SEAS 6414 – Python Applications in Data Analytics

THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

Dr. Adewale Akinfaderin

Lecture 3 - Spring 2024

Introduction to Numpy

- NumPy, or Numerical Python, is crucial for numerical computing in Python.
- It acts as a standard interface for many scientific computing packages.
- Understanding NumPy aids in using pandas for array-oriented computing.
- Facilitates integration with C, C++, or FORTRAN libraries via C API.
- Essential for applications requiring interfacing with legacy codebases.

Key Features of NumPy

- NumPy's ndarray enables efficient multi-dimensional array operations.
- Mathematical functions allow fast operations across entire data arrays.
- Tools for reading/writing array data and handling memory-mapped files.
- Offers linear algebra, random number generation, and Fourier transform capabilities.
- Efficient memory usage and performance compared to Python sequences.

NumPy in Advanced Data Analysis

- Critical for effective array-based data manipulation and computation.
- Supports fast operations for data cleaning, subsetting, and filtering.
- Efficient in performing common array algorithms and descriptive statistics.
- Enhances data alignment and complex relational data manipulations.
- Enables concise expression of conditional logic and group-wise data operations.

NumPy's Significance and Historical Context

- Serves as a foundation for pandas, especially in handling tabular data.
- Designed for large-scale data efficiency, leveraging contiguous memory blocks.
- Streamlines complex computations with minimal overhead.
- Originates from Numeric library in 1995, unified as NumPy in 2005.
- A key player in Python's evolution into a major scientific computing language.

Numpy Performance

To give you an idea of the performance difference, consider a NumPy array of one million integers, and the equivalent Python list:

```
In [7]: import numpy as np

In [8]: my_arr = np.arange(1_000_000)

In [9]: my_list = list(range(1_000_000))
```

Now let's multiply each sequence by 2:

```
In [10]: %timeit my_arr2 = my_arr * 2
309 us +- 7.48 us per loop (mean +- std. dev. of 7 runs, 1000 loops each)

In [11]: %timeit my_list2 = [x * 2 for x in my_list]
46.4 ms +- 526 us per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

NumPy-based algorithms are generally 10 to 100 times faster (or more) than their pure Python counterparts and use significantly less memory.

Numpy ndarray

One of the key features of NumPy is its N-dimensional array object, or ndarray, which is a fast, flexible container for large datasets in Python. Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements.

To give you a flavor of how NumPy enables batch computations with similar syntax to scalar values on built-in Python objects, I first import NumPy and create a small array:

```
In [12]: import numpy as np

In [13]: data = np.array([[1.5, -0.1, 3], [0, -3, 6.5]])

In [14]: data
Out[14]:
array([[ 1.5, -0.1,  3. ],
       [ 0. , -3. ,  6.5]])
```

Numpy ndarray

I then write mathematical operations with `data`:

```
In [15]: data * 10
Out[15]:
array([[ 15.,  -1.,  30.],
       [  0., -30.,  65.]])

In [16]: data + data
Out[16]:
array([[ 3. , -0.2,  6. ],
       [ 0. , -6. , 13. ]])
```

In the first example, all of the elements have been multiplied by 10. In the second, the corresponding values in each "cell" in the array have been added to each other.

Numpy ndarray

An ndarray is a generic multidimensional container for homogeneous data; that is, all of the elements must be the same type. Every array has a `shape`, a tuple indicating the size of each dimension, and a `dtype`, an object describing the *data type* of the array:

```
In [17]: data.shape
Out[17]: (2, 3)

In [18]: data.dtype
Out[18]: dtype('float64')
```

Creating ndarrays

The easiest way to create an array is to use the `array` function. This accepts any sequence-like object (including other arrays) and produces a new NumPy array containing the passed data. For example, a list is a good candidate for conversion:

```
In [19]: data1 = [6, 7.5, 8, 0, 1]

In [20]: arr1 = np.array(data1)

In [21]: arr1
Out[21]: array([6. , 7.5, 8. , 0. , 1. ])
```

Nested sequences, like a list of equal-length lists, will be converted into a multidimensional array:

```
In [22]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

In [23]: arr2 = np.array(data2)

In [24]: arr2
Out[24]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

Creating ndarrays

Since `data2` was a list of lists, the NumPy array `arr2` has two dimensions, with shape inferred from the data. We can confirm this by inspecting the `ndim` and `shape` attributes:

```
In [25]: arr2.ndim
Out[25]: 2

In [26]: arr2.shape
Out[26]: (2, 4)
```

Unless explicitly specified (discussed in Data Types for ndarrays), `numpy.array` tries to infer a good data type for the array that it creates. The data type is stored in a special `dtype` metadata object; for example, in the previous two examples we have:

```
In [27]: arr1.dtype
Out[27]: dtype('float64')

In [28]: arr2.dtype
Out[28]: dtype('int64')
```

