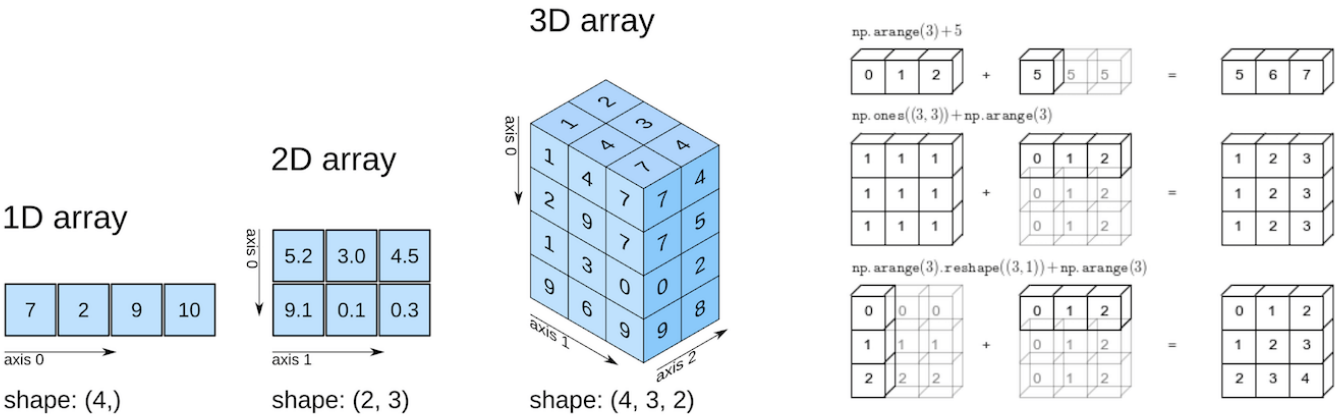
Numerical Computing with Python and Numpy



This tutorial series is a beginner-friendly introduction to programming and data analysis using the Python programming language. These tutorials take a practical and coding-focused approach. The best way to learn the material is to execute the code and experiment with it yourself.

This tutorial covers the following topics:

- Working with numerical data in Python
- Going from Python lists to Numpy arrays
- Multi-dimensional Numpy arrays and their benefits
- Array operations, broadcasting, indexing, and slicing
- Working with CSV data files using Numpy

How to run the code

This tutorial is an executable Jupyter notebook hosted on Jovian. You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on Google Colab or Kaggle to use these platforms.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up Python, download the notebook and install the required libraries. We recommend using the Conda distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

> **Jupyter Notebooks**: This tutorial is a Jupyter notebook - a document made of *cells*. Each cell can contain code written in Python or explanations in plain English. You can execute code cells and view the results, e.g., numbers, messages, graphs, tables, files, etc., instantly within the notebook. Jupyter is a powerful platform for experimentation and analysis. Don't be afraid to mess around with the code & break things -

> you'll learn a lot by encountering and fixing errors. You can use the "Kernel > Restart & Clear Output" menu option to clear all outputs and start again from the top.

Working with numerical data

The "data" in *Data Analysis* typically refers to numerical data, e.g., stock prices, sales figures, sensor measurements, sports scores, database tables, etc. The Numpy library provides specialized data structures, functions, and other tools for numerical computing in Python. Let's work through an example to see why & how to use Numpy for working with numerical data.

> Suppose we want to use climate data like the temperature, rainfall, and humidity to determine if a region is well suited for growing apples. A simple approach for doing this would be to formulate the relationship between the annual yield of apples (tons per hectare) and the climatic conditions like the average temperature (in degrees Fahrenheit), rainfall (in millimeters) & average relative humidity (in percentage) as a linear equation.
>
> `yield_of_apples = w1 * temperature + w2 * rainfall + w3 * humidity`

We're expressing the yield of apples as a weighted sum of the temperature, rainfall, and humidity. This equation is an approximation since the actual relationship may not necessarily be linear, and there may be other factors involved. But a simple linear model like this often works well in practice.

