Lecture 4-3

More NumPy

Week 5 Friday

Miles Chen, PhD

Based on Python Data Science Handbook by Jake VanderPlas

In [1]:
```
import numpy as np
```

Concatenating Arrays

Concatenating Arrays

In [2]:

```
x = np.arange(4)
y = np.arange(100, 104)
print(x)
print(y)
```

```
[0 1 2 3]
[100 101 102 103]
```

Concatenating Arrays

In [2]:

```
x = np.arange(4)
y = np.arange(100, 104)
print(x)
print(y)
```

```
[0 1 2 3]
[100 101 102 103]
```

In [3]:

```
np.concatenate([x,y])
```

Out[3]:

```
array([  0,   1,   2,   3, 100, 101, 102, 103])
```

Concatenating Arrays

In [2]:

```
x = np.arange(4)
y = np.arange(100, 104)
print(x)
print(y)
```

```
[0 1 2 3]
[100 101 102 103]
```

In [3]:

```
np.concatenate([x,y])
```

Out[3]:

```
array([  0,   1,   2,   3, 100, 101, 102, 103])
```

`np.concatenate` has an argument for axis. The axes are 0-indexed.

Concatenating Arrays

In [2]:

```
x = np.arange(4)
y = np.arange(100, 104)
print(x)
print(y)
```

```
[0 1 2 3]
[100 101 102 103]
```

In [3]:

```
np.concatenate([x,y])
```

Out[3]:

```
array([  0,   1,   2,   3, 100, 101, 102, 103])
```

`np.concatenate` has an argument for axis. The axes are 0-indexed.

In [4]:

```
np.concatenate([x,y], axis = 0)
```

Out[4]:

```
array([  0,   1,   2,   3, 100, 101, 102, 103])
```

Let's try to concatenate in the other direction. We specify axis = 1

Let's try to concatenate in the other direction. We specify axis = 1

In [5]:
```
np.concatenate([x,y], axis = 1) # throws an error
```

```
---------------------------------------------------------------------------
AxisError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_25452/1729112478.py in <module>
----> 1 np.concatenate([x,y], axis = 1) # throws an error

<__array_function__ internals> in concatenate(*args, **kwargs)

AxisError: axis 1 is out of bounds for array of dimension 1
```

Let's try to concatenate in the other direction. We specify axis = 1

In [5]:
```
np.concatenate([x,y], axis = 1) # throws an error
```

```
---------------------------------------------------------------------------
AxisError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_25452/1729112478.py in <module>
----> 1 np.concatenate([x,y], axis = 1) # throws an error

<__array_function__ internals> in concatenate(*args, **kwargs)

AxisError: axis 1 is out of bounds for array of dimension 1
```

In [6]:
```
x.shape # you can't use axis with index 1, because axis index 1 does not exist
```

Out[6]:
```
(4,)
```

Let's try to concatenate in the other direction. We specify axis = 1

In [5]:
```
np.concatenate([x,y], axis = 1) # throws an error
```

```
---------------------------------------------------------------------------
AxisError                                 Traceback (most recent call las
t)
~\AppData\Local\Temp/ipykernel_25452/1729112478.py in <module>
----> 1 np.concatenate([x,y], axis = 1) # throws an error

<__array_function__ internals> in concatenate(*args, **kwargs)

AxisError: axis 1 is out of bounds for array of dimension 1
```

In [6]:
```
x.shape # you can't use axis with index 1, because axis index 1 does not exist
```

Out[6]:
```
(4,)
```

In [7]:
```
np.vstack([x,y])   # vstack will vertically stack unidimensional arrays
```

Out[7]:
```
array([[  0,   1,   2,   3],
       [100, 101, 102, 103]])
```

In [8]:

```
x.reshape(1,4)
```

Out[8]:

```
array([[0, 1, 2, 3]])
```

In [8]:
```
x.reshape(1,4)
```

Out[8]:
```
array([[0, 1, 2, 3]])
```

In [9]:
```
y.reshape(1,4)
```

Out[9]:
```
array([[100, 101, 102, 103]])
```

In [8]:
```
x.reshape(1,4)
```
Out[8]:
```
array([[0, 1, 2, 3]])
```
In [9]:
```
y.reshape(1,4)
```
Out[9]:
```
array([[100, 101, 102, 103]])
```
In [10]:
```
np.concatenate([x.reshape(1,4), y.reshape(1,4)], axis = 0)
```
Out[10]:
```
array([[  0,   1,   2,   3],
       [100, 101, 102, 103]])
```

In [8]:
```
x.reshape(1,4)
```

Out[8]:
```
array([[0, 1, 2, 3]])
```

In [9]:
```
y.reshape(1,4)
```

Out[9]:
```
array([[100, 101, 102, 103]])
```

In [10]:
```
np.concatenate([x.reshape(1,4), y.reshape(1,4)], axis = 0)
```

Out[10]:
```
array([[  0,   1,   2,   3],
       [100, 101, 102, 103]])
```

note that when I concatenate along axis 0 for a 2-dimensional array, it concatenates by rows. In a 2D array, index 0 is for rows, and index 1 is for columns.

