NumPy

This notebook offers comprehensive guidance on the NumPy library, which is advantageous for students utilising Python for data processing. It is therefore imperative to peruse it meticulously.

> "Let's be clear: the work of science has nothing whatever to do with consensus. Consensus is the business of politics. Science, on the contrary, requires only one investigator who happens to be right, which means that he or she has results that are verifiable by reference to the real world. In science consensus is irrelevant. What is relevant is reproducible results." – Michael Crichton

Overview

NumPy is a first-rate library for numerical programming

- Widely used in academia, finance and industry.
- Mature, fast, stable and under continuous development.

We have already seen some code involving NumPy in the preceding lectures.

In this lecture, we will start a more systematic discussion of both

- NumPy arrays and
- the fundamental array processing operations provided by NumPy.

References

- The official NumPy documentation.

NumPy Arrays

The essential problem that NumPy solves is fast array processing.

The most important structure that NumPy defines is an array data type formally called a numpy.ndarray.

NumPy arrays power a large proportion of the scientific Python ecosystem.

Let's first import the library.

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

To create a NumPy array containing only zeros we use np.zeros

In [ ]:
```
a = np.zeros(3)
a
```

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

In [ ]:
```
type(a)
```

Out[ ]:
```
numpy.ndarray
```

NumPy arrays are somewhat like native Python lists, except that

- Data *must be homogeneous* (all elements of the same type).
- These types must be one of the data types (`dtypes`) provided by NumPy.

The most important of these dtypes are:

- float64: 64 bit floating-point number
- int64: 64 bit integer
- bool: 8 bit True or False

There are also dtypes to represent complex numbers, unsigned integers, etc.

On modern machines, the default dtype for arrays is `float64`

In [ ]:
```
a = np.zeros(3)
type(a[0])
```

Out[ ]:
```
numpy.float64
```

If we want to use integers we can specify as follows:

In [ ]:
```
a = np.zeros(3, dtype=int)
type(a[0])
```

Out[ ]:
```
numpy.int64
```

Shape and Dimension

Consider the following assignment

In [ ]:
```
z = np.zeros(10)
```

Here `z` is a *flat* array with no dimension — neither row nor column vector.

The dimension is recorded in the `shape` attribute, which is a tuple

In [ ]:
```
z.shape
```

Out[ ]:
```
(10,)
```

Here the shape tuple has only one element, which is the length of the array (tuples with one element end with a comma).

To give it dimension, we can change the `shape` attribute

In [ ]:
```
z.shape = (10, 1)
z
```

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

In [ ]:
```
z = np.zeros(4)
z.shape = (2, 2)
z
```

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

In the last case, to make the 2 by 2 array, we could also pass a tuple to the `zeros()` function, as in `z = np.zeros((2, 2))`.

Creating Arrays

As we've seen, the `np.zeros` function creates an array of zeros.

You can probably guess what `np.ones` creates.

Related is `np.empty`, which creates arrays in memory that can later be populated with data

In [ ]:
```
z = np.empty(3)
z
```

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

The numbers you see here are garbage values.

(Python allocates 3 contiguous 64 bit pieces of memory, and the existing contents of those memory slots are interpreted as `float64` values)

To set up a grid of evenly spaced numbers use `np.linspace`

In [ ]:
```
z = np.linspace(2, 4, 5)  # From 2 to 4, with 5 elements
```

To create an identity matrix use either `np.identity` or `np.eye`

In [ ]:
```
z = np.identity(2)
z
```

Out[ ]:
```
array([[1., 0.],
       [0., 1.]])
```

In addition, NumPy arrays can be created from Python lists, tuples, etc. using `np.array`

In [ ]:
```
z = np.array([10, 20])                 # ndarray from Python list
z
```

Out[ ]:
```
array([10, 20])
```

In [ ]:
```
type(z)
```

Out[ ]:
```
numpy.ndarray
```

In [ ]:
```
z = np.array((10, 20), dtype=float)    # Here 'float' is equivalent to 'n
z
```

Out[ ]:
```
array([10., 20.])
```

In [ ]:
```
z = np.array([[1, 2], [3, 4]])         # 2D array from a list of lists
z
```

Out[ ]:
```
array([[1, 2],
       [3, 4]])
```

See also `np.asarray`, which performs a similar function, but does not make a distinct copy of data already in a NumPy array.

In [ ]:
```
na = np.linspace(10, 20, 2)
na is np.asarray(na)   # Does not copy NumPy arrays
```

Out[ ]:
```
True
```

In [ ]:
```
na is np.array(na)     # Does make a new copy --- perhaps unnecessarily
```

Out[ ]:
```
False
```

To read in the array data from a text file containing numeric data use `np.loadtxt` or `np.genfromtxt`—see the documentation for details.

Array Indexing

For a flat array, indexing is the same as Python sequences:

In [ ]:
```
z = np.linspace(1, 2, 5)
z
```

Out[ ]:
```
array([1.  , 1.25, 1.5 , 1.75, 2.  ])
```

In [ ]:
```
z[0]
```

Out[ ]:
```
1.0
```

```
In [ ]: z[0:2]  # Two elements, starting at element 0
```

```
Out[ ]: array([1.  , 1.25])
```

```
In [ ]: z[-1]
```

```
Out[ ]: 2.0
```

For 2D arrays the index syntax is as follows:

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

```
Out[ ]: array([[1, 2],
               [3, 4]])
```

```
In [ ]: z[0, 0]
```

```
Out[ ]: 1
```

```
In [ ]: z[0, 1]
```

```
Out[ ]: 2
```

And so on.