Based on some statical analysis of historical data, we might come up with reasonable values for the weights `w1`, `w2`, and `w3`. Here's an example set of values:

```
w1, w2, w3 = 0.3, 0.2, 0.5
```

Given some climate data for a region, we can now predict the yield of apples. Here's some sample data:

Region	Temp. (F)	Rainfall (mm)	Humidity (%)
Kanto	73	67	43
Johto	91	88	64
Hoenn	87	134	58
Sinnoh	102	43	37
Unova	69	96	70

To begin, we can define some variables to record climate data for a region.

```
kanto_temp = 73
kanto_rainfall = 67
kanto_humidity = 43
```

We can now substitute these variables into the linear equation to predict the yield of apples.

```
kanto_yield_apples = kanto_temp * w1 + kanto_rainfall * w2 + kanto_humidity * w3
kanto_yield_apples
```

```
56.8
```

```
print("The expected yield of apples in Kanto region is {} tons per hectare.".format(kan
```

```
The expected yield of apples in Kanto region is 56.8 tons per hectare.
```

To make it slightly easier to perform the above computation for multiple regions, we can represent the climate data for each region as a vector, i.e., a list of numbers.

```
kanto = [73, 67, 43]
johto = [91, 88, 64]
hoenn = [87, 134, 58]
sinnoh = [102, 43, 37]
unova = [69, 96, 70]
```

The three numbers in each vector represent the temperature, rainfall, and humidity data, respectively.

We can also represent the set of weights used in the formula as a vector.

```
weights = [w1, w2, w3]
```

We can now write a function `crop_yield` to calcuate the yield of apples (or any other crop) given the climate data and the respective weights.

```
def crop_yield(region, weights):
    result = 0
    for x, w in zip(region, weights):
        result += x * w
    return result
```

```
crop_yield(kanto, weights)
```

```
56.8
```

```
crop_yield(johto, weights)
```

```
76.9
```

```
crop_yield(unova, weights)
```

```
74.9
```

Details on Zip function →

```
for x, w in zip(kanto, weights):
    print(x)
    print(w)

print(zip(kanto, weights))
print(tuple(zip(kanto, weights)))
a = ("John", "Charles")
b = ("Jenny", "Christy", "Monica")

print(tuple(zip(a,b)))
```

```
73
0.3
67
0.2
43
0.5
<zip object at 0x000001B80C70DEC0>
((73, 0.3), (67, 0.2), (43, 0.5))
(('John', 'Jenny'), ('Charles', 'Christy'))
```

zip creates pair of matching index in list

directly printing zip prints the reference so to print the value we need to pass it to tuple.

zip creates pair of matching index, for missing index no pair created eg:- no pair for Monica

Going from Python lists to Numpy arrays

The calculation performed by the `crop_yield` (element-wise multiplication of two vectors and taking a sum of the results) is also called the *dot product*. Learn more about dot product here: https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces/dot-cross-products/v/vector-dot-product-and-vector-length .

The Numpy library provides a built-in function to compute the dot product of two vectors. However, we must first convert the lists into Numpy arrays.

Let's install the Numpy library using the `pip` package manager.

```
!pip install numpy --upgrade --quiet
```

Next, let's import the `numpy` module. It's common practice to import numpy with the alias `np`.

```
import numpy as np
```

// we are simply creating alias alongwith import

We can now use the `np.array` function to create Numpy arrays.

```
kanto = np.array([73, 67, 43])
```

```
kanto
```

```
array([73, 67, 43])
```

```
weights = np.array([w1, w2, w3])
```

```
weights
```

```
array([0.3, 0.2, 0.5])
```

Numpy arrays have the type `ndarray`.

```
type(kanto)
```

```
numpy.ndarray
```

// ndarray are not normal python list rather they are special numpy arrays

```
type(weights)
```

```
numpy.ndarray
```

Just like lists, Numpy arrays support the indexing notation `[]`.

```
weights[0]
```

```
0.3
```

```
kanto[2]
```

```
43
```

Operating on Numpy arrays

We can now compute the dot product of the two vectors using the `np.dot` function.

```
np.dot(kanto, weights)
```

```
56.8
```

We can achieve the same result with low-level operations supported by Numpy arrays: performing an element-wise multiplication and calculating the resulting numbers' sum.

```
(kanto * weights).sum()
```

```
56.8
```

The `*` operator performs an element-wise multiplication of two arrays if they have the same size. The `sum` method calculates the sum of numbers in an array.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
```

```
arr1 * arr2
```

```
array([ 4, 10, 18])
```

```
arr2.sum()
```

```
15
```

Benefits of using Numpy arrays

Numpy arrays offer the following benefits over Python lists for operating on numerical data:

- **Ease of use**: You can write small, concise, and intuitive mathematical expressions like `(kanto * weights).sum()` rather than using loops & custom functions like `crop_yield`.
- **Performance**: Numpy operations and functions are implemented internally in C++, which makes them much faster than using Python statements & loops that are interpreted at runtime

Here's a comparison of dot products performed using Python loops vs. Numpy arrays on two vectors with a million elements each.