In [8]:
```python
x.reshape(1,4)
```

Out[8]:
```
array([[0, 1, 2, 3]])
```

In [9]:
```python
y.reshape(1,4)
```

Out[9]:
```
array([[100, 101, 102, 103]])
```

In [10]:
```python
np.concatenate([x.reshape(1,4), y.reshape(1,4)], axis = 0)
```

Out[10]:
```
array([[  0,   1,   2,   3],
       [100, 101, 102, 103]])
```

note that when I concatenate along axis 0 for a 2-dimensional array, it concatenates by rows. In a 2D array, index 0 is for rows, and index 1 is for columns.

In [11]:
```python
np.concatenate([x.reshape(1,4), y.reshape(1,4)], axis = 1)
```

Out[11]:
```
array([[  0,   1,   2,   3, 100, 101, 102, 103]])
```

In [12]:

```
xm = np.arange(6).reshape((2,3))
ym = np.arange(100,106,1).reshape((2,3))
print(xm)
print(ym)
```

```
[[0 1 2]
 [3 4 5]]
[[100 101 102]
 [103 104 105]]
```

In [12]:
```
xm = np.arange(6).reshape((2,3))
ym = np.arange(100,106,1).reshape((2,3))
print(xm)
print(ym)
```

```
[[0 1 2]
 [3 4 5]]
[[100 101 102]
 [103 104 105]]
```

In [13]:
```
xm.shape
```

Out[13]:
```
(2, 3)
```

In [12]:

```
xm = np.arange(6).reshape((2,3))
ym = np.arange(100,106,1).reshape((2,3))
print(xm)
print(ym)
```

```
[[0 1 2]
 [3 4 5]]
[[100 101 102]
 [103 104 105]]
```

In [13]:

```
xm.shape
```

Out[13]:

```
(2, 3)
```

In [14]:

```
ym.shape
```

Out[14]:

```
(2, 3)
```

In [15]:

```
print(np.concatenate([xm,ym])) # default behavior concatenates on axis 0
```

```
[[  0   1   2]
 [  3   4   5]
 [100 101 102]
 [103 104 105]]
```

In [15]:

```
print(np.concatenate([xm,ym])) # default behavior concatenates on axis 0
```

```
[[  0   1   2]
 [  3   4   5]
 [100 101 102]
 [103 104 105]]
```

In [16]:

```
print(np.concatenate([xm,ym], axis = 0))
# axes are reported as rows, then columns.
# concatenating along axis 0 will concatenate along rows
```

```
[[  0   1   2]
 [  3   4   5]
 [100 101 102]
 [103 104 105]]
```

In [15]:

```
print(np.concatenate([xm,ym])) # default behavior concatenates on axis 0
```

```
[[  0   1   2]
 [  3   4   5]
 [100 101 102]
 [103 104 105]]
```

In [16]:

```
print(np.concatenate([xm,ym], axis = 0))
# axes are reported as rows, then columns.
# concatenating along axis 0 will concatenate along rows
```

```
[[  0   1   2]
 [  3   4   5]
 [100 101 102]
 [103 104 105]]
```

In [17]:

```
print(np.concatenate([xm,ym], axis = 1))
# concatenating along axis 1 will concatenate along columns
```

```
[[  0   1   2 100 101 102]
 [  3   4   5 103 104 105]]
```

In [18]:

```python
np.vstack([xm, ym])
```

Out[18]:

```
array([[  0,   1,   2],
       [  3,   4,   5],
       [100, 101, 102],
       [103, 104, 105]])
```

In [18]:
```
np.vstack([xm, ym])
```

Out[18]:
```
array([[  0,   1,   2],
       [  3,   4,   5],
       [100, 101, 102],
       [103, 104, 105]])
```

In [19]:
```
np.hstack([xm, ym])
```

Out[19]:
```
array([[  0,   1,   2, 100, 101, 102],
       [  3,   4,   5, 103, 104, 105]])
```

In [18]:
```python
np.vstack([xm, ym])
```

Out[18]:
```
array([[  0,   1,   2],
       [  3,   4,   5],
       [100, 101, 102],
       [103, 104, 105]])
```

In [19]:
```python
np.hstack([xm, ym])
```

Out[19]:
```
array([[  0,   1,   2, 100, 101, 102],
       [  3,   4,   5, 103, 104, 105]])
```

You can always use vstack and hstack for 2D arrays.

