NUMPY BASICS

Arithmetic with NumPy Arrays

- **vectorization**：express batch operations on data without writing any for loops

```
In [51]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])

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

In [53]: arr * arr
Out[53]:
array([[  1.,   4.,   9.],
       [ 16.,  25.,  36.]])

In [54]: arr - arr
Out[54]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Arithmetic with NumPy Arrays

- Arithmetic operations with scalars :

```
In [55]: 1 / arr
Out[55]:
array([[ 1.    ,  0.5   ,  0.3333],
       [ 0.25  ,  0.2   ,  0.1667]])

In [56]: arr ** 0.5
Out[56]:
array([[ 1.    ,  1.4142,  1.7321],
       [ 2.    ,  2.2361,  2.4495]])
```

Arithmetic with NumPy Arrays

- Comparisons between arrays of the same size：

```
In [57]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])

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

In [59]: arr2 > arr
Out[59]:
array([[False,  True, False],
       [ True, False,  True]], dtype=bool)
```

Basic Indexing and Slicing

- One-dimensional arrays are simple:

```
In [60]: arr = np.arange(10)

In [61]: arr
Out[61]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [62]: arr[5]
Out[62]: 5

In [63]: arr[5:8]
Out[63]: array([5, 6, 7])
```

Basic Indexing and Slicing

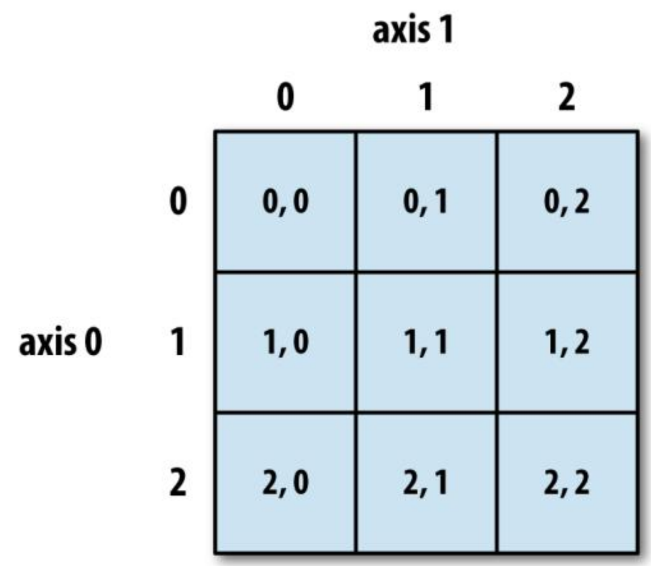
- Any modifications will be reflected in the source array:

```
In [64]: arr[5:8] = 12

In [65]: arr
Out[65]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
In [66]: arr_slice = arr[5:8]

In [67]: arr_slice
Out[67]: array([12, 12, 12])
In [68]: arr_slice[1] = 12345

In [69]: arr
Out[69]: array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,
   9])
In [70]: arr_slice[:] = 64

