numpy

March 21, 2022

1 NUMPY QUICK TUTORIAL

1.1 About NumPy

NumPy (**Numerical Python**) is an open source Python library that's used in almost every field of science and engineering. It's the universal standard for working with numerical data in Python, and it's at the core of the scientific **Python** and **PyData ecosystems**. NumPy users include everyone from beginning coders to experienced researchers doing state-of-the-art scientific and industrial research and development. The NumPy API is used extensively in **Pandas**, **SciPy**, **Matplotlib**, **scikit-learn**, **scikit-image** and most other data science and scientific Python packages.

The **NumPy** library contains multidimensional array and matrix data structures (you'll find more information about this in later sections). It provides **ndarray**, a homogeneous ***n-dimensional*** array object, with methods to efficiently operate on it. NumPy can be used to perform a wide variety of mathematical operations on arrays. It adds powerful data structures to Python that guarantee efficient calculations with arrays and matrices and it supplies an enormous library of high-level mathematical functions that operate on these arrays and matrices.

1.2 Installing NumPy

To install NumPy, I strongly recommend using a scientific Python distribution.

If you already have Python, you can install NumPy with:

```
conda install numpy
```

or

```
pip install numpy
```

If you don't have Python yet, you might want to consider using Anaconda. It's the easiest way to get started. The good thing about getting this distribution is the fact that you don't need to worry too much about separately installing NumPy or any of the major packages that you'll be using for your data analyses, like pandas, Scikit-Learn, etc.

1.3 How to import NumPy

To access NumPy and its functions import it in your Python code like this:

```
import numpy as np
```

The shorten name `np` stand for numpy, is a better readability of code using NmuPy. This is a widely adopted convention that you should follow so that anyone working with your code can easily understand it.

1.4 What is the difference between a Python list and a NumPy array?

NumPy gives you an enormous range of fast and efficient ways of creating arrays and manipulating numerical data inside them. While a Python list can contain different data types within a single list, all of the elements in a NumPy array should be homogeneous. The mathematical operations that are meant to be performed on arrays would be extremely inefficient if the arrays weren't homogeneous.

> **Why use NumPy** NumPy arrays are faster and more compact than Python lists. An array consumes less memory and is convenient to use. NumPy uses much less memory to store data and it provides a mechanism of specifying the data types. This allows the code to be optimized even further.

1.5 What is an array?

An **array** is a central data structure of the NumPy library. An array is a grid of values and it contains information about the raw data, how to locate an element, and how to interpret an element. It has a grid of elements that can be indexed in various ways. The elements are all of the same type, referred to as the array `dtype`.

An array can be indexed by a tuple of nonnegative integers, by booleans, by another array, or by integers. The **rank** of the array is the number of dimensions. The **shape** of the array is a tuple of integers giving the **size** of the array along each dimension.

One way we can initialize NumPy arrays is from Python lists, using nested lists for two or higher-dimensional data.

For example:

```
[1]: import numpy as np
```

```
[2]: a = np.array([1, 2, 3, 4, 5, 6])
```

or

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

We can access the elements in the array using square brackets `[]`. When you're accessing elements, remember that indexing in NumPy start at 0 (like Python indexing). That means that if you want to access the first element in your array, you will be accesssing element "0".

```
[4]: a[0]
```

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

1.6 More information about arrays

This section covers `1D array`, `2D array`, `ndarray`, `vector`, `matrix`.

You might occasionally hear an array referred to as a "ndarray," which is shorthand for "N-dimensional array." An N-dimensional array is simply an array with any number of dimensions. You might also hear **1-D**, or one-dimensional array, **2-D**, or two-dimensional array, and so on. The NumPy ndarray class is used to represent both matrices and vectors. A **vector** is an array with a single dimension (there's no difference between row and column vectors), while a **matrix** refers to an array with two dimensions. For **3-D** or higher dimensional arrays, the term **tensor** is also commonly used.

What are the attributes of an array?

An array is usually a fixed-size container of items of the same type and size. The number of dimensions and items in an array is defined by its shape. The shape of an array is a tuple of non-negative integers that specify the sizes of each dimension.

In NumPy, dimensions are called axes. This means that if you have a 2D array that looks like this:

```
[[0., 0., 0.],
[1., 1., 1.]]
```

Your array has 2 axes. The first axis has a length of 2 and the second axis has a length of 3.

Just like in other Python container objects, the contents of an array can be accessed and modified by indexing or slicing the array. Unlike the typical container objects, different arrays can share the same data, so changes made on one array might be visible in another.

Array **attributes** reflect information intrinsic to the array itself. If you need to get, or even set, properties of an array without creating a new array, you can often access an array through its attributes.

1.7 How to create a basic array?

This section covers `np.array()`, `np.zeros()`, `np.ones()`, `np.empty()`, `np.arange()`, `np.linspace()`, `dtype`.

To create an NumPy array, you can use the function `np.array()`

All you need to do to create a simple array is pass a list to it. If you choose to, you can also specify the tyoe of data in your list.

```
[5]: a = np.array([1, 2, 3])
     a
```

```
[5]: array([1, 2, 3])
```

Beside creating an array from a sequence of elements, you can easily create an array filled with 0's.

```
[6]: np.zeros(2)
```

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

Or an array filles with 1's

```
[7]: np.ones(5)
```

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

Or even an empty array! The function `np.empty()` creates an array whose initial contents is random and depends on the state of the memory. The reason to use `empty` over `zeros` (or something similar) is speed - just make sure to fill every element afterward!

```
[8]: np.empty(4)
```

```
[8]: array([2.12199579e-314, 3.38277379e-292, 4.48611606e-321, 3.79442416e-321])
```

You can create an array with a range of elements.

```
[9]: np.arange(4)
```

```
[9]: array([0, 1, 2, 3])
```

And even an array that contains a range of evenly spaced intervals. To do this, you will specify the **first number**, **last number**, and the **step**.

```
[10]: np.arange(0, 10, 2)
```

```
[10]: array([0, 2, 4, 6, 8])
```

You can also use `np.linspace()` to create an array eith values that are spaced in a specified interval.

```
[11]: np.linspace(0, 10, num=5)
```

```
[11]: array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

1.7.1 Specifying your data type

While the default data type is floating point (`np.float64`), you can explicitly specify which data type you want using the `dtype` keyword.