```
# Python lists
arr1 = list(range(1000000))
arr2 = list(range(1000000, 2000000))

# Numpy arrays
arr1_np = np.array(arr1)
arr2_np = np.array(arr2)
```

```
%%time
result = 0
for x1, x2 in zip(arr1, arr2):
    result += x1*x2
result
```

```
CPU times: user 151 ms, sys: 1.35 ms, total: 153 ms
Wall time: 152 ms
```

```
833332333333500000
```

```
%%time
np.dot(arr1_np, arr2_np)
```

```
CPU times: user 1.96 ms, sys: 751 µs, total: 2.71 ms
Wall time: 1.6 ms
```

```
833332333333500000
```

As you can see, using `np.dot` is 100 times faster than using a `for` loop. This makes Numpy especially useful while working with really large datasets with tens of thousands or millions of data points.

Let's save our work before continuing.

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
[jovian] Updating notebook "aakashns/python-numerical-computing-with-numpy" on https://jovian.ai/
[jovian] Uploading notebook..
[jovian] Capturing environment..
[jovian] Committed successfully! https://jovian.ai/aakashns/python-numerical-computing-with-numpy
```

```
'https://jovian.ai/aakashns/python-numerical-computing-with-numpy'
```

Multi-dimensional Numpy arrays

We can now go one step further and represent the climate data for all the regions using a single 2-dimensional Numpy array.

```
climate_data = np.array([[73, 67, 43],
                         [91, 88, 64],
                         [87, 134, 58],
                         [102, 43, 37],
                         [69, 96, 70]])
```

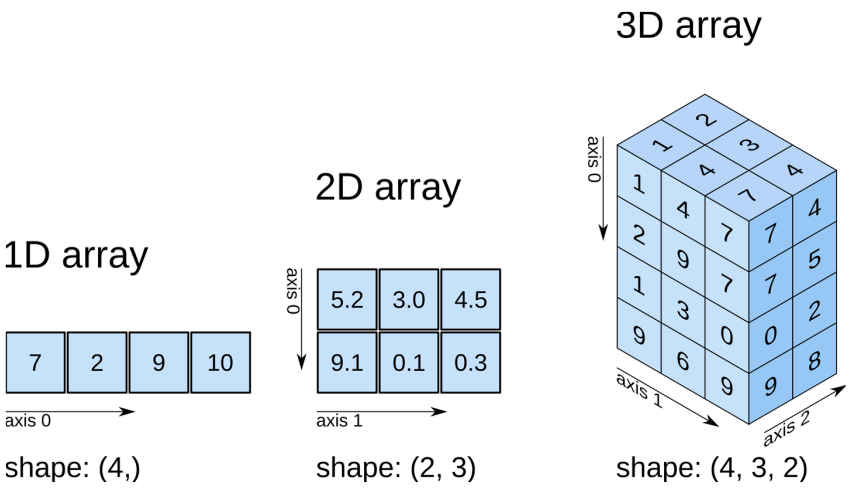
```
climate_data
```

```
array([[ 73,  67,  43],
       [ 91,  88,  64],
       [ 87, 134,  58],
```

```
       [102,  43,  37],
       [ 69,  96,  70]])
```

If you've taken a linear algebra class in high school, you may recognize the above 2-d array as a matrix with five rows and three columns. Each row represents one region, and the columns represent temperature, rainfall, and humidity, respectively.

Numpy arrays can have any number of dimensions and different lengths along each dimension. We can inspect the length along each dimension using the `.shape` property of an array.




```
# 2D array (matrix)
climate_data.shape
```

```
(5, 3)
```

```
weights
```

```
array([0.3, 0.2, 0.5])
```

```
# 1D array (vector)
weights.shape
```

```
(3,)
```

```
# 3D array
arr3 = np.array([
    [[11, 12, 13],
     [13, 14, 15]],
    [[15, 16, 17],
     [17, 18, 19.5]]])
```

think of it like each square bracket is one dimension

```
arr3.shape
```

```
(2, 2, 3)
```

All the elements in a numpy array have the same data type. You can check the data type of an array using the `.dtype` property.

```
weights.dtype
```

```
dtype('float64')
```

```
climate_data.dtype
```

```
dtype('int64')
```

If an array contains even a single floating point number, all the other elements are also converted to floats.

```
arr3.dtype
```

```
dtype('float64')
```

We can now compute the predicted yields of apples in all the regions, using a single matrix multiplication between `climate_data` (a 5x3 matrix) and `weights` (a vector of length 3). Here's what it looks like visually:

$$\begin{bmatrix} 73 & 67 & 43 \\ 91 & 88 & 64 \\ \vdots & \vdots & \vdots \\ 69 & 96 & 70 \end{bmatrix} \times \begin{bmatrix} w_{11} \\ w_{12} \\ w_{13} \end{bmatrix}$$

You can learn about matrices and matrix multiplication by watching the first 3-4 videos of this playlist: https://www.youtube.com/watch?v=xyAuNHPsq-g&list=PLFD0EB975BA0CC1E0&index=1 .

