비교, 마스크, 부울 로직

.

이번절에서는 NumPy 배열 내의 값을 검사하고 조작하는 데 부울 마스크를 사용하는 법을 다룬다. 마스킹은 특정 기준에 따라 배열의 값을 추출하거나 수정, 계산, 조작할 때 사용한다. 특정 값보다 더 큰 값을 모두 세거나 특정 임계치를 넘어서는 이상치를 모두 제거하려는 경우가 여기에 해당한다. NumPy 에서 부울 마스킹은 종종 이러한 유형의 작업을 수행하기에 가장 효율적인 방법이다.

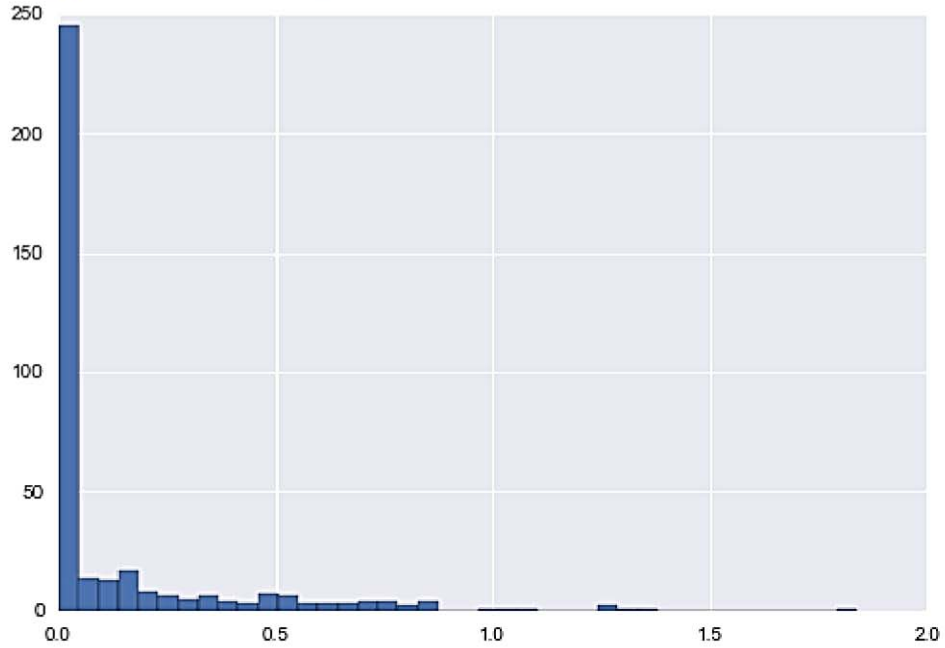
예제 : 비언 날 세기

어느 도시의 일 년간 일일 강수량을 표시한 일련의 데이터를 가지고 있다고 하자,. 예를 들어, 여기서는 Pandas 를 이용해 2014 년 시애틀의 일일 강수량 통계치를 불러올 것이다.

```
In [1]:import numpy as np
       import pandas as pd
# Pandas 를 이용해 인치 단위의 강수량 데이터를 NumPy 배열로 추출
rainfall = pd.read_csv('data/Seattle2014.csv')['PRCP'].values
inches = rainfall / 254.0  # 1/10mm -> inches
inches.shape
Out[1]:(365,)
```

배열에는 2014 년 1 월 1 일부터 12 월 31 일까지 인치 단위의 강수량을 나타내는 365 개의 값이 들어있다.

```
In [2]:%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()  # set plot styles
In [3]:plt.hist(inches, 40);
```



이 히스토그램은 데이터가 어떤 모습인지 대략 떠올리게 해준다. 비가 많이 오기로 유명한 시애틀이지만 2014 년에는 대다수 날에 강수량이 0 에 가까운 모습을 볼 수 있다. 그렇지만 이 그래프는 '그해 비가 온 날은 몇일일까?','비가 온날의 평균 강수량은 얼마인가?','0.5 인치 이상 비가 온 날은 며칠이나 될까?'등 우리가 알고자 하는 정보를 전달하기에는 적합하지 않다.

