파이썬 라이브러리를 활용한 데이터 분석

4장 유니버설 함수 등

2020.06.19금 2h

4장 numpy 기본

배열 관련 함수
난수

4.2 유니버설 함수

- **Ufuc**
  - Ndarray 안의 원소별로 연산을 수행하는 함수
  - 단항 함수
    - **np.sqrt(arr)**
    - **np.exp(arr)**
      - **[2.7183(원소0), 2.7183(원소1), 2.7183(원소2), ...]**
    - **rem, whole = np.modf(arr)**
      - **소수(rem)와 정수(whole)로 부분으로 나눈 배열 2개를 반환**
      - **내장함수 divmod()의 벡터화 함수**
    - **out 인자를 지정 가능**
      - **np.sqrt(arr1, arr2)**
  - 이항 함수
    - **np.maximun(arr1, arr2)**

단항 함수

Table 4-3. Unary ufuncs

Function	Description
`abs, fabs`	Compute the absolute value element-wise for integer, floating-point, or complex values
`sqrt`	Compute the square root of each element (equivalent to `arr ** 0.5`)
`square`	Compute the square of each element (equivalent to `arr ** 2`)
`exp`	Compute the exponent e^x of each element
`log, log10, log2, log1p`	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
`sign`	Compute the sign of each element: 1 (positive), 0 (zero), or −1 (negative)
`ceil`	Compute the ceiling of each element (i.e., the smallest integer greater than or equal to that number)
`floor`	Compute the floor of each element (i.e., the largest integer less than or equal to each element)
`rint`	Round elements to the nearest integer, preserving the `dtype`
`modf`	Return fractional and integral parts of array as a separate array
`isnan`	Return boolean array indicating whether each value is `NaN` (Not a Number)
`isfinite, isinf`	Return boolean array indicating whether each element is finite (non-`inf`, non-`NaN`) or infinite, respectively
`cos, cosh, sin, sinh, tan, tanh`	Regular and hyperbolic trigonometric functions
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Inverse trigonometric functions
`logical_not`	Compute truth value of `not x` element-wise (equivalent to `~arr`).

이항 함수

Table 4-4. Binary universal functions

Function	Description
`add`	Add corresponding elements in arrays
`subtract`	Subtract elements in second array from first array
`multiply`	Multiply array elements
`divide, floor_divide`	Divide or floor divide (truncating the remainder)
`power`	Raise elements in first array to powers indicated in second array
`maximum, fmax`	Element-wise maximum; `fmax` ignores NaN
`minimum, fmin`	Element-wise minimum; `fmin` ignores NaN
`mod`	Element-wise modulus (remainder of division)
`copysign`	Copy sign of values in second argument to values in first argument
`greater, greater_equal, less, less_equal, equal, not_equal`	Perform element-wise comparison, yielding boolean array (equivalent to infix operators `>, >=, <, <=, ==, !=`)
`logical_and, logical_or, logical_xor`	Compute element-wise truth value of logical operation (equivalent to infix operators `&, |, ^`)

4.3 배열을 이용한 배열지향 프로그래밍

- **Numpy 배열의 장점**
  - 반복문 사용하지 않고 다양한 데이터 처리 가능
    - **벡터화**
    - **순수 파이썬 연산에 비해 10~n00배 빠름**
      - **사례: 배열 브로드캐스팅**
- **메쉬그리드 함수**
  - xs, ys = np.meshgrid(x, y)
    - **인자, 2 개의 1차원 배열**
    - **가능한 모든 (x, y) 짝을 만들 수 있는 2차원 배열 두 개를 반환**
  - 그리드 상의 두 점을 간단하게 계산
    - **z = np.sqrt(xs ** 2 + ys ** 2)**

4.3.1 배열 연산으로 조건절 표현

- **numpy.where function**
  - a vectorized version of the ternary expression x if condition else y.
  - 다음은 느림
    - **Suppose we wanted to take a value from xarr whenever the corresponding value in cond is True, and otherwise take the value from yarr. A list comprehension doing this might look like:**

```
In [165]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])

In [166]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

In [167]: cond = np.array([True, False, True, True, False])

In [168]: result = [(x if c else y)
   .....:           for x, y, c in zip(xarr, yarr, cond)]

In [169]: result
Out[169]: [1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999999999, 2.5]
```

  - 다음이 해결책

```
In [170]: result = np.where(cond, xarr, yarr)

In [171]: result
Out[171]: array([ 1.1,  2.2,  1.3,  1.4,  2.5])
```

