NUMPY

What is NumPy?

Numpy is an open-source library for working efficiently with arrays.

Why is NumPy?

It dramatically improves the ease and performance of working with multidimensional arrays.

Some of Numpy's advantages:

1. Mathematical operations on NumPy's ndarray objects are up to 50x faster than iterating over native Python lists using loops. The efficiency gains are primarily due to NumPy storing array elements in an ordered single location within memory, eliminating redundancies by having all elements be the same type and making full use of modern CPUs. The efficiency advantages become particularly apparent when operating on arrays with thousands or millions of elements, which are pretty standard within data science.
2. It offers an Indexing syntax for easily accessing portions of data within an array.
3. It contains built-in functions that improve quality of life when working with arrays and math, such as functions for linear algebra, array transformations, and matrix math.
4. It requires fewer lines of code for most mathematical operations than native Python lists.

What's the relationship between NumPy, SciPy, Scikit-learn, and Pandas?

- NumPy provides a foundation on which other data science packages are built, including SciPy, Scikit-learn, and Pandas.

- SciPy provides a menu of libraries for scientific computations. It extends NumPy by including integration, interpolation, signal processing, more linear algebra functions, descriptive and inferential statistics, numerical optimizations, and more.

- Scikit-learn extends NumPy and SciPy with advanced machine-learning algorithms.

-
- Pandas extends NumPy by providing functions for exploratory data analysis, statistics, and data visualization. It can be thought of as Python's equivalent to Microsoft Excel spreadsheets for working with and exploring tabular data (tutorial).

  An Alternative to MATLAB?

  Many readers will likely be familiar with the commercial scientific computing software MATLAB. When used together with other Python libraries like Matplotlib, NumPy can be considered as a fully-fledged alternative to MATLAB's core functionality.

  Python is quite an attractive alternative to MATLAB for the following reasons:

- Python is open-source, which means that you have the option of inspecting the source code yourself.
- Access the vast and ever-growing possibilities open to Python users.
- Unlike MATLAB, Python and Numpy are free. No further explanation is needed!

  Installation

```
import numpy as np
```

  List of useful NumPy functions

  NumPy has numerous useful functions. You can see the full list of functions in the NumPy docs. As an overview, here are some of the most popular and useful ones to give you a sense of what NumPy can do. We will cover many of them in this tutorial.

- Array creation: arange, array, copy, empty, empty_like, eye, fromfile, fromfunction, identity, linspace, logspace, mgrid, ogrid, ones, ones_like, r_, zeros, zeros_like
- Conversions: ndarray.astype, atleast_1d, atleast_2d, atleast_3d, mat
- Manipulations: array_split, column_stack, concatenate, diagonal, dsplit, dstack, hsplit, hstack, ndarray.item, newaxis, ravel, repeat, reshape, resize, squeeze, swapaxes, take, transpose, vsplit, vstack
- Questions: all, any, nonzero, where

- Ordering: argmax, argmin, argsort, max, min, ptp, searchsorted, sort
- Operations: choose, compress, cumprod, cumsum, inner, ndarray.fill, imag, prod, put, putmask, real, sum
- Basic Statistics: cov, mean, std, var
- Basic Linear Algebra: cross, dot, outer, linalg.svd, vdot

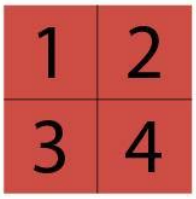
NumPy arrays

The NumPy array - an n-dimensional data structure - is the central object of the NumPy package.

A one-dimensional NumPy array can be thought of as a vector, a two-dimensional array as a matrix (i.e., a set of vectors), and a three-dimensional array as a tensor (i.e., a set of matrices).

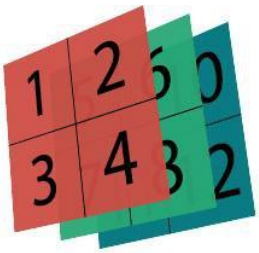
Vector

```
np.array([1, 2])
```



Matrix

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



3D Matrix

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

Array data types

An array can consist of integers, floating-point numbers, or strings. Within an array, the data type must be consistent (e.g., all integers or all floats).