In [71]: arr
Out[71]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

Basic Indexing and Slicing

- In a 2D array, the elements at each index are one-dimensional arrays:

```
In [72]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [73]: arr2d[2]
Out[73]: array([7, 8, 9])
In [74]: arr2d[0][2]
Out[74]: 3

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

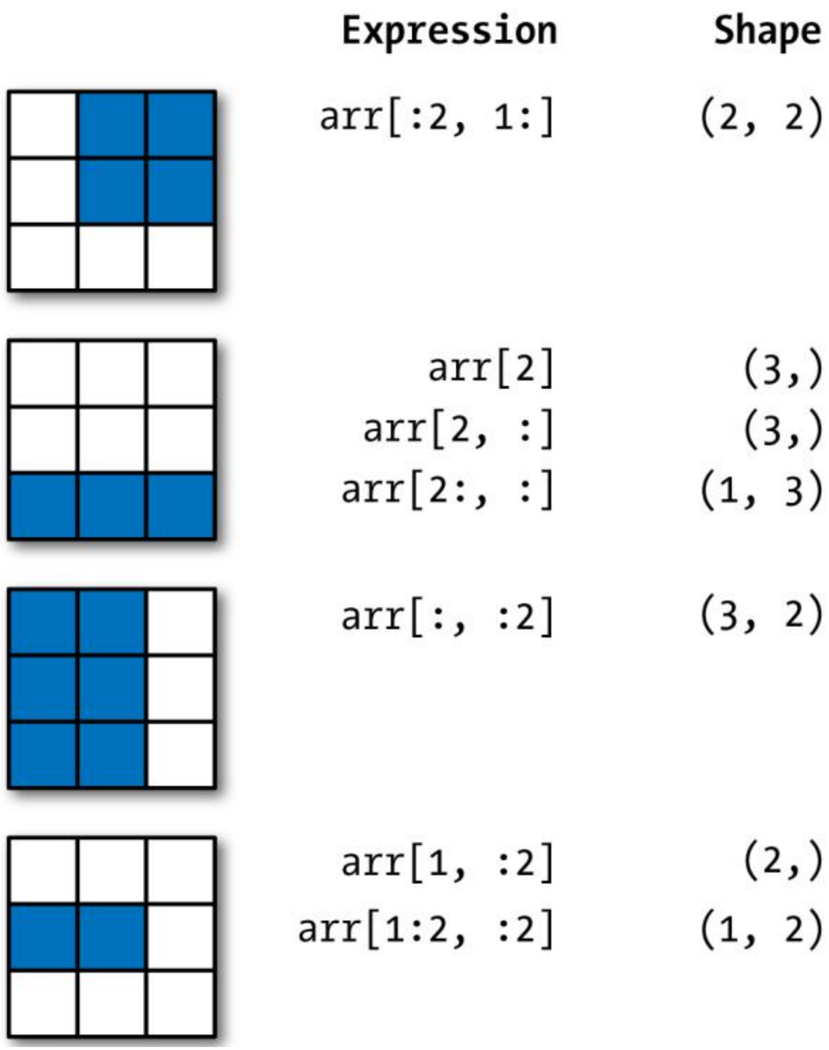


Basic Indexing and Slicing

- In the 2 × 2 × 3 array arr3d:

```
In [76]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

In [77]: arr3d
Out[77]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
In [78]: arr3d[0]
Out[78]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Indexing with slices



By mixing integer indexes and slices, you get lower dimensional slices.

Boolean Indexing

- Suppose each **name** corresponds to a **row** in the data array and we wanted to **select all the rows** with corresponding name `'Bob'`.

```
In [98]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [99]: data = np.random.randn(7, 4)

In [101]: data
Out[101]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

Boolean Indexing

```
In [102]: names == 'Bob'
Out[102]: array([ True, False, False,  True, False, False, False], dtype=bool)
In [103]: data[names == 'Bob']
Out[103]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.669 , -0.4386, -0.5397,  0.477 ]])
```

- *You can mix and match boolean arrays with slices or integers :*

```
In [104]: data[names == 'Bob', 2:]
Out[104]:
array([[ 0.769 ,  1.2464],
       [-0.5397,  0.477 ]])

In [105]: data[names == 'Bob', 3]
Out[105]: array([ 1.2464,  0.477 ])
```

Boolean Indexing

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

```
In [107]: data[~(names == 'Bob')]
Out[107]:
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

- Use boolean operators like `& (and)` and `| (or)`.
- Selecting data from an array by **boolean indexing** always creates **a copy** of the data, even if the returned array is unchanged.

Boolean Indexing

- To set all of the **negative values** in data to **0** we need only do:

```
In [113]: data[data < 0] = 0

In [114]: data
Out[114]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.0072,  0.    ,  0.275 ,  0.2289],
       [ 1.3529,  0.8864,  0.    ,  0.    ],
       [ 1.669 ,  0.    ,  0.    ,  0.477 ],
       [ 3.2489,  0.    ,  0.    ,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [ 0.    ,  0.    ,  0.    ,  0.    ]])
```

Fancy Indexing

- ***Fancy Indexing***:index using integer arrays.

```
In [117]: arr = np.empty((8, 4))

In [118]: for i in range(8):
   .....:     arr[i] = i

In [119]: arr
Out[119]:
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.]])
```

Fancy Indexing

- ***Fancy Indexing***:index using integer arrays.

```
In [117]: arr = np.empty((8, 4))

In [118]: for i in range(8):
   .....:     arr[i] = i

In [119]: arr
Out[119]:
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.]])
```

Fancy Indexing

- you can simply pass a list or ndarray of integers specifying the desired order:

```
In [120]: arr[[4, 3, 0, 6]]
Out[120]:
array([[ 4.,  4.,  4.,  4.],
       [ 3.,  3.,  3.,  3.],
       [ 0.,  0.,  0.,  0.],
       [ 6.,  6.,  6.,  6.]])
```

- Using negative indices selects rows from the end:

```
In [121]: arr[[-3, -5, -7]]
Out[121]:
array([[ 5.,  5.,  5.,  5.],
       [ 3.,  3.,  3.,  3.],
       [ 1.,  1.,  1.,  1.]])
```

Fancy Indexing

- Passing **multiple index arrays** does something slightly different:

```
In [122]: arr = np.arange(32).reshape((8, 4))

In [123]: arr
Out[123]:
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 [124]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[124]: array([ 4, 23, 29, 10])
```

Caution:list indices!