```
[12]: x = np.ones(3, dtype=np.int64)
      x
```

```
[12]: array([1, 1, 1], dtype=int64)
```

1.8 Adding, removing, and sorting elements

This section covers `np.sort()`, `np.concatenate()`.

Sorting an element is simple with `np.sort()`. You can specify the axis, kind and oder when you call the function.

If you start with this array:

```
[13]: arr = np.array([-1, -4, 100, 3, 99, 1.5, 0, 15])
```

You can quickly sort the numbers in ascending order with

```
[14]: np.sort(arr)
```

```
[14]: array([ -4. ,  -1. ,   0. ,   1.5,   3. ,  15. ,  99. , 100. ])
```

In addition to sort, which returns a sorted copy of an array, you can use:

- `argsort`, which is an indirect sort along a specified.
- `lexsort`, which is an indirect stable sort on multiple keys.
- `searchsorted`, which will find elements in a sorted array.
- `partition`, which is a partial sort.

If you start with these arrays

```
[15]: a = np.array([1, 2, 3, 4])
      b = np.array([5, 6, 7, 8])
```

You can concatenate them with `np.concatenate()`.

```
[16]: np.concatenate((a, b))
```

```
[16]: array([1, 2, 3, 4, 5, 6, 7, 8])
```

Or if you start with these arrays

```
[17]: x = np.array([[1, 2], [3, 4]])
      y = np.array([[5, 6], [7, 8]])
```

You can concatenate them with

```
[18]: np.concatenate((x, y), axis=0)
```

```
[18]: array([[1, 2],
             [3, 4],
             [5, 6],
             [7, 8]])
```

```
[19]: np.concatenate((x, y), axis=1)
```

```
[19]: array([[1, 2, 5, 6],
             [3, 4, 7, 8]])
```

In order to remove elements from an array, it's simple to use indexing to select the elements that you want to keep.

1.9 How to know the shape and size of an array?

This section covers `ndarray.ndim`, `ndarray.size`, `ndarray.shape`.

- `ndarray.ndim` will tell you the number of axes, or dimensions of the array.
- `ndarray.size` will tell you the total number of elements of the array. This is the *product* of the elements of the array's shape.
- `ndarray.shape` will display a tuple of integers that indicate the number of elements stored along each dimension of the array. If, for example, you have a 2-D array with 2 rows and 3 columns, the shape of your array is (2, 3).

For example, if you create this array:

```
[20]: arr_example = np.array(
          [[[0, 1, 2, 3],
           [4, 5, 6, 7]]] * 3
      )

      arr_example
```

```
[20]: array([[[0, 1, 2, 3],
              [4, 5, 6, 7]],

             [[0, 1, 2, 3],
              [4, 5, 6, 7]],

             [[0, 1, 2, 3],
              [4, 5, 6, 7]]])
```

To find the number of dimensions of the array, run:

```
[21]: arr_example.ndim
```

```
[21]: 3
```

To find the total number of elements in the array, run:

```
[22]: arr_example.size
```

```
[22]: 24
```

And to find the shape of your array, run:

arr_example.shape

1.10 Can I reshape an array?

This section covers `arr.reshape()`

Using `arr.reshape()` will give a new shape to an array without changing the data. Just remember that when you use the reshape method, the array you want to produce needs to have the same number of elements as the original array. If you start with an array with 12 elements, you'll need to make sure that your new array also has a total of 12 elements.

If you start with this array

```
[23]: a = np.arange(12)
      a
```

```
[23]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

You can use `reshape()` to reshape your array. For example, you can reshape this array to an array with 3 rows and 4 columns

```
[24]: b = a.reshape((3, 4))
      b
```

```
[24]: array([[ 0,  1,  2,  3],
             [ 4,  5,  6,  7],
             [ 8,  9, 10, 11]])
```

With `np.reshape()`, you can specify a few optional parameters.

```
[25]: np.reshape(a, newshape=(6, 2), order="C")
```

```
[25]: array([[ 0,  1],
             [ 2,  3],
             [ 4,  5],
             [ 6,  7],
             [ 8,  9],
             [10, 11]])
```

- `a` is the array to be reshaped
- `newshape` is the new shape you want. You can specify an integer or a tuple of integers. If you specify a integer, the result will be an array of that length. The shape should be compatible with the original shape.
- `order:` `C` means to read/write the elements using C-like index order, F means to read/write the elements using Fortran-like index order, A means to read/write the elements in Fortran-

like index order if a is Fortran contiguous in memory, C-like order otherwise. (This is an optional parameter and doesn't need to be specified.)

1.11 How to convert a 1D array into a 2D array (how to add an new axis to an array)?

This section covers `np.newaxis`, `np.expand_dims`.

You can use `np.newaxis` and `np.expand_dims` to increase the dimensions of your existing array.

Using `np.newaxis` will increase the dimensions of your array by one dimension when used once. This means that a **1D** array will become a **2D** array, a **2D** array will become a **3D** array, and so on.

For example, if you start with this array

[26]:
```
x = np.array([-3, -2, -1, 0, 1, 2, 3])
x.shape
```

[26]:
```
(7,)
```

You can use `np.newaxis` to add a new axis.

[27]:
```
y = x[np.newaxis, :]
y.shape
```

[27]:
```
(1, 7)
```

You can explicitly convetr a 1D array with either a row vector or a column vector using `np.newaxis`. For example, you can convert a 1D array to a row vector by inserting an axis along the first dimension.

[28]:
```
row_vector = x[np.newaxis, :]
row_vector.shape
```

[28]:
```
(1, 7)
```

Or, of a column vector, you can insert an new axis along the second dimension.

[29]:
```
col_vector = x[:, np.newaxis]
col_vector.shape
```

[29]:
```
(7, 1)
```

You can also expand an array by inserting a new axis at a specified position with `np.expand_dims`.