Need an array with mixed data types? Consider using Numpy's *record array* format or pandas dataframes instead

Defining arrays

Using np.array()

To define an array manually, we can use the `np.array()` function.

```
np.array([[1,2],[3,4]])  # 2x2 matrix
```

NumPy functions:

Defining arrays: np.arange()

The function np.arange() is great for creating vectors easily. Here, we create a vector with values spanning 1 up to 4:

```
np.arange(1,5)
```

OUT:

```
array([1, 2, 3, 4])
```

Defining arrays: np.zeros, np.ones, np.full

In many programming tasks, it can be useful to initialize a variable and then write a value to it later in the code. If that variable happens to be a NumPy array, a common approach would be to create it as an array with zeros in every element.

We can do this using `np.zeros()`. Here, we create an array of zeros with three rows and one column.

```
np.zeros((3,1))
```

OUT:

```
array([[0.],
       [0.],
       [0.]])
```

You can also initialize an array with ones instead of zeros:

```
np.ones((3, 1))
```

OUT:

```
array([[1.],
       [1.],
       [1.]])
```

`np.full()` creates an array repeating a fixed value (defaults to zero). Here we create a 2x3 array with the number 7 in each element:

```
np.full((2,3),7)
```

OUT:

```
array([[7, 7, 7],
       [7, 7, 7]])
```

Making arrays in this way is also helpful for appending columns or rows to an existing arrays, which will be covered a little later.

Array shape

All arrays have a shape accessible using `.shape`.
For example, let's get the shape of a vector, matrix, and tensor.

```
vector = np.arange(5)
print("Vector shape:", vector.shape)

matrix = np.ones([3, 2])
print("Matrix shape:", matrix.shape)

tensor = np.zeros([2, 3, 3])
print("Tensor shape:", tensor.shape)
```

OUT:

```
Vector shape: (5,)
Matrix shape: (3, 2)
Tensor shape: (2, 3, 3)
```

The shape of the vector is one-dimensional. The first number in its shape is the number of elements (or rows). For the matrix, `.shape` tells us we have three rows and two columns. The tensor is slightly different. The first number is how many matrices/slices we have. The second gives the number of rows. The third provides the number of columns.

Reshaping arrays

We can reshape an array into any compatible dimensions using `.reshape` .

For example, say we want a 3x3 matrix where each element is incremented from 1 to 9. Easy:

```
arr = np.arange(1, 10)
print(arr, '\n')

# Reshape to 3x3 matrix
arr = arr.reshape(3, 3)
print(arr, '\n')

# Reshape back to the original size
arr = arr.reshape(9)
print(arr)
```

OUT:

```
[1 2 3 4 5 6 7 8 9]

[[1 2 3]
 [4 5 6]
 [7 8 9]]

[1 2 3 4 5 6 7 8 9]
```

Numpy can try to infer one of the dimensions if you use -1. You will still need to have precisely the correct number of digits for the inference to work.

```
arr = np.arange(1, 10).reshape(3, -1)
print(arr)
```

OUT:

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Reading data from a file into an array

Usually, data sets are too large to define manually. Instead, the most common use case is to import data from a data file into a NumPy array.

As an example, let's take some publicly-available data from the U.S. Energy Information Administration. The dataset we'll explore contains information on electricity generation in the USA from a range of sources. You can download the file, MER_T07_02A.csv,

here: https://www.eia.gov/totalenergy/data/browser/csv.php?tbl=T07.02A.

Because the data file is a CSV file, we'll use the `csv` module to import the data. It's worth noting that NumPy also has functions to read other types of data files directly into NumPy arrays, such as `np.genfromtxt()` for text files.

Here we're just reading the CSV file row-by-row, appending to a list, and then converting to a NumPy array:

```
import csv

data = []

with open('MER_T07_02A.csv', 'r') as csvfile:
    file_reader = csv.reader(csvfile, delimiter=',')
```

```
    for row in file_reader:
        data.append(row)

data = np.array(data) #convert the list of lists to a NumPy array

data.shape
```

OUT:

```
(8399, 6)
```

For this two-dimensional array, we have 8230 rows and 6 columns of data.

