ADVANCED NUMPY

In [193]:

```
import numpy as np
import pandas as pd
import numba as nb
```

In [ ]:

```
# Advanced NumPy

# Let's go deeper into the NumPy library for array computing.
    # This will include more internal detail about the ndarray type
    # and more advanced array manipulations and algorithms.
```

In [ ]:

```
# Ndarray Object Internals

# The NumPy ndarray provides a means to interpret a block of homogeneous data (either cont
iguous or strided)
    # as a multidimensional array object.
    # The data type, or dtype, determines how the data is interpreted as being floating po
int, integer,
    # boolean, or any of the other types.

# Part of what makes ndarray flexible is that every array object is a strided view on a bl
ock of data.
    # You might wonder, for example, how the array view arr[::2, ::-1] does not copy any d
ata.
    # The reason is that the ndarray is more than just a chunk of memory and a dtype;
    # it also has “striding” information that enables the array to move through memory wit
h varying step sizes.
    # More precisely, the ndarray internally consists of the following:

#        • A pointer to data—that is, a block of data in RAM or in a memory-mapped file

#        • The data type or dtype, describing fixed-size value cells in the array

#        • A tuple indicating the array’s shape

#        • A tuple of strides, integers indicating the number of bytes to “step” in order
 to
#            advance one element along a dimension
```

In [3]:

```
# For example, a 10 × 5 array would have shape (10, 5):

np.ones((10, 5)).shape
```

Out[3]:

```
(10, 5)
```

In [4]:

```
# A typical (C order) 3 × 4 × 5 array of float64 (8-byte) values has strides (160, 40,8)
   # (knowing about the strides can be useful because, in general, the larger the strides
 on a particular axis,
     # the more costly it is to perform computation along that axis):

np.ones((3, 4, 5), dtype=np.float64).strides
```

Out[4]:

```
(160, 40, 8)
```

In [ ]:

```
# While it is rare that a typical NumPy user would be interested in the array strides,
    # they are the critical ingredient in constructing “zero-copy” array views. Strides ca
n even be negative,
    # which enables an array to move “backward” through memory
    # (this would be the case, for example, in a slice like obj[::-1] or obj[:, ::-1]).
```

In [6]:

```
# NumPy dtype Hierarchy

# You may occasionally have code that needs to check whether an array contains integers,
    # floating-point numbers, strings, or Python objects.
    # Because there are multiple types of floating-point numbers (float16 through float12
8),
    # checking that the dtype is among a list of types would be very verbose.
    # Fortunately, the dtypes have superclasses such as np.integer and np.floating,
    # which can be used in conjunction with the np.issubdtype function:


ints = np.ones(10, dtype=np.uint16)

ints
```

Out[6]:

```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1], dtype=uint16)
```

In [7]:

```
floats = np.ones(10, dtype=np.float32)

floats
```

Out[7]:

```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.], dtype=float32)
```

In [8]:

```
np.issubdtype(ints.dtype, np.integer)
```

Out[8]:

```
True
```

In [9]:

```
np.issubdtype(floats.dtype, np.floating)
```

Out[9]:

```
True
```

In [10]:

```
# You can see all of the parent classes of a specific dtype by calling the type's mro meth
od:

np.float64.mro()
```

Out[10]:

```
[numpy.float64,
 numpy.floating,
 numpy.inexact,
 numpy.number,
 numpy.generic,
 float,
 object]
```

In [11]:

```
# Therefore, we also have:

np.issubdtype(ints.dtype, np.number)
```

Out[11]:

```
True
```

In [ ]:

```
# Most NumPy users will never have to know about this, but it occasionally comes in handy.
```

In [ ]:

```
# Advanced Array Manipulation

# There are many ways to work with arrays beyond fancy indexing, slicing, and boolean subs
etting.
    # While much of the heavy lifting for data analysis applications is handled by higher-
level functions
    # in pandas, you may at some point need to write a data algorithm that is not found
    # in one of the existing libraries.
```

In [12]:

```
# Reshaping Arrays

# In many cases, you can convert an array from one shape to another without copying any da
ta.
    # To do this, pass a tuple indicating the new shape to the reshape array instance meth
od.
    # For example, suppose we had a one-dimensional array of values that we wished to rear
range into a matrix:


arr = np.arange(8)

arr
```

Out[12]:

```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

In [17]:

```
arr.reshape((4,2))
```

Out[17]:

```
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

In [18]:

```
# A multidimensional array can also be reshaped:


arr.reshape((4, 2)).reshape((2, 4))
```

Out[18]:

```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

In [19]:

```
# One of the passed shape dimensions can be -1, in which case the value used for tha dimen
sion
    # will be inferred from the data:


arr = np.arange(15)

arr
```

Out[19]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

In [20]:

```
arr.reshape((5, -1))
```

Out[20]:

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

In [22]:

```
# Since an array’s shape attribute is a tuple, it can be passed to reshape, too:


other_arr = np.ones((3, 5))

other_arr
```

Out[22]:

```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

In [23]:

```
other_arr.shape
```

Out[23]:

```
(3, 5)
```

In [24]:

```
arr.reshape(other_arr.shape)
```

Out[24]:

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

In [25]:

```
# The opposite operation of reshape from one-dimensional to a higher dimension
    # is typically known as flattening or raveling:


arr = np.arange(15).reshape((5, 3))

arr
```

Out[25]:

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

In [26]:

```
arr.ravel()
```

Out[26]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

In [27]:

```
# ravel does not produce a copy of the underlying values if the values in the result
    # were contiguous in the original array.
    # The flatten method behaves like ravel except it always returns a copy of the data:

arr.flatten()
```

Out[27]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

In [ ]:

```
# The data can be reshaped or raveled in different orders. This is a slightly nuanced topi
c for new NumPy users
    # and is therefore the next subtopic.
```

In [28]:

```
# C Versus Fortran Order

# NumPy gives you control and flexibility over the layout of your data in memory.
    # By default, NumPy arrays are created in row major order.
    # Spatially this means that if you have a two-dimensional array of data,
    # the items in each row of the array are stored in adjacent memory locations.
    # The alternative to row major ordering is column major order,
    # which means that values within each column of data are stored in adjacent memory loc
ations.

# For historical reasons, row and column major order are also know as C and Fortran order,
respectively.
    # In the FORTRAN 77 language, matrices are all column major.

# Functions like reshape and ravel accept an order argument indicating the order to use th
e data in the array.
    # This is usually set to 'C' or 'F' in most cases
    # (there are also less commonly used options 'A' and 'K'; see the NumPy documentatio
n):


arr = np.arange(12).reshape((3, 4))

arr
```

Out[28]:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [29]:

```
arr.ravel()
```

Out[29]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

In [30]:

```
arr.ravel('C')
```

Out[30]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

In [31]:

```
arr.ravel('F')
```

Out[31]:

```
array([ 0,  4,  8,  1,  5,  9,  2,  6, 10,  3,  7, 11])
```

In [32]:

```
arr.ravel('A')
```

Out[32]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

In [33]:

```
arr.ravel('K')
```

Out[33]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

In [ ]:

```
# Reshaping arrays with more than two dimensions can be a bit mind-bending.
    # The key difference between C and Fortran order is the way in which the dimensions ar
e walked:

#        C/row major order
#            Traverse higher dimensions first (e.g., axis 1 before advancing on axis 0).

#        Fortran/column major order
#            Traverse higher dimensions last (e.g., axis 0 before advancing on axis 1).
```

In [34]:

```
# Concatenating and Splitting Arrays

# numpy.concatenate takes a sequence (tuple, list, etc.) of arrays and joins them together
    # in order along the input axis:


arr1 = np.array([[1, 2, 3], [4, 5, 6]])

arr1
```

Out[34]:

```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [36]:

```
arr2 = np.array([[7, 8, 9], [10, 11, 12]])

arr2
```

Out[36]:

```
array([[ 7,  8,  9],
       [10, 11, 12]])
```

In [37]:

```
np.concatenate([arr1, arr2], axis=0)
```

Out[37]:

```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

In [40]:

```
np.concatenate([arr1, arr2], axis=1)
```

Out[40]:

```
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

In [41]:

```
# There are some convenience functions, like vstack and hstack, for common kinds of concat
enation.
    # The preceding operations could have been expressed as:

np.vstack((arr1, arr2))
```

Out[41]:

```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

In [42]:

```
np.hstack((arr1,arr2))
```

Out[42]:

```
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

In [43]:

```
# split, on the other hand, slices apart an array into multiple arrays along an axis:


arr = np.random.randn(5, 2)

arr
```

Out[43]:

```
array([[ 1.33222118, -1.81806039],
       [-0.15346484, -0.13432537],
       [-1.20499111,  1.7143637 ],
       [ 0.29597033, -0.46980145],
       [ 0.28043885,  0.71384464]])
```

In [44]:

```
first, second, third = np.split(arr, [1, 3])

first
```

Out[44]:

```
array([[ 1.33222118, -1.81806039]])
```

In [45]:

```
second
```

Out[45]:

```
array([[-0.15346484, -0.13432537],
       [-1.20499111,  1.7143637 ]])
```

In [46]:

```
third
```

Out[46]:

```
array([[ 0.29597033, -0.46980145],
       [ 0.28043885,  0.71384464]])
```

In [ ]:

```
# The value [1, 3] passed to np.split indicate the indices at which to split the array int
o pieces.