For example:

```
[30]: z = np.expand_dims(x, axis=1)
      z.shape
```

```
[30]: (7, 1)
```

```
[31]: z = np.expand_dims(x, axis=0)
      z.shape
```

```
[31]: (1, 7)
```

1.12 Indexing an Slicing

You can index and slice NumPy in the same way you can slice in Python lists.

```
[32]: data = np.array([-1, 0, 1])
      data[1]
```

```
[32]: 0
```

```
[33]: data[0:2]
```

```
[33]: array([-1,  0])
```

```
[34]: data[1:]
```

```
[34]: array([0, 1])
```

```
[35]: data[-2:]
```

```
[35]: array([0, 1])
```

You may want to take a section of your array or a specific array elements to use in further analysis or additional operations. To do that, you'll need to subset, slice, and/or index your array.

If you want to select values from your array that fulfill certain conditions, it's straightforward with NumPy.

For example, if you start with this array:

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

You can easily print all of the values in the array that less than 5.

```
[37]: a[a < 5]
```

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

You can select elements that are divisible by 2.

```
[38]: div_by_2 = a[a % 2 == 0]
      div_by_2
```

```
[38]: array([ 2,  4,  6,  8, 10, 12])
```

Or you can select elemetns that satify two conditions using the `&` and `|` operators.

```
[39]: c = a[(a > 2) & (a < 10)]
      c
```

```
[39]: array([3, 4, 5, 6, 7, 8, 9])
```

You can also make use of the logical operator `&` and `|` in order to return boolean values that specify whether of not the values in an array fulfill a certain condition. This can be useful with arrays that contain names or other categorical values.

```
[40]: nine_up = (a > 9) | (a == 9)
      nine_up
```

```
[40]: array([[False, False, False, False],
             [False, False, False, False],
             [ True,  True,  True,  True]])
```

You can also use `np.nonzero()` to print the indices of elements that are, for example, less than 7.

```
[41]: b = np.nonzero(a < 7)
      b
```

```
[41]: (array([0, 0, 0, 0, 1, 1], dtype=int64),
       array([0, 1, 2, 3, 0, 1], dtype=int64))
```

In this example, a tuple of arrays was returned: one for each dimension. The first array represents the row indices where these values are found, and the second array represents the column indices where the values are found.

If you want to generate a list of coordinate where the elements exist, you can zip the arrays, iterate over the list of coordinates and print them. For example:

```
[42]: list_of_coordinates = list(zip(b[0], b[1]))
      for coord in list_of_coordinates:
          print(coord)
```

```
(0, 0)
(0, 1)
(0, 2)
(0, 3)
```

```
(1, 0)
(1, 1)
```

You can also use `np.nonzero()` to print the elements in an array that less than 7 with.

```
[43]: a[b]
```

```
[43]: array([1, 2, 3, 4, 5, 6])
```

If the element you're looking for doesn't exist in the array, then the returned array of indices will be empty. For example:

```
[44]: not_there = np.nonzero(a == 40)
      not_there
```

```
[44]: (array([], dtype=int64), array([], dtype=int64))
```

1.13 How to create an array from exsiting data

> This section covers `slicing and indexing`, `np.vstack()`, `np.hstack()`, `np.hsplit()`, `view()`, `copy()`.

You can easily create a new array from a section of an existing array.

Let's say you have this array:

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

You can create a new array from a section of your array any time by specifying where you want to slice your array.

```
[46]: b = a[2:7]
      b
```

```
[46]: array([3, 4, 5, 6, 7])
```

Here, you grabbed a section of your array from index position 2 through index position 7.

You can also stack two existing attays, both vertically and horizontally. Let's say you have to array, `a1` and `a2`:

```
[47]: a1 = np.array([[1, 1],
                     [2, 2]])
      a2 = np.array([[3, 3],
                     [4, 4]])
```

You can stack them vertically with `np.vstack()`.

```
[48]: np.vstack((a1, a2))
```

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

Or stack them horizontally with `np.hstack()`.

```
[49]: np.hstack((a1, a2))
```

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

You can split an array into several smaller arrays using `np.hsplit()`. You can specify either the number of equally shaped array to return or the columns *after* which the division should occur.

```
[50]: x = np.arange(1, 25).reshape(2, 12)
      x
```

```
[50]: array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12],
              [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]])
```

If you want to split this array into three equally shaped arrays, you would run:

```
[51]: np.hsplit(x, 3)
```

```
[51]: [array([[ 1,  2,  3,  4],
               [13, 14, 15, 16]]),
       array([[ 5,  6,  7,  8],
               [17, 18, 19, 20]]),
       array([[ 9, 10, 11, 12],
               [21, 22, 23, 24]])]
```

If you want to split your array after the third and fourth column, you'd run:

```
[52]: np.hsplit(x, (3, 4))
```

```
[52]: [array([[ 1,  2,  3],
               [13, 14, 15]]),
       array([[ 4],
               [16]]),
       array([[ 5,  6,  7,  8,  9, 10, 11, 12],
               [17, 18, 19, 20, 21, 22, 23, 24]])]
```

You can use `view` method to create a new array object that looks at the same data as your original array (a *shallow copy*).`view`

Views are an important NumPy concept! NumPy functions, as well as operations like indexing and slicing, will return views whenever possible. This saves memory and is faster (no copy of data has to be made). However it's important to be aware of this - modifying data in a view also modifies the original array!

Let's say you create this array:

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

Now we create an array by slicing and modify the first element of. This will the corresponding elements in as well!

```
[54]: b1 = a[0, :]
      b1
```

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

```
[55]: b1[0] = 99
      b1
```

```
[55]: array([99,  2,  3,  4])
```

```
[56]: a
```

```
[56]: array([[99,  2,  3,  4],
             [ 5,  6,  7,  8],
             [ 9, 10, 11, 12]])
```

Using the method will make a complete copy of the array and its data (a *deep copy*). To use this on your array, you could run `copy`.

```
[57]: b2 = a.copy()
      b2 is a
```

```
[57]: False
```

1.14 Basic array operations

This section covers addition, subtraction, multiplication, division and more.

