NumPy Tutorial

NumPy is an amazing scientific computing library that is used by numerous other Python Data Science libraries. It contains many mathematical, array and string functions that are extremely useful. Along with all the basic math functions you'll also find them for Linear Algebra, Statistics, Simulation, etc.

I recommend that you use the Jupyter setup I use with Anaconda so that you'll have access to all of NumPys dependencies.

NumPy utilizes vector (1D Arrays) and matrice arrays (2D Arrays).

NumPy Arrays : Creating Arrays

In [3]:
```
import numpy as np
# Provides beautiful plots of numerous type that are either
# static, animated and/or interactive
import matplotlib.pylab as plt
from numpy import random

# A Python list
list_1 = [1, 2, 3, 4, 5]

# Create NumPy 1 dimensional (a axis) array list object of type byte (-128 to 127)
# A N-dimensional array is a usyally fixed size multidimensional
# array that contains items of the same type.
np_arr_1 = np.array(list_1, dtype=np.int8)
np_arr_1
# Create multidimenional list
m_list_1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Create NumPy multidimensional (2 axis) array without defining type
np_m_arr_1 = np.array(m_list_1)

# You can also create arrays by defining the start value,
# stop value (up to but not include stop) and step amount
np.arange(1, 10)

# With floats define start, end and number of values
np.linspace(0, 5, 7)

# You can create a 3 item array of zeroes
np.zeros(4)

# You can create multidimensional arrays of zeroes by passing
# a tuple with the 1st value being rows and the 2nd columns
np.zeros((2, 3))

# Create array of 1s
np.ones((2, 3))

# Get number of items in the array
np_m_arr_1.size

# Create array with defined values
np_arr_2 = np.array([1, 2, 3, 4, 5, 6])

# Get type for array
np_arr_2.dtype

# Data Types
# Boolean : np.bool_
# Char : np.byte
# Short : np.short
# Integer : np.short
# Long : np.int_
# Float : np.single & np.float32
# Double : np.double & np.float64
# np.int8 : -128 to 127
# np.int16 : -32768 to 32767
```

```
# np.int32 : -2147483648 to 2147483647
# np.int64 : -9223372036854775808 to 9223372036854775807

# Create random 5 value 1D array from 10 to 50
np.random.randint(10, 50, 5)
# Create random matrix 2x3 with values between 10 and 50
np.random.randint(10, 50, size=(2, 3))

# Get help with a function
np.random.randint?
```

Slicing and Indexes

In [222…

```
# Change value at index
# np_m_arr_1[0,0] = 2
# np_m_arr_1.itemset((0,1), 1)
np_m_arr_1
# Get size of array
np_m_arr_1.shape
# Get value by index
np_m_arr_1[0,1]
np_m_arr_1.item(0,1)

# Get specific indices
np.take(np_m_arr_1, [0, 3, 6])
# Replace provided index values with new values
np.put(np_m_arr_1, [0, 3, 6], [10, 10, 10])
print(np_m_arr_1)

# Start at 1st through 5th with 2 step
np_arr_1[:5:2]
# Get 2nd value from each row
np_m_arr_1[:,1]
# Flip Array
np_arr_1[::-1]

# Get evens
evens = np_m_arr_1[np_m_arr_1%2==0]
evens

# Get values > 5
np_m_arr_1[np_m_arr_1 > 5]
# 5 < value < 9
np_m_arr_1[(np_m_arr_1 > 5) & (np_m_arr_1 < 9)]
# 5 < value or value = 10
np_m_arr_1[(np_m_arr_1 > 5) | (np_m_arr_1 == 10)]

# Find uniques
np.unique(np_m_arr_1)
```

```
[[10  2  3]
 [10  5  6]
 [10  8  9]]
```

Out[222… array([2, 3, 5, 6, 8, 9, 10])

Reshaping Arrays

In [165…
```
# Reshape array to 1 by 9
np_m_arr_1.reshape((1, 9))
np_m_arr_1
# Reshape array to 2 by 5 (Items are either lost or 0s added)
np_m_arr_1.resize((2,5))
np_m_arr_1
# Transpose axes
np_m_arr_1.transpose()
# Swap axes
np_m_arr_1.swapaxes(0,1)
# Flatten in order
np_m_arr_1.flatten()
# Flatten in column order
np_m_arr_1.flatten('F')
# Sort rows
np_m_arr_1.sort(axis=1)
print(np_m_arr_1)
# Sort columns
np_m_arr_1.sort(axis=0)
np_m_arr_1
```

```
[[ 2  3  5 10 10]
 [ 0  6  8  9 10]]
```

