Co-funded by the
Erasmus+ Programme
of the European Union

DeCAIR

NumPy: Arrays and Vectorized Computation

Prof. Gheith Abandah

Reference

- Wes McKinney, **Python for Data Analysis**: Data Wrangling with Pandas, NumPy, and IPython, O'Reilly Media, 3rd Edition, 2022.
  - Material: https://github.com/wesm/pydata-book

Outline

Introduction

4.1 The NumPy ndarray: A Multidimensional Array Object

4.2 Universal Functions: Fast Element-Wise Array Functions

4.3 Array-Oriented Programming with Arrays

4.4 File Input and Output with Arrays

4.5 Linear Algebra

4.6 Pseudorandom Number Generation

NumPy: Numerical Python

- One of the most important foundational packages for **fast numerical computing** in Python.
- Most computational packages providing scientific functionality use NumPy’s **array objects** for **data exchange**.
- NumPy internally stores data in a **contiguous block of memory**.
- NumPy’s library of **algorithms written in the C language** can operate on this memory without any type checking or other overhead.

NumPy is Fast

```
In [7]: import numpy as np
In [8]: my_arr = np.arange(1000000)
In [9]: my_list = list(range(1000000))

In [10]: %time for _ in range(10): my_arr2 = my_arr * 2
CPU times: user 20 ms, sys: 50 ms, total: 70 ms
Wall time: 72.4 ms

In [11]: %time for _ in range(10): my_list2 = [x * 2 for x in my_list]
CPU times: user 760 ms, sys: 290 ms, total: 1.05 s
Wall time: 1.05 s
```

Outline

Introduction

4.1 The NumPy ndarray: A Multidimensional Array Object

4.2 Universal Functions: Fast Element-Wise Array Functions

4.3 Array-Oriented Programming with Arrays

4.4 File Input and Output with Arrays

4.5 Linear Algebra

4.6 Pseudorandom Number Generation

- Creating ndarrays
- Data Types for ndarrays
- Arithmetic with NumPy Arrays
- Basic Indexing and Slicing
- Boolean Indexing
- Fancy Indexing
- Transposing Arrays and Swapping Axes

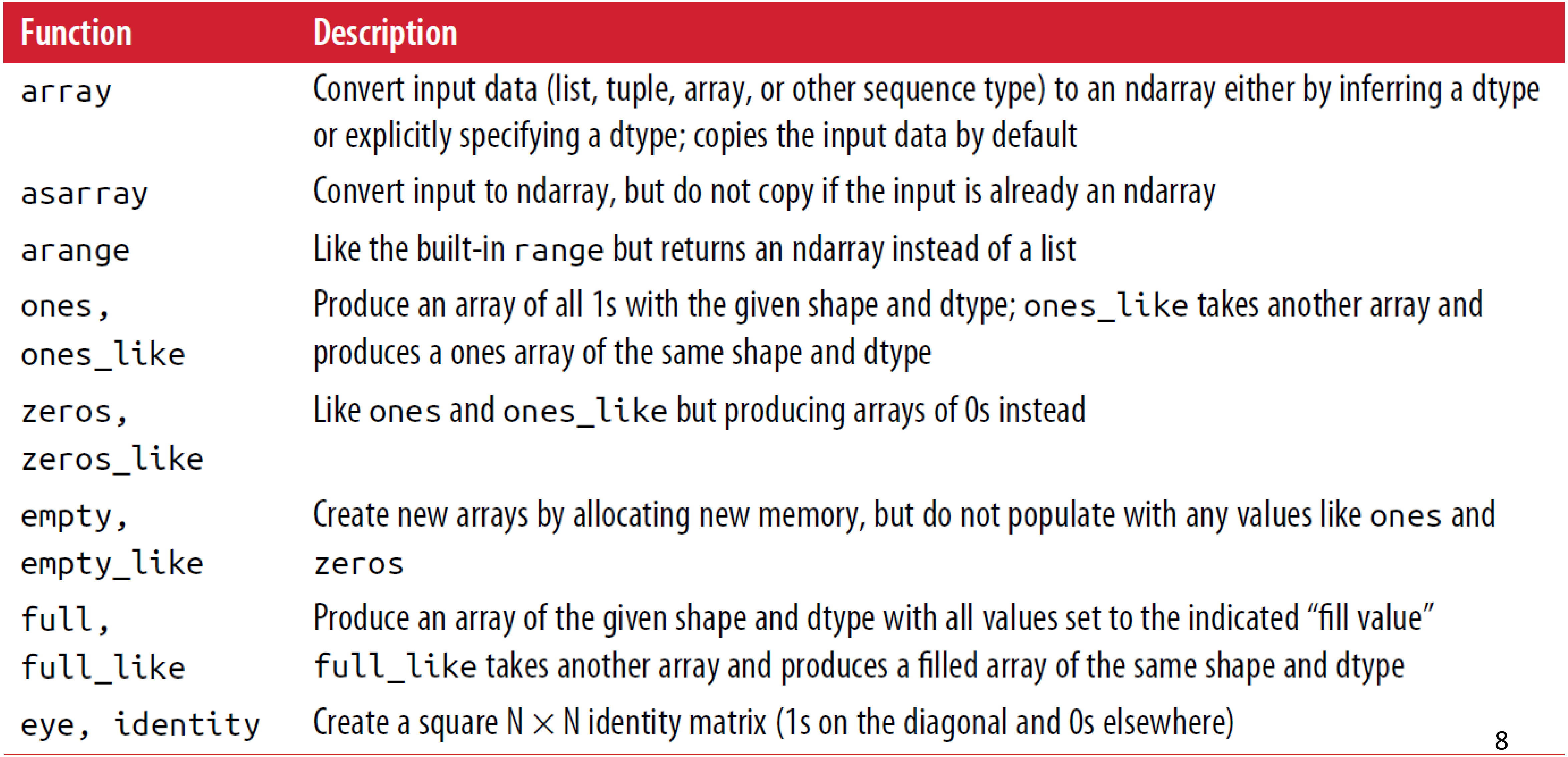
Creating ndarrays

- You can create NumPy arrays **from lists**.