Note that indices are still zero-based, to maintain compatibility with Python sequences.

Columns and rows can be extracted as follows

```
In [ ]: z[0, :]
```

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

```
In [ ]: z[:, 1]
```

```
Out[ ]: array([2, 4])
```

NumPy arrays of integers can also be used to extract elements

```
In [ ]: z = np.linspace(2, 4, 5)
        z
```

```
Out[ ]: array([2. , 2.5, 3. , 3.5, 4. ])
```

```
In [ ]: indices = np.array((0, 2, 3))
        z[indices]
```

```
Out[ ]: array([2. , 3. , 3.5])
```

Finally, an array of `dtype bool` can be used to extract elements

```
In [ ]: z
```

```
Out[ ]: array([2. , 2.5, 3. , 3.5, 4. ])
```

```
In [ ]: d = np.array([0, 1, 1, 0, 0], dtype=bool)
        d
```

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

```
In [ ]: z[d]
```

```
Out[ ]: array([2.5, 3. ])
```

We’ll see why this is useful below.

An aside: all elements of an array can be set equal to one number using slice notation

```
In [ ]: z = np.empty(3)
        z
```

```
Out[ ]: array([2. , 3. , 3.5])
```

```
In [ ]: z[:] = 42
        z
```

```
Out[ ]: array([42., 42., 42.])
```

Array Methods

Arrays have useful methods, all of which are carefully optimized

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

```
Out[ ]: array([4, 3, 2, 1])
```

```
In [ ]: a.sort()              # Sorts a in place
        a
```

```
Out[ ]: array([1, 2, 3, 4])
```

```
In [ ]: a.sum()               # Sum
```

```
Out[ ]: 10
```

```
In [ ]: a.mean()              # Mean
```

```
Out[ ]: 2.5
```

```
In [ ]: a.max()               # Max
```

```
Out[ ]: 4
```

```
In [ ]: a.argmax()            # Returns the index of the maximal element
```

```
Out[ ]: 3
```

```
In [ ]: a.cumsum()              # Cumulative sum of the elements of a
```

```
Out[ ]: array([ 1,  3,  6, 10])
```

```
In [ ]: a.cumprod()             # Cumulative product of the elements of a
```

```
Out[ ]: array([ 1,  2,  6, 24])
```

```
In [ ]: a.var()                 # Variance
```

```
Out[ ]: 1.25
```

```
In [ ]: a.std()                 # Standard deviation
```

```
Out[ ]: 1.118033988749895
```

```
In [ ]: a.shape = (2, 2)
        a.T                     # Equivalent to a.transpose()
```

```
Out[ ]: array([[1, 3],
               [2, 4]])
```

Another method worth knowing is `searchsorted()`.

If `z` is a nondecreasing array, then `z.searchsorted(a)` returns the index of the first element of `z` that is `>= a`

```
In [ ]: z = np.linspace(2, 4, 5)
        z
```

```
Out[ ]: array([2. , 2.5, 3. , 3.5, 4. ])
```

```
In [ ]: z.searchsorted(2.2)
```

```
Out[ ]: 1
```

Many of the methods discussed above have equivalent functions in the NumPy namespace

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

```
In [ ]: np.sum(a)
```

```
Out[ ]: 10
```

```
In [ ]: np.mean(a)
```

```
Out[ ]: 2.5
```

Arithmetic Operations

The operators `+`, `-`, `*`, `/` and `**` all act *elementwise* on arrays

```
In [ ]: a = np.array([1, 2, 3, 4])
        b = np.array([5, 6, 7, 8])
        a + b
```

```
Out[ ]: array([ 6,  8, 10, 12])
```

```
In [ ]: a * b
```

```
Out[ ]: array([ 5, 12, 21, 32])
```

We can add a scalar to each element as follows

```
In [ ]: a + 10
```

```
Out[ ]: array([11, 12, 13, 14])
```

Scalar multiplication is similar

```
In [ ]: a * 10
```

```
Out[ ]: array([10, 20, 30, 40])
```

The two-dimensional arrays follow the same general rules

```
In [ ]: A = np.ones((2, 2))
        B = np.ones((2, 2))
        A + B
```

```
Out[ ]: array([[2., 2.],
               [2., 2.]])
```

```
In [ ]: A + 10
```

```
Out[ ]: array([[11., 11.],
               [11., 11.]])
```

```
In [ ]: A * B
```

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

In particular, `A * B` is *not* the matrix product, it is an element-wise product.

Matrix Multiplication

With Anaconda's scientific Python package based around Python 3.5 and above, one can use the `@` symbol for matrix multiplication, as follows:

```
In [ ]: A = np.ones((2, 2))
        B = np.ones((2, 2))
        A @ B
```

```
Out[ ]: array([[2., 2.],
               [2., 2.]])
```

(For older versions of Python and NumPy you need to use the np.dot function)

We can also use `@` to take the inner product of two flat arrays

```
In [ ]: A = np.array((1, 2))
        B = np.array((10, 20))
        A @ B
```

```
Out[ ]: 50
```

In fact, we can use `@` when one element is a Python list or tuple

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

```
Out[ ]: array([[1, 2],
               [3, 4]])
```

```
In [ ]: A @ (0, 1)
```

```
Out[ ]: array([2, 4])
```

Since we are post-multiplying, the tuple is treated as a column vector.