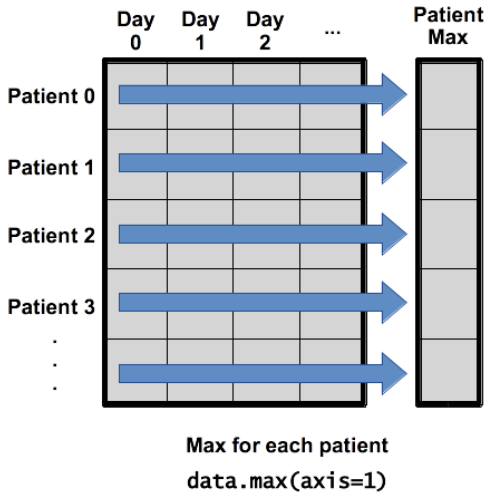
4.3.2 수학 메소드와 통계 메소드

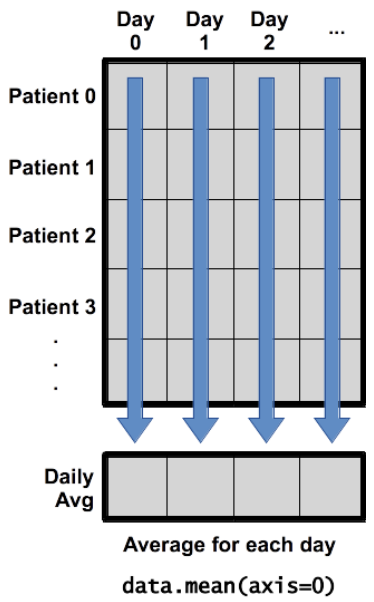
- **배열 메소드**
  - 배열전체 혹은 배열에서 한 축을 따르는 자료에 대한 통계를 계산하는 수학 함수에 사용
  - 전체 합, 평균, 표준편차 사용 방법 2가지
    - **np.mean(arr)**
    - **arr.mean()**

```
In [177]: arr = np.random.randn(5, 4)

In [178]: arr
Out[178]:
array([[ 2.1695, -0.1149,  2.0037,  0.0296],
       [ 0.7953,  0.1181, -0.7485,  0.585 ],
       [ 0.1527, -1.5657, -0.5625, -0.0327],
       [-0.929 , -0.4826, -0.0363,  1.0954],
       [ 0.9809, -0.5895,  1.5817, -0.5287]])

In [179]: arr.mean()
Out[179]: 0.19607051119998253

In [180]: np.mean(arr)
Out[180]: 0.19607051119998253

In [181]: arr.sum()
Out[181]: 3.9214102239996507
```

축에 따른 연산

- **축 옵션 axis=**
  - 0, row, 행
  - 1, column, 열






```
In [177]: arr = np.random.randn(5, 4)

In [178]: arr
Out[178]:
array([[ 2.1695, -0.1149,  2.0037,  0.0296],
       [ 0.7953,  0.1181, -0.7485,  0.585 ],
       [ 0.1527, -1.5657, -0.5625, -0.0327],
       [-0.929 , -0.4826, -0.0363,  1.0954],
       [ 0.9809, -0.5895,  1.5817, -0.5287]])
```

```
In [182]: arr.mean(axis=1)
Out[182]: array([ 1.022 ,  0.1875, -0.502 , -0.0881,  0.3611])

In [183]: arr.sum(axis=0)
Out[183]: array([ 3.1693, -2.6345,  2.2381,  1.1486])
```


https://bham-carpentries.github.io/2018-12-17-bham_python-novice-inflammation/11-loaddata/index.html

누산 함수: accumulation functions

- 중간 누적 계산 값을 갖고 있는 배열을 반환

```
In [184]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])

In [185]: arr.cumsum()
Out[185]: array([ 0,  1,  3,  6, 10, 15, 21, 28])
```

- 2차원 배열도 가능
  - 행을 따라 누적
  - 열을 따라 누적

```
In [186]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

In [187]: arr
Out[187]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

In [188]: arr.cumsum(axis=0)
Out[188]:
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]])

In [189]: arr.cumprod(axis=1)
Out[189]:
array([[  0,   0,   0],
       [  3,  12,  60],
       [  6,  42, 336]])
```

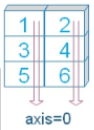
arr.cumsum(axis=?)

cumsum

Cumulative sum of elements of array.

```
import numpy as my_np
my_array = my_np.array([1,2,3,4])
print(my_np.cumsum(my_array)) # [ 1  3  6 10]
```

`axis` : Optional, Cumulative sum based on the specified axis. If no axis is given (default) then flattened array is used (above code).