- Arrays have `.ndim` and `.shape` attributes.

```
data2 = [[1,2,3,4], [5,6,7,8]]
arr2 = np.array(data2)
arr2
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])


arr2.ndim
2
arr2.shape
(2, 4)
```

Array Creation Functions



Function	Description
`array`	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype; 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
`ones`, `ones_like`	Produce an array of all 1s with the given shape and dtype; `ones_like` takes another array and produces a ones array of the same shape and dtype
`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 dtype with all values set to the indicated "fill value" `full_like` takes another array and produces a filled array of the same shape and dtype
`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 to interpret a chunk of memory as a particular type of data.
- Arrays have **`.dtype`** attribute.

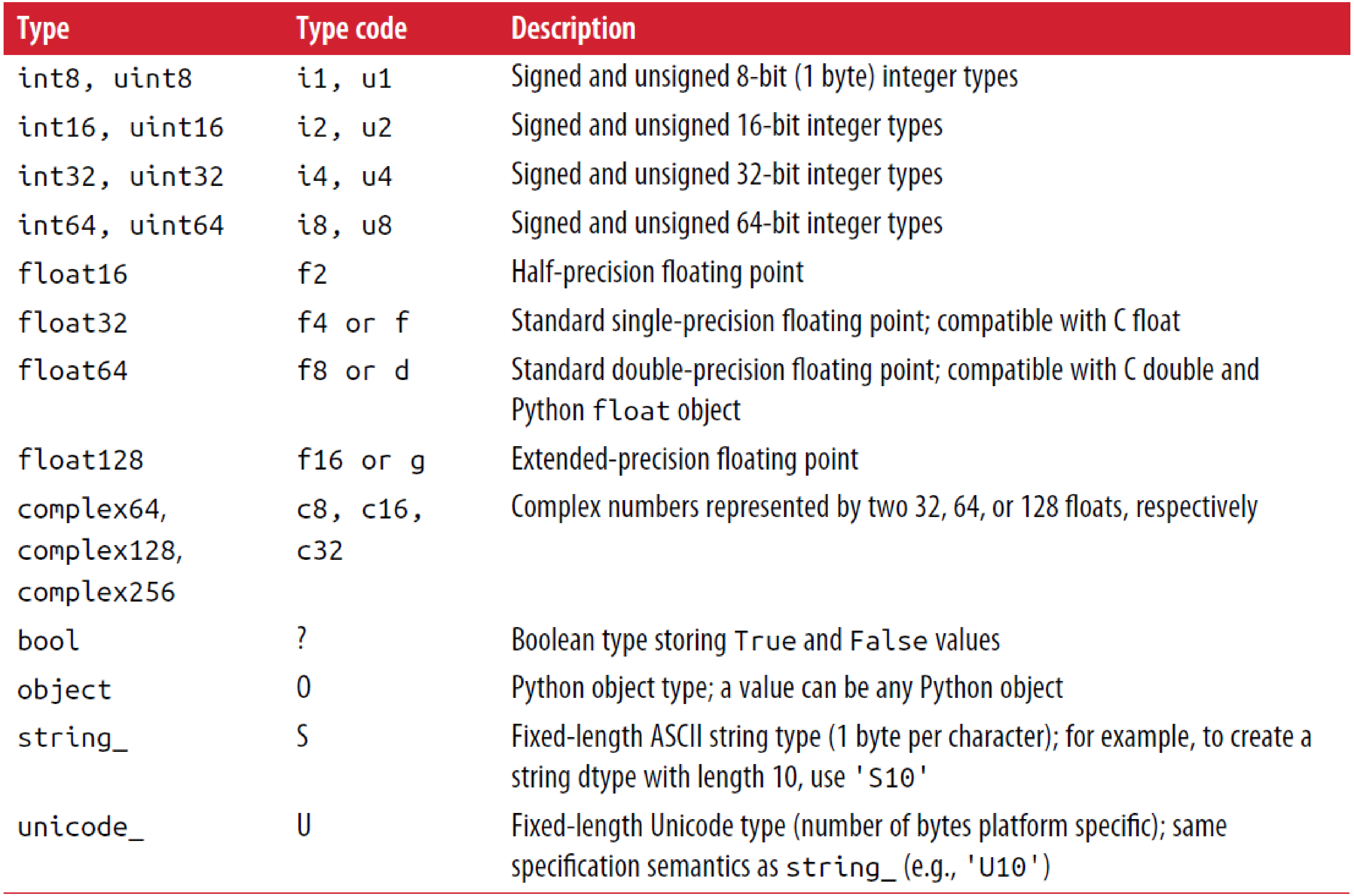
```
a = np.full((2, 3, 2), 7,
            dtype=int)
a
array([[[ 7, 7],
        [ 7, 7],
        [ 7, 7]],
       [[ 7, 7],
        [ 7, 7],
        [ 7, 7]]])
a.dtype
dtype('int32')
```

Or `np.int32`
Or 'i4'

NumPy Data Types

int

float



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
`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 dtype 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 dtype to another.

```
af = np.array([3.7, -1.2, -2.6, 0.5])
ai = af.astype(np.int32)
ai
array([ 3, -1, -2, 0], dtype=int32)
```

- NumPy can convert **strings to numbers**, but Pandas is better.

```
as = np.array(['1.25', '-9.6', '42'],
              dtype=np.string_)
af = as.astype(float)
array([ 1.25, -9.6 , 42. ])
```

Arithmetic with NumPy Arrays

- Any arithmetic operations between **equal-size** arrays applies the operation **element-wise**.

