NumPy for MATLAB users

Introduction

MATLAB® and NumPy have a lot in common, but NumPy was created to work with Python, not to be a MATLAB clone. This guide will help MATLAB users get started with NumPy.

Some key differences

In MATLAB, the basic type, even for scalars, is a multidimensional array. Array assignments in MATLAB are stored as 2D arrays of double precision floating point numbers, unless you specify the number of dimensions and type. Operations on the 2D instances of these arrays are modeled on matrix operations in linear algebra.	In NumPy, the basic type is a multidimensional `array`. Array assignments in NumPy are usually stored as n-dimensional arrays with the minimum type required to hold the objects in sequence, unless you specify the number of dimensions and type. NumPy performs operations element-by-element, so multiplying 2D arrays with `*` is not a matrix multiplication – it's an element-by-element multiplication. (The `@` operator, available since Python 3.5, can be used for conventional matrix multiplication.)
MATLAB numbers indices from 1; `a(1)` is the first element. See note INDEXING	NumPy, like Python, numbers indices from 0; `a[0]` is the first element.
MATLAB's scripting language was created for linear algebra so the syntax for some array manipulations is more compact than NumPy's. On the other hand, the API for adding GUIs and creating full-fledged applications is more or less an afterthought.	NumPy is based on Python, a general-purpose language. The advantage to NumPy is access to Python libraries including: SciPy, Matplotlib, Pandas, OpenCV, and more. In addition, Python is often embedded as a scripting language in other software, allowing NumPy to be used there too.
MATLAB array slicing uses pass-by-value semantics, with a lazy copy-on-write scheme to prevent creating copies until they are needed. Slicing operations copy parts of the array.	NumPy array slicing uses pass-by-reference, that does not copy the arguments. Slicing operations are views into an array.

Rough equivalents

The table below gives rough equivalents for some common MATLAB expressions. These are similar expressions, not equivalents. For details, see the documentation.

In the table below, it is assumed that you have executed the following commands in Python:

```
import numpy as np
from scipy import io, integrate, linalg, signal
from scipy.sparse.linalg import eigs
```

Also assume below that if the Notes talk about "matrix" that the arguments are two-dimensional entities.

General purpose equivalents

MATLAB	NumPy	Notes

MATLAB	NumPy	Notes
`help func`	`info(func)` or `help(func)` or `func?` (in IPython)	get help on the function *func*
`which func`	see note HELP	find out where *func* is defined
`type func`	`np.source(func)` or `func??` (in IPython)	print source for *func* (if not a native function)
`% comment`	`# comment`	comment a line of code with the text `comment`
`for i=1:3` `    fprintf('%i\n',i)` `end`	`for i in range(1, 4):` `   print(i)`	use a for-loop to print the numbers 1, 2, and 3 using **range**
`a && b`	`a and b`	short-circuiting logical AND operator (Python native operator); scalar arguments only
`a || b`	`a or b`	short-circuiting logical OR operator (Python native operator); scalar arguments only
`>> 4 == 4` `ans = 1` `>> 4 == 5` `ans = 0`	`>>> 4 == 4` `True` `>>> 4 == 5` `False`	The boolean objects in Python are `True` and `False`, as opposed to MATLAB logical types of `1` and `0`.
`a=4` `if a==4` `    fprintf('a = 4\n')` `elseif a==5` `    fprintf('a = 5\n')` `end`	`a = 4` `if a == 4:` `    print('a = 4')` `elif a == 5:` `    print('a = 5')`	create an if-else statement to check if `a` is 4 or 5 and print result
`1*i`, `1*j`, `1i`, `1j`	`1j`	complex numbers
`eps`	`np.finfo(float).eps` or `np.spacing(1)`	Upper bound to relative error due to rounding in 64-bit floating point arithmetic.
`load data.mat`	`io.loadmat('data.mat')`	Load MATLAB variables saved to the file `data.mat`. (Note: When saving arrays to `data.mat` in MATLAB/Octave, use a recent binary format. **`scipy.io.loadmat`** will create a dictionary with the saved arrays and further information.)