Broadcasting

(This section extends an excellent discussion of broadcasting provided by Jake VanderPlas.)

> **Note**
>
> Broadcasting is a very important aspect of NumPy. At the same time, advanced broadcasting is relatively complex and some of the details below can be skimmed on first pass.

In element-wise operations, arrays may not have the same shape.

When this happens, NumPy will automatically expand arrays to the same shape whenever possible.

This useful (but sometimes confusing) feature in NumPy is called **broadcasting**.

The value of broadcasting is that

- `for` loops can be avoided, which helps numerical code run fast and
- broadcasting can allow us to implement operations on arrays without actually creating some dimensions of these arrays in memory, which can be important when arrays are large.

For example, suppose `a` is a 3×3 array (`a -> (3, 3)`), while `b` is a flat array with three elements (`b -> (3,)`).

When adding them together, NumPy will automatically expand `b -> (3,)` to `b -> (3, 3)`.

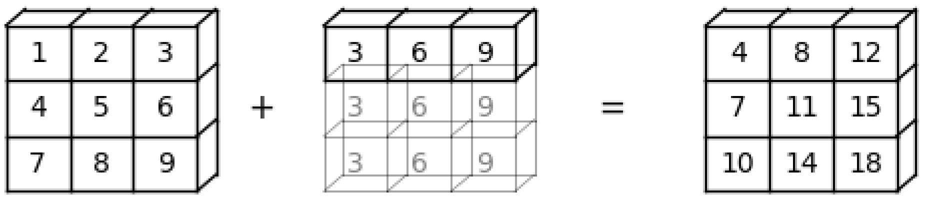
The element-wise addition will result in a 3×3 array

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

a + b
```

Out[ ]:
```
array([[ 4,  8, 12],
       [ 7, 11, 15],
       [10, 14, 18]])
```

Here is a visual representation of this broadcasting operation:

In [ ]:
```
# Adapted and modified based on the code in the book written by Jake Vand
# Originally from astroML: see http://www.astroml.org/book_figures/append

import numpy as np
from matplotlib import pyplot as plt


def draw_cube(ax, xy, size, depth=0.4,
              edges=None, label=None, label_kwargs=None, **kwargs):
    """draw and label a cube.  edges is a list of numbers between
    1 and 12, specifying which of the 12 cube edges to draw"""
    if edges is None:
        edges = range(1, 13)

    x, y = xy

    if 1 in edges:
        ax.plot([x, x + size],
                [y + size, y + size], **kwargs)
    if 2 in edges:
        ax.plot([x + size, x + size],
                [y, y + size], **kwargs)
    if 3 in edges:
        ax.plot([x, x + size],
                [y, y], **kwargs)
    if 4 in edges:
        ax.plot([x, x],
                [y, y + size], **kwargs)

    if 5 in edges:
        ax.plot([x, x + depth],
                [y + size, y + depth + size], **kwargs)
    if 6 in edges:
        ax.plot([x + size, x + size + depth],
                [y + size, y + depth + size], **kwargs)
    if 7 in edges:
        ax.plot([x + size, x + size + depth],
                [y, y + depth], **kwargs)
    if 8 in edges:
        ax.plot([x, x + depth],
                [y, y + depth], **kwargs)

    if 9 in edges:
        ax.plot([x + depth, x + depth + size],
                [y + depth + size, y + depth + size], **kwargs)
    if 10 in edges:
        ax.plot([x + depth + size, x + depth + size],
                [y + depth, y + depth + size], **kwargs)
    if 11 in edges:
        ax.plot([x + depth, x + depth + size],
                [y + depth, y + depth], **kwargs)
    if 12 in edges:
        ax.plot([x + depth, x + depth],
                [y + depth, y + depth + size], **kwargs)

    if label:
        if label_kwargs is None:
            label_kwargs = {}
        ax.text(x + 0.5 * size, y + 0.5 * size, label,
                ha='center', va='center', **label_kwargs)

solid = dict(c='black', ls='-', lw=1,
             label_kwargs=dict(color='k'))
dotted = dict(c='black', ls='-', lw=0.5, alpha=0.5,
              label_kwargs=dict(color='gray'))
depth = 0.3

# Draw a figure and axis with no boundary
fig = plt.figure(figsize=(5, 1), facecolor='w')
ax = plt.axes([0, 0, 1, 1], xticks=[], yticks=[], frameon=False)

# first block
draw_cube(ax, (1, 7.5), 1, depth, [1, 2, 3, 4, 5, 6, 9], '1', **solid)
draw_cube(ax, (2, 7.5), 1, depth, [1, 2, 3, 6, 9], '2', **solid)
draw_cube(ax, (3, 7.5), 1, depth, [1, 2, 3, 6, 7, 9, 10], '3', **solid)

draw_cube(ax, (1, 6.5), 1, depth, [2, 3, 4], '4', **solid)
draw_cube(ax, (2, 6.5), 1, depth, [2, 3], '5', **solid)
draw_cube(ax, (3, 6.5), 1, depth, [2, 3, 7, 10], '6', **solid)

draw_cube(ax, (1, 5.5), 1, depth, [2, 3, 4], '7', **solid)
draw_cube(ax, (2, 5.5), 1, depth, [2, 3], '8', **solid)
draw_cube(ax, (3, 5.5), 1, depth, [2, 3, 7, 10], '9', **solid)

