

Comparisons, Masks, and Boolean Logic

This chapter covers the use of **Boolean masks** to **examine** and **manipulate values** within NumPy arrays.

Masking comes up when you want to **extract, modify, count**, or otherwise **manipulate** values in an array **based on some criterion**:

For **example**, you might wish to **count all values greater than** a certain value, or **remove all outliers** that are above some threshold.

In NumPy, **Boolean masking** is often the **most efficient** way to accomplish these types of tasks.

Example: Counting Rainy Days

Imagine you have **a series of data** that represents **the amount of precipitation** each day for a year in a given city.

For **example**, here we'll **load** the daily rainfall statistics for the city of Seattle in 2015, using **Pandas** (see Part 3):

```
In [ ]: import numpy as np
        from vega_datasets import data

        # Use DataFrame operations to extract rainfall as a NumPy array
        rainfall_mm = np.array(
            data.seattle_weather().set_index('date')['precipitation']
        )
        len(rainfall_mm)
```

```
Out[ ]: 365
```

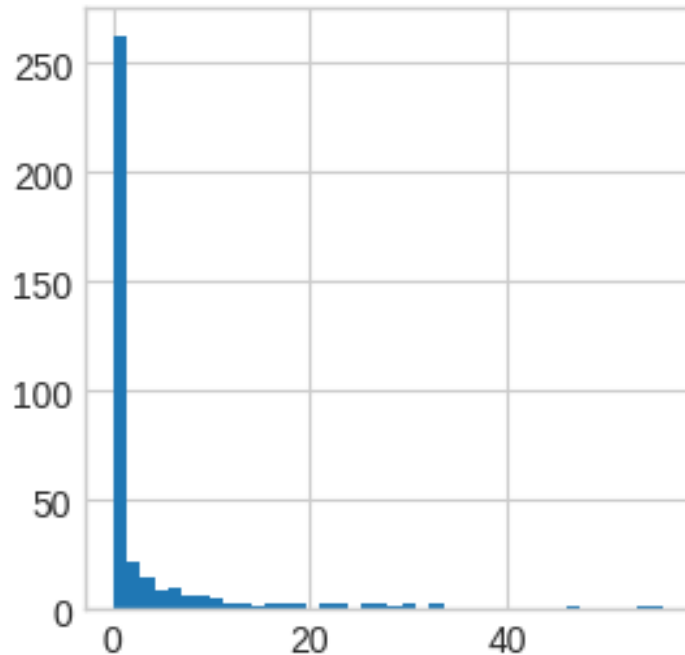
The array contains **365 values**, giving daily **rainfall in millimeters from January 1 to December 31, 2015**.

As a first **quick visualization**, let's look at the **histogram** of rainy days in the following figure, which was generated using **Matplotlib** (we will explore this tool more fully in Part 4):

```
In [ ]: %matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
```

```
<ipython-input-1-444997ff90a3>:3: MatplotlibDeprecationWarning:
The seaborn styles shipped by Matplotlib are deprecated since
3.6, as they no longer correspond to the styles shipped by seaborn.
However, they will remain available as 'seaborn-v0_8-<style>'.
Alternatively, directly use the seaborn API instead.
  plt.style.use('seaborn-whitegrid')
```

```
In [ ]: plt.hist(rainfall_mm, 40);  
plt.rcParams['figure.figsize'] = [2, 2]
```



This **histogram** gives us a **general idea** of what the data looks like:

Despite the city's **rainy reputation**, the vast **majority of days in Seattle saw near zero** measured rainfall in 2015.

But this **doesn't** do a good job of **conveying some information** we'd like to see. For example:

- How many rainy days were there in the year?
- What was the average precipitation on those rainy days?
- How many days were there with more than 10 mm of rainfall?

One approach to this would be to answer these questions **by hand**:

We could **loop through the data**, incrementing a counter each time we see values in some desired range.

But for reasons discussed throughout this chapter, **such an approach is very inefficient** from the standpoint of both **time writing code** and **time computing** the result.

We saw in Computation on NumPy Arrays: Universal Functions that NumPy's **ufuncs can be used in place of loops** to do **fast element-wise arithmetic** operations on arrays.

In the same way, we can use other **ufuncs** to do **element-wise comparisons** over arrays, and we can **then manipulate the results** to **answer the questions** we have.

We'll leave the data aside for now, and discuss some **general tools** in NumPy to use **masking** to **quickly answer these types of questions**.