MATLAB	NumPy	Notes
`ode45`	`integrate.solve_ivp(f)`	integrate an ODE with Runge-Kutta 4,5
`ode15s`	`integrate.solve_ivp(f, method='BDF')`	integrate an ODE with BDF method

Linear algebra equivalents

MATLAB	NumPy	Notes
`ndims(a)`	`np.ndim(a)` or `a.ndim`	number of dimensions of array `a`
`numel(a)`	`np.size(a)` or `a.size`	number of elements of array `a`
`size(a)`	`np.shape(a)` or `a.shape`	“size” of array `a`
`size(a,n)`	`a.shape[n-1]`	get the number of elements of the n-th dimension of array `a`. (Note that MATLAB uses 1 based indexing while Python uses 0 based indexing, See note INDEXING)
`[ 1 2 3; 4 5 6 ]`	`np.array([[1. ,2. ,3.], [4. ,5. ,6.]])`	define a 2x3 2D array
`[ a b; c d ]`	`np.block([[a, b], [c, d]])`	construct a matrix from blocks `a`, `b`, `c`, and `d`
`a(end)`	`a[-1]`	access last element in MATLAB vector (1xn or nx1) or 1D NumPy array `a` (length n)
`a(2,5)`	`a[1, 4]`	access element in second row, fifth column in 2D array `a`
`a(2,:)`	`a[1]` or `a[1, :]`	entire second row of 2D array `a`
`a(1:5,:)`	`a[0:5]` or `a[:5]` or `a[0:5, :]`	first 5 rows of 2D array `a`
`a(end-4:end,:)`	`a[-5:]`	last 5 rows of 2D array `a`

MATLAB	NumPy	Notes
`a(1:3,5:9)`	`a[0:3, 4:9]`	The first through third rows and fifth through ninth columns of a 2D array, `a`.
`a([2,4,5],[1,3])`	`a[np.ix_([1, 3, 4], [0, 2])]`	rows 2,4 and 5 and columns 1 and 3. This allows the matrix to be modified, and doesn't require a regular slice.
`a(3:2:21,:)`	`a[2:21:2,:]`	every other row of `a`, starting with the third and going to the twenty-first
`a(1:2:end,:)`	`a[ ::2,:]`	every other row of `a`, starting with the first
`a(end:-1:1,:)` or `flipud(a)`	`a[::-1,:]`	`a` with rows in reverse order
`a([1:end 1],:)`	`a[np.r_[:len(a),0]]`	`a` with copy of the first row appended to the end
`a.'`	`a.transpose()` or `a.T`	transpose of `a`
`a'`	`a.conj().transpose()` or `a.conj().T`	conjugate transpose of `a`
`a * b`	`a @ b`	matrix multiply
`a .* b`	`a * b`	element-wise multiply
`a./b`	`a/b`	element-wise divide
`a.^3`	`a**3`	element-wise exponentiation
`(a > 0.5)`	`(a > 0.5)`	matrix whose i,jth element is (a_ij > 0.5). The MATLAB result is an array of logical values 0 and 1. The NumPy result is an array of the boolean values `False` and `True`.
`find(a > 0.5)`	`np.nonzero(a > 0.5)`	find the indices where (`a` > 0.5)

MATLAB	NumPy	Notes
`a(:,find(v > 0.5))`	`a[:,np.nonzero(v > 0.5)[0]]`	extract the columns of `a` where vector v > 0.5
`a(:,find(v>0.5))`	`a[:, v.T > 0.5]`	extract the columns of `a` where column vector v > 0.5
`a(a<0.5)=0`	`a[a < 0.5]=0`	`a` with elements less than 0.5 zeroed out
`a .* (a>0.5)`	`a * (a > 0.5)`	`a` with elements less than 0.5 zeroed out
`a(:) = 3`	`a[:] = 3`	set all values to the same scalar value
`y=x`	`y = x.copy()`	NumPy assigns by reference
`y=x(2,:)`	`y = x[1, :].copy()`	NumPy slices are by reference
`y=x(:)`	`y = x.flatten()`	turn array into vector (note that this forces a copy). To obtain the same data ordering as in MATLAB, use `x.flatten('F')`.
`1:10`	`np.arange(1., 11.)` or `np.r_[1.:11.]` or `np.r_[1:10:10j]`	create an increasing vector (see note RANGES)
`0:9`	`np.arange(10.)` or `np.r_[:10.]` or `np.r_[:9:10j]`	create an increasing vector (see note RANGES)
`[1:10]'`	`np.arange(1.,11.)[:, np.newaxis]`	create a column vector
`zeros(3,4)`	`np.zeros((3, 4))`	3x4 two-dimensional array full of 64-bit floating point zeros
`zeros(3,4,5)`	`np.zeros((3, 4, 5))`	3x4x5 three-dimensional array full of 64-bit floating point zeros
`ones(3,4)`	`np.ones((3, 4))`	3x4 two-dimensional array full of 64-bit floating point ones
`eye(3)`	`np.eye(3)`	3x3 identity matrix

