Using the NumPy Package

Chris Cornwell

September 4, 2025

Outline

Outline

Getting started with NumPy

Not built-in – must import NumPy into Python session. We will also want to track runtime, so we import the package `time`.

Getting started with NumPy

Not built-in – must import NumPy into Python session. We will also want to track runtime, so we import the package `time`.

```python
import numpy as np
import time
```

Create a shortcut, `np`, for NumPy. This is a common convention.

Getting started with NumPy

Not built-in – must import NumPy into Python session. We will also want to track runtime, so we import the package `time`.

```python
import numpy as np
import time
```

Create a shortcut, `np`, for NumPy. This is a common convention.

- Depending on how you are interacting with Python, may have to *install* the `numpy` package before the first use. Open a command terminal (`Ctrl+``, in VSCode on Windows) and type the appropriate command below:

  `py -m pip install numpy` (Windows)

  `python3-m pip install numpy` (macOS)

  `sudo pip install numpy` (Linux based)

Getting started with NumPy

Not built-in – must import NumPy into Python session. We will also want to track runtime, so we import the package `time`.

```
import numpy as np
import time
```

Create a shortcut, `np`, for NumPy. This is a common convention.

- Depending on how you are interacting with Python, may have to *install* the `numpy` package before the first use. Open a command terminal (`Ctrl+``, in VSCode on Windows) and type the appropriate command below:
  `py -m pip install numpy` (Windows)
  `python3-m pip install numpy` (macOS)
  `sudo pip install numpy` (Linux based)

When installing other packages, replace `numpy` with the package name.

Outline

Basic NumPy arrays

The main type of object in NumPy is the `ndarray` (n-dimensional array), which is constructed from a list using the command

`np.array(the_list)`.

Basic NumPy arrays

The main type of object in NumPy is the `ndarray` (n-dimensional array), which is constructed from a list using the command

`np.array(the_list)`.

If items in `the_list` are of numeric type, then think of the resulting `ndarray` as like a vector. Operations on NumPy arrays work like vectors in linear algebra.

Basic NumPy arrays

The main type of object in NumPy is the `ndarray` (n-dimensional array), which is constructed from a list using the command

`np.array(the_list)`.

If items in `the_list` are of numeric type, then think of the resulting `ndarray` as like a vector. Operations on NumPy arrays work like vectors in linear algebra.

Example:

```
v = np.array([-1, 1, 1])
w = np.array([0.5, 0, 1.1])
# print the (vector) sum: [-0.5  1.   2.1]
print(v + w)
# prints [1.0, 0.0, 2.2]
print(2*w)
```

More than `1d`

A NumPy array from a list containing numeric types makes a vector – also known as a 1-*dimensional array* (Python language), or a *tensor of order* 1 (mathematics).

More than 1d

A NumPy array from a list containing numeric types makes a vector – also known as a 1-*dimensional array* (Python language), or a *tensor of order* 1 (mathematics).

A 2-dimensional array, or tensor of order 2, is like a matrix. You construct it with `np.array( )` from a list of lists – each of the same length.

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

Each “inside list” is a row. The array A is a 2×3 matrix.

More than 1d

A NumPy array from a list containing numeric types makes a vector – also known as a 1-*dimensional array* (Python language), or a *tensor of order* 1 (mathematics).

A 2-dimensional array, or tensor of order 2, is like a matrix. You construct it with `np.array( )` from a list of lists – each of the same length.

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

Each "inside list" is a row. The array A is a 2×3 matrix.

Every array in NumPy has an attribute `shape`.

- Previous slide: `v = np.array([-1,1,1])` has `v.shape = (3,)`.
- The matrix here: `A.shape` is equal to `(2, 3)`.

Operations on arrays

Multiplying two arrays: most recent version of Python uses the `@` symbol.[1]

When the arrays are both matrices, it computes their matrix product; when one is a vector, it computes the matrix-vector product; when both are vectors, it computes the dot product.

[1] In older versions, matrix multiplication is `np.matmul()` and dot product is `np.dot()`.

Operations on arrays

Multiplying two arrays: most recent version of Python uses the @ symbol.[1]

When the arrays are both matrices, it computes their matrix product; when one is a vector, it computes the matrix-vector product; when both are vectors, it computes the dot product.