# second block
draw_cube(ax, (6, 7.5), 1, depth, [1, 2, 3, 4, 5, 6, 9], '3', **solid)
draw_cube(ax, (7, 7.5), 1, depth, [1, 2, 3, 6, 9], '6', **solid)
draw_cube(ax, (8, 7.5), 1, depth, [1, 2, 3, 6, 7, 9, 10], '9', **solid)

draw_cube(ax, (6, 6.5), 1, depth, range(2, 13), '3', **dotted)
draw_cube(ax, (7, 6.5), 1, depth, [2, 3, 6, 7, 9, 10, 11], '6', **dotted)
draw_cube(ax, (8, 6.5), 1, depth, [2, 3, 6, 7, 9, 10, 11], '9', **dotted)

draw_cube(ax, (6, 5.5), 1, depth, [2, 3, 4, 7, 8, 10, 11, 12], '3', **dot
draw_cube(ax, (7, 5.5), 1, depth, [2, 3, 7, 10, 11], '6', **dotted)
draw_cube(ax, (8, 5.5), 1, depth, [2, 3, 7, 10, 11], '9', **dotted)

# third block
draw_cube(ax, (12, 7.5), 1, depth, [1, 2, 3, 4, 5, 6, 9], '4', **solid)
draw_cube(ax, (13, 7.5), 1, depth, [1, 2, 3, 6, 9], '8', **solid)
draw_cube(ax, (14, 7.5), 1, depth, [1, 2, 3, 6, 7, 9, 10], '12', **solid)

draw_cube(ax, (12, 6.5), 1, depth, [2, 3, 4], '7', **solid)
draw_cube(ax, (13, 6.5), 1, depth, [2, 3], '11', **solid)
draw_cube(ax, (14, 6.5), 1, depth, [2, 3, 7, 10], '15', **solid)

draw_cube(ax, (12, 5.5), 1, depth, [2, 3, 4], '10', **solid)
draw_cube(ax, (13, 5.5), 1, depth, [2, 3], '14', **solid)
draw_cube(ax, (14, 5.5), 1, depth, [2, 3, 7, 10], '18', **solid)

ax.text(5, 7.0, '+', size=12, ha='center', va='center')
ax.text(10.5, 7.0, '=', size=12, ha='center', va='center');
```



How about `b -> (3, 1)`?

In this case, NumPy will automatically expand `b -> (3, 1)` to `b -> (3, 3)`.

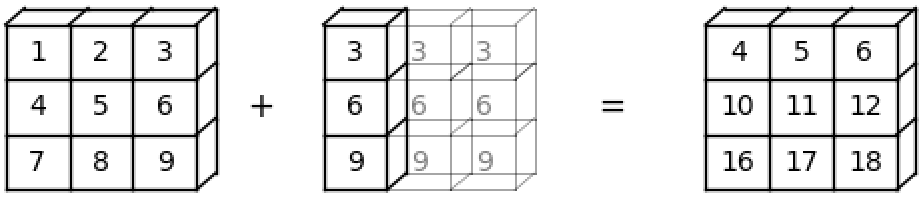
Element-wise addition will then result in a 3×3 matrix

In [ ]:
```
b.shape = (3, 1)

a + b
```

Out[ ]:
```
array([[ 4,  5,  6],
       [10, 11, 12],
       [16, 17, 18]])
```

Here is a visual representation of this broadcasting operation:

In [ ]:
```
fig = plt.figure(figsize=(5, 1), facecolor='w')
ax = plt.axes([0, 0, 1, 1], xticks=[], yticks=[], frameon=False)

# first block
draw_cube(ax, (1, 7.5), 1, depth, [1, 2, 3, 4, 5, 6, 9], '1', **solid)
draw_cube(ax, (2, 7.5), 1, depth, [1, 2, 3, 6, 9], '2', **solid)
draw_cube(ax, (3, 7.5), 1, depth, [1, 2, 3, 6, 7, 9, 10], '3', **solid)

draw_cube(ax, (1, 6.5), 1, depth, [2, 3, 4], '4', **solid)
draw_cube(ax, (2, 6.5), 1, depth, [2, 3], '5', **solid)
draw_cube(ax, (3, 6.5), 1, depth, [2, 3, 7, 10], '6', **solid)

draw_cube(ax, (1, 5.5), 1, depth, [2, 3, 4], '7', **solid)
draw_cube(ax, (2, 5.5), 1, depth, [2, 3], '8', **solid)
draw_cube(ax, (3, 5.5), 1, depth, [2, 3, 7, 10], '9', **solid)

# second block
draw_cube(ax, (6, 7.5), 1, depth, [1, 2, 3, 4, 5, 6, 7, 9, 10], '3', **so
draw_cube(ax, (7, 7.5), 1, depth, [1, 2, 3, 6, 7, 9, 10], '3', **dotted)
draw_cube(ax, (8, 7.5), 1, depth, [1, 2, 3, 6, 7, 9, 10], '3', **dotted)

draw_cube(ax, (6, 6.5), 1, depth, [2, 3, 4, 7, 10], '6', **solid)
draw_cube(ax, (7, 6.5), 1, depth, [2, 3, 6, 7, 9, 10, 11], '6', **dotted)
draw_cube(ax, (8, 6.5), 1, depth, [2, 3, 6, 7, 9, 10, 11], '6', **dotted)

draw_cube(ax, (6, 5.5), 1, depth, [2, 3, 4, 7, 10], '9', **solid)
draw_cube(ax, (7, 5.5), 1, depth, [2, 3, 7, 10, 11], '9', **dotted)
draw_cube(ax, (8, 5.5), 1, depth, [2, 3, 7, 10, 11], '9', **dotted)

