MATLAB–Python–Julia cheatsheet¶

Dependencies and Setup¶

In the Python code we assume that you have already run `import numpy as np`

In the Julia, we assume you are using **v1.0.2 or later** with Compat **v1.3.0 or later** and have run `using LinearAlgebra, Statistics, Compat`

Creating Vectors¶

MATLAB	PYTHON	JULIA

MATLAB	PYTHON	JULIA
Row vector: size (1, n)		
`A = [1 2 3]`	`A = np.array([1, 2, 3]).reshape(1, 3)`	`A = [1 2 3]`
Column vector: size (n, 1)		
`A = [1; 2; 3]`	`A = np.array([1, 2, 3]).reshape(3, 1)`	`A = [1 2 3]'`
1d array: size (n,)		
Not possible	`A = np.array([1, 2, 3])`	`A = [1; 2; 3]` or `A = [1, 2, 3]`
Integers from j to n with step size k		
`A = j:k:n`	`A = np.arange(j, n+1, k)`	`A = j:k:n`
Linearly spaced vector of k points		
`A = linspace(1, 5, k)`	`A = np.linspace(1, 5, k)`	`A = range(1, 5, length = k)`

Creating Matrices¶

MATLAB	PYTHON	JULIA
Create a matrix		
`A = [1 2; 3 4]`	`A = np.array([[1, 2], [3, 4]])`	`A = [1 2; 3 4]`

MATLAB	PYTHON	JULIA

2 x 2 matrix of zeros

`A = zeros(2, 2)`	`A = np.zeros((2, 2))`	`A = zeros(2, 2)`

2 x 2 matrix of ones

`A = ones(2, 2)`	`A = np.ones((2, 2))`	`A = ones(2, 2)`

2 x 2 identity matrix

`A = eye(2, 2)`	`A = np.eye(2)`	`A = I # will adopt`
		`# 2x2 dims if demanded by`
		`# neighboring matrices`

Diagonal matrix

`A = diag([1 2 3])`	`A = np.diag([1, 2, 3])`	`A = Diagonal([1, 2, 3])`

Uniform random numbers

`A = rand(2, 2)`	`A = np.random.rand(2, 2)`	`A = rand(2, 2)`

Normal random numbers

`A = randn(2, 2)`	`A = np.random.randn(2, 2)`	`A = randn(2, 2)`

MATLAB	PYTHON	JULIA

Sparse Matrices

```
A = sparse(2, 2)
A(1, 2) = 4
A(2, 2) = 1
```

```
from scipy.sparse import
coo_matrix

A = coo_matrix(([4, 1],
                ([0, 1],
[1, 1])),
                shape=(2,
2))
```

```
using SparseArrays
A = spzeros(2, 2)
A[1, 2] = 4
A[2, 2] = 1
```

Tridiagonal Matrices

```
A = [1 2 3 NaN;
     4 5 6 7;
     NaN 8 9 0]
spdiags(A',[-1 0 1], 4, 4)
```

```
import sp.sparse as sp
diagonals = [[4, 5, 6, 7],
[1, 2, 3], [8, 9, 10]]
sp.diags(diagonals, [0, -1,
2]).toarray()
```

```
x = [1, 2, 3]
y = [4, 5, 6, 7]
z = [8, 9, 10]
Tridiagonal(x, y, z)
```

Manipulating Vectors and Matrices¶

MATLAB	PYTHON	JULIA

Transpose

```
A.'
```

```
A.T
```

```
transpose(A)
```

Complex conjugate transpose (Adjoint)

```
A'
```

```
A.conj()
```

```
A'
```

MATLAB	PYTHON	JULIA
Concatenate horizontally		

`A = [[1 2] [1 2]]`
or
`A = horzcat([1 2], [1 2])`
	`B = np.array([1, 2])`
	`A = np.hstack((B, B))`
		`A = [[1 2] [1 2]]`
		or
		`A = hcat([1 2], [1 2])`

Concatenate vertically		

`A = [[1 2]; [1 2]]`
or
`A = vertcat([1 2], [1 2])`
	`B = np.array([1, 2])`
	`A = np.vstack((B, B))`
		`A = [[1 2]; [1 2]]`
		or
		`A = vcat([1 2], [1 2])`