Transposing Arrays and Swapping Axes

- **Arrays** have the special **T** attribute.
- When doing **matrix computations**, you may do this very often.

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

In [128]: arr.T
Out[128]:
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

Transposing Arrays and Swapping Axes

- Suppose

$$a^T = (a_1, \ a_2), \quad b = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

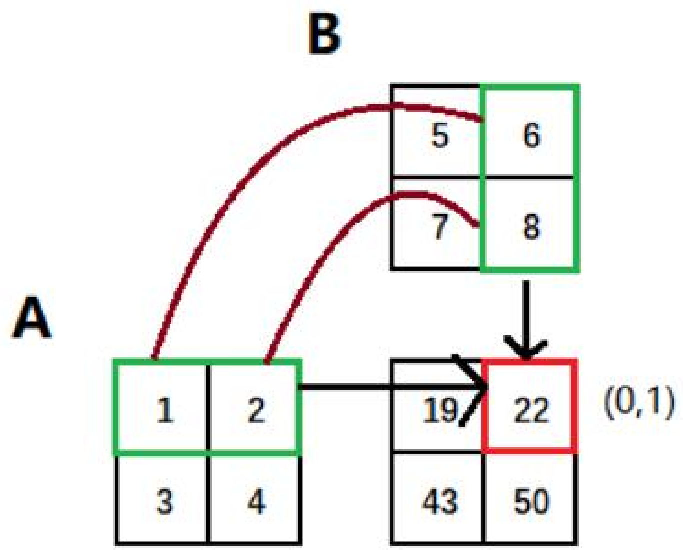
then

$$a^T \cdot b = a_1 b_1 + a_2 b_2$$

$$b \otimes a^T = \begin{pmatrix} b_1 a_1 & b_1 a_2 \\ b_2 a_1 & b_2 a_2 \end{pmatrix}$$

Transposing Arrays and Swapping Axes

- Computing the inner matrix product using np.dot:



Transposing Arrays and Swapping Axes

```
array([[-0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329],
       [-2.3594, -0.1995, -1.542 ],
       [-0.9707, -1.307 ,  0.2863],
       [ 0.378 , -0.7539,  0.3313],
       [ 1.3497,  0.0699,  0.2467]])

In [131]: np.dot(arr.T, arr)
Out[131]:
array([[ 9.2291,  0.9394,  4.948 ],
       [ 0.9394,  3.7662, -1.3622],
       [ 4.948 , -1.3622,  4.3437]])
```

Transposing Arrays and Swapping Axes

- About ***transpose()***:

```
Out[11]: array([[0, 1],
                [2, 3]])

In  [12]: import numpy as np
          x.transpose(0,1)

Out[12]: array([[0, 1],
                [2, 3]])

In  [13]: x.transpose(1,0)

Out[13]: array([[0, 2],
                [1, 3]])
```

Transposing Arrays and Swapping Axes

- For **higher dimensional** arrays,:

```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [134]: arr.transpose((1, 0, 2))
Out[134]:
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],
       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

Transposing Arrays and Swapping Axes

- The method `swapaxes`, returns a view on the data without making a copy.

```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [136]: arr.swapaxes(1, 2)
Out[136]:
array([[[ 0,  4],
        [ 1,  5],
        [ 2,  6],
        [ 3,  7]],
       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])
```

$$\begin{bmatrix} 000, 001, 002, 003 \\ 010, 011, 012, 013 \\ 100, 101, 102, 103 \\ 110, 111, 112, 113 \end{bmatrix}$$

?

Universal Functions: Fast Element-Wise ArrayFunctions

- `ufunc`: **fast vectorized wrappers** for simple functions that take one or more **scalar values** and produce one or more scalar results.
  - `sqrt` or `exp`:

```
Out[138]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [139]: np.sqrt(arr)
Out[139]:
array([ 0.    ,  1.    ,  1.4142,  1.7321,  2.    ,  2.2361,  2.4495,
        2.6458,  2.8284,  3.    ])

In [140]: np.exp(arr)
Out[140]:
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 ArrayFunctions