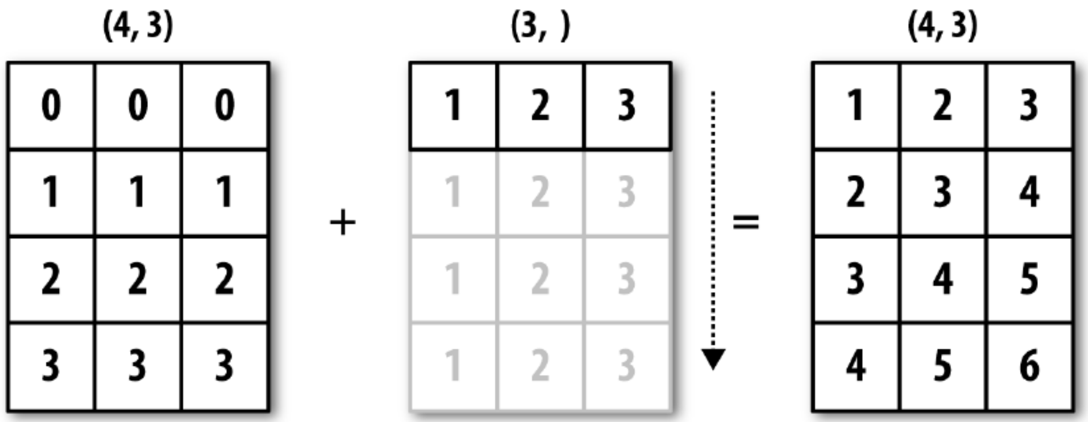
```
arr = np.array([[1., 2., 3.],
                [4., 5., 6.]])
arr * arr
array([[ 1., 4., 9.],
       [ 16., 25., 36.]])
```

- Arithmetic operations with **scalars propagate** the scalar argument to **each element** in the array.

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

Arithmetic with NumPy Arrays

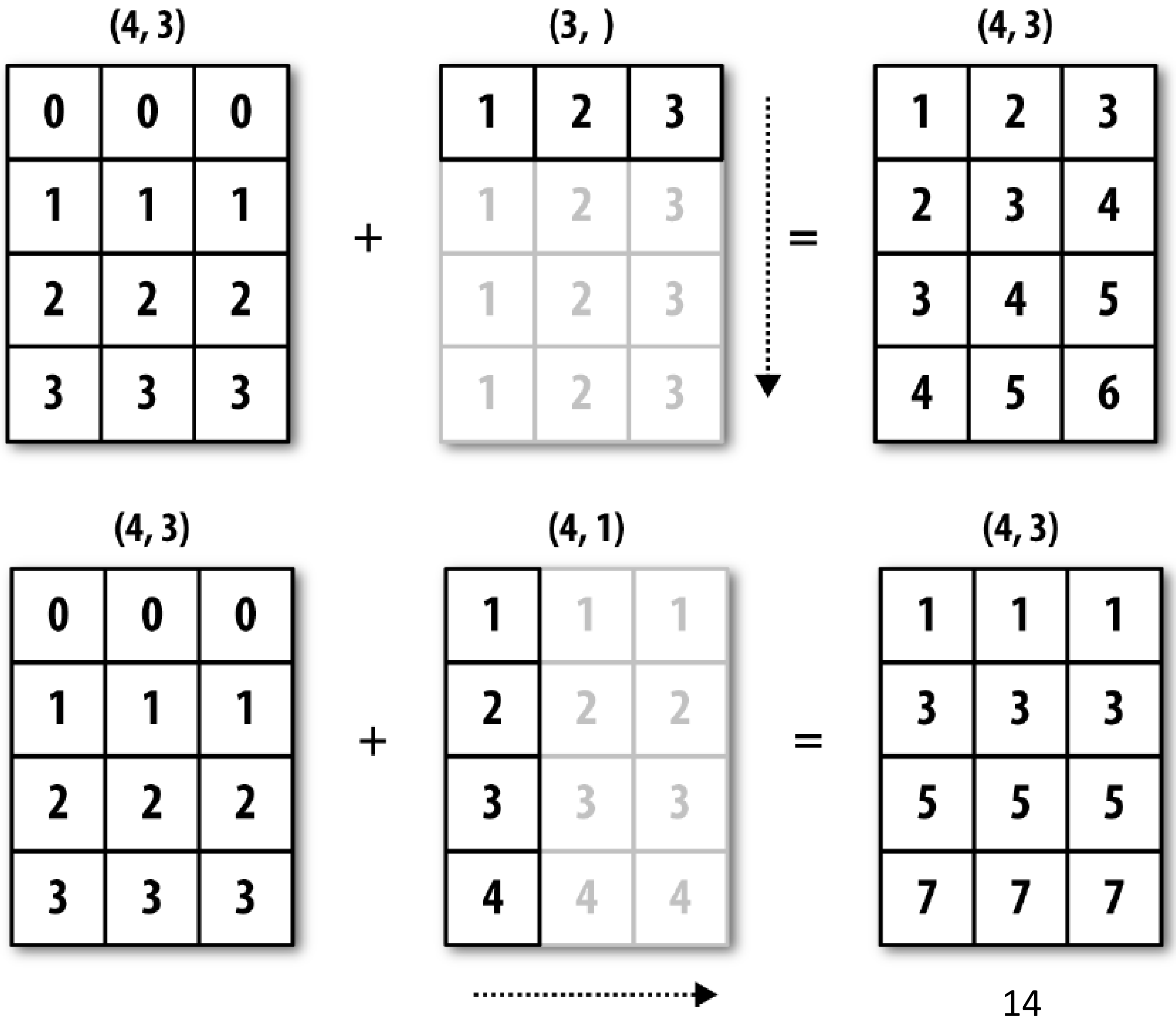
- Operations between **differently sized arrays** is called **broadcasting**.



```
a1 = np.array([[0., 0., 0.],
               [1., 1., 1.],
               [2., 2., 2.],
               [3., 3., 3.]])
a2 = np.array([1., 2., 3.])
a1 + a2
array([[1., 2., 3.],
       [2., 3., 4.],
       [3., 4., 5.],
       [4., 5., 6.]])
```

