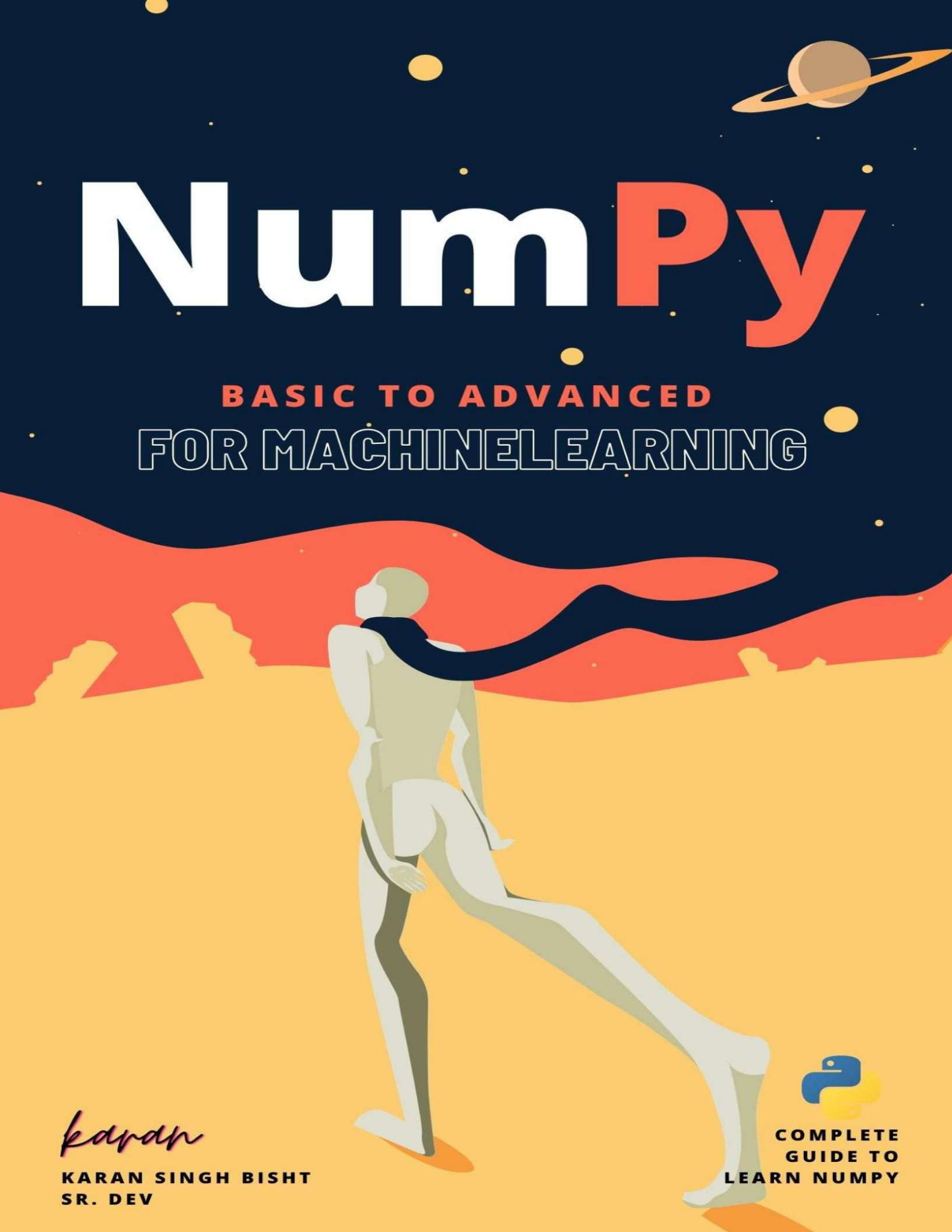
NumPy

BASIC TO ADVANCED

FOR MACHINELEARNING



karan

KARAN SINGH BISHT
SR. DEV

COMPLETE
GUIDE TO
LEARN NUMPY

NumPy
From Basic To Advanced

COVERED IN THIS BOOK ALL THE TOPICS OF NUMPY FROM BASIC TO ADVANCED

-Karan Singh Bisht

Senior Software Developer

- OUTLINE
- NumPy

- Learning by Reading
- Access 2-D Arrays
- Access 3-D Arrays
- Negative Indexing
- NumPy Array Slicing
- Slicing arrays
- Slicing 2-D Arrays
- Test Yourself With Exercises
- NumPy Data Types
- Data Types in Python
- Data Types in NumPy
- Checking the Data Type of an Array
- Creating Arrays With a Defined Data Type
- What if a Value Can Not Be Converted?
- Converting Data Type on Existing Arrays
- What if a Value Can Not Be Converted?
- Converting Data Type on Existing Arrays
- Check if Array Owns its Data
- NumPy Array Shape
- Shape of an Array
- Get the Shape of an Array
- What does the shape tuple represent?
- NumPy Array Reshaping
- Reshaping arrays
- Reshape From 1-D to 2-D
- Operations using NumPy
- NumPy – A Replacement for MatLab
- Learning by Examples
- NumPy Introduction
- What is NumPy?
- Why Use NumPy?
- Why is NumPy Faster Than Lists?
- Which Language is NumPy written in?
- Dimensions in Arrays
- 0-D Arrays
- 1-D Arrays
- 2-D Arrays
- 3-D arrays
- Check Number of Dimensions?
- Higher Dimensional Arrays
- NumPy Array Indexing
- Access Array Elements
- Reshape From 1-D to 3-D
- Can We Reshape Into any Shape?
- Unknown Dimension
- NumPy Array Iterating
- Iterating Arrays
- Iteratin 3-D Arrays
-
- Iterating Arrays Using nditer()
- Iterating on Each Scalar Element
- Iterating Array With Different Data Types
- Iterating With Different Step Size
- Enumerated Iteration Using ndenumerate()
- NumPy Joining Array
- Joining NumPy Arrays
- Joining Arrays Using Stack Functions
- Stacking Along Rows
- Stacking Along Columns
- Stacking Along Height (depth)
- NumPy Splitting Array
- Splitting NumPy Arrays
- Example
- Example
- Split Into Arrays
- Example
- Splitting 2-D Arrays
- NumPy Searching Arrays
- Searching Arrays
- Search Sorted
- Search From the Right Side
- Multiple Values
- NumPy Sorting Arrays
- Plotting a Distplot
- Plotting a Distplot Without the Histogram
- Normal (Gaussian) Distribution

- Where is the NumPy Codebase?
- Installation of NumPy
- Checking NumPy Version
- Create a NumPy ndarray Object
- Sorting Arrays
- Sorting a 2-D Array
- NumPy Filter Array
- Filtering Arrays
- Creating the Filter Array
- # Create an empty list
- # go through each element in arr
- # Create an empty list
- # go through each element in arr
- Random Numbers in NumPy
- What is a Random Number?
- Pseudo Random and True Random.
- Can we make truly random numbers?
- Generate Random Number
- Generate Random Float
- Generate Random Array
- Generate Random Number From Array
- Random Data Distribution
- What is Data Distribution?
- Random Distribution
- Random Permutations
- Random Permutations of Elements
- Shuffling Arrays
- Seaborn
- Visualize Distributions With Seaborn
- Install Seaborn.
- Distplots
- Import Matplotlib
- Import Seaborn
- Array siblings: chararray, maskedarray, matrix
- chararray: vectorized string operations
- masked_array missing data
- Key and Imports
- Normal Distribution
- Visualization of Normal Distribution
- Binomial Distribution
- Binomial Distribution is a Discrete Distribution.
- Visualization of Binomial Distribution
- Result
- Difference Between Normal and Binomial Distribution
- Poisson Distribution
- Poisson Distribution
- Visualization of Poisson Distribution
- Difference Between Normal and Poisson Distribution
- Difference Between Poisson and Binomial Distribution
- Uniform Distribution
- Uniform Distribution
- Visualization of Uniform Distribution
- Logistic Distribution
- Logistic Distribution
- Difference Between Logistic and Normal Distribution
- Multinomial Distribution
- Multinomial Distribution
- Exponential Distribution
- Visualization of Exponential Distribution
- Result
- Relation Between Poisson and Exponential Distribution
- Chi Square Distribution
- Chi Square Distribution
- Visualization of Chi Square Distribution
- Rayleigh Distribution
- Visualization of Rayleigh Distribution
- Similarity Between Rayleigh and Chi Square Distribution
- Pareto Distribution
- Pareto Distribution
- Visualization of Pareto Distribution
- Zipf Distribution
- Visualization of Zipf Distribution
- NumPy ufuncs

- Importing/exporting
- Creating Arrays
- Inspecting Properties
- Copying/sorting/reshaping
- Adding/removing Elements
- Combining/splitting
- Indexing/slicing/subsetting
- Scalar Math
- Vector Math
- Statistics
-
- Angles of Each Value in Arrays
- NumPy Set Operations
- What is a Set
- Create Sets in NumPy
- Finding Union
- Finding Intersection
- Finding Difference
- Finding Symmetric Difference
- Advanced NumPy
- Life of ndarray
- Block of memory
- >>>
- Re-interpretation / viewing
- Indexing scheme: strides
- C and Fortran order
- Broadcasting
- More tricks: diagonals
- CPU cache effects
- Findings in dissection
- Universal functions
- What they are?
- Parts of an Ufunc
- Making it easier
- Exercise: building an ufunc from scratch
- Solution: building an ufunc from scratch
- Generalized ufuncs
- Status in NumPy
- Generalized ufunc loop
- Interoperability features
- Sharing multidimensional, typed
- What are ufuncs?
- Why use ufuncs?
- What is Vectorization?
- Add the Elements of Two Lists
- Create Your Own ufunc
- How To Create Your Own ufunc
- Check if a Function is a ufunc
- Simple Arithmetic
- NumPy Summations
- Summations
- Summation Over an Axis
- Cummulative Sum
- NumPy Products
- Product Over an Axis
- Cummulative Product
- Example
- NumPy Differences
- Differences
- NumPy LCM Lowest Common Multiple
- Finding LCM (Lowest Common Multiple)
- Finding LCM in Arrays
- NumPy GCD Greatest Common Denominator
- Finding GCD (Greatest Common Denominator)
- Finding GCD in Arrays
- NumPy Trigonometric Functions
- Trigonometric Functions
- Convert Degrees Into Radians
- Radians to Degrees
- Finding Angles
- Angles of Each Value in Arrays
- Hypotenues
- NumPy Hyperbolic Functions
- Hyperbolic Functions
- Finding Angles

data • The old buffer protocol • The old buffer protocol • Array interface protocol	

NumPy

NumPy is a Python library.

NumPy is used for working with arrays.

NumPy is short for "Numerical Python".

Learning by Reading

Starting with a basic introduction and ends up with creating and plotting random data sets, and working with NumPy functions:

NumPy, which stands for Numerical Python, is a library consisting of multidimensional array objects and a collection of routines for processing those arrays. Using NumPy, mathematical and logical operations on arrays can be performed.

NumPy is a Python package. It stands for ‘Numerical Python’. It is a library consisting of multidimensional array objects and a collection of routines for processing of array.

Numeric, the ancestor of NumPy, was developed by Jim Hugunin. Another package Numarray was also developed, having some additional functionalities. In 2005, Travis Oliphant created NumPy package by incorporating the features of Numarray into Numeric package. There are many contributors to this open-source project.

Operations using NumPy

Using NumPy, a developer can perform the following operations −

- Mathematical and logical operations on arrays.
- Fourier transforms and routines for shape manipulation.
- Operations related to linear algebra. NumPy has in-built functions for linear algebra and random number generation.

NumPy – A Replacement for MatLab

NumPy is often used along with packages like **SciPy** (Scientific Python) and **Matplotlib** (plotting library). This combination is widely used as a replacement for MatLab, a popular platform for technical computing. However, Python alternative to MatLab is now seen as a more modern and complete programming language.

It is open-source, which is an added advantage of NumPy.

The most important object defined in NumPy is an N-dimensional array type called **ndarray**. It describes the collection of items of the same type. Items in the collection can be accessed using a zero-based index.

Every item in a ndarray takes the same size as the block in the memory. Each element in ndarray is an object of the data-type object (called **dtype**).

Any item extracted from ndarray object (by slicing) is represented by a Python object of one of array scalar types. The following diagram shows a relationship between ndarray, data-type object (dtype) and array scalar type −

An instance of ndarray class can be constructed by different array creation routines

described later in the tutorial. The basic ndarray is created using an array function in NumPy as follows-

numpy.array

It creates a ndarray from any object exposing an array interface, or from any method that returns an array.

Learning by Examples

In our "Try it Yourself" editor, you can use the NumPy module, and modify the code to see the result.

Example

Create a NumPy array:

```
import numpy as np

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

print(arr)

print(type(arr))
```

NumPy Introduction

What is NumPy?

NumPy is a Python library used for working with arrays.

It also has functions for working in domain of linear algebra, fourier transform, and matrices.

NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.

NumPy stands for Numerical Python.

Why Use NumPy?

In Python we have lists that serve the purpose of arrays, but they are slow to process.

NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.

The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.

Arrays are very frequently used in data science, where speed and resources are very important.

Data Science: is a branch of computer science where we study how to store, use and analyze data for deriving information from it.

Why is NumPy Faster Than Lists?

NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.

This behavior is called locality of reference in computer science.

This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

Which Language is NumPy written in?

NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.

Where is the NumPy Codebase?

The source code for NumPy is located at this github repository https://github.com/numpy/numpy

github: enables many people to work on the same codebase.

Installation of NumPy

If you have Python and PIP already installed on a system, then installation of NumPy is very easy.

Install it using this command:

```
C:\Users\Your Name>pip install numpy
```

If this command fails, then use a python distribution that already has NumPy installed like, Anaconda, Spyder etc.

Import NumPy

Once NumPy is installed, import it in your applications by adding the import keyword:

```
import numpy
```

Now NumPy is imported and ready to use.

Example

```
import numpy

arr = numpy.array([1, 2, 3, 4, 5])

print(arr)
```

NumPy as np

NumPy is usually imported under the np alias.

alias: In Python alias are an alternate name for referring to the same thing.

Create an alias with the as keyword while importing:

```
import numpy as np
```

Now the NumPy package can be referred to as np instead of numpy.

Example

```
import numpy as np

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

print(arr)
```

Checking NumPy Version

The version string is stored under __version__ attribute.

Example

```
import numpy as np

print(np.__version__)
```

NumPy Creating Arrays

Create a NumPy ndarray Object

NumPy is used to work with arrays. The array object in NumPy is called ndarray.

We can create a NumPy ndarray object by using the array() function.

Example

```
import numpy as np

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

print(arr)

print(type(arr))
```

type(): This built-in Python function tells us the type of the object passed to it. Like in above code it shows that arr is numpy.ndarray type.

To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray:

Example

Use a tuple to create a NumPy array:

```
import numpy as np

arr = np.array((1, 2, 3, 4, 5))

print(arr)
```

Dimensions in Arrays

A dimension in arrays is one level of array depth (nested arrays).

nested array: are arrays that have arrays as their elements.

0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

Example

Create a 0-D array with value 42

```
import numpy as np

arr = np.array(42)

print(arr)
```

1-D Arrays

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.

These are the most common and basic arrays.

Example

Create a 1-D array containing the values 1,2,3,4,5:

```
import numpy as np

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

print(arr)
```

2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array.

These are often used to represent matrix or 2nd order tensors.

NumPy has a whole sub module dedicated towards matrix operations called numpy.mat

Example

Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6:

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

print(arr)
```

3-D arrays

An array that has 2-D arrays (matrices) as its elements is called 3-D array.

These are often used to represent a 3rd order tensor.

Example

Create a 3-D array with two 2-D arrays, both containing two arrays with the values 1,2,3 and 4,5,6:

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(arr)
```

Check Number of Dimensions?

NumPy Arrays provides the ndim attribute that returns an integer that tells us how many dimensions the array have.

Example

Check how many dimensions the arrays have:

```
import numpy as np


a = np.array(42)

b = np.array([1, 2, 3, 4, 5])

c = np.array([[1, 2, 3], [4, 5, 6]])

d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])


print(a.ndim)

print(b.ndim)

print(c.ndim)

print(d.ndim)
```

Higher Dimensional Arrays

An array can have any number of dimensions.

When the array is created, you can define the number of dimensions by using the ndmin argument.

Example

Create an array with 5 dimensions and verify that it has 5 dimensions:

```
import numpy as np

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

print(arr)
```

```
print('number of dimensions :', arr.ndim)
```

In this array the innermost dimension (5th dim) has 4 elements, the 4th dim has 1 element that is the vector, the 3rd dim has 1 element that is the matrix with the vector, the 2nd dim has 1 element that is 3D array and 1st dim has 1 element that is a 4D array.

NumPy Array Indexing

Access Array Elements

Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

Example

Get the first element from the following array:

```
import numpy as np

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

print(arr[0])
```

Example

Get the second element from the following array.

```
import numpy as np

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

print(arr[1])
```

Example

Get third and fourth elements from the following array and add them.

```
import numpy as np

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

print(arr[2] + arr[3])
```

Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Think of 2-D arrays like a table with rows and columns, where the row represents the dimension and the index represents the column.

Example

Access the element on the first row, second column:

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('2nd element on 1st row: ', arr[0, 1])
```

Example

Access the element on the 2nd row, 5th column:

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('5th element on 2nd row: ', arr[1, 4])
```

Access 3-D Arrays

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

Example

Access the third element of the second array of the first array:

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

print(arr[0, 1, 2])
```

Example Explained

arr[0, 1, 2] prints the value 6.

And this is why:

The first number represents the first dimension, which contains two arrays:

[[1, 2, 3], [4, 5, 6]]

and:

[[7, 8, 9], [10, 11, 12]]

Since we selected 0, we are left with the first array:

[[1, 2, 3], [4, 5, 6]]

The second number represents the second dimension, which also contains

two arrays:

[1, 2, 3]

and:

[4, 5, 6]

Since we selected 1, we are left with the second array:

[4, 5, 6]

The third number represents the third dimension, which contains three values:

4

5

6

Since we selected 2, we end up with the third value:

6

Negative Indexing

Use negative indexing to access an array from the end.

Example

Print the last element from the 2nd dim:

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('Last element from 2nd dim: ', arr[1, -1])
```

NumPy Array Slicing

Slicing arrays

Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index like this: [*start*:*end*].

We can also define the step, like this: [*start*:*end*:*step*].

If we don't pass start its considered 0

If we don't pass end its considered length of array in that dimension

If we don't pass step its considered 1

Example

Slice elements from index 1 to index 5 from the following array:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5])
```

Note: The result *includes* the start index, but *excludes* the end index.

Example

Slice elements from index 4 to the end of the array:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[4:])
```

Example

Slice elements from the beginning to index 4 (not included):

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[:4])
```

Negative Slicing

Use the minus operator to refer to an index from the end:

Example

Slice from the index 3 from the end to index 1 from the end:

```
import numpy as n
arr = np.array([1, 2, 3, 4, 5, 6, 7])
```

```
print(arr[-3:-1])
```

STEP

Use the step value to determine the step of the slicing:

Example

Return every other element from index 1 to index 5:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5:2])
```

Example

Return every other element from the entire array:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[::2])
```

Slicing 2-D Arrays

Example

From the second element, slice elements from index 1 to index 4 (not included):

```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[1, 1:4])
```

Note: Remember that *second element* has index 1.

Example

From both elements, return index 2:

```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[0:2, 2])
```

Example

From both elements, slice index 1 to index 4 (not included), this will return a 2-D array:

```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
```

```
print(arr[0:2, 1:4])
```

Exercise:

Insert the correct slicing syntax to print the following selection of the array:

Everything from (including) the second item to (not including) the fifth item.

```
arr = np.array([10, 15, 20, 25, 30, 35, 40])

print(arr)
```

NumPy Data Types

Data Types in Python

By default Python have these data types:

- strings - used to represent text data, the text is given under quote marks. e.g. "ABCD"
- integer - used to represent integer numbers. e.g. -1, -2, -3
- float - used to represent real numbers. e.g. 1.2, 42.42
- boolean - used to represent True or False.
- complex - used to represent complex numbers. e.g. 1.0 + 2.0j, 1.5 +

Data Types in NumPy

NumPy has some extra data types, and refer to data types with one character, like i for integers, u for unsigned integers etc.

Below is a list of all data types in NumPy and the characters used to represent them.

- 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)

Checking the Data Type of an Array

The NumPy array object has a property called dtype that returns the data type of the array:

Example

Get the data type of an array object:

```
import numpy as np
```

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


print(arr.dtype)
```

Example

Get the data type of an array containing strings:

```
import numpy as np


arr = np.array(['apple', 'banana', 'cherry'])


print(arr.dtype)
```

Creating Arrays With a Defined Data Type

We use the array() function to create arrays, this function can take an optional argument: dtype that allows us to define the expected data type of the array elements:

Example

Create an array with data type string:

```
import numpy as np

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

```
print(arr)

print(arr.dtype)
```

For i, u, f, S and U we can define size as well.

Example

Create an array with data type 4 bytes integer:

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='i4')

print(arr)

print(arr.dtype)
```

What if a Value Can Not Be Converted?

If a type is given in which elements can't be casted then NumPy will raise a ValueError.

ValueError: In Python ValueError is raised when the type of passed argument to a function is unexpected/incorrect.

Example

A non integer string like 'a' can not be converted to integer (will raise an error):

```
import numpy as np

arr = np.array(['a', '2', '3'], dtype='i')
```

Converting Data Type on Existing Arrays

The best way to change the data type of an existing array, is to make a copy of the array with the astype() method.

The astype() function creates a copy of the array, and allows you to specify the data type as a parameter.

The data type can be specified using a string, like 'f' for float, 'i' for integer etc. or you can use the data type directly like float for float and int for integer.

Example

Change data type from float to integer by using 'i' as parameter value:

```
import numpy as np

arr = np.array([1.1, 2.1, 3.1])

newarr = arr.astype('i')

print(newarr)

print(newarr.dtype)
```

Example

Change data type from float to integer by using int as parameter value:

```
import numpy as np

arr = np.array([1.1, 2.1, 3.1])

newarr = arr.astype(int)

print(newarr)

print(newarr.dtype)
```

Example

Change data type from integer to boolean:

```
import numpy as np

arr = np.array([1, 0, 3])

newarr = arr.astype(bool)

print(newarr)

print(newarr.dtype)
```

What if a Value Can Not Be Converted?

If a type is given in which elements can't be casted then NumPy will raise a ValueError.

ValueError: In Python ValueError is raised when the type of passed argument to a function is unexpected/incorrect.

Example

A non integer string like 'a' can not be converted to integer (will raise an error):

```
import numpy as np

arr = np.array(['a', '2', '3'], dtype='i')
```

Converting Data Type on Existing Arrays

The best way to change the data type of an existing array, is to make a copy of the array with the astype() method.

The astype() function creates a copy of the array, and allows you to specify the data type as a parameter.

