

print(df.head()) ; Python에서는 head()는 앞의 5개의 데이터를 보여준다.

## a b

1 1 4

2 2 5

3 3 6

R 코드

# pandas 비교

# head() ; R에서는 head()는 앞의 6개의 데이터를 보여준다.

# head(df)

## a b

1 1 4

2 2 5

3 3 6

## tail

**기능)**

: 데이터의 뒤의 5개의 자료를 보여준다.

**비교)**

Pandas: 데이터의 뒤의 5개의 자료를 보여준다.

R : tail() 함수는 Pandas.head()와는 다르게 데이터의 뒤의 6개의 자료를 보여준다.

**예시)**

Python 코드

# tail

print(df.tail()) ; Python에서는 tail()는 뒤의 5개의 데이터를 보여준다.

## a b

1 1 4

2 2 5

3 3 6

R 코드

# pandas 비교

# tail() ; R에서는 tail()는 뒤의 6개의 데이터를 보여준다.

# tail(df)

## a b

1 1 4

2 2 5

3 3 6

## shape

**기능)**

: 데이터의 행과 열을 확인할 수 있는 기능이다.

**비교)**

Pandas, R: 데이터의 행과 열을 확인할 수 있는 기능으로 동일하다.

**예시)**

Python 코드

# shape

print(df.shape) ; 데이터의 행과 열을 확인

## (3, 2)

R 코드

# pandas 비교

# dim() ; 데이터의 행과 열을 확인

# dim(df)

##[1] 3 2

## T(Transpose)

**기능)**

: 데이터의 행과 열을 전치시켜주는 기능이다.

Pandas, R: 데이터의 행과 열을 전치시켜주는 기능으로 동일하다.

**예시)**

Python 코드

# T

print(df.T) ; 데이터의 행과 열을 전치

## 1 2 3

a 1 2 3

b 4 5 6

R 코드

# pandas 비교

# t() ; 데이터의 행과 열을 전치

# t(df)

##[,1] [,2] [,3]

a 1 2 3

b 4 5 6

## describe

**기능)**

: 데이터의 요약통계를 확일할 수 있는 기능이다.

**비교)**

Pandas: 데이터의 요약통계를 확인할 수 있으며, R과 다른 점은 표준편차를 추가적으로 알려주며, 이때 ddof라는 옵션이 기본값이 0으로 설정되어 모표준편차를 사용 한다는 차이점과 원소의 개수도 알려주는 기능이 다르다는 점이 있다.

R: summary() 함수를 이용하여 요약통계를 확인할 수 있으며, 파이썬과 다른 점은 R은 표본분산 및 표본표준편차를 사용하고 파이썬과 다르게 각 요약통계에서 원소의 개수를 알려주지는 않는다.

**예시)**

Python 코드

# describe

print(df.describe()) ; 데이터의 요약통계 확인

## a b

count 3.0 3.0

mean 2.0 5.0

std 1.0 1.0

min 1.0 4.0

25% 1.5 4.5

50% 2.0 5.0

75% 2.5 5.5

max 3.0 6.0

R 코드

# pandas 비교

# summary() ; 데이터의 요약통계 확인

# summary(df)

## a b

Min. :1.0 Min. :4.0

1st Qu.:1.5 1st Qu.:4.5

Median :2.0 Median :5.0

Mean :2.0 Mean :5.0

3rd Qu.:2.5 3rd Qu.:5.5

Max. :3.0 Max. :6.0

## notnull

**기능)**

: 결측치를 없애주는 기능이다.

**비교)**

pandas, R: 결측치를 없애주는 기능으로 동일하다.

**예시)**

Python 코드

# notnull

print(c[c.notnull()]) ; 데이터의 NA, NULL 값 등의 결측치를 없애는 기능

## 1 1

2 2

4 4

dtype: object

R 코드

# pandas 비교

# na.omit ; 데이터의 NA값 등의 결측치를 없애는 기능

# na.omit(c)

##[1] 1 2 4 ; 결측치를 제거한 결과값

attr(,"na.action")

[1] 3 ; 결측치가 존재했던 위치

attr(,"class")

[1] "omit"

## get\_dummies()

**기능)**

: 데이터프레임의 컬럼의 원소 빈도수를 확인하는 기능이다.

**비교)**

pandas, R: 데이터프레임의 특정 컬럼의 원소 빈도수를 확인하는 기능으로 동일하다.

**예시)**

Python 코드

# get\_dummies

pd.get\_dummies(df['a']) ; 데이터프레임 df의 a컬럼 원소의 빈도수 확인

## 1 2 3

1 1 0 0

2 0 1 0

3 0 0 1

R 코드

# pandas 비교

# table() ; 데이터프레임 df의 a컬럼 원소의 빈도수 확인

# table(df$a)

##1 2 3

1 1 1

# 결론

대체적으로 NA값의 자동처리 부분에서는 numpy가 우수한 모습을 보였고, 자료형에 따른 데이터의 처리는 R에서 우수한 모습을 보였다.

속도면에서는 Python의 numpy/pandas로 생성한 자료구조들이 약간씩 앞서 있었고, 멀티 인덱싱을 통한 색인 부분에서도 Python이 전체적으로 나은 모습을 보였다.

**결론적으로**, 직관적이고 간결한 문법을 통한 간편한 사용성 및 자료형의 손쉬운 전처리를 원한다면 R이 우수했고, 멀티인덱싱 및 C언어 기반의 패키지를 통한 빠른 대용량 데이터 처리는 Python이 더 우수하다고 느껴졌다. 그러나, Python과 R에 있는 모든 옵션을 사용해 본 것은 아니기 때문에 객관적인 비교는 어려웠다.

# 참고자료

서론 NumPy / Pandas 설명

<https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxwdGhlc29ufGd4OjIzZmYyZjQxMDJiNDg3ZDU>

Numpy append

<https://076923.github.io/posts/Python-numpy-14/>

pandas empty

<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.empty.html>

결론

<https://www.datacamp.com/community/tutorials/r-or-python-for-data-analysis?utm_source=adwords_ppc&utm_medium=cpc&utm_campaignid=12492439802&utm_adgroupid=122563403961&utm_device=c&utm_keyword=python%20and%20r&utm_matchtype=b&utm_network=g&utm_adpostion=&utm_creative=504158804833&utm_targetid=aud-299261629654:kwd-305517187105&utm_loc_interest_ms=&utm_loc_physical_ms=1009846&gclid=Cj0KCQiAoY-PBhCNARIsABcz770WQKYwiwmynQwkKEDo4Ejuds0iWYx1zrqMy_iobRJbSKxoTEPr_3UaAjXaEALw_wcB>