The Broadcasting Rule

- Two arrays are compatible for broadcasting if for **each trailing dimension** (i.e., starting 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.**



Basic Indexing and Slicing

- **Similar** to **Python** for **one-dimensional** arrays.

```
arr = np.arange(6)
arr[3:5] = 12
arr
array([ 0, 1, 2, 12, 12, 5])
```

- Array slices are **views** on the original array.

```
arr_slice = arr[3:5]
arr_slice[1] = 1000
arr
array([ 0, 1, 2, 12, 1000, 5])
```

Contrast to `arr[3:5].copy()`

Basic Indexing and Slicing

- In a **two-dimensional** array, individual elements can be **accessed**:
  - **recursively** or
  - by passing a **comma-separated list** of indices
- In **multi-dimensional** arrays, if you **omit later indices**, the **returned** object will be a **lower dimensional array** of all the data along the higher dimensions.

```
a = np.array([[1, 2, 3], [4, 5, 6]])
a[0][2]
3
a[1, 2]
6
```

```
a = np.zeros((2, 3, 4))
a[0].shape
(3, 4)
```

Basic Indexing and Slicing

- ndarrays can be **sliced** with the familiar syntax.
- Multiple slices
- Slice in a row
- Using **:** to take the entire access
  - Slices are different than indices

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

```
arr2d[:, 0]
array([1, 4, 7])
```

Boolean Indexing

- **Use Boolean arrays** to **select items** with **`True`**.

```
names = np.array(['Bob', 'Joe',
          'Will', 'Bob', 'Joe'])
data = np.random.randn(5, 3)
data
array([[ 0.2817,  0.769 ,  1.2464],
       [-1.2962,  0.275 ,  0.2289],
       [ 0.8864, -2.0016, -0.3718],
       [-0.4386, -0.5397,  0.477 ],
       [-0.8312, -2.3702, -1.8608]])
```

The Boolean array must be of the same length as the array axis it's indexing.

```
data[names == 'Bob']
array([[ 0.2817,  0.769 , 1.2464],
       [-0.4386, -0.5397, 0.477 ]])

data[names == 'Bob', 2]
array([ 1.2464, 0.477 ])
```

Boolean Indexing

- The operators **!=**, **<**, **<=**, **>**, **>=**, **~**, **&** (and), and **|** (or) can be used to build Boolean arrays.
- **Setting values** with Boolean arrays also works.

```
data[data < 0] = 0
data
array([[ 0.2817,  0.769 ,  1.2464],
       [ 0.    ,  0.275 ,  0.2289],
       [ 0.8864,  0.    ,  0.    ],
       [ 0.    ,  0.    ,  0.477 ],
       [ 0.    ,  0.    ,  0.    ]])
```

Fancy Indexing

- Is **indexing using integer arrays**.
- Creates **new array**.

```
arr = np.arange(20).reshape((5, 4))
arr
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

```
arr[[4, 3, 0]]
array([[16, 17, 18, 19],
       [12, 13, 14, 15],
       [ 0,  1,  2,  3]])
```

```
arr[[1, 2], [0, 2]]
array([ 4, 10])
```

The result is always one-dimensional

Transposing Arrays and Swapping Axes

- Transposing returns a view without copying anything using:
  1. `T` special attribute
  2. `.transpose((1,0))` method

```
arr = np.arange(15).reshape((3, 5))
arr
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
arr.T
array([[ 0, 5, 10],
       [ 1, 6, 11],
       [ 2, 7, 12],
       [ 3, 8, 13],
       [ 4, 9, 14]])
```

Transposing Arrays and Swapping Axes

- For dimensions higher than 2, `transpose` accepts a tuple of axis numbers to permute the axes.
- `swapaxes` takes a pair of axis numbers and switches the indicated axes to rearrange the data.

```
arr = np.arange(24).reshape((2, 3, 4))
arr.shape
(2, 3, 4)

arr.T.shape
(4, 3, 2)

arr.transpose((0, 2, 1)).shape
(2, 4, 3)