# Array concatenation functions


#            Function                    Description

#            concatenate                 Most general function, concatenates collection of
 arrays along one axis
#            vstack, row_stack           Stack arrays row-wise (along axis 0)
#            hstack                      Stack arrays column-wise (along axis 1)
#            column_stack                Like hstack, but converts 1D arrays to 2D column v
ectors first
#            dstack                      Stack arrays “depth”-wise (along axis 2)
#            split                       Split array at passed locations along a particular
axis
#            hsplit/vsplit               Convenience functions for splitting on axis 0 and
 1, respectively
```

In [47]:

```
# Stacking helpers: r_ and c_

# There are two special objects in the NumPy namespace, r_ and c_, that make stacking arra
ys more concise:


arr = np.arange(6)

arr
```

Out[47]:

```
array([0, 1, 2, 3, 4, 5])
```

In [49]:

```
arr1 = arr.reshape((3, 2))

arr1
```

Out[49]:

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

In [50]:

```
arr2 = np.random.randn(3, 2)

arr2
```

Out[50]:

```
array([[ 0.29376076,  0.68752243],
       [-2.53306114,  1.66854045],
       [ 0.01309395,  1.70704312]])
```

In [51]:

```
np.r_[arr1, arr2]
```

Out[51]:

```
array([[ 0.        ,  1.        ],
       [ 2.        ,  3.        ],
       [ 4.        ,  5.        ],
       [ 0.29376076,  0.68752243],
       [-2.53306114,  1.66854045],
       [ 0.01309395,  1.70704312]])
```

In [54]:

```
np.c_[arr1, arr2]
```

Out[54]:

```
array([[ 0.        ,  1.        ,  0.29376076,  0.68752243],
       [ 2.        ,  3.        , -2.53306114,  1.66854045],
       [ 4.        ,  5.        ,  0.01309395,  1.70704312]])
```

In [60]:

```
np.c_[np.r_[arr1, arr2], arr]
```

Out[60]:

```
array([[ 0.        ,  1.        ,  0.        ],
       [ 2.        ,  3.        ,  1.        ],
       [ 4.        ,  5.        ,  2.        ],
       [ 0.29376076,  0.68752243,  3.        ],
       [-2.53306114,  1.66854045,  4.        ],
       [ 0.01309395,  1.70704312,  5.        ]])
```

In [61]:

```
# These additionally can translate slices to arrays:

np.c_[1:6, -10:-5]
```

Out[61]:

```
array([[  1, -10],
       [  2,  -9],
       [  3,  -8],
       [  4,  -7],
       [  5,  -6]])
```

In [62]:

```
np.r_[1:6, -10:-5]
```

Out[62]:

```
array([  1,   2,   3,   4,   5, -10,  -9,  -8,  -7,  -6])
```

In [ ]:

```
# See the docstring for more on what you can do with c_ and r_.
```

In [63]:

```
# Repeating Elements: tile and repeat

# Two useful tools for repeating or replicating arrays to produce larger arrays
    # are the repeat and tile functions.
    # repeat replicates each element in an array some number of times, producing a larger
 array:


arr = np.arange(3)

arr
```

Out[63]:

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

In [64]:

```
arr.repeat(3)
```

Out[64]:

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

In [65]:

```
arr.repeat(4)
```

Out[65]:

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

In [66]:

```
arr.repeat(5)
```

Out[66]:

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

In [69]:

```
# The need to replicate or repeat arrays can be less common with NumPy than it is with oth
er array programming
    # frameworks like MATLAB. One reason for this is that broadcasting often fills this ne
ed better.

# By default, if you pass an integer, each element will be repeated that number of times.
    # If you pass an array of integers, each element can be repeated a different number of
times:


arr.repeat([2, 3, 4])
```

Out[69]:

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

In [70]:

```
arr.repeat([4, 7, 9])
```

Out[70]:

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

In [71]:

```
# Multidimensional arrays can have their elements repeated along a particular axis.


arr = np.random.randn(2, 2)

arr
```

Out[71]:

```
array([[-0.64286408, -0.88167946],
       [-0.03991212, -1.66391726]])
```

In [72]:

```
arr.repeat(2, axis=0)
```

Out[72]:

```
array([[-0.64286408, -0.88167946],
       [-0.64286408, -0.88167946],
       [-0.03991212, -1.66391726],
       [-0.03991212, -1.66391726]])
```

In [73]:

```
arr.repeat(2, axis=1)
```

Out[73]:

```
array([[-0.64286408, -0.64286408, -0.88167946, -0.88167946],
       [-0.03991212, -0.03991212, -1.66391726, -1.66391726]])
```

In [74]:

```
arr.repeat(7, axis=0)
```

Out[74]:

```
array([[-0.64286408, -0.88167946],
       [-0.64286408, -0.88167946],
       [-0.64286408, -0.88167946],
       [-0.64286408, -0.88167946],
       [-0.64286408, -0.88167946],
       [-0.64286408, -0.88167946],
       [-0.64286408, -0.88167946],
       [-0.03991212, -1.66391726],
       [-0.03991212, -1.66391726],
       [-0.03991212, -1.66391726],
       [-0.03991212, -1.66391726],
       [-0.03991212, -1.66391726],
       [-0.03991212, -1.66391726],
       [-0.03991212, -1.66391726]])
```

In [75]:

```
arr.repeat(7, axis=1)
```

Out[75]:

```
array([[-0.64286408, -0.64286408, -0.64286408, -0.64286408, -0.64286408,
        -0.64286408, -0.64286408, -0.88167946, -0.88167946, -0.88167946,
        -0.88167946, -0.88167946, -0.88167946, -0.88167946],
       [-0.03991212, -0.03991212, -0.03991212, -0.03991212, -0.03991212,
        -0.03991212, -0.03991212, -1.66391726, -1.66391726, -1.66391726,
        -1.66391726, -1.66391726, -1.66391726, -1.66391726]])
```

In [76]:

```
# Note that if no axis is passed, the array will be flattened first, which is likely not w
hat you want.
    # Similarly, you can pass an array of integers when repeating a multidimensional array
    # to repeat a given slice a different number of times:


arr.repeat([2, 3], axis=0)
```

Out[76]:

```
array([[-0.64286408, -0.88167946],
       [-0.64286408, -0.88167946],
       [-0.03991212, -1.66391726],
       [-0.03991212, -1.66391726],
       [-0.03991212, -1.66391726]])
```

In [77]:

```
arr.repeat([2, 3], axis=1)
```

Out[77]:

```
array([[-0.64286408, -0.64286408, -0.88167946, -0.88167946, -0.88167946],
       [-0.03991212, -0.03991212, -1.66391726, -1.66391726, -1.66391726]])
```

In [78]:

```
# tile, on the other hand, is a shortcut for stacking copies of an array along an axis.
    # Visually you can think of it as being akin to “laying down tiles”:


arr
```

Out[78]:

```
array([[-0.64286408, -0.88167946],
       [-0.03991212, -1.66391726]])
```

In [79]:

```
np.tile(arr, 2)
```

Out[79]:

```
array([[-0.64286408, -0.88167946, -0.64286408, -0.88167946],
       [-0.03991212, -1.66391726, -0.03991212, -1.66391726]])
```

In [80]:

```
np.tile(arr, 7)
```

Out[80]:

```
array([[-0.64286408, -0.88167946, -0.64286408, -0.88167946, -0.64286408,
        -0.88167946, -0.64286408, -0.88167946, -0.64286408, -0.88167946,
        -0.64286408, -0.88167946, -0.64286408, -0.88167946],
       [-0.03991212, -1.66391726, -0.03991212, -1.66391726, -0.03991212,
        -1.66391726, -0.03991212, -1.66391726, -0.03991212, -1.66391726,
        -0.03991212, -1.66391726, -0.03991212, -1.66391726]])
```

In [81]:

```
# The second argument is the number of tiles; with a scalar, the tiling is made row by ro
w,
    # rather than column by column.
    # The second argument to tile can be a tuple indicating the layout of the “tiling”:

arr
```

Out[81]:

```
array([[-0.64286408, -0.88167946],
       [-0.03991212, -1.66391726]])
```

In [82]:

```
np.tile(arr, (2, 1))
```

Out[82]:

```
array([[-0.64286408, -0.88167946],
       [-0.03991212, -1.66391726],
       [-0.64286408, -0.88167946],
       [-0.03991212, -1.66391726]])
```

In [85]:

```
np.tile(arr, (1, 2))
```

Out[85]:

```
array([[-0.64286408, -0.88167946, -0.64286408, -0.88167946],
       [-0.03991212, -1.66391726, -0.03991212, -1.66391726]])
```

In [86]:

```
np.tile(arr, (3, 2))
```

Out[86]:

```
array([[-0.64286408, -0.88167946, -0.64286408, -0.88167946],
       [-0.03991212, -1.66391726, -0.03991212, -1.66391726],
       [-0.64286408, -0.88167946, -0.64286408, -0.88167946],
       [-0.03991212, -1.66391726, -0.03991212, -1.66391726],
       [-0.64286408, -0.88167946, -0.64286408, -0.88167946],
       [-0.03991212, -1.66391726, -0.03991212, -1.66391726]])
```

In [87]:

```
np.tile(arr, (2, 3))
```

Out[87]:

```
array([[-0.64286408, -0.88167946, -0.64286408, -0.88167946, -0.64286408,
        -0.88167946],
       [-0.03991212, -1.66391726, -0.03991212, -1.66391726, -0.03991212,
        -1.66391726],
       [-0.64286408, -0.88167946, -0.64286408, -0.88167946, -0.64286408,
        -0.88167946],
       [-0.03991212, -1.66391726, -0.03991212, -1.66391726, -0.03991212,
        -1.66391726]])
```

In [83]:

```
np.tile(arr, (2, 7))
```

Out[83]:

```
array([[-0.64286408, -0.88167946, -0.64286408, -0.88167946, -0.64286408,
        -0.88167946, -0.64286408, -0.88167946, -0.64286408, -0.88167946,
        -0.64286408, -0.88167946, -0.64286408, -0.88167946],
       [-0.03991212, -1.66391726, -0.03991212, -1.66391726, -0.03991212,
        -1.66391726, -0.03991212, -1.66391726, -0.03991212, -1.66391726,
        -0.03991212, -1.66391726, -0.03991212, -1.66391726],
       [-0.64286408, -0.88167946, -0.64286408, -0.88167946, -0.64286408,
        -0.88167946, -0.64286408, -0.88167946, -0.64286408, -0.88167946,
        -0.64286408, -0.88167946, -0.64286408, -0.88167946],
       [-0.03991212, -1.66391726, -0.03991212, -1.66391726, -0.03991212,
        -1.66391726, -0.03991212, -1.66391726, -0.03991212, -1.66391726,
        -0.03991212, -1.66391726, -0.03991212, -1.66391726]])
```

In [84]:

```
np.tile(arr, (7, 2))
```

Out[84]:

```
array([[-0.64286408, -0.88167946, -0.64286408, -0.88167946],
       [-0.03991212, -1.66391726, -0.03991212, -1.66391726],
       [-0.64286408, -0.88167946, -0.64286408, -0.88167946],
       [-0.03991212, -1.66391726, -0.03991212, -1.66391726],
       [-0.64286408, -0.88167946, -0.64286408, -0.88167946],
       [-0.03991212, -1.66391726, -0.03991212, -1.66391726],
       [-0.64286408, -0.88167946, -0.64286408, -0.88167946],
       [-0.03991212, -1.66391726, -0.03991212, -1.66391726],
       [-0.64286408, -0.88167946, -0.64286408, -0.88167946],
       [-0.03991212, -1.66391726, -0.03991212, -1.66391726],
       [-0.64286408, -0.88167946, -0.64286408, -0.88167946],
       [-0.03991212, -1.66391726, -0.03991212, -1.66391726],
       [-0.64286408, -0.88167946, -0.64286408, -0.88167946],
       [-0.03991212, -1.66391726, -0.03991212, -1.66391726]])
```

In [88]:

```
# Fancy Indexing Equivalents: take and put

# One way to get and set subsets of arrays is by fancy indexing using integer arrays:


arr = np.arange(10) * 100

arr
```

Out[88]:

```
array([  0, 100, 200, 300, 400, 500, 600, 700, 800, 900])
```

In [89]:

```
inds = [7, 1, 2, 6]

inds
```

Out[89]:

```
[7, 1, 2, 6]
```

In [90]:

```
arr[inds]
```

Out[90]:

```
array([700, 100, 200, 600])
```

In [91]:

```
# There are alternative ndarray methods that are useful in the special case of only making
    # a selection on a single axis:


arr.take(inds)
```

Out[91]:

```
array([700, 100, 200, 600])
```

In [92]:

```
arr.put(inds, 42)

arr
```

Out[92]:

```
array([  0,  42,  42, 300, 400, 500,  42,  42, 800, 900])
```

In [93]:

```
arr.put(inds, [40, 41, 42, 43])

arr
```

Out[93]:

```
array([  0,  41,  42, 300, 400, 500,  43,  40, 800, 900])
```

In [95]:

```
# To use take along other axes, you can pass the axis keyword:


inds = [2, 0, 2, 1]

inds
```

Out[95]:

```
[2, 0, 2, 1]
```

In [96]:

```
arr = np.random.randn(2, 4)

arr
```

Out[96]:

```
array([[-0.6864408 ,  0.07286014,  0.58338652, -2.01298925],
       [ 0.34688456,  0.14186142,  0.18422074,  0.77167161]])
```

In [97]:

```
arr.take(inds, axis=1)
```

Out[97]:

```
array([[ 0.58338652, -0.6864408 ,  0.58338652,  0.07286014],
       [ 0.18422074,  0.34688456,  0.18422074,  0.14186142]])
```

In [99]:

```
# put does not accept an axis argument but rather indexes into the flattened (onedimension
al, C order) version
    # of the array.
    # Thus, when you need to set elements using an index array on other axes,
    # it is often easiest to use fancy indexing.
```

In [6]:

```
# Broadcasting

# Broadcasting describes how arithmetic works between arrays of different shapes.
    # It can be a powerful feature, but one that can cause confusion, even for experienced
users.
    # The simplest example of broadcasting occurs when combining a scalar value with an ar
ray:


arr = np.arange(5)

arr
```

Out[6]:

```
array([0, 1, 2, 3, 4])
```

In [7]:

```
arr * 4
```

Out[7]:

```
array([ 0,  4,  8, 12, 16])
```

In [8]:

```
# Above we say that the scalar value 4 has been broadcast to all of the other elements in
    # the multiplication operation.

# For example, we can demean each column of an array by subtracting the column means.
    # In this case, it is very simple:


arr = np.random.randn(4, 3)

arr
```

Out[8]:

```
array([[-0.4290365 , -1.23478276,  0.04131859],
       [ 0.41796202, -1.44762384,  0.42056144],
       [ 0.00224038,  0.47297833, -0.26550207],
       [-0.49401702, -1.02432231,  0.52866203]])
```

In [9]:

```
arr.mean()
```

Out[9]:

```
-0.2509634766353159
```

In [10]:

```
arr.mean(0)
```

Out[10]:

```
array([-0.12571278, -0.80843765,  0.18126   ])
```

In [11]:

```
arr.mean(1)
```

Out[11]:

```
array([-0.54083356, -0.20303346,  0.06990555, -0.32989243])
```

In [13]:

```
demeaned = arr - arr.mean(0)

demeaned
```

Out[13]:

```
array([[-0.30332372, -0.42634511, -0.13994141],
       [ 0.5436748 , -0.6391862 ,  0.23930144],
       [ 0.12795316,  1.28141597, -0.44676207],
       [-0.36830424, -0.21588466,  0.34740203]])
```

In [15]:

```
demeaned.mean()
```

Out[15]:

```
-1.3877787807814457e-17
```

In [16]:

```
demeaned.mean(0)
```

Out[16]:

```
array([ 1.38777878e-17, -2.77555756e-17,  0.00000000e+00])
```

In [17]:

```
demeaned.mean(1)
```

Out[17]:

```
array([-0.28987008,  0.04793002,  0.32086902, -0.07892896])
```

In [ ]:

```
# Demeaning the rows as a broadcast operation requires a bit more care.
    # Fortunately, broadcasting potentially lower dimensional values across any dimension
 of an array
    # (like subtracting the row means from each column of a two-dimensional array)
    # is possible as long as you follow the rules.


# This brings us to:

#                                 The Broadcasting Rule
# Two arrays are compatible for broadcasting if for each trailing dimension (i.e., startin
g from the end)
# the axis lengths match or if either of the lengths is 1. Broadcasting is then performed
 over the missing
# or length 1 dimensions.
```

In [18]:

```
# Even as an experienced NumPy user, I often find myself having to pause and draw a diagra
m as I think about
    # the broadcasting rule.
    # Consider the last example and suppose we wished instead to subtract the mean value f
rom each row.
    # Since arr.mean(0) has length 3, it is compatible for broadcasting across axis 0 beca
use the trailing
    # dimension in arr is 3 and therefore matches.
    # According to the rules, to subtract over axis 1 (i.e., subtract the row mean from ea
ch row),
    # the smaller array must have shape (4, 1):

arr
```

Out[18]:

```
array([[-0.4290365 , -1.23478276,  0.04131859],
       [ 0.41796202, -1.44762384,  0.42056144],
       [ 0.00224038,  0.47297833, -0.26550207],
       [-0.49401702, -1.02432231,  0.52866203]])
```

In [21]:

```
row_means = arr.mean(1)

row_means
```

Out[21]:

```
array([-0.54083356, -0.20303346,  0.06990555, -0.32989243])
```

In [22]:

```
row_means.shape
```

Out[22]:

```
(4,)
```

In [23]:

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

Out[23]:

```
array([[-0.54083356],
       [-0.20303346],
       [ 0.06990555],
       [-0.32989243]])
```

In [25]:

```
demeaned = arr - row_means.reshape((4,1))

demeaned
```

Out[25]:

```
array([[ 0.11179706, -0.6939492 ,  0.58215215],
       [ 0.62099549, -1.24459038,  0.6235949 ],
       [-0.06766516,  0.40307278, -0.33540762],
       [-0.16412459, -0.69442987,  0.85855446]])
```

In [26]:

```
demeaned.mean()
```

Out[26]:

```
1.850371707708594e-17
```

In [27]:

```
demeaned.mean(0)
```

Out[27]:

```
array([ 0.1252507 , -0.55747417,  0.43222347])
```

In [28]:

```
demeaned.mean(1)
```

Out[28]:

```
array([-3.70074342e-17,  3.70074342e-17,  0.00000000e+00,  0.00000000e+00])
```

In [29]:

```
# Broadcasting Over Other Axes

# Broadcasting with higher dimensional arrays can seem even more mind-bending,
    # but it is really a matter of following the rules. If you don’t, you’ll get an error
 like this:

arr - arr.mean(1)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-29-9e4a803024ca> in <module>
      4     # butvit is really a matter of following the rules. If you don’t,
you’ll get an error like this:
      5
----> 6 arr - arr.mean(1)

ValueError: operands could not be broadcast together with shapes (4,3) (4,)
```

In [30]:

```
# It’s quite common to want to perform an arithmetic operation with a lower dimensional ar
ray
    # across axes other than axis 0.
    # According to the broadcasting rule, the “broadcast dimensions” must be 1 in the smal
ler array.
    # In the example of row demeaning shown here, this meant reshaping the row means to be
shape (4, 1)
    # instead of (4,):

arr - arr.mean(1).reshape((4, 1))
```

Out[30]:

```
array([[ 0.11179706, -0.6939492 ,  0.58215215],
       [ 0.62099549, -1.24459038,  0.6235949 ],
       [-0.06766516,  0.40307278, -0.33540762],
       [-0.16412459, -0.69442987,  0.85855446]])
```

In [32]:

```
# In the three-dimensional case, broadcasting over any of the three dimensions is only a m
atter of reshaping
    # the data to be shape-compatible.

# A common problem, therefore, is needing to add a new axis with length 1 specifically
    # for broadcasting purposes.
    # Using reshape is one option, but inserting an axis requires constructing a tuple
    # indicating the new shape.This can often be a tedious exercise.
    # Thus, NumPy arrays offer a special syntax for inserting new axes by indexing.
    # We use the special np.newaxis attribute along with “full” slices to insert the new a
xis:


arr = np.zeros((4, 4))

arr
```

Out[32]:

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

In [34]:

```
arr_3d = arr[:, np.newaxis, :]

arr_3d
```

Out[34]:

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

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

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

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

In [35]:

```
arr_3d.shape
```

Out[35]:

```
(4, 1, 4)
```

In [37]:

```
arr_1d = np.random.normal(size=3)

arr_1d
```

Out[37]:

```
array([-0.40719218, -1.32142006,  0.12264432])
```

In [38]:

```
arr_1d[:, np.newaxis]
```

Out[38]:

```
array([[-0.40719218],
       [-1.32142006],
       [ 0.12264432]])
```

In [39]:

```
arr_1d[np.newaxis, :]
```

Out[39]:

```
array([[-0.40719218, -1.32142006,  0.12264432]])
```

In [40]:

```
# Thus, if we had a three-dimensional array and wanted to demean axis 2, say, we would nee
d to write:

arr = np.random.randn(3, 4, 5)

arr
```

Out[40]:

```
array([[[ 0.22067351,  3.01315715,  1.3245668 ,  0.73755868,
          0.20639973],
        [-0.6198023 ,  0.66000681, -0.82840268, -1.11622781,
          0.86700759],
        [-0.47165603, -0.71247087, -1.07217153, -0.11514507,
          0.52126238],
        [-1.22809747,  0.45778851,  0.69904621,  1.6742796 ,
          1.07299997]],

       [[ 0.07624852,  0.46179254, -0.82756269, -1.23275627,
          2.32956618],
        [-0.82614275, -0.77474342, -1.77655293, -0.91926351,
          0.50336781],
        [-1.08196146,  0.29968226,  0.96002316, -0.25009993,
         -0.7572684 ],
        [ 0.74146467, -1.00601945,  0.34643083,  1.12784706,
          0.44480431]],

       [[-0.59561434,  0.51490082,  1.87081664, -0.84672811,
          1.33119253],
        [ 0.55605734, -1.23795331, -1.47202384,  0.71790392,
          0.29493959],
        [ 0.55308327, -0.71064273,  0.31722041, -1.27594141,
         -0.42941873],
        [ 0.65510907,  1.57775787,  1.11032321,  0.72963188,
         -0.73807179]]])
```

In [41]:

```
depth_means = arr.mean(2)

depth_means
```

Out[41]:

```
array([[ 1.10047117, -0.20748368, -0.37003622,  0.53520336],
       [ 0.16145766, -0.75866696, -0.16592487,  0.33090548],
       [ 0.45491351, -0.22821526, -0.30913984,  0.66695005]])
```

In [42]:

```
depth_means.shape
```

Out[42]:

```
(3, 4)
```

In [44]:

```
demeaned = arr - depth_means[:, :, np.newaxis]

demeaned
```

Out[44]:

```
array([[[-0.87979767,  1.91268598,  0.22409563, -0.3629125 ,
         -0.89407144],
        [-0.41231862,  0.86749049, -0.620919  , -0.90874413,
          1.07449127],
        [-0.1016198 , -0.34243465, -0.7021353 ,  0.25489115,
          0.8912986 ],
        [-1.76330084, -0.07741486,  0.16384285,  1.13907624,
          0.5377966 ]],

       [[-0.08520913,  0.30033488, -0.98902035, -1.39421393,
          2.16810853],
        [-0.06747579, -0.01607646, -1.01788597, -0.16059655,
          1.26203477],
        [-0.91603658,  0.46560713,  1.12594804, -0.08417506,
         -0.59134353],
        [ 0.41055918, -1.33692494,  0.01552535,  0.79694157,
          0.11389883]],

       [[-1.05052785,  0.05998731,  1.41590313, -1.30164161,
          0.87627902],
        [ 0.7842726 , -1.00973805, -1.24380858,  0.94611918,
          0.52315485],
        [ 0.86222311, -0.40150289,  0.62636025, -0.96680157,
         -0.12027889],
        [-0.01184098,  0.91080782,  0.44337316,  0.06268183,
         -1.40502184]]])
```

In [45]:

```
demeaned.mean(2)
```

Out[45]:

```
array([[-4.44089210e-17,  0.00000000e+00, -4.44089210e-17,
        -4.44089210e-17],
       [ 8.88178420e-17,  0.00000000e+00,  4.44089210e-17,
         1.11022302e-17],
       [ 0.00000000e+00,  0.00000000e+00,  6.66133815e-17,
         0.00000000e+00]])
```

In [ ]:

```
# You might be wondering if there’s a way to generalize demeaning over an axis without sac
rificing performance.
    # There is, but it requires some indexing gymnastics:


def demean_axis(arr, axis=0):
    means = arr.mean(axis)

    # This generalizes things like [:, :, np.newaxis] to N dimensions
    indexer = [slice(None)] * arr.ndim
    indexer[axis] = np.newaxis
    return arr - means[indexer]
```

In [47]:

```
# Setting Array Values by Broadcasting

# The same broadcasting rule governing arithmetic operations also applies to setting value
s via array indexing.
    # In a simple case, we can do things like:


arr = np.zeros((4, 3))

arr
```

Out[47]:

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

In [49]:

```
arr[:] = 5

arr
```

Out[49]:

```
array([[5., 5., 5.],
       [5., 5., 5.],
       [5., 5., 5.],
       [5., 5., 5.]])
```

In [50]:

```
# However, if we had a one-dimensional array of values we wanted to set into the columns o
f the array,
    # we can do that as long as the shape is compatible:


col = np.array([1.28, -0.42, 0.44, 1.6])

col
```

Out[50]:

```
array([ 1.28, -0.42,  0.44,  1.6 ])
```

In [51]:

```
arr[:] = col[:, np.newaxis]

arr
```

Out[51]:

```
array([[ 1.28,  1.28,  1.28],
       [-0.42, -0.42, -0.42],
       [ 0.44,  0.44,  0.44],
       [ 1.6 ,  1.6 ,  1.6 ]])
```

In [52]:

```
arr[:2] = [[-1.37], [0.509]]

arr
```

Out[52]:

```
array([[-1.37 , -1.37 , -1.37 ],
       [ 0.509,  0.509,  0.509],
       [ 0.44 ,  0.44 ,  0.44 ],
       [ 1.6  ,  1.6  ,  1.6  ]])
```

In [ ]:

```
#Advanced ufunc Usage

# While many NumPy users will only make use of the fast element-wise operations provided
    # by the universal functions, there are a number of additional features that occasiona
lly can help you
    # write more concise code without loops.