Another property of a NumPy array that we may wish to know is its data type. This information is stored in the dtype attribute. Calling dtype reveals that our array is made up of strings:

```
data.dtype.type
```

OUT:

```
numpy.str_
```

Saving

When we are ready to save our data, we can use the `save` function.

```
np.save(open('data.npy', 'wb'), data)
# Saves data to a binary file with the .npy extension
```

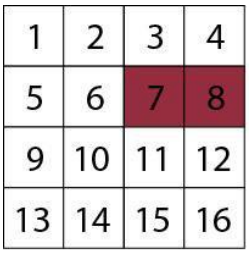
Indexing

At some point, it will become necessary to index (select) subsets of a NumPy array. For instance, you might want to plot one column of data or perform a manipulation of that column. NumPy uses the same indexing notation as MATLAB.

Basics of indexing notation

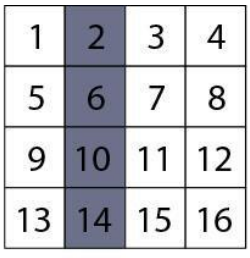
- Commas separate axes of an array.
- Colons mean "through". For example, x[0:4] means the first 5 rows (rows 0 through 4) of x.
- Negative numbers mean "from the end of the array." For example, x[-1] means the last row of x.
- Blanks before or after colons means "the rest of". For example, x[3:] means the rest of the rows in x after row 3. Similarly, x[:3] means all the rows up to row 3. x[:] means all rows of x.
- When there are fewer indices than axes, the missing indices are considered complete slices. For example, in a 3-axis array, x[0,0] means all data in the 3rd axis of the 1st row and 1st column.
- Dots "..." mean as many colons as needed to produce a complete indexing tuple. For example, x[1,2,...] is the same as x[1,2,:,:,:].
  In the following code, we'll explore some useful examples of selecting subsets from an array.

Examples



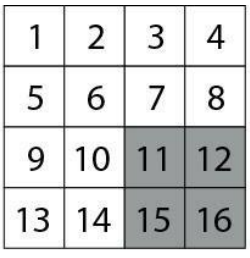
Row one, columns two to four

```
>>> arr[1, 2:4]
array([7, 8])
```



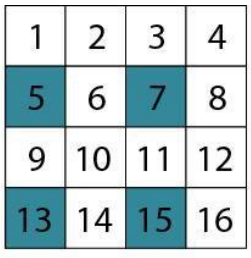
All rows in column one

```
>>> arr[:, 1]
array([2,  6, 10, 14])
```



All rows after row two,
all columns after column two

```
>>> arr[2:, 2:]
array([[11, 12],
       [15, 16]])
```



Every other row after row one,
every other column

```
>>> arr[1::2, ::2]
array([[5, 7],
       [13, 15]])
```

Indexing example 1: Colons and commas

Let's say we are interested in the first ten rows in the 4th column. We can use the following syntax to index this array section: __array[start_row:end_row, col]__

```
data[0:10,4]
```

Indexing example 1: Colons as *all* rows or columns

A colon can also denote *all* rows, or *all* columns. Here, we index all rows of column 4.

```
data[:,4]
```

Indexing example 3: Subset of columns

We can use the same format for any dimension of an array. The general syntax is: `array[start_row:end_row, start_col:end_col]`. The following indexes all rows and the second column up to (but not including) the 4th column:

```
data[:,2:4]
```

OUT:

```
array([['Value', 'Column_Order'],
       ['135451.32', '1'],
       ['154519.994', '1'],
       ...,
       ['334166.94', '13'],
       ['314400.63', '13'],
       ['301776.141', '13']], dtype='<U80')
```

Indexing example 4: Explicitly specifying column numbers

What if the columns we need are not next to each other? Instead of indexing a range of columns, it can be useful to specify them explicitly. To explicitly specify particular columns, we just include them in a list. Let's index the five rows after the header, selecting only columns 2 and 3. This time, we'll write the output to a new array named *subset* that we can re-use in the following example.

```
subset = data[1:6, [2,3]]
subset
```

OUT:

```
array([['135451.32', '1'],
       ['154519.994', '1'],
       ['185203.657', '1'],
       ['195436.666', '1'],
       ['218846.325', '1']], dtype='<U80')
```

Indexing example 5: Mask arrays

Another convenient way to index certain sections of a NumPy array is to use a mask array. A mask array, also known as a logical array, contains boolean elements (i.e. True or False). Indexing of a given array element is determined by the value of the mask array's corresponding element.