# third block
draw_cube(ax, (12, 7.5), 1, depth, [1, 2, 3, 4, 5, 6, 9], '4', **solid)
draw_cube(ax, (13, 7.5), 1, depth, [1, 2, 3, 6, 9], '5', **solid)
draw_cube(ax, (14, 7.5), 1, depth, [1, 2, 3, 6, 7, 9, 10], '6', **solid)

draw_cube(ax, (12, 6.5), 1, depth, [2, 3, 4], '10', **solid)
draw_cube(ax, (13, 6.5), 1, depth, [2, 3], '11', **solid)
draw_cube(ax, (14, 6.5), 1, depth, [2, 3, 7, 10], '12', **solid)

draw_cube(ax, (12, 5.5), 1, depth, [2, 3, 4], '16', **solid)
draw_cube(ax, (13, 5.5), 1, depth, [2, 3], '17', **solid)
draw_cube(ax, (14, 5.5), 1, depth, [2, 3, 7, 10], '18', **solid)

ax.text(5, 7.0, '+', size=12, ha='center', va='center')
ax.text(10.5, 7.0, '=', size=12, ha='center', va='center');
```



The previous broadcasting operation is equivalent to the following `for` loop

In [ ]:
```
row, column = a.shape
result = np.empty((3, 3))
for i in range(row):
    for j in range(column):
        result[i, j] = a[i, j] + b[i]

result
```

Out[ ]:
```
array([[ 4.,  5.,  6.],
       [10., 11., 12.],
       [16., 17., 18.]])
```

In some cases, both operands will be expanded.

When we have `a -> (3,)` and `b -> (3, 1)`, `a` will be expanded to `a -> (3, 3)`, and `b` will be expanded to `b -> (3, 3)`.

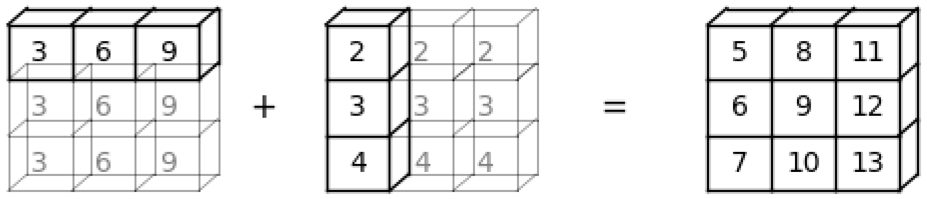
In this case, element-wise addition will result in a 3×3 matrix

In [ ]:
```
a = np.array([3, 6, 9])
b = np.array([2, 3, 4])
b.shape = (3, 1)

a + b
```

```
Out[ ]: array([[ 5,  8, 11],
               [ 6,  9, 12],
               [ 7, 10, 13]])
```

Here is a visual representation of this broadcasting operation:

```
In [ ]: # Draw a figure and axis with no boundary
fig = plt.figure(figsize=(5, 1), facecolor='w')
ax = plt.axes([0, 0, 1, 1], xticks=[], yticks=[], frameon=False)

# first block
draw_cube(ax, (1, 7.5), 1, depth, [1, 2, 3, 4, 5, 6, 9], '3', **solid)
draw_cube(ax, (2, 7.5), 1, depth, [1, 2, 3, 6, 9], '6', **solid)
draw_cube(ax, (3, 7.5), 1, depth, [1, 2, 3, 6, 7, 9, 10], '9', **solid)

draw_cube(ax, (1, 6.5), 1, depth, range(2, 13), '3', **dotted)
draw_cube(ax, (2, 6.5), 1, depth, [2, 3, 6, 7, 9, 10, 11], '6', **dotted)
draw_cube(ax, (3, 6.5), 1, depth, [2, 3, 6, 7, 9, 10, 11], '9', **dotted)

draw_cube(ax, (1, 5.5), 1, depth, [2, 3, 4, 7, 8, 10, 11, 12], '3', **dot
draw_cube(ax, (2, 5.5), 1, depth, [2, 3, 7, 10, 11], '6', **dotted)
draw_cube(ax, (3, 5.5), 1, depth, [2, 3, 7, 10, 11], '9', **dotted)

# second block
draw_cube(ax, (6, 7.5), 1, depth, [1, 2, 3, 4, 5, 6, 7, 9, 10], '2', **so
draw_cube(ax, (7, 7.5), 1, depth, [1, 2, 3, 6, 7, 9, 10], '2', **dotted)
draw_cube(ax, (8, 7.5), 1, depth, [1, 2, 3, 6, 7, 9, 10], '2', **dotted)

draw_cube(ax, (6, 6.5), 1, depth, [2, 3, 4, 7, 10], '3', **solid)
draw_cube(ax, (7, 6.5), 1, depth, [2, 3, 6, 7, 9, 10, 11], '3', **dotted)
draw_cube(ax, (8, 6.5), 1, depth, [2, 3, 6, 7, 9, 10, 11], '3', **dotted)

draw_cube(ax, (6, 5.5), 1, depth, [2, 3, 4, 7, 10], '4', **solid)
draw_cube(ax, (7, 5.5), 1, depth, [2, 3, 7, 10, 11], '4', **dotted)
draw_cube(ax, (8, 5.5), 1, depth, [2, 3, 7, 10, 11], '4', **dotted)