Math Operators with numpy arrays

Math Operators with numpy arrays

In [20]:

```
print(x)
print(y)
```

```
[0 1 2 3]
[100 101 102 103]
```

Math Operators with numpy arrays

In [20]:
```
print(x)
print(y)
```

```
[0 1 2 3]
[100 101 102 103]
```

In [21]:
```
x + 5
```

Out[21]:
```
array([5, 6, 7, 8])
```

Math Operators with numpy arrays

In [20]:

```
print(x)
print(y)
```

```
[0 1 2 3]
[100 101 102 103]
```

In [21]:

```
x + 5
```

Out[21]:

```
array([5, 6, 7, 8])
```

In [22]:

```
x + y  # elementwise addition
```

Out[22]:

```
array([100, 102, 104, 106])
```

Math Operators with numpy arrays

In [20]:
```
print(x)
print(y)
```

```
[0 1 2 3]
[100 101 102 103]
```

In [21]:
```
x + 5
```

Out[21]:
```
array([5, 6, 7, 8])
```

In [22]:
```
x + y  # elementwise addition
```

Out[22]:
```
array([100, 102, 104, 106])
```

In [23]:
```
x * y
```

Out[23]:
```
array([  0, 101, 204, 309])
```

Math Operators with numpy arrays

In [20]:
```
print(x)
print(y)
```

```
[0 1 2 3]
[100 101 102 103]
```

In [21]:
```
x + 5
```

Out[21]:
```
array([5, 6, 7, 8])
```

In [22]:
```
x + y  # elementwise addition
```

Out[22]:
```
array([100, 102, 104, 106])
```

In [23]:
```
x * y
```

Out[23]:
```
array([  0, 101, 204, 309])
```

In [24]:
```
np.sum(x * y)
```

Out[24]:
```
614
```

In [25]:
```
np.dot(x,y)    # 0 * 100 + 1 * 101 + 2 * 102 + 3 * 103
```

Out[25]:
```
614
```

In [25]:
```
np.dot(x,y)   # 0 * 100 + 1 * 101 + 2 * 102 + 3 * 103
```
Out[25]:
```
614
```
In [26]:
```
x @ y # matrix multiplication
```
Out[26]:
```
614
```

In [27]:

```
print(xm)
print(ym)
```

```
[[0 1 2]
 [3 4 5]]
[[100 101 102]
 [103 104 105]]
```

In [27]:
```
print(xm)
print(ym)
```

```
[[0 1 2]
 [3 4 5]]
[[100 101 102]
 [103 104 105]]
```

In [28]:
```
xm + 5
```

Out[28]:
```
array([[ 5,  6,  7],
       [ 8,  9, 10]])
```

In [27]:

```
print(xm)
print(ym)
```

```
[[0 1 2]
 [3 4 5]]
[[100 101 102]
 [103 104 105]]
```

In [28]:

```
xm + 5
```

Out[28]:

```
array([[ 5,  6,  7],
       [ 8,  9, 10]])
```

In [29]:

```
xm + ym  # elementwise addition
```

Out[29]:

```
array([[100, 102, 104],
       [106, 108, 110]])
```

In [30]:

```
print(xm)
print(ym)
```

```
[[0 1 2]
 [3 4 5]]
[[100 101 102]
 [103 104 105]]
```

In [30]:
```
print(xm)
print(ym)
```

```
[[0 1 2]
 [3 4 5]]
[[100 101 102]
 [103 104 105]]
```

In [31]:
```
xm * ym # element-wise multiplication
```

Out[31]:
```
array([[  0, 101, 204],
       [309, 416, 525]])
```

In [30]:

```
print(xm)
print(ym)
```

```
[[0 1 2]
 [3 4 5]]
[[100 101 102]
 [103 104 105]]
```

In [31]:

```
xm * ym # element-wise multiplication
```

Out[31]:

```
array([[  0, 101, 204],
       [309, 416, 525]])
```

In [32]:

```
np.multiply(xm, ym) # element-wise multiplication
```

Out[32]:

```
array([[  0, 101, 204],
       [309, 416, 525]])
```

In [33]:

```
print(xm)
print(ym)
```

```
[[0 1 2]
 [3 4 5]]
[[100 101 102]
 [103 104 105]]
```

In [33]:

```
print(xm)
print(ym)
```

```
[[0 1 2]
 [3 4 5]]
[[100 101 102]
 [103 104 105]]
```

In [34]:

```
np.dot(xm, ym.T)
```

Out[34]:

```
array([[ 305,  314],
       [1214, 1250]])
```

In [33]:

```
print(xm)
print(ym)
```

```
[[0 1 2]
 [3 4 5]]
[[100 101 102]
 [103 104 105]]
```

In [34]:

```
np.dot(xm, ym.T)
```

Out[34]:

```
array([[ 305,  314],
       [1214, 1250]])
```

In [35]:

```
xm.dot(ym.T)
```

Out[35]:

```
array([[ 305,  314],
       [1214, 1250]])
```

In [33]:
```
print(xm)
print(ym)
```

```
[[0 1 2]
 [3 4 5]]
[[100 101 102]
 [103 104 105]]
```