세부 분석¶

이 문제를 해결할 수 있는 한 가지 접근 방식은 그 질문들에 직접 답하는 것이다. 데이터를 처음부터 끝까지 확인하면서 원하는 범위 안에 있는 값을 볼 때마다 카운트를 1 씩 증가하는 것이다. 이번 장에서 설명한 여러 가지 이유로 그런 접근 방식은 코드 작성시간과 결과 시간 측면에서 매우 비효율적이다. 'NumPy 배열 연산 : 유니버설 함수'에서 루프 대신 NumPy 유니버설 함수를 사용해 배열의 요소 단위 산술 연산을 빠르게 수행할 수 있다는 것을 알았다. 같은 방식으로 다른 ufunc 를 사용해 배열에서 요소 단위로 비교하면 궁금해하는 질문에 대한 답을 얻을 수 있다. 잠시 데이터는 접어두고 이러한 유형의 질문에 신속하게 답하기위해 마스킹을 사용하는 NumPy 의 일반적인 도구 및 가지를 살펴보자.

Ufunc 으로서의 비교 연산자¶

```
In [4]:x = np.array([1, 2, 3, 4, 5])
In [5]:x < 3  # 보다 적음
Out[5]:array([ True,  True, False, False, False], dtype=bool)
In [6]:x > 3  # 보다 큼
Out[6]:array([False, False, False,  True,  True], dtype=bool)
In [7]:x <= 3  # 보다 작거나 같음
Out[7]:array([ True,  True,  True, False, False], dtype=bool)
In [8]:x >= 3  # 보다 크거나 같음
Out[8]:array([False, False,  True,  True,  True], dtype=bool)
In [9]:x != 3  # 같지 않음
Out[9]:array([ True,  True, False,  True,  True], dtype=bool)
In [10]:x == 3  # 같음
Out[10]:array([False, False,  True, False, False], dtype=bool)
```

또한 두 배열을 항목별로 비교할 수 있으며 복합 표현식을 적용할 수도 있다.

```
In [11]:(2 * x) == (x ** 2)
Out[11]:array([False,  True, False, False, False], dtype=bool)
```

Operator	Equivalent ufunc	Operator	Equivalent ufunc
==	np.equal	!=	np.not_equal
<	np.less	<=	np.less_equal
>	np.greater	>=	np.greater_equal

산술 연산자와 마찬가지로 비교 연산자도 NumPy 의 ufuncfh 구현된다. 예를 들어, x<3 이러고 쓰면 NumPy 는 내부적으로 np.less(x,3)을 사용한다. 다음은 비교 연산자와 그에 대응 하는 ufunc 를 정리한 것이다.

```
In [12]:rng = np.random.RandomState(0)
        x = rng.randint(10, size=(3, 4))
        x
Out[12]:array([[5, 0, 3, 3],
               [7, 9, 3, 5],
              [2, 4, 7, 6]])
```

```
In [13]:x < 6
Out[13]:array([[ True,  True,  True,  True],
               [False, False,  True,  True],
               [ True,  True, False, False]], dtype=bool)
```

각 결과는 부울 배열이며, NumPy 는 이 부울 결과로 작업하기 위한 간단한 패턴을 다양하게 제공한다.

부울 배열로 작업하기¶

부울배열이 있을 때 여러가지 유용한 연산을 수행할 수 있다.

```
In [14]:print(x)
[[5 0 3 3]
 [7 9 3 5]

 [2 4 7 6]]
```

요소의 개수 세기¶

부울 배열에서 True 인 요소의 개수를 세는 데는 np.count_nonzero 가 유용하다.:

```
In [15]:# how many values less than 6?
         np.count_nonzero(x < 6)
Out[15]:8
```

6 보다 작은 배열 요소가 8 개 있다는 것을 알았다 이정보를 알아내는 또 다른 방법은 np.sum 을 사용하는 것인데. 이 경우 False 는 0 으로, True 는 1 로 해석된다.

```
In [16]:np.sum(x < 6)
Out[16]:8
```

sum()의 장점은 다른 NumPy 집계 함수와 같이 행이나 열을 따라 계산할 수도 있다는 점이다.

```
In [17]:# how many values less than 6 in each row?
         np.sum(x < 6, axis=1)
Out[17]:array([4, 2, 2])
```

이 코드는 행렬에서 각 행의 6 보다 작은 값의 개수를 센다.