Creating ndarrays

In addition to `numpy.array`, there are a number of other functions for creating new arrays. As examples, `numpy.zeros` and `numpy.ones` create arrays of 0s or 1s, respectively, with a given length or shape. `numpy.empty` creates an array without initializing its values to any particular value. To create a higher dimensional array with these methods, pass a tuple for the shape:

```
In [29]: np.zeros(10)
Out[29]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

In [30]: np.zeros((3, 6))
Out[30]:
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])

In [31]: np.empty((2, 3, 2))
Out[31]:
array([[[0., 0.],
        [0., 0.],
        [0., 0.]],
       [[0., 0.],
        [0., 0.],
        [0., 0.]]])
```

Creating ndarrays

`numpy.arange` is an array-valued version of the built-in Python `range` function:

```
In [32]: np.arange(15)
Out[32]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Function	Description
`array`	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a data type or explicitly specifying a data type; copies the input data by default
`asarray`	Convert input to ndarray, but do not copy if the input is already an ndarray
`arange`	Like the built-in `range` but returns an ndarray instead of a list

Creating ndarrays

`ones`, `ones_like`	Produce an array of all 1s with the given shape and data type; `ones_like` takes another array and produces a `ones` array of the same shape and data type
`zeros`, `zeros_like`	Like `ones` and `ones_like` but producing arrays of 0s instead
`empty`, `empty_like`	Create new arrays by allocating new memory, but do not populate with any values like `ones` and `zeros`
`full`, `full_like`	Produce an array of the given shape and data type with all values set to the indicated "fill value"; `full_like` takes another array and produces a filled array of the same shape and data type
`eye`, `identity`	Create a square N × N identity matrix (1s on the diagonal and 0s elsewhere)

Data Types for ndarrays

The *data type* or `dtype` is a special object containing the information (or *metadata*, data about data) the ndarray needs to interpret a chunk of memory as a particular type of data:

```
In [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [34]: arr2 = np.array([1, 2, 3], dtype=np.int32)

In [35]: arr1.dtype
Out[35]: dtype('float64')

In [36]: arr2.dtype
Out[36]: dtype('int32')
```

Data types are a source of NumPy's flexibility for interacting with data coming from other systems. In most cases they provide a mapping directly onto an underlying disk or memory representation, which makes it possible to read and write binary streams of data to disk and to connect to code written in a low-level language like C or FORTRAN. The numerical data types are named the same way: a type name, like `float` or `int`, followed by a number indicating the number of bits per element. A standard double-precision floating-point value (what's used under the hood in Python's `float` object) takes up 8 bytes or 64 bits. Thus, this type is known in NumPy as `float64`. See Table 4.2 for a full listing of NumPy's supported data types.

Data Types for ndarrays

Table 4.2: NumPy data types

Type	Type code	Description
`int8, uint8`	`i1, u1`	Signed and unsigned 8-bit (1 byte) integer types
`int16, uint16`	`i2, u2`	Signed and unsigned 16-bit integer types
`int32, uint32`	`i4, u4`	Signed and unsigned 32-bit integer types
`int64, uint64`	`i8, u8`	Signed and unsigned 64-bit integer types
`float16`	`f2`	Half-precision floating point
`float32`	`f4 or f`	Standard single-precision floating point; compatible with C float
`float64`	`f8 or d`	Standard double-precision floating point; compatible with C double and Python `float` object
`float128`	`f16 or g`	Extended-precision floating point

Data Types for ndarrays

`complex64`, `complex128`, `complex256`	`c8`, `c16`, `c32`	Complex numbers represented by two 32, 64, or 128 floats, respectively
`bool`	?	Boolean type storing `True` and `False` values
`object`	O	Python object type; a value can be any Python object
`string_`	S	Fixed-length ASCII string type (1 byte per character); for example, to create a string data type with length 10, use `'S10'`
`unicode_`	U	Fixed-length Unicode type (number of bytes platform specific); same specification semantics as `string_` (e.g., `'U10'`)

Data Types for ndarrays

You can explicitly convert or *cast* an array from one data type to another using ndarray's `astype` method:

```
In [37]: arr = np.array([1, 2, 3, 4, 5])

In [38]: arr.dtype
Out[38]: dtype('int64')

In [39]: float_arr = arr.astype(np.float64)

In [40]: float_arr
Out[40]: array([1., 2., 3., 4., 5.])

In [41]: float_arr.dtype
Out[41]: dtype('float64')
```

Data Types for ndarrays

In this example, integers were cast to floating point. If I cast some floating-point numbers to be of integer data type, the decimal part will be truncated:

```
In [42]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [43]: arr
Out[43]: array([ 3.7, -1.2, -2.6,  0.5, 12.9, 10.1])

In [44]: arr.astype(np.int32)
Out[44]: array([ 3, -1, -2,  0, 12, 10], dtype=int32)
```

If you have an array of strings representing numbers, you can use `astype` to convert them to numeric form:

```
In [45]: numeric_strings = np.array(["1.25", "-9.6", "42"], dtype=np.string_)

In [46]: numeric_strings.astype(float)
Out[46]: array([ 1.25, -9.6 , 42.  ])
```

Data Types for ndarrays

If casting were to fail for some reason (like a string that cannot be converted to `float64`), a `ValueError` will be raised. Before, I was a bit lazy and wrote `float` instead of `np.float64`; NumPy aliases the Python types to its own equivalent data types.

You can also use another array's `dtype` attribute:

```
In [47]: int_array = np.arange(10)