arr.swapaxes(1, 0).shape
(3, 2, 4)
```

Outline

Introduction

4.1 The NumPy ndarray: A Multidimensional Array Object

4.2 Universal Functions: Fast Element-Wise Array Functions

4.3 Array-Oriented Programming with Arrays

4.4 File Input and Output with Arrays

4.5 Linear Algebra

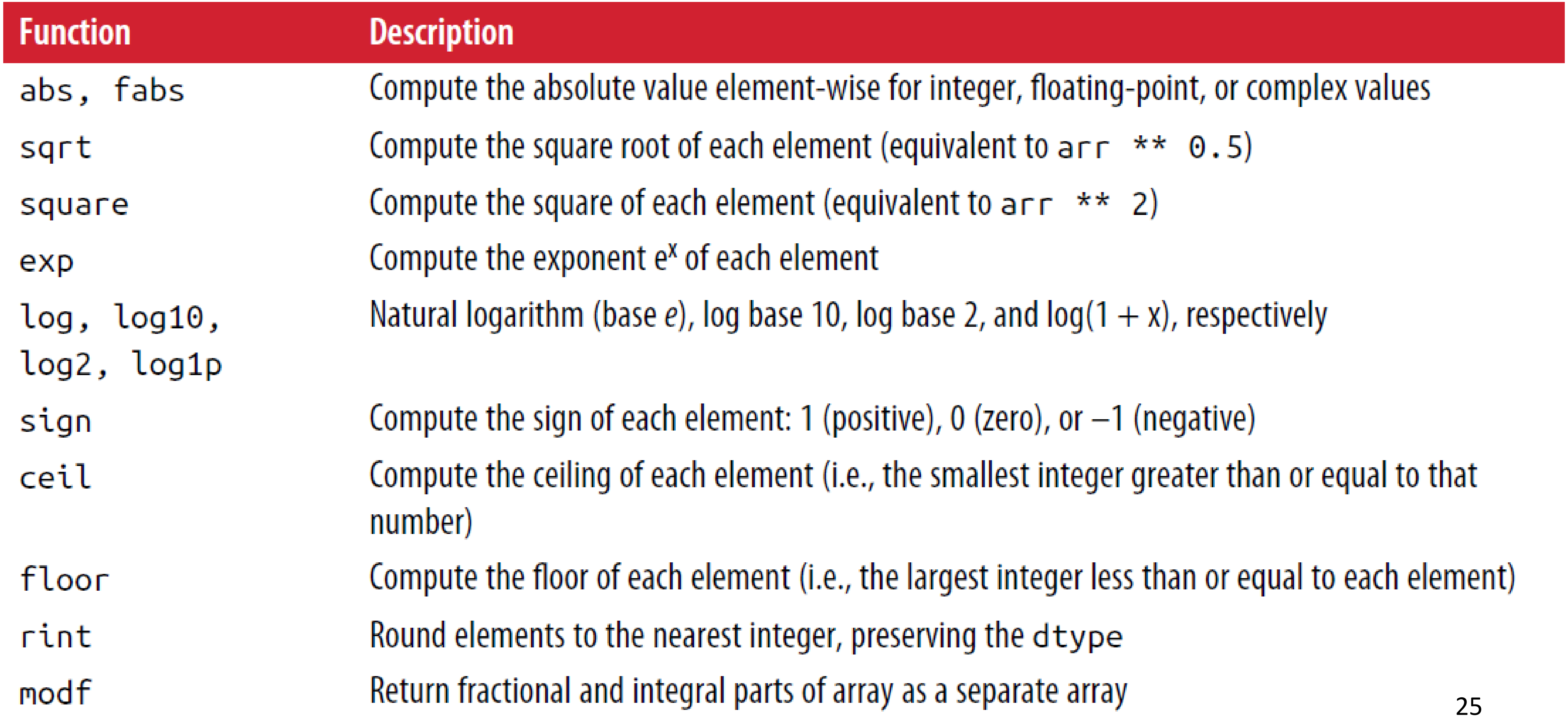
4.6 Pseudorandom Number Generation

4.2 Universal Functions: Fast Element-Wise Array Functions

- Rich set of fast functions.
- **ufunc** is a function that performs element-wise operations.
- Accepts an optional **out** argument that allows them to operate **in-place**.
- There are **unary** and **binary** functions.

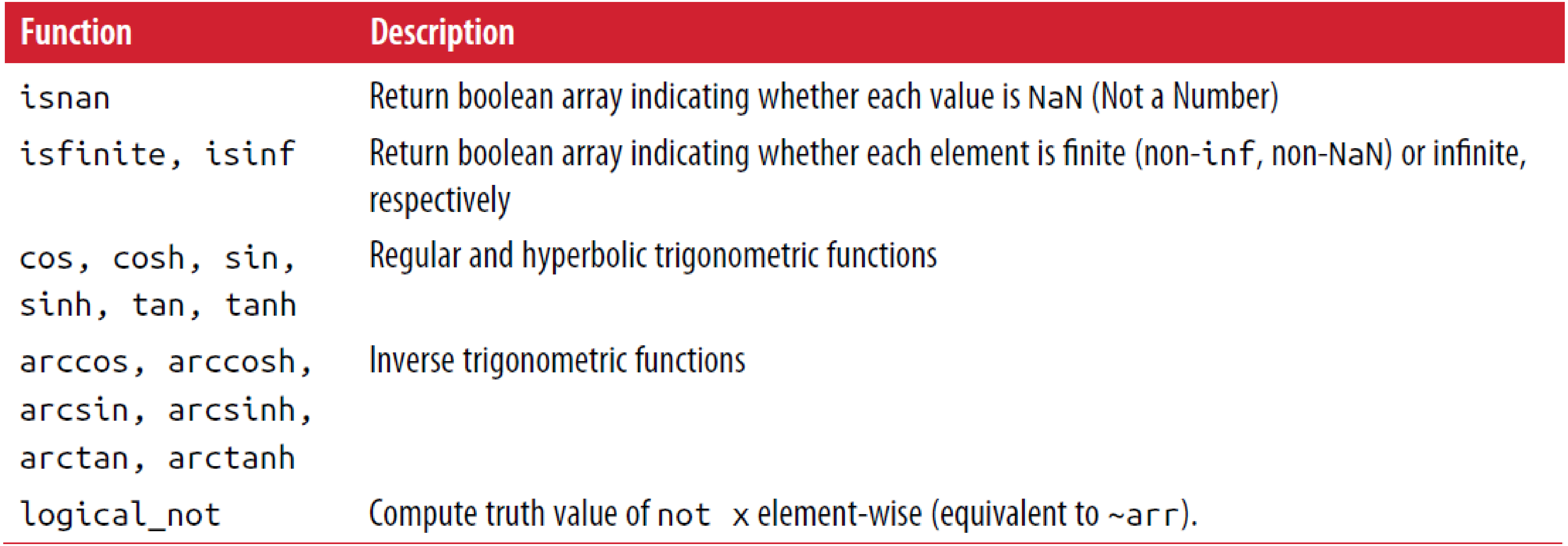
```
arr = np.arange(4)
np.sqrt(arr)
array([ 0. , 1. , 1.4142, 1.7321])
arr
array([0, 1, 2, 3])