# third block
draw_cube(ax, (12, 7.5), 1, depth, [1, 2, 3, 4, 5, 6, 9], '5', **solid)
draw_cube(ax, (13, 7.5), 1, depth, [1, 2, 3, 6, 9], '8', **solid)
draw_cube(ax, (14, 7.5), 1, depth, [1, 2, 3, 6, 7, 9, 10], '11', **solid)

draw_cube(ax, (12, 6.5), 1, depth, [2, 3, 4], '6', **solid)
draw_cube(ax, (13, 6.5), 1, depth, [2, 3], '9', **solid)
draw_cube(ax, (14, 6.5), 1, depth, [2, 3, 7, 10], '12', **solid)

draw_cube(ax, (12, 5.5), 1, depth, [2, 3, 4], '7', **solid)
draw_cube(ax, (13, 5.5), 1, depth, [2, 3], '10', **solid)
draw_cube(ax, (14, 5.5), 1, depth, [2, 3, 7, 10], '13', **solid)

ax.text(5, 7.0, '+', size=12, ha='center', va='center')
ax.text(10.5, 7.0, '=', size=12, ha='center', va='center');
```



While broadcasting is very useful, it can sometimes seem confusing.

For example, let's try adding `a -> (3, 2)` and `b -> (3,)`.

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

a + b
```

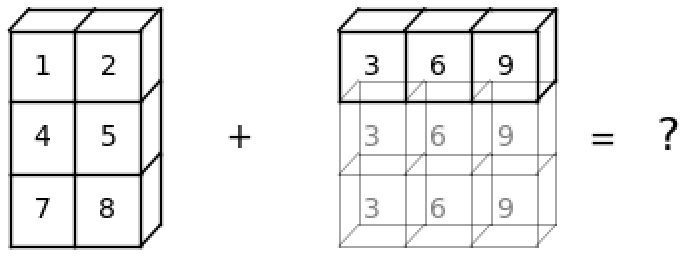
```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
/Users/cheney_gao/Desktop/Intermediate Macroeconomics/numpy.ipynb Cell 113
line 7
      <a href='vscode-notebook-cell:/Users/cheney_gao/Desktop/Intermediat
e%20Macroeconomics/numpy.ipynb#Y221sZmlsZQ%3D%3D?line=0'>1</a> a = np.arra
y(
      <a href='vscode-notebook-cell:/Users/cheney_gao/Desktop/Intermediat
e%20Macroeconomics/numpy.ipynb#Y221sZmlsZQ%3D%3D?line=1'>2</a>       [[1,
2],
      <a href='vscode-notebook-cell:/Users/cheney_gao/Desktop/Intermediat
e%20Macroeconomics/numpy.ipynb#Y221sZmlsZQ%3D%3D?line=2'>3</a>        [4,
5],
      <a href='vscode-notebook-cell:/Users/cheney_gao/Desktop/Intermediat
e%20Macroeconomics/numpy.ipynb#Y221sZmlsZQ%3D%3D?line=3'>4</a>        [7,
8]])
      <a href='vscode-notebook-cell:/Users/cheney_gao/Desktop/Intermediat
e%20Macroeconomics/numpy.ipynb#Y221sZmlsZQ%3D%3D?line=4'>5</a> b = np.arra
y([3, 6, 9])
----> <a href='vscode-notebook-cell:/Users/cheney_gao/Desktop/Intermediat
e%20Macroeconomics/numpy.ipynb#Y221sZmlsZQ%3D%3D?line=6'>7</a> a + b

ValueError: operands could not be broadcast together with shapes (3,2)
(3,)
```

The `ValueError` tells us that operands could not be broadcast together.

Here is a visual representation to show why this broadcasting cannot be executed:

In [ ]:
```
# Draw a figure and axis with no boundary
fig = plt.figure(figsize=(3, 1.3), facecolor='w')
ax = plt.axes([0, 0, 1, 1], xticks=[], yticks=[], frameon=False)

# first block
draw_cube(ax, (1, 7.5), 1, depth, [1, 2, 3, 4, 5, 6, 9], '1', **solid)
draw_cube(ax, (2, 7.5), 1, depth, [1, 2, 3, 6, 7, 9, 10], '2', **solid)

draw_cube(ax, (1, 6.5), 1, depth, [2, 3, 4], '4', **solid)
draw_cube(ax, (2, 6.5), 1, depth, [2, 3, 7, 10], '5', **solid)

draw_cube(ax, (1, 5.5), 1, depth, [2, 3, 4], '7', **solid)
draw_cube(ax, (2, 5.5), 1, depth, [2, 3, 7, 10], '8', **solid)

# second block
```

```
draw_cube(ax, (6, 7.5), 1, depth, [1, 2, 3, 4, 5, 6, 9], '3', **solid)
draw_cube(ax, (7, 7.5), 1, depth, [1, 2, 3, 6, 9], '6', **solid)
draw_cube(ax, (8, 7.5), 1, depth, [1, 2, 3, 6, 7, 9, 10], '9', **solid)

draw_cube(ax, (6, 6.5), 1, depth, range(2, 13), '3', **dotted)
draw_cube(ax, (7, 6.5), 1, depth, [2, 3, 6, 7, 9, 10, 11], '6', **dotted)
draw_cube(ax, (8, 6.5), 1, depth, [2, 3, 6, 7, 9, 10, 11], '9', **dotted)

draw_cube(ax, (6, 5.5), 1, depth, [2, 3, 4, 7, 8, 10, 11, 12], '3', **dot
draw_cube(ax, (7, 5.5), 1, depth, [2, 3, 7, 10, 11], '6', **dotted)
draw_cube(ax, (8, 5.5), 1, depth, [2, 3, 7, 10, 11], '9', **dotted)


ax.text(4.5, 7.0, '+', size=12, ha='center', va='center')
ax.text(10, 7.0, '=', size=12, ha='center', va='center')
ax.text(11, 7.0, '?', size=16, ha='center', va='center');
```



We can see that NumPy cannot expand the arrays to the same size.