In [34]:
```
np.dot(xm, ym.T)
```

Out[34]:
```
array([[ 305,  314],
       [1214, 1250]])
```

In [35]:
```
xm.dot(ym.T)
```

Out[35]:
```
array([[ 305,  314],
       [1214, 1250]])
```

In [36]:
```
xm @ ym.T
```

Out[36]:
```
array([[ 305,  314],
       [1214, 1250]])
```

Basic Math

Basic Math

In [37]:
```
x = np.arange(4)
print(x)
```

```
[0 1 2 3]
```

Basic Math

In [37]:

```
x = np.arange(4)
print(x)
```

```
[0 1 2 3]
```

In [38]:

```
print(x + 4)
```

```
[4 5 6 7]
```

Basic Math

In [37]:
```
x = np.arange(4)
print(x)
```

```
[0 1 2 3]
```

In [38]:
```
print(x + 4)
```

```
[4 5 6 7]
```

In [39]:
```
print(x - 5)
```

```
[-5 -4 -3 -2]
```

Basic Math

In [37]:
```
x = np.arange(4)
print(x)
```

```
[0 1 2 3]
```

In [38]:
```
print(x + 4)
```

```
[4 5 6 7]
```

In [39]:
```
print(x - 5)
```

```
[-5 -4 -3 -2]
```

In [40]:
```
print(x * 2)
```

```
[0 2 4 6]
```

In [41]:

```
print(x / 2)
```

```
[0.  0.5 1.  1.5]
```

In [41]:

```
print(x / 2)
```

```
[0.  0.5 1.  1.5]
```

In [42]:

```
print(-x)
```

```
[ 0 -1 -2 -3]
```

In [41]:
```
print(x / 2)
```

```
[0.  0.5 1.  1.5]
```

In [42]:
```
print(-x)
```

```
[ 0 -1 -2 -3]
```

In [43]:
```
print(x ** 2)
```

```
[0 1 4 9]
```

In [41]:
```
print(x / 2)
```

```
[0.  0.5 1.  1.5]
```

In [42]:
```
print(-x)
```

```
[ 0 -1 -2 -3]
```

In [43]:
```
print(x ** 2)
```

```
[0 1 4 9]
```

In [44]:
```
print(x % 2) # modulo division
```

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

In [41]:
```
print(x / 2)
```

```
[0.  0.5 1.  1.5]
```

In [42]:
```
print(-x)
```

```
[ 0 -1 -2 -3]
```

In [43]:
```
print(x ** 2)
```

```
[0 1 4 9]
```

In [44]:
```
print(x % 2) # modulo division
```

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

In [45]:
```
print(abs(x)) # abs
```

```
[0 1 2 3]
```

Trig functions

note that the functions are preceeded by np.

Trig functions

note that the functions are preceeded by np.

In [46]:

```
theta = np.linspace(0, np.pi, 5)
print(theta)
```

```
[0.         0.78539816 1.57079633 2.35619449 3.14159265]
```

Trig functions

note that the functions are preceeded by np.

In [46]:

```
theta = np.linspace(0, np.pi, 5)
print(theta)
```

```
[0.         0.78539816 1.57079633 2.35619449 3.14159265]
```

In [47]:

```
print(np.sin(theta))
```

```
[0.00000000e+00 7.07106781e-01 1.00000000e+00 7.07106781e-01
 1.22464680e-16]
```

Trig functions

note that the functions are preceeded by np.

In [46]:

```
theta = np.linspace(0, np.pi, 5)
print(theta)
```

```
[0.         0.78539816 1.57079633 2.35619449 3.14159265]
```

In [47]:

```
print(np.sin(theta))
```

```
[0.00000000e+00 7.07106781e-01 1.00000000e+00 7.07106781e-01
 1.22464680e-16]
```

In [48]:

```
print(np.cos(theta))
```

```
[ 1.00000000e+00  7.07106781e-01  6.12323400e-17 -7.07106781e-01
 -1.00000000e+00]
```

Trig functions

note that the functions are preceeded by np.

In [46]:
```
theta = np.linspace(0, np.pi, 5)
print(theta)
```

```
[0.         0.78539816 1.57079633 2.35619449 3.14159265]
```

In [47]:
```
print(np.sin(theta))
```

```
[0.00000000e+00 7.07106781e-01 1.00000000e+00 7.07106781e-01
 1.22464680e-16]
```

In [48]:
```
print(np.cos(theta))
```

```
[ 1.00000000e+00  7.07106781e-01  6.12323400e-17 -7.07106781e-01
 -1.00000000e+00]
```

In [49]:
```
print(np.tan(theta))
```

```
[ 0.00000000e+00  1.00000000e+00  1.63312394e+16 -1.00000000e+00
 -1.22464680e-16]
```

Log and Exp

Log and Exp

In [50]:

```
x = np.array([1, 10, 100])
print(np.log(x))    # natural log
print(np.log10(x)) # common log
```

```
[0.         2.30258509 4.60517019]
[0. 1. 2.]
```

Log and Exp

In [50]:

```
x = np.array([1, 10, 100])
print(np.log(x))    # natural log
print(np.log10(x)) # common log
```

```
[0.         2.30258509 4.60517019]
[0. 1. 2.]
```

In [51]:

```
y = np.arange(3)
print(np.exp(y))   # e^y
```

```
[1.         2.71828183 7.3890561 ]
```

Log and Exp

In [50]:

```
x = np.array([1, 10, 100])
print(np.log(x))    # natural log
print(np.log10(x)) # common log
```

```
[0.         2.30258509 4.60517019]
[0. 1. 2.]
```

In [51]:

```
y = np.arange(3)
print(np.exp(y))   # e^y
```

```
[1.         2.71828183 7.3890561 ]
```

In [52]:

```
print(np.exp2(y))   # 2^y
```

```
[1. 2. 4.]
```

Log and Exp

In [50]:

```
x = np.array([1, 10, 100])
print(np.log(x))    # natural log
print(np.log10(x)) # common log
```

```
[0.         2.30258509 4.60517019]
[0. 1. 2.]
```

In [51]:

```
y = np.arange(3)
print(np.exp(y))   # e^y
```

```
[1.         2.71828183 7.3890561 ]
```