- `add` or `maximum`:

```
In [143]: x
Out[143]:
array([-0.0119,  1.0048,  1.3272, -0.9193, -1.5491,  0.0222,  0.7584,
       -0.6605])

In [144]: y
Out[144]:
array([ 0.8626, -0.01  ,  0.05  ,  0.6702,  0.853 , -0.9559, -0.0235,
       -2.3042])

In [145]: np.maximum(x, y)
Out[145]:
array([ 0.8626,  1.0048,  1.3272,  0.6702,  0.853 ,  0.0222,  0.7584,
       -0.6605])
```

Universal Functions: Fast Element-Wise ArrayFunctions

- arrays. modf:

```
In [147]: arr
Out[147]: array([-3.2623, -6.0915, -6.663 ,  5.3731,  3.6182,  3.45  ,  5.0077])

In [148]: remainder, whole_part = np.modf(arr)

In [149]: remainder
Out[149]: array([-0.2623, -0.0915, -0.663 ,  0.3731,  0.6182,  0.45  ,  0.0077])

In [150]: whole_part
Out[150]: array([-3., -6., -6.,  5.,  3.,  3.,  5.])
```

Universal Functions: Fast Element-Wise ArrayFunctions

Function	Description
`abs, fabs`	Compute the absolute value element-wise for integer, floating-point, or complex values
`sqrt`	Compute the square root of each element (equivalent to `arr ** 0.5`)
`square`	Compute the square of each element (equivalent to `arr ** 2`)
`exp`	Compute the exponent e^x of each element
`log, log10, log2, log1p`	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
`sign`	Compute the sign of each element: 1 (positive), 0 (zero), or –1 (negative)
`ceil`	Compute the ceiling of each element (i.e., the smallest integer greater than or equal to that number)
`floor`	Compute the floor of each element (i.e., the largest integer less than or equal to each element)
`rint`	Round elements to the nearest integer, preserving the `dtype`
`modf`	Return fractional and integral parts of array as a separate array
`isnan`	Return boolean array indicating whether each value is `NaN` (Not a Number)
`isfinite, isinf`	Return boolean array indicating whether each element is finite (non-`inf`, non-`NaN`) or infinite, respectively
`cos, cosh, sin, sinh, tan, tanh`	Regular and hyperbolic trigonometric functions
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Inverse trigonometric functions
`logical_not`	Compute truth value of `not x` element-wise (equivalent to `~arr`).

Universal Functions: Fast Element-Wise ArrayFunctions

Function	Description
add	Add corresponding elements in arrays
subtract	Subtract elements in second array from first array
multiply	Multiply array elements
divide, floor_divide	Divide or floor divide (truncating the remainder)
power	Raise elements in first array to powers indicated in second array
maximum, fmax	Element-wise maximum; fmax ignores NaN
minimum, fmin	Element-wise minimum; fmin ignores NaN
mod	Element-wise modulus (remainder of division)
copysign	Copy sign of values in second argument to values in first argument
greater, greater_equal, less, less_equal, equal, not_equal	Perform element-wise comparison, yielding boolean array (equivalent to infix operators >, >=, <, <=, ==, !=)
logical_and, logical_or, logical_xor	Compute element-wise truth value of logical operation (equivalent to infix operators & |, ^)