MATLAB	NumPy	Notes
`diag(a)`	`np.diag(a)`	returns a vector of the diagonal elements of 2D array, `a`
`diag(v,0)`	`np.diag(v, 0)`	returns a square diagonal matrix whose nonzero values are the elements of vector, `v`

`rng(42,'twister')`
`rand(3,4)`
	`from numpy.random import default_rng`
	`rng = default_rng(42)`
	`rng.random(3, 4)`
	
	or older version: `random.rand((3, 4))`
		generate a random 3x4 array with default random number generator and seed = 42

`linspace(1,3,4)`	`np.linspace(1,3,4)`	4 equally spaced samples between 1 and 3, inclusive
`[x,y]=meshgrid(0:8,0:5)`	`np.mgrid[0:9.,0:6.]` or `np.meshgrid(r_[0:9.],r_[0:6.]`	two 2D arrays: one of x values, the other of y values
	`ogrid[0:9.,0:6.]` or `np.ix_(np.r_[0:9.],np.r_[0:6.]`	the best way to eval functions on a grid
`[x,y]=meshgrid([1,2,4],[2,4,5])`	`np.meshgrid([1,2,4],[2,4,5])`	
	`ix_([1,2,4],[2,4,5])`	the best way to eval functions on a grid
`repmat(a, m, n)`	`np.tile(a, (m, n))`	create m by n copies of `a`
`[a b]`	`np.concatenate((a,b),1)` or `np.hstack((a,b))` or `np.column_stack((a,b))` or `np.c_[a,b]`	concatenate columns of `a` and `b`
`[a; b]`	`np.concatenate((a,b))` or `np.vstack((a,b))` or `np.r_[a,b]`	concatenate rows of `a` and `b`
`max(max(a))`	`a.max()` or `np.nanmax(a)`	maximum element of `a` (with ndims(a)<=2 for MATLAB, if there are NaN's, `nanmax` will ignore these and return largest value)
`max(a)`	`a.max(0)`	maximum element of each column of array `a`

MATLAB	NumPy	Notes
`max(a,[],2)`	`a.max(1)`	maximum element of each row of array `a`
`max(a,b)`	`np.maximum(a, b)`	compares `a` and `b` element-wise, and returns the maximum value from each pair
`norm(v)`	`np.sqrt(v @ v)` or `np.linalg.norm(v)`	L2 norm of vector `v`
`a & b`	`logical_and(a,b)`	element-by-element AND operator (NumPy ufunc) See note LOGICOPS
`a | b`	`np.logical_or(a,b)`	element-by-element OR operator (NumPy ufunc) See note LOGICOPS
`bitand(a,b)`	`a & b`	bitwise AND operator (Python native and NumPy ufunc)
`bitor(a,b)`	`a | b`	bitwise OR operator (Python native and NumPy ufunc)
`inv(a)`	`linalg.inv(a)`	inverse of square 2D array `a`
`pinv(a)`	`linalg.pinv(a)`	pseudo-inverse of 2D array `a`
`rank(a)`	`linalg.matrix_rank(a)`	matrix rank of a 2D array `a`
`a\b`	`linalg.solve(a, b)` if `a` is square; `linalg.lstsq(a, b)` otherwise	solution of a x = b for x
`b/a`	Solve `a.T x.T = b.T` instead	solution of x a = b for x
`[U,S,V]=svd(a)`	`U, S, Vh = linalg.svd(a), V = Vh.T`	singular value decomposition of `a`
`c=chol(a)` where `a==c'*c`	`c = linalg.cholesky(a)` where `a == c@c.T`	Cholesky factorization of a 2D array (`chol(a)` in MATLAB returns an upper triangular 2D array, but **`cholesky`** returns a lower triangular 2D array)