np.sqrt(arr, arr)
array([ 0. , 1. , 1.4142, 1.7321])
arr
array([ 0. , 1. , 1.4142, 1.7321])
```

Unary Universal Functions



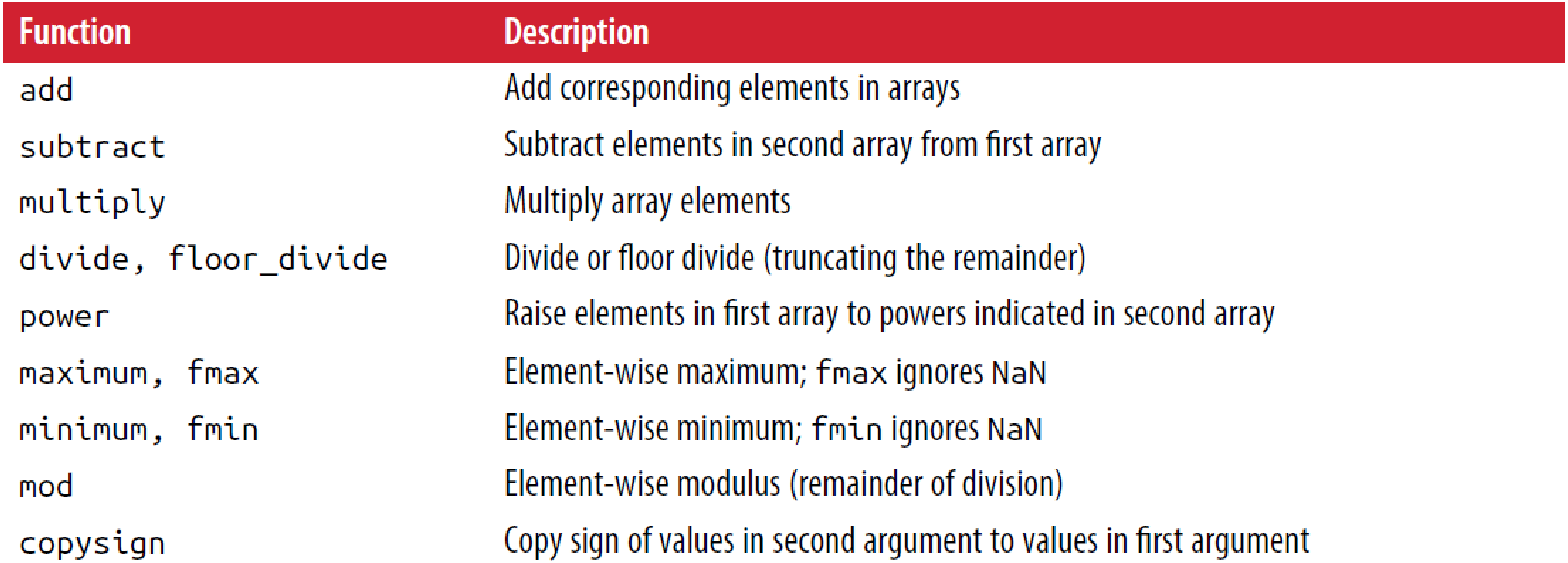
Function	Description
`abs, fabs`	Compute the absolute value element-wise for integer, floating-point, or complex values
`sqrt`	Compute the square root of each element (equivalent to `arr ** 0.5`)
`square`	Compute the square of each element (equivalent to `arr ** 2`)
`exp`	Compute the exponent e^x of 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

Unary Universal Functions – cont.



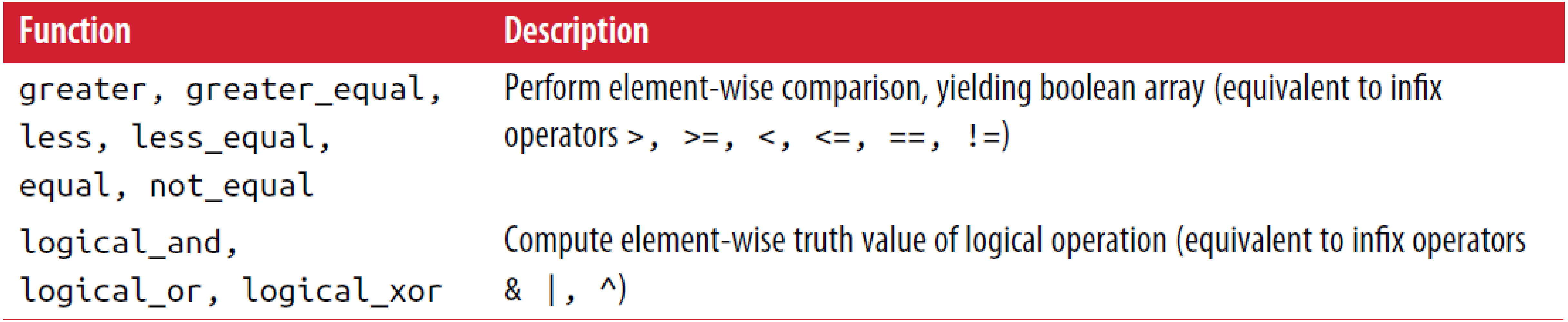
Function	Description
`isnan`	Return boolean array indicating whether each value is `NaN` (Not a Number)
`isfinite, isinf`	Return boolean array indicating whether each element is finite (non-`inf`, non-`NaN`) or infinite, respectively
`cos, cosh, sin, sinh, tan, tanh`	Regular and hyperbolic trigonometric functions
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Inverse trigonometric functions
`logical_not`	Compute truth value of `not x` element-wise (equivalent to `~arr`).

Binary Universal Functions



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

Binary Universal Functions – cont.



Function	Description
`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 `& |, ^`)

Outline

Introduction

4.1 The NumPy ndarray: A Multidimensional Array Object

4.2 Universal Functions: Fast Element-Wise Array Functions

4.3 Array-Oriented Programming with Arrays

- Expressing Conditional Logic as Array Operations
- Mathematical and Statistical Methods
- Methods for Boolean Arrays
- Sorting
- Unique and Other Set Logic

4.4 File Input and Output with Arrays

4.5 Linear Algebra

4.6 Pseudorandom Number Generation

Expressing Conditional Logic as Array Operations

- Python **ternary** expression:

  value = *true-expr* `if` condition `else` *false-expr*

- NumPy has **`np.where()`** function that accepts
  - **Boolean array**
  - **True expression**
  - **False expression**