Out[165…
```
array([[ 0,  3,  5,  9, 10],
       [ 2,  6,  8, 10, 10]])
```

Stacking & Splitting

In [195…
```
# Generate random arrays
ss_arr_1 = np.random.randint(10, size=(2, 2))
print("ss_arr_1\n", ss_arr_1)
ss_arr_2 = np.random.randint(10, size=(2, 2))
print("ss_arr_2\n", ss_arr_2)

# Stack arr_2 under arr_1
np.vstack((ss_arr_1, ss_arr_2))
# Stack horizontally
np.hstack((ss_arr_1, ss_arr_2))

# Delete 2nd row on each array
ss_arr_3 = np.delete(ss_arr_1, 1, 0)
ss_arr_4 = np.delete(ss_arr_2, 1, 0)
print("ss_arr_3\n", ss_arr_3)
print("ss_arr_4\n", ss_arr_4)
# Combine arrays
np.column_stack((ss_arr_3, ss_arr_4))
# Stack in a 2D array
np.row_stack((ss_arr_3, ss_arr_4))

# Generate 2x10 array
ss_arr_5 = np.random.randint(10, size=(2, 10))
print("ss_arr_5\n", ss_arr_5)
```

```
# Split into 5 arrays taking from both arrays in multidimensional array
np.hsplit(ss_arr_5, 5)
# Split after 2nd & 4th column
np.hsplit(ss_arr_5, (2, 4))
```

```
ss_arr_1
 [[5 9]
 [3 8]]
ss_arr_2
 [[7 1]
 [5 6]]
ss_arr_3
 [[5 9]]
ss_arr_4
 [[7 1]]
ss_arr_5
 [[6 7 1 4 1 1 2 2 1 5]
 [8 3 9 7 6 0 1 9 9 2]]
```

Out[195…

```
[array([[6, 7],
        [8, 3]]),
 array([[1, 4],
        [9, 7]]),
 array([[1, 1, 2, 2, 1, 5],
        [6, 0, 1, 9, 9, 2]])]
```

Copying

In [209…

```
cp_arr_1 = np.random.randint(10, size=(2, 2))
# Both variables point at the same array
cp_arr_2 = cp_arr_1
# Change value
cp_arr_1[0,0] = 2
print("cp_arr_1\n", cp_arr_1)
print("cp_arr_2\n", cp_arr_2)
# Create a view of data where changes don't effect original
cp_arr_3 = cp_arr_1.view()
print("cp_arr_3\n", cp_arr_3)
cp_arr_3 = cp_arr_3.flatten('F')
print("cp_arr_3\n", cp_arr_3)
print("cp_arr_1\n", cp_arr_1)
# Copy and create new array
cp_arr_4 = cp_arr_1.copy()
```

```
cp_arr_1
 [[2 4]
 [6 4]]
cp_arr_2
 [[2 4]
 [6 4]]
cp_arr_3
 [[2 4]
 [6 4]]
cp_arr_3
 [2 6 4 4]
cp_arr_1
 [[2 4]
 [6 4]]
```