For example, say that A is the matrix $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ from before, v is the vector $(-1, 1, 1)$, and let B and u be the matrix and vector defined in the code below.

```
B = np.array([[1, 0], [1, -1], [1, 1]])
u = np.array([1, 1, 0])
(A @ B, A @ v, v @ u)
```

[1] In older versions, matrix multiplication is `np.matmul()` and dot product is `np.dot()`.

Operations on arrays

Multiplying two arrays: most recent version of Python uses the @ symbol.[1]

When the arrays are both matrices, it computes their matrix product; when one is a vector, it computes the matrix-vector product; when both are vectors, it computes the dot product.

For example, say that A is the matrix $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ from before, v is the vector $(-1, 1, 1)$, and let B and u be the matrix and vector defined in the code below.

```
B = np.array([[1, 0], [1, -1], [1, 1]])
u = np.array([1, 1, 0])
(A @ B, A @ v, v @ u)
```

Output: (`array([[6, 1], [15, 1]]), array([4, 7]), 0`).

[1]In older versions, matrix multiplication is `np.matmul()` and dot product is `np.dot()`.

Indexing and slicing arrays

Items in 1d array are accessed the same way as in a list

e.g., `v[0]` is first item in `v`, at index 0.

For a 2d array, say the matrix `A`, we can access the item in the row `i` and column `j` by `A[i, j]`.

[2] The colon here says to take all indices in that position.

[3] First position: “get first 3 indices”; Second position: “last 3 indices”.

Indexing and slicing arrays

Items in 1d array are accessed the same way as in a list

e.g., `v[0]` is first item in `v`, at index 0.

For a 2d array, say the matrix `A`, we can access the item in the row `i` and column `j` by `A[i, j]`.

"Slicing" operation for arrays (also works with lists). Examples:

1. Typing `A[: , 0]` will give first column of the matrix.[2]

[2] The colon here says to take all indices in that position.

[3] First position: "get first 3 indices"; Second position: "last 3 indices".

Indexing and slicing arrays

Items in 1d array are accessed the same way as in a list

e.g., `v[0]` is first item in `v`, at index 0.

For a 2d array, say the matrix `A`, we can access the item in the row `i` and column `j` by `A[i, j]`.

"Slicing" operation for arrays (also works with lists). Examples:

1. Typing `A[: , 0]` will give first column of the matrix.[2]
2. By typing `M[:3, -3:]`, can get top-right 3×3 submatrix of `M`.[3]

[2]The colon here says to take all indices in that position.

[3]First position: "get first 3 indices"; Second position: "last 3 indices".

Indexing and slicing arrays

Items in 1d array are accessed the same way as in a list

e.g., `v[0]` is first item in `v`, at index 0.

For a 2d array, say the matrix `A`, we can access the item in the row `i` and column `j` by `A[i, j]`.

"Slicing" operation for arrays (also works with lists). Examples:

1. Typing `A[: , 0]` will give first column of the matrix.[2]
2. By typing `M[:3, -3:]`, can get top-right 3×3 submatrix of `M`.[3]

With arrays (not lists), can even use non-consecutive indices; *e.g.*, `A[: ,[0,2]]` gets two columns of `A` that are not adjacent.

[2]The colon here says to take all indices in that position.

[3]First position: "get first 3 indices"; Second position: "last 3 indices".

Indexing and slicing arrays

Items in 1d array are accessed the same way as in a list

e.g., `v[0]` is first item in `v`, at index 0.

For a 2d array, say the matrix `A`, we can access the item in the row `i` and column `j` by `A[i, j]`.

"Slicing" operation for arrays (also works with lists). Examples:

1. Typing `A[: , 0]` will give first column of the matrix.[2]
2. By typing `M[:3, -3:]`, can get top-right 3×3 submatrix of `M`.[3]

With arrays (not lists), can even use non-consecutive indices; *e.g.*, `A[: ,[0,2]]` gets two columns of `A` that are not adjacent.

If `A` is a 2d array, its transpose is `A.T`.

[2] The colon here says to take all indices in that position.

[3] First position: "get first 3 indices"; Second position: "last 3 indices".

Outline

Constructing special matrices

Some types of matrices are used a lot; would be cumbersome to always contruct row lists ourselves (*e.g.*, in 100×100 matrix).

Constructing special matrices

Some types of matrices are used a lot; would be cumbersome to always contruct row lists ourselves (*e.g.*, in 100×100 matrix).

> **Zero matrix:** The command `np.zeros((m, n))` constructs an $m \times n$ matrix with all entries equal to zero.