In [52]:

```
print(np.exp2(y))   # 2^y
```

```
[1. 2. 4.]
```

In [53]:

```
print(np.power(3, y)) # power ^ y
```

```
[1 3 9]
```

Aggregates

you can use `sum()`

or `np.sum()`

`np.sum()` is faster than sum, but doesn't always behave the same way

Aggregates

you can use `sum()`

or `np.sum()`

`np.sum()` is faster than sum, but doesn't always behave the same way

In [54]:

```
x = np.arange(100)
print(x)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96 97 98 99]
```

Aggregates

you can use `sum()`

or `np.sum()`

`np.sum()` is faster than sum, but doesn't always behave the same way

In [54]:
```
x = np.arange(100)
print(x)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96 97 98 99]
```

In [55]:
```
print(sum(x))
```

```
4950
```

Aggregates

you can use `sum()`

or `np.sum()`

`np.sum()` is faster than sum, but doesn't always behave the same way

In [54]:

```
x = np.arange(100)
print(x)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96 97 98 99]
```

In [55]:

```
print(sum(x))
```

```
4950
```

In [56]:

```
print(np.sum(x))
```

```
4950
```

In [ ]:

```
big_array = np.random.rand(10000)
%timeit sum(big_array)
%timeit np.sum(big_array)  # the np version is much faster
```

min and max

min and max

In [ ]:

```
print(min(big_array))
print(max(big_array))
```

min and max

In [ ]:

```
print(min(big_array))
print(max(big_array))
```

In [ ]:

```
print(np.min(big_array))
print(np.max(big_array))
```

min and max

In [ ]:

```
print(min(big_array))
print(max(big_array))
```

In [ ]:

```
print(np.min(big_array))
print(np.max(big_array))
```

In [ ]:

```
%timeit min(big_array)
%timeit np.min(big_array)  # the np version is much faster
```

summaries for matrices

summaries for matrices

In [ ]:

```
np.random.seed(1)
# M = np.random.random((3, 4))
M = np.arange(12)
np.random.shuffle(M)
M = np.reshape(M, (3,4))
print(M)
```

summaries for matrices

In [ ]:

```
np.random.seed(1)
# M = np.random.random((3, 4))
M = np.arange(12)
np.random.shuffle(M)
M = np.reshape(M, (3,4))
print(M)
```

In [ ]:

```
sum(M) # regular sum function
```

summaries for matrices

In [ ]:

```
np.random.seed(1)
# M = np.random.random((3, 4))
M = np.arange(12)
np.random.shuffle(M)
M = np.reshape(M, (3,4))
print(M)
```

In [ ]:

```
sum(M) # regular sum function
```

In [ ]:

```
np.sum(M) # np.sum function
```

In [ ]:

```
print(M)
```

In [ ]:

```
print(M)
```

In [ ]:

```
np.sum(M, axis = 0)   # np.sum function with axis specified
# matrices have two dimensions
# 0 is rows, 1 is columns
# np.sum axis = 0, will sum over rows, so you end up getting column totals
```

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

In [ ]:
```
np.sum(M, axis = 0)   # np.sum function with axis specified
# matrices have two dimensions
# 0 is rows, 1 is columns
# np.sum axis = 0, will sum over rows, so you end up getting column totals
```

In [ ]:
```
np.sum(M, axis = 1)
```

In [ ]:

```
print(M)
```

In [ ]:

```
np.sum(M, axis = 0)   # np.sum function with axis specified
# matrices have two dimensions
# 0 is rows, 1 is columns
# np.sum axis = 0, will sum over rows, so you end up getting column totals
```

In [ ]:

```
np.sum(M, axis = 1)
```

In [ ]:

```
np.min(M, axis = 0)
```

In [ ]:

```
print(M)
```

In [ ]:

```
print(M)
```

In [ ]:

```
np.std(M)
```

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

In [ ]:
```
np.std(M)
```

In [ ]:
```
np.std(M, axis = 0)
```

In [ ]:

```
print(M)
```

In [ ]:

```
np.std(M)
```

In [ ]:

```
np.std(M, axis = 0)
```

In [ ]:

```
np.mean(M, axis = 1)
```

Summaries for higher dimensional arrays

Summaries for higher dimensional arrays

In [ ]:

```
np.random.seed(1)
A = np.ones(24)
np.random.shuffle(A)
A = np.reshape(A, (2, 3, 4)) # two sheets, 3 rows, 4 columns
print(A)
```

Summaries for higher dimensional arrays

In [ ]:

```
np.random.seed(1)
A = np.ones(24)
np.random.shuffle(A)
A = np.reshape(A, (2, 3, 4)) # two sheets, 3 rows, 4 columns
print(A)
```

In [ ]:

```
np.sum(A, axis = 0) # sum across "sheets"
```

Summaries for higher dimensional arrays

In [ ]:

```
np.random.seed(1)
A = np.ones(24)
np.random.shuffle(A)
A = np.reshape(A, (2, 3, 4)) # two sheets, 3 rows, 4 columns
print(A)
```

In [ ]:

```
np.sum(A, axis = 0) # sum across "sheets"
```

In [ ]:

```
np.sum(A, axis = 1) # sum across rows
```

Summaries for higher dimensional arrays

In [ ]:

```
np.random.seed(1)
A = np.ones(24)
np.random.shuffle(A)
A = np.reshape(A, (2, 3, 4)) # two sheets, 3 rows, 4 columns
print(A)
```

In [ ]:

```
np.sum(A, axis = 0) # sum across "sheets"
```

In [ ]:

```
np.sum(A, axis = 1) # sum across rows
```

In [ ]:

```
np.sum(A, axis = 2) # sum across columns
```

dealing with nan

nan is the float value for something that is not a number. We often use it in the place of a missing value. nan only exists in float type.

dealing with nan

nan is the float value for something that is not a number. We often use it in the place of a missing value. nan only exists in float type.