It is because, when `b` is expanded from `b -> (3,)` to `b -> (3, 3)`, NumPy cannot match `b` with `a -> (3, 2)`.

Things get even trickier when we move to higher dimensions.

To help us, we can use the following list of rules:

- *Step 1:* When the dimensions of two arrays do not match, NumPy will expand the one with fewer dimensions by adding dimension(s) on the left of the existing dimensions.
  - For example, if `a -> (3, 3)` and `b -> (3,)`, then broadcasting will add a dimension to the left so that `b -> (1, 3)`;
  - If `a -> (2, 2, 2)` and `b -> (2, 2)`, then broadcasting will add a dimension to the left so that `b -> (1, 2, 2)`;
  - If `a -> (3, 2, 2)` and `b -> (2,)`, then broadcasting will add two dimensions to the left so that `b -> (1, 1, 2)` (you can also see this process as going through *Step 1* twice).
- *Step 2:* When the two arrays have the same dimension but different shapes, NumPy will try to expand dimensions where the shape index is 1.
  - For example, if `a -> (1, 3)` and `b -> (3, 1)`, then broadcasting will expand dimensions with shape 1 in both `a` and `b` so that `a -> (3, 3)` and `b -> (3, 3)`;
  - If `a -> (2, 2, 2)` and `b -> (1, 2, 2)`, then broadcasting will expand the first dimension of `b` so that `b -> (2, 2, 2)`;

- If `a -> (3, 2, 2)` and `b -> (1, 1, 2)`, then broadcasting will expand `b` on all dimensions with shape 1 so that `b -> (3, 2, 2)`.

Here are code examples for broadcasting higher dimensional arrays

```
In [ ]: # a -> (2, 2, 2) and  b -> (1, 2, 2)

        a = np.array(
            [[[1, 2],
              [2, 3]],

             [[2, 3],
              [3, 4]]])
        print(f'the shape of array a is {a.shape}')

        b = np.array(
            [[1,7],
             [7,1]])
        print(f'the shape of array b is {b.shape}')

        a + b
```

```
the shape of array a is (2, 2, 2)
the shape of array b is (2, 2)
```

```
Out[ ]: array([[[ 2,  9],
                [ 9,  4]],

               [[ 3, 10],
                [10,  5]]])
```

```
In [ ]: # a -> (3, 2, 2) and b -> (2,)

        a = np.array(
            [[[1, 2],
              [3, 4]],

             [[4, 5],
              [6, 7]],

             [[7, 8],
              [9, 10]]])
        print(f'the shape of array a is {a.shape}')

        b = np.array([3, 6])
        print(f'the shape of array b is {b.shape}')

        a + b
```

```
the shape of array a is (3, 2, 2)
the shape of array b is (2,)
```

```
Out[ ]: array([[[ 4,  8],
                [ 6, 10]],

               [[ 7, 11],
                [ 9, 13]],

               [[10, 14],
                [12, 16]]])
```

- *Step 3:* After Step 1 and 2, if the two arrays still do not match, a `ValueError` will be raised. For example, suppose `a -> (2, 2, 3)` and `b -> (2, 2)`
  - By *Step 1*, `b` will be expanded to `b -> (1, 2, 2)`;
  - By *Step 2*, `b` will be expanded to `b -> (2, 2, 2)`;
  - We can see that they do not match each other after the first two steps. Thus, a `ValueError` will be raised

In [ ]:
```
a = np.array(
    [[[1, 2, 3],
      [2, 3, 4]],
     
     [[2, 3, 4],
      [3, 4, 5]]])
print(f'the shape of array a is {a.shape}')

b = np.array(
    [[1,7],
     [7,1]])
print(f'the shape of array b is {b.shape}')

a + b
```

```
the shape of array a is (2, 2, 3)
the shape of array b is (2, 2)
```

```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
/Users/cheney_gao/Desktop/Intermediate Macroeconomics/numpy.ipynb Cell 120
line 1
      <a href='vscode-notebook-cell:/Users/cheney_gao/Desktop/Intermediat
e%20Macroeconomics/numpy.ipynb#Y231sZmlsZQ%3D%3D?line=8'>9</a> b = np.arra
y(
     <a href='vscode-notebook-cell:/Users/cheney_gao/Desktop/Intermediate%
20Macroeconomics/numpy.ipynb#Y231sZmlsZQ%3D%3D?line=9'>10</a>     [[1,7],
     <a href='vscode-notebook-cell:/Users/cheney_gao/Desktop/Intermediate%
20Macroeconomics/numpy.ipynb#Y231sZmlsZQ%3D%3D?line=10'>11</a>      [7,
1]])
     <a href='vscode-notebook-cell:/Users/cheney_gao/Desktop/Intermediate%
20Macroeconomics/numpy.ipynb#Y231sZmlsZQ%3D%3D?line=11'>12</a> print(f'the
shape of array b is {b.shape}')
---> <a href='vscode-notebook-cell:/Users/cheney_gao/Desktop/Intermediate%
20Macroeconomics/numpy.ipynb#Y231sZmlsZQ%3D%3D?line=13'>14</a> a + b

ValueError: operands could not be broadcast together with shapes (2,2,3)
(2,2)
```

Mutability and Copying Arrays

NumPy arrays are mutable data types, like Python lists.