Once you have created your arrays, you can start to work with them. Let's say, for example, that you've created two arrays, one called "data" and one called "ones".

You can add the arrays together with the plus sign.

[58]:
```
data = np.array([1, 2])
ones = np.ones(2, dtype=np.int32)
data + ones
```

[58]:
```
array([2, 3])
```

You can, of course, do more than just adddition!

[59]:
```
data - ones
```

[59]:
```
array([0, 1])
```

[60]:
```
data * ones
```

[60]:
```
array([1, 2])
```

[61]:
```
data / data
```

[61]:
```
array([1., 1.])
```

Basic operations are simple with NumPy. If you want to find sum of the element in the array, you'd use `sum` method. This works for 1D arrays, 2D arrays and arrays in higher dimensions.

[62]:
```
a = np.array([1, 2, 3, 4, 5])
a.sum()
```

[62]:
```
15
```

To add the rows and columns in a 2D array, you would specify the axis.

If you start with this array:

[63]:
```
b = np.array([[1, 1, 1], [2, 2, 2], [3, 3, 3]])
```

You can also sum over the axis of rows with:

[64]:
```
b.sum(axis=0)
```

[64]:
```
array([6, 6, 6])
```

[65]:
```
b.sum(axis=1)
```

[65]:
```
array([3, 6, 9])
```

1.15 Broadcasting

There are times when you might want to carry out an operation between an array and a single number (also called *an operation between a vector and a scalar*) or between arrays of two different sizes. For example, your array (we'll call it "data") might contain information about distance in miles but you want to convert the information to kilometers. You can perform this operation with:

```
[66]: data = np.array([1.0, 2.0])
      data * 1.6
```

```
[66]: array([1.6, 3.2])
```

NumPy understands that the multiplication should happen with each cell. That concept is called **broadcasting**. Broadcasting is a mechanism that allows NumPy to perform operations on arrays of different shapes. The dimensions of your array must be compatible, for example, when the dimensions of both arrays are equal or when one of them is 1. If the dimensions are not compatible, you will get a `ValueError`

1.16 More useful array operations

This section covers maximum, minimum, sum, mean, product, standard deviation, and more.

NumPy also perform aggregation functions. You can easily get the average, get the result of multiplying the elements together, to get the standard deviation, and more.

```
[67]: data.max()
```

```
[67]: 2.0
```

```
[68]: data.min()
```

```
[68]: 1.0
```

```
[69]: data.sum()
```

```
[69]: 3.0
```

Let's start with this array, called "a"

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

It's very commom to want to aggregate along a row or column. By default, every NumPy aggregation function will return the aggregate of the entire array. To find the sum or the minimum of the elements in your array, run:

```
[71]: a.sum()
```

```
[71]: 78
```

```
[72]: a.min()
```

```
[72]: 1
```

You can specify on which axis you want to aggregation function to be computed. For example, you can find the minimum value within each colomn by specifying `axis=0`

```
[73]: a.min(axis=0)
```

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

Or the minimum value within row by specifying `axis=1`

```
[74]: a.min(axis=1)
```

```
[74]: array([1, 5, 9])
```

1.17 Creating matrices

You can pass Python lists of lists to create a 2D array (or ”matrix”) to represent them in NumPy.

```
[75]: data = np.array([[1, 2], [3, 4], [5, 6]])
      data
```

```
[75]: array([[1, 2],
             [3, 4],
             [5, 6]])
```

Indexing and slicing operations are useful when you’re manipulating matrices.

```
[76]: data[0, 1]
```

```
[76]: 2
```

```
[77]: data[1:3]
```

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

```
[78]: data[0:2, 0]
```

```
[78]: array([1, 3])
```

You can aggregate matrices the same way you aggregated vectors

```
[79]: data.max()
```

```
[79]: 6
```

```
[80]: data.min()
```

```
[80]: 1
```

```
[81]: data.sum()
```

```
[81]: 21
```

You can aggregate all the values in a matrix and you can aggregate them across columns or rows using the parameter. To illustrate this point, let's look at a slightly modified dataset.

```
[82]: data = np.array([[1, 2], [5, 3], [4, 6]])
      data
```

```
[82]: array([[1, 2],
             [5, 3],
             [4, 6]])
```

```
[83]: data.max(axis=0)
```

```
[83]: array([5, 6])
```

```
[84]: data.max(axis=1)
```

```
[84]: array([2, 5, 6])
```

One you have created your matrices, you can add and multiply them using arithmetic operators if you have two matrices that are the same size.

```
[85]: data = np.array([[1, 2], [3, 4]])
      ones = np.ones((2, 2), dtype=int)
      data + ones
```

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

You can do these arithmetic operations on matrices of different sizes, but only if one matrix has only one column or one row. In this case, NumPy will use its broadcast rules for the operation.

```
[86]: data = np.array([[1, 2], [3, 4], [5, 6]])
      ones_row = np.ones((1, 2), dtype=int)
      data + ones_row
```

```
[86]: array([[2, 3],
             [4, 5],
             [6, 7]])
```

Be aware that when NumPy prints N-dimensional arrays, the last axis is looped over the fasted while the first acis is the slowest. For instance:

```
[87]: np.ones((4, 3, 2))
```

```
[87]: array([[[1., 1.],
              [1., 1.],
              [1., 1.]],

             [[1., 1.],
              [1., 1.],
              [1., 1.]],

             [[1., 1.],
              [1., 1.],
              [1., 1.]],

             [[1., 1.],
              [1., 1.],
              [1., 1.]]])
```

There are ofen instances where ew want NumPy to initialize the values of an array. NumPy offers functions like `np.ones()` and `np.zeros()` and `np.random.Generator` the class for random number generation for that. All you need to do is pass in the number of elements you want it to generate.

```
[88]: np.ones(3)
```

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

```
[89]: np.zeros(4)
```

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

```
[90]: rng = np.random.default_rng()
      rng.random(3)
```

```
[90]: array([0.64589492, 0.83950025, 0.0690902 ])
```

You can use `np.ones()`, `np.zeros()` and `random` to create a 2D array if you give them a tuple describing the dimensions of the matrix.

```
[91]: np.ones((3, 2))
```

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

```
[92]: np.zeros((3, 2))
```

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

```
[93]: rng.random((3, 2))
```

```
[93]: array([[0.59742971, 0.45004825],
             [0.80868541, 0.44230679],
             [0.36368876, 0.5909894 ]])
```

1.18 Generating random numbers

The use of random number generation is an important part of the configuration ans evaluation of many numerical and machine learning algorithms. Whether you need to randomly initialize weights in an artificial neural network, split data into random sets, or randomly shuffle your dataset, being able to generate randoms (actually, repeatable psuedo-random numbers) is essential.