In [ ]:

```
x = float("nan")  # direct creation of nan
print(x)
print(type(x))
```

dealing with nan

nan is the float value for something that is not a number. We often use it in the place of a missing value. nan only exists in float type.

In [ ]:

```
x = float("nan")  # direct creation of nan
print(x)
print(type(x))
```

In [ ]:

```
y = float("inf")  # y is the float representation of infinity
print(y / y)  # these calculations will yield a nan result
print(y - y)
```

dealing with nan

nan is the float value for something that is not a number. We often use it in the place of a missing value. nan only exists in float type.

In [ ]:

```
x = float("nan")  # direct creation of nan
print(x)
print(type(x))
```

In [ ]:

```
y = float("inf")  # y is the float representation of infinity
print(y / y)  # these calculations will yield a nan result
print(y - y)
```

In [ ]:

```
np.sum([x, 2])
```

dealing with nan

nan is the float value for something that is not a number. We often use it in the place of a missing value. nan only exists in float type.

In [ ]:

```
x = float("nan")   # direct creation of nan
print(x)
print(type(x))
```

In [ ]:

```
y = float("inf")   # y is the float representation of infinity
print(y / y)   # these calculations will yield a nan result
print(y - y)
```

In [ ]:

```
np.sum([x, 2])
```

In [ ]:

```
np.nansum([x, 2])    # in R you have the option na.rm = TRUE
```

The following table provides a list of useful aggregation functions available in NumPy:

Function Name	NaN-safe Version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute mean of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

Broadcasting

This is a similar concept to recyling values in R, but only works when the dimensions are compatible

Broadcasting

This is a similar concept to recyling values in R, but only works when the dimensions are compatible

In [ ]:

```
a = np.array([1,2,3])
b = np.array([4,5,6])
print(a + b)
```

Broadcasting

This is a similar concept to recyling values in R, but only works when the dimensions are compatible

In [ ]:

```
a = np.array([1,2,3])
b = np.array([4,5,6])
print(a + b)
```

In [ ]:

```
c = np.array([7,8])
print(a + c)  # doesn't work
```

In [ ]:

```
print(a)
```

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

In [ ]:
```
e = np.ones([3,3])
print(e)
```

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

In [ ]:
```
e = np.ones([3,3])
print(e)
```

In [ ]:
```
print(e + a)   # the array a gets 'broadcast' across all three rows
```

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

In [ ]:
```
e = np.ones([3,3])
print(e)
```

In [ ]:
```
print(e + a)   # the array a gets 'broadcast' across all three rows
```

In [ ]:
```
print(a.reshape([3,1]))   # we reshape a to be a 3x1 array
```

In [ ]:

```
print(a)
```

In [ ]:

```
e = np.ones([3,3])
print(e)
```

In [ ]:

```
print(e + a)   # the array a gets 'broadcast' across all three rows
```

In [ ]:

```
print(a.reshape([3,1]))  # we reshape a to be a 3x1 array
```

In [ ]:

```
print(e + a.reshape([3,1])) # the reshaped array is broadcast across columns
```

In [ ]:

```
d = np.vstack([a,b])  # we stack the arrays a and b vertically
print(d)
```

In [ ]:

```
d = np.vstack([a,b])  # we stack the arrays a and b vertically
print(d)
```

In [ ]:

```
a
```

In [ ]:

```
d = np.vstack([a,b])  # we stack the arrays a and b vertically
print(d)
```

In [ ]:

```
a
```

In [ ]:

```
print(d + a)  # a is broadcast across row
```

In [ ]:

```
print(c)
```

In [ ]:

```
print(c)
```

In [ ]:

```
print(d)
```

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

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

In [ ]:
```
print(d + c)  # c does not have compatible dimensions
```

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

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

In [ ]:
```
print(d + c)   # c does not have compatible dimensions
```

In [ ]:
```
print(d + c.reshape([2,1]))   # after we reshape c to be a column, we can broadcast it
```

In [ ]:

```
e = np.arange(10).reshape((10, 1))
f = np.arange(11)
print(e)
print(f)
```

In [ ]:

```
e = np.arange(10).reshape((10, 1))
f = np.arange(11)
print(e)
print(f)
```

In [ ]:

```
print(e + f)  ## e and f are broadcast into compatible matrices and then added
```

In [ ]:
```
print(e * f)  ## e and f are broadcast into compatible matrices and then multiplied eleme
```

In [ ]:
```
print(e * f)  ## e and f are broadcast into compatible matrices and then multiplied eleme
```

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

In [ ]:
```
print(e * f)  ## e and f are broadcast into compatible matrices and then multiplied eleme
```