MATLAB	NumPy	Notes
`[V,D]=eig(a)`	`D,V = linalg.eig(a)`	eigenvalues λ and eigenvectors $\bar{v}$ of `a`, where $\lambda\bar{v} = \mathbf{a}\bar{v}$
`[V,D]=eig(a,b)`	`D,V = linalg.eig(a, b)`	eigenvalues λ and eigenvectors $\bar{v}$ of `a`, `b` where $\lambda\mathbf{b}\bar{v} = \mathbf{a}\bar{v}$
`[V,D]=eigs(a,3)`	`D,V = eigs(a, k = 3)`	find the `k=3` largest eigenvalues and eigenvectors of 2D array, `a`
`[Q,R,P]=qr(a,0)`	`Q,R = linalg.qr(a)`	QR decomposition
`[L,U,P]=lu(a)` where `a==P'*L*U`	`P,L,U = linalg.lu(a)` where `a == P@L@U`	LU decomposition (note: P(MATLAB) == transpose(P(NumPy)))
`conjgrad`	`cg`	Conjugate gradients solver
`fft(a)`	`np.fft(a)`	Fourier transform of `a`
`ifft(a)`	`np.ifft(a)`	inverse Fourier transform of `a`
`sort(a)`	`np.sort(a)` or `a.sort(axis=0)`	sort each column of a 2D array, `a`
`sort(a, 2)`	`np.sort(a, axis = 1)` or `a.sort(axis = 1)`	sort the each row of 2D array, `a`
`[b,I]=sortrows(a,1)`	`I = np.argsort(a[:, 0]); b = a[I,:]`	save the array `a` as array `b` with rows sorted by the first column
`x = Z\y`	`x = linalg.lstsq(Z, y)`	perform a linear regression of the form $\mathbf{Zx} = \mathbf{y}$
`decimate(x, q)`	`signal.resample(x, np.ceil(len(x)/q))`	downsample with low-pass filtering
`unique(a)`	`np.unique(a)`	a vector of unique values in array `a`
`squeeze(a)`	`a.squeeze()`	remove singleton dimensions of array `a`. Note that MATLAB will always return arrays of 2D or higher while NumPy will return arrays of 0D or higher

Notes

Submatrix: Assignment to a submatrix can be done with lists of indices using the `ix_` command. E.g., for 2D array `a`, one might do: `ind=[1, 3]; a[np.ix_(ind, ind)] += 100`.

HELP: There is no direct equivalent of MATLAB's `which` command, but the commands **`help`** and **`numpy.source`** will usually list the filename where the function is located. Python also has an `inspect` module (do `import inspect`) which provides a `getfile` that often works.

INDEXING: MATLAB uses one based indexing, so the initial element of a sequence has index 1. Python uses zero based indexing, so the initial element of a sequence has index 0. Confusion and flamewars arise because each has advantages and disadvantages. One based indexing is consistent with common human language usage, where the "first" element of a sequence has index 1. Zero based indexing simplifies indexing. See also a text by prof.dr. Edsger W. Dijkstra.

RANGES: In MATLAB, `0:5` can be used as both a range literal and a 'slice' index (inside parentheses); however, in Python, constructs like `0:5` can *only* be used as a slice index (inside square brackets). Thus the somewhat quirky `r_` object was created to allow NumPy to have a similarly terse range construction mechanism. Note that `r_` is not called like a function or a constructor, but rather *indexed* using square brackets, which allows the use of Python's slice syntax in the arguments.