With `Generator.integersendpoint=True`, you can generate random integers from low (remember that this is inclusive with NumPy) to high (exclusive). You can set to make the high number inclusive.

You can generate a 2x4 array of random integers between 0 and 4 with:

```
[94]: rng.integers(5, size=(2, 4))
```

```
[94]: array([[0, 4, 1, 0],
             [1, 3, 0, 3]], dtype=int64)
```

1.19 How to get unique items and counts?

This section cover `np.unique()`.

You can find the unique elements in an array easily with `np.unique()`.

For example, if you start with this array:

```
[95]: a = np.array([-1, -1, 3, 4, 2, 3, 4, 5, 100, 99])
```

you can use `np.unique()` to print the unique values in your array.

```
[96]: unique_values = np.unique(a)
      unique_values
```

```
[96]: array([ -1,   2,   3,   4,   5,  99, 100])
```

To get the indices of unique values in a numPy array (an array of first index positions of unique values in the array), just pass the argument `return_index=True` in as well as your array.

```
[97]: unique_values, indices_list = np.unique(a,return_index=True)
      indices_list
```

```
[97]: array([0, 4, 2, 3, 7, 9, 8], dtype=int64)
```

You can pass the argument `return_counts=True` in along with your array to get the frequency count of unique values in a NumPy array.

```
[98]: unique_values, occurence_count = np.unique(a, return_counts=True)
      occurence_count
```

```
[98]: array([2, 1, 2, 2, 1, 1, 1], dtype=int64)
```

This also work with 2d array! If you start with this array:

```
[99]: a_2d = np.array(
          [[1, 2, 3, 4],
           [5, 6, 7, 8],
           [9, 10, 11, 12]]
      )
```

You can find unique value with:

```
[100]: unique_values = np.unique(a_2d)
       unique_values
```

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

If the axis argument isn't passed, your 2D array will be flattened.

If you want to get the unique rows or columns, make sure to pass the argument `axis=`. To find the unique rows, pass `axis=0`, specify and for columns, pass `axis=1`.

```
[101]: unique_rows = np.unique(a_2d, axis=0)
       unique_rows
```

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

```
[102]: unique_columns = np.unique(a_2d, axis=1)
       unique_columns
```

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

1.20 Transposing and reshaping a matrix

This section covers `arr.reshape()`, `arr.transpose()`, `arr.T`.

It's common to need to transpose your matrices. NumPy arrays have the property `.T` that allows you to transpose a matrix.

You may need to switch the dimensions of a matrix. This can happen when, for example, you have a model that expects a certain input shape that is different from your dataset. This is where the method `reshape` can be useful. You simply need to pass in the new dimensions that you want for the matrix.

```
[103]: data = np.array([[1, 2], [3, 4], [5, 6]])
       data.reshape((2, 3))
```

```
[103]: array([[1, 2, 3],
              [4, 5, 6]])
```

```
[104]: data.reshape((3, 2))
```

```
[104]: array([[1, 2],
              [3, 4],
              [5, 6]])
```

You can also use `arr.transpose()` to reverse or change the axes of an array arrcoding to the values you specify.

If you start with this array:

```
[105]: arr = np.arange(6).reshape((2, 3))
       arr
```

```
[105]: array([[0, 1, 2],
              [3, 4, 5]])
```

You can transpose your array with `arr.transpose()`.

```
[106]: arr.transpose()
```

```
[106]: array([[0, 3],
              [1, 4],
              [2, 5]])
```

You can also use `arr.T`.

```
[107]: arr.T
```

```
[107]: array([[0, 3],
              [1, 4],
        [2, 5]])
```

1.21 How to reverse an array?

This section covers `np.flip()`.

NumPy's function allows you to flip, or reverse, the contents of an array along an axis. When using `np.flip(_`, specify the array you would like to reverse and the acis. If you don't specify the axis, NumPy will revesr the contents along all of the axes of your input array.

1.21.1 Reversing a 1D array

If you begin with a 1D array like this one:

[108]:
```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
```

You can reverse it with:

[109]:
```
reversed_arr = np.flip(arr)
reversed_arr
```

[109]:
```
array([8, 7, 6, 5, 4, 3, 2, 1])
```

1.21.2 Reversing a 2D array

A 2D array works much the same way.

If you start with this array:

[110]:
```
arr_2d = np.array(
    [[1, 2, 3, 4],
     [5, 6, 7, 8],
     [9, 10, 11, 12]]
)
```

You can reverse the content in all of the rows and all of the columns with:

[111]:
```
reversed_arr = np.flip(arr_2d)
reversed_arr
```

[111]:
```
array([[12, 11, 10,  9],
       [ 8,  7,  6,  5],
       [ 4,  3,  2,  1]])
```

You can easily reverse only the rows with:

```
[112]: reversed_arr_rows = np.flip(arr_2d, axis=0)
       reversed_arr_rows
```

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

Or reverse only the columns with:

```
[113]: reversed_arr_cols = np.flip(arr_2d, axis=1)
       reversed_arr_cols
```

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

You can also reverse the contents of only one column or row. For example, you can reverse the contents of the row at index position 1 (the second row):

```
[114]: arr_2d[1] = np.flip(arr_2d[1])
       arr_2d
```

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

You can also reverse the column at index position 1 (the second column):

```
[115]: arr_2d[:, 1] = np.flip(arr_2d[:, 1])
       arr_2d
```

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

1.22 Reshaping and flattening multidimensional arrays

This section covers `.flatten()` and `.ravel()`.