In [48]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)

In [49]: int_array.astype(calibers.dtype)
Out[49]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

There are shorthand type code strings you can also use to refer to a `dtype`:

```
In [50]: zeros_uint32 = np.zeros(8, dtype="u4")

In [51]: zeros_uint32
Out[51]: array([0, 0, 0, 0, 0, 0, 0, 0], dtype=uint32)
```

Arithmetic with NumPy Arrays

Arrays are important because they enable you to express batch operations on data without writing any `for` loops. NumPy users call this *vectorization*. Any arithmetic operations between equal-size arrays apply the operation element-wise:

```
In [52]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])

In [53]: arr
Out[53]:
array([[1., 2., 3.],
       [4., 5., 6.]])

In [54]: arr * arr
Out[54]:
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])

In [55]: arr - arr
Out[55]:
array([[0., 0., 0.],
       [0., 0., 0.]])
```

Arithmetic with NumPy Arrays

Arithmetic operations with scalars propagate the scalar argument to each element in the array:

```
In [56]: 1 / arr
Out[56]:
array([[1.    , 0.5   , 0.3333],
       [0.25  , 0.2   , 0.1667]])

In [57]: arr ** 2
Out[57]:
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])
```

Comparisons between arrays of the same size yield Boolean arrays:

```
In [58]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])

In [59]: arr2
Out[59]:
array([[ 0.,  4.,  1.],
       [ 7.,  2., 12.]])

In [60]: arr2 > arr
Out[60]:
array([[False,  True, False],
       [ True, False,  True]])
```

Numpy: Basic Indexing and Slicing

NumPy array indexing is a deep topic, as there are many ways you may want to select a subset of your data or individual elements. One-dimensional arrays are simple; on the surface they act similarly to Python lists:

```
In [61]: arr = np.arange(10)

In [62]: arr
Out[62]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [63]: arr[5]
Out[63]: 5

In [64]: arr[5:8]
Out[64]: array([5, 6, 7])

In [65]: arr[5:8] = 12

In [66]: arr
Out[66]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

As you can see, if you assign a scalar value to a slice, as in `arr[5:8] = 12`, the value is propagated (or *broadcast* henceforth) to the entire selection.

Numpy: Basic Indexing and Slicing

With higher dimensional arrays, you have many more options. In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

```
In [73]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [74]: arr2d[2]
Out[74]: array([7, 8, 9])
```

Thus, individual elements can be accessed recursively. But that is a bit too much work, so you can pass a comma-separated list of indices to select individual elements. So these are equivalent:

```
In [75]: arr2d[0][2]
Out[75]: 3

In [76]: arr2d[0, 2]
Out[76]: 3
```

Numpy: Basic Indexing and Slicing

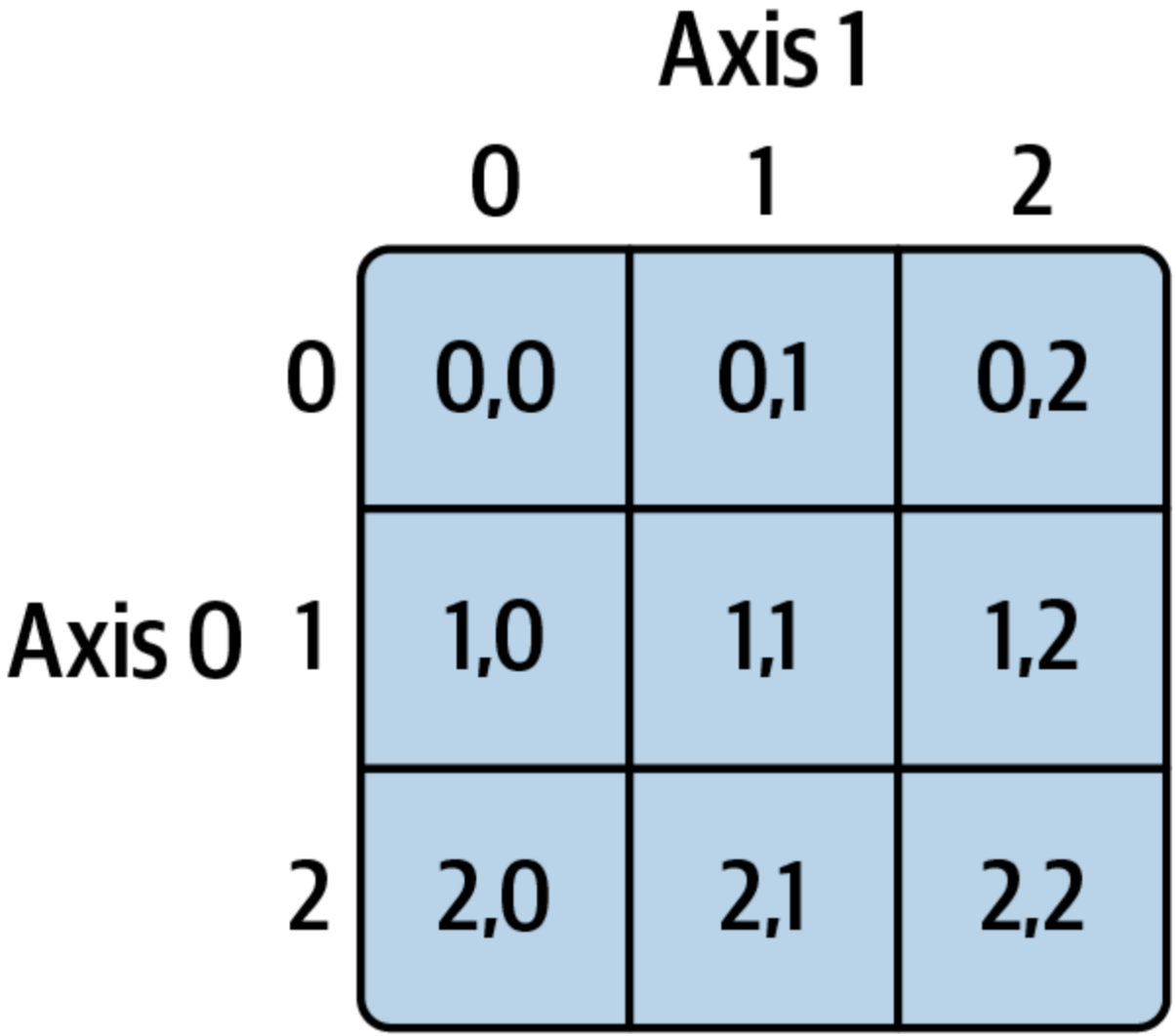


Figure 4.1: Indexing elements in a NumPy array

Numpy: Basic Indexing and Slicing

In multidimensional arrays, if you omit later indices, the returned object will be a lower dimensional ndarray consisting of all the data along the higher dimensions. So in the $2 \times 2 \times 3$ array `arr3d`:

```
In [77]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

In [78]: arr3d
Out[78]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

`arr3d[0]` is a 2×3 array:

```
In [79]: arr3d[0]
Out[79]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Numpy: Basic Indexing and Slicing

Both scalar values and arrays can be assigned to `arr3d[0]`:

```
In [80]: old_values = arr3d[0].copy()

In [81]: arr3d[0] = 42

In [82]: arr3d
Out[82]:
array([[[42, 42, 42],
        [42, 42, 42]],
       [[ 7,  8,  9],
        [10, 11, 12]]])

In [83]: arr3d[0] = old_values

In [84]: arr3d
Out[84]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

Numpy: Basic Indexing and Slicing

Similarly, `arr3d[1, 0]` gives you all of the values whose indices start with `(1, 0)`, forming a one-dimensional array:

```
In [85]: arr3d[1, 0]
Out[85]: array([7, 8, 9])
```

This expression is the same as though we had indexed in two steps:

```
In [86]: x = arr3d[1]

In [87]: x
Out[87]:
array([[ 7,  8,  9],
       [10, 11, 12]])

In [88]: x[0]
Out[88]: array([7, 8, 9])
```

Note that in all of these cases where subsections of the array have been selected, the returned arrays are views.

Numpy: Basic Indexing and Slicing

Indexing with slices

Like one-dimensional objects such as Python lists, ndarrays can be sliced with the familiar syntax:

```
In [89]: arr
Out[89]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])

In [90]: arr[1:6]
Out[90]: array([ 1,  2,  3,  4, 64])
```

Consider the two-dimensional array from before, `arr2d`. Slicing this array is a bit different:

```
In [91]: arr2d
Out[91]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [92]: arr2d[:2]
Out[92]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Numpy: Basic Indexing and Slicing

As you can see, it has sliced along axis 0, the first axis. A slice, therefore, selects a range of elements along an axis. It can be helpful to read the expression `arr2d[:2]` as "select the first two rows of `arr2d`."

You can pass multiple slices just like you can pass multiple indexes:

```
In [93]: arr2d[:2, 1:]
Out[93]:
array([[2, 3],
       [5, 6]])
```

When slicing like this, you always obtain array views of the same number of dimensions. By mixing integer indexes and slices, you get lower dimensional slices.

For example, I can select the second row but only the first two columns, like so:

```
In [94]: lower_dim_slice = arr2d[1, :2]
```

Numpy: Basic Indexing and Slicing

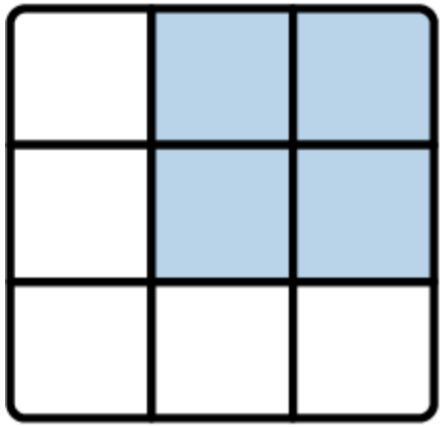
```
In [97]: arr2d[:, :1]
Out[97]:
array([[1],
       [4],
       [7]])
```

Of course, assigning to a slice expression assigns to the whole selection:

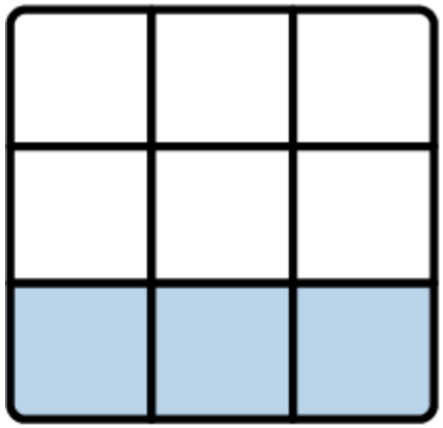
```
In [98]: arr2d[:2, 1:] = 0