LOGICOPS: `&` or `|` in NumPy is bitwise AND/OR, while in MATLAB & and `|` are logical AND/OR. The two can appear to work the same, but there are important differences. If you would have used MATLAB's `&` or `|` operators, you should use the NumPy ufuncs `logical_and`/`logical_or`. The notable differences between MATLAB's and NumPy's `&` and `|` operators are:

- Non-logical {0,1} inputs: NumPy's output is the bitwise AND of the inputs. MATLAB treats any non-zero value as 1 and returns the logical AND. For example `(3 & 4)` in NumPy is `0`, while in MATLAB both `3` and `4` are considered logical true and `(3 & 4)` returns `1`.
- Precedence: NumPy's & operator is higher precedence than logical operators like `<` and `>`; MATLAB's is the reverse.

If you know you have boolean arguments, you can get away with using NumPy's bitwise operators, but be careful with parentheses, like this: `z = (x > 1) & (x < 2)`. The absence of NumPy operator forms of `logical_and` and `logical_or` is an unfortunate consequence of Python's design.

RESHAPE and LINEAR INDEXING: MATLAB always allows multi-dimensional arrays to be accessed using scalar or linear indices, NumPy does not. Linear indices are common in MATLAB programs, e.g. `find()` on a matrix returns them, whereas NumPy's find behaves differently. When converting MATLAB code it might be necessary to first reshape a matrix to a linear sequence, perform some indexing operations and then reshape back. As reshape (usually) produces views onto the same storage, it should be possible to do this fairly efficiently. Note that the scan order used by reshape in NumPy defaults to the 'C' order, whereas MATLAB uses the Fortran order. If you are simply converting to a linear sequence and back this doesn't matter. But if you are converting reshapes from MATLAB code which relies on the scan order, then this MATLAB code: `z = reshape(x,3,4);` should become `z = x.reshape(3,4,order='F').copy()` in NumPy.

'array' or 'matrix'? Which should I use?

Historically, NumPy has provided a special matrix type, *np.matrix*, which is a subclass of ndarray which makes binary operations linear algebra operations. You may see it used in some existing code instead of *np.array*. So, which one to use?

Short answer

Use arrays.

- They support multidimensional array algebra that is supported in MATLAB
- They are the standard vector/matrix/tensor type of NumPy. Many NumPy functions return arrays, not matrices.
- There is a clear distinction between element-wise operations and linear algebra operations.

- You can have standard vectors or row/column vectors if you like.

Until Python 3.5 the only disadvantage of using the array type was that you had to use `dot` instead of `*` to multiply (reduce) two tensors (scalar product, matrix vector multiplication etc.). Since Python 3.5 you can use the matrix multiplication `@` operator.

Given the above, we intend to deprecate `matrix` eventually.

Long answer

NumPy contains both an `array` class and a `matrix` class. The `array` class is intended to be a general-purpose n-dimensional array for many kinds of numerical computing, while `matrix` is intended to facilitate linear algebra computations specifically. In practice there are only a handful of key differences between the two.

- Operators `*` and `@`, functions `dot()`, and `multiply()`:
  - For `array`, **``*`` means element-wise multiplication**, while **``@`` means matrix multiplication**; they have associated functions `multiply()` and `dot()`. (Before Python 3.5, `@` did not exist and one had to use `dot()` for matrix multiplication).
  - For `matrix`, **``*`` means matrix multiplication**, and for element-wise multiplication one has to use the `multiply()` function.
- Handling of vectors (one-dimensional arrays)
  - For `array`, the **vector shapes 1xN, Nx1, and N are all different things**. Operations like `A[:,1]` return a one-dimensional array of shape N, not a two-dimensional array of shape Nx1. Transpose on a one-dimensional `array` does nothing.
  - For `matrix`, **one-dimensional arrays are always upconverted to 1xN or Nx1 matrices** (row or column vectors). `A[:,1]` returns a two-dimensional matrix of shape Nx1.
- Handling of higher-dimensional arrays (ndim > 2)
  - `array` objects **can have number of dimensions > 2**;
  - `matrix` objects **always have exactly two dimensions**.
- Convenience attributes
  - `array` **has a .T attribute**, which returns the transpose of the data.
  - `matrix` **also has .H, .I, and .A attributes**, which return the conjugate transpose, inverse, and `asarray()` of the matrix, respectively.
- Convenience constructor
  - The `array` constructor **takes (nested) Python sequences as initializers**. As in, `array([[1,2,3],[4,5,6]])`.
  - The `matrix` constructor additionally **takes a convenient string initializer**. As in `matrix("[1 2 3; 4 5 6]")`.