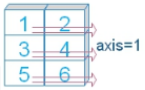



```
import numpy as my_np
my_array = my_np.array([[1,2],
                        [3,4],
                        [5,6]])
print(my_np.cumsum(my_array,axis=0))
```

Output is here

```
[[ 1  2]
 [ 4  6]
 [ 9 12]]
```

axis=1




```
import numpy as my_np
my_array = my_np.array([[1,2],
                        [3,4],
                        [5,6]])
print(my_np.cumsum(my_array,axis=1))
```

Output is here

```
[[ 1  3]
 [ 3  7]
 [ 5 11]]
```

`out` Optional , output can be stored in the array

```
import numpy as my_np
my_array = my_np.array([1,2,3,4])
my_out=my_np.array([0,0,0,0])
my_np.cumsum(my_array,out=my_out)
print(my_out) # [ 1  3  6 10]
```

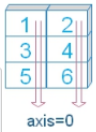
arr.cumsum(axis=?)

cumprod

Cumulative product of elements of array.

```
import numpy as my_np
my_array = my_np.array([1,2,3,4])
print(my_np.cumprod(my_array)) # [ 1  2  6 24]
```

`axis` : Optional, Cumulative product based on the specified *axis*. If no *axis* is given (default) then flattened array is used (above code).

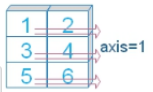



```
import numpy as my_np
my_array = my_np.array([[1,2],
                        [3,4],
                        [5,6]])
print(my_np.cumprod(my_array,axis=0))
```

Output is here

```
[[ 1  2]
 [ 3  8]
 [15 48]]
```

axis=1




```
import numpy as my_np
my_array = my_np.array([[1,2],
                        [3,4],
                        [5,6]])
print(my_np.cumprod(my_array,axis=1))
```

Output is here

```
[[ 1  2]
 [ 3 12]
 [ 5 30]]
```

`out` Optional , output can be stored in the array

```
import numpy as my_np
my_array = my_np.array([1,2,3,4])
my_out=my_np.array([0,0,0,0])
my_np.cumprod(my_array,out=my_out)
print(my_out) # [ 1  2  6 24]
```

4.5 선형 대수

- **행렬 연산**
  - 행렬 곱(내적)
    - **np.dot(a, b)**
    - **a.dot(b)**

$$\begin{matrix} A & & B & & A * B \end{matrix}$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 6 & 3 \\ 5 & 2 \\ 4 & 1 \end{pmatrix} = \begin{pmatrix} 1*6+2*5+3*4 & 1*3+2*2+3*1 \\ 4*6+5*5+6*4 & 4*3+5*2+6*1 \end{pmatrix}$$

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

$$C_{ij} = \sum_k A_{ik} B_{kj} = A_{ik} B_{kj}$$

행렬과 벡터의 내적(곱)

- **(2 x 3) * (3,)**
  - 뒤 벡터를 (1, 3)의 2차원 배열로 간주하여 계산
  - 결과는 1차원 배열

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} * \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

```
In [127]: x
Out[127]: array([[1., 2., 3.],
                 [4., 5., 6.]])

In [128]: np.ones(3)
Out[128]: array([1., 1., 1.])

In [129]: x @ np.ones(3)
Out[129]: array([ 6., 15.])
```

4.6 난수 생성

- **모듈 numpy.random**
  - 표준 내장 모듈보다 수십 배 빠름
- **시드 값**
  - 정해진 시드 값에 따라 일정한 난수가 생성
- **격리된 난수 생성기**
  - np.random.RandomState(seed)

```
In [245]: rng = np.random.RandomState(1234)

In [246]: rng.randn(10)
Out[246]:
array([ 0.4714, -1.191 ,  1.4327, -0.3127, -0.7206,  0.8872,  0.8596,
       -0.6365,  0.0157, -2.2427])
```