Reshape (to 5 rows, 2 columns)		
`A = reshape(1:10, 5, 2)`	`A = A.reshape(5, 2)`	`A = reshape(1:10, 5, 2)`
Convert matrix to vector		
`A(:)`	`A = A.flatten()`	`A[:]`
Flip left/right		
`fliplr(A)`	`np.fliplr(A)`	`reverse(A, dims = 2)`
Flip up/down		
`flipud(A)`	`np.flipud(A)`	`reverse(A, dims = 1)`
Repeat matrix (3 times in the row dimension, 4 times in the column dimension)		
`repmat(A, 3, 4)`	`np.tile(A, (4, 3))`	`repeat(A, 3, 4)`

MATLAB	PYTHON	JULIA

Preallocating/Similar

```
x = rand(10)
y = zeros(size(x, 1),
size(x, 2))
```

N/A similar type

```
x = np.random.rand(3, 3)
y = np.empty_like(x)

# new dims
y = np.empty((2, 3))
```

```
x = rand(3, 3)
y = similar(x)
# new dims
y = similar(x, 2, 2)
```

Broadcast a function over a collection/matrix/vector

```
f = @(x) x.^2
g = @(x, y) x + 2 + y.^2
x = 1:10
y = 2:11
f(x)
g(x, y)
```

Functions broadcast directly

```
def f(x):
    return x**2
def g(x, y):
    return x + 2 + y**2
x = np.arange(1, 10, 1)
y = np.arange(2, 11, 1)
f(x)
g(x, y)
```

Functions broadcast directly

```
f(x) = x^2
g(x, y) = x + 2 + y^2
x = 1:10
y = 2:11
f.(x)
g.(x, y)
```

Accessing Vector/Matrix Elements¶

MATLAB	PYTHON	JULIA

MATLAB	PYTHON	JULIA
Access one element		
`A(2, 2)`	`A[1, 1]`	`A[2, 2]`
Access specific rows		
`A(1:4, :)`	`A[0:4, :]`	`A[1:4, :]`
Access specific columns		
`A(:, 1:4)`	`A[:, 0:4]`	`A[:, 1:4]`
Remove a row		
`A([1 2 4], :)`	`A[[0, 1, 3], :]`	`A[[1, 2, 4], :]`
Diagonals of matrix		
`diag(A)`	`np.diag(A)`	`diag(A)`
Get dimensions of matrix		
`[nrow ncol] = size(A)`	`nrow, ncol = np.shape(A)`	`nrow, ncol = size(A)`

Mathematical Operations¶

MATLAB	PYTHON	JULIA
Dot product		

`dot(A, B)`	`np.dot(A, B)` **or** `A @ B`	`dot(A, B)`
		
		`A ⋅ B # \cdot<TAB>`

MATLAB	PYTHON	JULIA

Matrix multiplication

`A * B`	`A @ B`	`A * B`

Inplace matrix multiplication

Not possible

```
x = np.array([1,
2]).reshape(2, 1)
A = np.array(([1, 2], [3,
4]))
y = np.empty_like(x)
np.matmul(A, x, y)
```

```
x = [1, 2]
A = [1 2; 3 4]
y = similar(x)
mul!(y, A, x)
```

Element-wise multiplication

`A .* B`	`A * B`	`A .* B`

Matrix to a power

`A^2`	`np.linalg.matrix_power(A, 2)`	`A^2`

Matrix to a power, elementwise

`A.^2`	`A**2`	`A.^2`

Inverse

`inv(A)`	`np.linalg.inv(A)`	`inv(A)`
or		or
`A^(-1)`		`A^(-1)`

Determinant

`det(A)`	`np.linalg.det(A)`	`det(A)`

MATLAB	PYTHON	JULIA

Eigenvalues and eigenvectors

`[vec, val] = eig(A)`	`val, vec = np.linalg.eig(A)`	`val, vec = eigen(A)`

Euclidean norm

`norm(A)`	`np.linalg.norm(A)`	`norm(A)`

Solve linear system $Ax = b$ (when A is square)

`A\b`	`np.linalg.solve(A, b)`	`A\b`

Solve least squares problem $Ax = b$ (when A is rectangular)

`A\b`	`np.linalg.lstsq(A, b)`	`A\b`

Sum / max / min¶

MATLAB	PYTHON	JULIA

MATLAB	PYTHON	JULIA

Sum / max / min of each column

`sum(A, 1)`
`max(A, [], 1)`
`min(A, [], 1)`
	`sum(A, 0)`
	`np.amax(A, 0)`
	`np.amin(A, 0)`
		`sum(A, dims = 1)`
		`maximum(A, dims = 1)`
		`minimum(A, dims = 1)`