The data type can be specified using a string, like 'f' for float, 'i' for integer etc. or you can use the data type directly like float for float and int for integer.

Example

Change data type from float to integer by using 'i' as parameter value:

```
import numpy as np

arr = np.array([1.1, 2.1, 3.1])

newarr = arr.astype('i')

print(newarr)

print(newarr.dtype)
```

Example

Change data type from float to integer by using int as parameter value:

```
import numpy as np

arr = np.array([1.1, 2.1, 3.1])

newarr = arr.astype(int)

print(newarr)

print(newarr.dtype)
```

Example

Change data type from integer to boolean:

```
import numpy as np

arr = np.array([1, 0, 3])

newarr = arr.astype(bool)

print(newarr)

print(newarr.dtype)
```

Check if Array Owns its Data

As mentioned above, copies *owns* the data, and views *does not own* the data, but how can we check this?

Every NumPy array has the attribute base that returns None if the array owns the data.

Otherwise, the base attribute refers to the original object.

Example

Print the value of the base attribute to check if an array owns it's data or not:

```
import numpy as np

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

x = arr.copy()

y = arr.view()
```

```
print(x.base)

print(y.base)
```

The copy returns None.

The view returns the original array.

NumPy Array Shape

Shape of an Array

The shape of an array is the number of elements in each dimension.

Get the Shape of an Array

NumPy arrays have an attribute called shape that returns a tuple with each index having the number of corresponding elements.

Example

Print the shape of a 2-D array:

```
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

print(arr.shape)
```

The example above returns (2, 4), which means that the array has 2 dimensions, where the first dimension has 2 elements and the second has 4.

Example

Create an array with 5 dimensions using ndmin using a vector with values 1,2,3,4 and verify that last dimension has value 4:

```
import numpy as np

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

print(arr)

print('shape of array :', arr.shape)
```

What does the shape tuple represent?

Integers at every index tells about the number of elements the corresponding dimension has.

In the example above at index-4 we have value 4, so we can say that 5th (4 + 1 th) dimension has 4 elements.

NumPy Array Reshaping

Reshaping arrays

Reshaping means changing the shape of an array.

The shape of an array is the number of elements in each dimension.

By reshaping we can add or remove dimensions or change number of elements in each dimension.

Reshape From 1-D to 2-D

Example

Convert the following 1-D array with 12 elements into a 2-D array.

The outermost dimension will have 4 arrays, each with 3 elements:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(4, 3)

print(newarr)
```

Reshape From 1-D to 3-D

Example

Convert the following 1-D array with 12 elements into a 3-D array.

The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(2, 3, 2)

print(newarr)
```

Can We Reshape Into any Shape?

Yes, as long as the elements required for reshaping are equal in both shapes.

We can reshape an 8 elements 1D array into 4 elements in 2 rows 2D array but we cannot reshape it into a 3 elements 3 rows 2D array as that would require 3x3 = 9 elements.

Example

Try converting 1D array with 8 elements to a 2D array with 3 elements in each dimension (will raise an error):

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

newarr = arr.reshape(3, 3)

print(newarr)
```

Returns Copy or View?

Example

Check if the returned array is a copy or a view:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

print(arr.reshape(2, 4).base)
```

The example above returns the original array, so it is a view.

Unknown Dimension

You are allowed to have one "unknown" dimension.

Meaning that you do not have to specify an exact number for one of the dimensions in the reshape method.

Pass -1 as the value, and NumPy will calculate this number for you.

Example

Convert 1D array with 8 elements to 3D array with 2x2 elements:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

newarr = arr.reshape(2, 2, -1)

print(newarr)
```

Note: We can not pass -1 to more than one dimension.

Flattening the arrays

Flattening array means converting a multidimensional array into a 1D array.

We can use reshape(-1) to do this.

Example

Convert the array into a 1D array:

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

newarr = arr.reshape(-1)

print(newarr)
```

Note: There are a lot of functions for changing the shapes of arrays in numpy flatten, ravel and also for rearranging the elements rot90, flip, fliplr, flipud

etc. These fall under Intermediate to Advanced section of numpy.

NumPy Array Iterating

Iterating Arrays

Iterating means going through elements one by one.

As we deal with multi-dimensional arrays in numpy, we can do this using basic for loop of python.

If we iterate on a 1-D array it will go through each element one by one.

Example

Iterate on the elements of the following 1-D array:

```
import numpy as np

arr = np.array([1, 2, 3])

for x in arr:

  print(x)
```

Iterating 2-D Arrays

In a 2-D array it will go through all the rows.

Example

Iterate on the elements of the following 2-D array:

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

for x in arr:

  print(x)
```

If we iterate on a *n*-D array it will go through n-1th dimension one by one.

To return the actual values, the scalars, we have to iterate the arrays in each dimension.

Example

Iterate on each scalar element of the 2-D array:

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

for x in arr:

  for y in x:

    print(y)
```

Iterating 3-D Arrays

In a 3-D array it will go through all the 2-D arrays.

Example

Iterate on the elements of the following 3-D array:

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

for x in arr:

  print(x)
```

To return the actual values, the scalars, we have to iterate the arrays in each dimension.

Example

Iterate down to the scalars:

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

for x in arr:

  for y in x:

    for z in y:

        print(z)
```

Iterating Arrays Using nditer()

The function nditer() is a helping function that can be used from very basic to very advanced iterations. It solves some basic issues which we face in iteration, lets go through it with examples.

Iterating on Each Scalar Element

In basic for loops, iterating through each scalar of an array we need to use *n* for loops which can be difficult to write for arrays with very high dimensionality.

Example

Iterate through the following 3-D array:

```
import numpy as np

arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

for x in np.nditer(arr):

  print(x)
```

Iterating Array With Different Data Types

We can use op_dtypes argument and pass it the expected datatype to change the datatype of elements while iterating.

NumPy does not change the data type of the element in-place (where the element is in array) so it needs some other space to perform this action, that extra space is called buffer, and in order to enable it in nditer() we pass flags=['buffered'].

Example

Iterate through the array as a string:

```
import numpy as np

arr = np.array([1, 2, 3])

for x in np.nditer(arr, flags=['buffered'], op_dtypes=['S']):

  print(x)
```

Iterating With Different Step Size

We can use filtering and followed by iteration.

Example

Iterate through every scalar element of the 2D array skipping 1 element:

```
import numpy as np


arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])


for x in np.nditer(arr[:, ::2]):

  print(x)
```

Enumerated Iteration Using ndenumerate()

Enumeration means mentioning sequence number of somethings one by one.

Sometimes we require corresponding index of the element while iterating, the ndenumerate() method can be used for those usecases.

Example

Enumerate on following 1D arrays elements:

```
import numpy as np

arr = np.array([1, 2, 3])

for idx, x in np.ndenumerate(arr):

  print(idx, x)
```

Example

Enumerate on following 2D array's elements:

```
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

for idx, x in np.ndenumerate(arr):

  print(idx, x)
```

NumPy Joining Array

Joining NumPy Arrays

Joining means putting contents of two or more arrays in a single array.

In SQL we join tables based on a key, whereas in NumPy we join arrays by axes.

We pass a sequence of arrays that we want to join to the concatenate() function, along with the axis. If axis is not explicitly passed, it is taken as 0.

Example

Join two arrays

```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.concatenate((arr1, arr2))

print(arr)
```

Example

Join two 2-D arrays along rows (axis=1):

```
import numpy as np

arr1 = np.array([[1, 2], [3, 4]])

arr2 = np.array([[5, 6], [7, 8]])

arr = np.concatenate((arr1, arr2), axis=1)

print(arr)
```

Joining Arrays Using Stack Functions

Stacking is same as concatenation, the only difference is that stacking is done along a new axis.

We can concatenate two 1-D arrays along the second axis which would result in putting them one over the other, ie. stacking.

We pass a sequence of arrays that we want to join to the stack() method along with the axis. If axis is not explicitly passed it is taken as 0.

Example

```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.stack((arr1, arr2), axis=1)

print(arr)
```

Stacking Along Rows

NumPy provides a helper function: hstack() to stack along rows.

Example

```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.hstack((arr1, arr2))

print(arr)
```

Stacking Along Columns

NumPy provides a helper function: vstack() to stack along columns.

Example

```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.vstack((arr1, arr2))

print(arr)
```

Stacking Along Height (depth)

NumPy provides a helper function: dstack() to stack along height, which is the same as depth.

Example

```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.dstack((arr1, arr2))

print(arr)
```

NumPy Splitting Array

Splitting NumPy Arrays

Splitting is reverse operation of Joining.

Joining merges multiple arrays into one and Splitting breaks one array into multiple.

We use array_split() for splitting arrays, we pass it the array we want to split and the number of splits.

Example

Split the array in 3 parts:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6])

newarr = np.array_split(arr, 3)
```

```
print(newarr)
```

Note: The return value is an array containing three arrays.

If the array has less elements than required, it will adjust from the end accordingly.

Example

Split the array in 4 parts:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6])

newarr = np.array_split(arr, 4)

print(newarr)
```

Note: We also have the method split() available but it will not adjust the elements when elements are less in source array for splitting like in example above, array_split() worked properly but split() would fail.

Split Into Arrays

The return value of the array_split() method is an array containing each of the split as an array.

If you split an array into 3 arrays, you can access them from the result just like any array element:

Example

Access the splitted arrays:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6])

newarr = np.array_split(arr, 3)

print(newarr[0])

print(newarr[1])

print(newarr[2])
```

Splitting 2-D Arrays

Use the same syntax when splitting 2-D arrays.

Use the array_split() method, pass in the array you want to split and the number of splits you want to do.

Example

Split the 2-D array into three 2-D arrays.

```
import numpy as np

arr = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])

newarr = np.array_split(arr, 3)

print(newarr)
```

The example above returns three 2-D arrays.

Let's look at another example, this time each element in the 2-D arrays contains 3 elements.

Example

Split the 2-D array into three 2-D arrays.

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18]])

newarr = np.array_split(arr, 3)

print(newarr)
```

The example above returns three 2-D arrays.

In addition, you can specify which axis you want to do the split around.

The example below also returns three 2-D arrays, but they are split along the row (axis=1).

Example

Split the 2-D array into three 2-D arrays along rows.

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18]])

newarr = np.array_split(arr, 3, axis=1)

print(newarr)
```

An alternate solution is using hsplit() opposite of hstack()

Example

Use the hsplit() method to split the 2-D array into three 2-D arrays along rows.

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18]])

newarr = np.hsplit(arr, 3)

print(newarr)
```

Note: Similar alternates to vstack() and dstack() are available as vsplit() and dsplit().

NumPy Searching Arrays

Searching Arrays

You can search an array for a certain value, and return the indexes that get a match.

To search an array, use the where() method.

Example

Find the indexes where the value is 4:

```
import numpy as np

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

x = np.where(arr == 4)
```

```
print(x)
```

The example above will return a tuple: (array([3, 5, 6],)

Which means that the value 4 is present at index 3, 5, and 6.

Example

Find the indexes where the values are even:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

x = np.where(arr%2 == 0)

print(x)
```

Example

Find the indexes where the values are odd:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

x = np.where(arr%2 == 1)

print(x)
```

Search Sorted

There is a method called searchsorted() which performs a binary search in the array, and returns the index where the specified value would be inserted to

maintain the search order.

The searchsorted() method is assumed to be used on sorted arrays.

Example

Find the indexes where the value 7 should be inserted:

```
import numpy as np

arr = np.array([6, 7, 8, 9])

x = np.searchsorted(arr, 7)

print(x)
```

Example explained: The number 7 should be inserted on index 1 to remain the sort order.

The method starts the search from the left and returns the first index where the number 7 is no longer larger than the next value.

Search From the Right Side

By default the left most index is returned, but we can give side='right' to return the right most index instead.

Example

Find the indexes where the value 7 should be inserted, starting from the right:

```
import numpy as np

arr = np.array([6, 7, 8, 9])

x = np.searchsorted(arr, 7, side='right')

print(x)
```

Example explained: The number 7 should be inserted on index 2 to remain the sort order.

The method starts the search from the right and returns the first index where the number 7 is no longer less than the next value.

Multiple Values

To search for more than one value, use an array with the specified values.

Example

Find the indexes where the values 2, 4, and 6 should be inserted:

```
import numpy as np

arr = np.array([1, 3, 5, 7])

x = np.searchsorted(arr, [2, 4, 6])

print(x)
```

The return value is an array: [1 2 3] containing the three indexes where 2, 4, 6 would be inserted in the original array to maintain the order.

NumPy Sorting Arrays

Sorting Arrays

Sorting means putting elements in an *ordered sequence*.

Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.

The NumPy ndarray object has a function called sort(), that will sort a specified array.

Example

Sort the array:

```
import numpy as np

arr = np.array([3, 2, 0, 1])

print(np.sort(arr))
```

Note: This method returns a copy of the array, leaving the original array unchanged.

You can also sort arrays of strings, or any other data type:

Example

Sort the array alphabetically:

```
import numpy as np

arr = np.array(['banana', 'cherry', 'apple'])

print(np.sort(arr))
```

Example

Sort a boolean array:

```
import numpy as np

arr = np.array([True, False, True])

print(np.sort(arr))
```

Sorting a 2-D Array

If you use the sort() method on a 2-D array, both arrays will be sorted:

Example

Sort a 2-D array:

```
import numpy as np

arr = np.array([[3, 2, 4], [5, 0, 1]])

print(np.sort(arr))
```

NumPy Filter Array

Filtering Arrays

Getting some elements out of an existing array and creating a new array out of them is called *filtering*.

In NumPy, you filter an array using a *boolean index list*.

A *boolean index list* is a list of booleans corresponding to indexes in the array.

If the value at an index is True that element is contained in the filtered array, if the value at that index is False that element is excluded from the filtered array.

Example

Create an array from the elements on index 0 and 2:

```
import numpy as np

arr = np.array([41, 42, 43, 44])

x = [True, False, True, False]

newarr = arr[x]

print(newarr)
```

The example above will return [41, 43], why?

Because the new filter contains only the values where the filter array had the value True, in this case, index 0 and 2.

Creating the Filter Array

In the example above we hard-coded the True and False values, but the common use is to create a filter array based on conditions.

Example

Create a filter array that will return only values higher than 42:

```
import numpy as np

arr = np.array([41, 42, 43, 44])

# Create an empty list

filter_arr = []

# go through each element in arr

for element in arr:

  # if the element is higher than 42, set the value to True, otherwise False:

  if element > 42:
```

```
    filter_arr.append(True)
  else:
      filter_arr.append(False)
newarr = arr[filter_arr]
print(filter_arr)
print(newarr)
```

Example

Create a filter array that will return only even elements from the original array:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
# Create an empty list
filter_arr = []
# go through each element in arr
for element in arr:
  # if the element is completely divisble by 2, set the value to True, otherwise False
  if element % 2 == 0:
    filter_arr.append(True)
  else:
```

```
    filter_arr.append(False)

newarr = arr[filter_arr]

print(filter_arr)

print(newarr)
```

Creating Filter Directly From Array

The above example is quite a common task in NumPy and NumPy provides a nice way to tackle it.

We can directly substitute the array instead of the iterable variable in our condition and it will work just as we expect it to.

Example

Create a filter array that will return only values higher than 42:

```
import numpy as np


arr = np.array([41, 42, 43, 44])


filter_arr = arr > 42


newarr = arr[filter_arr]
```

```
print(filter_arr)

print(newarr)
```

Example

Create a filter array that will return only even elements from the original array:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

filter_arr = arr % 2 == 0

newarr = arr[filter_arr]

print(filter_arr)

print(newarr)
```

Random Numbers in NumPy

What is a Random Number?

Random number does NOT mean a different number every time. Random means something that can

not be predicted logically.

Pseudo Random and True Random.

Computers work on programs, and programs are definitive set of instructions. So it means there must be some algorithm to generate a random number as well.

If there is a program to generate random number it can be predicted, thus it is not truly random.

Random numbers generated through a generation algorithm are called *pseudo random*.

Can we make truly random numbers?

Yes. In order to generate a truly random number on our computers we need to get the random data from some

outside source. This outside source is generally our keystrokes, mouse movements, data on network

etc.

We do not need truly random numbers, unless its related to security (e.g. encryption keys) or the basis of application is the randomness (e.g. Digital roulette wheels).

In this tutorial we will be using pseudo random numbers.

Generate Random Number

NumPy offers the random module to work with random numbers.

Example

Generate a random integer from 0 to 100:

```
from numpy import random

x = random.randint(100)

print(x)
```

Generate Random Float

The random module's rand() method returns a random float between 0 and 1.

Example

Generate a random float from 0 to 1:

```
from numpy import random

x = random.rand()

print(x)
```

Generate Random Array

In NumPy we work with arrays, and you can use the two methods from the above examples to make random arrays.

Integers

The randint() method takes a size parameter where you can specify the shape of an array.

Example

Generate a 1-D array containing 5 random integers from 0 to 100:

```
from numpy import random

x=random.randint(100, size=(5))

print(x)
```

Example

Generate a 2-D array with 3 rows, each row containing 5 random integers from 0 to 100:

```
from numpy import random

x = random.randint(100, size=(3, 5))

print(x)
```

Floats

The rand() method also allows you to specify the shape of the array.

Example

Generate a 1-D array containing 5 random floats:

```
from numpy import random

x = random.rand(5)

print(x)
```

Example

Generate a 2-D array with 3 rows, each row containing 5 random numbers:

```
from numpy import random

x = random.rand(3, 5)

print(x)
```

Generate Random Number From Array

The choice() method allows you to generate a random value based on an array of values.

The choice() method takes an array as a parameter and randomly returns one of the values.

Example

Return one of the values in an array:

```
from numpy import random

x = random.choice([3, 5, 7, 9])

print(x)
```

The choice() method also allows you to return an *array* of values.

Add a size parameter to specify the shape of the array.

Example

Generate a 2-D array that consists of the values in the array parameter (3, 5, 7, and 9):

```
from numpy import random

x = random.choice([3, 5, 7, 9], size=(3, 5))

print(x)
```

Random Data Distribution

What is Data Distribution?

Data Distribution is a list of all possible values, and how often each value occurs.

Such lists are important when working with statistics and data science.

The random module offer methods that returns randomly generated data distributions.

Random Distribution

A random distribution is a set of random numbers that follow a certain *probability density function*.

Probability Density Function: A function that describes a continuous probability. i.e. probability of all values in an array.

We can generate random numbers based on defined probabilities using the choice() method of the random module.

The choice() method allows us to specify the probability for each value.

The probability is set by a number between 0 and 1, where 0 means that the value will never occur and 1 means that the value will always occur.

Example

Generate a 1-D array containing 100 values, where each value has to be 3, 5, 7 or 9.

The probability for the value to be 3 is set to be 0.1

The probability for the value to be 5 is set to be 0.3

The probability for the value to be 7 is set to be 0.6

The probability for the value to be 9 is set to be 0

```
from numpy import random

x = random.choice([3, 5, 7, 9], p=[0.1, 0.3, 0.6, 0.0], size=(100))

print(x)
```

The sum of all probability numbers should be 1.

Even if you run the example above 100 times, the value 9 will never occur.

You can return arrays of any shape and size by specifying the shape in the size parameter.

Example

Same example as above, but return a 2-D array with 3 rows, each containing 5 values.

```
from numpy import random

x = random.choice([3, 5, 7, 9], p=[0.1, 0.3, 0.6, 0.0], size=(3, 5))

print(x)
```

Random Permutations

Random Permutations of Elements

A permutation refers to an arrangement of elements. e.g. [3, 2, 1] is a permutation of [1, 2, 3] and vice-versa.

The NumPy Random module provides two methods for this: shuffle() and permutation().

Shuffling Arrays

Shuffle means changing arrangement of elements in-place. i.e. in the array

itself.

Example

Randomly shuffle elements of following array:

```
from numpy import random

import numpy as np

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

random.shuffle(arr)

print(arr)
```

The shuffle() method makes changes to the original array.

Generating Permutation of Arrays

Example

Generate a random permutation of elements of following array:

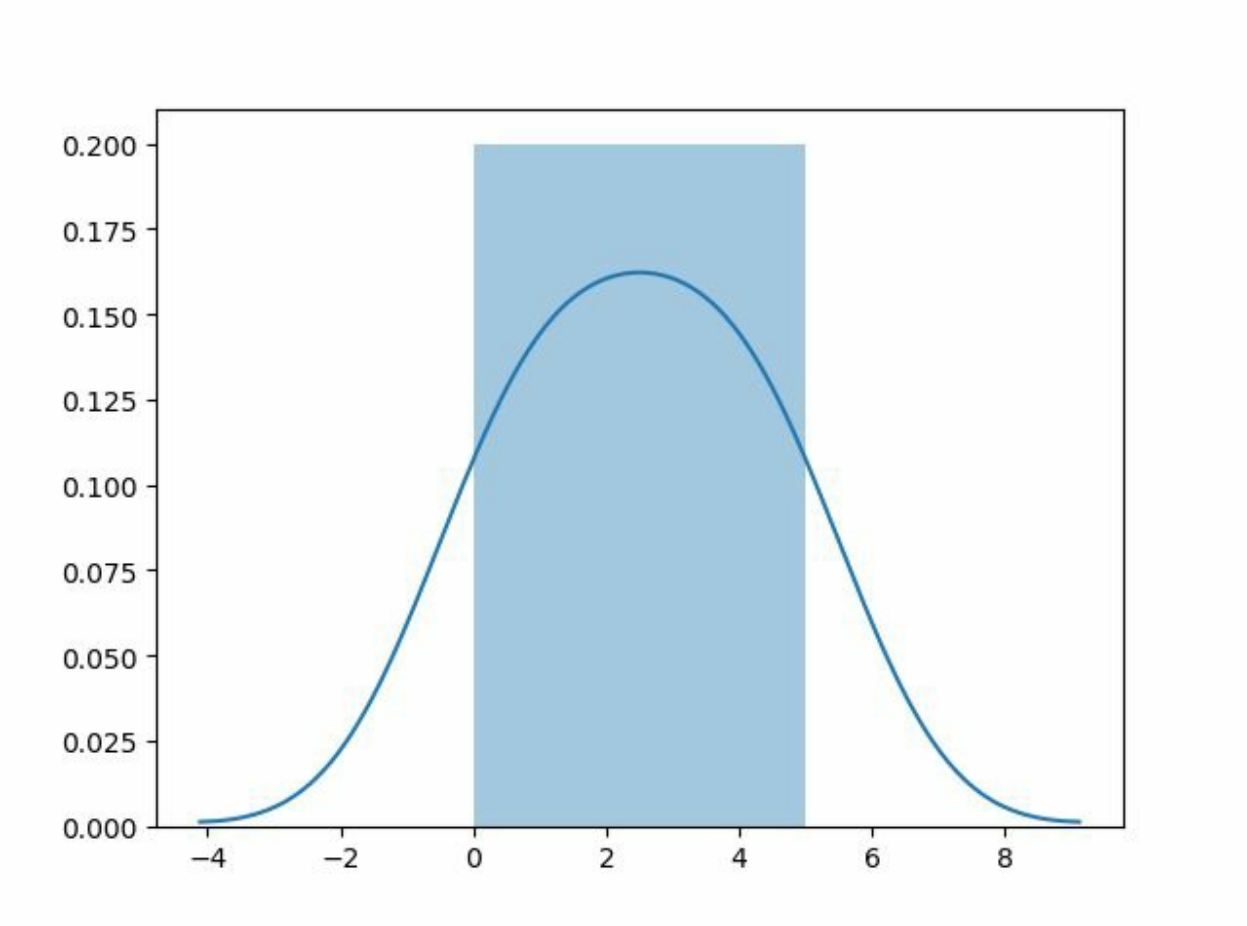
```
from numpy import random

import numpy as np

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

print(random.permutation(arr))
```

The permutation() method *returns* a re-arranged array (and leaves the original array un-changed).

