

The statement `np.any(np.isnan(a))` is a nice idiom for detecting if any nans remain in the array. This is a good check that we have in fact replaced all elements of the initialized array with numbers.

Array **concatenation** is the ordered collection of arrays. The `np.concatenate()` function returns a concatenation of arrays given as a tuple to its first argument. For instance,

```
a = np.array([[0, 1], [2, 3]]) # 2x2
b = np.array([[4, 5]]) # 1x2
np.concatenate((a, b)) # => [[0, 1], [2, 3], [4, 5]] (3x2)
```

The `axis` optional argument, 0 by default, determines the dimension along which the array concatenates. For instance, with the same `a` and `b` from above,

```
np.concatenate((a, b), axis=0) # => [[0, 1], [2, 3], [4, 5]] (3x2)
np.concatenate((a, b.T), axis=1) # => [[0, 1, 4], [2, 3, 5]] (2x3)
```

Here we have used the **transpose** array attribute, which returns a view of the array with its axes swapped (see section 3.2.1). The arrays to be concatenated must have matching dimensions except in the `axis` dimension.

### Box 3.1 Further Reading

- NumPy Developers (2024c), for a basic and short introduction to NumPy

## 3.2 Manipulating, Operating On, and Mapping Over Arrays

In this section, we learn to manipulate, operate on, and map over NumPy arrays.



### 3.2.1 Array Manipulation Functions and Methods

NumPy has many powerful functions and methods for manipulating arrays. We cover only those most frequently useful to us, here; for a full list and documentation, see (NumPy Developers 2024a).

**3.2.1.1 Sorting** To sort an array, the `np.sort(a)` function returns a sorted copy of `a` and the `a.sort()` method will sort (mutate) `a` itself. For instance,

```
a = np.array([6, -3, 0, 9, -6])
np.sort(a) # => [-6, -3, 0, 6, 9] (copy)
a.sort() # a: [-6, -3, 0, 6, 9]
```

The function and the method have the same optional arguments, the most useful of which is `axis: int`, the axis along which to sort. The default is `-1` (i.e., the last dimension).

**3.2.1.2 Transposing** The mathematical matrix transpose (i.e., swapping dimensions by flipping the matrix along its diagonal) can be obtained for a Python matrix via a few different techniques. The following three techniques neither mutate the original matrix nor return transposed copies; rather, they return a transposed view of the original matrix:

```
A = np.array([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]) # 3x4
A.T # Transpose attribute (view)
A.transpose() # Transpose method (view)
np.transpose(A) # Transpose function (view)
```

All three transpose statements return a  $4 \times 3$  array view that prints as follows:

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

The original array `A` remains the same and is linked to the transposed view objects. To get a transposed copy, append the `copy()` method to any of these statements.

Unlike for matrices, a vector transpose view is no different than the original vector. However, a **row vector** (i.e., 2D array with first axis of length 1) or a **column vector** (i.e., 2D array with second axis of length 1) can be created by adding an axis to a vector. For instance,

```
a = np.array([0, 1, 2, 3]) # A vector
a.T # => [0, 1, 2, 3] (same vector view)
a[np.newaxis, :] # => [[0, 1, 2, 3]] (1x4 row vector view)
a[:, np.newaxis] # => [[0], [1], [2], [3]] (4x1 column vector view)
```

The following are the shapes of these objects:

- `a.shape` returns `(4,)` (i.e., a 1D array of size 4)
- `a[np.newaxis, :].shape` returns `(1, 4)` (i.e., a 2D view of shape  $1 \times 4$ )
- `a[:, np.newaxis].shape` returns `(4, 1)` (i.e., a 2D view of shape  $4 \times 1$ )

Constructing row and column vectors will be important for computing mathematical matrix-vector multiplication. They can also be properly transposed back-and-forth between row and column vectors.

**3.2.1.3 Reshaping** Transposing, as we have seen, is one way to reshape an array. Another way is to use the `np.reshape(a: np.ndarray, newshape: tuple)` function or the equivalent method for array `a`, `a.reshape(newshape: tuple)`. Both return a view of the original array with its elements filling the newly shaped array. The argument `newshape` may be an `int`, in which case the array is flattened 1D array, or a `tuple` following the usual pattern of an array shape. The number of elements in the new view must equal that of the original array. For instance,

```
A = np.array([[0, 1, 2], [3, 4, 5]]) # A 2x3 matrix
Ar = A.reshape((3,2)) # => [[0, 1], [2, 3], [4, 5]] (3x2 view)
```

This also provides a second way of forming a row or column vector view from a 1D array; for example,

```
a = np.array([0, 1, 2])
a_row = np.reshape((1, len(a))) # 1x3 row vector view
a_col = np.reshape((len(a), 1)) # 3x1 column vector view
```