In [99]: arr2d
Out[99]:
array([[1, 0, 0],
       [4, 0, 0],
       [7, 8, 9]])
```

Numpy: Basic Indexing and Slicing

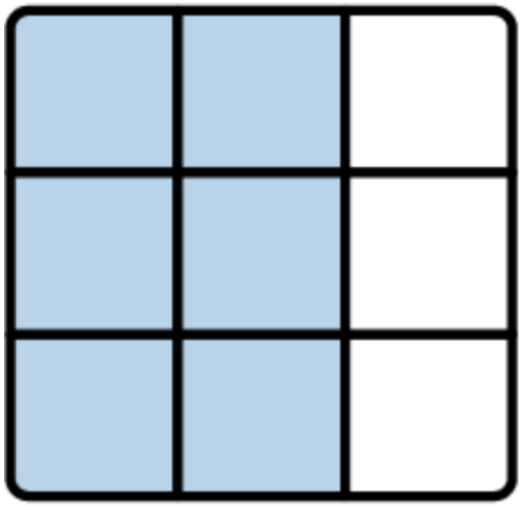


Expression	Shape
`arr[:2,1:]`	(2,2)

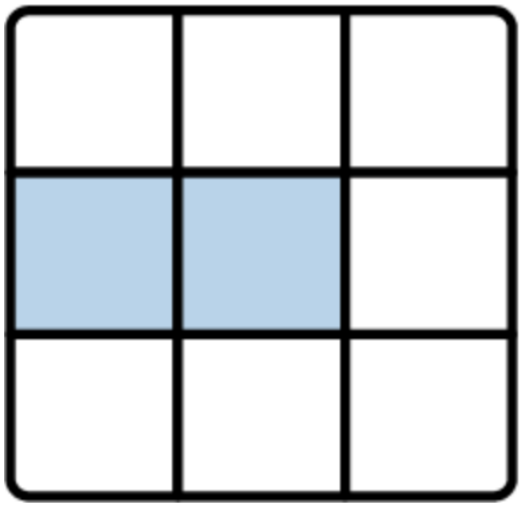


Expression	Shape
`arr[2]`	(3,)
`arr[2, :]`	(3,)
`arr[2:, :]`	(1,3)

Numpy: Basic Indexing and Slicing



```
arr[:, :2]        (3,2)
```



```
arr[1, :2]        (2,)
arr[1:2, :2]      (1,2)
```

Numpy: Basic Indexing and Slicing

Boolean Indexing

Let's consider an example where we have some data in an array and an array of names with duplicates:

```
In [100]: names = np.array(["Bob", "Joe", "Will", "Bob", "Will", "Joe", "Joe"])

In [101]: data = np.array([[4, 7], [0, 2], [-5, 6], [0, 0], [1, 2],
   .....:                  [-12, -4], [3, 4]])

In [102]: names
Out[102]: array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')

In [103]: data
Out[103]:
array([[  4,   7],
       [  0,   2],
       [ -5,   6],
       [  0,   0],
       [  1,   2],
       [-12,  -4],
       [  3,   4]])
```

Numpy: Basic Indexing and Slicing

Suppose each name corresponds to a row in the `data` array and we wanted to select all the rows with the corresponding name `"Bob"`. Like arithmetic operations, comparisons (such as `==`) with arrays are also vectorized. Thus, comparing `names` with the string `"Bob"` yields a Boolean array:

```
In [104]: names == "Bob"
Out[104]: array([ True, False, False,  True, False, False, False])
```

This Boolean array can be passed when indexing the array:

```
In [105]: data[names == "Bob"]
Out[105]:
array([[4, 7],
       [0, 0]])
```

The Boolean array must be of the same length as the array axis it's indexing. You can even mix and match Boolean arrays with slices or integers (or sequences of integers; more on this later).

Numpy: Basic Indexing and Slicing

To select everything but `"Bob"` you can either use `!=` or negate the condition using `~`:

```
In [108]: names != "Bob"
Out[108]: array([False,  True,  True, False,  True,  True,  True])

In [109]: ~(names == "Bob")
Out[109]: array([False,  True,  True, False,  True,  True,  True])

In [110]: data[~(names == "Bob")]
Out[110]:
array([[  0,   2],
       [ -5,   6],
       [  1,   2],
       [-12,  -4],
       [  3,   4]])
```

Numpy: Basic Indexing and Slicing

To select two of the three names to combine multiple Boolean conditions, use Boolean arithmetic operators like `&` (and) and `|` (or):

```
In [113]: mask = (names == "Bob") | (names == "Will")

In [114]: mask
Out[114]: array([ True, False,  True,  True,  True, False, False])