There are two popular ways to flatten an array: `flatten` and `ravel`. The primary difference between the two is that the new array created using `ravel` is actually a reference to the parent array. This means that any changes to the new array will affect the parent array as well. Since does not create a copy, it's memory efficent.

If you start with this array:

[116]:
```
x = np.array(
    [[1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]]
)
```

You can use `flatten` to flatten your array into 1D array.

[117]:
```
x.flatten()
```

[117]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

When you use `flatten`, change to your new array won't change the parent array.

[118]:
```
a1 = x.flatten()
a1[0] = 99
x
```

[118]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

But when you use `ravel`, the change you make to the new array will affect the parent array.

[119]:
```
a2 = x.ravel()
a2[0] = 99
x
```

[119]:
```
array([[99,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

1.23 How to access the docstring for more information

This section covers `help()`, `?` and `??`.

When it comes to the data science ecosystem, Python and NumPy are built with the user in mind. One of the best examples of this is the built-in `help()` access to documentation. Every object contains the reference to a string, which is known as the **docstring**. In most cases, this docstring contains a quick and concise summary of the object and how to use it. Python has a built-in function that can help you access this information. This means that nearly any time you need more information, you can use `help()` to quickly find the information that you need.

[120]:
```
help(max)
```

```
Help on built-in function max in module builtins:
```

```
max(…)
    max(iterable, *[, default=obj, key=func]) -> value
    max(arg1, arg2, *args, *[, key=func]) -> value

    With a single iterable argument, return its biggest item. The
    default keyword-only argument specifies an object to return if
    the provided iterable is empty.
    With two or more arguments, return the largest argument.
```

Because access to additional information is so useful, IPython uses the character as a shorthand for accessing this documentation along with other relevant information. IPython is a command shell for interactive computing in multiple languages.

[121]:
```
max?
```

```
Docstring:
max(iterable, *[, default=obj, key=func]) -> value
max(arg1, arg2, *args, *[, key=func]) -> value

With a single iterable argument, return its biggest item. The
default keyword-only argument specifies an object to return if
the provided iterable is empty.
With two or more arguments, return the largest argument.
Type:      builtin_function_or_method
```

You can even use this notation for object methods and objects themselves.

Let's say you create this array:

[122]:
```
a = np.array([1, 2, 3, 4, 5, 6])
```

Then you can obtain a lot of useful information (first details about itself, followed by the docstring of of which is an instance):

[123]:
```
a?
```

```
Type:            ndarray
String form:     [1 2 3 4 5 6]
Length:          6
File:
c:\users\user\appdata\local\programs\python\python39\lib\site-
packages\numpy\__init__.py
Docstring:       <no docstring>
Class docstring:
ndarray(shape, dtype=float, buffer=None, offset=0,
        strides=None, order=None)

An array object represents a multidimensional, homogeneous array
of fixed-size items.  An associated data-type object describes the
```

```
format of each element in the array (its byte-order, how many bytes it
occupies in memory, whether it is an integer, a floating point number,
or something else, etc.)

Arrays should be constructed using `array`, `zeros` or `empty` (refer
to the See Also section below).  The parameters given here refer to
a low-level method (`ndarray(…)`) for instantiating an array.

For more information, refer to the `numpy` module and examine the
methods and attributes of an array.

Parameters
----------
(for the __new__ method; see Notes below)

shape : tuple of ints
    Shape of created array.
dtype : data-type, optional
    Any object that can be interpreted as a numpy data type.
buffer : object exposing buffer interface, optional
    Used to fill the array with data.
offset : int, optional
    Offset of array data in buffer.
strides : tuple of ints, optional
    Strides of data in memory.
order : {'C', 'F'}, optional
    Row-major (C-style) or column-major (Fortran-style) order.

Attributes
----------
T : ndarray
    Transpose of the array.
data : buffer
    The array's elements, in memory.
dtype : dtype object
    Describes the format of the elements in the array.
flags : dict
    Dictionary containing information related to memory use, e.g.,
    'C_CONTIGUOUS', 'OWNDATA', 'WRITEABLE', etc.
flat : numpy.flatiter object
    Flattened version of the array as an iterator.  The iterator
    allows assignments, e.g., ``x.flat = 3`` (See `ndarray.flat` for
    assignment examples; TODO).
imag : ndarray
    Imaginary part of the array.
real : ndarray
    Real part of the array.
size : int
```

```
    Number of elements in the array.
itemsize : int
    The memory use of each array element in bytes.
nbytes : int
    The total number of bytes required to store the array data,
    i.e., ``itemsize * size``.
ndim : int
    The array's number of dimensions.
shape : tuple of ints
    Shape of the array.
strides : tuple of ints
    The step-size required to move from one element to the next in
    memory. For example, a contiguous ``(3, 4)`` array of type
    ``int16`` in C-order has strides ``(8, 2)``.  This implies that
    to move from element to element in memory requires jumps of 2 bytes.
    To move from row-to-row, one needs to jump 8 bytes at a time
    (``2 * 4``).
ctypes : ctypes object
    Class containing properties of the array needed for interaction
    with ctypes.
base : ndarray
    If the array is a view into another array, that array is its `base`
    (unless that array is also a view).  The `base` array is where the
    array data is actually stored.

See Also
--------
array : Construct an array.
zeros : Create an array, each element of which is zero.
empty : Create an array, but leave its allocated memory unchanged (i.e.,
        it contains "garbage").
dtype : Create a data-type.

Notes
-----
There are two modes of creating an array using ``__new__``:

1. If `buffer` is None, then only `shape`, `dtype`, and `order`
   are used.
2. If `buffer` is an object exposing the buffer interface, then
   all keywords are interpreted.

No ``__init__`` method is needed because the array is fully initialized
after the ``__new__`` method.