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

In [ ]:
```
d.reshape((1,6)) + d.reshape((6,1))
```

Boolean Operators in NumPy

Boolean Operators in NumPy

In [ ]:

```
x = np.arange(6)
print(x)
```

Boolean Operators in NumPy

In [ ]:

```
x = np.arange(6)
print(x)
```

In [ ]:

```
print(x < 3)
```

Boolean Operators in NumPy

In [ ]:

```
x = np.arange(6)
print(x)
```

In [ ]:

```
print(x < 3)
```

In [ ]:

```
print(x >= 3)
```

Boolean Operators in NumPy

In [ ]:

```
x = np.arange(6)
print(x)
```

In [ ]:

```
print(x < 3)
```

In [ ]:

```
print(x >= 3)
```

In [ ]:

```
print(x == 3)
```

In [ ]:

```
# the results can then be used to subset
print(x[x >= 3])
```

In [ ]:

```
# the results can then be used to subset
print(x[x >= 3])
```

In [ ]:

```
np.sum(x >= 3) # True = 1, False = 0, so sum counts how many are true
```

In [ ]:
```
# the results can then be used to subset
print(x[x >= 3])
```

In [ ]:
```
np.sum(x >= 3) # True = 1, False = 0, so sum counts how many are true
```

In [ ]:
```
np.mean(x >= 3)  # finds the proportion that is True
```

In [ ]:

```
# the results can then be used to subset
print(x[x >= 3])
```

In [ ]:

```
np.sum(x >= 3) # True = 1, False = 0, so sum counts how many are true
```

In [ ]:

```
np.mean(x >= 3)  # finds the proportion that is True
```

In [ ]:

```
print(~(x == 3)) # use the tilde for negation of boolean values
```

In [ ]:

```
print(~x == 3) # be careful if you leave off parenthesis
```

In [ ]:

```
print(~x == 3) # be careful if you leave off parenthesis
```

In [ ]:

```
~x
```

Working with matrices

Working with matrices

In [ ]:

```
y = np.arange(12).reshape([3,4])
print(y)
```

Working with matrices

In [ ]:

```
y = np.arange(12).reshape([3,4])
print(y)
```

In [ ]:

```
print(y >= 6)
```

Working with matrices

In [ ]:

```
y = np.arange(12).reshape([3,4])
print(y)
```

In [ ]:

```
print(y >= 6)
```

In [ ]:

```
np.sum(y >= 6)
```

Working with matrices

In [ ]:

```
y = np.arange(12).reshape([3,4])
print(y)
```

In [ ]:

```
print(y >= 6)
```

In [ ]:

```
np.sum(y >= 6)
```

In [ ]:

```
np.sum(y >= 6, axis = 0)  # you can perform sums and other aggregate functions axis-wise
```

Working with matrices

In [ ]:
```
y = np.arange(12).reshape([3,4])
print(y)
```

In [ ]:
```
print(y >= 6)
```

In [ ]:
```
np.sum(y >= 6)
```

In [ ]:
```
np.sum(y >= 6, axis = 0)   # you can perform sums and other aggregate functions axis-wise
```

In [ ]:
```
np.sum(y >= 6, axis = 1)
```

Bitwise (element-wise) Boolean operators

Bitwise (element-wise) Boolean operators

In [ ]:

```
a = np.array([True, True, False, False])
b = np.array([True, False, True, False])
print(a)
print(b)
```

Bitwise (element-wise) Boolean operators

In [ ]:

```
a = np.array([True, True, False, False])
b = np.array([True, False, True, False])
print(a)
print(b)
```

In [ ]:

```
print(a & b) # bitwise and
```

Bitwise (element-wise) Boolean operators

In [ ]:

```
a = np.array([True, True, False, False])
b = np.array([True, False, True, False])
print(a)
print(b)
```

In [ ]:

```
print(a & b) # bitwise and
```

In [ ]:

```
print(a | b) # bitwise or
```

Bitwise (element-wise) Boolean operators

In [ ]:

```
a = np.array([True, True, False, False])
b = np.array([True, False, True, False])
print(a)
print(b)
```

In [ ]:

```
print(a & b) # bitwise and
```

In [ ]:

```
print(a | b) # bitwise or
```

In [ ]:

```
print(a ^ b) # bitwise xor (exclusive or)
```

In [ ]:

```
print(~a)  # bitwise not
```

In [ ]:
```
print(~a)  # bitwise not
```

In [ ]:
```
np.any(a)
```

In [ ]:
```
print(~a)  # bitwise not
```

In [ ]:
```
np.any(a)
```

In [ ]:
```
np.all(a)
```

fancy indexing

Regular lists in python do not support fancy indexing, but NumPy does!

fancy indexing

Regular lists in python do not support fancy indexing, but NumPy does!

In [ ]:

```
np.random.seed(1)
x = np.random.randint(100, size = 10)
print(x)
```

fancy indexing

Regular lists in python do not support fancy indexing, but NumPy does!