Basic Math

In [4]:
```
arr_3 = np.array([1, 2, 3, 4])
arr_4 = np.array([2, 4, 6, 8])
# Add values
arr_3 + arr_4
# Subtract
arr_3 - arr_4
# Multiply
arr_3 * arr_4
# Divide
arr_3 / arr_4
# Random 4 digit 1D array between 0 to 100
arr_5 = random.randint(100, size=(4))
arr_5
# Random 2 by 3 digit 2D array between 0 to 100
arr_6 = random.randint(100, size=(2, 3))
arr_6
# 4 random floats
random.rand(4)
# Get random value from an array
random.choice(arr_3)

# Sum of values in array
arr_3.sum()
# Sum columns
print(arr_6)
arr_6.sum(axis=0)
# Cumulative sum of rows
arr_6.cumsum(axis=1)

# Min of each row
arr_6.min(axis=1)
# Max of each column
arr_6.max(axis=0)

print("arr_3", arr_3)
print("arr_4", arr_4)
# Add individual numbers to array
np.add(arr_3, 5)
```

```
# Add arrays
np.add(arr_3, arr_4)
# Subtract
np.subtract(arr_3, arr_4)
# Multiply
np.multiply(arr_3, arr_4)
# Divide
np.divide(arr_3, arr_4)

arr_5 = np.array([[1, 2], [3, 4]])
arr_6 = np.array([[2, 4], [6, 9]])
print("arr_5\n", arr_5)
print("arr_6\n", arr_6)
# Divides elements in 1st array by 2nd array and returns remainder
np.remainder(arr_6, arr_5)

# Return values in 1st array to powers defined in 2nd array
np.power(arr_6, arr_5)
# Square root
np.sqrt(arr_3)
# Cube root
np.cbrt(arr_3)
# Absolute value of every element
np.absolute([-1, -2])
# Exponential of all elements in array
np.exp(arr_3)
# log functions
np.log(arr_3)
np.log2(arr_3)
np.log10(arr_3)
# Greatest common divisor
np.gcd.reduce([9, 12, 15])
# Lowest common multiple
np.lcm.reduce([9, 12, 15])

# Round down
np.floor([1.2, 2.5])
# Round up
np.ceil([1.2, 2.5])

# Can receive 6 values and square them
sq_arr = np.arange(6)**2
sq_arr[arr_3]

arr_7 = random.randint(100, size=(5, 3))
print("arr_7\n", arr_7)
# Get index for max value per column
mc_index = arr_7.argmax(axis=0)
mc_index
# Get numbers corresponding to indexes
max_nums = arr_7[mc_index]
arr_7[mc_index, range(arr_7.shape[1])]
```

```
[[67  7 80]
 [67 56 17]]
arr_3 [1 2 3 4]
arr_4 [2 4 6 8]
arr_5
 [[1 2]
 [3 4]]
arr_6
 [[2 4]
 [6 9]]
arr_7
 [[22 65 89]
 [62 17 79]
 [74 92 46]
 [94 69 48]
 [48 81 43]]
```

Out[4]: array([94, 92, 89])

Reading from Files

In [6]:
```
# Pandas is used to manipulate tabular data and more
import pandas as pd
# Import using NumPy
from numpy import genfromtxt

# Read table of data from CSV file and convert to Numpy array
ic_sales = pd.read_csv('icecreamsales.csv').to_numpy()
ic_sales

# Read data using NumPy
ic_sales_2 = genfromtxt('icecreamsales.csv', delimiter=',')
# Remove NANs
ic_sales_2 = [row[~np.isnan(row)] for row in ic_sales_2]
ic_sales_2
```

Out[6]:
```
[array([], dtype=float64),
 array([ 37., 292.]),
 array([ 40., 228.]),
 array([ 49., 324.]),
 array([ 61., 376.]),
 array([ 72., 440.]),
 array([ 79., 496.]),
 array([ 83., 536.]),
 array([ 81., 556.]),
 array([ 75., 496.]),
 array([ 64., 412.]),
 array([ 53., 324.]),
 array([ 40., 320.])]
```

Statistics Functions

In [7]:
```
# Array 1 - 5
sarr_1 = np.arange(1, 6)
np.mean(sarr_1)
```

```
np.median(sarr_1)
np.average(sarr_1)
np.std([4, 6, 3, 5, 2]) # Standard Deviation
np.var([4, 6, 3, 5, 2]) # Variance
# Also nanmedian, nanmean, nanstd, nanvar

print("ic_sales\n", ic_sales)
# Get the 50th percentile of the data
np.percentile(ic_sales, 50, axis=0)
# Get 1st column
ic_sales[:,0]

# Correlation coefficient : Measure of correlation between data
# Closer to 1 the more the data is correlated
np.corrcoef(ic_sales[:,0], ic_sales[:,1])

# Calculating Regression line
# Σ(x-x̄)*(y-ȳ) / Σ(x-x̄)2
temp_mean = np.mean(ic_sales[:,0])
sales_mean = np.mean(ic_sales[:,1])
numerator = np.sum(((ic_sales[:,0] - temp_mean)*(ic_sales[:,1] - sales_mean)))
denominator = np.sum(np.square(ic_sales[:,0] - temp_mean))
slope = numerator/denominator
# Calculate y intercept
y_i = sales_mean - slope * temp_mean
y_i
reg_arr = ic_sales[:,0] * slope + y_i
reg_arr
```

```
ic_sales
 [[ 37 292]
 [ 40 228]
 [ 49 324]
 [ 61 376]
 [ 72 440]
 [ 79 496]
 [ 83 536]
 [ 81 556]
 [ 75 496]
 [ 64 412]
 [ 53 324]
 [ 40 320]]
```

Out[7]:
```
array([256.01208459, 273.8864465 , 327.50953224, 399.00697989,
       464.54630691, 506.25315137, 530.08563392, 518.16939265,
       482.42066882, 416.88134181, 351.34201479, 273.8864465 ])
```