Array-Oriented Programming with Arrays

- Suppose we wished to evaluate the function `sqrt(x^2 + y^2)`

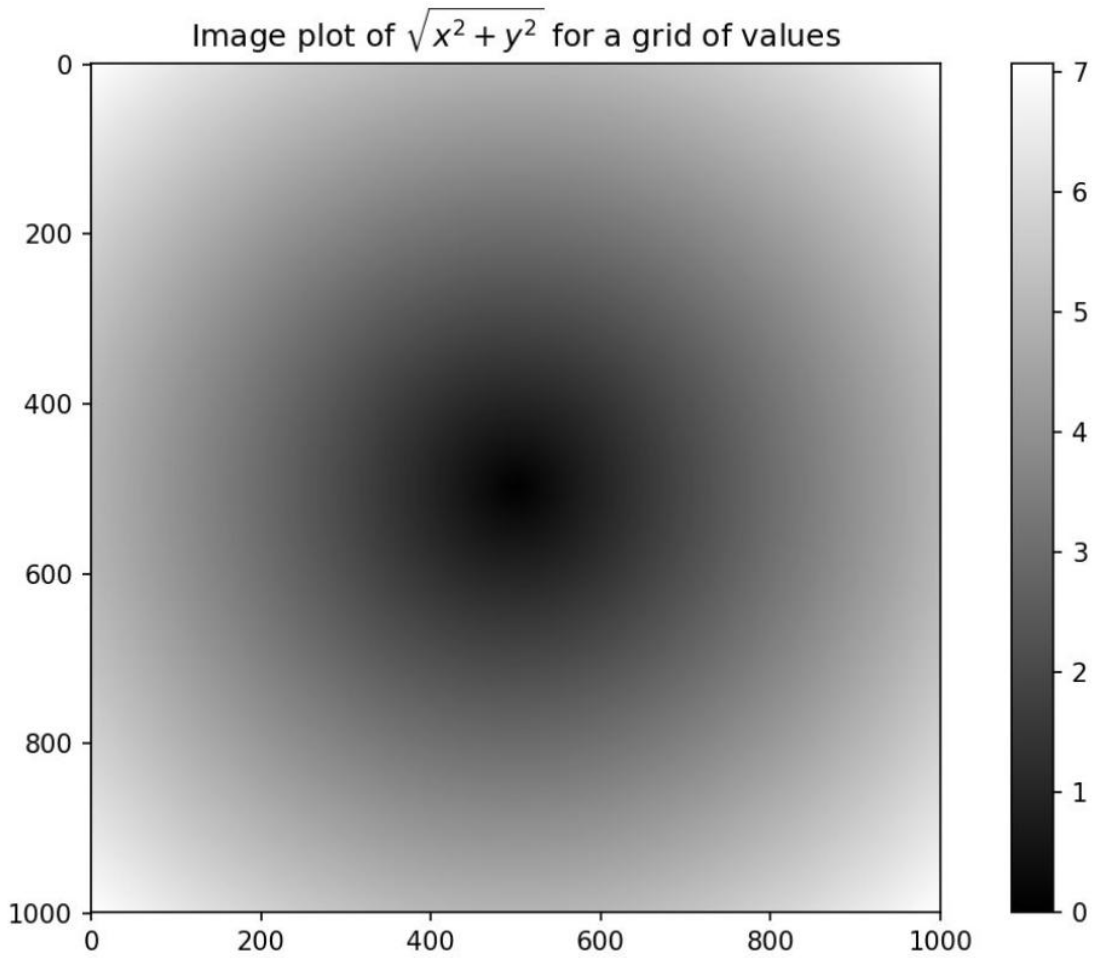
```
In [155]: points = np.arange(-5, 5, 0.01) # 1000 equally spaced points

In [156]: xs, ys = np.meshgrid(points, points)

In [157]: ys
Out[157]:
array([[-5.  , -5.  , -5.  , ..., -5.  , -5.  , -5.  ],
       [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
       [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [ 4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],
       [ 4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],
       [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
In [158]: z = np.sqrt(xs ** 2 + ys ** 2)
```

Array-Oriented Programming with Arrays

- About `np.meshgrid`

```
In [26]: a
Out[26]: array([-5, -4, -3, -2, -1,  0,  1,  2,  3,  4])

In [27]: b
Out[27]: array([7, 8, 9])

In [28]: xs,ys=np.meshgrid(a,b)

In [30]: xs
Out[30]: array([[-5, -4, -3, -2, -1,  0,  1,  2,  3,  4],
                [-5, -4, -3, -2, -1,  0,  1,  2,  3,  4],
                [-5, -4, -3, -2, -1,  0,  1,  2,  3,  4]])

In [31]: ys
Out[31]: array([[7, 7, 7, 7, 7, 7, 7, 7, 7, 7],
                [8, 8, 8, 8, 8, 8, 8, 8, 8, 8],
                [9, 9, 9, 9, 9, 9, 9, 9, 9, 9]])
```

Array-Oriented Programming with Arrays



Expressing Conditional Logic as Array Operations

- Suppose we wanted to take a value from `xarr` whenever the corresponding value in `cond` is `True`, otherwise take the value from `yarr`.

```
In [165]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])

In [166]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

In [167]: cond = np.array([True, False, True, True, False])
```

- With `np.where` you can write this very concisely:

```
In [170]: result = np.where(cond, xarr, yarr)
```

Expressing Conditional Logic as Array Operations

- A typical use of `where` in data analysis is to **produce a new array** of values based on another array.
- The **second and third arguments** to `np.where` can be **scalars**.
- Suppose：To a random matrix,you wanted to **replace** all positive values with 2 and all negative values with –2.