Seaborn

Visualize Distributions With Seaborn

Seaborn is a library that uses Matplotlib underneath to plot graphs. It will be used to visualize random distributions.



Install Seaborn.

If you have Python and PIP already installed on a system, install it using this command:

```
C:\Users\Your Name>pip install seaborn
```

If you use Jupyter, install Seaborn using this command:

```
C:\Users\Your Name>!pip install seaborn
```

Distplots

Distplot stands for distribution plot, it takes as input an array and plots a curve corresponding to the distribution of points in the array.

Import Matplotlib

Import the pyplot object of the Matplotlib module in your code using the following statement:

```
import matplotlib.pyplot as plt
```

You can learn about the Matplotlib module in our Matplotlib Tutorial.

Import Seaborn

Import the Seaborn module in your code using the following statement:

```
import seaborn as sns
```

Plotting a Distplot

Example

```
import matplotlib.pyplot as plt

import seaborn as sns

sns.distplot([0, 1, 2, 3, 4, 5])

plt.show()
```

Plotting a Distplot Without the Histogram

Example

```
import matplotlib.pyplot as plt

import seaborn as sns

sns.distplot([0, 1, 2, 3, 4, 5], hist=False)

plt.show()
```

Note: We will be using: sns.distplot(arr, hist=False) to visualize random distributions in this tutorial.

Normal (Gaussian) Distribution

Normal Distribution

The Normal Distribution is one of the most important distributions.

It is also called the Gaussian Distribution after the German mathematician Carl Friedrich Gauss.

It fits the probability distribution of many events, eg. IQ Scores, Heartbeat etc.

Use the random.normal() method to get a Normal Data Distribution.

It has three parameters:

loc - (Mean) where the peak of the bell exists.

scale - (Standard Deviation) how flat the graph distribution should be.

size - The shape of the returned array.

Example

Generate a random normal distribution of size 2x3:

```
from numpy import random

x = random.normal(size=(2, 3))

print(x)
```

Example

Generate a random normal distribution of size 2x3 with mean at 1 and standard deviation of 2:

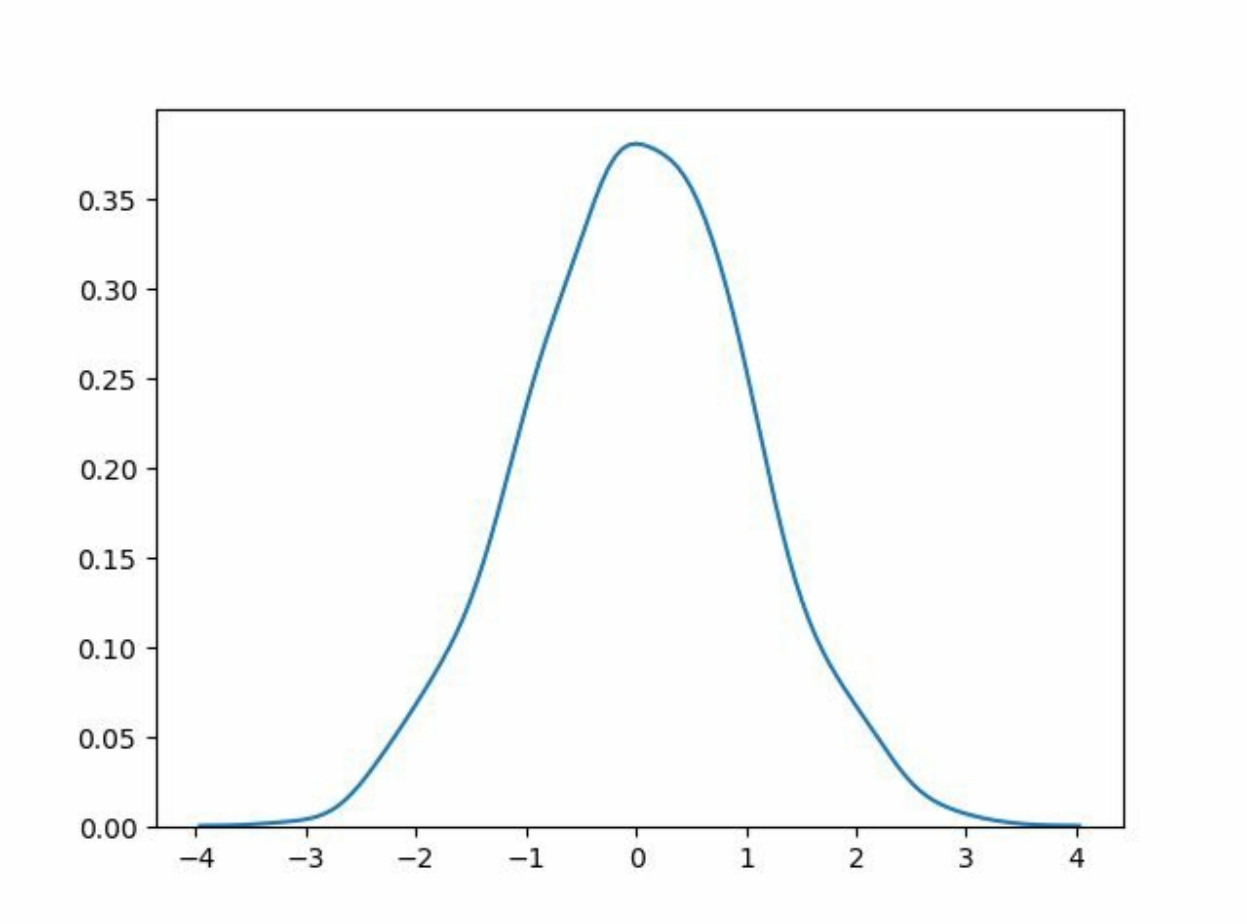
```
from numpy import random

x = random.normal(loc=1, scale=2, size=(2, 3))

print(x)
```

Visualization of Normal Distribution

Example

```
from numpy import random
```

```
import matplotlib.pyplot as plt

import seaborn as sns

sns.distplot(random.normal(size=1000), hist=False)

plt.show()
```



Note: The curve of a Normal Distribution is also known as the Bell Curve because of the bell-shaped curve.

Binomial Distribution

Binomial Distribution is a *Discrete Distribution*.

It describes the outcome of binary scenarios, e.g. toss of a coin, it will either be head or tails.

It has three parameters:

n - number of trials.

p - probability of occurence of each trial (e.g. for toss of a coin 0.5 each).

size - The shape of the returned array.

Discrete Distribution:The distribution is defined at separate set of events, e.g. a coin toss's result is discrete as it can be only head or tails whereas height of people is continuous as it can be 170, 170.1, 170.11 and so on.

Example

Given 10 trials for coin toss generate 10 data points:

```
from numpy import random

x = random.binomial(n=10, p=0.5, size=10)

print(x)
```

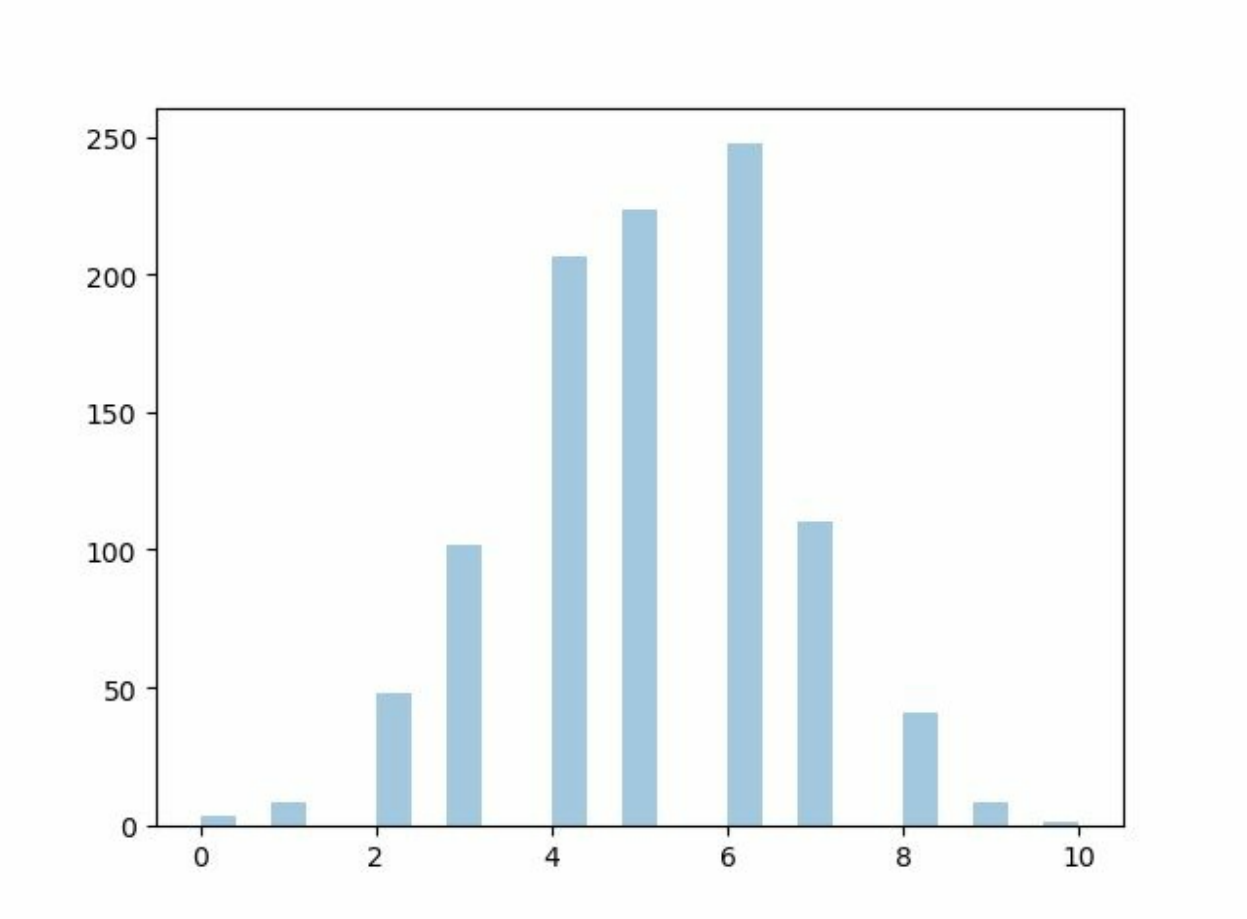
Visualization of Binomial Distribution

Example

```
from numpy import random

import matplotlib.pyplot as plt

import seaborn as sns
```

```
sns.distplot(random.binomial(n=10, p=0.5, size=1000), hist=True,
kde=False)

plt.show()
```

Result



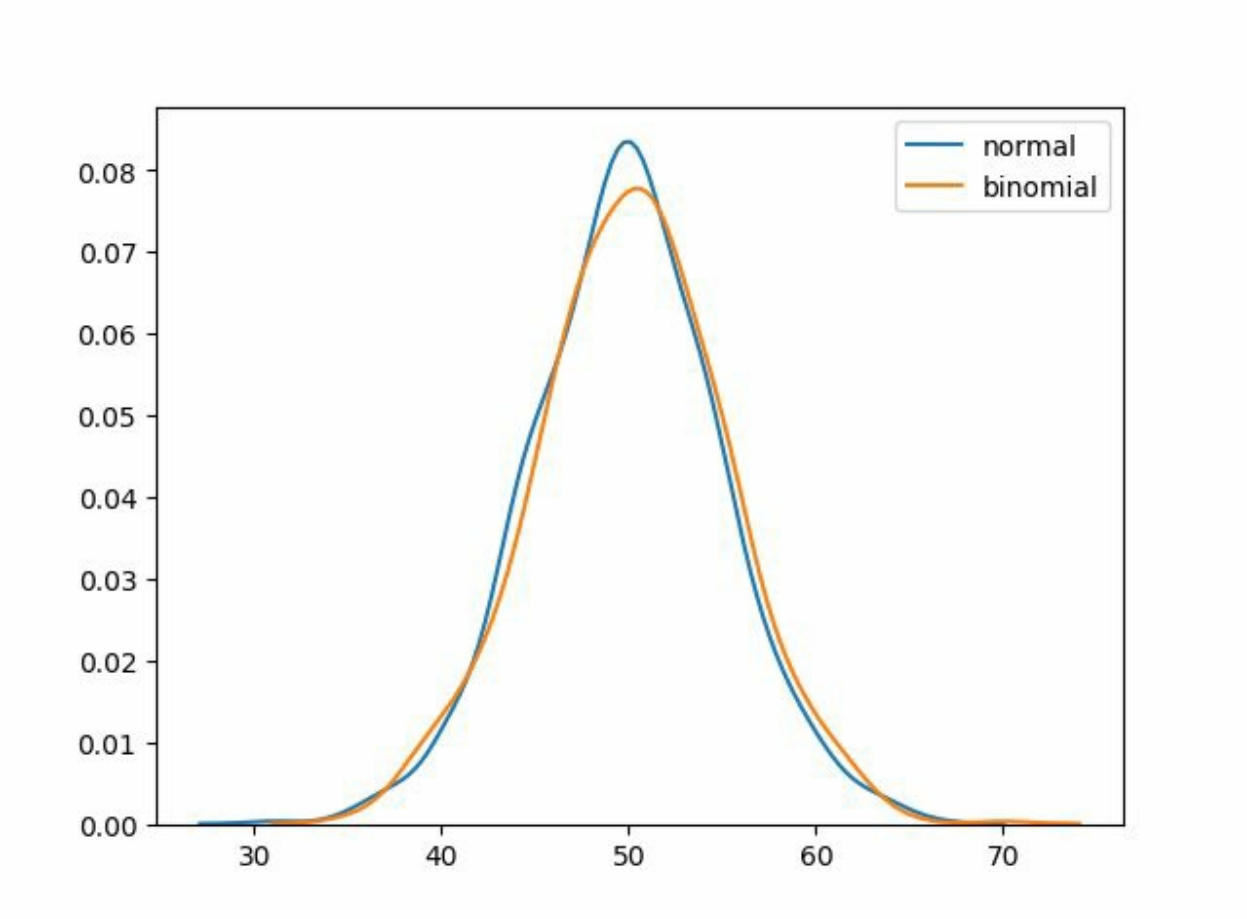
Difference Between Normal and Binomial Distribution

The main difference is that normal distribution is continous whereas binomial is discrete, but if there are enough data points it will be quite similar to normal distribution with certain loc and scale.

Example

```
from numpy import random

import matplotlib.pyplot as plt

import seaborn as sns

sns.distplot(random.normal(loc=50, scale=5, size=1000), hist=False,
label='normal')

sns.distplot(random.binomial(n=100, p=0.5, size=1000), hist=False,
label='binomial')

plt.show()
```

Result



Poisson Distribution

Poisson Distribution

Poisson Distribution is a *Discrete Distribution*.

It estimates how many times an event can happen in a specified time.

e.g. If someone eats twice a day what is probability he will eat thrice?

It has two parameters:

lam - rate or known number of occurences e.g. 2 for above problem.

size - The shape of the returned array.

Example

Generate a random 1x10 distribution for occurence 2:

```
from numpy import random

x = random.poisson(lam=2, size=10)

print(x)
```

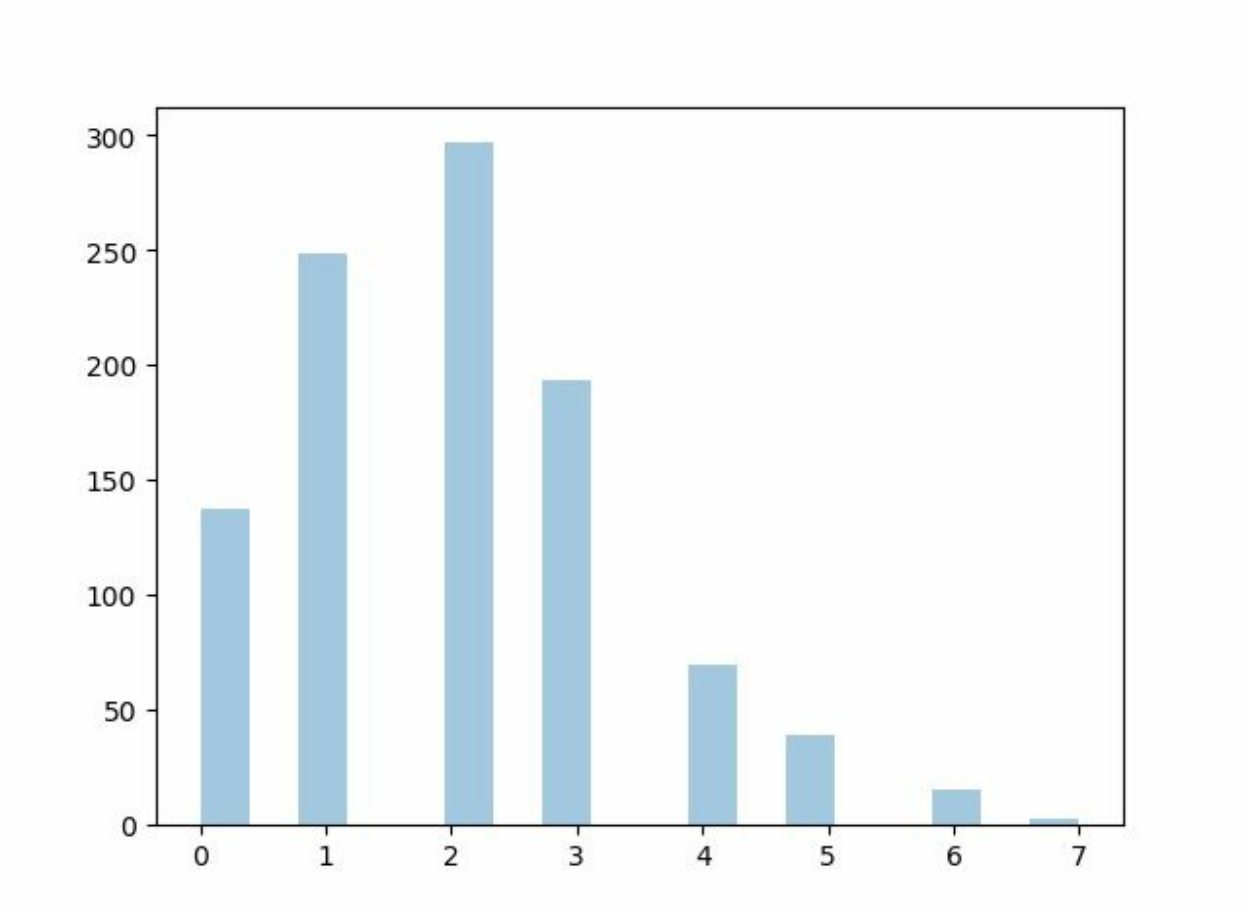
Visualization of Poisson Distribution

Example

```
from numpy import random

import matplotlib.pyplot as plt
```

```
import seaborn as sns


sns.distplot(random.poisson(lam=2, size=1000), kde=False)

plt.show()
```

Result



Difference Between Normal and Poisson Distribution

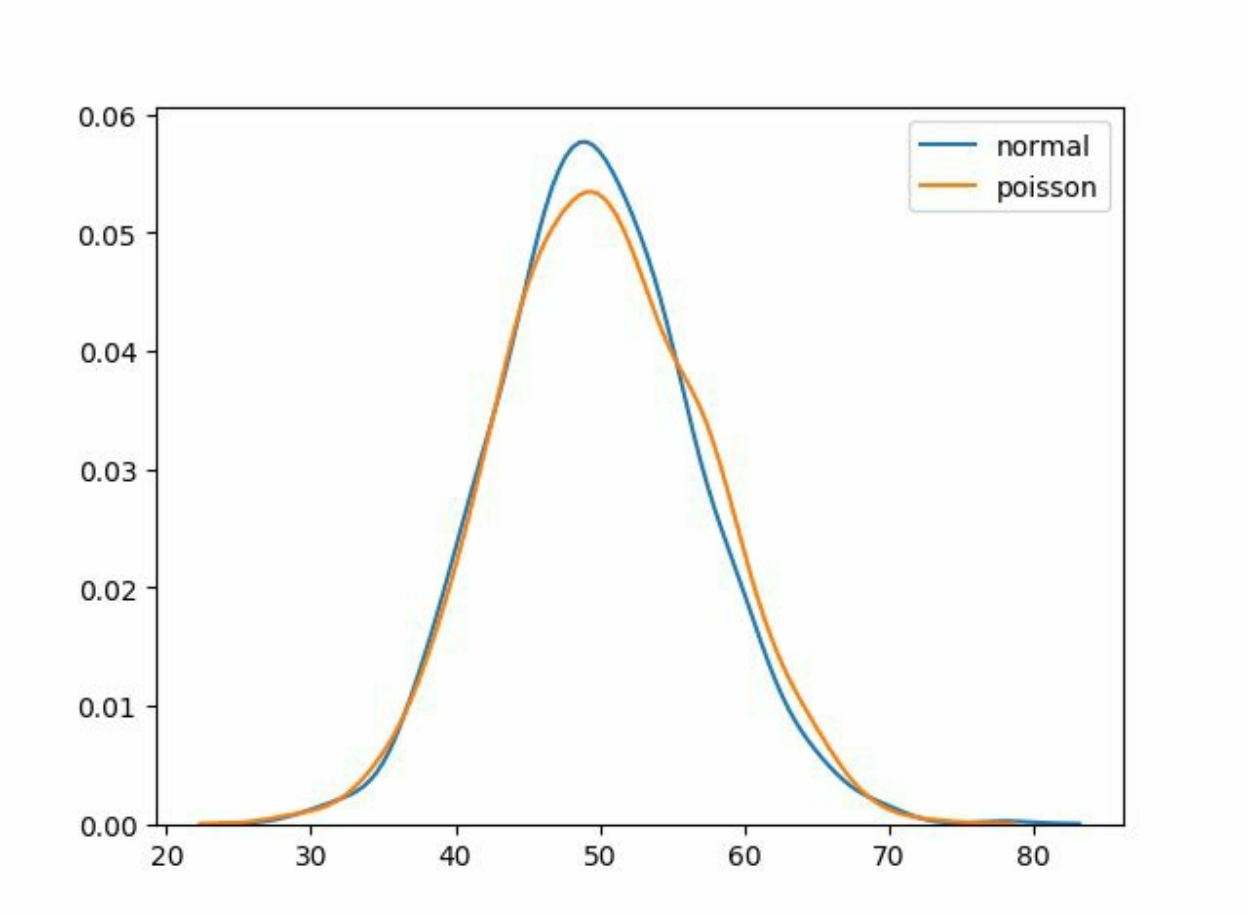
Normal distribution is continous whereas poisson is discrete.