Constructing special matrices

Some types of matrices are used a lot; would be cumbersome to always contruct row lists ourselves (*e.g.*, in 100×100 matrix).

Zero matrix: The command `np.zeros((m, n))` constructs an $m \times n$ matrix with all entries equal to zero.

Diagonal matrix: If `d` is a 1D array of length n, the command `np.diag(d)` constructs an $n \times n$ diagonal matrix with `d` as its diagonal entries.

Constructing special matrices

Some types of matrices are used a lot; would be cumbersome to always contruct row lists ourselves (*e.g.*, in 100×100 matrix).

Zero matrix: The command `np.zeros((m, n))` constructs an $m \times n$ matrix with all entries equal to zero.

Diagonal matrix: If `d` is a 1D array of length n, the command `np.diag(d)` constructs an $n \times n$ diagonal matrix with `d` as its diagonal entries.

Identity matrix: The command `np.identity(n)` constructs the $n \times n$ identity matrix (and `np.eye(n)` does also).

Using NumPy for linear algebra

In addition to the product operations on arrays, NumPy has a library with many functions for linear algebra – called `linalg`.

Using NumPy for linear algebra

In addition to the product operations on arrays, NumPy has a library with many functions for linear algebra – called `linalg`.

- For example, for the determinant of a square matrix `M`, you type

  np.linalg.det(`M`)

Using NumPy for linear algebra

In addition to the product operations on arrays, NumPy has a library with many functions for linear algebra – called `linalg`.

- For example, for the determinant of a square matrix `M`, you type

  np.linalg.det(`M`)

- Many other linear algebra functions in this package (see the docs here).
  Some are only implemented with square matrices (and perhaps only invertible ones), even though it would make sense to implement them in more general settings – for example, `np.linalg.solve(A,b)` only solves the system of equations `Ax=b` if `A` is a square matrix.

If you're given a 1d array and want to get the array of square roots...

[4] Technically, there is a `for` loop in the background, but it happens in C and works much faster.

If you're given a 1d array and want to get the array of square roots...

- First thought: go through each entry in array, taking square root (and re-assigning as you go). In Python, this is not efficient.

[4] Technically, there is a `for` loop in the background, but it happens in C and works much faster.

Broadcasting, universal functions

If you're given a 1d array and want to get the array of square roots...

- First thought: go through each entry in array, taking square root (and re-assigning as you go). In Python, this is not efficient.

Efficient way in NumPy is called *broadcasting*. If `v` is the array, type

```
sqrt_v = np.sqrt(v)
```

The function `np.sqrt()` takes the square root of each entry in `v`; you don't need to write a **`for`** loop.[4]

[4] Technically, there is a **`for`** loop in the background, but it happens in C and works much faster.

Broadcasting, universal functions

If you're given a 1d array and want to get the array of square roots...

- First thought: go through each entry in array, taking square root (and re-assigning as you go). In Python, this is not efficient.

Efficient way in NumPy is called *broadcasting*. If v is the array, type

```
sqrt_v = np.sqrt(v)
```

The function `np.sqrt()` takes the square root of each entry in v; you don't need to write a `for` loop.[4]

Functions that work on arrays this way are quite common in NumPy. They are called **`ufuncs`** (universal functions).

[4]Technically, there is a `for` loop in the background, but it happens in C and works much faster.

Broadcasting, universal functions

If you're given a 1d array and want to get the array of square roots...

- First thought: go through each entry in array, taking square root (and re-assigning as you go). In Python, this is not efficient.

Efficient way in NumPy is called *broadcasting*. If v is the array, type

```
sqrt_v = np.sqrt(v)
```

The function `np.sqrt()` takes the square root of each entry in v; you don't need to write a `for` loop.[4]

Functions that work on arrays this way are quite common in NumPy. They are called **ufuncs** (universal functions). Other examples of ufuncs:

`np.abs()`, `np.sum()`, `np.maximum()`, `np.minimum()`, `np.exp()`, `np.log()`.

[4] Technically, there is a `for` loop in the background, but it happens in C and works much faster.

More on broadcasting

Many basic operations with NumPy arrays also use broadcasting. Here are a few examples with an array `v`.

1. To add a number, say `1.2`, to every array entry: type `v+1.2`.
2. To multiply every entry by `1.2`: type `1.2*v`.

[5] In mathematics, this product on vectors is called the Hadamard product.

More on broadcasting

Many basic operations with NumPy arrays also use broadcasting. Here are a few examples with an array `v`.