Trig Functions

In [8]:
```
# Generate array of 200 values between -pi & pi
t_arr = np.linspace(-np.pi, np.pi, 200)

# Plot with x axis & y axis data
# plt.plot(t_arr, np.sin(t_arr)) # SIN
# plt.plot(t_arr, np.cos(t_arr)) # COS
# plt.plot(t_arr, np.tan(t_arr)) # TAN
```

```
# Display plot
# plt.show()

# Provides inverse of If y = cos(x), x = arccos(y)
np.arcsin(1)
np.arccos(1)
np.arctan(0)

# Also arctan2, sinh, cosh, tanh, arcsinh, arccosh, arctanh

# Radians to degrees
np.rad2deg(np.pi)
# Degrees to radians
np.deg2rad(180)

# Hypotenuse c = √w² + h²
np.hypot(10,10)
```

Out[8]: 14.142135623730951

Matrix Functions

In [10]:
```
from numpy import linalg as LA

print("arr_5\n", arr_5)
print("arr_6\n", arr_6)
arr_8 = np.array([[5, 6], [7, 8]])

# Matrix multiplication with Dot Product
# (1 * 2) + (2 * 6) = 14 [0,0]
# (1 * 4) + (2 * 9) = 22 [0,1]
# (3 * 2) + (4 * 6) = 30 [1,0]
# (3 * 4) + (4 * 9) = 12 + 36 = 48 [1,1]
np.dot(arr_5, arr_6)
# Compute dot product of 2 or more arrays
LA.multi_dot([arr_5, arr_6, arr_8])

# Inner product
# (1 * 2) + (2 * 4) = 10 [0,0]
# (1 * 6) + (2 * 9) = 24 [0,1]
# (3 * 2) + (4 * 4) = 22 [1,0]
# (3 * 6) + (4 * 9) = 54 [1,1]
np.inner(arr_5, arr_6)
np.dot(arr_5, arr_6)

# Tensor Dot Product
# (1 * 1) + (2 * 2) + (3 * 3) + (4 * 4) = 30
# (5 * 1) + (6 * 2) + (7 * 3) + (8 * 4) =
arr_9 = np.array([[[1, 2],
        [3, 4]],
       [[5, 6],
        [7, 8]]])
arr_10 = np.array([[1, 2],
       [3, 4]], dtype=object)
np.tensordot(arr_9, arr_10)
```

```
# Einstein Summation : Provides many ways to perform
# operations on multiple arrays
arr_11 = np.array([0, 1])
arr_12 = np.array([[0, 1, 2, 3], [4, 5, 6, 7]])
# Left Side of -> : 1 axis for arr_11 and 2 axis for arr_12
# Right of -> : Array we want (1D Array)
# ij : Means multiply arr_11 single item by each column of arr_12 and sum
# [0, 4 + 5 + 6 + 7]
np.einsum('i,ij->i', arr_11, arr_12)
# Sum values in arr_11
np.einsum('i->', arr_11)
# Dot Product
print("arr_3\n", arr_3)
print("arr_4\n", arr_4)
np.einsum('i,i->', arr_3, arr_4)
# Matrix multiplication
np.einsum('ij,jk', arr_5, arr_6)
# Get diagonal
np.einsum('ii', arr_5)
# Transpose
np.einsum('ji', arr_5)

# Raise matrix to the power of n
# Given [[a, b], [c, d]]
# [[a² + bc, ab +db], [ac + dc, d² + bc]
LA.matrix_power(arr_5, 2)

# Kronecker product of 2 arrays
# Given [[a, b], [c, d]], [[e, f], [g, h]]
# [[a*e, a*f, b*e, b*f], [a*g, a*h, b*g, b*h], ...]
np.kron(arr_5, arr_6)

# Compute eigenvalues
LA.eig(arr_5) # Returns eigenvectors
LA.eigvals(arr_5)

# Get Vector Norm sqrt(sum(x**2))
LA.norm(arr_5)

# Get Multiplicative Inverse of a matrix
LA.inv(arr_5)

# Get Condition number of matrix
LA.cond(arr_5)

# Determinates are used to compute volume, area, to solve systems
# of equations and more. It is a way you can multiply values in a
# matrix to get 1 number.
# For a matrix to have an inverse its determinate must not equal 0
# det([[a, b], [c, d]]) = a*d - b*c
arr_12 = np.array([[1, 2], [3, 4]])
# 1*4 - 2*3 = -2
LA.det(arr_12)

# Determinate of 3x3 Matrix
```

```
# det([[a, b, c], [d, e, f], [g, h, i]]) = a*e*i - b*d*i + c*d*h
# - a*f*h + b*f*g - c*e*g

# When we multiply a matrix times its inverse we get the identity
# matrix [[1,0],[0,1]] for a 2x2 matrix
# Calculate the inverse 1/(a*d - b*c) * [[d, -b], [-c, a]]
# 1/(4 - 6) = -.5 -> [[-.5*4, -.5*-2], [-.5*-3, -.5*a]]
arr_12_i = LA.inv(arr_12)
arr_12_i

np.dot(arr_12, arr_12_i)

# Solving Systems of Linear Equations
# If you have 3x + 5 = 9x -> 5 = 6x -> x = 5/6
# If you have x + 4y = 10 & 6x + 18y = 42
# Isolate x -> x = 10 - 4y
# 6(10 - 4y) + 18y = 42 -> 60 - 24y + 18y = 42 - > -6y = -18 -> y = 3
# x + 4*3 = 10 -> x = -2
arr_13 = np.array([[1, 4], [6, 18]])
arr_14 = np.array([10, 42])
# Solve will solve this for you as well
LA.solve(arr_13, arr_14)

# Return a identity matrix with defined number of rows and columns
np.eye(2, 2, dtype=int)
```

```
arr_5
 [[1 2]
 [3 4]]
arr_6
 [[2 4]
 [6 9]]
arr_3
 [1 2 3 4]
arr_4
 [2 4 6 8]
```

