NumPy Package

What is NumPy?

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

NumPy for Matlab users

NumPy	Matlab
Python uses 0 (zero) based indexing. The initial element of a sequence is found using a[0].	MATLAB® uses 1 (one) based indexing. The initial element of a sequence is found using a(1).
Uses multidimensional **array***	Uses multidimensional **Matrice**
2x3 matrix literal array([[1.,2.,3.], [4.,5.,6.]])	2x3 matrix literal [1 2 3; 4 5 6]
access element in second row, fifth column a[1,4]	access element in second row, fifth column a(2,5)

For more information, visit: https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html

* There is a **matrix** type in NumPy for 2-dim linear algebra. But we prefer array

Let’s Start Programming

NumPy for Matlab users

General Purpose Equivalents

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`	`source(func)` or `func??` (in Ipython)	print source for *func* (if not a native function)
`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
`1*i, 1*j, 1i, 1j`	`1j`	complex numbers
`eps`	`np.spacing(1)`	Distance between 1 and the nearest floating point number.
`ode45`	`scipy.integrate.ode(f).set_integrator('dopri5')`	integrate an ODE with Runge-Kutta 4,5
`ode15s`	`scipy.integrate.ode(f).set_integrator('vode', method='bdf', order=5)`	integrate an ODE with BDF method

Linear Algebra Equivalents

MATLAB	NumPy	Notes
`ndims(a)`	`ndim(a)` or `a.ndim`	get the number of dimensions of an array
`numel(a)`	`size(a)` or `a.size`	get the number of elements of an array
`size(a)`	`shape(a)` or `a.shape`	get the "size" of the matrix
`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 ]`	`array([[1.,2.,3.], [4.,5.,6.]])`	2x3 matrix literal
`[ a b; c d ]`	`vstack([hstack([a,b]), hstack([c,d])])` or `bmat('a b; c d').A`	construct a matrix from blocks `a`, `b`, `c`, and `d`
`a(end)`	`a[-1]`	access last element in the 1xn matrix `a`
`a(2,5)`	`a[1,4]`	access element in second row, fifth column
`a(2,:)`	`a[1]` or `a[1,:]`	entire second row of `a`
`a(1:5,:)`	`a[0:5]` or `a[:5]` or `a[0:5,:]`	the first five rows of `a`
`a(end-4:end,:)`	`a[-5:]`	the last five rows of `a`
`a(1:3,5:9)`	`a[0:3][:,4:9]`	rows one to three and columns five to nine of `a`. This gives read-only access.
`a([2,4,5],[1,3])`	`a[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[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.dot(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 0s and 1s. The NumPy result is an array of the boolean values `False` and `True`.
`find(a>0.5)`	`nonzero(a>0.5)`	find the indices where (`a` > 0.5)
`a(:,find(v>0.5))`	`a[:,nonzero(v>0.5)[0]]`	extract the columms of `a` where vector v > 0.5
`a(:,find(v>0.5))`	`a[:,v.T>0.5]`	extract the columms 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)
`1:10`	`arange(1.,11.)` or `r_[1.:11.]` or `r_[1:10:10j]`	create an increasing vector (see note *RANGES*)
`0:9`	`arange(10.)` or `r_[:10.]` or `r_[:9:10j]`	create an increasing vector (see note *RANGES*)
`[1:10]'`	`arange(1.,11.)[:, newaxis]`	create a column vector
`zeros(3,4)`	`zeros((3,4))`	3x4 two-dimensional array full of 64-bit floating point zeros

`zeros(3,4,5)`	`zeros((3,4,5))`	3x4x5 three-dimensional array full of 64-bit floating point zeros
`ones(3,4)`	`ones((3,4))`	3x4 two-dimensional array full of 64-bit floating point ones
`eye(3)`	`eye(3)`	3x3 identity matrix
`diag(a)`	`diag(a)`	vector of diagonal elements of `a`
`diag(a,0)`	`diag(a,0)`	square diagonal matrix whose nonzero values are the elements of `a`
`rand(3,4)`	`random.rand(3,4)`	random 3x4 matrix
`linspace(1,3,4)`	`linspace(1,3,4)`	4 equally spaced samples between 1 and 3, inclusive
`[x,y]=meshgrid(0:8,0:5)`	`mgrid[0:9.,0:6.]` or `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 `ix_(r_[0:9.],r_[0:6.]`	the best way to eval functions on a grid
`[x,y]=meshgrid([1,2,4],[2,4,5])`	`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)`	`tile(a, (m, n))`	create m by n copies of `a`
`[a b]`	`concatenate((a,b),1)` or `hstack((a,b))` or `column_stack((a,b))` or `c_[a,b]`	concatenate columns of `a` and `b`
`[a; b]`	`concatenate((a,b))` or `vstack((a,b))` or `r_[a,b]`	concatenate rows of `a` and `b`
`max(max(a))`	`a.max()`	maximum element of `a` (with ndims(a)<=2 for matlab)
`max(a)`	`a.max(0)`	maximum element of each column of matrix `a`
`max(a,[],2)`	`a.max(1)`	maximum element of each row of matrix `a`
`max(a,b)`	`maximum(a, b)`	compares `a` and `b` element-wise, and returns the maximum value from each pair

`norm(v)`	`sqrt(dot(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`	`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 matrix `a`
`pinv(a)`	`linalg.pinv(a)`	pseudo-inverse of matrix `a`
`rank(a)`	`linalg.matrix_rank(a)`	matrix rank of a 2D array / matrix `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`
`chol(a)`	`linalg.cholesky(a).T`	cholesky factorization of a matrix (`chol(a)` in matlab returns an upper triangular matrix, but `linalg.cholesky(a)` returns a lower triangular matrix)
`[V,D]=eig(a)`	`D,V = linalg.eig(a)`	eigenvalues and eigenvectors of `a`
`[V,D]=eig(a,b)`	`V,D = np.linalg.eig(a,b)`	eigenvalues and eigenvectors of `a`, `b`
`[V,D]=eigs(a,k)`		find the `k` largest eigenvalues and eigenvectors of `a`
`[Q,R,P]=qr(a,0)`	`Q,R = scipy.linalg.qr(a)`	QR decomposition
`[L,U,P]=lu(a)`	`L,U = scipy.linalg.lu(a)` or `LU,P=scipy.linalg.lu_factor(a)`	LU decomposition (note: P(Matlab) == transpose(P(numpy)))

`conjgrad`	`scipy.sparse.linalg.cg`	Conjugate gradients solver
`fft(a)`	`fft(a)`	Fourier transform of `a`
`ifft(a)`	`ifft(a)`	inverse Fourier transform of `a`
`sort(a)`	`sort(a)` or `a.sort()`	sort the matrix
`[b,I] = sortrows(a,i)`	`I = argsort(a[:,i]), b=a[I,:]`	sort the rows of the matrix
`regress(y,X)`	`linalg.lstsq(X,y)`	multilinear regression
`decimate(x, q)`	`scipy.signal.resample(x, len(x)/q)`	downsample with low-pass filtering
`unique(a)`	`unique(a)`	
`squeeze(a)`	`a.squeeze()`	

Table Reference:
https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html

How to debug?

- Google it!
  - StackOverflow
  - Scipy.Org