Comparison Operators as Ufuncs

Computation on NumPy Arrays: Universal Functions introduced **ufuncs**, and focused in particular on arithmetic operators.

We saw that using `+`, `-`, `*`, `/`, and other operators on arrays leads to **element-wise operations**.

NumPy also implements **comparison operators** such as `<` (**less than**) and `>` (**greater than**) as **element-wise ufuncs**.

The **result** of these comparison operators is always **an array with a Boolean data type**.

All **six of the standard comparison operations** are available:

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

```
In [ ]: x < 3  # less than
```

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

```
In [ ]: x > 3  # greater than
```

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

```
In [ ]: x <= 3  # less than or equal
```

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

```
In [ ]: x >= 3  # greater than or equal
```

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

```
In [ ]: x != 3  # not equal
```

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

```
In [ ]: x == 3  # equal
```

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


It is also possible to do an **element-wise comparison** of **two arrays**, and to include compound expressions:

```
In [ ]: (2 * x) == (x ** 2)
```

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

```
In [ ]: 2 * x
```

```
Out[ ]: array([ 2,  4,  6,  8, 10])
```

```
In [ ]: x ** 2
```

```
Out[ ]: array([ 1,  4,  9, 16, 25])
```

As in the case of arithmetic operators, the **comparison operators** are implemented as **ufuncs** in NumPy.

For example, when you write `x < 3`, internally NumPy uses `np.less(x, 3)`.

A **summary of the comparison operators** and their **equivalent ufuncs** is shown here:

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

Just as in the case of arithmetic ufuncs, these **will work on arrays of any size and shape**.

Here is a **two-dimensional example**:

```
In [ ]: rng = np.random.default_rng(seed=1701)
x = rng.integers(10, size=(3, 4))
x
```

```
Out[ ]: array([[9, 4, 0, 3],
               [8, 6, 3, 1],
               [3, 7, 4, 0]])
```

```
In [ ]: x < 6
```

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

In each case, the **result is a Boolean array**, and **NumPy provides a number of straightforward patterns for working with these Boolean results**.

Working with Boolean Arrays

Given a **Boolean array**, there are a **host of useful operations** you can do.

We'll work with `x`, the two-dimensional array we created earlier:

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

```
[[9 4 0 3]
 [8 6 3 1]
 [3 7 4 0]]
```

Counting Entries

To **count the number of True entries** in a Boolean array, `np.count_nonzero` is useful:

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

```
Out[ ]: 8
```

We see that there are **eight array entries that are less than 6**.

Another way to get at this information is to use `np.sum`.

In this case, `False` is interpreted as `0`, and `True` is interpreted as `1`:

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

```
Out[ ]: 8
```

The **benefit of `np.sum`** is that, like with other NumPy aggregation functions, this summation can be done **along rows or columns as well**:

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

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

This **counts the number of values less than 6 in each row of the matrix**.

If we're interested in quickly **checking whether any or all the values are `True`**, we can use (you guessed it) **`np.any`** or **`np.all`**:

```
In [ ]: # are there any values greater than 8?  
np.any(x > 8)
```

Out[]: True

```
In [ ]: # are there any values less than zero?  
np.any(x < 0)
```

Out[]: False

```
In [ ]: # are all values less than 10?  
np.all(x < 10)
```

Out[]: True

```
In [ ]: # are all values equal to 6?  
np.all(x == 6)
```

Out[]: False

np.all and **np.any** can be used **along particular axes** as well.

For example:

```
In [ ]: # are all values in each row less than 8?  
np.all(x < 8, axis=1)
```

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

Here all the elements in the third row are less than 8, while this is not the case for others.

Finally, a quick warning: as mentioned in [Aggregations: Min, Max, and Everything In Between](#),

Python has built-in `sum`, `any`, and `all` functions.

These have a **different syntax than the NumPy** versions, and in particular will fail or produce unintended results when used on **multidimensional arrays**.

Be sure that you are using `np.sum`, `np.any`, and `np.all` for these examples!

Boolean Operators

We've already seen how we might **count**, say, **all days with less than 20 mm of rain**, or **all days with more than 10 mm of rain**.

But what if we want to know **how many days there were with more than 10 mm and less than 20 mm of rain**?

We can accomplish this with Python's **bitwise logic operators**, `&`, `|`, `^`, and `~`.