In [115]: data[mask]
Out[115]:
array([[ 4,  7],
       [-5,  6],
       [ 0,  0],
       [ 1,  2]])
```

Selecting data from an array by Boolean indexing and assigning the result to a new variable *always* creates a copy of the data, even if the returned array is unchanged.

Numpy: Basic Indexing and Slicing

Setting values with Boolean arrays works by substituting the value or values on the righthand side into the locations where the Boolean array's values are `True`. To set all of the negative values in `data` to 0, we need only do:

```
In [116]: data[data < 0] = 0

In [117]: data
Out[117]:
array([[4, 7],
       [0, 2],
       [0, 6],
       [0, 0],
       [1, 2],
       [0, 0],
       [3, 4]])
```

Numpy: Basic Indexing and Slicing

Fancy Indexing

Fancy indexing is a term adopted by NumPy to describe indexing using integer arrays. Suppose we had an 8×4 array:

```
In [120]: arr = np.zeros((8, 4))

In [121]: for i in range(8):
   .....:     arr[i] = i

In [122]: arr
Out[122]:
array([[0., 0., 0., 0.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [3., 3., 3., 3.],
       [4., 4., 4., 4.],
       [5., 5., 5., 5.],
       [6., 6., 6., 6.],
       [7., 7., 7., 7.]])
```

Numpy: Basic Indexing and Slicing

To select a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order:

```
In [123]: arr[[4, 3, 0, 6]]
Out[123]:
array([[4., 4., 4., 4.],
       [3., 3., 3., 3.],
       [0., 0., 0., 0.],
       [6., 6., 6., 6.]])
```

Hopefully this code did what you expected! Using negative indices selects rows from the end:

```
In [124]: arr[[-3, -5, -7]]
Out[124]:
array([[5., 5., 5., 5.],
       [3., 3., 3., 3.],
       [1., 1., 1., 1.]])
```

Numpy: Basic Indexing and Slicing

Passing multiple index arrays does something slightly different; it selects a one-dimensional array of elements corresponding to each tuple of indices:

```
In [125]: arr = np.arange(32).reshape((8, 4))

In [126]: arr
Out[126]:
array([[ 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]])

In [127]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[127]: array([ 4, 23, 29, 10])
```

Numpy: Basic Indexing and Slicing

Here the elements `(1, 0), (5, 3), (7, 1)`, and `(2, 2)` were selected. The result of fancy indexing with as many integer arrays as there are axes is always one-dimensional.

The behavior of fancy indexing in this case is a bit different from what some users might have expected (myself included), which is the rectangular region formed by selecting a subset of the matrix's rows and columns. Here is one way to get that:

```
In [128]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[128]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Numpy: Basic Indexing and Slicing

Keep in mind that fancy indexing, unlike slicing, always copies the data into a new array when assigning the result to a new variable. If you assign values with fancy indexing, the indexed values will be modified:

```
In [129]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[129]: array([ 4, 23, 29, 10])

In [130]: arr[[1, 5, 7, 2], [0, 3, 1, 2]] = 0

In [131]: arr
Out[131]:
array([[ 0,  1,  2,  3],
       [ 0,  5,  6,  7],
       [ 8,  9,  0, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22,  0],
       [24, 25, 26, 27],
       [28,  0, 30, 31]])
```

Transposing Arrays and Swapping Axes

Transposing is a special form of reshaping that similarly returns a view on the underlying data without copying anything. Arrays have the `transpose` method and the special `T` attribute:

```
In [132]: arr = np.arange(15).reshape((3, 5))

In [133]: arr
Out[133]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

In [134]: arr.T
Out[134]:
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

Transposing Arrays and Swapping Axes

When doing matrix computations, you may do this very often—for example, when computing the inner matrix product using `numpy.dot`:

```
In [135]: arr = np.array([[0, 1, 0], [1, 2, -2], [6, 3, 2], [-1, 0, -1], [1, 0, 1
]])

In [136]: arr
Out[136]:
array([[ 0,  1,  0],
       [ 1,  2, -2],
       [ 6,  3,  2],
       [-1,  0, -1],
       [ 1,  0,  1]])

In [137]: np.dot(arr.T, arr)
Out[137]:
array([[39, 20, 12],
       [20, 14,  2],
       [12,  2, 10]])
```

Transposing Arrays and Swapping Axes

Simple transposing with `.T` is a special case of swapping axes. ndarray has the method `swapaxes`, which takes a pair of axis numbers and switches the indicated axes to rearrange the data:

```
In [139]: arr
Out[139]:
array([[ 0,  1,  0],
       [ 1,  2, -2],
       [ 6,  3,  2],
       [-1,  0, -1],
       [ 1,  0,  1]])

In [140]: arr.swapaxes(0, 1)
Out[140]:
array([[ 0,  1,  6, -1,  1],
       [ 1,  2,  3,  0,  0],
       [ 0, -2,  2, -1,  1]])
```

`swapaxes` similarly returns a view on the data without making a copy.

Pseudorandom Number Generation

The `numpy.random` module supplements the built-in Python `random` module with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions. For example, you can get a 4 × 4 array of samples from the standard normal distribution using `numpy.random.standard_normal`:

```
In [141]: samples = np.random.standard_normal(size=(4, 4))

In [142]: samples
Out[142]:
array([[-0.2047,  0.4789, -0.5194, -0.5557],
       [ 1.9658,  1.3934,  0.0929,  0.2817],
       [ 0.769 ,  1.2464,  1.0072, -1.2962],
       [ 0.275 ,  0.2289,  1.3529,  0.8864]])
```

Pseudorandom Number Generation

Python's built-in `random` module, by contrast, samples only one value at a time. As you can see from this benchmark, `numpy.random` is well over an order of magnitude faster for generating very large samples:

```
In [143]: from random import normalvariate

In [144]: N = 1_000_000

In [145]: %timeit samples = [normalvariate(0, 1) for _ in range(N)]
490 ms +- 2.23 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)

In [146]: %timeit np.random.standard_normal(N)
32.6 ms +- 271 us per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

Pseudorandom Number Generation

These random numbers are not truly random (rather, *pseudorandom*) but instead are generated by a configurable random number generator that determines deterministically what values are created. Functions like `numpy.random.standard_normal` use the `numpy.random` module's default random number generator, but your code can be configured to use an explicit generator:

```
In [147]: rng = np.random.default_rng(seed=12345)

In [148]: data = rng.standard_normal((2, 3))
```

The `seed` argument is what determines the initial state of the generator, and the state changes each time the `rng` object is used to generate data. The generator object `rng` is also isolated from other code which might use the `numpy.random` module:

```
In [149]: type(rng)
Out[149]: numpy.random._generator.Generator
```

Pseudorandom Number Generation

Method	Description
`permutation`	Return a random permutation of a sequence, or return a permuted range
`shuffle`	Randomly permute a sequence in place
`uniform`	Draw samples from a uniform distribution
`integers`	Draw random integers from a given low-to-high range
`standard_normal`	Draw samples from a normal distribution with mean 0 and standard deviation 1
`binomial`	Draw samples from a binomial distribution
`normal`	Draw samples from a normal (Gaussian) distribution
`beta`	Draw samples from a beta distribution
`chisquare`	Draw samples from a chi-square distribution
`gamma`	Draw samples from a gamma distribution
`uniform`	Draw samples from a uniform [0, 1) distribution

Universal Functions: Fast Element-Wise Array Functions

A universal function, or *ufunc*, is a function that performs element-wise operations on data in ndarrays. You can think of them as fast vectorized wrappers for simple functions that take one or more scalar values and produce one or more scalar results.

Many ufuncs are simple element-wise transformations, like `numpy.sqrt` or `numpy.exp`:

```
In [150]: arr = np.arange(10)

In [151]: arr
Out[151]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [152]: np.sqrt(arr)
Out[152]:
array([0.    , 1.    , 1.4142, 1.7321, 2.    , 2.2361, 2.4495, 2.6458,
       2.8284, 3.    ])

In [153]: np.exp(arr)
Out[153]:
array([   1.    ,    2.7183,    7.3891,   20.0855,   54.5982,  148.4132,
        403.4288, 1096.6332, 2980.958 , 8103.0839])
```

Universal Functions: Fast Element-Wise Array Functions

These are referred to as *unary* ufuncs. Others, such as `numpy.add` or `numpy.maximum`, take two arrays (thus, *binary* ufuncs) and return a single array as the result:

```
In [154]: x = rng.standard_normal(8)

In [155]: y = rng.standard_normal(8)

In [156]: x
Out[156]:
array([-1.3678,  0.6489,  0.3611, -1.9529,  2.3474,  0.9685, -0.7594,
        0.9022])

In [157]: y
Out[157]:
array([-0.467 , -0.0607,  0.7888, -1.2567,  0.5759,  1.399 ,  1.3223,
       -0.2997])

In [158]: np.maximum(x, y)
Out[158]:
array([-0.467 ,  0.6489,  0.7888, -1.2567,  2.3474,  1.399 ,  1.3223,
        0.9022])
```

Universal Functions: Fast Element-Wise Array Functions

In this example, `numpy.maximum` computed the element-wise maximum of the elements in `x` and `y`.

While not common, a ufunc can return multiple arrays. `numpy.modf` is one example: a vectorized version of the built-in Python `math.modf`, it returns the fractional and integral parts of a floating-point array:

```
In [159]: arr = rng.standard_normal(7) * 5

In [160]: arr
Out[160]: array([ 4.5146, -8.1079, -0.7909,  2.2474, -6.718 , -0.4084,  8.6237])

In [161]: remainder, whole_part = np.modf(arr)

In [162]: remainder
Out[162]: array([ 0.5146, -0.1079, -0.7909,  0.2474, -0.718 , -0.4084,  0.6237])

In [163]: whole_part
Out[163]: array([ 4., -8., -0.,  2., -6., -0.,  8.])
```