First, we define a NumPy array of True/False values, where the True values are the ones we want to keep. Then we mask the `subset` array from the previous example. The result is retaining only the rows that correspond to elements that are True in the mask array.

```
mask_array = np.array([False, True, False, True, True])

subset[mask_array]
```

OUT:

```
array([['154519.994', '1'],
       ['195436.666', '1'],
       ['218846.325', '1']], dtype='<U80')
```

Concatenating

NumPy also provides useful functions for concatenating (i.e., joining) arrays. Let's say we wanted to restrict our attention to the first and the last three rows of our dataset. First, we'll define new sub-arrays as follows:

```
array_start = data[:3,:]
array_start
```

```
array_end = data[-3:,:]
array_end
```

To concatenate these arrays we can use `np.vstack`, where the *v* denotes vertical, or row-wise, stacking of the sub-arrays:

```
np.vstack((array_start, array_end))
```

Here we've stacked the first three rows and last three rows on top of each other.

The horizontal counterpart of `np.vstack()` is `np.hstack()`, which combines sub-arrays column-wise. For higher dimensional joins, the most common function is `np.concatenate()`. The syntax for this function is similar to the 2D versions, with the additional requirement of specifying the axis along which concatenation should be performed.
Calling `np.concatenate((array_start, array_end), axis = 0)` would generate identical output to using `np.vstack()`. Axis=1 would generate identical output to using `np.hstack()`.

Splitting

The opposite of concatenating (i.e., joining) arrays is splitting them. To split an array, NumPy provides the following commands:

- `hsplit`: splits along the horizontal axis
- `vsplit`: splits along the vertical axis
- `dsplit`: Splits an array along the 3rd axis (depth)
- `array_split`: lets you specify the axis to use in splitting

Adding/Removing Elements

NumPy provides several functions for adding or deleting data from an array:

- `resize`: Returns a new array with the specified shape, with zeros as placeholders in all the new cells.
- `append`: Adds values to the end of an array
- `insert`: Adds values in the middle of an array
- `delete`: Returns a new array with given data removed
- `unique`: Finds only the unique values of an array

Sorting

There are several useful functions for sorting array elements. Some of the available sorting algorithms include `quicksort`, `heapsort`, `mergesort`, and `timesort`.
For example, here's how you'd merge sort the columns of an array:

```
a = np.array([[3,8,1,2], [9,5,4,8]])
np.sort(a, axis=1, kind='mergesort')              # Sort by column
```

OUT:

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

No Copy vs. Shallow Copy vs. Deep Copy

A common source of confusion NumPy beginners is knowing when data is and isn't copied into a new object.

No copy: function calls and assignments:

```
print(id(a))

# Object "b" points to object "a". No new object is created.
b = a
```

```
# Python passes objects as references. No copy is made.
def f(x):
    print(id(x))

f(b)
```

OUT:

```
2270228861696
2270228861696
```

Notice the id of `b` is the same as `a`, even if it's passed into a function.

View/Shallow Copy: Arrays that share some data. The view method creates an object looking at the same data. Slicing an array returns a view of that array.

```
# View
a = b.view()

# The shape of b doesn't change
a = a.reshape((4, 2))

# Slice
# a[:] is a view of "a".
a[:] = 5
```

Deep copy: Use the `copy` method to make a complete copy of an array and all its data.