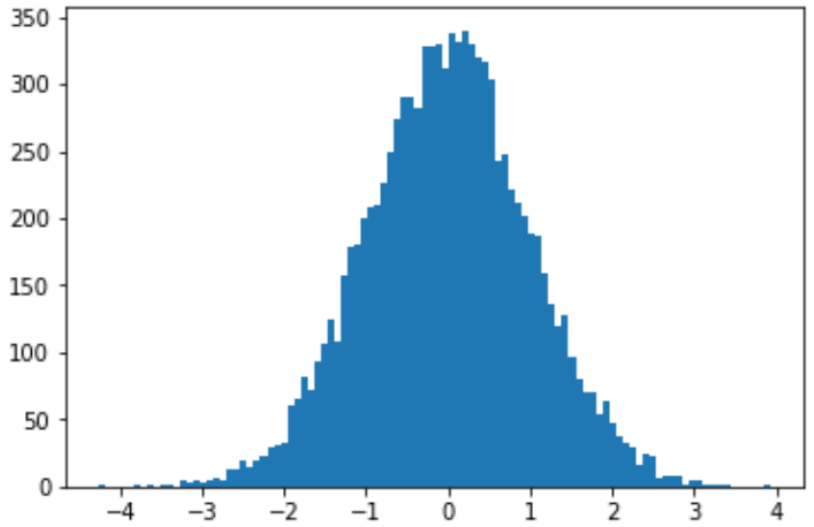
난수

- 난수 발생 및 배열 생성을 생성하는 numpy.random 모듈
  - np.random.normal
    - 지정한 평균과 표준편차인 정규분포
      - **np.random.normal(mean, std, (2, 3))**
      - **np.random.normal(mean, std, 1000)**
  - np.random.randn
    - 평균 0, 표준편차 1인 표준 정규분포
      - **np.random.randn(2, 4)**
      - **np.random.randn(10000)**
  - np.random.rand
    - [0, 1) 균등분포
      - **np.random.rand(3, 2)**
      - **np.random.rand(10000)**
  - np.random.random
    - [0, 1) 균등분포
      - **np.random.random((2, 4))**
      - **np.random.random(1000)**
  - np.random.randint
    - 지정한 [start, stop-1] 사이의 정수 표본 추출
      - **np.random.randint(5, 10, size=(2, 4))**
      - **np.random.randint(-100, 100, 10000)**

난수 기반 배열 생성 **normal**

- **np.random.normal**
  - normal(loc=0.0, scale=1.0, size=None)
  - 정규 분포 확률 밀도에서 표본 추출
  - loc: 정규 분포의 평균
  - scale: 표준편차
  - np.random.normal이 생성한 난수는 정규 분포의 형상을 갖음
  - 다음 예제는 정규 분포로 10000개 표본을 뽑은 결과를 히스토그램으로 표현한 예
  - 표본 10000개의 배열을 100개 구간으로 구분할 때, 정규 분포 형태를 보이고 있음

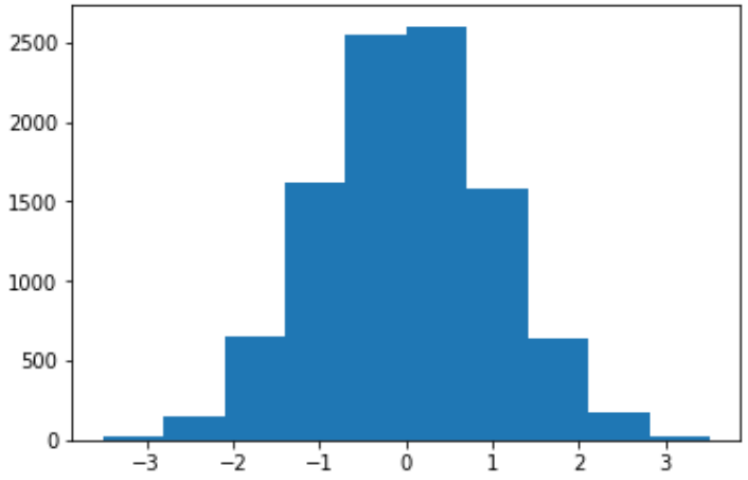
```
In [3]: data = np.random.normal(0, 1, 10000)
import matplotlib.pyplot as plt
plt.hist(data, bins=100)
plt.show()
```



난수 기반 배열 생성 **randn**

- **np.random.randn**
  - numpy.random.randn(d0, d1, ..., dn)
  - (d0, d1, ..., dn) shape 배열 생성 후 난수로 초기화
  - 난수: 표준 정규 분포(standard normal distribution)에서 표본 추출
  - np.random.randn은 정규 분포로 표본 추출
  - 다음 예제는 정규 분포로 10000개를 표본 추출한 결과를 히스토그램으로 표현한 예
  - 표본 10000개의 배열을 10개 구간으로 구분했을 때 정규 분포 형태를 보임

```
In [7]: data = np.random.randn(10000)
        import matplotlib.pyplot as plt
        plt.hist(data, bins=10)
        plt.show()
```