```
s = 'one'
1 if s == 'one' else 0
1
```

```
a = [[ 1,-1],     b = [[1, 2],
     [-1, 1]]          [3, 4]]
```

```
np.where(a > 0, 5, b)
array([[5, 2],
       [3, 5]])
```

Mathematical and Statistical Methods

- Mathematical functions that **compute** statistics about an **entire array**.
- Call the **instance method** or the top-level **NumPy function**.
- Can compute **along an axis**.
- Not all functions are **reductions**.

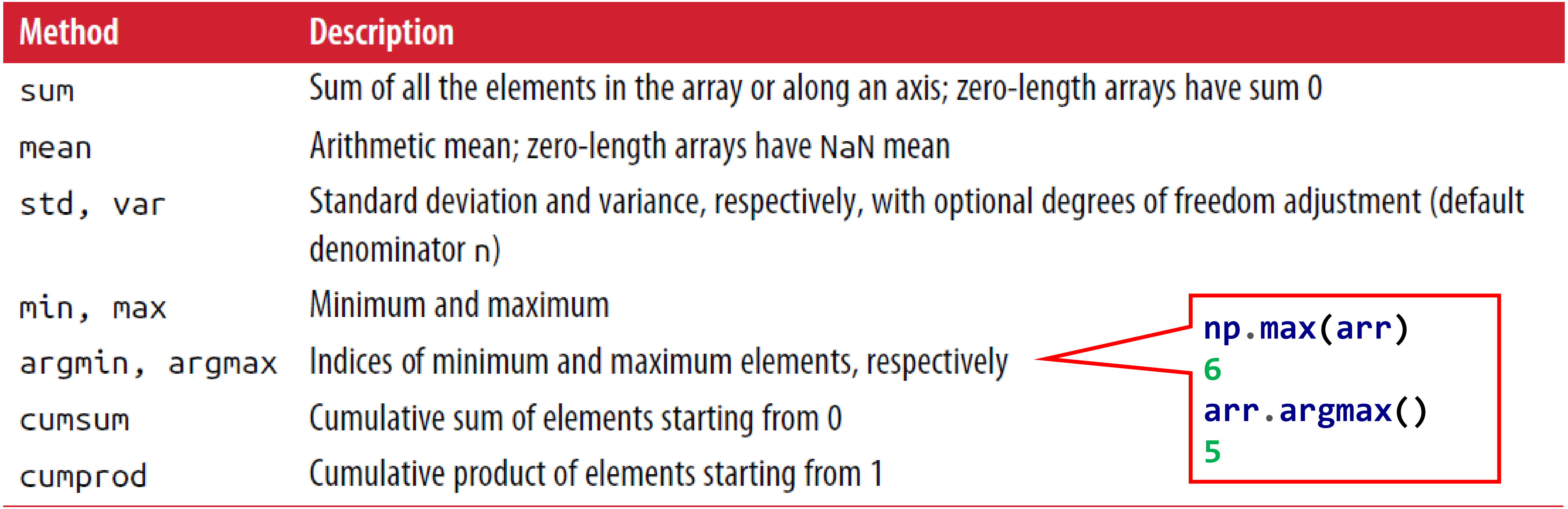
```
arr = [[1, 2, 3],
       [4, 5, 6]]
np.sum(arr)
21
arr.sum()
21
arr.sum(axis = 0)
array([5, 7, 9])
arr.sum(axis = 1)
array([ 6, 15])
arr.cumsum()
array([ 1, 3, 6, 10, 15, 21])
```

Basic Array Statistical Methods

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\mathrm{Var}(X) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

std = sqrt(var)



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

```
np.max(arr)
6
arr.argmax()
5
```

Methods for Boolean Arrays

- Boolean values are coerced to `1` (`True`) and `0` (`False`).

- Useful function:
  - `any()`
  - `all()`

```
arr = [[1, 2, 3],
       [4, 5, 6]]
arr > 4
array([[False, False, False],
       [False,  True,  True]])
(arr > 4).sum()
2
(arr > 4).any()
True
(arr > 4).all()
False
```

Sorting

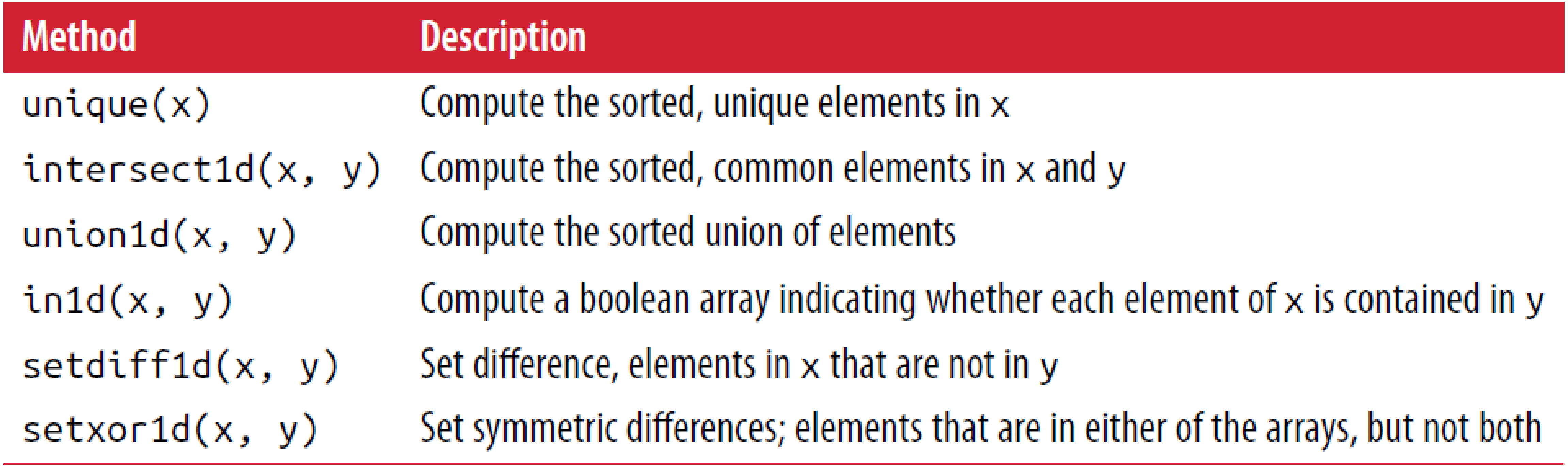
- The top-level **NumPy** **`np.sort`** supports:
  1. Along the **last axis** (default)
  2. **Any axis** you select
  3. **Flattened**

- The **instance method** sorts **in-place** and supports:
  1. Along the **last axis**
  2. **Any axis** you select