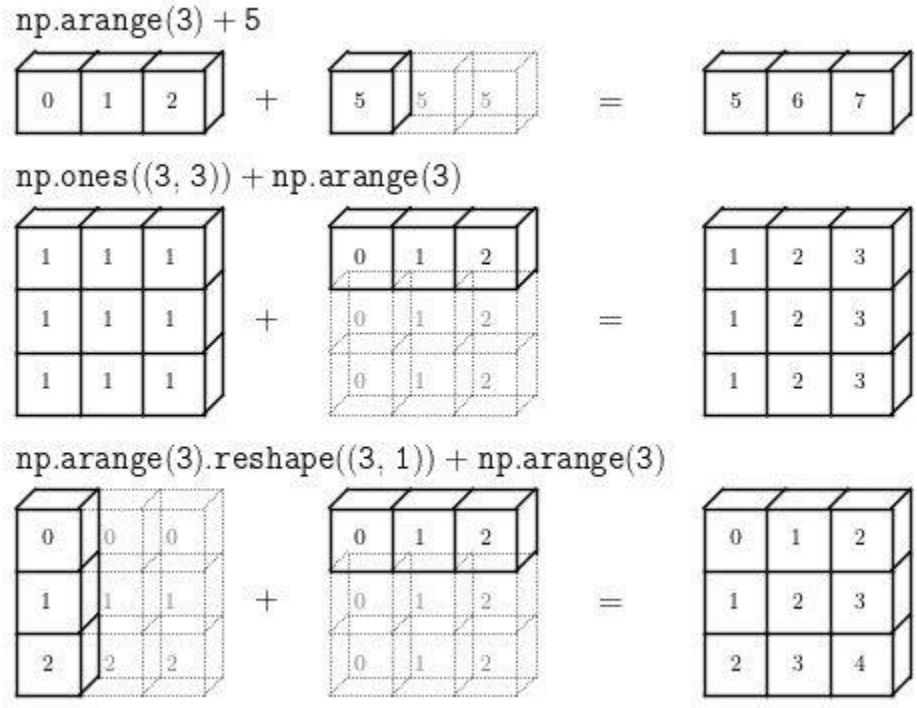
```
c = a.copy()
```

The `copy()` method creates the new array object `c` that is identical to `a`.

Section 2: Must-know tools

Let's now look at three NumPy tools that are especially handy in data science applications: *broadcasting*, *vectorization*, and *pseudo-random number generation*. For this section, we'll put our electricity dataset aside in favor of even more straightforward examples.

Broadcasting



Broadcasting is a process performed by NumPy that allows mathematical operations to work with objects that don't necessarily have compatible dimensions.

Let's explore broadcasting using some examples.

Broadcasting example 1: Adding a scalar to a matrix

Suppose we would like to add 1 to each element of a 2x2 array. With NumPy arrays, it is as simple as defining the array and adding 1:

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

```
array_a + 1
```

OUT:

```
array([[2, 3],
       [4, 5]])
```

Keep in mind that any university linear algebra instructor would be furious if you even mentioned the notion of adding a scalar to a matrix. And not without reason: it isn't a mathematically valid operation.

However, what NumPy is doing in the background *is* valid. NumPy creates a second array with value 1 for all elements (depicted by transparent blocks in the above figure). NumPy then adds the second array to the first one.
In other words, NumPy has *broadcast* the scalar to a new array of appropriate dimensions to perform the computation.
Numpy accomplishes broadcasting in a very computationally efficient way, which is one of the key advantages of using broadcasting in your code. Broadcasting may also make your code simpler and more readable.

Let's look at some more examples.

Broadcasting example 2: Multiplying a matrix by a scalar

Multiplication works the same way as addition.

```
array_a * 2
```

OUT:

```
array([[2, 4],
       [6, 8]])
```

Broadcasting example 3:

We can use broadcasting in cases beyond just overcoming the dimensional mismatch between a scalar and an array. NumPy can also broadcast arrays to enable computations with other arrays.

Let's say that each row of `array_a`, defined above, is a collection of two objects. The coordinates of the first object (first row of `array_a`) is located at (x = 1, y = 2), and the other object (second row of `array_a`) is located at (x = 3, y = 4). To find the coordinates of both objects if they both were translated by 3 units in the x direction and 1 unit in the y direction, all we would need to do is add (3, 1) to `array_a`:

```
array_a + np.array([3, 1])
```

OUT:

```
array([[4, 3],
       [6, 5]])
```

This time, NumPy created a second 2x2 matrix (in the background), with both rows equal to [3, 1], to perform the operation. In other words, Numpy *broadcasts* the 1x2 array to an array appropriate to perform the operation with the 2x2 array.
The operation is equivalent to the one depicted in the second row of the above figure.