# ufunc Instance Methods

# Each of NumPy’s binary ufuncs has special methods for performing certain kinds
    # of special vectorized operations:


#            Method                 Description

#            reduce(x)              Aggregate values by successive applications of the oper
ation

#            accumulate(x)          Aggregate values, preserving all partial aggregates

#            reduceat(x, bins)      “Local” reduce or “group by”; reduce contiguous slices
 of data
#                                     to produce aggregated array

#            outer(x, y)            Apply operation to all pairs of elements in x and y;
#                                     the resulting array has shape x.shape + y.shape

```

In [53]:

```
# reduce takes a single array and aggregates its values, optionally along an axis,
    # by performing a sequence of binary operations.
    # For example, an alternative way to sum elements in an array is to use np.add.reduce:


arr = np.arange(10)

arr
```

Out[53]:

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [54]:

```
np.add.reduce(arr)
```

Out[54]:

```
45
```

In [55]:

```
arr.sum()
```

Out[55]:

```
45
```

In [57]:

```
# The starting value (0 for add) depends on the ufunc.
    # If an axis is passed, the reduction is performed along that axis.
    # This allows you to answer certain kinds of questions in a concise way.
    # As a less trivial example, we can use np.logical_and to check whether the values
    # in each row of an array are sorted:


np.random.seed(12346) # for reproducibility

arr = np.random.randn(5, 5)

arr
```

Out[57]:

```
array([[-8.99822478e-02,  7.59372617e-01,  7.48336101e-01,
        -9.81497953e-01,  3.65775545e-01],
       [-3.15442628e-01, -8.66135605e-01,  2.78568155e-02,
        -4.55597723e-01, -1.60189223e+00],
       [ 2.48256116e-01, -3.21536673e-01, -8.48730755e-01,
         4.60468309e-04, -5.46459347e-01],
       [ 2.53915229e-01,  1.93684246e+00, -7.99504902e-01,
        -5.69159281e-01,  4.89244731e-02],
       [-6.49092950e-01, -4.79535727e-01, -9.53521432e-01,
         1.42253882e+00,  1.75403128e-01]])
```

In [64]:

```
arr[::2].sort(1) # sort a few rows

arr[:, :-1] < arr[:, 1:]
```

Out[64]:

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

In [65]:

```
np.logical_and.reduce(arr[:, :-1] < arr[:, 1:], axis=1)
```

Out[65]:

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

In [66]:

```
# Note that logical_and.reduce is equivalent to the all method.

# accumulate is related to reduce like cumsum is related to sum.
    # It produces an array of the same size with the intermediate “accumulated” values:


arr = np.arange(15).reshape((3, 5))

arr
```

Out[66]:

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

In [67]:

```
np.add.accumulate(arr, axis=1)
```

Out[67]:

```
array([[ 0,  1,  3,  6, 10],
       [ 5, 11, 18, 26, 35],
       [10, 21, 33, 46, 60]], dtype=int32)
```

In [68]:

```
np.add.accumulate(arr, axis=0)
```

Out[68]:

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  7,  9, 11, 13],
       [15, 18, 21, 24, 27]], dtype=int32)
```

In [69]:

```
np.add.accumulate(arr)
```

Out[69]:

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  7,  9, 11, 13],
       [15, 18, 21, 24, 27]], dtype=int32)
```

In [70]:

```
# outer performs a pairwise cross-product between two arrays:


arr = np.arange(3).repeat([1, 2, 2])

arr
```

Out[70]:

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

In [71]:

```
np.multiply.outer(arr, np.arange(5))
```

Out[71]:

```
array([[0, 0, 0, 0, 0],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 2, 4, 6, 8],
       [0, 2, 4, 6, 8]])
```

In [74]:

```
#The output of outer will have a dimension that is the sum of the dimensions of the input
s:


x, y = np.random.randn(3, 4), np.random.randn(5)

x
```

Out[74]:

```
array([[ 0.73289771,  0.50470465, -0.7892592 ,  0.5391877 ],
       [ 1.29070685,  0.86761856,  0.41133011,  0.44593599],
       [-0.3171888 , -1.04929141,  1.34589315,  0.35600969]])
```

In [75]:

```
y
```

Out[75]:

```
array([-0.09152874, -0.53496417, -0.03601325, -0.25911386, -0.19944861])
```

In [76]:

```
x, y
```

Out[76]:

```
(array([[ 0.73289771,  0.50470465, -0.7892592 ,  0.5391877 ],
        [ 1.29070685,  0.86761856,  0.41133011,  0.44593599],
        [-0.3171888 , -1.04929141,  1.34589315,  0.35600969]]),
 array([-0.09152874, -0.53496417, -0.03601325, -0.25911386, -0.19944861]))
```

In [78]:

```
result = np.subtract.outer(x, y)

result
```

Out[78]:

```
array([[[ 0.82442645,  1.26786188,  0.76891096,  0.99201157,
          0.93234632],
        [ 0.59623339,  1.03966882,  0.5407179 ,  0.76381851,
          0.70415326],
        [-0.69773046, -0.25429503, -0.75324595, -0.53014534,
         -0.58981059],
        [ 0.63071644,  1.07415187,  0.57520095,  0.79830156,
          0.7386363 ]],

       [[ 1.38223559,  1.82567102,  1.3267201 ,  1.5498207 ,
          1.49015545],
        [ 0.95914729,  1.40258272,  0.9036318 ,  1.12673241,
          1.06706716],
        [ 0.50285885,  0.94629427,  0.44734335,  0.67044396,
          0.61077871],
        [ 0.53746473,  0.98090016,  0.48194924,  0.70504985,
          0.6453846 ]],

       [[-0.22566006,  0.21777537, -0.28117555, -0.05807494,
         -0.11774019],
        [-0.95776267, -0.51432725, -1.01327817, -0.79017756,
         -0.84984281],
        [ 1.43742189,  1.88085732,  1.3819064 ,  1.60500701,
          1.54534175],
        [ 0.44753843,  0.89097386,  0.39202294,  0.61512355,
          0.5554583 ]]])
```

In [79]:

```
result.shape
```

Out[79]:

```
(3, 4, 5)
```

In [80]:

```
# The last method, reduceat, performs a “local reduce,” in essence an array groupby operat
ion
    # in which slices of the array are aggregated together.
    # It accepts a sequence of “bin edges” that indicate how to split and aggregate the va
lues:


arr = np.arange(10)

arr
```

Out[80]:

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [81]:

```
np.add.reduceat(arr, [0, 5, 8])
```

Out[81]:

```
array([10, 18, 17], dtype=int32)
```

In [82]:

```
# The results are the reductions (here, sums) performed over arr[0:5], arr[5:8], and arr
[8:].
    # As with the other methods, you can pass an axis argument:


arr = np.multiply.outer(np.arange(4), np.arange(5))

arr
```

Out[82]:

```
array([[ 0,  0,  0,  0,  0],
       [ 0,  1,  2,  3,  4],
       [ 0,  2,  4,  6,  8],
       [ 0,  3,  6,  9, 12]])
```

In [90]:

```
np.add.reduceat(arr, [0, 2, 4], axis=1)
```

Out[90]:

```
array([[ 0,  0,  0],
       [ 1,  5,  4],
       [ 2, 10,  8],
       [ 3, 15, 12]], dtype=int32)
```

In [95]:

```
# Writing New ufuncs in Python

# There are a number of facilities for creating your own NumPy ufuncs.
    # The most general is to use the NumPy C API, but let's look at pure Python ufuncs.

# numpy.frompyfunc accepts a Python function along with a specification for the number of
 inputs and outputs.
    # For example, a simple function that adds element-wise would be specified as:


def add_elements(x, y):
    return x + y
```

In [97]:

```
add_them = np.frompyfunc(add_elements, 2, 1)

add_them
```

Out[97]:

```
<ufunc '? (vectorized)'>
```

In [98]:

```
add_them(np.arange(8), np.arange(8))
```

Out[98]:

```
array([0, 2, 4, 6, 8, 10, 12, 14], dtype=object)
```

In [100]:

```
# Functions created using frompyfunc always return arrays of Python objects,
    # which can be inconvenient.
    # Fortunately, there is an alternative (but slightly less featureful) function, numpy.
vectorize,
    # that allows you to specify the output type:


add_them = np.vectorize(add_elements, otypes=[np.float64])

add_them
```

Out[100]:

```
<numpy.vectorize at 0xa010275188>
```

In [101]:

```
add_them(np.arange(8), np.arange(8))
```

Out[101]:

```
array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14.])
```

In [109]:

```
# These functions provide a way to create ufunc-like functions,
    # but they are very slow because they require a Python function call to compute each e
lement,
    # which is a lot slower than NumPy's C-based ufunc loops:

arr = np.random.randn(10000)

arr
```

Out[109]:

```
array([ 0.10533048, -0.81101068, -1.14124808, ...,  1.16323222,
       -0.80433614,  0.08255916])
```

In [112]:

```
%timeit add_them(arr, arr)
```

```
5.67 ms ± 723 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [113]:

```
%timeit np.add(arr, arr)
```

```
17.9 µs ± 1.05 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

In [115]:

```
# Structured and Record Arrays

# You may have noticed up until now that ndarray is a homogeneous data container;
    # that is, it represents a block of memory in which each element takes up the same num
ber of bytes,
    # determined by the dtype.
    # On the surface, this would appear to not allow you to represent heterogeneous or tab
ular-like data.
    # A structured array is an ndarray in which each element can be thought of as represen
ting a struct in C
    # (hence the "structured" name) or a row in a SQL table with multiple named fields:

dtype = [('x', np.float64), ('y', np.int32)]

dtype
```

Out[115]:

```
[('x', numpy.float64), ('y', numpy.int32)]
```

In [116]:

```
sarr = np.array([(1.5, 6), (np.pi, -2)], dtype=dtype)

sarr
```

Out[116]:

```
array([(1.5       ,  6), (3.14159265, -2)],
      dtype=[('x', '<f8'), ('y', '<i4')])
```

In [117]:

```
# There are several ways to specify a structured dtype (see the online NumPy documentatio
n).
    # One typical way is as a list of tuples with (field_name, field_data_type).
    # Now, the elements of the array are tuple-like objects whose elements can be accessed
like a dictionary:


sarr[0]
```

Out[117]:

```
(1.5, 6)
```

In [118]:

```
sarr[0]['y']
```

Out[118]:

```
6
```

In [119]:

```
# The field names are stored in the dtype.names attribute.
    # When you access a field on the structured array, a strided view on the data is retur
ned,
    # thus copying nothing:


sarr['x']
```

Out[119]:

```
array([1.5       , 3.14159265])
```

In [120]:

```
# Nested dtypes and Multidimensional Fields

# When specifying a structured dtype, you can additionally pass a shape (as an int or tupl
e):


dtype = [('x', np.int64, 3), ('y', np.int32)]

dtype
```

Out[120]:

```
[('x', numpy.int64, 3), ('y', numpy.int32)]
```

In [123]:

```
arr = np.zeros(4, dtype=dtype)

arr
```

Out[123]:

```
array([([0, 0, 0], 0), ([0, 0, 0], 0), ([0, 0, 0], 0), ([0, 0, 0], 0)],
      dtype=[('x', '<i8', (3,)), ('y', '<i4')])
```

In [124]:

```
# In this case, the x field now refers to an array of length 3 for each record:


arr[0]['x']
```

Out[124]:

```
array([0, 0, 0], dtype=int64)
```

In [128]:

```
# Conveniently, accessing arr['x'] then returns a two-dimensional array
    # instead of a one-dimensional array as in prior examples:


arr['x']
```

Out[128]:

```
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]], dtype=int64)
```

In [130]:

```
# This enables you to express more complicated, nested structures as a single block of mem
ory in an array.
    # You can also nest dtypes to make more complex structures.
    # Here is an example:


dtype = [('x', [('a', 'f8'), ('b', 'f4')]), ('y', np.int32)]

dtype
```

Out[130]:

```
[('x', [('a', 'f8'), ('b', 'f4')]), ('y', numpy.int32)]
```

In [131]:

```
data = np.array([((1, 2), 5), ((3, 4), 6)], dtype=dtype)

data['x']
```

Out[131]:

```
array([(1., 2.), (3., 4.)], dtype=[('a', '<f8'), ('b', '<f4')])
```

In [132]:

```
data['y']
```

Out[132]:

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

In [133]:

```
data['x']['a']
```

Out[133]:

```
array([1., 3.])
```

In [ ]:

```
# pandas DataFrame does not support this feature directly, though it is similar to hierarc
hical indexing.
```

In [ ]:

```
# Why Use Structured Arrays?

# Compared with, say, a pandas DataFrame, NumPy structured arrays are a comparatively low-
level tool.
    # They provide a means to interpreting a block of memory as a tabular structure
    # with arbitrarily complex nested columns.
    # Since each element in the array is represented in memory as a fixed number of bytes,
    # structured arrays providea very fast and efficient way of writing data to
    # and from disk (including memory maps), transporting it over the network, and other s
uch uses.

# As another common use for structured arrays, writing data files as fixed-length
    # record byte streams is a common way to serialize data in C and C++ code,
    # which is commonly found in legacy systems in industry.
    # As long as the format of the file is known
    # (the size of each record and the order, byte size, and data type of each element),
    # the data can be read into memory with np.fromfile.
```

In [135]:

```
# More About Sorting

# Like Python’s built-in list, the ndarray sort instance method is an in-place sort,
    # meaning that the array contents are rearranged without producing a new array:


arr = np.random.randn(6)

arr
```

Out[135]:

```
array([-1.4199867 , -0.04596205,  1.27691904,  1.86071261, -0.31313357,
       -0.22521732])
```

In [142]:

```
# When sorting arrays in-place, remember that if the array is a view on a different ndarra
y,
    # the original array will be modified:


arr = np.random.randn(3, 5)

arr
```

Out[142]:

```
array([[-0.28487369, -0.41144101, -0.39848164, -0.85722451,  0.04447328],
       [-1.92493444, -0.18016988, -0.61207016, -0.02848961, -0.00939751],
       [ 1.16893736,  0.34476488,  0.8829256 , -1.63350964, -0.05559684]])
```

In [146]:

```
arr[:, 0].sort() # Sort first column values in-place

arr
```

Out[146]:

```
array([[-1.92493444, -0.41144101, -0.39848164, -0.85722451,  0.04447328],
       [-0.28487369, -0.18016988, -0.61207016, -0.02848961, -0.00939751],
       [ 1.16893736,  0.34476488,  0.8829256 , -1.63350964, -0.05559684]])
```

In [147]:

```
# On the other hand, numpy.sort creates a new, sorted copy of an array.
    # Otherwise, it accepts the same arguments (such as kind) as ndarray.sort:


arr = np.random.randn(5)

arr
```

Out[147]:

```
array([ 0.23207772,  0.94006741, -0.1233083 ,  0.43651008, -1.0445751 ])
```

In [148]:

```
np.sort(arr)
```

Out[148]:

```
array([-1.0445751 , -0.1233083 ,  0.23207772,  0.43651008,  0.94006741])
```

In [149]:

```
arr
```

Out[149]:

```
array([ 0.23207772,  0.94006741, -0.1233083 ,  0.43651008, -1.0445751 ])
```

In [150]:

```
# All of these sort methods take an axis argument for sorting the sections of data
    # along the passed axis independently:


arr = np.random.randn(3, 5)

arr
```

Out[150]:

```
array([[ 0.37039574, -0.09319317,  0.15047656,  0.60120384,  0.2195671 ],
       [-2.01196694, -0.69332216, -0.24604222, -1.00853706,  0.52522324],
       [-2.70506475,  1.24593288,  0.75426158,  0.77936106,  0.96449794]])
```

In [152]:

```
arr.sort(axis=1)

arr
```

Out[152]:

```
array([[-0.09319317,  0.15047656,  0.2195671 ,  0.37039574,  0.60120384],
       [-2.01196694, -1.00853706, -0.69332216, -0.24604222,  0.52522324],
       [-2.70506475,  0.75426158,  0.77936106,  0.96449794,  1.24593288]])
```

In [153]:

```
arr.sort(axis=0)

arr
```

Out[153]:

```
array([[-2.70506475, -1.00853706, -0.69332216, -0.24604222,  0.52522324],
       [-2.01196694,  0.15047656,  0.2195671 ,  0.37039574,  0.60120384],
       [-0.09319317,  0.75426158,  0.77936106,  0.96449794,  1.24593288]])
```

In [154]:

```
# You may notice that none of the sort methods have an option to sort in descending order.
    # This is a problem in practice because array slicing produces views,
    # thus not producing a copy or requiring any computational work.
    # Many Python users are familiar with the “trick” that for a list values,
    # values[::-1] returns a list in reverse order.
    # The same is true for ndarrays:


arr[:, ::-1]
```

Out[154]:

```
array([[ 0.52522324, -0.24604222, -0.69332216, -1.00853706, -2.70506475],
       [ 0.60120384,  0.37039574,  0.2195671 ,  0.15047656, -2.01196694],
       [ 1.24593288,  0.96449794,  0.77936106,  0.75426158, -0.09319317]])
```

In [156]:

```
# Indirect Sorts: argsort and lexsort

# In data analysis you may need to reorder datasets by one or more keys.
    # For example, a table of data about some students might need to be sorted by last nam
e,
    # then by first name.
    # This is an example of an indirect sort, and if you’ve read the pandas-related books
    # you must have already seen many higher-level examples.
    # Given a key or keys (an array of values or multiple arrays of values),
    # you wish to obtain an array of integer indices (I refer to them colloquially as inde
xers)
    # that tells you how to reorder the data to be in sorted order.
    # Two methods for this are argsort and numpy.lexsort.
    # As an example:


values = np.array([5, 0, 1, 3, 2])

values
```

Out[156]:

```
array([5, 0, 1, 3, 2])
```

In [158]:

```
indexer = values.argsort()

indexer
```

Out[158]:

```
array([1, 2, 4, 3, 0], dtype=int64)
```

In [159]:

```
values[indexer]
```

Out[159]:

```
array([0, 1, 2, 3, 5])
```

In [160]:

```
# As a more complicated example, this code reorders a two-dimensional array by its first r
ow:


arr = np.random.randn(3, 5)

arr
```

Out[160]:

```
array([[ 0.97544999,  1.63128663, -0.54927156, -0.3058123 ,  0.29766513],
       [ 0.45944142,  1.36198492,  0.49668379, -0.49842716, -0.39958665],
       [-1.2787087 ,  0.81086645, -0.13821823,  0.48103853, -0.44511099]])
```

In [161]:

```
arr[0] = values
```

In [162]:

```
arr
```

Out[162]:

```
array([[ 5.        ,  0.        ,  1.        ,  3.        ,  2.        ],
       [ 0.45944142,  1.36198492,  0.49668379, -0.49842716, -0.39958665],
       [-1.2787087 ,  0.81086645, -0.13821823,  0.48103853, -0.44511099]])
```

In [163]:

```
arr[:, arr[0].argsort()]
```

Out[163]:

```
array([[ 0.        ,  1.        ,  2.        ,  3.        ,  5.        ],
       [ 1.36198492,  0.49668379, -0.39958665, -0.49842716,  0.45944142],
       [ 0.81086645, -0.13821823, -0.44511099,  0.48103853, -1.2787087 ]])
```

In [165]:

```
# Lexsort is similar to argsort, but it performs an indirect lexicographical sort on multi
ple key arrays.
    # Suppose we wanted to sort some data identified by first and last names:


first_name = np.array(['Bob', 'Jane', 'Steve', 'Bill', 'Barbara'])

first_name
```

Out[165]:

```
array(['Bob', 'Jane', 'Steve', 'Bill', 'Barbara'], dtype='<U7')
```

In [166]:

```
last_name = np.array(['Jones', 'Arnold', 'Arnold', 'Jones', 'Walters'])

last_name
```

Out[166]:

```
array(['Jones', 'Arnold', 'Arnold', 'Jones', 'Walters'], dtype='<U7')
```

In [167]:

```
sorter = np.lexsort((first_name, last_name))

sorter
```

Out[167]:

```
array([1, 2, 3, 0, 4], dtype=int64)
```

In [168]:

```
zip(last_name[sorter], first_name[sorter])
```

Out[168]:

```
<zip at 0xa01026aac8>
```

In [ ]:

```
# lexsort can be a bit confusing the first time you use it because the order in which the
    # keys are used to order the data starts with the last array passed.
    # Above, last_name was used before first_name.