In other words, their contents can be altered (mutated) in memory after initialization.

We already saw examples above.

Here's another example:

```
In [ ]: a = np.array([42, 44])
        a
```

```
Out[ ]: array([42, 44])
```

```
In [ ]: a[-1] = 0  # Change last element to 0
        a
```

```
Out[ ]: array([42,  0])
```

Mutability leads to the following behavior (which can be shocking to MATLAB programmers…)

```
In [ ]: a = np.random.randn(3)
        a
```

```
Out[ ]: array([ 1.68588937,  0.25933835, -0.21873682])
```

```
In [ ]: b = a
        b[0] = 0.0
        a
```

```
Out[ ]: array([ 0.        ,  0.25933835, -0.21873682])
```

What's happened is that we have changed `a` by changing `b`.

The name `b` is bound to `a` and becomes just another reference to the array

Hence, it has equal rights to make changes to that array.

This is in fact the most sensible default behavior!

It means that we pass around only pointers to data, rather than making copies.

Making copies is expensive in terms of both speed and memory.

Making Copies

It is of course possible to make `b` an independent copy of `a` when required.

This can be done using `np.copy`

```
In [ ]: a = np.random.randn(3)
        a
```

```
Out[ ]: array([-0.86338429,  0.60952498, -0.17400361])
```

```
In [ ]: b = np.copy(a)
        b
```

```
Out[ ]: array([-0.86338429,  0.60952498, -0.17400361])
```

Now `b` is an independent copy (called a *deep copy*)

```
In [ ]: b[:] = 1
        b
```

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

```
In [ ]: a
```

```
Out[ ]: array([-0.86338429,  0.60952498, -0.17400361])
```

Note that the change to `b` has not affected `a`.

Additional Functionality

Let’s look at some other useful things we can do with NumPy.

Vectorized Functions

NumPy provides versions of the standard functions `log`, `exp`, `sin`, etc. that act *element-wise* on arrays

```
In [ ]: z = np.array([1, 2, 3])
        np.sin(z)
```

```
Out[ ]: array([0.84147098, 0.90929743, 0.14112001])
```

This eliminates the need for explicit element-by-element loops such as

```
In [ ]: n = len(z)
        y = np.empty(n)
        for i in range(n):
            y[i] = np.sin(z[i])
```

Because they act element-wise on arrays, these functions are called *vectorized functions*.

In NumPy-speak, they are also called *ufuncs*, which stands for “universal functions”.

As we saw above, the usual arithmetic operations (`+`, `*`, etc.) also work element-wise, and combining these with the ufuncs gives a very large set of fast element-wise functions.

```
In [ ]: z
```

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

```
In [ ]: (1 / np.sqrt(2 * np.pi)) * np.exp(- 0.5 * z**2)
```

```
Out[ ]: array([0.24197072, 0.05399097, 0.00443185])
```

Not all user-defined functions will act element-wise.

For example, passing the function `f` defined below a NumPy array causes a `ValueError`

```
In [ ]: def f(x):
            return 1 if x > 0 else 0
```

The NumPy function `np.where` provides a vectorized alternative:

```
In [ ]: x = np.random.randn(4)
        x

Out[ ]: array([ 2.172564  , -0.85934724,  0.42405473,  0.21113942])
```

```
In [ ]: np.where(x > 0, 1, 0)  # Insert 1 if x > 0 true, otherwise 0

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

You can also use `np.vectorize` to vectorize a given function

```
In [ ]: f = np.vectorize(f)
        f(x)                # Passing the same vector x as in the previous exampl

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

However, this approach doesn’t always obtain the same speed as a more carefully crafted vectorized function.

Comparisons

As a rule, comparisons on arrays are done element-wise

```
In [ ]: z = np.array([2, 3])
        y = np.array([2, 3])
        z == y

Out[ ]: array([ True,  True])
```

```
In [ ]: y[0] = 5
        z == y

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

```
In [ ]: z != y

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

The situation is similar for `>`, `<`, `>=` and `<=`.

We can also do comparisons against scalars

```
In [ ]: z = np.linspace(0, 10, 5)
        z
```

```
Out[ ]: array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

```
In [ ]: z > 3
```

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

This is particularly useful for *conditional extraction*

```
In [ ]: b = z > 3
        b
```

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

```
In [ ]: z[b]
```

```
Out[ ]: array([ 5. ,  7.5, 10. ])
```

Of course we can—and frequently do—perform this in one step

```
In [ ]: z[z > 3]
```

```
Out[ ]: array([ 5. ,  7.5, 10. ])
```

Sub-packages

NumPy provides some additional functionality related to scientific programming through its sub-packages.

We've already seen how we can generate random variables using np.random

```
In [ ]: z = np.random.randn(10000)  # Generate standard normals
        y = np.random.binomial(10, 0.5, size=1000)    # 1,000 draws from Bin(10,
        y.mean()
```

```
Out[ ]: 5.011
```

Another commonly used subpackage is np.linalg

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

        np.linalg.det(A)           # Compute the determinant
```

```
Out[ ]: -2.0000000000000004
```

```
In [ ]: np.linalg.inv(A)           # Compute the inverse
```

```
Out[ ]: array([[-2. ,  1. ],
               [ 1.5, -0.5]])
```