Examples
--------
These examples illustrate the low-level `ndarray` constructor.  Refer
```

```
to the `See Also` section above for easier ways of constructing an
ndarray.

First mode, `buffer` is None:

>>> np.ndarray(shape=(2,2), dtype=float, order='F')
array([[0.0e+000, 0.0e+000], # random
       [     nan, 2.5e-323]])

Second mode:

>>> np.ndarray((2,), buffer=np.array([1,2,3]),
…              offset=np.int_().itemsize,
…              dtype=int) # offset = 1*itemsize, i.e. skip first element
array([2, 3])
```

This also works for functions and other objects that **you** create. Just remember to include a docstring with your function using a string literal (or arounf your documentation).

[124]:
```python
def double(a):
    """Return a * 2"""
    return a * 2
```

You can obtain information about the function.

[125]:
```
double?
```

```
Signature:
double(a)
Docstring: Return a * 2
File:
c:\users\user\appdata\local\temp\ipykernel_1792\831218930.py
Type:      function
```

You can reach another level of information by reading the source codde of the object you're interested in. Using a double question mark `??` allows you to access the source code.

[126]:
```
double??
```

```
Signature:
double(a)
Source:
def
double(a):

    """Return a * 2"""

    return a *
2
File:
```

```
c:\users\user\appdata\local\temp\ipykernel_1792\831218930.py
Type:      function
```

If the object in question is compiled in a language other than Python, using ??? will return the same information as . You'll find this with a lot of built-in objects and types, for example:

```
[127]: len?
```

```
Signature: len(obj,
/)
Docstring: Return the number of items in a container.
Type:      builtin_function_or_method
```

and:

```
[128]: len??
```

```
Signature: len(obj,
/)
Docstring: Return the number of items in a container.
Type:      builtin_function_or_method
```

have the same output because they were compiled in a programming language other than Python.

1.24 Working with mathmatical formulas

The ease of implementing mathematical formulas that work on arrays is one of the things that make NumPy so widely used in the scientific Python community.

For example, this is the mean square error formula (a central formula used in supervised machine learning models that deal with regression):

`MeanSquareError = 1/n * sum_1^n(Y_{prediction_i} - Y_i)^2`

Implementing this formula is simple and straightforward in NumPy:

```
error = (1/n) * np.sum(np.square(predictions - labels))
```

1.25 How to save and load Numpy objects

1.26 Importing and exporting a CSV

1.27 Plotting arrays with matplotlib

If you need to generate a plot for your values, it's very simple with Matplotlib.

For example, you may have an array like this one:

```
[129]: x = np.array(np.random.randint(1, 20, 15))
```

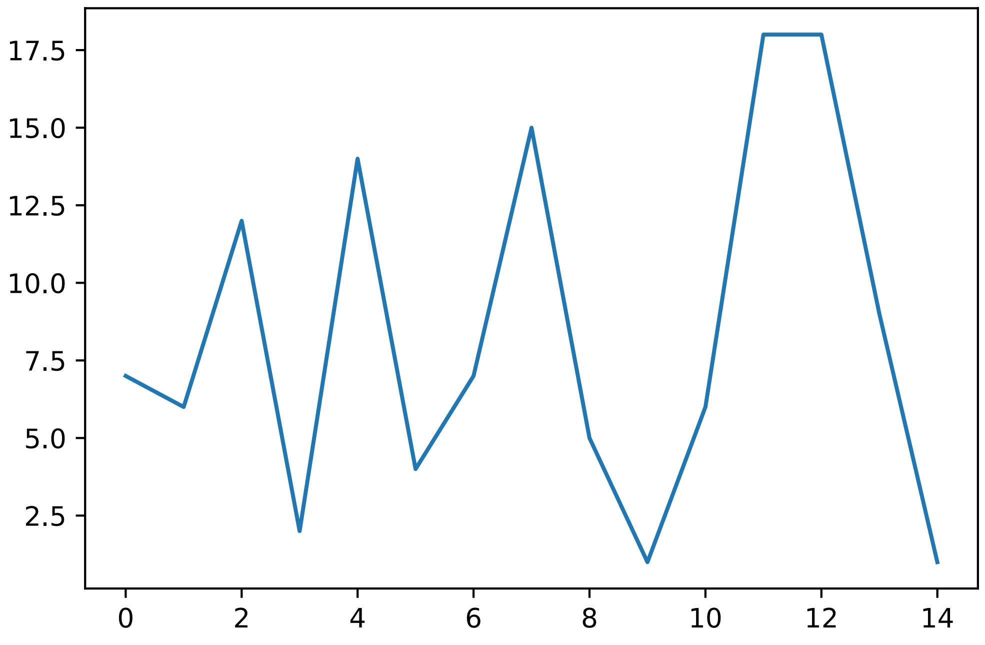
If you already have Matplotlib installed, you can import it with:

```
[130]: import matplotlib.pyplot as plt
```

All you need to do to plot your values is run:

```
[131]: plt.plot(x)
```

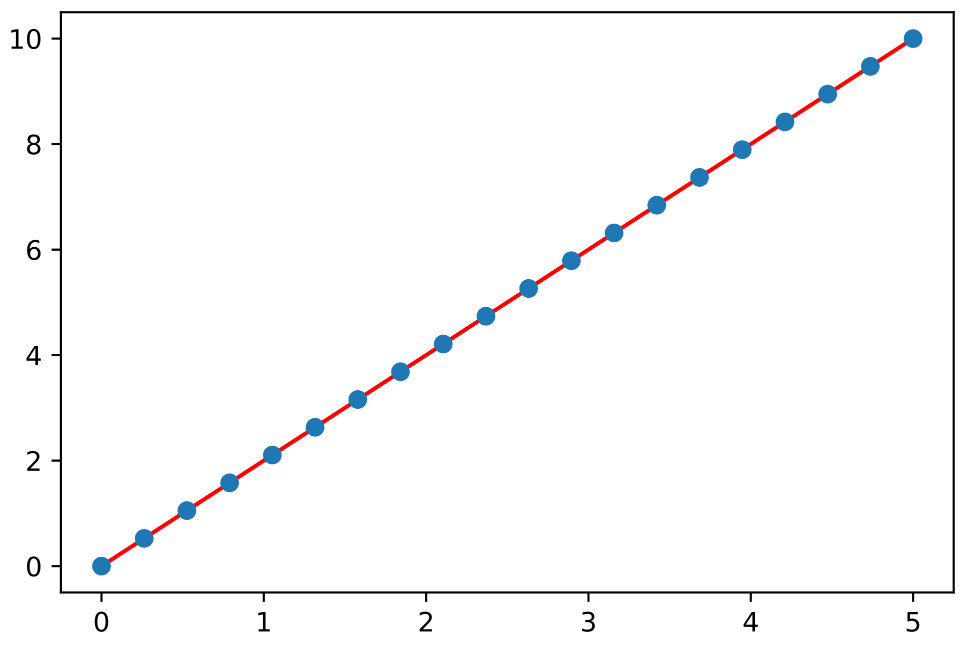
```
[131]: [<matplotlib.lines.Line2D at 0x1d96afcafd0>]
```



For example, you can plot a 1D array like this:

```
[132]: x = np.linspace(0, 5, 20)
       y = np.linspace(0, 10, 20)
       plt.plot(x, y, 'red')
       plt.plot(x, y, 'o')
```

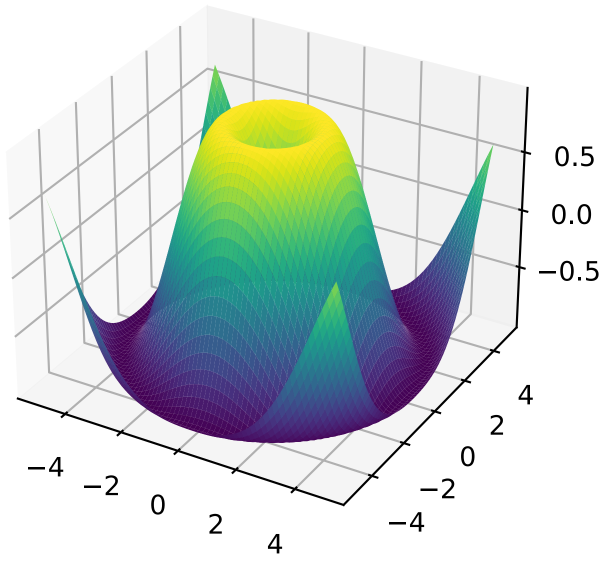
```
[132]: [<matplotlib.lines.Line2D at 0x1d96b101c70>]
```



With Matplotlib, you have access to an enormous number of visualization options

```
[133]: fig = plt.figure()
       ax = fig.add_subplot(projection='3d')
       X = np.arange(-5, 5, 0.15)
       Y = np.arange(-5, 5, 0.15)
       X, Y = np.meshgrid(X, Y)
       R = np.sqrt(X**2 + Y**2)
       Z = np.sin(R)

       ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='viridis')
```

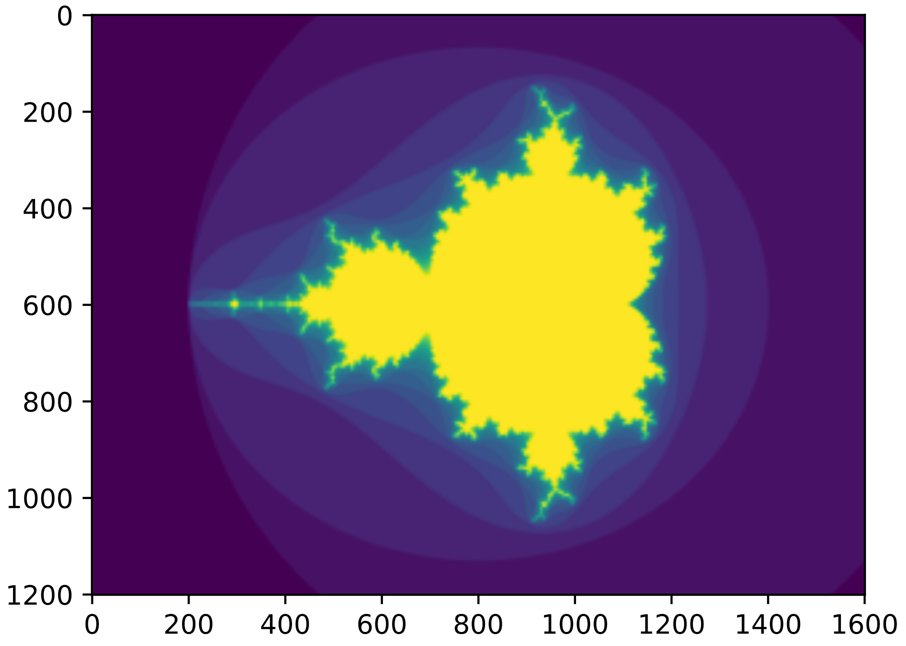
```
[133]: <mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x1d96b174eb0>
```



You can look at the following example to see how to use boolean indexing to generate an image of the Mandelbrot set:

[134]:
```
import numpy as np
import matplotlib.pyplot as plt

def mandelbrot(h, w, maxit=20, r=2):
    """Return an image of the Mandelbrot fractal of size (h, w)."""
    x = np.linspace(-2.5, 1.5, 4*h + 1)
    y = np.linspace(-1.5, 1.5, 3*w + 1)
    A, B = np.meshgrid(x, y)
    C = A + B*1j
    z = np.zeros_like(C)
    divtime = maxit + np.zeros(z.shape, dtype=int)

    for i in range(maxit):
        z = z**2 + C
        diverge = abs(z) > r
        div_now = diverge & (divtime == maxit)
        divtime[div_now] = i
        z[diverge] = r

    return divtime

plt.clf()
plt.imshow(mandelbrot(400, 400))
```

```
[134]: <matplotlib.image.AxesImage at 0x1d96b3598b0>
```


0
200
400
600
800
1000
1200
0
200
400
600
800
1000
1200
1400
1600