# pandas methods like Series’s and DataFrame’s sort_values method are implemented with var
iants
    # of these functions (which also must take into account missing values).
```

In [170]:

```
# Alternative Sort Algorithms

# A stable sorting algorithm preserves the relative position of equal elements.
    # This can be especially important in indirect sorts where the relative ordering is me
aningful:


values = np.array(['2:first', '2:second', '1:first', '1:second', '1:third'])

values
```

Out[170]:

```
array(['2:first', '2:second', '1:first', '1:second', '1:third'],
      dtype='<U8')
```

In [171]:

```
key = np.array([2, 2, 1, 1, 1])

key
```

Out[171]:

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

In [172]:

```
indexer = key.argsort(kind='mergesort')

indexer
```

Out[172]:

```
array([2, 3, 4, 0, 1], dtype=int64)
```

In [173]:

```
values.take(indexer)
```

Out[173]:

```
array(['1:first', '1:second', '1:third', '2:first', '2:second'],
      dtype='<U8')
```

In [ ]:

```
# The only stable sort available is mergesort, which has guaranteed O(n log n) performance
    # (for complexity buffs), but its performance is on average worse than the default qui
cksort method.
    # This is not something that most users will ever have to think about,
    # but it’s useful to know that it’s there.

# A summary of available array sorting methods and their relative performance (and perform
ance guarantees):


#                Kind            Speed    Stable    Work space     Worst case
#                'quicksort'     1        No        0              O(n^2)
#                'mergesort'     2        Yes       n / 2          O(n log n)
#                'heapsort'      3        No        0              O(n log n)
```

In [175]:

```
# Partially Sorting Arrays

# One of the goals of sorting can be to determine the largest or smallest elements in an a
rray.
    # NumPy has optimized methods, numpy.partition and np.argpartition,
    # for partitioning an array around the k-th smallest element:


np.random.seed(12345)

arr = np.random.randn(20)

arr
```

Out[175]:

```
array([-0.20470766,  0.47894334, -0.51943872, -0.5557303 ,  1.96578057,
        1.39340583,  0.09290788,  0.28174615,  0.76902257,  1.24643474,
        1.00718936, -1.29622111,  0.27499163,  0.22891288,  1.35291684,
        0.88642934, -2.00163731, -0.37184254,  1.66902531, -0.43856974])
```

In [177]:

```
np.partition(arr, 3)
```

Out[177]:

```
array([-2.00163731, -1.29622111, -0.5557303 , -0.51943872, -0.37184254,
       -0.43856974, -0.20470766,  0.28174615,  0.76902257,  0.47894334,
        1.00718936,  0.09290788,  0.27499163,  0.22891288,  1.35291684,
        0.88642934,  1.39340583,  1.96578057,  1.66902531,  1.24643474])
```

In [178]:

```
# After you call partition(arr, 3), the first three elements in the result are the smalles
t three values
    # in no particular order.
    # numpy.argpartition, similar to numpy.argsort, returns the indices that rearrange the
data
    # into the equivalent order:


indices = np.argpartition(arr, 3)

indices
```

Out[178]:

```
array([16, 11,  3,  2, 17, 19,  0,  7,  8,  1, 10,  6, 12, 13, 14, 15,  5,
        4, 18,  9], dtype=int64)
```

In [179]:

```
arr.take(indices)
```

Out[179]:

```
array([-2.00163731, -1.29622111, -0.5557303 , -0.51943872, -0.37184254,
       -0.43856974, -0.20470766,  0.28174615,  0.76902257,  0.47894334,
        1.00718936,  0.09290788,  0.27499163,  0.22891288,  1.35291684,
        0.88642934,  1.39340583,  1.96578057,  1.66902531,  1.24643474])
```

In [181]:

```
# numpy.searchsorted: Finding Elements in a Sorted Array

#  searchsorted is an array method that performs a binary search on a sorted array,
    # returning the location in the array where the value would need to be inserted to mai
ntain sortedness:


arr = np.array([0, 1, 7, 12, 15])

arr
```

Out[181]:

```
array([ 0,  1,  7, 12, 15])
```

In [182]:

```
arr.searchsorted(9)
```

Out[182]:

```
3
```

In [183]:

```
# You can also pass an array of values to get an array of indices back:


arr.searchsorted([0, 8, 11, 16])
```

Out[183]:

```
array([0, 3, 3, 5], dtype=int64)
```

In [184]:

```
# You might have noticed that searchsorted returned 0 for the 0 element.
    # This is because the default behavior is to return the index at the left side of a gr
oup of equal values:


arr = np.array([0, 0, 0, 1, 1, 1, 1])

arr
```

Out[184]:

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

In [185]:

```
arr.searchsorted([0, 1])
```

Out[185]:

```
array([0, 3], dtype=int64)
```

In [186]:

```
arr.searchsorted([0, 1], side='right')
```

Out[186]:

```
array([3, 7], dtype=int64)
```

In [188]:

```
# As another application of searchsorted, suppose we had an array of values between 0 and
 10,000,
    # and a separate array of “bucket edges” that we wanted to use to bin the data:


data = np.floor(np.random.uniform(0, 10000, size=50))

data
```

Out[188]:

```
array([2449., 7928., 4951., 9150., 9453., 5332., 2524., 7208., 3674.,
       4986., 2265., 3535., 6508., 3129., 7687., 7818., 8524., 9499.,
       1073., 9107., 3360., 8263., 8981.,  427., 1957., 2945., 6269.,
        862., 1429., 5158., 6893., 8566., 6473., 5816., 7111., 2524.,
       9001., 4422.,  205., 9596., 6522., 5132., 6823., 4895., 9264.,
       5158.,  721., 5675., 6152., 9415.])
```

In [189]:

```
bins = np.array([0, 100, 1000, 5000, 10000])

bins
```

Out[189]:

```
array([    0,   100,  1000,  5000, 10000])
```

In [191]:

```
#To then get a labeling of which interval each data point belongs to (where 1 would mean t
he bucket [0, 100)),
    # we can simply use searchsorted:


labels = bins.searchsorted(data)

labels
```

Out[191]:

```
array([3, 4, 3, 4, 4, 4, 3, 4, 3, 3, 3, 3, 4, 3, 4, 4, 4, 4, 3, 4, 3, 4,
       4, 2, 3, 3, 4, 2, 3, 4, 4, 4, 4, 4, 4, 3, 4, 3, 2, 4, 4, 4, 4, 3,
       4, 4, 2, 4, 4, 4], dtype=int64)
```

In [194]:

```
# This, combined with pandas’s groupby, can be used to bin data:

pd.Series(data).groupby(labels).mean()
```

Out[194]:

```
2     553.750000
3    3132.375000
4    7482.733333
dtype: float64
```

In [217]:

```
# Writing Fast NumPy Functions with Numba

# Numba is an open source project that creates fast functions for NumPy-like data using CP
Us, GPUs,
    # or other hardware. It uses the LLVM Project to translate Python code into compiled m
achine code.