Like with the standard arithmetic operators, **NumPy overloads these as ufuncs that work element-wise on (usually Boolean) arrays**.

For example, we can address this sort of **compound question** as follows:

```
In [ ]: np.sum((rainfall_mm > 10) & (rainfall_mm < 20))
```

```
Out[ ]: 16
```

This tells us that there were **16 days with rainfall of between 10 and 20 millimeters.**

```
In [ ]: rainfall_mm > 10
```

```
Out[ ]: array([[False, False, False,  True, False, False, False, Fals  
e, False,  
        False, False, False, False, False, False, False, Tru  
e,  True,  
        False, False, False, False, False, False, False, Fals  
e, False,  
        False, False, False, False, False, False, False, Fals  
e,  True,  
        True,  True, False, False, False, False, False, Fals  
e, False,  
        False, False, False, False, False, False, False, Fals  
e, False,  
        False, False, False,  True, False, False, False, Fals  
e, False,  
        False, False, False, False, False, False, False, Fals  
e, False,  
        True,  True, False, False, False, False, False, Fals  
e, False,  
        False, False, False, False, False, False, False, Fals  
e, False,  
        False, False, False, False, False, False, False, Fals
```

[illegible]

False, False, False, False, False, False, False, False, False,
False,
False, False, False, False, False, False, False, False, False,
False,
False, False, False, False, False, False, False, False, False,
False,
True, False, False, False, False, False, False, False, False,
False,
False, False, False, False, False, False, False, True, True,
False,
False, False, False, False, False, False, False, False, False,
False,
False, False, False, False, False, False, False, False, False,
False,
False, False, False, False, False, False, False, False, False,
False,
False, False, False, False, False, False, False, False, False,
False,
False, False, False, True, False, False, False, False, False

```
e, False,
    False, False, False, False, False, False, False, False, False,
e, False,
    False, False, False, False, False, True, True, True,
e, False,
    False, False, False, False, True, False, False, False,
e, False,
    False, True, True, True, False, True, False, False,
e, False,
    False, False, False, False, False, False, False, False,
e, False,
    False, True, False, True, False, True, True, True,
e, True,
    True, False, False, True, False, False, False, False,
e, True,
    True, False, False, True, False, False, False, False,
e, False,
    False, False, False, False, False])
```

The **parentheses here are important**. Because of **operator precedence rules**, with the **parentheses removed** this expression would be evaluated as follows, which results in an **error**:

```
rainfall_mm > (10 & rainfall_mm) < 20
```

Let's demonstrate a more **complicated expression**. Using **De Morgan's laws**, we can compute the **same result in a different manner**:

```
In [ ]: np.sum(~( (rainfall_mm <= 10) | (rainfall_mm >= 20) ))
```

```
Out[ ]: 16
```

Combining comparison operators and Boolean operators on arrays can lead to a **wide range of efficient logical operations**.

The following **table** summarizes the bitwise **Boolean operators** and their **equivalent ufuncs**:

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

Using **these tools**, we can start to **answer many of the questions** we might have about our weather data.

Here are **some examples** of results we can compute when **combining masking with aggregations**:

```
In [ ]: print("Number days without rain: ", np.sum(rainfall_mm == 0))
        print("Number days with rain:      ", np.sum(rainfall_mm != 0))
        print("Days with more than 10 mm: ", np.sum(rainfall_mm > 10))
```



```
print("Rainy days with < 5 mm: ", np.sum((rainfall_mm > 0)  
                                           (rainfall_mm < 5)))
```

```
Number days without rain: 221  
Number days with rain: 144  
Days with more than 10 mm: 34  
Rainy days with < 5 mm: 83
```

Boolean Arrays as Masks

In the preceding section we looked at **aggregates computed directly on Boolean arrays**.

A **more powerful** pattern is to use **Boolean arrays as masks**, to **select** particular **subsets of the data** themselves.

```
In [ ]: x
```

```
Out[ ]: array([[9, 4, 0, 3],
               [8, 6, 3, 1],
               [3, 7, 4, 0]])
```

Suppose we want an **array of all values in the array that are less than, say, 5.**

We can **obtain a Boolean array for this condition** easily, as we've already seen:

```
In [ ]: x < 5
```

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

Now, to **select** these values from the array, we can simply **index on this Boolean array**; this is known as a **masking** operation:

```
In [ ]: x[x < 5]
```

```
Out[ ]: array([4, 0, 3, 3, 1, 3, 4, 0])
```

What is **returned is a one-dimensional array** filled with **all the values** that **meet this condition**.