But we can see that similar to binomial for a large enough poisson distribution it will become similar to normal distribution with certain std dev and mean.

Example

```
from numpy import random

import matplotlib.pyplot as plt

import seaborn as sns

sns.distplot(random.normal(loc=50, scale=7, size=1000), hist=False, label='normal')

sns.distplot(random.poisson(lam=50, size=1000), hist=False, label='poisson')

plt.show()
```

Result



Difference Between Poisson and Binomial Distribution

The difference is very subtle it is that, binomial distribution is for discrete trials, whereas poisson distribution is for continuous trials.

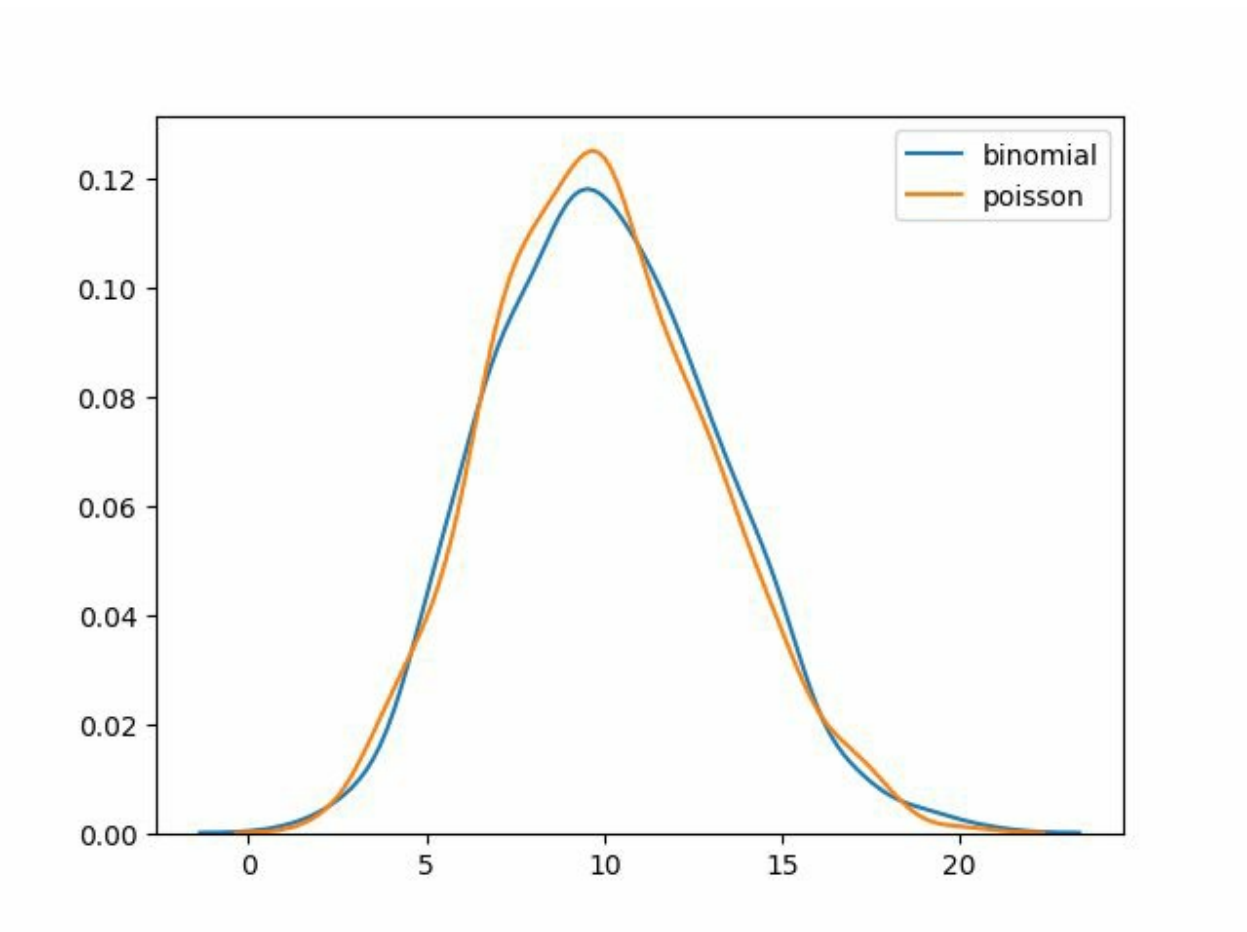
But for very large n and near-zero p binomial distribution is near identical to poisson distribution such that n * p is nearly equal to lam.

Example

```
from numpy import random
```

```
import matplotlib.pyplot as plt

import seaborn as sns

sns.distplot(random.binomial(n=1000, p=0.01, size=1000), hist=False,
label='binomial')

sns.distplot(random.poisson(lam=10, size=1000), hist=False, label='poisson')

plt.show()
```

Result



Uniform Distribution

Uniform Distribution

Used to describe probability where every event has equal chances of occuring.

E.g. Generation of random numbers.

It has three parameters:

a - lower bound - default 0 .0.

b - upper bound - default 1.0.

size - The shape of the returned array.

Example

Create a 2x3 uniform distribution sample:

```
from numpy import random

x = random.uniform(size=(2, 3))

print(x)
```

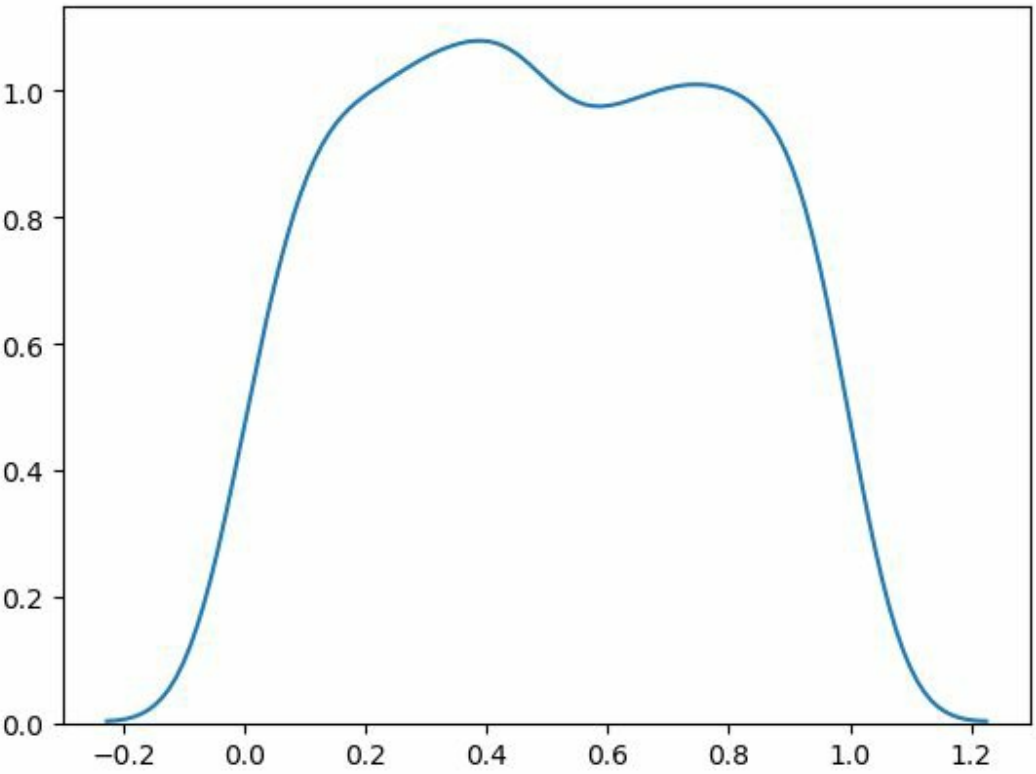
Visualization of Uniform Distribution

Example

```
from numpy import random
```

```
import matplotlib.pyplot as plt

import seaborn as sns

sns.distplot(random.uniform(size=1000), hist=False)

plt.show()
```

Result



Logistic Distribution

Logistic Distribution

Logistic Distribution is used to describe growth.

Used extensively in machine learning in logistic regression, neural networks etc.

It has three parameters:

loc - mean, where the peak is. Default 0.

scale - standard deviation, the flatness of distribution. Default 1.

size - The shape of the returned array.

Example

Draw 2x3 samples from a logistic distribution with mean at 1 and stddev 2.0:

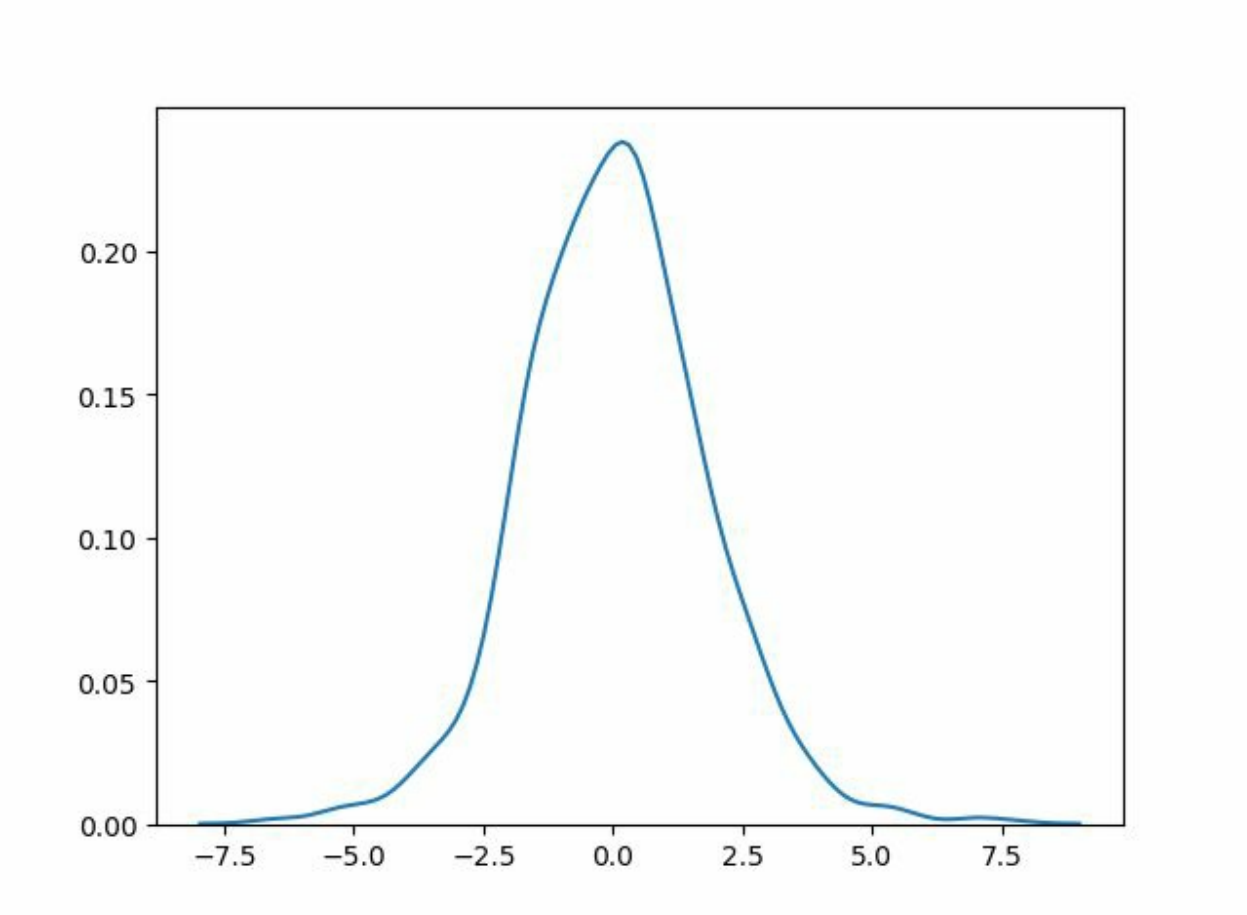
```
from numpy import random

x = random.logistic(loc=1, scale=2, size=(2, 3))

print(x)
```

Visualization of Logistic Distribution

Example

```
from numpy import random

import matplotlib.pyplot as plt

import seaborn as sns

sns.distplot(random.logistic(size=1000), hist=False)

plt.show()
```

Result



Difference Between Logistic and Normal Distribution

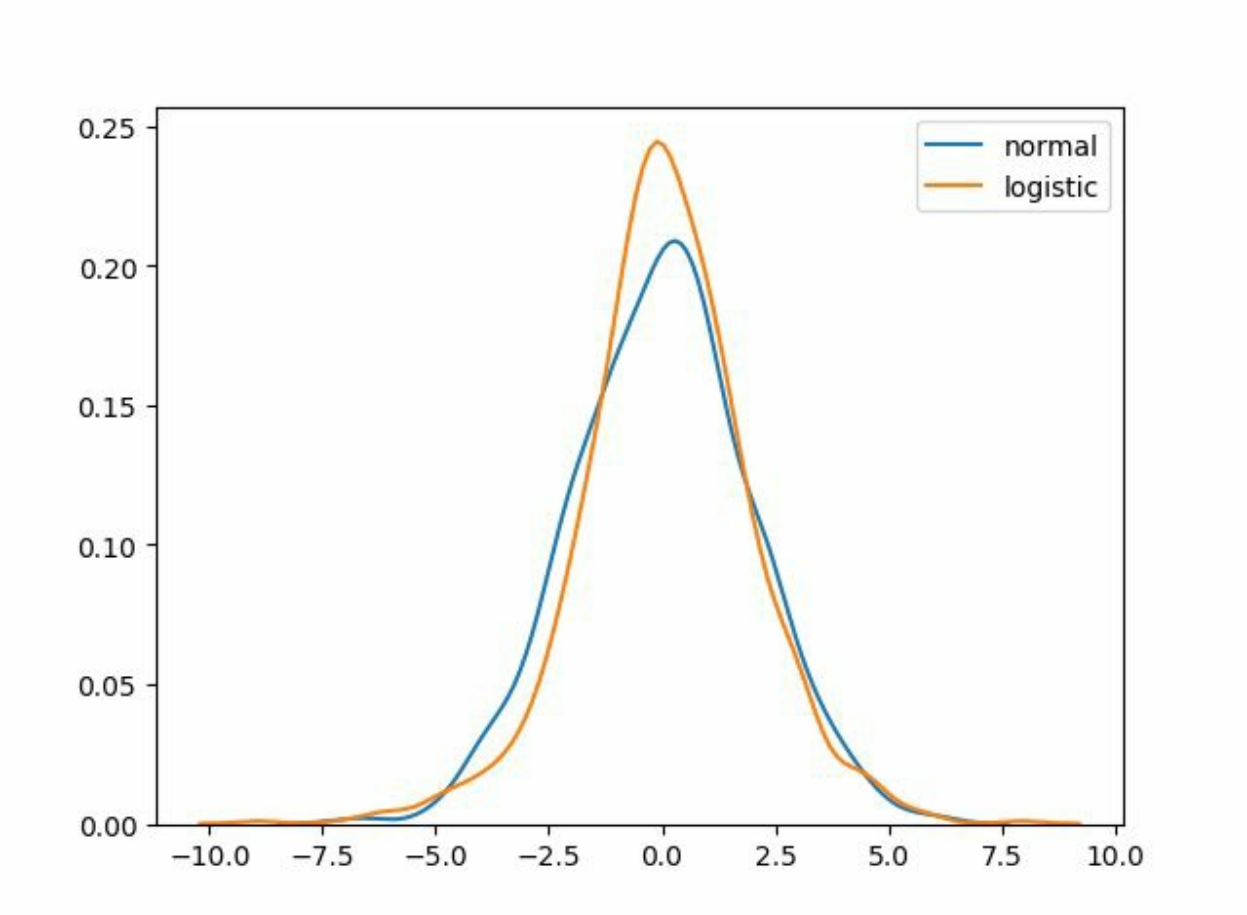
Both distributions are near identical, but logistic distribution has more area under the tails. ie. It representage more possibility of occurence of an events further away from mean.

For higher value of scale (standard deviation) the normal and logistic distributions are near identical apart from the peak.

Example

```
from numpy import random
import matplotlib.pyplot as plt
import seaborn as sns
sns.distplot(random.normal(scale=2, size=1000), hist=False, label='normal')
sns.distplot(random.logistic(size=1000), hist=False, label='logistic')
plt.show()
```

Result



Multinomial Distribution

Multinomial Distribution

Multinomial distribution is a generalization of binomial distribution.

It describes outcomes of multi-nomial scenarios unlike binomial where scenarios must be only one of two. e.g. Blood type of a population, dice roll outcome.

It has three parameters:

n - number of possible outcomes (e.g. 6 for dice roll).

pvals - list of probabilties of outcomes (e.g. [1/6, 1/6, 1/6, 1/6, 1/6, 1/6] for dice roll).

size - The shape of the returned array.

Example

Draw out a sample for dice roll:

```
from numpy import random

x = random.multinomial(n=6, pvals=[1/6, 1/6, 1/6, 1/6, 1/6, 1/6])

print(x)
```

Note: Multinomial samples will NOT produce a single value! They will produce one value for each pval.

Note: As they are generalization of binomial distribution their visual representation and similarity of normal distribution is same as that of multiple binomial distributions.

Exponential Distribution

Exponential distribution is used for describing time till next event e.g. failure/success etc.

It has two parameters:

scale - inverse of rate (see lam in poisson distribution) defaults to 1.0.

size - The shape of the returned array.

Example

Draw out a sample for exponential distribution with 2.0 scale with 2x3 size:

```
from numpy import random
```

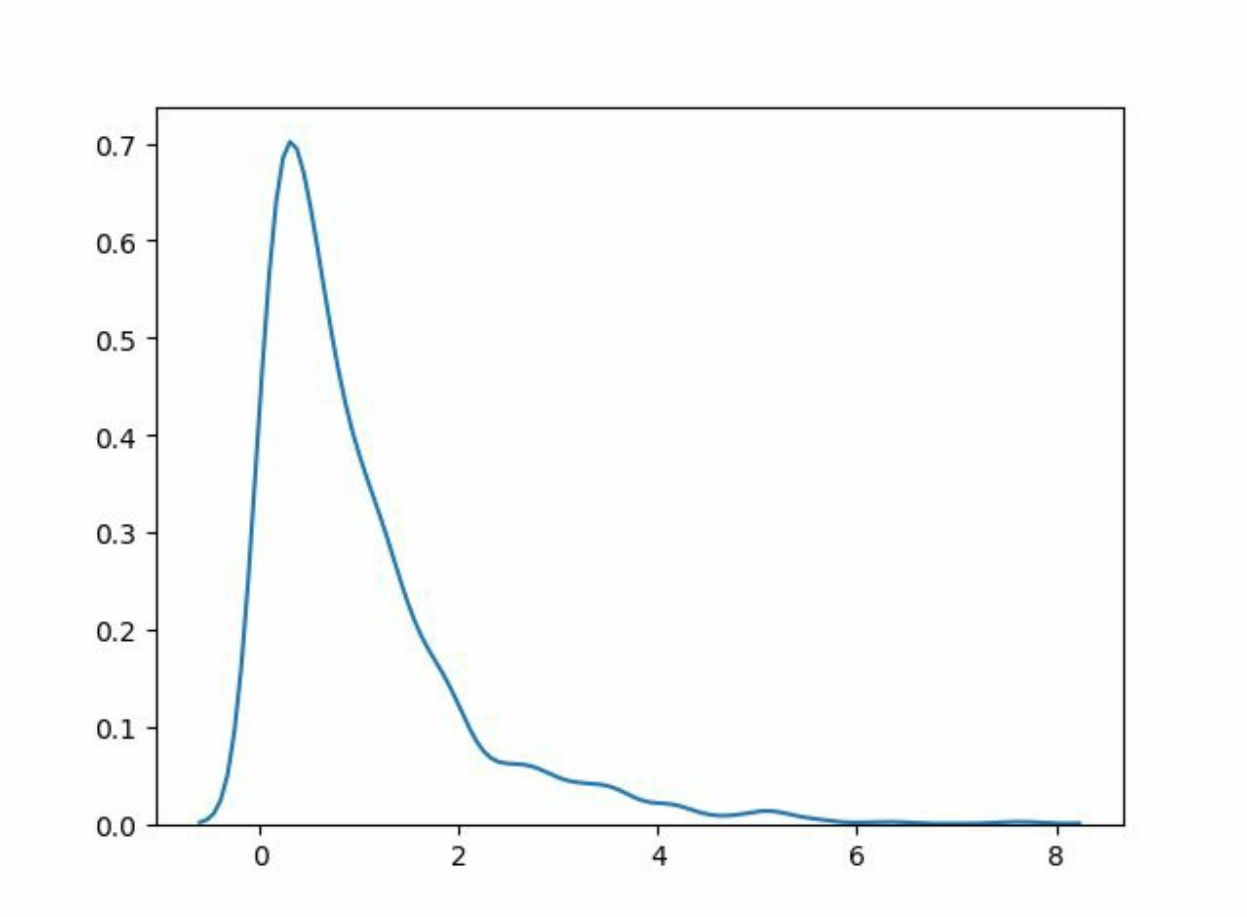
```
x = random.exponential(scale=2, size=(2, 3))

print(x)
```

Visualization of Exponential Distribution

Example

```
from numpy import random

import matplotlib.pyplot as plt

import seaborn as sns

sns.distplot(random.exponential(size=1000), hist=False)

plt.show()
```

Result



Relation Between Poisson and Exponential Distribution

Poisson distribution deals with number of occurences of an event in a time period whereas exponential distribution deals with the time between these events.

Chi Square Distribution

Chi Square Distribution

Chi Square distribution is used as a basis to verify the hypothesis.

It has two parameters:

df - (degree of freedom).

size - The shape of the returned array.