There are pros and cons to using both:

- `array`
  - `:)` Element-wise multiplication is easy: `A*B`.
  - `:(` You have to remember that matrix multiplication has its own operator, `@`.
  - `:)` You can treat one-dimensional arrays as *either* row or column vectors. `A @ v` treats `v` as a column vector, while `v @ A` treats `v` as a row vector. This can save you having to type a lot of transposes.
  - `:)` `array` is the “default” NumPy type, so it gets the most testing, and is the type most likely to be returned by 3rd party code that uses NumPy.
  - `:)` Is quite at home handling data of any number of dimensions.
  - `:)` Closer in semantics to tensor algebra, if you are familiar with that.
  - `:)` *All* operations (`*`, `/`, `+`, `-` etc.) are element-wise.
  - `:(` Sparse matrices from `scipy.sparse` do not interact as well with arrays.
- `matrix`
  - `:\\` Behavior is more like that of MATLAB matrices.
  - `<:(` Maximum of two-dimensional. To hold three-dimensional data you need `array` or perhaps a Python list of `matrix`.
  - `<:(` Minimum of two-dimensional. You cannot have vectors. They must be cast as single-column or single-row matrices.

    - `<:(` Since `array` is the default in NumPy, some functions may return an `array` even if you give them a `matrix` as an argument. This shouldn't happen with NumPy functions (if it does it's a bug), but 3rd party code based on NumPy may not honor type preservation like NumPy does.
    - `:)` `A*B` is matrix multiplication, so it looks just like you write it in linear algebra (For Python >= 3.5 plain arrays have the same convenience with the `@` operator).
    - `<:(` Element-wise multiplication requires calling a function, `multiply(A,B)`.
    - `<:(` The use of operator overloading is a bit illogical: `*` does not work element-wise but `/` does.
    - Interaction with `scipy.sparse` is a bit cleaner.

The `array` is thus much more advisable to use. Indeed, we intend to deprecate `matrix` eventually.

Customizing your environment

In MATLAB the main tool available to you for customizing the environment is to modify the search path with the locations of your favorite functions. You can put such customizations into a startup script that MATLAB will run on startup.

NumPy, or rather Python, has similar facilities.

- To modify your Python search path to include the locations of your own modules, define the `PYTHONPATH` environment variable.
- To have a particular script file executed when the interactive Python interpreter is started, define the `PYTHONSTARTUP` environment variable to contain the name of your startup script.

Unlike MATLAB, where anything on your path can be called immediately, with Python you need to first do an 'import' statement to make functions in a particular file accessible.

For example you might make a startup script that looks like this (Note: this is just an example, not a statement of "best practices"):

```
# Make all numpy available via shorter 'np' prefix
import numpy as np
#
# Make the SciPy linear algebra functions available as linalg.func()
# e.g. linalg.lu, linalg.eig (for general l*B@u==A@u solution)
from scipy import linalg
#
# Define a Hermitian function
def hermitian(A, **kwargs):
    return np.conj(A,**kwargs).T
# Make a shortcut for hermitian:
#    hermitian(A) --> H(A)
H = hermitian
```

To use the deprecated *matrix* and other *matlib* functions:

```
# Make all matlib functions accessible at the top level via M.func()
import numpy.matlib as M
# Make some matlib functions accessible directly at the top level via, e.g.
rand(3,3)
from numpy.matlib import matrix,rand,zeros,ones,empty,eye
```

Links

Another somewhat outdated MATLAB/NumPy cross-reference can be found at http://mathesaurus.sf.net/

An extensive list of tools for scientific work with Python can be found in the topical software page.

See List of Python software: scripting for a list of software that use Python as a scripting language

MATLAB® and SimuLink® are registered trademarks of The MathWorks, Inc.

Previous
Miscellaneous

Next
Building from source