Getting Started with NumPy

NumPy (or Numpy): Python Library

- NumPy is a short form of “Numerical Python”
- PyData ecosystem heavily depends on numpy
- Incredibly fast (mostly written in C/C+, partially python)
- Provide array object up to 50x faster than python List
  - Codebase: https://github.com/numpy/numpy
- Installation
  - conda install numpy (if you have already installed anaconda)
  - pip install numpy (via pip package manager)

NumPy

- The array object in NumPy is called ndarray
- ```
  import numpy as np
  arr = np.array([1, 2, 3])
  print(arr)
  print(type(arr))
  print(np.__version__)
  ```
- Output
  ```
  [1, 2, 3]
  <class 'numpy.ndarray'>
  1.18.2
  ```
  ```

NumPy Array Dimensions

- Vectors
  - 1-D arrays
    - arr = np.array([1, 2, 3, 4, 5])
- Matrices
  - 2-D arrays
    - arr = np.array([[1, 2, 3], [4, 5, 6]])
- 3-D arrays
  - arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
- 0-D arrays
  - arr = np.array(42)
- Check dimension
  - print(arr.ndim)

You can also define the dimensions :

```
arr = np.array([1, 2, 3, 4], ndmin=5)
```

NumPy: Creating Arrays Quickly

- np.arange(0, 10) # [0 1 2 3 4 5 6 7 8 9] (like range function)
- np.zeros (3) # [0. 0. 0.]
- np.zeros ((2, 2)) # (row, col)
- np.ones(4), np.ones((3, 4))
- np.linspace(0, 2, 5) # [0. 0.5 1. 1.5 2.]
    - np.linspace(start, stop, num = 50, endpoint = True , retstep = False, dtype = None, axis = 0)
- np.eye(3) # Square matrix
    - np.eye(R, C = None, k = 0, dtype = type <‘float’>)

NumPy: Array Indexing

- `arr = np.array([1, 2, 3])`
  `print(arr[0])# 1`
- `arr_2d = np.array([[1,2,3], [4,5,6]])`
  `print(arr_2d[0][1])# 2  row = 0, col = 1`
  `print(arr_2d[0,1])# 2  row = 0, col = 1`
- `np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]]) # 3D`
  `print(arr[0, 1, 2]) # 6`
- Negative Indexing is also possible
- `print('Last element from 2nd dim: ', arr_2d[1, -1]) # 6`

NumPy: Array Slicing

- *[start:end:step]* # excludes the end index
- `arr = np.array([1, 2, 3, 4, 5])`
  `print(arr[1:3]), print(arr[1:]), print(arr[:4])`
- `print(arr[1:4:2])`
- Negative slicing is also possible
  - `print(arr[-3:-1])`
- Useful function
  - `arr.max()/min() # return index location`
  - `arr.shape`
  - `arr.dtype`

NumPy Broadcasting

- Set of rules by which NumPy lets you apply arithmetic operations (e.g., addition, subtraction, multiplication, etc.) between arrays of different sizes and shapes
- arr = np.array([1, 2, 3, 4, 5])

  arr[0:3] = 100 # [100 100 100   4   5]
- array1 = [1, 2, 3]

  array2 = [[1], [2], [3]]

  print(array1 + array 2)

```
[[2 3 4]
 [3 4 5]
 [4 5 6]]
```

- Does not need python loop, copies of data

```
# 1-D array
array1 = np.array([1, 2, 3])
# for scalar operation
number = 5
# add scalar and 1-D array
sum = array1 + number
print(sum) # [6 7 8]
```

NumPy Data Types

Default Data Types

- strings
- integer
- float
- boolean
- complex

Data types in NumPy

- i - integer
- b - boolean
- u - unsigned integer
- f - float
- c - complex float
- m - timedelta
- M - datetime
- O - object
- S - string
- U - unicode string
- V - fixed chunk of memory for other type (void)