Sum / max / min of each row

`sum(A, 2)`
`max(A, [], 2)`
`min(A, [], 2)`
	`sum(A, 1)`
	`np.amax(A, 1)`
	`np.amin(A, 1)`
		`sum(A, dims = 2)`
		`maximum(A, dims = 2)`
		`minimum(A, dims = 2)`

Sum / max / min of entire matrix

`sum(A(:))`
`max(A(:))`
`min(A(:))`
	`np.sum(A)`
	`np.amax(A)`
	`np.amin(A)`
		`sum(A)`
		`maximum(A)`
		`minimum(A)`

Cumulative sum / max / min by row

`cumsum(A, 1)`
`cummax(A, 1)`
`cummin(A, 1)`
	`np.cumsum(A, 0)`
	`np.maximum.accumulate(A, 0)`
	`np.minimum.accumulate(A, 0)`
		`cumsum(A, dims = 1)`
		`accumulate(max, A, dims = 1)`
		`accumulate(min, A, dims = 1)`

Cumulative sum / max / min by column

`cumsum(A, 2)`
`cummax(A, 2)`
`cummin(A, 2)`
	`np.cumsum(A, 1)`
	`np.maximum.accumulate(A, 1)`
	`np.minimum.accumulate(A, 1)`
		`cumsum(A, dims = 2)`
		`accumulate(max, A, dims = 2)`
		`accumulate(min, A, dims = 2)`

Programming¶

MATLAB	PYTHON	JULIA

MATLAB	PYTHON	JULIA

Comment one line

```
% This is a comment
```

```
# This is a comment
```

```
# This is a comment
```

Comment block

```
%{
Comment block
%}
```

```
# Block
# comment
# following PEP8
```

```
#=
Comment block
=#
```

For loop

```
for i = 1:N
   % do something
end
```

```
for i in range(n):
    # do something
```

```
for i in 1:N
   # do something
end
```

While loop

```
while i <= N
   % do something
end
```

```
while i <= N:
    # do something
```

```
while i <= N
   # do something
end
```

If

```
if i <= N
   % do something
end
```

```
if i <= N:
  # do something
```

```
if i <= N
   # do something
end
```

If / else

```
if i <= N
   % do something
else
   % do something else
end
```

```
if i <= N:
    # do something
else:
    # so something else
```

```
if i <= N
   # do something
else
   # do something else
end
```

MATLAB	PYTHON	JULIA

Print text and variable

```
x = 10
fprintf('x = %d \n', x)
```

```
x = 10
print(f'x = {x}')
```

```
x = 10
println("x = $x")
```

Function: anonymous

```
f = @(x) x^2
```

```
f = lambda x: x**2
```

```
f = x -> x^2
# can be rebound
```

Function

```
function out  = f(x)
   out = x^2
end
```

```
def f(x):
    return x**2
```

```
function f(x)
   return x^2
end

f(x) = x^2 # not anon!
```

Tuples

```
t = {1 2.0 "test"}
t{1}
```

Can use cells but watch performance

```
t = (1, 2.0, "test")
t[0]
```

```
t = (1, 2.0, "test")
t[1]
```

Named Tuples/ Anonymous Structures

```
m.x = 1
m.y = 2

m.x
```

```
from collections import
namedtuple

mdef = namedtuple('m', 'x
y')
m = mdef(1, 2)

m.x
```

```
# vanilla
m = (x = 1, y = 2)
m.x

# constructor
using Parameters
mdef = @with_kw (x=1, y=2)
m = mdef() # same as above
m = mdef(x = 3)
```

MATLAB	PYTHON	JULIA

Closures

```
a = 2.0
f = @(x) a + x
f(1.0)
```

```
a = 2.0
def f(x):
    return a + x
f(1.0)
```

```
a = 2.0
f(x) = a + x
f(1.0)
```

Inplace Modification

No consistent or simple syntax to achieve this (https://blogs.mathworks.com/loren/2007/03/22/in-place-operations-on-data/)

```
def f(x):
    x **=2
    return

x = np.random.rand(10)
f(x)
```

```
function f!(out, x)
    out .= x.^2
end
x = rand(10)
y = similar(x)
f!(y, x)
```

Credits

This cheat sheet was created by Victoria Gregory (https://github.com/vgregory757), Andrij Stachurski (http://drdrij.com/), Natasha Watkins (https://github.com/natashawatkins) and other collaborators on behalf of QuantEcon (http://quantecon.org/).