### 3.2.2 Operations on Arrays and Broadcasting

The basic arithmetic operators  $+$ ,  $-$ ,  $\times$ , and  $/$  can be applied to NumPy arrays with the operators  $+$ ,  $-$ ,  $*$ , and  $/$ , respectively. These operations are applied **element-wise** as the following example demonstrates:

```
a = np.array([0, 1, 2])
b = np.array([3, 4, 5])
a + b # => [3, 4, 7]
a - b # => [-3, -3, -3]
a * b # => [0, 4, 10]
a / b # => [0, 0.25, 0.4]
```

**3.2.2.1 Broadcasting** In the cases above, the array shapes matched exactly. However, it is convenient to be able to perform these types of operations on arrays of different size such that the smaller array dimensions are **broadcast** (i.e., stretched or copied) to fill in the portions of the array it is missing. The simplest case is for an operation between a 0D array (i.e., a scalar) and another array, as in the following cases:

```
a = np.array([0, 1, 2])
a + 4 # = a + [4, 4, 4] => [4, 5, 6]
a - 4 # = a - [4, 4, 4] => [-4, -3, -2]
a * 4 # = a * [4, 4, 4] => [0, 4, 8]
a / 4 # = a / [4, 4, 4] => [0, 0.25, 0.5]
```

Here the scalar 4 was broadcast to match the (larger) a array with shape (3,) and added element-wise.

Broadcasting is quite general and works for operations between arrays of many dimensions. Dimensions of two arrays are compatible if they are of equal size or if one has size 1, in which case it can be broadcast. The dimensions are compared from last to first. If one array runs out of dimensions, the rest are treated as 1. Here are some examples of compatible array dimensions in each column:

$$\begin{array}{ccc}
 3 \times 3 & 7 \times 1 \times 4 & 9 \times 7 \times 4 \\
 1 \times 3 & 4 \times 4 & 7 \times 1 \\
 3 \times 1 & 7 \times 4 \times 1 & 500 \times 1 \times 1 \times 4 \\
 4 \times 3 \times 1 & 1 \times 4 & 9 \times 1 \times 1
 \end{array}$$

Operations on arrays with compatible dimensions will be broadcast automatically. This is not only convenient, in most cases it is also much more efficient (in terms of memory usage and computation time) than constructing the arrays or executing loops.<sup>2</sup> Therefore, we usually prefer broadcasting.

**3.2.2.2 Matrix Multiplication** Matrix multiplication can be performed with the `@` operator. For instance, consider the matrices and column vector

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}, \quad \text{and} \quad x = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}.$$

Further consider the following matrix products:

$$AB, \quad Ax, \quad \text{and} \quad B^T Ax.$$

The following code computes these products:

```
A = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]]) # 3x3
B = np.array([[0, 1], [2, 3], [4, 5]]) # 3x2
x = np.array([[0], [1], [2]]) # 3x1
A @ B # => [[10, 13], [28, 40], [46, 67]] (3x2 matrix)
A @ x # => [[5], [14], [23]] (3x1 column vector)
B.T @ A @ x # => [[120], [162]] (2x1 column vector)
```

The `@` operator is equivalent to the use of the `np.matmul()` function. A related function is `np.dot(a, b)`, which takes the dot product of `a` and `b`. If `a` and `b` are matrices, this is equivalent to `a @ b`. However, in this case `np.matmul()` and `a @ b` are preferred.

**3.2.2.3 Other Matrix Operations** We have considered matrix transposes and multiplication. Other common mathematical matrix operations include addition, subtraction, and scalar multiplication. These are element-wise operations, so we can simply use NumPy's usual `+`, `-`, and `*` operators, respectively.

The multiplicative inverse  $A^{-1}$  of a matrix  $A$  can be computed with the `np.linalg.inv()` function from the `linalg` module. For example,

```
A = np.array([[1, 0, 0], [0, 2, 0], [0, 0, 4]]) # 3x3
np.linalg.inv(A) # => [[1., 0., 0.], [0., 0.5, 0.], [0., 0., 0.25]]
```

2. Loops are executed in broadcasting, but these are loops in the more-efficient C programming language (in which Python is written), not in Python.

If the matrix is not invertible, the exception `LinAlgError: Singular matrix` is raised.

**3.2.2.4 Element-Wise Mathematical Functions** NumPy has mathematical functions that automatically operate element-wise on arrays. Trigonometric functions include `np.sin()`, `np.cos()`, and `np.tan()`; exponential and logarithmic functions include `np.exp()`, `np.log()`, and `np.log10()`; hyperbolic functions include `np.sinh()`, `np.cosh()`, and `np.tanh()`; complex-number functions include `np.real()`, `np.imag()`, and `np.angle()`; rounding functions include `np.round()`, `np.ceil()`, and `np.floor()`. All these functions operate element-wise, as shown in the following example:

```
x = np.linspace(0, 2*np.pi, 5)
np.round(np.sin(x), 10) # => [0., 1., 0., -1., -0.] (round to 10 dec.)
np.round(np.cos(x), 10) # => [1., 0., -1., -0., 1.] (round to 10 dec.)
```

This element-wise operation is not only convenient, it is highly optimized. NumPy takes advantage of precompiled C functions for performing these operations, so they execute much faster than would a Python loop through each element. The element-wise operation is called **vectorization** (n.b., sometimes this is the term given to the sometimes-attendant optimization), and NumPy takes great advantage of this, which is one of its key features.