Expressing Conditional Logic as Array Operations

```
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 0.2229,  0.0513, -1.1577,  0.8167],
       [ 0.4336,  1.0107,  1.8249, -0.9975],
       [ 0.8506, -0.1316,  0.9124,  0.1882]])

In [174]: arr > 0
Out[174]:
array([[False, False, False, False],
       [ True,  True, False,  True],
       [ True,  True,  True, False],
       [ True, False,  True,  True]], dtype=bool)

In [175]: np.where(arr > 0, 2, -2)
Out[175]:
array([[-2, -2, -2, -2],
       [ 2,  2, -2,  2],
       [ 2,  2,  2, -2],
       [ 2, -2,  2,  2]])
```

Expressing Conditional Logic as Array Operations

- Also,I can replace all positive values in `arr` with the constant 2:

```
In [176]: np.where(arr > 0, 2, arr) # set only positive values to 2
Out[176]:
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 2.    ,  2.    , -1.1577,  2.    ],
       [ 2.    ,  2.    ,  2.    , -0.9975],
       [ 2.    , -0.1316,  2.    ,  2.    ]])
```

Mathematical and Statistical Methods

- Some NumPy functions like `sum`, `mean`, `cumsum`,`cumprod`,`std`, `any`,`all`,`etc`.

```
array([[ 2.1695, -0.1149,  2.0037,  0.0296],
       [ 0.7953,  0.1181, -0.7485,  0.585 ],
       [ 0.1527, -1.5657, -0.5625, -0.0327],
       [-0.929 , -0.4826, -0.0363,  1.0954],
       [ 0.9809, -0.5895,  1.5817, -0.5287]])

In [179]: arr.mean()
Out[179]: 0.19607051119998253

In [180]: np.mean(arr)
Out[180]: 0.19607051119998253

In [181]: arr.sum()
Out[181]: 3.9214102239996507
```

```
In [182]: arr.mean(axis=1)
Out[182]: array([ 1.022 ,  0.1875, -0.502 , -0.0881,  0.3611])

In [183]: arr.sum(axis=0)
Out[183]: array([ 3.1693, -2.6345,  2.2381,  1.1486])
```

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

In [188]: arr.cumsum(axis=0)
Out[188]:
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]])

In [189]: arr.cumprod(axis=1)
Out[189]:
array([[  0,   0,   0],
       [  3,  12,  60],
       [  6,  42, 336]])
```

Methods for Boolean Arrays

```
In [190]: arr = np.random.randn(100)

In [191]: (arr > 0).sum() # Number of positive values
Out[191]: 42

In [192]: bools = np.array([False, False, True, False])

In [193]: bools.any()
Out[193]: True

In [194]: bools.all()
Out[194]: False
```

Mathematical and Statistical Methods

- Basic array 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; zero-length arrays have `NaN` mean
`std, var`	Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator `n`)
`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

Sorting

- NumPy arrays can be sorted in-place with the `sort` method:

```
array([[ 0.6033,  1.2636, -0.2555],
       [-0.4457,  0.4684, -0.9616],
       [-1.8245,  0.6254,  1.0229],
       [ 1.1074,  0.0909, -0.3501],
       [ 0.218 , -0.8948, -1.7415]])

In [201]: arr.sort(1)

In [202]: arr
Out[202]:
array([[-0.2555,  0.6033,  1.2636],
       [-0.9616, -0.4457,  0.4684],
       [-1.8245,  0.6254,  1.0229],
       [-0.3501,  0.0909,  1.1074],
       [-1.7415, -0.8948,  0.218 ]])
```

Sorting

- Computing the **quantiles** of an array is to sort it and select the value at a particular rank:

```
In [203]: large_arr = np.random.randn(1000)

In [204]: large_arr.sort()

In [205]: large_arr[int(0.05 * len(large_arr))] # 5% quantile
Out[205]: -1.5311513550102103
```

Unique and Other Set Logic

- `np.unique`, returns the sorted unique values in an array:

```
In [206]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [207]: np.unique(names)
Out[207]:
array(['Bob', 'Joe', 'Will'],
      dtype='<U4')

In [208]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [209]: np.unique(ints)
Out[209]: array([1, 2, 3, 4])
```

Unique and Other Set Logic

- `np.in1d`, tests membership of the values in one array in another,returning a boolean array:

```
In [211]: values = np.array([6, 0, 0, 3, 2, 5, 6])

In [212]: np.in1d(values, [2, 3, 6])
Out[212]: array([ True, False, False,  True,  True, False,  True], dtype=bool)
```

File Input and Output with Arrays

- `np.save` and `np.load` can efficiently saving and loading array data on disk.
- Arrays are saved by default with file extension `.npy`:

```
In [213]: arr = np.arange(10)

In [214]: np.save('some_array', arr)
In [215]: np.load('some_array.npy')
Out[215]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

File Input and Output with Arrays

- `np.savez` , save multiple arrays in an uncompressed archive:

```
In [216]: np.savez('array_archive.npz', a=arr, b=arr)

In [217]: arch = np.load('array_archive.npz')

In [218]: arch['b']
Out[218]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

a dict-like object

Linear Algebra

- The function `dot` (or `@`) , is equivalent to `np.dot(x, y)`.