Example

Draw out a sample for chi squared distribution with degree of freedom 2 with size 2x3:

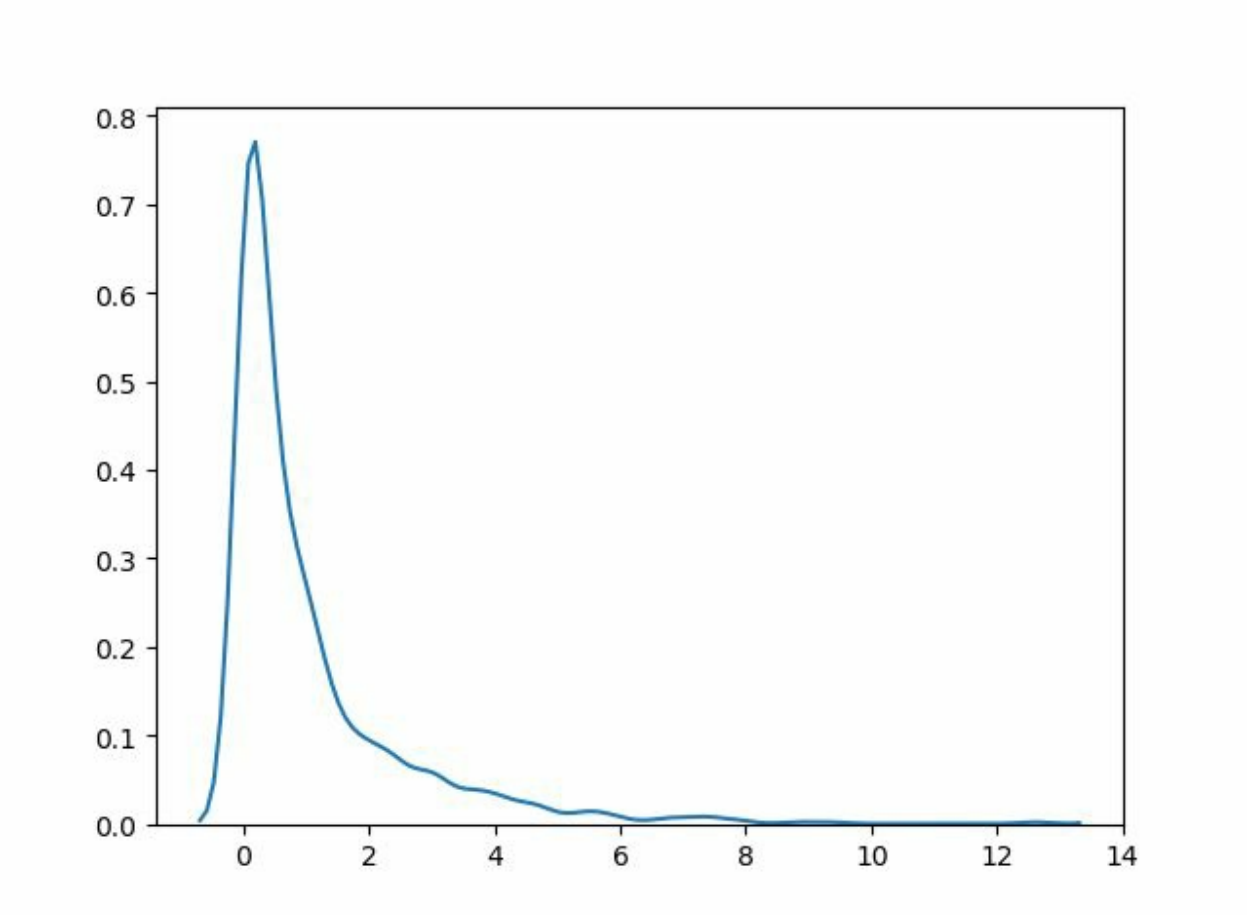
```
from numpy import random

x = random.chisquare(df=2, size=(2, 3))

print(x)
```

Visualization of Chi Square Distribution

Example

```
from numpy import random

import matplotlib.pyplot as plt

import seaborn as sns

sns.distplot(random.chisquare(df=1, size=1000), hist=False)

plt.show()
```

Result



Rayleigh Distribution

Rayleigh distribution is used in signal processing.

It has two parameters:

scale - (standard deviation) decides how flat the distribution will be default 1.0).

size - The shape of the returned array.

Example

Draw out a sample for rayleigh distribution with scale of 2 with size 2x3:

```
from numpy import random
```

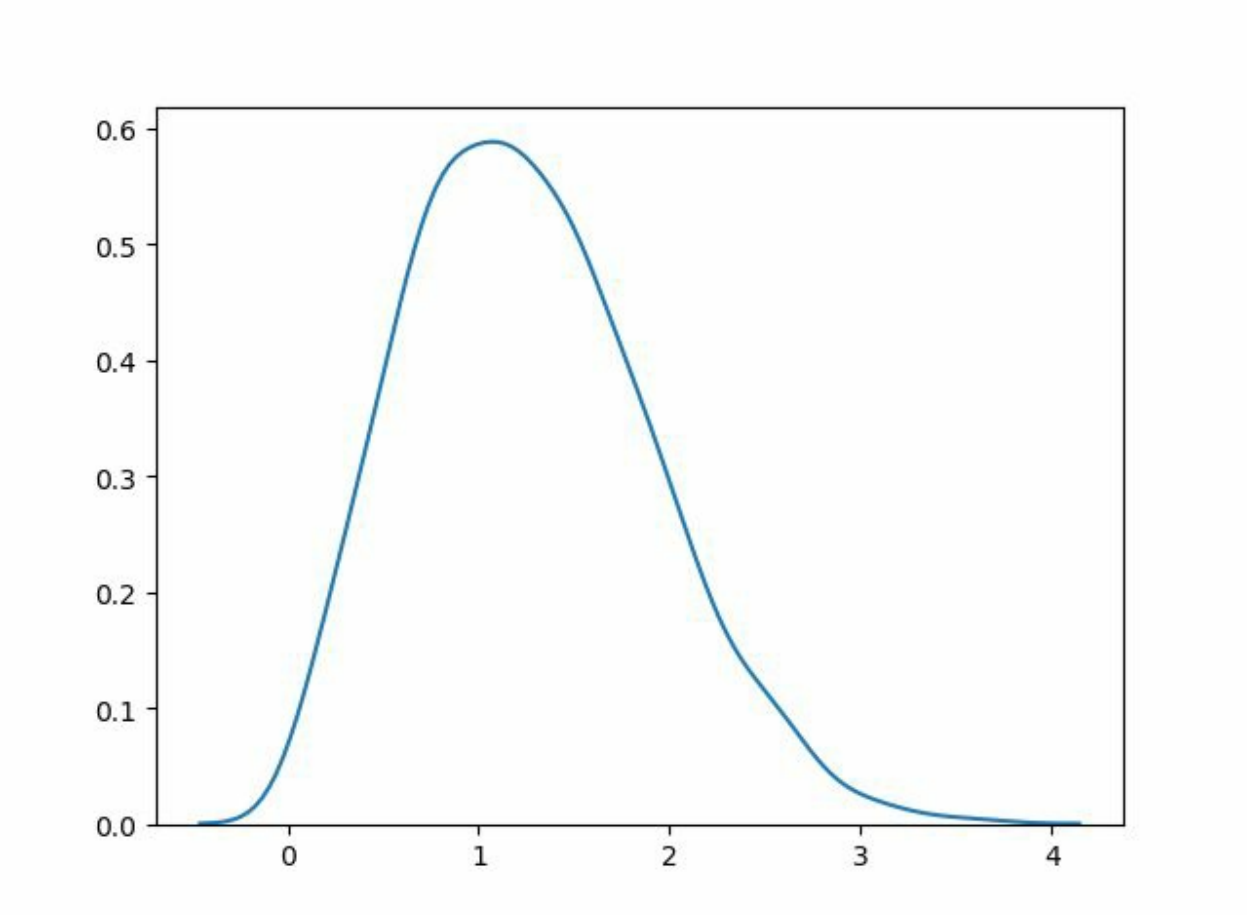
```
x = random.rayleigh(scale=2, size=(2, 3))

print(x)
```

Visualization of Rayleigh Distribution

Example

```
from numpy import random

import matplotlib.pyplot as plt

import seaborn as sns

sns.distplot(random.rayleigh(size=1000), hist=False)

plt.show()
```

Result



Similarity Between Rayleigh and Chi Square Distribution

At unit stddev the and 2 degrees of freedom rayleigh and chi square represent the same distributions.

Pareto Distribution

Pareto Distribution

A distribution following Pareto's law i.e. 80-20 distribution (20% factors cause 80% outcome).

It has two parameter:

a - shape parameter.

size - The shape of the returned array.

Example

Draw out a sample for pareto distribution with shape of 2 with size 2x3:

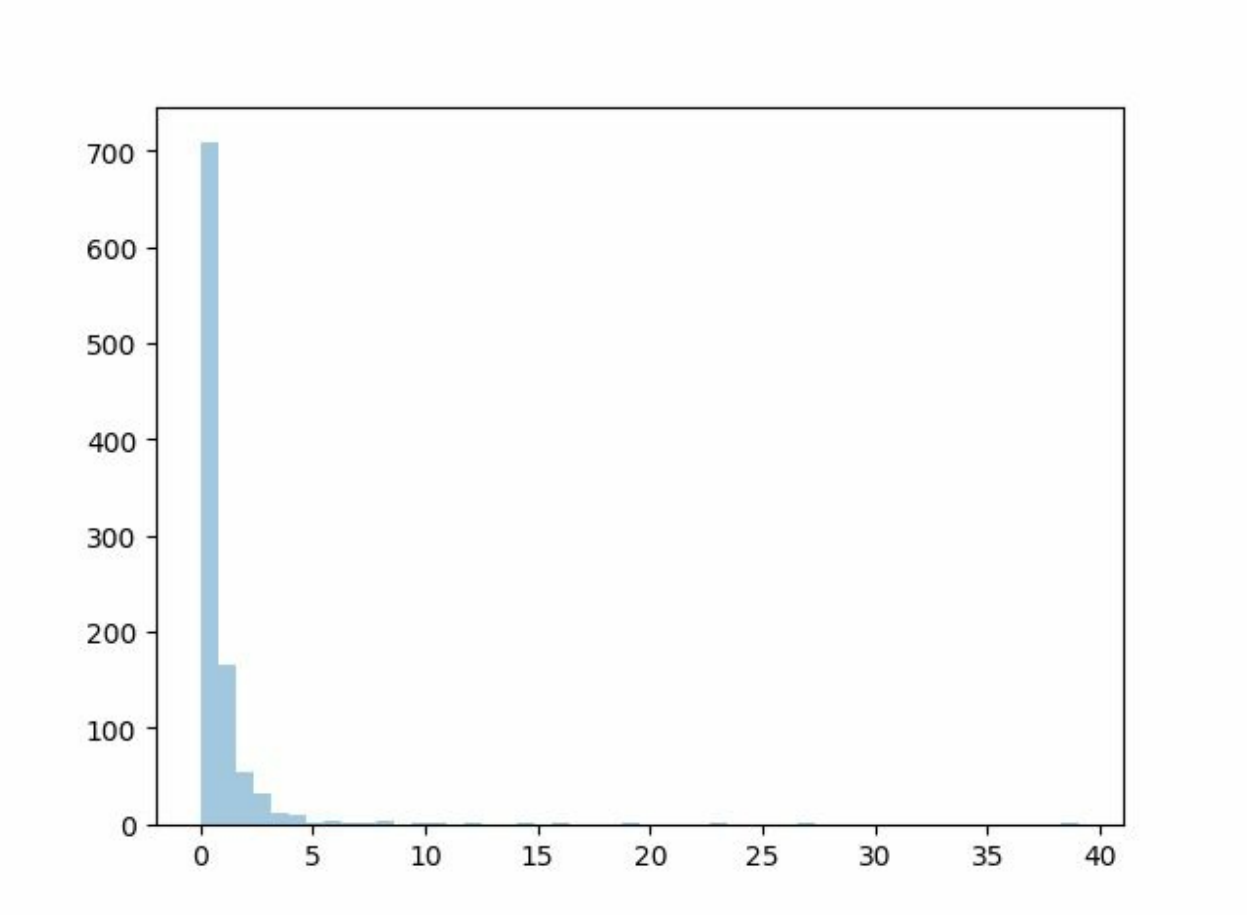
```
from numpy import random

x = random.pareto(a=2, size=(2, 3)

print(x)
```

Visualization of Pareto Distribution

Example

```
from numpy import random

import matplotlib.pyplot as plt

import seaborn as sns

sns.distplot(random.pareto(a=2, size=1000), kde=False)

plt.show()
```

Result



Zipf Distribution

Zipf distritutions are used to sample data based on zipf's law.

Zipf's Law: In a collection the nth common term is 1/n times of the most common term. E.g. 5th common word in english has occurs nearly 1/5th times as of the most used word.

It has two parameters:

a - distribution parameter.

size - The shape of the returned array.

Example

Draw out a sample for zipf distribution with distribution parameter 2 with size 2x3:

```
from numpy import random

x = random.zipf(a=2, size=(2, 3))

print(x)
```

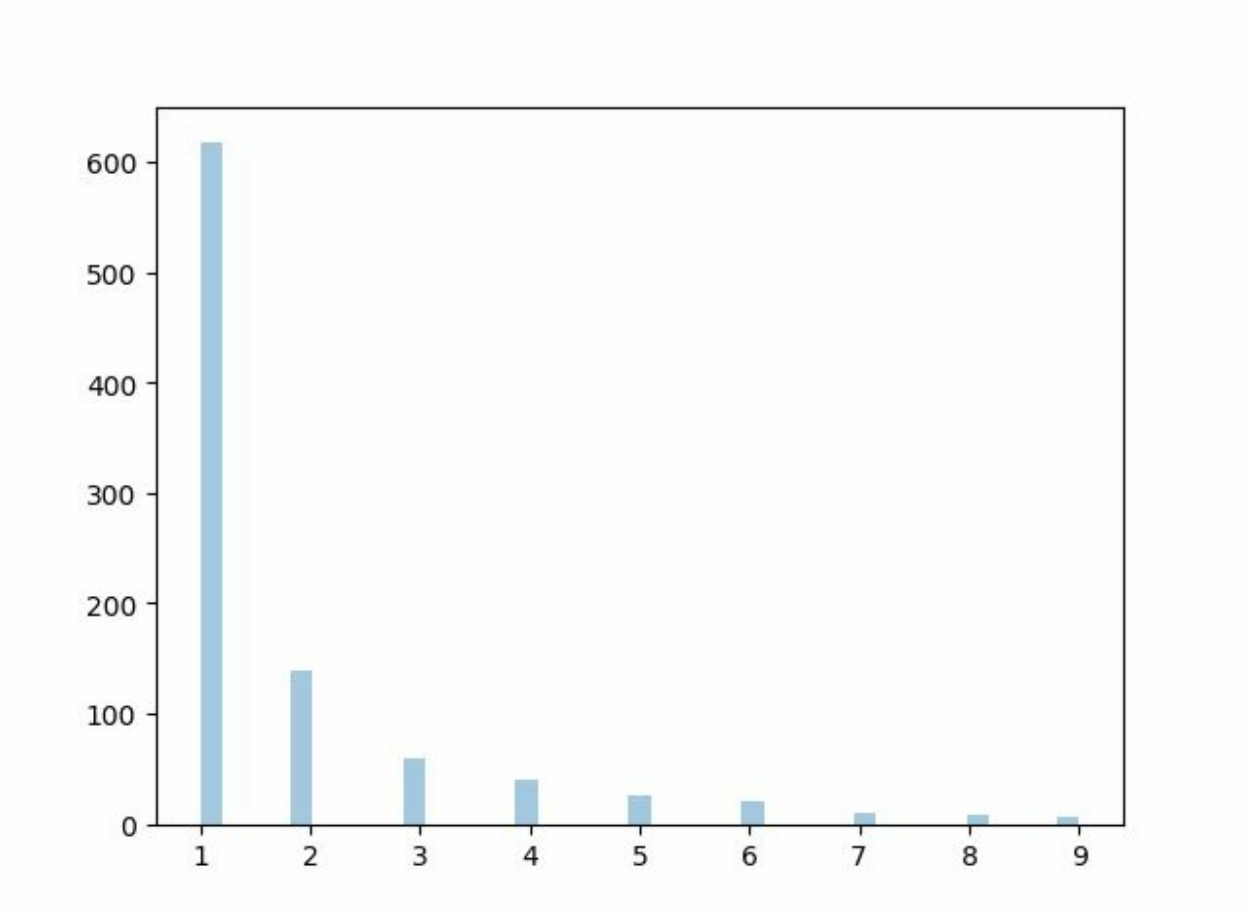
Visualization of Zipf Distribution

Sample 1000 points but plotting only ones with value < 10 for more meaningful chart.

Example

```
from numpy import random

import matplotlib.pyplot as plt

import seaborn as sns

x = random.zipf(a=2, size=1000)

sns.distplot(x[x<10], kde=False)

plt.show()
```

Result



NumPy ufuncs

What are ufuncs?

ufuncs stands for "Universal Functions" and they are NumPy functions that operates on the ndarray object.

Why use ufuncs?

ufuncs are used to implement *vectorization* in NumPy which is way faster than iterating over elements.

They also provide broadcasting and additional methods like reduce, accumulate etc. that are very helpful for computation.

ufuncs also take additional arguments, like:

where boolean array or condition defining where the operations should take place.

dtype defining the return type of elements.

out output array where the return value should be copied.

What is Vectorization?

Converting iterative statements into a vector based operation is called vectorization.

It is faster as modern CPUs are optimized for such operations.

Add the Elements of Two Lists

list 1: [1, 2, 3, 4]

list 2: [4, 5, 6, 7]

One way of doing it is to iterate over both of the lists and then sum each elements.

Example

Without ufunc, we can use Python's built-in zip() method:

```
x = [1, 2, 3, 4]

y = [4, 5, 6, 7]

z = []

for i, j in zip(x, y):
```

```
  z.append(i + j)

print(z)
```

NumPy has a ufunc for this, called add(x, y) that will produce the same result.

Example

With ufunc, we can use the add() function:

```
import numpy as np

x = [1, 2, 3, 4]

y = [4, 5, 6, 7]

z = np.add(x, y)

print(z)
```

Create Your Own ufunc

How To Create Your Own ufunc

To create you own ufunc, you have to define a function, like you do with normal functions in Python, then you add it to your NumPy ufunc library with the frompyfunc() method.

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.

Example

Create your own ufunc for addition:

```
import numpy as np

def myadd(x, y):
  return x+y

myadd = np.frompyfunc(myadd, 2, 1)

print(myadd([1, 2, 3, 4], [5, 6, 7, 8]))
```

Check if a Function is a ufunc

Check the *type* of a function to check if it is a ufunc or not.

A ufunc should return <class 'numpy.ufunc'>.

Example

Check if a function is a ufunc:

```
import numpy as np

print(type(np.add))
```

If it is not a ufunc, it will return another type, like this built-in NumPy function for joining two or more arrays:

Example

Check the type of another function: concatenate():

```
import numpy as np
```

```
print(type(np.concatenate))
```

If the function is not recognized at all, it will return an error:

Example

Check the type of something that does not exist. This will produce an error:

```
import numpy as np

print(type(np.blahblah))
```

To test if the function is a ufunc in an if statement, use the numpy.ufunc value (or np.ufunc if you use np as an alias for numpy):

Example

Use an if statement to check if the function is a ufunc or not:

```
import numpy as np

if type(np.add) == np.ufunc:

  print('add is ufunc')

else:

  print('add is not ufunc')
```

Simple Arithmetic

You could use arithmetic operators + - * / directly between NumPy arrays, but this section discusses an extension of the same where we have functions

that can take any array-like objects e.g. lists, tuples etc. and perform arithmetic *conditionally*.

Arithmetic Conditionally: means that we can define conditions where the arithmetic operation should happen.

All of the discussed arithmetic functions take a where parameter in which we can specify that condition.

Addition

The add() function sums the content of two arrays, and return the results in a new array.

Example

Add the values in arr1 to the values in arr2:

```
import numpy as np

arr1 = np.array([10, 11, 12, 13, 14, 15])

arr2 = np.array([20, 21, 22, 23, 24, 25])

newarr = np.add(arr1, arr2)

print(newarr)
```

The example above will return [30 32 34 36 38 40] which is the sums of 10+20, 11+21, 12+22 etc.

Subtraction

The subtract() function subtracts the values from one array with the values from another array, and return the results in a new array.

Example

Subtract the values in arr2 from the values in arr1:

```
import numpy as np

arr1 = np.array([10, 20, 30, 40, 50, 60])

arr2 = np.array([20, 21, 22, 23, 24, 25])

newarr = np.subtract(arr1, arr2)

print(newarr)
```

The example above will return [-10 -1 8 17 26 35] which is the result of 10-20, 20-21, 30-22 etc.

Multiplication

The multiply() function multiplies the values from one array with the values from another array, and return the results in a new array.

Example

Multiply the values in arr1 with the values in arr2:

```
import numpy as np

arr1 = np.array([10, 20, 30, 40, 50, 60])

arr2 = np.array([20, 21, 22, 23, 24, 25])

newarr = np.multiply(arr1, arr2)

print(newarr)
```

The example above will return [200 420 660 920 1200 1500] which is the

result of 10*20, 20*21, 30*22 etc.

Division

The divide() function divides the values from one array with the values from another array, and return the results in a new array.

Example

Divide the values in arr1 with the values in arr2:

```
import numpy as np

arr1 = np.array([10, 20, 30, 40, 50, 60])

arr2 = np.array([3, 5, 10, 8, 2, 33])

newarr = np.divide(arr1, arr2)

print(newarr)
```

The example above will return [3.33333333 4. 3. 5. 25. 1.81818182] which is the result of 10/3, 20/5, 30/10 etc.

Power

The power() function rises the values from the first array to the power of the values of the second array, and return the results in a new array.

Example

Raise the valules in arr1 to the power of values in arr2:

```
import numpy as np

arr1 = np.array([10, 20, 30, 40, 50, 60])

arr2 = np.array([3, 5, 6, 8, 2, 33])

newarr = np.power(arr1, arr2)

print(newarr)
```

The example above will return [1000 3200000 729000000 6553600000000 2500 0] which is the result of 10*10*10, 20*20*20*20*20, 30*30*30*30*30*30 etc.

Remainder

Both the mod() and the remainder() functions return the remainder of the values in the first array corresponding to the values in the second array, and return the results in a new array.

Example

Return the remainders:

```
import numpy as np

arr1 = np.array([10, 20, 30, 40, 50, 60])

arr2 = np.array([3, 7, 9, 8, 2, 33])

newarr = np.mod(arr1, arr2)

print(newarr)
```

The example above will return [1 6 3 0 0 27] which is the remainders when you divide 10 with 3 (10%3), 20 with 7 (20%7) 30 with 9 (30%9) etc.

You get the same result when using the remainder() function:

Example

Return the remainders:

```
import numpy as np

arr1 = np.array([10, 20, 30, 40, 50, 60])

arr2 = np.array([3, 7, 9, 8, 2, 33])

newarr = np.remainder(arr1, arr2)

print(newarr)
```

Quotient and Mod

The divmod() function return both the quotient and the the mod. The return value is two arrays, the first array contains the quotient and second array contains the mod.