```
arr = [[1, 4, 3],
       [5, 2, 6]]
np.sort(arr)
array([[1, 3, 4],
       [2, 5, 6]])
np.sort(arr, axis = None)
array([1, 2, 3, 4, 5, 6])
arr.sort(axis = 0)
arr
array([[1, 2, 3],
       [5, 4, 6]])
```

Unique and Other Set Logic

- NumPy has some basic **set operations** for **one-dimensional** ndarrays.



Method	Description
`unique(x)`	Compute the sorted, unique elements in x
`intersect1d(x, y)`	Compute the sorted, common elements in x and y
`union1d(x, y)`	Compute the sorted union of elements
`in1d(x, y)`	Compute a boolean array indicating whether each element of x is contained in y
`setdiff1d(x, y)`	Set difference, elements in x that are not in y
`setxor1d(x, y)`	Set symmetric differences; elements that are in either of the arrays, but not both

Outline

Introduction

4.1 The NumPy ndarray: A Multidimensional Array Object

4.2 Universal Functions: Fast Element-Wise Array Functions

4.3 Array-Oriented Programming with Arrays

4.4 File Input and Output with Arrays

4.5 Linear Algebra

4.6 Pseudorandom Number Generation

4.4 File Input and Output with Arrays

- NumPy is able to **save** and **load** data to and from disk either in **text** or **binary format**.
- `np.save` and `np.load` are used for efficiently saving and loading in binary format.
- For multiple arrays, use `np.savez`. Load dictionary-like.

```
np.save('file_1', arr)
…
loaded_arr = np.load('file_1.npy')

np.savez('file_2.npz',
         a=arr, b=arr2)
…
arch = np.load('file_2.npz')
arch['b']
array([0, 1, 2, 3, 4, 5])
```

Outline

Introduction

4.1 The NumPy ndarray: A Multidimensional Array Object

4.2 Universal Functions: Fast Element-Wise Array Functions

4.3 Array-Oriented Programming with Arrays

4.4 File Input and Output with Arrays

4.5 Linear Algebra

4.6 Pseudorandom Number Generation

4.5 Linear Algebra

- **NumPy supports linear algebra**, like matrix multiplication, decompositions, determinants, and other square matrix math.
- ***** is element wise operator.
- Use **`np.dot`** for matrix multiplication.

```
a = [[ 1, 2],       b = [[-1, -1],
     [ 3, 4]]            [ 1,  1]]
```

```
a * b
array([[-1, -2],
       [ 3,  4]])
```

```
np.dot(a, b)
array([[1, 1],
       [1, 1]])
```

`≡ a @ b`

Commonly used `numpy.linalg` functions

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
`eig`	Compute the eigenvalues and eigenvectors of a square matrix
`inv`	Compute the inverse of a square matrix
`pinv`	Compute the Moore-Penrose pseudo-inverse of a matrix
`qr`	Compute the QR decomposition
`svd`	Compute the singular value decomposition (SVD)
`solve`	Solve the linear system Ax = b for x, where A is a square matrix
`lstsq`	Compute the least-squares solution to `Ax = b`

Commonly used `numpy.linalg` functions

```
from numpy.linalg import det
a = [[ 1, 2],
     [ 3, 4]]
np.diag(a)
array([1, 4])
np.trace(a)
5
det(a)
-2.
```

```
from numpy.linalg import inv

inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])

a.dot(inv(a))
array([[1.000e+00, 1.110e-16],
       [0.000e+00, 1.000e+00]])
```

Outline

Introduction

4.1 The NumPy ndarray: A Multidimensional Array Object

4.2 Universal Functions: Fast Element-Wise Array Functions

4.3 Array-Oriented Programming with Arrays

4.4 File Input and Output with Arrays

4.5 Linear Algebra

4.6 Pseudorandom Number Generation

4.6 Pseudorandom Number Generation

- **`numpy.random`** provides functions for **efficiently** generating whole arrays of sample values from many kinds of **probability distributions**.
- Example: **Normal distribution**.

Array shape

```
np.random.randn(2)
array([-0.16455161,  0.58873714])

np.random.normal(loc=3.,
                 scale=.01, size=(3, 2))
array([[3.01793583, 3.0055783 ],
       [3.00251166, 3.00951863],
       [2.99502288, 2.99333826]])
```

4.6 Pseudorandom Number Generation

- Generates **pseudorandom** numbers by an algorithm with **deterministic** behavior based on the **seed**.
- You can change the global seed using **`seed()`**.
- **`RandomState()` creates** a random number generator **isolated** from others.

```
np.random.seed(7)
```

```
rng = np.random.RandomState(7)
rng.randn(10)
…
```

Important 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

Simulating 10 coin flips:

```
draws = np.random.randint(0, 2, size=10)
steps = np.where(draws > 0, 1, -1)
```

Homework 4

- Solve the homework on **NumPy**

Summary

Introduction

4.1 The NumPy ndarray: A Multidimensional Array Object

4.2 Universal Functions: Fast Element-Wise Array Functions

4.3 Array-Oriented Programming with Arrays

4.4 File Input and Output with Arrays

4.5 Linear Algebra

4.6 Pseudorandom Number Generation