```
In [225]: x
Out[225]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

In [226]: y
Out[226]:
array([[  6.,  23.],
       [ -1.,   7.],
       [  8.,   9.]])

In [227]: x.dot(y)
Out[227]:
array([[  28.,   64.],
       [  67.,  181.]])
```

```
In [229]: np.dot(x, np.ones(3))
Out[229]: array([  6.,  15.])
```

Linear Algebra

The inverse of a square matrix **A**,

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I},$$

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}.$$

so：

$$A^{-1} = \frac{1}{|A|} A^{*}$$

Linear Algebra

- the determinant |A| :

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21}$$

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{21}a_{32}a_{13}$$

$$- a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} - a_{13}a_{22}a_{31}$$

Linear Algebra

- the determinant |A| :

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21}$$

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{21}a_{32}a_{13}$$

$$- a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} - a_{13}a_{22}a_{31}$$

Linear Algebra

- Adjoint matrix A^* :

$$\mathbf{A}^* = \begin{matrix} A_{11} & A_{21} & \dots\dots & A_{n1} \\ A_{12} & A_{22} & \dots\dots & A_{n2} \\ \dots & \dots & \dots & \dots \\ A_{1n} & A_{2n} & \dots\dots & A_{nn} \end{matrix}$$

for example

$$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 0 & -1 \\ 0 & 1 & 1 \end{pmatrix}^* = \begin{pmatrix} 1 & 1 & -2 \\ -1 & 1 & 4 \\ 1 & -1 & -2 \end{pmatrix}$$

Linear Algebra

Function	Description
diag	Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal
dot	Matrix multiplication
trace	Compute the sum of the diagonal elements
det	Compute the matrix determinant

Pseudorandom Number Generation

- The `numpy.random` module can efficiently generating whole arrays of sample values from many kinds of **probability distributions**.
- You can get a 4 × 4 array of samples from the **standard normal distribution** using normal.

```
In [238]: samples = np.random.normal(size=(4, 4))

In [239]: samples
Out[239]:
array([[ 0.5732,  0.1933,  0.4429,  1.2796],
       [ 0.575 ,  0.4339, -0.7658, -1.237 ],
       [-0.5367,  1.8545, -0.92  , -0.1082],
       [ 0.1525,  0.9435, -1.0953, -0.144 ]])
```

Pseudorandom Number Generation

- Python's built-in random module, by contrast, `numpy.random` is well over an order of magnitude faster for generating very large samples:

```
In [240]: from random import normalvariate

In [241]: N = 1000000

In [242]: %timeit samples = [normalvariate(0, 1) for _ in range(N)]
1.77 s +- 126 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)

In [243]: %timeit np.random.normal(size=N)
61.7 ms +- 1.32 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

Pseudorandom Number Generation

Partial list of numpy.random functions

Function	Description
`seed`	Seed the random number generator
`permutation`	Return a random permutation of a sequence, or return a permuted range
`shuffle`	Randomly permute a sequence in-place
`rand`	Draw samples from a uniform distribution
`randint`	Draw random integers from a given low-to-high range
`randn`	Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface)
`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

Example: Random Walks

- Consider a simple random walk starting at 0 with steps of **1** and **–1** occurring with **equal probability**.
  - Python using the built-in random module:

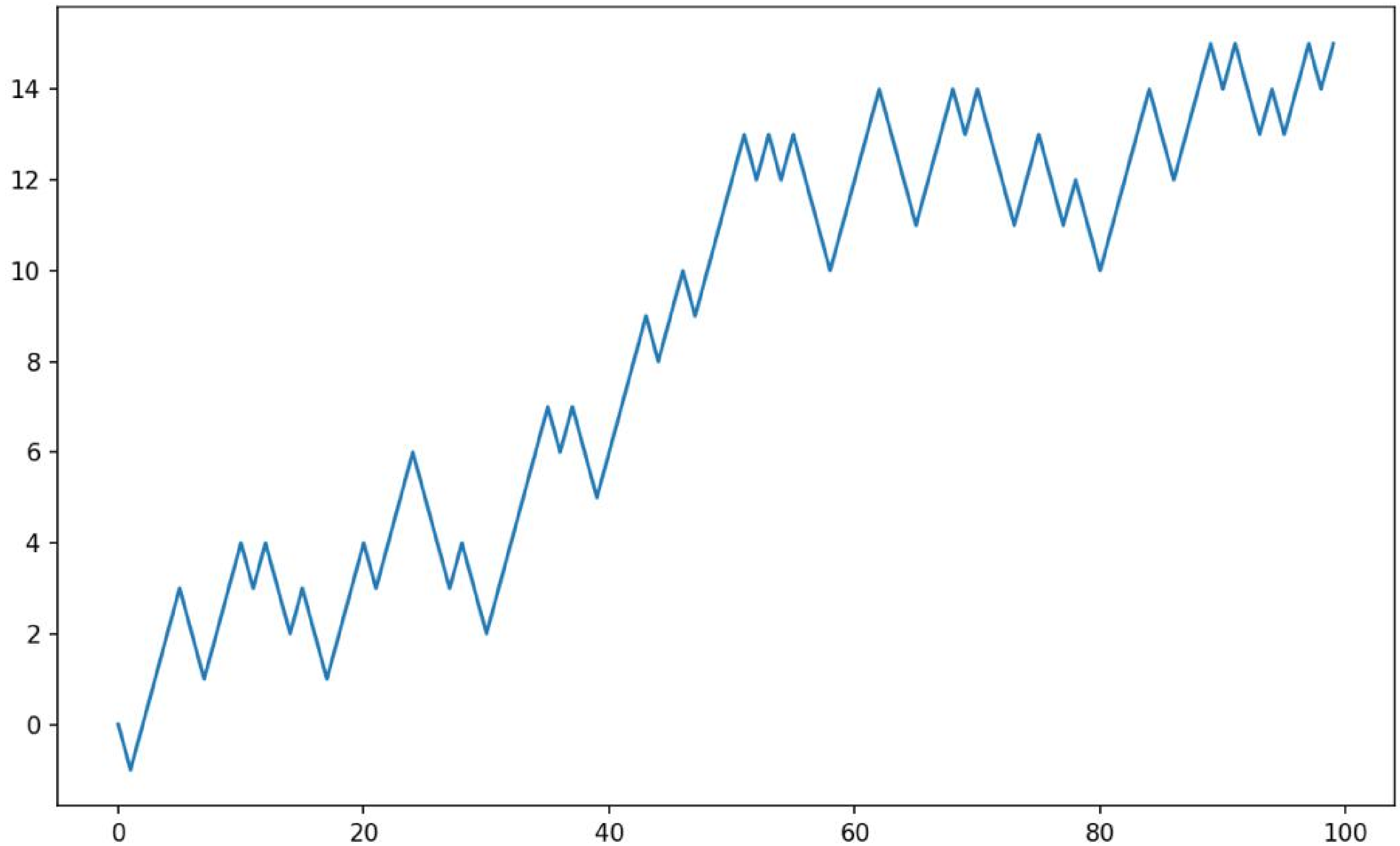
```
In [247]: import random
   .....: position = 0
   .....: walk = [position]
   .....: steps = 1000
   .....: for i in range(steps):
   .....:     step = 1 if random.randint(0, 1) else -1
   .....:     position += step
   .....:     walk.append(position)
In [249]: plt.plot(walk[:100])
```

Example: Random Walks

- use the `np.random` module to compute the cumulative sum:

```
In [251]: nsteps = 1000

In [252]: draws = np.random.randint(0, 2, size=nsteps)

In [253]: steps = np.where(draws > 0, 1, -1)

In [254]: walk = steps.cumsum()

In [255]: walk.min()
Out[255]: -3

In [256]: walk.max()
Out[256]: 31

In [257]: (np.abs(walk) >= 10).argmax()
Out[257]: 37
```

Example: Random Walks



Simulating Many Random Walks at Once

- If passed a 2-tuple, the `numpy.random` functions will generate a two-dimensional array of draws.

```
In [258]: nwalks = 5000

In [259]: nsteps = 1000

In [260]: draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0 or 1

In [261]: steps = np.where(draws > 0, 1, -1)

In [262]: walks = steps.cumsum(1)

In [264]: walks.max()
Out[264]: 138

In [265]: walks.min()
Out[265]: -133
```