Example

Return the quotient and mod:

```
import numpy as np

arr1 = np.array([10, 20, 30, 40, 50, 60])

arr2 = np.array([3, 7, 9, 8, 2, 33])

newarr = np.divmod(arr1, arr2)

print(newarr)
```

The example above will return:

(array([3, 2, 3, 5, 25, 1]), array([1, 6, 3, 0, 0, 27]))

The first array represents the quotients, (the integer value when you divide 10 with 3, 20 with 7, 30 with 9 etc.

The second array represents the remainders of the same divisions.

Absolute Values

Both the absolute() and the abs() functions functions do the same absolute operation element-wise but we should use absolute() to avoid confusion with python's inbuilt math.abs()

Example

Return the quotient and mod:

```
import numpy as np

arr = np.array([-1, -2, 1, 2, 3, -4])

newarr = np.absolute(arr)

print(newarr)
```

The example above will return [1 2 1 2 3 4].

Rounding Decimals

There are primarily five ways of rounding off decimals in NumPy:

- truncation
- fix

- rounding
- floor
- ceil

Truncation

Remove the decimals, and return the float number closest to zero. Use the trunc() and fix() functions.

Example

Truncate elements of following array:

```
import numpy as np

arr = np.trunc([-3.1666, 3.6667])

print(arr)
```

Example

Same example, using fix():

```
import numpy as np

arr = np.fix([-3.1666, 3.6667])

print(arr)
```

Rounding

The around() function increments preceding digit or decimal by 1 if >=5 else do nothing.

E.g. round off to 1 decimal point, 3.16666 is 3.2

Example

Round off 3.1666 to 2 decimal places:

```
import numpy as np

arr = np.around(3.1666, 2)

print(arr)
```

Floor

The floor() function rounds off decimal to nearest lower integer.

E.g. floor of 3.166 is 3.

Example

Floor the elements of following array:

```
import numpy as np

arr = np.floor([-3.1666, 3.6667])

print(arr)
```

Note: The floor() function returns floats, unlike the trunc() function who returns integers.

Ceil

The ceil() function rounds off decimal to nearest upper integer.

E.g. ceil of 3.166 is 4.

Example

Ceil the elements of following array:

```
import numpy as np

arr = np.ceil([-3.1666, 3.6667])

print(arr)
```

Logs

NumPy provides functions to perform log at the base 2, e and 10.

We will also explore how we can take log for any base by creating a custom ufunc.

All of the log functions will place -inf or inf in the elements if the log can not be computed.

Log at Base 2

Use the log2() function to perform log at the base 2.

Example

Find log at base 2 of all elements of following array:

```
import numpy as np

arr = np.arange(1, 10)

print(np.log2(arr))
```

Note: The arange(1, 10) function returns an array with integers starting from

1 (included) to 10 (not included).

Log at Base 10

Use the log10() function to perform log at the base 10.

Example

Find log at base 10 of all elements of following array:

```
import numpy as np

arr = np.arange(1, 10)

print(np.log10(arr))
```

Natural Log, or Log at Base e

Use the log() function to perform log at the base e.

Example

Find log at base e of all elements of following array:

```
import numpy as np

arr = np.arange(1, 10)

print(np.log(arr))
```

Log at Any Base

NumPy does not provide any function to take log at any base, so we can use the frompyfunc() function along with inbuilt function math.log() with two input parameters and one output parameter:

Example

```
from math import log

import numpy as np

nplog = np.frompyfunc(log, 2, 1)

print(nplog(100, 15))
```

NumPy Summations

Summations

What is the difference between summation and addition?

Addition is done between two arguments whereas summation happens over n elements.

Example

Add the values in arr1 to the values in arr2:

```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([1, 2, 3])

newarr = np.add(arr1, arr2)
```

```
print(newarr)
```

Returns: [2 4 6]

Example

Sum the values in arr1 and the values in arr2:

```
import numpy as np


arr1 = np.array([1, 2, 3])

arr2 = np.array([1, 2, 3])


newarr = np.sum([arr1, arr2])


print(newarr)
```

Returns: 12

Summation Over an Axis

If you specify axis=1, NumPy will sum the numbers in each array.

Example

Perform summation in the following array over 1st axis:

```
import numpy as np
```

```
arr1 = np.array([1, 2, 3])

arr2 = np.array([1, 2, 3])

newarr = np.sum([arr1, arr2], axis=1

print(newarr)
```

Returns: [6 6]

Cummulative Sum

Cummulative sum means partially adding the elements in array.

E.g. The partial sum of [1, 2, 3, 4] would be [1, 1+2, 1+2+3, 1+2+3+4] = [1, 3, 6, 10].

Perfom partial sum with the cumsum() function.

Example

Perform cummulative summation in the following array:

```
import numpy as np

arr = np.array([1, 2, 3])

newarr = np.cumsum(arr)

print(newarr)
```

Returns: [1 3 6]

NumPy Products

Products

To find the product of the elements in an array, use the prod() function.

Example

Find the product of the elements of this array:

```
import numpy as np

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

x = np.prod(arr)

print(x)
```

Returns: 24 because 1*2*3*4 = 24

Example

Find the product of the elements of two arrays:

```
import numpy as np

arr1 = np.array([1, 2, 3, 4])

arr2 = np.array([5, 6, 7, 8])

x = np.prod([arr1, arr2])


print(x)
```

Returns: 40320 because 1*2*3*4*5*6*7*8 = 40320

Product Over an Axis

If you specify axis=1, NumPy will return the product of each array.

Example

Perform summation in the following array over 1st axis:

```
import numpy as np

arr1 = np.array([1, 2, 3, 4])

arr2 = np.array([5, 6, 7, 8])

newarr = np.prod([arr1, arr2], axis=1)

print(newarr)
```

Returns: [24 1680]

Cummulative Product

Cummulative product means taking the product partially.

E.g. The partial product of [1, 2, 3, 4] is [1, 1*2, 1*2*3, 1*2*3*4] = [1, 2, 6, 24]

Perfom partial sum with the cumprod() function.

Example

Take cummulative product of all elements for following array:

```
import numpy as np

arr = np.array([5, 6, 7, 8])

newarr = np.cumprod(arr)

print(newarr)
```

Returns: [5 30 210 1680]

NumPy Differences

Differences

A discrete difference means subtracting two successive elements.

E.g. for [1, 2, 3, 4], the discrete difference would be [2-1, 3-2, 4-3] = [1, 1, 1]

To find the discrete difference, use the diff() function.

Example

Compute discrete difference of the following array:

```
import numpy as np

arr = np.array([10, 15, 25, 5])

newarr = np.diff(arr)

print(newarr)
```

Returns: [5 10 -20] because 15-10=5, 25-15=10, and 5-25=-20

We can perform this operation repeatedly by giving parameter n.

E.g. for [1, 2, 3, 4], the discrete difference with n = 2 would be [2-1, 3-2, 4-3] = [1, 1, 1] , then, since n=2, we will do it once more, with the new result: [1-1, 1-1] = [0, 0]

Example

Compute discrete difference of the following array twice:

```
import numpy as np

arr = np.array([10, 15, 25, 5])

newarr = np.diff(arr, n=2)

print(newarr)
```

Returns: [5 -30] because: 15-10=5, 25-15=10, and 5-25=-20 AND 10-5=5 and -20-10=-30

NumPy LCM Lowest Common Multiple

Finding LCM (Lowest Common Multiple)

The Lowest Common Multiple is the least number that is common multiple of both of the numbers.

Example

Find the LCM of the following two numbers:

```
import numpy as np
```

```
num1 = 4

num2 = 6

x = np.lcm(num1, num2)

print(x)
```

Returns: 12 because that is the lowest common multiple of both numbers (4*3=12 and 6*2=12).

Finding LCM in Arrays

To find the Lowest Common Multiple of all values in an array, you can use the reduce() method.

The reduce() method will use the ufunc, in this case the lcm() function, on each element, and reduce the array by one dimension.

Example

Find the LCM of the values of the following array:

```
import numpy as np

arr = np.array([3, 6, 9])

x = np.lcm.reduce(arr)

print(x)
```

Returns: 18 because that is the lowest common multiple of all three numbers (3*6=18, 6*3=18 and 9*2=18).

Example

Find the LCM of all of an array where the array contains all integers from 1 to 10:

```
import numpy as np

arr = np.arange(1, 11)

x = np.lcm.reduce(arr)

print(x)
```

NumPy GCD Greatest Common Denominator

Finding GCD (Greatest Common Denominator)

The GCD (Greatest Common Denominator), also known as HCF (Highest Common Factor) is the biggest number that is a common factor of both of the numbers.

Example

Find the HCF of the following two numbers:

```
import numpy as np

num1 = 6

num2 = 9

x = np.gcd(num1, num2)
```

```
print(x)
```

Returns: 3 because that is the highest number both numbers can be divided by (6/3=2 and 9/3=3).

Finding GCD in Arrays

To find the Highest Common Factor of all values in an array, you can use the reduce() method.

The reduce() method will use the ufunc, in this case the gcd() function, on each element, and reduce the array by one dimension.

Example

Find the GCD for all of the numbers in following array:

```
import numpy as np

arr = np.array([20, 8, 32, 36, 16])

x = np.gcd.reduce(arr)

print(x)
```

Returns: 4 because that is the highest number all values can be divided by.

NumPy Trigonometric Functions

Trigonometric Functions

NumPy provides the ufuncs sin(), cos() and tan() that take values in radians and produce the corresponding sin, cos and tan values.

Example

Find sine value of PI/2:

```
import numpy as np


x = np.sin(np.pi/2)


print(x)
```

Example

Find sine values for all of the values in arr:

```
import numpy as np

arr = np.array([np.pi/2, np.pi/3, np.pi/4, np.pi/5])

x = np.sin(arr)


print(x)
```

Convert Degrees Into Radians

By default all of the trigonometric functions take radians as parameters but we can convert radians to degrees and vice versa as well in NumPy.

Note: radians values are pi/180 * degree_values.

Example

Convert all of the values in following array arr to radians:

```
import numpy as np

arr = np.array([90, 180, 270, 360])

x = np.deg2rad(arr)

print(x)
```

Radians to Degrees

Example

Convert all of the values in following array arr to degrees:

```
import numpy as np

arr = np.array([np.pi/2, np.pi, 1.5*np.pi, 2*np.pi])

x = np.rad2deg(arr)

print(x)
```

Finding Angles

Finding angles from values of sine, cos, tan. E.g. sin, cos and tan inverse (arcsin, arccos, arctan).

NumPy provides ufuncs arcsin(), arccos() and arctan() that produce radian values for corresponding sin, cos and tan values given.

Example

Find the angle of 1.0:

```
import numpy as np

x = np.arcsin(1.0)

print(x)
```

Angles of Each Value in Arrays

Example

Find the angle for all of the sine values in the array

```
import numpy as np

arr = np.array([1, -1, 0.1])

x = np.arcsin(arr)

print(x)
```

Hypotenues

Finding hypotenues using pythagoras theorem in NumPy.

NumPy provides the hypot() function that takes the base and perpendicular values and produces hypotenues based on pythagoras theorem.

Example

Find the hypotenues for 4 base and 3 perpendicular:

```
import numpy as np
```

```
base = 3

perp = 4

x = np.hypot(base, perp)

print(x)
```

NumPy Hyperbolic Functions

Hyperbolic Functions

NumPy provides the ufuncs sinh(), cosh() and tanh() that take values in radians and produce the corresponding sinh, cosh and tanh values..

Example

Find sinh value of PI/2:

```
import numpy as np

x = np.sinh(np.pi/2)

print(x)
```

Example

Find cosh values for all of the values in arr:

```
import numpy as np

arr = np.array([np.pi/2, np.pi/3, np.pi/4, np.pi/5])

x = np.cosh(arr)


print(x)
```

Finding Angles

Finding angles from values of hyperbolic sine, cos, tan. E.g. sinh, cosh and tanh inverse (arcsinh, arccosh, arctanh).

Numpy provides ufuncs arcsinh(), arccosh() and arctanh() that produce radian values for corresponding sinh, cosh and tanh values given.

Example

Find the angle of 1.0:

```
import numpy as np

x = np.arcsinh(1.0)

print(x)
```

Angles of Each Value in Arrays

Example

Find the angle for all of the tanh values in array:

```
import numpy as np

arr = np.array([0.1, 0.2, 0.5])

x = np.arctanh(arr)

print(x)
```

NumPy Set Operations

What is a Set

A set in mathematics is a collection of unique elements.

Sets are used for operations involving frequent intersection, union and difference operations.

Create Sets in NumPy

We can use NumPy's unique() method to find unique elements from any array. E.g. create a set array, but remember that the set arrays should only be 1-D arrays.

Example

Convert following array with repeated elements to a set:

```
import numpy as np

arr = np.array([1, 1, 1, 2, 3, 4, 5, 5, 6, 7])

x = np.unique(arr)


print(x)
```

Finding Union

To find the unique values of two arrays, use the union1d() method.

Example

Find union of the following two set arrays:

```
import numpy as np
```

```
arr1 = np.array([1, 2, 3, 4])

arr2 = np.array([3, 4, 5, 6])

newarr = np.union1d(arr1, arr2)

print(newarr)
```

Finding Intersection

To find only the values that are present in both arrays, use the intersect1d() method.

Example

Find intersection of the following two set arrays:

```
import numpy as np

arr1 = np.array([1, 2, 3, 4])

arr2 = np.array([3, 4, 5, 6])

newarr = np.intersect1d(arr1, arr2, assume_unique=True)

print(newarr)
```

Note: the intersect1d() method takes an optional argument assume_unique, which if set to True can speed up computation. It should always be set to True when dealing with sets.

Finding Difference

To find only the values in the first set that is NOT present in the seconds set, use the setdiff1d() method.

Example

Find the difference of the set1 from set2:

```
import numpy as np

set1 = np.array([1, 2, 3, 4])
```

```
set2 = np.array([3, 4, 5, 6])

newarr = np.setdiff1d(set1, set2, assume_unique=True)


print(newarr)
```

Note: the setdiff1d() method takes an optional argument assume_unique, which if set to True can speed up computation. It should always be set to True when dealing with sets.

Finding Symmetric Difference

To find only the values that are NOT present in BOTH sets, use the setxor1d() method.

Example

Find the symmetric difference of the set1 and set2:

```
import numpy as np

set1 = np.array([1, 2, 3, 4])

set2 = np.array([3, 4, 5, 6])

newarr = np.setxor1d(set1, set2, assume_unique=True)

print(newarr)
```

Note: the setxor1d() method takes an optional argument assume_unique, which if set to True can speed up computation. It should always be set to True when dealing with sets.

Advanced NumPy

NumPy is at the base of Python's scientific stack of tools. Its purpose to implement efficient operations on many items in a block of memory. Understanding how it works in detail helps in making efficient use of its flexibility, taking useful shortcuts.

This section covers:

- **Anatomy of NumPy arrays, and its consequences. Tips and tricks.**
- **Universal functions: what, why, and what to do if you want a new one.**
- **Integration with other tools: NumPy offers several ways to wrap any data in an ndarray, without unnecessary copies.**
- **Recently added features, and what's in them: PEP 3118 buffers, generalized ufuncs,** …

Prerequisites

- NumPy
- Cython
- Pillow (Python imaging library, used in a couple of examples)

In this section, numpy will be imported as follows:

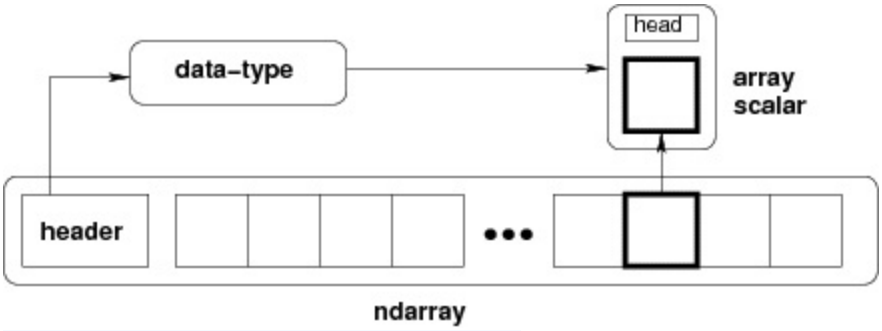
```
>>>
>>> import numpy as np

```

Life of ndarray

ndarray =

block of memory + indexing scheme + data type descriptor

- raw data
- how to locate an element
- how to interpret an element




```
typedef struct PyArrayObject {
        PyObject_HEAD

        /* Block of memory */
        char *data;

        /* Data type descriptor */
        PyArray_Descr *descr;

        /* Indexing scheme */
        int nd;
        npy_intp *dimensions;
        npy_intp *strides;

        /* Other stuff */
        PyObject *base;
        int flags;
        PyObject *weakreflist;
} PyArrayObject;
```

Block of memory

```
>>>
>>> x = np.array([1, 2, 3], dtype=np.int32)
>>> x.data
<... at ...>
>>> bytes(x.data)
'\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
```

Memory address of the data:

```
>>>
>>> x.__array_interface__['data'][0]
64803824
```

The whole `__array_interface__`:

```
>>>
>>> x.__array_interface__
{'data': (35828928, False),
'descr': [('', '<i4')],
'shape': (4,),
'strides': None,
'typestr': '<i4',
'version': 3}
```

Reminder: two **ndarrays** may share the same memory:

```
>>>
>>> x = np.array([1, 2, 3, 4])
>>> y = x[:-1]
>>> x[0] = 9
>>> y
array([9, 2, 3])
```

Memory does not need to be owned by an **ndarray**:

```
>>>
```

```
>>> x = b'1234'      # The 'b' is for "bytes", necessary in Python 3
```

x is a string (in Python 3 a bytes), we can represent its data as an array of ints:

```
>>>
>>> y = np.frombuffer(x, dtype=np.int8)
>>> y.data
<... at ...>
>>> y.base is x
True

>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : False
  WRITEABLE : False
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

The owndata and writeable flags indicate status of the memory block.

Data types

The descriptor

dtype describes a single item in the array:

type	**scalar type** of the data, one of: int8, int16, float64, *et al.* (fixed size) str, unicode, void (flexible size)

itemsize	**size** of the data block
byteorder	**byte order**: big-endian > / little-endian < / not applicable |
fields	sub-dtypes, if it's a **structured data type**
shape	shape of the array, if it's a **sub-array**

```
>>>
>>> np.dtype(int).type
<type 'numpy.int64'>
>>> np.dtype(int).itemsize
8
>>> np.dtype(int).byteorder
'='
```

Example: reading .wav files

The .wav file header:

chunk_id	"RIFF"
chunk_size	4-byte unsigned little-endian integer
format	"WAVE"
fmt_id	"fmt "
fmt_size	4-byte unsigned little-endian integer
audio_fmt	2-byte unsigned little-endian integer
num_channels	2-byte unsigned little-endian integer
sample_rate	4-byte unsigned little-endian integer
byte_rate	4-byte unsigned little-endian integer
block_align	2-byte unsigned little-endian integer
bits_per_sample	2-byte unsigned little-endian integer
data_id	"data"
data_size	4-byte unsigned little-endian integer

- 44-byte block of raw data (in the beginning of the file)
- … followed by data_size bytes of actual sound data.

The .wav file header as a NumPy *structured* data type:

```
>>>
>>> wav_header_dtype = np.dtype([
...     ("chunk_id", (bytes, 4)), # flexible-sized scalar type, item size 4
...     ("chunk_size", "<u4"),    # little-endian unsigned 32-bit integer
...     ("format", "S4"),         # 4-byte string
...     ("fmt_id", "S4"),
...     ("fmt_size", "<u4"),
...     ("audio_fmt", "<u2"),     #
...     ("num_channels", "<u2"),  # .. more of the same ...
...     ("sample_rate", "<u4"),   #
...     ("byte_rate", "<u4"),
...     ("block_align", "<u2"),
...     ("bits_per_sample", "<u2"),
...     ("data_id", ("S1", (2, 2))), # sub-array, just for fun!
...     ("data_size", "u4"),
...     #
...     # the sound data itself cannot be represented here:
...     # it does not have a fixed size
...     ])
```

See also wavreader.py

```
>>>
>>> wav_header_dtype['format']
dtype('S4')
>>> wav_header_dtype.fields
dict_proxy({'block_align': (dtype('uint16'), 32), 'format': (dtype('S4'), 8),
'data_id': (dtype(('S1', (2, 2))), 36), 'fmt_id': (dtype('S4'), 12), 'byte_rate':
(dtype('uint32'), 28), 'chunk_id': (dtype('S4'), 0), 'num_channels':
(dtype('uint16'), 22), 'sample_rate': (dtype('uint32'), 24), 'bits_per_sample':
(dtype('uint16'), 34), 'chunk_size': (dtype('uint32'), 4), 'fmt_size':
```

```
(dtype('uint32'), 16), 'data_size': (dtype('uint32'), 40), 'audio_fmt':
(dtype('uint16'), 20)})
>>> wav_header_dtype.fields['format']
(dtype('S4'), 8)
```

- The first element is the sub-dtype in the structured data, corresponding to the name format
- The second one is its offset (in bytes) from the beginning of the item

Exercise

Mini-exercise, make a “sparse” dtype by using offsets, and only some of the fields:

```
>>>
>>> wav_header_dtype = np.dtype(dict(
... names=['format', 'sample_rate', 'data_id'],
... offsets=[offset_1, offset_2, offset_3], # counted from start of structure in
bytes
... formats=list of dtypes for each of the fields,
... ))
```

and use that to read the sample rate, and data_id (as sub-array).