값 중 하나라도 참이 있는지 또는 모든 값이 참인지 빠르게 확인하고 싶다면 np.any()나 np.all()을 사용하면 된다.

```
In [18]:# 8 보다 큰 값이 하나라도 있는가>
         np.any(x > 8)
Out[18]:True
In [19]:# 8 보다 작은 값이 하나라도 있는가?
         np.any(x < 0)
Out[19]:False
In [20]:# 모든 값이 10 보다 작은가?
         np.all(x < 10)
Out[20]:True
In [21]:# 모든 값이 6 과 같은가?
         np.all(x == 6)
Out[21]:False
```

np.all() 과 np.any()는 특정 축을 따라 사용할 수도 있다. :

```
In [22]:# 각 행의 모든 값이 8 보다 작은가?
         np.all(x < 8, axis=1)
Out[22]:array([ True, False,  True], dtype=bool)
```

첫번째와 세 번째 행의 모든 요소는 8 보다 작지만 두번째 행은 그렇지 않다는 것을 알 수 있다.

마지막으로 주의할 점은 '집계:최소값, 최대값 그리고 그 사이의 모든 것에서 언급했듯이 파이썬은 내장 함수로 sum(), any(), all()함수를 가지고 있다는 사실이다. 이것들을 NumPy 함수와는 다른 구문을 가지고 있으며, 특히 다차원 배열에서 사용할 때 실패하거나 의도하지 않은 결과를 만들어 낼 것이다.

부울 연산자¶

앞에서 이미 비가 4 인치보다 작게 내린 날이나 2 인치보다 많이 내린 날을 어떻게 셀 수 있는지 살펴봤다 그렇다면 비가 4 인치보다 적고 1 인치보다 많이 온 날을 알 수 있을까? 그 답은 파이썬의 비트 단위 로직 연산자 &, |, ^, ~를 얻을 수 있다. 표준 산술 연산자와 마찬가지로 Numpy 는 이 연산자를 배열의 요소 단위로 동작하는 유니버설 함수로 오버로딩한다.

예를 들어, 이러한 복합적인 문제는 다음과 같이 해결할 수 있다.

```
In [23]:np.sum((inches > 0.5) & (inches < 1))
Out[23]:29
```

이로써 0.5 인치와 1.0 인치 사이의 강수량을 보인날이 29 일임을 알게 됐다.

여기서는 괄호가 중요하다는 사실을 명심하라. 연산자 선행 규칙에 따라 이 표현식에서 괄호를 제거하면 다음과 같이 연산이 수행되어 결국 에러가 발생한다.

```
In [23]:np.sum(inches > (0.5 & inches) < 1)
```

A AND B 와 NOT(A OR B)가 같음(논리학 입문 과정을 수강했다면 기억할 것이다)을 이용하면 다른 방식으로도 같은 결과를 계산할 수 있다.

```
In [24]:np.sum(~( (inches <= 0.5) | (inches >= 1) ))
Out[24]:29
```

배열에서 비교연산자와 부울 연산자를 결합하면 여러가지 효율적인 로직 연산을 할 수 있다.

Operator	Equivalent ufunc	Operator	Equivalent ufunc
&	np.bitwise_and	|	np.bitwise_or
^	np.bitwise_xor	~	np.bitwise_not

이 도구를 사용해 날씨 데이터에 대한 질문의 답을 찾을 수 있다. 여기에 마스킹과 집계 함수를 결합헤서 계산할 수 있는 결과값에 대한 몇 가지 예제를 소개한다.

```
In [25]:print("Number days without rain:      ", np.sum(inches == 0))
        print("Number days with rain:         ", np.sum(inches != 0))
        print("Days with more than 0.5 inches:", np.sum(inches > 0.5))
        print("Rainy days with < 0.2 inches  :", np.sum((inches > 0) &
                                                (inches < 0.2)))
Number days without rain:       215
Number days with rain:          150
Days with more than 0.5 inches: 37
Rainy days with < 0.2 inches  : 75
```

마스크로서의 부울 배열¶

앞에서 부울 배열에서 직접 계산하는 집계 함수를 살펴봤다. 더 강력한 패턴은 부울 배열을 마스크로 사용해 데이터자체의 특정 부분 집합을 선택하는 것이다. 앞에서 생성한 x 배열로 돌아가 5 보다 작은 배열 내 값들을 모두 구한다고 생각해 보자.

```
In [26]:x
Out[26]:array([[5, 0, 3, 3],
               [7, 9, 3, 5],
               [2, 4, 7, 6]])
```