For even more examples of broadcasting, the best place to look is the documentation.
Let's now move on to another essential tool: vectorization.

Vectorization

Vectorization is the process of modifying code to utilize array operation methods. Array operations can be computed internally by NumPy using a lower-level language, which leads to many benefits:

- Vectorized code tends to execute much faster than equivalent code that uses loops (such as for-loops and while-loops). Usually a *lot* faster. Therefore, vectorization can be very important for machine learning, where we often work with large datasets
- Vectorized code can often be more compact. Having fewer lines of code to write can potentially speed-up the code-writing process, make code more readable, and reduce the risk of errors

Vectorization Example 1

Let's consider a problem where we have two *one-dimensional* arrays, `a` and `b`, and we need to multiply each element in `a` with the corresponding element in `b`. First we'll define some arbitrary values for `a` and `b`:

```
a = np.arange(1,51)
b = np.arange(51,101)

print("array a:", a)
print("\narray b:", b)
```

OUT:

```
array a: [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
 49 50]

array b: [ 51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86
  87  88  89  90  91  92  93  94  95  96  97  98  99 100]
```

First we'll multiply the elements using a simple Python loop. This is the non-vectorized version:

```
def non_vectorized_output(a, b):
    output = []
    for j in range(len(a)):
        output.append(a[j]*b[j])
    return output
```

Calculating the speed up

Using Jupyter's magic `timeit` command, we can calculate how long it takes to run this function over many executions:

```
nv_time = %timeit -o non_vectorized_output(a, b)
```

OUT:

```
21.4 µs ± 506 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

The `%timeit -o` command will run a function over many executions and store the timing results in a variable. You can also just run `%timeit non_vectorized_output(a, b)` if you don't care about storing the result in a variable.
Now we'll use the multiplication operator between arrays to allow Numpy to handle the multiplication instead. This is the vectorized version:

```
def vectorized_output(a, b):
    return a * b
v_time = %timeit -o vectorized_output(a, b)
```

OUT:

```
701 ns ± 14.3 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

As you can see, the looping in the non-vectorized version is performed in pure Python (i.e., without using NumPy) with a for-loop. Although it would be challenging to make this non-vectorized code function any more compactly, it still occupies three more lines of code than the vectorized version. This compactness is in part because the looping in the vectorized version happens in the background.

It's clear that vectorized code is more compact, but what is the difference in computation time? Let's print the results in a way that's easier to read:

```
print('Non-vectorized version:', f'{1E6 * nv_time.average:0.2f}', 'microseconds per execution, average')

print('Vectorized version:', f'{1E6 * v_time.average:0.2f}', 'microseconds per execution, average')
```

```
print('Computation was', "%.0f" % (nv_time.average / v_time.average), 'times faster using vectorization')
```

OUT:

```
Non-vectorized version: 21.40 microseconds per execution, average
Vectorized version: 0.70 microseconds per execution, average
Computation was 31 times faster using vectorization
```

Vectorization example 2

In this second example, we'll evaluate a set of linear expressions. Again, this task could be accomplished either using for-loops or using vectorized code.

In this case, the vectorized version will use matrix multiplication to evaluate the linear expressions. If you're familiar with machine learning (ML), the next paragraph will provide some context about when you might encounter this in ML.

Pseudo-random number generation

Before we finish this section, there is one more NumPy functionality that we should cover: pseudo-random number generation.

Being able to generate pseudo-random numbers is often necessary in data science applications. Examples include modeling system noise and Monte Carlo simulations.