```
>>>
>>> f = open('data/test.wav', 'r')
>>> wav_header = np.fromfile(f, dtype=wav_header_dtype, count=1)
>>> f.close()
>>> print(wav_header)
[ ('RIFF', 17402L, 'WAVE', 'fmt ', 16L, 1, 1, 16000L, 32000L, 2, 16, [['d', 'a'],
['t', 'a']], 17366L)]
>>> wav_header['sample_rate']
array([16000], dtype=uint32)
```

Let’s try accessing the sub-array:

```
>>>
>>> wav_header['data_id']
array([[['d', 'a'],
        ['t', 'a']]],
      dtype='|S1')
>>> wav_header.shape
(1,)
>>> wav_header['data_id'].shape
(1, 2, 2)
```

When accessing sub-arrays, the dimensions get added to the end!

Note There are existing modules such as `wavfile`, `audiolab`, etc. for loading sound data…

Casting and re-interpretation/views

casting

- on assignment
- on array construction
- on arithmetic
- etc.
- and manually: `.astype(dtype)`

data re-interpretation

- manually: `.view(dtype)`

Casting

- Casting in arithmetic, in nutshell:

- only type (not value!) of operands matters
- largest “safe” type able to represent both is picked
- scalars can “lose” to arrays in some situations
- Casting in general copies data:
- >>>

```
>>> x = np.array([1, 2, 3, 4], dtype=np.float)
>>> x
array([1.,  2.,  3.,  4.])
>>> y = x.astype(np.int8)
>>> y
array([1, 2, 3, 4], dtype=int8)
>>> y + 1
array([2, 3, 4, 5], dtype=int8)
>>> y + 256
array([257, 258, 259, 260], dtype=int16)
>>> y + 256.0
array([257.,  258.,  259.,  260.])
>>> y + np.array([256], dtype=np.int32)
array([257, 258, 259, 260], dtype=int32)
```

- Casting on setitem: dtype of the array is not changed on item assignment:
- >>>

```
>>> y[:] = y + 1.5
>>> y
array([2, 3, 4, 5], dtype=int8)
```

Note Exact rules: see numpy documentation

Re-interpretation / viewing

- Data block in memory (4 bytes)

0x01	||	0x02	||	0x03	||	0x04

- 4 of uint8, OR,
- 4 of int8, OR,
- 2 of int16, OR,
- 1 of int32, OR,
- 1 of float32, OR,
- …
- How to switch from one to another?

1. Switch the dtype:
2. >>>

```
>>> x = np.array([1, 2, 3, 4], dtype=np.uint8)
>>> x.dtype = "<i2"
>>> x
array([ 513, 1027], dtype=int16)
>>> 0x0201, 0x0403
(513, 1027)
```

3.

0x01	0x02	||	0x03	0x04

Note little-endian: least significant byte is on the *left* in memory

2. Create a new view:
3. >>>

```
>>> y = x.view("<i4")
>>> y
array([67305985], dtype=int32)
>>> 0x04030201
67305985
```

4.

0x01	0x02	0x03	0x04

Note

- **.view() makes *views*, does not copy (or alter) the memory block**
- **only changes the dtype (and adjusts array shape):**
- >>>

```
>>> x[1] = 5
>>> y
array([328193], dtype=int32)
>>> y.base is x
True
```

-

Mini-exercise: data re-interpretation

See also view-colors.py

You have RGBA data in an array:

```
>>>
>>> x = np.zeros((10, 10, 4), dtype=np.int8)
>>> x[:, :, 0] = 1
>>> x[:, :, 1] = 2
>>> x[:, :, 2] = 3
>>> x[:, :, 3] = 4
```

where the last three dimensions are the R, B, and G, and alpha channels.

How to make a (10, 10) structured array with field names ‘r’, ‘g’, ‘b’, ‘a’ without copying data?

```
>>>
>>> y = ...

>>> assert (y['r'] == 1).all()
>>> assert (y['g'] == 2).all()
>>> assert (y['b'] == 3).all()
>>> assert (y['a'] == 4).all()
```

Solution

...

Another array taking exactly 4 bytes of memory:

```
>>>
```

```
>>> y = np.array([[1, 3], [2, 4]], dtype=np.uint8).transpose()
>>> x = y.copy()
>>> x
array([[1, 2],
       [3, 4]], dtype=uint8)
>>> y
array([[1, 2],
       [3, 4]], dtype=uint8)
>>> x.view(np.int16)
array([[ 513],
       [1027]], dtype=int16)
>>> 0x0201, 0x0403
(513, 1027)
>>> y.view(np.int16)
array([[ 769, 1026]], dtype=int16)
```

- What happened?
- … we need to look into what x[0,1] actually means

```
>>>
```

```
>>> 0x0301, 0x0402
(769, 1026)
```

Indexing scheme: strides

Main point

The question:

```
>>>
>>> x = np.array([[1, 2, 3],
...               [4, 5, 6],
...               [7, 8, 9]], dtype=np.int8)
>>> x.tobytes('A')
b'\x01\x02\x03\x04\x05\x06\x07\x08\t'
```

At which byte in ``x.data`` does the item ``x[1, 2]`` begin?

The answer (in NumPy)

- **strides**: the number of bytes to jump to find the next element
- 1 stride per dimension

```
 >>> x.strides
  (3, 1)
  >>> byte_offset = 3*1 + 1*2   # to find x[1, 2]
  >>> x.flat[byte_offset]
  6
  >>> x[1, 2]
  6
```

- simple, **flexible**

C and Fortran order

Note The Python built-in **bytes** returns bytes in C-order by default which can cause confusion when trying to inspect memory layout. We use **numpy.ndarray.tobytes()** with order=A instead, which preserves the C or F ordering of the bytes in memory.

```
>>>
>>> x = np.array([[1, 2, 3],
...               [4, 5, 6]], dtype=np.int16, order='C')
```

```
>>> x.strides
(6, 2)
>>> x.tobytes('A')
b'\x01\x00\x02\x00\x03\x00\x04\x00\x05\x00\x06\x00'
```

- Need to jump 6 bytes to find the next row
- Need to jump 2 bytes to find the next column

```
>>>
>>> y = np.array(x, order='F')
>>> y.strides
(2, 4)
>>> y.tobytes('A')
b'\x01\x00\x04\x00\x02\x00\x05\x00\x03\x00\x06\x00'
```

- Need to jump 2 bytes to find the next row
- Need to jump 4 bytes to find the next column
- Similarly to higher dimensions:
- C: last dimensions vary fastest (= smaller strides)
- F: first dimensions vary fastest
- $$\text{shape} = (d_1, d_2, ..., d_n)$$
$$\text{strides} = (s_1, s_2, ..., s_n)$$
$$s_j^C = d_{j+1} d_{j+2} ... d_n \times \text{itemsize}$$
$$s_j^F = d_1 d_2 ... d_{j-1} \times \text{itemsize}$$

Note Now we can understand the behavior of .view():

```
>>>
>>> y = np.array([[1, 3], [2, 4]], dtype=np.uint8).transpose()
>>> x = y.copy()
```

Transposition does not affect the memory layout of the data, only strides

```
>>>
```

```
>>> x.strides
(2, 1)
>>> y.strides
(1, 2)
```

```
>>>
```

```
>>> x.tobytes('A')
b'\x01\x02\x03\x04'
>>> y.tobytes('A')
b'\x01\x03\x02\x04'
```

- the results are different when interpreted as 2 of int16
- .copy() creates new arrays in the C order (by default)

Note In-place operations with views

Prior to NumPy version 1.13, in-place operations with views could result in **incorrect** results for large arrays. Since version 1.13, NumPy includes checks for *memory overlap* to guarantee that results are consistent with the non in-place version (e.g. a = a + a.T produces the same result as a += a.T). Note however that this may result in the data being copied (as if using a += a.T.copy()), ultimately resulting in more memory being used than might otherwise be expected for in-place operations!

Slicing with integers

- *Everything* can be represented by changing only shape, strides, and possibly adjusting the data pointer!

- Never makes copies of the data

```
>>>
>>> x = np.array([1, 2, 3, 4, 5, 6], dtype=np.int32)
>>> y = x[::-1]
>>> y
array([6, 5, 4, 3, 2, 1], dtype=int32)
>>> y.strides
(-4,)

>>> y = x[2:]
>>> y.__array_interface__['data'][0] - x.__array_interface__['data'][0]
8

>>> x = np.zeros((10, 10, 10), dtype=np.float)
>>> x.strides
(800, 80, 8)
>>> x[::2,::3,::4].strides
(1600, 240, 32)
```

- Similarly, transposes never make copies (it just swaps strides):
- >>>

```
>>> x = np.zeros((10, 10, 10), dtype=np.float)
>>> x.strides
(800, 80, 8)
>>> x.T.strides
(8, 80, 800)
```

-

But: not all reshaping operations can be represented by playing with strides:

```
>>>
>>> a = np.arange(6, dtype=np.int8).reshape(3, 2)
>>> b = a.T
>>> b.strides
(1, 2)
```

So far, so good. However:

```
>>>
>>> bytes(a.data)
b'\x00\x01\x02\x03\x04\x05'
>>> b
array([[0, 2, 4],
       [1, 3, 5]], dtype=int8)
>>> c = b.reshape(3*2)
>>> c
array([0, 2, 4, 1, 3, 5], dtype=int8)
```

Here, there is no way to represent the array `c` given one stride and the block of memory for `a`. Therefore, the `reshape` operation needs to make a copy here.

Example: fake dimensions with strides

Stride manipulation

```
>>>
>>> from numpy.lib.stride_tricks import as_strided
>>> help(as_strided)
as_strided(x, shape=None, strides=None)
  Make an ndarray from the given array with the given shape and strides
```

`as_strided` does **not** check that you stay inside the memory block bounds…

```
>>>
>>> x = np.array([1, 2, 3, 4], dtype=np.int16)
>>> as_strided(x, strides=(2*2, ), shape=(2, ))
array([1, 3], dtype=int16)
>>> x[::2]
array([1, 3], dtype=int16)
```

See also stride-fakedims.py

Exercise

```
array([1, 2, 3, 4], dtype=np.int8)

-> array([[1, 2, 3, 4],
          [1, 2, 3, 4],
          [1, 2, 3, 4]], dtype=np.int8)
```

using only as_strided.:

Hint: byte_offset = stride[0]*index[0] + stride[1]*index[1] + ...

Spoiler

...

Broadcasting

- Doing something useful with it: outer product of [1, 2, 3, 4] and [5, 6, 7]

```
>>>
>>> x = np.array([1, 2, 3, 4], dtype=np.int16)
>>> x2 = as_strided(x, strides=(0, 1*2), shape=(3, 4))
>>> x2
array([[1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4]], dtype=int16)

>>>
>>> y = np.array([5, 6, 7], dtype=np.int16)
>>> y2 = as_strided(y, strides=(1*2, 0), shape=(3, 4))
>>> y2
array([[5, 5, 5, 5],
       [6, 6, 6, 6],
       [7, 7, 7, 7]], dtype=int16)
```

```
>>>
>>> x2 * y2
array([[ 5, 10, 15, 20],
       [ 6, 12, 18, 24],
       [ 7, 14, 21, 28]], dtype=int16)
```

… seems somehow familiar …

```
>>>
>>> x = np.array([1, 2, 3, 4], dtype=np.int16)
>>> y = np.array([5, 6, 7], dtype=np.int16)
>>> x[np.newaxis,:] * y[:,np.newaxis]
array([[ 5, 10, 15, 20],
       [ 6, 12, 18, 24],
       [ 7, 14, 21, 28]], dtype=int16)
```

- Internally, array **broadcasting** is indeed implemented using 0-strides.

More tricks: diagonals

See also stride-diagonals.py

Challenge

- Pick diagonal entries of the matrix: (assume C memory order):

```
>>> x = np.array([[1, 2, 3],
...               [4, 5, 6],
...               [7, 8, 9]], dtype=np.int32)

>>> x_diag = as_strided(x, shape=(3,), strides=(???,))
```

-
- Pick the first super-diagonal entries [2, 6].
- And the sub-diagonals?

(Hint to the last two: slicing first moves the point where striding starts from.)

Solution

...

See also stride-diagonals.py

Challenge

Compute the tensor trace:

```
>>>
>>> x = np.arange(5*5*5*5).reshape(5, 5, 5, 5)
>>> s = 0
>>> for i in range(5):
...   for j in range(5):
...     s += x[j, i, j, i]
```

by striding, and using `sum()` on the result.

```
>>>
>>> y = as_strided(x, shape=(5, 5), strides=(TODO, TODO))
>>> s2 = ...
>>> assert s == s2
```

Solution

...

CPU cache effects

Memory layout can affect performance:

```
In [1]: x = np.zeros((20000,))

In [2]: y = np.zeros((20000*67,))[::67]
```

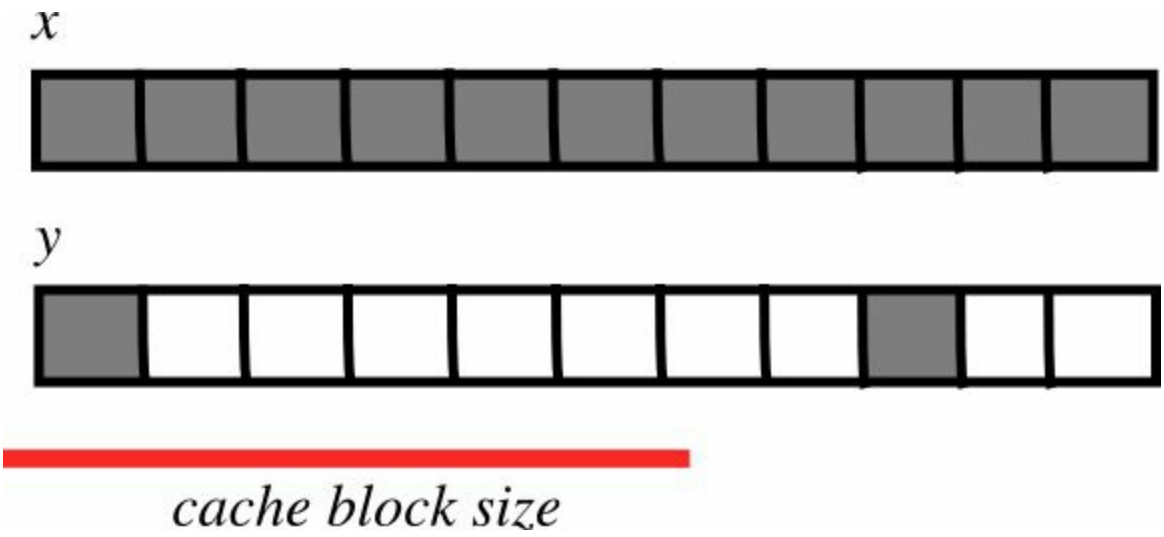
In [3]: x.shape, y.shape
((20000,), (20000,))

In [4]: %timeit x.sum()
100000 loops, best of 3: 0.180 ms per loop

In [5]: %timeit y.sum()
100000 loops, best of 3: 2.34 ms per loop

In [6]: x.strides, y.strides
((8,), (536,))

Smaller strides are faster?



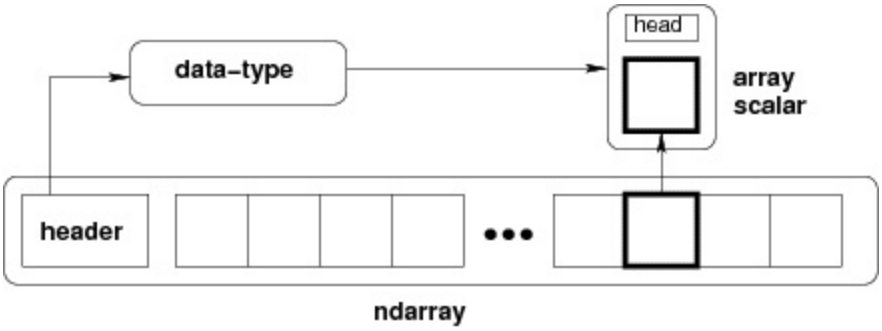
- CPU pulls data from main memory to its cache in blocks
- If many array items consecutively operated on fit in a single block (small stride):
- $\Rightarrow$ fewer transfers needed
- $\Rightarrow$ faster

See also

- numexpr is designed to mitigate cache effects when evaluating array expressions.

- numba is a compiler for Python code, that is aware of numpy arrays.

Findings in dissection



- *memory block*: may be shared, .base, .data
- *data type descriptor*: structured data, sub-arrays, byte order, casting, viewing, .astype(), .view()
- *strided indexing*: strides, C/F-order, slicing w/ integers, as_strided, broadcasting, stride tricks, diag, CPU cache coherence

Universal functions

What they are?

- Ufunc performs and elementwise operation on all elements of an array. Examples:

np.add, np.subtract, scipy.special.*, ...

-
- Automatically support: broadcasting, casting, …
- The author of an ufunc only has to supply the elementwise operation, NumPy takes care of the rest.
- The elementwise operation needs to be implemented in C (or, e.g.,

Cython)

Parts of an Ufunc

Provided by user

```
void ufunc_loop(void **args, int *dimensions, int *steps, void *data)
{
    /*
     * int8 output = elementwise_function(int8 input_1, int8 input_2)
     *
     * This function must compute the ufunc for many values at once,
     * in the way shown below.
     */
    char *input_1 = (char*)args[0];
    char *input_2 = (char*)args[1];
    char *output = (char*)args[2];
    int i;

    for (i = 0; i < dimensions[0]; ++i) {
        *output = elementwise_function(*input_1, *input_2);
        input_1 += steps[0];
        input_2 += steps[1];
        output += steps[2];
    }
}
```

1.

The NumPy part, built by

```
char types[3]

types[0] = NPY_BYTE   /* type of first input arg */
types[1] = NPY_BYTE   /* type of second input arg */
types[2] = NPY_BYTE   /* type of third input arg */

PyObject *python_ufunc = PyUFunc_FromFuncAndData(
```

```
    ufunc_loop,
    NULL,
    types,
    1, /* ntypes */
    2, /* num_inputs */
    1, /* num_outputs */
    identity_element,
    name,
    docstring,
    unused)
```

2.

- A ufunc can also support multiple different input-output type combinations.

Making it easier

3. ufunc_loop is of very generic form, and NumPy provides pre-made ones

PyUfunc_f_f	float elementwise_func(float input_1)
PyUfunc_ff_f	float elementwise_func(float input_1, float input_2)
PyUfunc_d_d	double elementwise_func(double input_1)
PyUfunc_dd_d	double elementwise_func(double input_1, double input_2)
PyUfunc_D_D	elementwise_func(npy_cdouble *input, npy_cdouble* output)
PyUfunc_DD_D	elementwise_func(npy_cdouble *in1, npy_cdouble *in2, npy_cdouble* out)

- Only elementwise_func needs to be supplied
- … except when your elementwise function is not in one of the above forms

Exercise: building an ufunc from scratch

The Mandelbrot fractal is defined by the iteration

$$z \leftarrow z^2 + c$$

where $c = x + iy$ is a complex number. This iteration is repeated – if z stays finite no matter how long the iteration runs, c belongs to the Mandelbrot set.

- Make ufunc called mandel(z0, c) that computes:

```
z = z0
for k in range(iterations):
    z = z*z + c
```

- say, 100 iterations or until z.real**2 + z.imag**2 > 1000. Use it to determine which c are in the Mandelbrot set.
- Our function is a simple one, so make use of the PyUFunc_* helpers.
- Write it in Cython

See also mandel.pyx, mandelplot.py

Reminder: some pre-made Ufunc loops:

PyUfunc_f_f	float elementwise_func(float input_1)
PyUfunc_ff_f	float elementwise_func(float input_1, float input_2)
PyUfunc_d_d	double elementwise_func(double input_1)
PyUfunc_dd_d	double elementwise_func(double input_1, double input_2)
PyUfunc_D_D	elementwise_func(complex_double *input,

	complex_double* output)
PyUfunc_DD_D	elementwise_func(complex_double *in1, complex_double *in2, complex_double* out)

Type codes:

NPY_BOOL, NPY_BYTE, NPY_UBYTE, NPY_SHORT, NPY_USHORT, NPY_INT, NPY_UINT,
NPY_LONG, NPY_ULONG, NPY_LONGLONG, NPY_ULONGLONG, NPY_FLOAT, NPY_DOUBLE,
NPY_LONGDOUBLE, NPY_CFLOAT, NPY_CDOUBLE,
NPY_CLONGDOUBLE, NPY_DATETIME,
NPY_TIMEDELTA, NPY_OBJECT, NPY_STRING, NPY_UNICODE, NPY_VOID

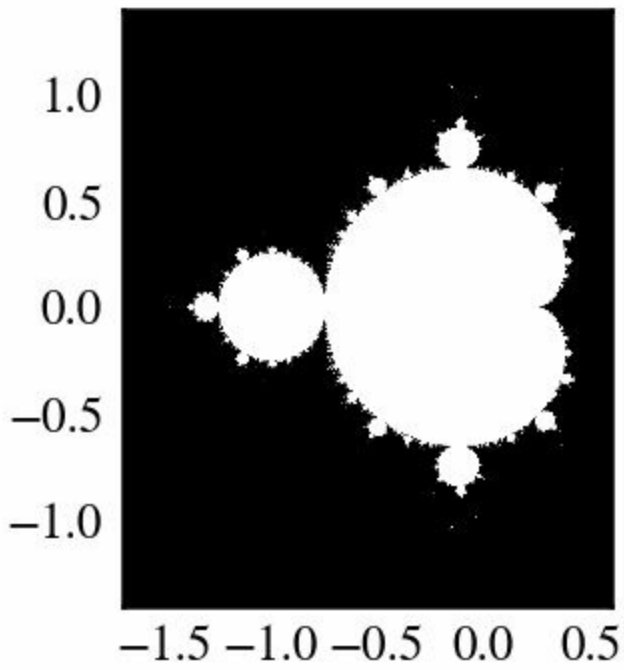
Solution: building an ufunc from scratch

```
# The elementwise function
# ------------------------

cdef void mandel_single_point(double complex *z_in,
                              double complex *c_in,
                              double complex *z_out) nogil:
    #
    # The Mandelbrot iteration
    #

    #
    # Some points of note:
    #
    # - It's *NOT* allowed to call any Python functions here.
    #
    #   The Ufunc loop runs with the Python Global Interpreter Lock released.
    #   Hence, the ``nogil``.
    #
    # - And so all local variables must be declared with ``cdef``
```

```
    #
    # - Note also that this function receives *pointers* to the data;
    #   the "traditional" solution to passing complex variables around
    #

    cdef double complex z = z_in[0]
    cdef double complex c = c_in[0]
    cdef int k  # the integer we use in the for loop

    # Straightforward iteration

    for k in range(100):
        z = z*z + c
        if z.real**2 + z.imag**2 > 1000:
            break

    # Return the answer for this point
    z_out[0] = z


# Boilerplate Cython definitions
#
# Pulls definitions from the Numpy C headers.
# -------------------------------------------

from numpy cimport import_array, import_ufunc
from numpy cimport (PyUFunc_FromFuncAndData,
                    PyUFuncGenericFunction)
from numpy cimport NPY_CDOUBLE
from numpy cimport PyUFunc_DD_D

# Required module initialization
# ------------------------------

import_array()
import_ufunc()
```

```
# The actual ufunc declaration
# --------------------------

cdef PyUFuncGenericFunction loop_func[1]
cdef char input_output_types[3]
cdef void *elementwise_funcs[1]

loop_func[0] = PyUFunc_DD_D

input_output_types[0] = NPY_CDOUBLE
input_output_types[1] = NPY_CDOUBLE
input_output_types[2] = NPY_CDOUBLE

elementwise_funcs[0] = <void*>mandel_single_point

mandel = PyUFunc_FromFuncAndData(
    loop_func,
    elementwise_funcs,
    input_output_types,
    1, # number of supported input types
    2, # number of input args
    1, # number of output args
    0, # `identity` element, never mind this
    "mandel", # function name
    "mandel(z, c) -> computes iterated z*z + c", # docstring
    0 # unused
    )

"""
Plot Mandelbrot
===============

Plot the Mandelbrot ensemble.
```

```
"""

import numpy as np
import mandel
x = np.linspace(-1.7, 0.6, 1000)
y = np.linspace(-1.4, 1.4, 1000)
c = x[None,:] + 1j*y[:,None]
z = mandel.mandel(c, c)

import matplotlib.pyplot as plt
plt.imshow(abs(z)**2 < 1000, extent=[-1.7, 0.6, -1.4, 1.4])
plt.gray()
plt.show()
```



Note Most of the boilerplate could be automated by these Cython modules:

https://github.com/cython/cython/wiki/MarkLodato-CreatingUfuncs

Several accepted input types

E.g. supporting both single- and double-precision versions