이미 본대로 이 조건에 맞는 부울 배열을 쉽게 얻을 수 있다.

```
In [27]:x < 5
Out[27]:array([[False,  True,  True,  True],
               [False, False,  True, False],
               [ True,  True, False, False]], dtype=bool)
```

이제 배열에서 조건에 맞는 값들을 선택하려면 간단히 이 부울 배열을 인덱스로 사용하면 된다. 이를 마스킹 연산이라고 한다.

```
In [28]:x[x < 5]
Out[28]:array([0, 3, 3, 3, 2, 4])
```

반환된 값은 이 조건에 맞는 모든 값, 다시 말해 마스크 배열이 True 인 위치에 있는 모든 값으로 채워진 1 차원 배열이다.

이제 원하는 대로 이 값들에 대해 자유롭게 연산을 수행할 수 있다. 예를 들어, 시애틀 강수량 데이터에 관한 몇가지 관련 통계치를 계산할 수 있다.

```
In [29]:# 비가 온 날에 대한 마스크 생성
rainy = (inches > 0)
# 여름에 해당하는 날에 대한 마스크 생성(6 월 21 일은 172 번째 날임)
days = np.arange(365)
summer = (days > 172) & (days < 262)
print("Median precip on rainy days in 2014 (inches):   ",
      np.median(inches[rainy]))
print("Median precip on summer days in 2014 (inches):  ",
      np.median(inches[summer]))
print("Maximum precip on summer days in 2014 (inches): ",
      np.max(inches[summer]))
print("Median precip on non-summer rainy days (inches):",
      np.median(inches[rainy & ~summer]))
Median precip on rainy days in 2014 (inches):    0.194881889764
Median precip on summer days in 2014 (inches):   0.0
Maximum precip on summer days in 2014 (inches):  0.850393700787
Median precip on non-summer rainy days (inches): 0.200787401575
```

부울 연산과 마스킹 연산, 잡계 연산을 결합하면 이러한 종류의 질문에 매우 빠르게 답할 수 있다.

키워드 and/or vs 연산자 &/| 사용하기¶

흔히 혼동하는 것 중 하나가 키워드 and / or 와 연산자 & / |의 차이다, 언제 어느 것을 사용할 것인가?:

```
In [30]:bool(42), bool(0)
Out[30]:(True, False)
In [31]:bool(42 and 0)
Out[31]:False
In [32]:bool(42 or 0)
Out[32]:True
```
& 와 |를 정수에 사용할 때 표현식은 그 요소의 비트에 대해 동작하므로 그 숫자를 구성하는 개별 비트에 and 와 or 를 적용하는 것과 같다.:
```
In [33]:bin(42)
Out[33]:'0b101010'
In [34]:bin(59)
Out[34]:'0b111011'
In [35]:bin(42 & 59)
Out[35]:'0b101010'
In [36]:bin(42 | 59)
Out[36]:'0b111011'
```

결과를 산출하기 위해 이진 표현에서 대응하는 비트를 비교한다는 점을 알아두자.
```
In [37]:A = np.array([1, 0, 1, 0, 1, 0], dtype=bool)
        B = np.array([1, 1, 1, 0, 1, 1], dtype=bool)
        A | B
Out[37]:array([ True,  True,  True, False,  True,  True], dtype=bool)
```

이 배열에 or 를 사용하는 것은 전체 배열 객체의 점이나 거짓을 평가하라는 것이로서, 잘 정의된 값은 아니다.:
```
In [38]:A or B
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-38-5d8e4f2e21c0> in <module>()
----> 1 A or B

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

Similarly, when doing a Boolean expression on a given array, you should use | or & rather than or or and:
```
In [39]:x = np.arange(10)
(x > 4) & (x < 8)
Out[39]:array([False, False, False, False, False,  True,  True,  True, False, False], dtype=bool)
```

잔체 배열의 참이나 거짓을 평가하려고 하면 전에 본것과 동일한 ValueError 가 발생할 것이다,:
```
In [40]:(x > 4) and (x < 8)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-40-3d24f1ffd63d> in <module>()
----> 1 (x > 4) and (x < 8)

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

And 와 or 는 전체 객체에 대해 단일 부울 평가를 수행하며, &와 |는 객체의 내용(개별 비트나 바이트)에 대해 여러 번 부울 평가를 수행한다. 부울 NumPy 배열에서는 대부분 후자를 선호한다.