Out[10]:
```
array([[1, 0],
       [0, 1]])
```

Saving & Loading NumPy Objects

In [12]:
```
arr_15 = np.array([[1, 2], [3, 4]])
# Save as randarray.npy
np.save('randarray', arr_15)
# Load saved array
arr_16 = np.load('randarray.npy')
arr_16

# Save as a CSV
np.savetxt('randcsv.csv', arr_15)
# Load CSV
arr_17 = np.loadtxt('randcsv.csv')
arr_17
```

Out[12]:
```
array([[1., 2.],
       [3., 4.]])
```

Financial Functions

In [25]:
```
# Install in Conda terminal with
# conda install pip
# pip install numpy-financial
import numpy_financial as npf

# Compute future value of $400 investment every month
# with an annual rate of 8% after 10 years
npf.fv(.08/12, 10*12, -400, -400)

# Calculate interest portion of payment on a loan of $3,000
# at 9.25% per year compounded monthly
# Period of loan (year)
period = np.arange(1*12) + 1
principle = 3000.00
# Interest Payment
ipmt = npf.ipmt(0.0925/12, period, 1*12, principle)
# Principle Payment
ppmt = npf.ppmt(0.0925/12, period, 1*12, principle)
for payment in period:
    index = payment - 1
    principle = principle + ppmt[index]
    print(f"{payment}   {np.round(ppmt[index], 2)}    {np.round(ipmt[index],2)}

# Compute number of payments to pay off $3,000 if you paid
# $150 per month with an interest rate of 9.25%
np.round(npf.nper(0.0925/12, -150, 3000.00), 2)

# Calculate net present value of cash flows of $4,000, $5,000
# $6,000, $7,000 after $15,000 investment with .08 rate per period
npf.npv(0.08, [-15000, 4000, 5000, 6000, 7000]).round(2)
```

```
1   -239.58    -23.12    2760.42
2   -241.42    -21.28    2519.0
3   -243.29    -19.42    2275.71
4   -245.16    -17.54    2030.55
5   -247.05    -15.65    1783.5
6   -248.95    -13.75    1534.55
7   -250.87    -11.83    1283.67
8   -252.81    -9.89    1030.87
9   -254.76    -7.95    776.11
10   -256.72    -5.98    519.39
11   -258.7    -4.0    260.69
12   -260.69    -2.01    0.0
```

Out[25]:
```
2898.6
```

Comparison Functions

In [157…
```
carr_1 = np.array([2, 3])
carr_2 = np.array([3, 2])
```

```
# Returns boolean based on whether arr_1 value Comparison arr_2 value
np.greater(carr_1, carr_2)
np.greater_equal(carr_1, carr_2)
np.less(carr_1, carr_2)
np.less_equal(carr_1, carr_2)
np.not_equal(carr_1, carr_2)
np.equal(carr_1, carr_2)
```

Out[157… array([False, False])