1. To add a number, say `1.2`, to every array entry: type `v+1.2`.
2. To multiply every entry by `1.2`: type `1.2*v`.
3. To square every entry: type `v**2`.
4. To multiply `v` by another array `w`, entry-wise[5]: type `v*w`.

[5] In mathematics, this product on vectors is called the Hadamard product.

More on broadcasting

Many basic operations with NumPy arrays also use broadcasting. Here are a few examples with an array `v`.

1. To add a number, say `1.2`, to every array entry: type `v+1.2`.
2. To multiply every entry by `1.2`: type `1.2*v`.
3. To square every entry: type `v**2`.
4. To multiply `v` by another array `w`, entry-wise[5]: type `v*w`.

Everything mentioned here works just as well on matrices (2d arrays), and generally on any nd array (higher order tensors).

[5] In mathematics, this product on vectors is called the Hadamard product.

More on broadcasting

Many basic operations with NumPy arrays also use broadcasting. Here are a few examples with an array `v`.

1. To add a number, say `1.2`, to every array entry: type `v+1.2`.
2. To multiply every entry by `1.2`: type `1.2*v`.
3. To square every entry: type `v**2`.
4. To multiply `v` by another array `w`, entry-wise[5]: type `v*w`.

Everything mentioned here works just as well on matrices (2d arrays), and generally on any nd array (higher order tensors).

Exercise. Use broadcasting to create a 100×100 matrix with all non-diagonal entries equal to -1 and diagonal entries equal to $\sqrt{2} - 1$.

[5] In mathematics, this product on vectors is called the Hadamard product.

Experiment with runtime for universal function

To check the efficiency of broadcasting, use the `time` package. Beforehand, make sure that you imported both `numpy` and `time` (see slide in first section).

Experiment with runtime for universal function

To check the efficiency of broadcasting, use the `time` package. Beforehand, make sure that you imported both `numpy` and `time` (see slide in first section).

Something simple: from a large identity matrix, we will get the exponential of the matrix (apply the function e^x to every entry).

Experiment with runtime for universal function

To check the efficiency of broadcasting, use the `time` package. Beforehand, make sure that you imported both `numpy` and `time` (see slide in first section).

Something simple: from a large identity matrix, we will get the exponential of the matrix (apply the function e^x to every entry).

First, we use a `for` loop. Run the code below in your Jupyter notebook.

```python
id_matrix = np.eye(1000)
exp_matrix = np.zeros((1000, 1000))
start = time.time()
for i in range(1000):
    for j in range(1000):
        exp_matrix[i,j] = np.exp(id_matrix[i,j])
end = time.time()
print(f"Seconds taken: {end-start}.")
```

Experiment with runtime for universal function

To check the efficiency of broadcasting, use the `time` package. Beforehand, make sure that you imported both `numpy` and `time` (see slide in first section).

Something simple: from a large identity matrix, we will get the exponential of the matrix (apply the function e^x to every entry).

First, we use a **`for`** loop. Run the code below in your Jupyter notebook.

```
id_matrix = np.eye(1000)
exp_matrix = np.zeros((1000, 1000))
start = time.time()
for i in range(1000):
    for j in range(1000):
        exp_matrix[i,j] = np.exp(id_matrix[i,j])
end = time.time()
print(f"Seconds taken: {end-start}.")
```

The output gives the number of seconds to run the computation. The exact time will vary based on your computer. Mine took around 0.55 seconds.

Experiment with runtime for universal function

Now, we will use broadcasting to compute the exponential of the identity matrix.

Experiment with runtime for universal function

Now, we will use broadcasting to compute the exponential of the identity matrix.

Run the following code in your Jupyter notebook.

```
id_matrix = np.eye(1000)
exp_matrix = np.zeros((1000, 1000))
start = time.time()
exp_matrix = np.exp(id_matrix)
end = time.time()
print(f"Seconds taken: {end-start}.")
```

Experiment with runtime for universal function

Now, we will use broadcasting to compute the exponential of the identity matrix.

Run the following code in your Jupyter notebook.

```
id_matrix = np.eye(1000)
exp_matrix = np.zeros((1000, 1000))
start = time.time()
exp_matrix = np.exp(id_matrix)
end = time.time()
print(f"Seconds taken: {end-start}.")
```

Again, the output is the number of seconds of runtime. For this approach with `np.exp()`, my computer took around 0.0045 seconds. That is over 100 times faster than using the loop!