In [ ]:

```
np.random.seed(1)
x = np.random.randint(100, size = 10)
print(x)
```

In [ ]:

```
index = [0, 1, 5]
print(x[index])
```

In [ ]:

```
a = [1, 4, 7]
b = [2, 3, 8]
ind = np.vstack([a,b])
print(ind)
```

In [ ]:

```
a = [1, 4, 7]
b = [2, 3, 8]
ind = np.vstack([a,b])
print(ind)
```

In [ ]:

```
print(x[ind])
```

In [ ]:

```
a = [1, 4, 7]
b = [2, 3, 8]
ind = np.vstack([a,b])
print(ind)
```

In [ ]:

```
print(x[ind])
```

In [ ]:

```
X = np.arange(12).reshape((3, 4))
print(X)
```

In [ ]:

```
a = [1, 4, 7]
b = [2, 3, 8]
ind = np.vstack([a,b])
print(ind)
```

In [ ]:

```
print(x[ind])
```

In [ ]:

```
X = np.arange(12).reshape((3, 4))
print(X)
```

In [ ]:

```
row = np.array([0, 1, 2])
col = np.array([2, 1, 3])
X[row, col]
```

- `np.sort()`
- `np.argsort()` gives the indexes of the values to have the proper sorting

- `np.sort()`
- `np.argsort()` gives the indexes of the values to have the proper sorting

In [ ]:

```
np.random.seed(2)
x = np.arange(5)
np.random.shuffle(x)
print(x)
```

- `np.sort()`
- `np.argsort()` gives the indexes of the values to have the proper sorting

In [ ]:

```
np.random.seed(2)
x = np.arange(5)
np.random.shuffle(x)
print(x)
```

In [ ]:

```
x.sort() # sorts x in place
print(x)
```

- `np.sort()`
- `np.argsort()` gives the indexes of the values to have the proper sorting

In [ ]:

```
np.random.seed(2)
x = np.arange(5)
np.random.shuffle(x)
print(x)
```

In [ ]:

```
x.sort() # sorts x in place
print(x)
```

In [ ]:

```
y = np.array([5, 2, 1, 4])
print(y)
print(y.argsort())
```

- `np.sort()`
- `np.argsort()` gives the indexes of the values to have the proper sorting

In [ ]:
```
np.random.seed(2)
x = np.arange(5)
np.random.shuffle(x)
print(x)
```

In [ ]:
```
x.sort() # sorts x in place
print(x)
```

In [ ]:
```
y = np.array([5, 2, 1, 4])
print(y)
print(y.argsort())
```

In [ ]:
```
d = y.argsort()
y[d]
```

Sorting along rows or columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the axis argument. For example:

Sorting along rows or columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the axis argument. For example:

In [ ]:
```
np.random.seed(1)
X = np.random.randint(0, 10, (4, 6))
print(X)
```

Sorting along rows or columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the axis argument. For example:

In [ ]:
```
np.random.seed(1)
X = np.random.randint(0, 10, (4, 6))
print(X)
```

In [ ]:
```
# sort each column of X
# np.sort returns a copy of X after sorted. It does not modify X
np.sort(X, axis=0)
```

Sorting along rows or columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the axis argument. For example:

In [ ]:
```
np.random.seed(1)
X = np.random.randint(0, 10, (4, 6))
print(X)
```

In [ ]:
```
# sort each column of X
# np.sort returns a copy of X after sorted. It does not modify X
np.sort(X, axis=0)
```

In [ ]:
```
# sort each row of X
np.sort(X, axis=1)
```

Sorting along rows or columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the axis argument. For example:

In [ ]:
```
np.random.seed(1)
X = np.random.randint(0, 10, (4, 6))
print(X)
```

In [ ]:
```
# sort each column of X
# np.sort returns a copy of X after sorted. It does not modify X
np.sort(X, axis=0)
```

In [ ]:
```
# sort each row of X
np.sort(X, axis=1)
```

In [ ]:
```
X[0,:] # selecting a row
```

Sorting along rows or columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the axis argument. For example:

In [ ]:
```
np.random.seed(1)
X = np.random.randint(0, 10, (4, 6))
print(X)
```

In [ ]:
```
# sort each column of X
# np.sort returns a copy of X after sorted. It does not modify X
np.sort(X, axis=0)
```

In [ ]:
```
# sort each row of X
np.sort(X, axis=1)
```

In [ ]:
```
X[0,:] # selecting a row
```

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

Sorting along rows or columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the axis argument. For example:

In [ ]:
```
np.random.seed(1)
X = np.random.randint(0, 10, (4, 6))
print(X)
```

In [ ]:
```
# sort each column of X
# np.sort returns a copy of X after sorted. It does not modify X
np.sort(X, axis=0)
```

In [ ]:
```
# sort each row of X
np.sort(X, axis=1)
```

In [ ]:
```
X[0,:] # selecting a row
```

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

In [ ]:
```
X[:,1].argsort()  # the argsort for the column index 1
```

Sorting along rows or columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the axis argument. For example:

In [ ]:
```
np.random.seed(1)
X = np.random.randint(0, 10, (4, 6))
print(X)
```

In [ ]:
```
# sort each column of X
# np.sort returns a copy of X after sorted. It does not modify X
np.sort(X, axis=0)
```

In [ ]:
```
# sort each row of X
np.sort(X, axis=1)
```

In [ ]:
```
X[0,:] # selecting a row
```

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

In [ ]:
```
X[:,1].argsort()  # the argsort for the column index 1
```

In [ ]:
```
print(X[ X[:,1].argsort() , : ])  # 'subset' X by the argsort to arrange X by the column
```