In other words, all the values in **positions at which the mask array is True** .

We are then **free to operate on these values** as we wish.

For example, we can compute some relevant statistics on our Seattle rain data:

```
In [ ]: # construct a mask of all rainy days
rainy = (rainfall_mm > 0)

# construct a mask of all summer days (June 21st is the 172nd
```

```
days = np.arange(365)
summer = (days > 172) & (days < 262)

print("Median precip on rainy days in 2015 (mm): ",
      np.median(rainfall_mm[rainy]))
print("Median precip on summer days in 2015 (mm): ",
      np.median(rainfall_mm[summer]))
print("Maximum precip on summer days in 2015 (mm): ",
      np.max(rainfall_mm[summer]))
print("Median precip on non-summer rainy days (mm):",
      np.median(rainfall_mm[rainy & ~summer]))
```

Median precip on rainy days in 2015 (mm): 3.8
Median precip on summer days in 2015 (mm): 0.0
Maximum precip on summer days in 2015 (mm): 32.5
Median precip on non-summer rainy days (mm): 4.1

By combining **Boolean operations**, **masking operations**, and **aggregates**, we can very quickly answer these sorts of questions about our dataset.

Using the Keywords and/or Versus the Operators &/|

One common point of **confusion** is the **difference** between the **keywords** `and` and `or` on the one hand, and the **operators** `&` and `|` on the other.

When would you **use** one versus the other?

The difference is this: `and` and `or` **operate on the object as a whole**, while `&` and `|` **operate on the elements within the object**.

When you use `and` or `or`, it is equivalent to asking Python to treat the **object** as a **single Boolean entity**.

In **Python**, all **nonzero integers will evaluate as** `True`. Thus:

```
In [ ]: bool(42), bool(0)
```

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

```
In [ ]: bool(42 and 0)
```

```
Out[ ]: False
```

```
In [ ]: bool(42 or 0)
```

```
Out[ ]: True
```

When you use `&` **and** `|` on integers, the expression operates on the **bitwise representation of the element**, applying the **and** or the **or** to the **individual bits making up the number**:

```
In [ ]: bin(42)
```

```
Out[ ]: '0b101010'
```

```
In [ ]: bin(59)
```

```
Out[ ]: '0b111011'
```

```
In [ ]: bin(42 & 59)
```

```
Out[ ]: '0b101010'
```

```
In [ ]: bin(42 | 59)
```

```
Out[ ]: '0b111011'
```

Notice that the **corresponding bits** of the **binary representation** are **compared** in order to yield the result.

When you have an **array of Boolean values** in NumPy, this can be thought of as a **string of bits** where `1 = True` and `0 = False`,

and `&` and `|` will operate **similarly to in the preceding examples:**

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

But if you use `or` **on these arrays** it will try to evaluate the truth or falsehood of the entire array object, which is **not a well-defined value:**

```
In [ ]: A or B
```



```
-----  
-----  
ValueError                                Traceback (most recent  
call last)  
/var/folders/xs/sptt9bk14s34rgxt7453p03r0000gp/T/ipykernel_9301  
0/3447948156.py in <module>  
----> 1 A or B  
  
ValueError: The truth value of an array with more than one elem  
ent is ambiguous. Use a.any() or a.all()
```

Similarly, when **evaluating a Boolean expression on a given array**,
you should use **|** or **&** rather than **or** or **and**:

```
In [ ]: x = np.arange(10)  
        (x > 4) & (x < 8)
```

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

Trying to **evaluate** the truth or falsehood of **the entire array** will give the same **ValueError** we saw previously:

```
In [ ]: (x > 4) and (x < 8)
```

```
-----  
-----  
ValueError                                Traceback (most recent  
call last)  
/var/folders/xs/sptt9bk14s34rgxt7453p03r0000gp/T/ipykernel_9301  
0/2869511139.py in <module>  
----> 1 (x > 4) and (x < 8)  
  
ValueError: The truth value of an array with more than one elem  
ent is ambiguous. Use a.any() or a.all()
```

So, remember this:

`and` `and` `or` perform a **single Boolean evaluation** on an entire object.

While `&` `and` `|` perform multiple Boolean **evaluations on the content** (the individual bits or bytes) of an object.

For **Boolean NumPy arrays, the latter is nearly always the desired operation.**