# To introduce Numba, let’s consider a pure Python function that computes
    # the expression (x - y).mean() using a for loop:


def mean_distance(x, y):
    nx = len(x)
    result = 0.0
    count = 0
    for i in range(nx):
        result += x[i] - y[i]
        count += 1
    return result / count
```

In [199]:

```
#This function is very slow:


x = np.random.randn(10000000)

x
```

Out[199]:

```
array([-1.56565729, -0.56254019, -0.03266414, ..., -2.57718168,
        0.42536042,  1.37299858])
```

In [200]:

```
y = np.random.randn(10000000)

y
```

Out[200]:

```
array([-0.24829772, -0.1873352 , -0.7450831 , ...,  2.10103003,
        0.26415342,  0.73032225])
```

In [203]:

```
%timeit mean_distance(x, y)
```

```
24 s ± 591 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [204]:

```
%timeit (x - y).mean()
```

```
231 ms ± 42.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [218]:

```
# The NumPy version is over 100 times faster.
    # We can turn this function into a compiled Numba function using the numba.jit functio
n:


import numba as nb

numba_mean_distance = nb.jit(mean_distance)

numba_mean_distance
```

Out[218]:

```
CPUDispatcher(<function mean_distance at 0x000000A01CA935E8>)
```

In [219]:

```
# We could also have written this as a decorator:


@nb.jit
def mean_distance(x, y):
    nx = len(x)
    result = 0.0
    count = 0
    for i in range(nx):
        result += x[i] - y[i]
        count += 1
    return result / count
```

In [223]:

```
# The resulting function is actually faster than the vectorized NumPy version:


%timeit numba_mean_distance(x, y)
```

```
2.04 µs ± 168 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

In [224]:

```
# Numba cannot compile arbitrary Python code, but it supports a significant subset of pure
Python
    # that is most useful for writing numerical algorithms.

# Numba is a deep library, supporting different kinds of hardware, modes of compilation, a
nd user extensions.
    # It is also able to compile a substantial subset of the NumPy Python API without expl
icit for loops.
    # Numba is able to recognize constructs that can be compiled to machine code,
    # while substituting calls to the CPython API for functions that it does not know how
 to compile.
    # Numba’s jit function has an option, nopython=True, which restricts allowed code to P
ython code that can
    # be compiled to LLVM without any Python C API calls. jit(nopython=True) has a shorter
alias numba.njit.

# In the previous example, we could have written:


from numba import float64, njit

@njit(float64(float64[:], float64[:]))
def mean_distance(x, y):
    return (x - y).mean()
```

In [ ]:

```
# I encourage you to learn more by reading the online documentation for Numba.
```

In [225]:

```
# Creating Custom numpy.ufunc Objects with Numba


# The numba.vectorize function creates compiled NumPy ufuncs, which behave like built-in u
funcs.
    # Let’s consider a Python implementation of numpy.add:


from numba import vectorize

@vectorize
def nb_add(x, y):
    return x + y
```

In [226]:

```
# Now we have:


x = np.arange(10)

x
```

Out[226]:

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [227]:

```
nb_add(x, x)
```

Out[227]:

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18], dtype=int64)
```

In [228]:

```
nb_add.accumulate(x, 0)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-228-e53ce9bd412f> in <module>
----> 1 nb_add.accumulate(x, 0)

ValueError: could not find a matching type for nb_add.accumulate, requested t
ype has type code 'l'
```

In [ ]:

```
# Advanced Array Input and Output

# Ordinarily in numpy, np.save and np.load are for storing arrays in binary format on dis
k.
    # There are a number of additional options to consider for more sophisticated use.
    # In particular, memory maps have the additional benefit of enabling you to work with
 datasets
    # that do not fit into RAM.
```

In [229]:

```
# Memory-Mapped Files

# A memory-mapped file is a method for interacting with binary data on disk as though it i
s stored
    # in an in-memory array.
    # NumPy implements a memmap object that is ndarray-like, enabling small segments of a
 large file to be read
    # and written without reading the whole array into memory.
    # Additionally, a memmap has the same methods as an in-memory array and thus can be su
bstituted
    # into many algorithms where an ndarray would be expected.

# To create a new memory map, use the function np.memmap and pass a file path, dtype,shap
e, and file mode:


mmap = np.memmap('mymmap', dtype='float64', mode='w+', shape=(10000, 10000))

mmap
```

Out[229]:

```
memmap([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]])
```

In [230]:

```
# Slicing a memmap returns views on the data on disk:


section = mmap[:5]

section
```

Out[230]:

```
memmap([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]])
```

In [232]:

```
# If you assign data to these, it will be buffered in memory (like a Python file object),
    # but you can write it to disk by calling flush:

section[:] = np.random.randn(5, 10000)

section
```

Out[232]:

```
memmap([[ 0.56711132, -0.12717231,  0.52758445, ..., -0.51964959,
         -0.55576608, -0.62599963],
        [-0.60118962, -0.2320138 ,  1.78400269, ...,  0.5460359 ,
         -0.81396878, -0.46026551],
        [ 0.14608177, -1.1583803 , -1.28189275, ...,  0.53420363,
          0.09238763, -1.28271782],
        [-1.5121209 ,  0.9521015 , -0.87179321, ..., -1.28613331,
          0.01396661, -0.4403933 ],
        [-0.80654508,  2.11578824, -2.0804114 , ...,  1.12830642,
         -1.14187654,  0.13829511]])
```

In [233]:

```
mmap.flush()      # writing it to disk
```

In [234]:

```
mmap
```

Out[234]:

```
memmap([[ 0.56711132, -0.12717231,  0.52758445, ..., -0.51964959,
         -0.55576608, -0.62599963],
        [-0.60118962, -0.2320138 ,  1.78400269, ...,  0.5460359 ,
         -0.81396878, -0.46026551],
        [ 0.14608177, -1.1583803 , -1.28189275, ...,  0.53420363,
          0.09238763, -1.28271782],
        ...,
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ]])
```

In [235]:

```
del mmap
```

In [236]:

```
# Whenever a memory map falls out of scope and is garbage-collected,
    # any changes will be flushed to disk also.
    # When opening an existing memory map, you still have to specify the dtype and shape,
    # as the file is only a block of binary data with no metadata on disk:


mmap = np.memmap('mymmap', dtype='float64', shape=(10000, 10000))

mmap
```

Out[236]:

```
memmap([[ 0.56711132, -0.12717231,  0.52758445, ..., -0.51964959,
         -0.55576608, -0.62599963],
        [-0.60118962, -0.2320138 ,  1.78400269, ...,  0.5460359 ,
         -0.81396878, -0.46026551],
        [ 0.14608177, -1.1583803 , -1.28189275, ...,  0.53420363,
          0.09238763, -1.28271782],
        ...,
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ]])
```

In [ ]:

```
# Memory maps also work with structured or nested dtypes as described in a previously
```

In [ ]:

```
# Performance Tips

# Getting good performance out of code utilizing NumPy is often straightforward,
    # as array operations typically replace otherwise comparatively extremely slow pure Py
thon loops.
    # The following list briefly summarizes some things to keep in mind:

#         • Convert Python loops and conditional logic to array operations and boolean arra
y operations
#         • Use broadcasting whenever possible
#         • Use arrays views (slicing) to avoid copying data
#         • Utilize ufuncs and ufunc methods

# If you can’t get the performance you require after exhausting the capabilities provided
 by NumPy alone,
    # consider writing code in C, Fortran, or Cython.
```

In [237]:

```
# The Importance of Contiguous Memory

# In some applications the memory layout of an array can significantly affect the speed of
computations.
    # This is based partly on performance differences having to do with the cache hierarch
y of the CPU;
    # operations accessing contiguous blocks of memory (e.g., summing the rows of a C orde
r array)
    # will generally be the fastest because the memory subsystem will buffer the appropria
te blocks of memory
    # into the ultrafast L1 or L2 CPU cache.
    # Also, certain code paths inside NumPy’s C codebase have been optimized for the conti
guous case
    # in which generic strided memory access can be avoided.

# To say that an array’s memory layout is contiguous means that the elements are stored in
memory in the order
    # that they appear in the array with respect to Fortran (columnmajor) or C (row major)
ordering.
    # By default, NumPy arrays are created as Ccontiguous or just simply contiguous.
    # A column major array, such as the transpose of a C-contiguous array,
    # is thus said to be Fortran-contiguous.
    # These properties can be explicitly checked via the flags attribute on the ndarray:


arr_c = np.ones((1000, 1000), order='C')

arr_c
```

Out[237]:

```
array([[1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       ...,
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.]])
```

In [238]:

```
arr_f = np.ones((1000, 1000), order='F')

arr_f
```

Out[238]:

```
array([[1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       ...,
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.]])
```

In [239]:

```
arr_c.flags
```

Out[239]:

```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

In [240]:

```
arr_f.flags
```

Out[240]:

```
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

In [242]:

```
arr_c.flags.c_contiguous
```

Out[242]:

```
True
```

In [243]:

```
arr_c.flags.f_contiguous
```

Out[243]:

```
False
```

In [244]:

```
arr_f.flags.c_contiguous
```

Out[244]:

```
False
```

In [245]:

```
arr_f.flags.f_contiguous
```

Out[245]:

```
True
```

In [250]:

```
# In this example, summing the rows of these arrays should, in theory, be faster for arr_c
than arr_f
    # since the rows are contiguous in memory.

# Here I check for sure using %timeit:


%timeit arr_c.sum(1)
```

```
5.77 ms ± 782 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [251]:

```
%timeit arr_f.sum(1)
```

```
2.54 ms ± 944 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [252]:

```
# When you’re looking to squeeze more performance out of NumPy, this is often a place to i
nvest some effort.
    # If you have an array that does not have the desired memory order,
    # you can use copy and pass either 'C' or 'F':

arr_f.copy('C').flags
```

Out[252]:

```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

In [253]:

```
arr_f.copy('F').flags
```

Out[253]:

```
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

In [254]:

```
arr_c.copy('F').flags
```

Out[254]:

```
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

In [255]:

```
arr_c.copy('C').flags
```

Out[255]:

```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

In [256]:

```
# When constructing a view on an array, keep in mind that the result is not guaranteed to
 be contiguous:

arr_c[:50].flags.contiguous
```

Out[256]:

```
True
```

In [257]:

```
arr_c[:, :50].flags
```

Out[257]:

```
  C_CONTIGUOUS : False
  F_CONTIGUOUS : False
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

The End