```
cdef void mandel_single_point(double complex *z_in,
                              double complex *c_in,
                              double complex *z_out) nogil:
    ...

cdef void mandel_single_point_singleprec(float complex *z_in,
                                         float complex *c_in,
                                         float complex *z_out) nogil:
    ...

cdef PyUFuncGenericFunction loop_funcs[2]
cdef char input_output_types[3*2]
cdef void *elementwise_funcs[1*2]

loop_funcs[0] = PyUFunc_DD_D
input_output_types[0] = NPY_CDOUBLE
input_output_types[1] = NPY_CDOUBLE
input_output_types[2] = NPY_CDOUBLE
elementwise_funcs[0] = <void*>mandel_single_point

loop_funcs[1] = PyUFunc_FF_F
input_output_types[3] = NPY_CFLOAT
input_output_types[4] = NPY_CFLOAT
input_output_types[5] = NPY_CFLOAT
elementwise_funcs[1] = <void*>mandel_single_point_singleprec

mandel = PyUFunc_FromFuncAndData(
    loop_func,
    elementwise_funcs,
    input_output_types,
    2, # number of supported input types   <----------------
    2, # number of input args
    1, # number of output args
```

```
    0, # `identity` element, never mind this
    "mandel", # function name
    "mandel(z, c) -> computes iterated z*z + c", # docstring
    0 # unused
    )
```

Generalized ufuncs

ufunc

output = elementwise_function(input)

Both output and input can be a single array element only.

generalized ufunc

output and input can be arrays with a fixed number of dimensions

For example, matrix trace (sum of diag elements):

input shape = (n, n)
output shape = () i.e. scalar

(n, n) -> ()

Matrix product:

input_1 shape = (m, n)
input_2 shape = (n, p)
output shape = (m, p)

(m, n), (n, p) -> (m, p)

- This is called the *"signature"* of the generalized ufunc
- The dimensions on which the g-ufunc acts, are *"core dimensions"*

Status in NumPy

- g-ufuncs are in NumPy already …
- new ones can be created with PyUFunc_FromFuncAndDataAndSignature
- most linear-algebra functions are implemented as g-ufuncs to enable working with stacked arrays:
- >>>

```
>>> import numpy as np
>>> np.linalg.det(np.random.rand(3, 5, 5))
array([ 0.00965823, -0.13344729,  0.04583961])
>>> np.linalg._umath_linalg.det.signature
'(m,m)->()'
```

-
- we also ship with a few g-ufuncs for testing, ATM:
- >>>

```
>>> import numpy.core.umath_tests as ut
>>> ut.matrix_multiply.signature
'(m,n),(n,p)->(m,p)'

>>> x = np.ones((10, 2, 4))
>>> y = np.ones((10, 4, 5))
>>> ut.matrix_multiply(x, y).shape
(10, 2, 5)
```

-
- in both examples the last two dimensions became *core dimensions*, and are modified as per the *signature*
- otherwise, the g-ufunc operates “elementwise”
- matrix multiplication this way could be useful for operating on many small matrices at once

Generalized ufunc loop

Matrix multiplication (m,n),(n,p) -> (m,p)

```
void gufunc_loop(void **args, int *dimensions, int *steps, void *data)
{
    char *input_1 = (char*)args[0];  /* these are as previously */
    char *input_2 = (char*)args[1];
    char *output = (char*)args[2];

    int input_1_stride_m = steps[3];  /* strides for the core dimensions */
    int input_1_stride_n = steps[4];  /* are added after the non-core */
    int input_2_strides_n = steps[5]; /* steps */
    int input_2_strides_p = steps[6];
    int output_strides_n = steps[7];
    int output_strides_p = steps[8];

    int m = dimension[1]; /* core dimensions are added after */
    int n = dimension[2]; /* the main dimension; order as in */
    int p = dimension[3]; /* signature */

    int i;

    for (i = 0; i < dimensions[0]; ++i) {
        matmul_for_strided_matrices(input_1, input_2, output,
                                    strides for each array...);

        input_1 += steps[0];
        input_2 += steps[1];
        output += steps[2];
    }
}
```

Interoperability features

Sharing multidimensional, typed data

Suppose you

1. Write a library than handles (multidimensional) binary data,
2. Want to make it easy to manipulate the data with NumPy, or whatever other library,
3. … but would **not** like to have NumPy as a dependency.

Currently, 3 solutions:

1. the “old” buffer interface
2. the array interface
3. the “new” buffer interface (**PEP 3118**)

The old buffer protocol

- Only 1-D buffers
- No data type information
- C-level interface; `PyBufferProcs tp_as_buffer` in the type object
- But it’s integrated into Python (e.g. strings support it)

Mini-exercise using Pillow (Python Imaging Library):

See also pilbuffer.py

```
>>>
>>> from PIL import Image
>>> data = np.zeros((200, 200, 4), dtype=np.int8)
>>> data[:, :] = [255, 0, 0, 255] # Red
>>> # In PIL, RGBA images consist of 32-bit integers whose bytes are [RR,GG,BB,AA]
>>> data = data.view(np.int32).squeeze()
>>> img = Image.frombuffer("RGBA", (200, 200), data, "raw", "RGBA", 0, 1)
>>> img.save('test.png')
```

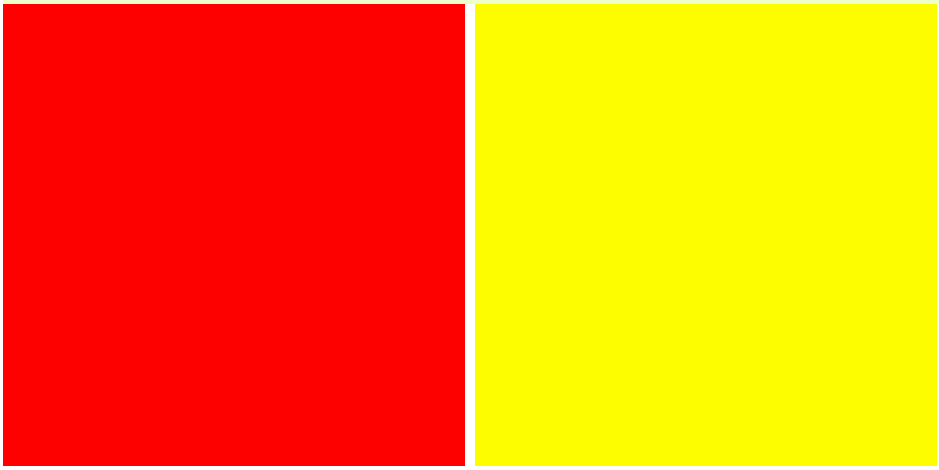
Q:

Check what happens if data is now modified, and img saved again.

The old buffer protocol

```
"""
From buffer
===========

Show how to exchange data between numpy and a library that only knows
the buffer interface.
"""

import numpy as np
import Image

# Let's make a sample image, RGBA format

x = np.zeros((200, 200, 4), dtype=np.int8)

x[:,:,0] = 254 # red
x[:,:,3] = 255 # opaque

data = x.view(np.int32) # Check that you understand why this is OK!

img = Image.frombuffer("RGBA", (200, 200), data)
img.save('test.png')

#
# Modify the original data, and save again.
#
# It turns out that PIL, which knows next to nothing about Numpy,
# happily shares the same data.
#
```

```
x[:,:,1] = 254
img.save('test2.png')
```



Array interface protocol

- Multidimensional buffers
- Data type information present
- NumPy-specific approach; slowly deprecated (but not going away)
- Not integrated in Python otherwise

See also Documentation:
http://numpy.org/doc/stable/reference/arrays.interface.html

```
>>>
>>> x = np.array([[1, 2], [3, 4]])
>>> x.__array_interface__
{'data': (171694552, False),      # memory address of data, is readonly?
'descr': [('', '<i4')],          # data type descriptor
'typestr': '<i4',                # same, in another form
'strides': None,                   # strides; or None if in C-order
'shape': (2, 2),
'version': 3,
}
```

::

```
>>>
```

```
>>> from PIL import Image
>>> img = Image.open('data/test.png')
>>> img.__array_interface__
{'data': ...,
'shape': (200, 200, 4),
'typestr': '|u1'}
>>> x = np.asarray(img)
>>> x.shape
(200, 200, 4)
```

Note A more C-friendly variant of the array interface is also defined.

Array siblings: chararray, maskedarray, matrix

chararray: vectorized string operations

```
>>>
```

```
>>> x = np.array(['a', '  bbb', '  ccc']).view(np.chararray)
>>> x.lstrip(' ')
chararray(['a', 'bbb', 'ccc'],
      dtype='...')
>>> x.upper()
chararray(['A', '  BBB', '  CCC'],
      dtype='...')
```

Note .view() has a second meaning: it can make an ndarray an instance of a specialized ndarray subclass

masked_array missing data

Masked arrays are arrays that may have missing or invalid entries.

For example, suppose we have an array where the fourth entry is invalid:

```
>>>
>>> x = np.array([1, 2, 3, -99, 5])

```

One way to describe this is to create a masked array:

```
>>>
>>> mx = np.ma.masked_array(x, mask=[0, 0, 0, 1, 0])
>>> mx
masked_array(data=[1, 2, 3, --, 5],
             mask=[False, False, False,  True, False],
       fill_value=999999)

```

Masked mean ignores masked data:

```
>>>
>>> mx.mean()
2.75
>>> np.mean(mx)
2.75

```

Not all NumPy functions respect masks, for instance np.dot, so check the return types.

The masked_array returns a **view** to the original array:

```
>>>
>>> mx[1] = 9
>>> x
array([  1,   9,   3, -99,   5])

```

The mask

You can modify the mask by assigning:

```
>>>
>>> mx[1] = np.ma.masked
>>> mx
masked_array(data=[1, --, 3, --, 5],
             mask=[False,  True, False,  True, False],
       fill_value=999999)

```

The mask is cleared on assignment:

```
>>>
>>> mx[1] = 9
>>> mx
masked_array(data=[1, 9, 3, --, 5],
             mask=[False, False, False,  True, False],
       fill_value=999999)

```

The mask is also available directly:

```
>>>
>>> mx.mask
array([False, False, False,  True, False])

```

The masked entries can be filled with a given value to get an usual array back:

```
>>>
>>> x2 = mx.filled(-1)
>>> x2
array([ 1,  9,  3, -1,  5])

```

The mask can also be cleared:

```
>>>
>>> mx.mask = np.ma.nomask
>>> mx
masked_array(data=[1, 9, 3, -99, 5],
             mask=[False, False, False, False, False],
       fill_value=999999)
```

Domain-aware functions

The masked array package also contains domain-aware functions:

```
>>>
>>> np.ma.log(np.array([1, 2, -1, -2, 3, -5]))
masked_array(data=[0.0, 0.6931471805599453, --, --, 1.0986122886681098,
--],
             mask=[False, False,  True,  True, False,  True],
       fill_value=1e+20)
```

Note Streamlined and more seamless support for dealing with missing data in arrays is making its way into NumPy 1.7. Stay tuned!

Example: Masked statistics

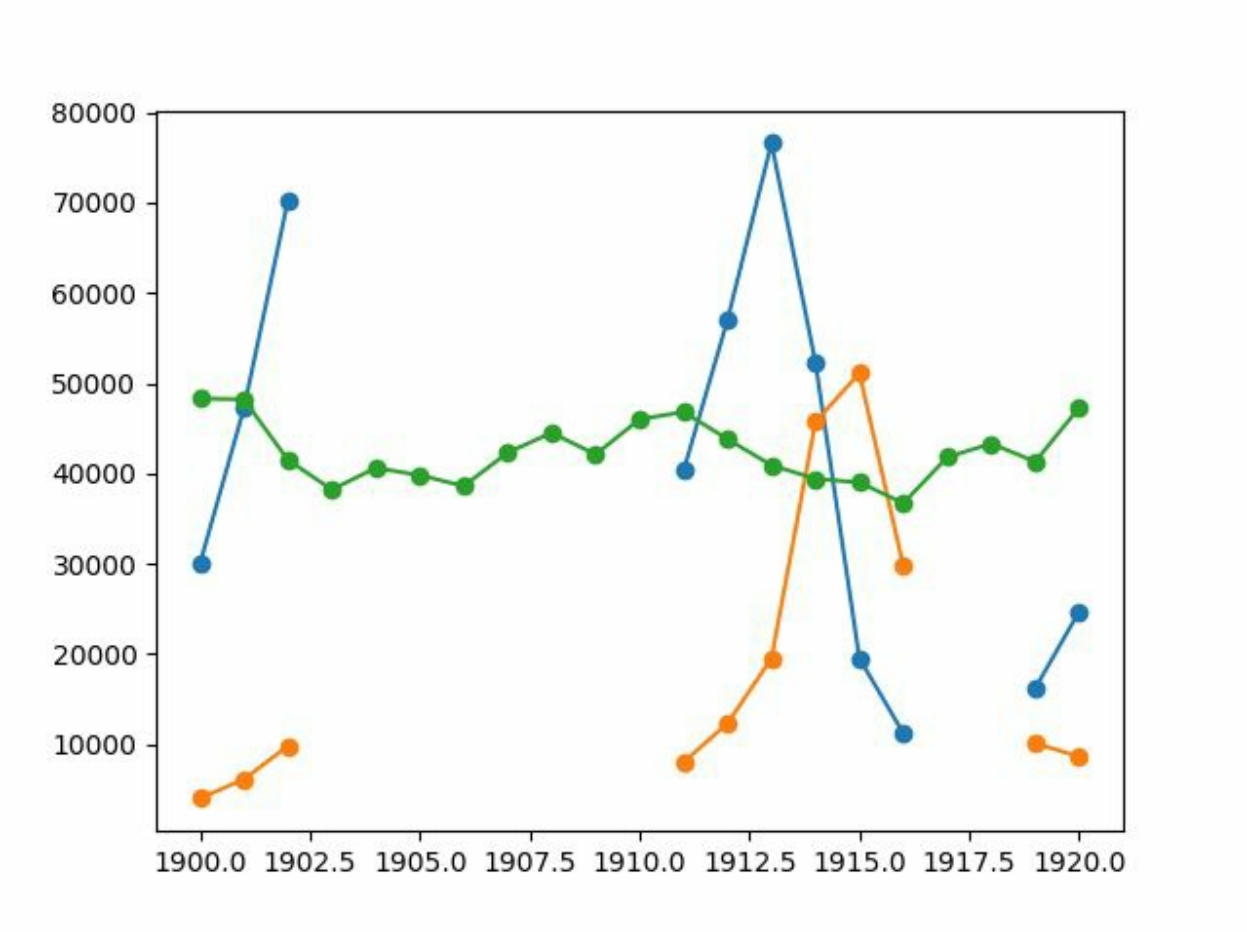
Canadian rangers were distracted when counting hares and lynxes in 1903-1910 and 1917-1918, and got the numbers are wrong. (Carrot farmers stayed alert, though.) Compute the mean populations over time, ignoring the invalid numbers.

```
>>>
>>> data = np.loadtxt('data/populations.txt')
>>> populations = np.ma.masked_array(data[:,1:])
>>> year = data[:, 0]

>>> bad_years = (((year >= 1903) & (year <= 1910))
```

```
...           | ((year >= 1917) & (year <= 1918)))
>>> # '&' means 'and' and '|' means 'or'
>>> populations[bad_years, 0] = np.ma.masked
>>> populations[bad_years, 1] = np.ma.masked

>>> populations.mean(axis=0)
masked_array(data=[40472.72727272727, 18627.272727272728, 42400.0],
             mask=[False, False, False],
       fill_value=1e+20)

>>> populations.std(axis=0)
masked_array(data=[21087.656489006717, 15625.799814240254,
3322.5062255844787],
             mask=[False, False, False],
       fill_value=1e+20)
```

Note that Matplotlib knows about masked arrays:

```
>>>
>>> plt.plot(year, populations, 'o-')
[<matplotlib.lines.Line2D object at ...>, ...]
```



recarray: purely convenience

```
>>>
>>> arr = np.array([('a', 1), ('b', 2)], dtype=[('x', 'S1'), ('y', int)])
>>> arr2 = arr.view(np.recarray)
recarray: purely convenience>>> arr2.x
chararray(['a', 'b'],
      dtype='|S1')
>>> arr2.y
array([1, 2])
```

matrix: convenience?

- always 2-D
- * is the matrix product, not the elementwise one

```
>>>
>>> np.matrix([[1, 0], [0, 1]]) * np.matrix([[1, 2], [3, 4]])
matrix([[1, 2],
        [3, 4]])
```

Summary

- Anatomy of the ndarray: data, dtype, strides.
- Universal functions: elementwise operations, how to make new ones
- Ndarray subclasses
- Various buffer interfaces for integration with other tools
- Recent additions: PEP 3118, generalized ufuncs

Why

- “There’s a bug?”
- “I don’t understand what this is supposed to do?”
- “I have this fancy code. Would you like to have it?”
- “I’d like to help! What can I do?”

Good bug report

Title: numpy.random.permutations fails **for** non-integer arguments

I'm trying to generate random permutations, using

numpy.random.permutations

When calling numpy.random.permutation with non-integer arguments it fails with a cryptic error message::

```
>>> np.random.permutation(12)
array([11,  5,  8,  4,  6,  1,  9,  3,  7,  2, 10,  0])
>>> np.random.permutation(12.) #doctest: +SKIP
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mtrand.pyx", line 3311, in mtrand.RandomState.permutation
  File "mtrand.pyx", line 3254, in mtrand.RandomState.shuffle
TypeError: len() of unsized object
```

This also happens with long arguments, and so np.random.permutation(X.shape[0]) where X is an array fails on 64 bit windows (where shape is a tuple of longs).

It would be great if it could cast to integer or at least raise a proper error for non-integer types.

For any query contact on markzuker827@gmail.com

Key and Imports

In this cheat sheet, we use the following shorthand:

arr | A NumPy Array object You'll also need to import numpy to get started:

import numpy as np

Importing/exporting

np.loadtxt('file.txt') | From a text file np.genfromtxt('file.csv',delimiter=',') | From a CSV file np.savetxt('file.txt',arr,delimiter=' ') | Writes to a text file

np.savetxt('file.csv',arr,delimiter=',') | Writes to a CSV file

Creating Arrays

np.array([1,2,3]) | One dimensional array np.array([(1,2,3),(4,5,6)]) | Two dimensional

array

np.zeros(3) | 1D array of length 3 all values 0

np.ones((3,4)) | 3x4 array with all values 1

np.eye(5) | 5x5 array of 0 with 1 on diagonal (Identity matrix)

np.linspace(0,100,6) | Array of 6 evenly divided values from 0 to 100 np.arange(0,10,3) | Array of values from 0 to less than 10 with step 3 (eg [0,3,6,9])

np.full((2,3),8) | 2x3 array with all values 8 np.random.rand(4,5) | 4x5 array of random floats between 0–1 np.random.rand(6,7)*100 | 6x7 array of random floats between 0–100 np.random.randint(5,size=(2,3)) | 2x3 array with random ints between 0–4

Inspecting Properties

arr.size | Returns number of elements in arr

arr.shape | Returns dimensions of arr (rows,columns)

arr.dtype | Returns type of elements in arr

arr.astype(dtype) | Convert arr elements to type dtype arr.tolist() | Convert arr to a Python list

np.info(np.eye) | View documentation for np.eye

Copying/sorting/reshaping

np.copy(arr) | Copies arr to new memory

arr.view(dtype) | Creates view of arr elements with type

dtype arr.sort() | Sorts arr

arr.sort(axis=0) | Sorts specific axis of

arr two_d_arr.flatten() | Flattens 2D array two_d_arr to 1D arr.T | Transposes arr (rows become columns and vice versa) arr.reshape(3,4) | Reshapes arr to 3 rows, 4 columns without changing data

arr.resize((5,6)) | Changes arr shape to 5x6 and fills new values with 0

Adding/removing Elements

np.append(arr,values) | Appends values to end of arr np.insert(arr,2,values) | Inserts values into arr before index 2 np.delete(arr,3,axis=0) | Deletes row on index 3 of arr np.delete(arr,4,axis=1) | Deletes column on index 4 of arr

Combining/splitting

np.concatenate((arr1,arr2),axis=0) | Adds arr2 as rows to the end of arr1

np.concatenate((arr1,arr2),axis=1) | Adds arr2 as columns to end of arr1

np.split(arr,3) | Splits arr into 3 sub-arrays

np.hsplit(arr,5) | Splits arr horizontally on the 5th index

Indexing/slicing/subsetting

arr[5] | Returns the element at index 5

arr[2,5] | Returns the 2D array element on index [2][5] arr[1]=4 | Assigns array element on index 1 the value 4 arr[1,3]=10 | Assigns array element on index [1][3] the value 10

arr[0:3] | Returns the elements at indices 0,1,2 (On a 2D array: returns rows 0,1,2)

arr[0:3,4] | Returns the elements on rows 0,1,2 at column 4 arr[:2] | Returns the elements at indices 0,1 (On a 2D array: returns rows 0,1)

arr[:,1] | Returns the elements at index 1 on all rows arr<5 | Returns an array with boolean values

(arr1<3) & (arr2>5) | Returns an array with boolean values ~arr | Inverts a boolean array

arr[arr<5] | Returns array elements smaller than 5

Scalar Math

np.add(arr,1) | Add 1 to each array element

np.subtract(arr,2) | Subtract 2 from each array element np.multiply(arr,3) | Multiply each array element by 3 np.divide(arr,4) | Divide each array element by 4 (returns np.nan for division by zero)

np.power(arr,5) | Raise each array element to the 5th power

Vector Math

np.add(arr1,arr2) | Elementwise add arr2 to arr1 np.subtract(arr1,arr2) | Elementwise subtract arr2 from arr1 np.multiply(arr1,arr2) | Elementwise multiply arr1 by arr2 np.divide(arr1,arr2) | Elementwise divide arr1 by arr2 np.power(arr1,arr2) | Elementwise raise arr1 raised to the power of arr2

np.array_equal(arr1,arr2) | Returns True if the arrays have the same elements and shape

np.sqrt(arr) | Square root of each element in the array np.sin(arr) | Sine of each element in the array

np.log(arr) | Natural log of each element in the array np.abs(arr) | Absolute value of each element in the array np.ceil(arr) | Rounds up to the nearest int

np.floor(arr) | Rounds down to the nearest int

np.round(arr) | Rounds to the nearest int

Statistics

np.mean(arr,axis=0) | Returns mean along specific axis arr.sum() | Returns sum of arr

arr.min() | Returns minimum value of arr

arr.max(axis=0) | Returns maximum value of specific axis np.var(arr) | Returns the variance of array

np.std(arr,axis=1) | Returns the standard deviation of specific axis

arr.corrcoef() | Returns correlation coefficient of array

*For any query contact **markzuker827@gmail.com***