```
arr =
np.array([1, 2, 3, 4], dtype='S’)

print(arr)
print(arr.dtype)
```

NumPy Array Copy vs View

- Copy is a new array, and the view is just a view of the original array
- ```
  arr = np.array([1, 2, 3, 4, 5])
  x = arr.copy()
  arr[0] = 42
  print(arr) # [42  2  3  4  5]
  print(x) # [1 2 3 4 5]
  ```
- ```
  arr = np.array([1, 2, 3, 4, 5])
  x = arr.view()
  arr[0] = 42
  print(arr) # [42  2  3  4  5]
  print(x) # [42  2  3  4  5]
  ```
- The view affected by the changes made to the original array
- Checking

  ```
  arr = np.array([1, 2, 3, 4, 5])
  x = arr.copy()
  y = arr.view()
  print(x.base) # None
  print(y.base) # [1 2 3 4 5]
  ```
- The copy returns None
- The view returns the original array

NumPy: Array Reshape

- `arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])`
  `newarr = arr.reshape(4, 3) # 1-D to 2-D`
- `arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])`
  `newarr = arr.reshape(2, 3, 2) # 1-D to 3-D`
- Reshape returns as view
- `newarr = arr.reshape(2, 2, -1) # numpy calculate dimension`
- `arr = np.array([[1, 2, 3], [4, 5, 6]])`
  `newarr = arr.reshape(-1) #` Flattening: [1 2 3 4 5 6]
- There are more functions for changing the shapes of arrays

NumPy: join, split, search, filter functions

- `arr = np.concatenate((arr1, arr2))`
- `Arr = np.stack((arr1, arr2), axis=1) # along column`
- `arr = np.hstack((arr1, arr2)) # along row`
- `newarr = np.array_split(arr, 3)`
- `x = np.where(arr == 4), x = np.where(arr%2 == 0)`
- `np.sort(arr)` # returns a copy of the array do not change original one
- ```
  arr = np.array([1, 2, 3, 4, 5, 6, 7])
  filter_arr = arr % 2 == 0
  newarr = arr[filter_arr]
  print(filter_arr) # [False  True False  True False  True False]
  print(newarr) # [2 4 6]
  ```
  ```

NumPy Random

- ```
  from numpy import random
  x = random.randint(100)
  print(x) # suppose 58
  ```
- `x = random.rand()`
- `x=random.randint(100, size=(5))`
- `x = random.randint(100, size=(3, 5)) # 2-D`
- `x = random.rand(5) #` 1-D array containing 5 random floats

```
random.seed(3)  # with seed
print(random.randint(1, 100))  # 65
random.seed(3)  # same seed
print(random.randint(1, 100))  # 65

# Without a seed (output will not be same)
print(random.randint(1, 100))  # 76
```
```

NumPy Universal Functions (ufuncs)

- Used to implement vectorization (Converting iterative statements into a vector-based operation)- element wise
- It is faster than iteration: vector do parallel computation
-
```
x = [1, 2, 3, 4]
y = [4, 5, 6, 7]
z = []
for i, j in zip(x, y):
  z.append(i + j)
print(z)
```

```
x = [1, 2, 3, 4]
y = [4, 5, 6, 7]
z = np.add(x, y)

print(z)
```

You can create your own ufunc

The `frompyfunc()` method takes the following arguments:
1. *function* - the name of the function.
2. *inputs* - the number of input arguments (arrays).
3. *outputs* - the number of output arrays.

- ```
  import numpy as np
  def myadd(x, y):
    return x+y
  myadd = np.frompyfunc(myadd, 2, 1)  # function, inputs, outputs
  print(myadd([1, 2, 3, 4], [5, 6, 7, 8]))
  ```

- ```
  print(type(np.add)) # <class 'numpy.ufunc'>
  ```
  - Check if a Function is a ufunc

NumPy Universal Functions (ufuncs)

- Simple arithmetic
- Rounding
- Log (at any base)
- Summations
- Products
- Trigonometric
- Hyperbolic
- Set operations and more...

NumPy Hands On and Quiz