Below we'll see how to generate random numbers (x) from two commonly encountered probability distributions: the uniform distribution and the normal (Gaussian) distribution.
For this, we'll import `matplotlib` and set some default plotting styles:

```
import matplotlib.pyplot as plt
import matplotlib

# Set some default plotting parameters
matplotlib.rcParams.update({'font.size': 16,
                            'figure.figsize': [10, 6],
                            'lines.markersize': 6})
```

For the Uniform, we'll generate a NumPy array with 1000 samples randomly selected from a uniform distribution using `random.rand`.
For the Normal, we'll generate a NumPy array with 1000 samples from a normal distribution centred at 5 with a standard deviation of 3 using `random.normal`:

```
uniform_data = np.random.rand(1000)
normal_data = np.random.normal(loc=5, scale=3.0, size=1000)
```

Here's a plot of the histograms for the Uniform data (first subplot) and the Normal data (second subplot):

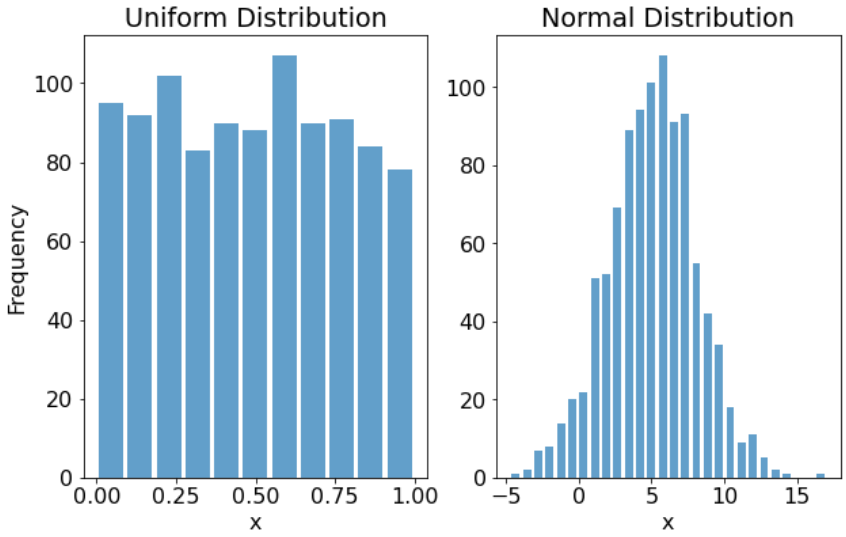
```
fig = plt.figure()  # Define the figure

ax1 = fig.add_subplot(1,2,1)  # define location of first subplot
ax1.hist(x=uniform_data, bins='auto',alpha=0.7, rwidth=0.85)
ax1.set_title("Uniform Distribution")

ax2 = fig.add_subplot(1,2,2)  # define location of second subplot
ax2.hist(x=normal_data, bins='auto', alpha=0.7, rwidth=0.7)
ax2.set_title("Normal Distribution")

ax1.set_ylabel('Frequency')
ax1.set_xlabel('x')
ax2.set_xlabel('x')
plt.show()
```

RESULT:



As we'd expect, uniform distribution's random values are more or less equally spaced between zero and one. By contrast, the values from the normal distribution take on the characteristic bell-curve shape.

We can now use the sets of random numbers we've just generated in further computations, but we'll leave that for another time. To wrap up this article, let's put everything we learned together using our electricity dataset.

Section 3: Putting it all together

Now that we know the basics of NumPy, broadcasting, and vectorization, we have everything we need to start diving into the electricity data that we imported at the start of this article.

Let's assume we'd like to understand how the USA's electricity generation has changed over time.

Viewing the data

Using what we've learned about indexing, we can start by separating the column labels from the rest of the data.

```
header = data[0,:]   # create a new NumPy array containing the column labels
data = data[1:,:]   # remove the header from the rest of the data

print('Header:\n',header, '\n\nFirst two rows:\n', data[:2,:])
```

OUT:

```
Header:
 ['MSN' 'YYYYMM' 'Value' 'Column_Order' 'Description' 'Unit']

First two rows:
 [['CLETPUS' '194913' '135451.32' '1'
  'Electricity Net Generation From Coal, All Sectors'
  'Million Kilowatthours']
 ['CLETPUS' '195013' '154519.994' '1'
```

```
 'Electricity Net Generation From Coal, All Sectors'
 'Million Kilowatthours']]
```

To understand how electricity generation has changed with time, we'll need to pay attention to column 1 (date), column 2 (energy generated), and column 4 (description).
In this dataset, rows containing monthly data express date in the format 'YYYYMM'. Rows containing annual data express the date in the format 'YYYY13'.