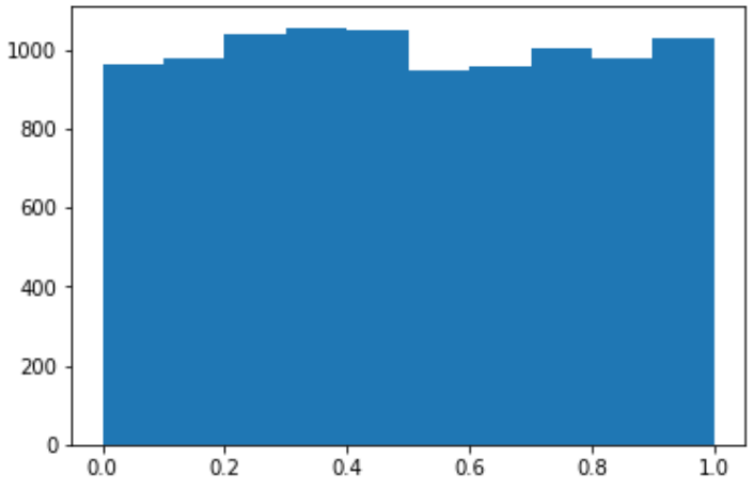


난수 기반 배열 생성 **rand**

- **np.random.rand**
  - numpy.random.rand(d0, d1, ..., dn)
  - Shape이 (d0, d1, ..., dn) 인 배열 생성 후 난수로 초기화
  - 난수: [0. 1)의 균등 분포(Uniform Distribution) 형상으로 표본 추출
  - Gaussian normal

  - np.random.rand는 균등한 비율로 표본 추출
  - 다음 예제는 균등 분포로 10000개를 표본 추출한 결과를 히스토그램으로 표현한 예
  - 표본 10000개의 배열을 10개 구간으로 구분했을 때 균등한 분포 형태를 보임

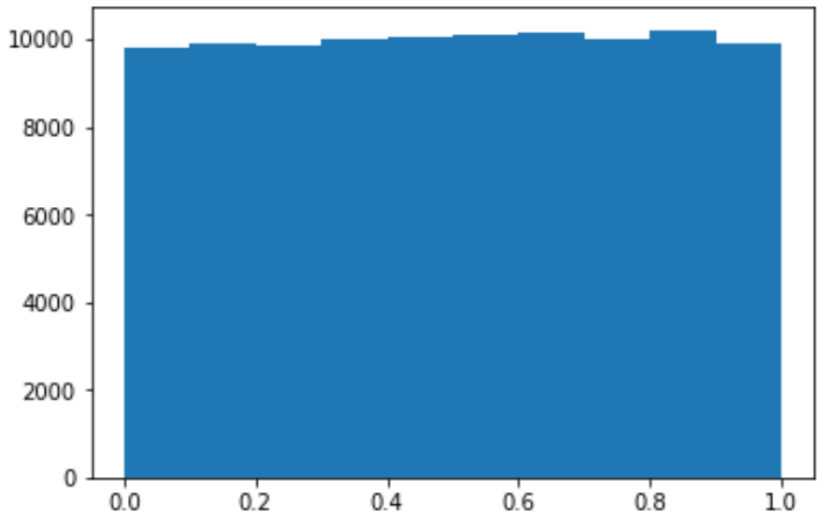
```
In [5]: data = np.random.rand(10000)
        import matplotlib.pyplot as plt
        plt.hist(data, bins=10)
        plt.show()
```



난수 기반 배열 생성 **random**

- **np.ramdom.random**
  - np.random.random(size=None)
  - 난수: [0., 1.)의 균등 분포(Uniform Distribution)에서 표본 추출
  - np.random.random은 균등 분포로 표본을 추출
  - 다음 예제는 정규 분포로 10000개를 표본 추출한 결과를 히스토그램으로 표현
  - 표본 10000개의 배열을 10개 구간으로 구분 했을때 정규 분포 형태를 보임

```
In [12]: data = np.random.random(100000)
         import matplotlib.pyplot as plt
         plt.hist(data, bins=10)
         plt.show()
```



난수 기반 배열 생성 **randint**

- **np.random.randint**
  - numpy.random.randint(low, high=None, size=None, dtype='l')
  - 지정된 shape으로 배열을 만들고 low 부터 high 미만의 범위에서 정수 표본 추출

  - 100에서 100의 범위에서 정수를 균등하게 표본 추출
  - 다음 예제에서 균등 분포로 10000개를 표본 추출한 결과를 히스토그램으로 표현한 예
  - 표본 10000개의 배열을 10개 구간으로 구분했을 때 균등한 분포 형태를 보임

```
In [10]: data = np.random.randint(-100, 100, 10000)
         import matplotlib.pyplot as plt
         plt.hist(data, bins=10)
         plt.show()
```