### 3.2.3 Mapping Over Arrays and Lambda Functions

As we have seen, NumPy includes many built-in functions that are vectorized (i.e., applied element-wise). Our own custom function and method definitions can (and often should) also be vectorized. Usually, nothing special is required because we can take advantage of NumPy's built-in functions and broadcasting. For instance, consider the following example, which defines a Python function corresponding to  $x \mapsto \sqrt{x} - 1$ :

```
def sqrt_m1(x: np.ndarray) -> np.ndarray:
    return np.sqrt(x) - 1
```

Here `np.sqrt(x)` is already vectorized and the subtraction is automatically broadcast, so our `sqrt_m1` is vectorized. Note that this will be much faster than a **for** loop through the elements of `x`.

In general, the application of a function to each element of an array (or iterable) object is called **mapping**. In plain Python, we can apply a function `f` to each element of a list `l` with the built-in function `map(f, l)`. This is effectively just a **for** loop, which is not particularly performant. Vectorization in NumPy allows us to usually avoid **for** loops or equivalent calls to `map()`.

**3.2.3.1 Lambda Functions** At times, it is convenient to write an **anonymous function**, often called a **lambda function**, which is a function that need not be given a name (although it can be). Mathematically, a lambda function can be expressed as, for instance,

$$x \mapsto (x + 2)^3.$$

The Python syntax for a corresponding lambda function is

```
| lambda x: (x + 2) ** 3
```

A lambda function can be applied directly to an argument. For instance,

```
| (lambda x: (x + 2) ** 3)(1) ## => 27
```

It can also be given a name, as in

```
| f = lambda x: (x + 2) ** 3
| f(1) # => 27
```

In some ways, this defeats the purpose of the lambda function. The PEP 8 style guide discourages this use.

So when is a lambda function actually useful? One case is for applying a non-vectorized function to a list. For instance,

```
| l = [1, 2, 3]
| list(map(lambda x: x ** 2, l)) # => [1, 4, 9]
```

However, in this and most cases where numerical computation, it is better to use the vectorization of NumPy. In the case of a non-numerical function mapping over a list of strings, the lambda function is a good choice, as in the following case:

```
| l = ["foo", "bar", "baz"]
| list(map(lambda s: s.capitalize(), l)) # => ["Foo", "Bar", "Baz"]
```

**3.2.3.2 Conditional Functions** There are some more complex custom functions that are difficult to vectorize. An example is a function with conditions. Consider the following function:

```
| def square_positive(x):
|     if x > 0:
|         return x ** 2
|     else:
|         return x
```

This function can be applied to a single number  $x$ , but it cannot take an array argument. One solution would be to write a **for** loop over the elements of  $x$ , but this would be inefficient.

The function `np.where(condition, a_true, a_false)` returns an array chosen from `a_true` and `a_false` based on the condition. Consider the following version of `square_positive()`:

```
def square_positive(x: np.ndarray) -> np.ndarray:
    return np.where(x > 0, x ** 2, x)
```

This is vectorized so it can be applied to arrays and it is much more performant than a **for**-loop solution.

### Box 3.2 Further Reading

- NumPy Developers (2024b), for a thorough introduction to NumPy
- NumPy Developers (2024a), for the API reference that describes NumPy classes, functions, and methods in detail


## 3.3 Input and Output

A program's **input** and **output** refer to the the information provided to a program from without and the information the program produces.

We have already seen examples of Python programs' output in the form of text printed to a console. Thus far, our programs have had no input because they have contained all the information they need.

In this section, we consider a few important types of Python input and output. In section 3.4, graphical output will be introduced.

### 3.3.1 User Input

A user can interact directly with a Python program via the built-in function `input(prompt)`. The prompt argument is printed to the console and the user can type in a response, finishing with the  key. For instance, consider the following program:

```
import fractions # Built-in module
response = input( # Solicit user input
    "What are my chances? (Enter a fraction): " # Prompt
)
if float(fractions.Fraction(response)) > 0.:
    print("So you're tellin' me there's a chance. YEAH!")
```

Running this program prints the following prompt:

```
| What are my chances? (Enter a fraction): |
```

Suppose the user enters `1/1_000_000`. This is read by the input function and stored as a `str` in the variable `response`. A string can be cast to a fraction using