Our dataset happens to contain generation data from many different energy sources, so let's determine which energy sources are present in this dataset by inspecting the descriptions (column 4).

The `np.unique()` function makes it easy to see all energy sources. As the name suggests, it will return all unique values in the array.

```
np.unique(data[:,4])
```

This dataset contains information from a total of 13 categories of energy sources.

Extracting wind energy data

Next, we'll extract a subset containing just the wind energy generation data. We'll be making extensive use of indexing with mask arrays, which we looked at earlier.

Let's start by retaining only the rows that contain wind data. First, we'll create a mask array that contains True entries for every row of wind data:

```
mask_array = (data[:,4] == 'Electricity Net Generation From Wind, All Sectors')
mask_array
```

OUT:

```
array([False, False, False, ..., False, False, False])
```

What this mask array essentially says is "get all rows where column four equals 'Electricity Net Generation...'"

Now we can use it to mask our data:

```
wind_data = data[mask_array]
wind_data
```

Did you notice that we used broadcasting to generate the mask array? Broadcasting allowed the generation of a new array based on the logical evaluation of whether each string element in an array was equal to a single string. In the above output, we notice that some of the early rows contain the string _Not Available_ in the 'Value' column. _Not Available_ suggests that records only began later on. Let's exclude the rows for which no records exist:

```
wind_data = wind_data[wind_data[:,2] != 'Not Available']
wind_data
```

Note that the above code performed indexing using a mask array. For compactness, we didn't explicitly define the mask array as a separate object.

Now, let's retain only the annual data. In other words, keep only the rows where the value in column 1 ends with '13'. To do this, we use list comprehension (a pure Python formalism) to generate the mask array to perform the indexing.

```
annual_mask_array = np.array(([x[-2:] == '13' for x in wind_data[:,1]]))

wind_data = wind_data[annual_mask_array]
wind_data[:5]
```

We have now successfully isolated the annual wind data.

It is worth noting that it is straightforward to save a NumPy array to a text file using the `np.savetxt()` function.

Just for fun, let's save our results to a comma-delimited csv file. We will request that NumPy converts everything to a string format before exporting.

```
np.savetxt('wind.csv',wind_data, fmt = '%s', delimiter = ',')
```

Now let's define a new NumPy array containing just the annual wind energy produced, which is contained in column two of our wind data array. We will convert information to a float data type:

```
energy = wind_data[:,2].astype(float)
energy
```

Success! Now that we finally have the data of interest in an array of floating-point numbers, we can start taking advantage of some NumPy functions that can quickly and easily perform numerical operations on our array.

Mathematical functions

NumPy offers many mathematical functions that can be called with the syntax `array.method()`. For instance, if we wanted to compute the sum of all elements in the array, we could use the function `array.sum()`:

```
print(f'Total wind energy generated in the USA since 1983 is {energy.sum()} Gigawatt-hours')
```

OUT:

```
Total wind energy generated in the USA since 1983 is 2235263.7029999997 Gigawatt-hours
```

Easy.

NumPy functions are also available to calculate things like the mean and standard deviation:

```
print(f'The average annual energy generated from wind is {energy.mean()} Gigawatt-hours, '
    f'with a standard deviation of {100 * energy.std() / energy.mean():.2f}%')
```

OUT:

```
The average annual energy generated from wind is 60412.532513513506 Gigawatt-hours, with a standard deviation of 147.70%
```

We can quickly answer many questions using these functions. Here are a couple more.

What was the maximum annual energy generated?

```
print(f'The highest recorded annual energy generated by wind power is {energy.max()} Gigawatt-hours')
```

OUT:

```
The highest recorded annual energy generated by wind power is 294906.32 Gigawatt-hours
```

And in what year did that occur?

```
index = energy.argmax() #this method returns the index of the maximum value in the array
print(f'The highest energy generation occured in the year {wind_data[index,1][:-2]}')
```

OUT:

```
The highest energy generation occured in the year 2019
```