Universal Functions: Fast Element-Wise Array Functions

Ufuncs accept an optional `out` argument that allows them to assign their results into an existing array rather than create a new one:

```
In [164]: arr
Out[164]: array([ 4.5146, -8.1079, -0.7909,  2.2474, -6.718 , -0.4084,  8.6237])

In [165]: out = np.zeros_like(arr)

In [166]: np.add(arr, 1)
Out[166]: array([ 5.5146, -7.1079,  0.2091,  3.2474, -5.718 ,  0.5916,  9.6237])

In [167]: np.add(arr, 1, out=out)
Out[167]: array([ 5.5146, -7.1079,  0.2091,  3.2474, -5.718 ,  0.5916,  9.6237])

In [168]: out
Out[168]: array([ 5.5146, -7.1079,  0.2091,  3.2474, -5.718 ,  0.5916,  9.6237])
```

Expressing Conditional Logic as Array Operations

The `numpy.where` function is a vectorized version of the ternary expression `x if condition else y`. Suppose we had a Boolean array and two arrays of values:

```
In [180]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])

In [181]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

In [182]: cond = np.array([True, False, True, True, False])
```

Suppose we wanted to take a value from `xarr` whenever the corresponding value in `cond` is `True`, and otherwise take the value from `yarr`. A list comprehension doing this might look like:

```
In [183]: result = [(x if c else y)
   .....:           for x, y, c in zip(xarr, yarr, cond)]

In [184]: result
Out[184]: [1.1, 2.2, 1.3, 1.4, 2.5]
```

Expressing Conditional Logic as Array Operations

This has multiple problems. First, it will not be very fast for large arrays (because all the work is being done in interpreted Python code). Second, it will not work with multidimensional arrays. With `numpy.where` you can do this with a single function call:

```
In [185]: result = np.where(cond, xarr, yarr)

In [186]: result
Out[186]: array([1.1, 2.2, 1.3, 1.4, 2.5])
```

The second and third arguments to `numpy.where` don't need to be arrays; one or both of them can be scalars. A typical use of `where` in data analysis is to produce a new array of values based on another array. Suppose you had a matrix of randomly generated data and you wanted to replace all positive values with 2 and all negative values with –2. This is possible to do with `numpy.where`:

Expressing Conditional Logic as Array Operations

```
In [187]: arr = rng.standard_normal((4, 4))

In [188]: arr
Out[188]:
array([[ 2.6182,  0.7774,  0.8286, -0.959 ],
       [-1.2094, -1.4123,  0.5415,  0.7519],
       [-0.6588, -1.2287,  0.2576,  0.3129],
       [-0.1308,  1.27  , -0.093 , -0.0662]])

In [189]: arr > 0
Out[189]:
array([[ True,  True,  True, False],
       [False, False,  True,  True],
       [False, False,  True,  True],
       [False,  True, False, False]])

In [190]: np.where(arr > 0, 2, -2)
Out[190]:
array([[ 2,  2,  2, -2],
       [-2, -2,  2,  2],
       [-2, -2,  2,  2],
       [-2,  2, -2, -2]])
```

Mathematical and Statistical Methods

```
In [192]: arr = rng.standard_normal((5, 4))

In [193]: arr
Out[193]:
array([[-1.1082,  0.136 ,  1.3471,  0.0611],
       [ 0.0709,  0.4337,  0.2775,  0.5303],
       [ 0.5367,  0.6184, -0.795 ,  0.3   ],
       [-1.6027,  0.2668, -1.2616, -0.0713],
       [ 0.474 , -0.4149,  0.0977, -1.6404]])

In [194]: arr.mean()
Out[194]: -0.087197444457434529

In [195]: np.mean(arr)
Out[195]: -0.087197444457434529

In [196]: arr.sum()
Out[196]: -1.743948891486906
```

Mathematical and Statistical Methods

Method	Description
`sum`	Sum of all the elements in the array or along an axis; zero-length arrays have sum 0
`mean`	Arithmetic mean; invalid (returns `NaN`) on zero-length arrays
`std, var`	Standard deviation and variance, respectively
`min, max`	Minimum and maximum
`argmin, argmax`	Indices of minimum and maximum elements, respectively
`cumsum`	Cumulative sum of elements starting from 0
`cumprod`	Cumulative product of elements starting from 1

File Input and Output with Arrays

`numpy.save` and `numpy.load` are the two workhorse functions for efficiently saving and loading array data on disk. Arrays are saved by default in an uncompressed raw binary format with file extension *.npy*:

```
In [231]: arr = np.arange(10)

In [232]: np.save("some_array", arr)
```

If the file path does not already end in *.npy*, the extension will be appended. The array on disk can then be loaded with `numpy.load`:

```
In [233]: np.load("some_array.npy")
Out[233]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

File Input and Output with Arrays

You can save multiple arrays in an uncompressed archive using `numpy.savez` and passing the arrays as keyword arguments:

```
In [234]: np.savez("array_archive.npz", a=arr, b=arr)
```

When loading an *.npz* file, you get back a dictionary-like object that loads the individual arrays lazily:

```
In [235]: arch = np.load("array_archive.npz")

In [236]: arch["b"]
Out[236]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

If your data compresses well, you may wish to use `numpy.savez_compressed` instead:

```
In [237]: np.savez_compressed("arrays_compressed.npz", a=arr, b=arr)
```

SEAS 6414 – Python Applications in Data Analytics

THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

Dr. Adewale Akinfaderin

Lecture 3 - Spring 2024