We can use the `np.matmul` function or the `@` operator to perform matrix multiplication.

```
np.matmul(climate_data, weights)
```

```
array([56.8, 76.9, 81.9, 57.7, 74.9])
```

```
climate_data @ weights
```

```
array([56.8, 76.9, 81.9, 57.7, 74.9])
```

Working with CSV data files

Numpy also provides helper functions reading from & writing to files. Let's download a file `climate.txt`, which contains 10,000 climate measurements (temperature, rainfall & humidity) in the following format:

```
temperature,rainfall,humidity
25.00,76.00,99.00
39.00,65.00,70.00
59.00,45.00,77.00
84.00,63.00,38.00
66.00,50.00,52.00
41.00,94.00,77.00
91.00,57.00,96.00
```

```
49.00,96.00,99.00
67.00,20.00,28.00
...
```

This format of storing data is known as *comma-separated values* or CSV.

> **CSVs**: A comma-separated values (CSV) file is a delimited text file that uses a comma to separate values. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. A CSV file typically stores tabular data (numbers and text) in plain text, in which case each line will have the same number of fields. (Wikipedia)

To read this file into a numpy array, we can use the `genfromtxt` function.

```
import urllib.request

urllib.request.urlretrieve(
    'https://gist.github.com/BirajCoder/a4ffcb76fd6fb221d76ac2ee2b8584e9/raw/4054f90adf
    'climate.txt')
```

```
('climate.txt', <http.client.HTTPMessage at 0x7ffa9c1e2668>)
```

```
climate_data = np.genfromtxt('climate.txt', delimiter=',', skip_header=1)
```

```
climate_data
```

```
array([[25., 76., 99.],
       [39., 65., 70.],
       [59., 45., 77.],
       ...,
       [99., 62., 58.],
       [70., 71., 91.],
       [92., 39., 76.]])
```

```
climate_data.shape
```

```
(10000, 3)
```

We can now perform a matrix multiplication using the `@` operator to predict the yield of apples for the entire dataset using a given set of weights.

```
weights = np.array([0.3, 0.2, 0.5])
```

```
yields = climate_data @ weights
```

```
yields
```

```
array([72.2, 59.7, 65.2, ..., 71.1, 80.7, 73.4])
```

```
yields.shape
```

```
(10000,)
```

Let's add the `yields` to `climate_data` as a fourth column using the `np.concatenate` function.

```
climate_results = np.concatenate((climate_data, yields.reshape(10000, 1)), axis=1)
```

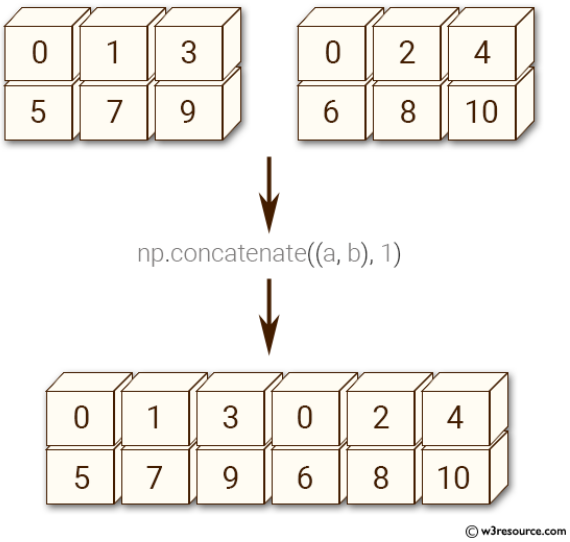
```
climate_results
```

```
array([[25. , 76. , 99. , 72.2],
       [39. , 65. , 70. , 59.7],
       [59. , 45. , 77. , 65.2],
       ...,
       [99. , 62. , 58. , 71.1],
       [70. , 71. , 91. , 80.7],
       [92. , 39. , 76. , 73.4]])
```

There are a couple of subtleties here:

- Since we wish to add new columns, we pass the argument `axis=1` to `np.concatenate`. The `axis` argument specifies the dimension for concatenation. (dimension starts from zero, dimensions are number of square brackets to reach your data)
- The arrays should have the same number of dimensions, and the same length along each except the dimension used for concatenation. We use the `np.reshape` function to change the shape of `yields` from `(10000,)` to `(10000,1)`.

Here's a visual explanation of `np.concatenate` along `axis=1` (can you guess what `axis=0` results in?):



The best way to understand what a Numpy function does is to experiment with it and read the documentation to learn about its arguments & return values. Use the cells below to experiment with `np.concatenate` and `np.reshape`.

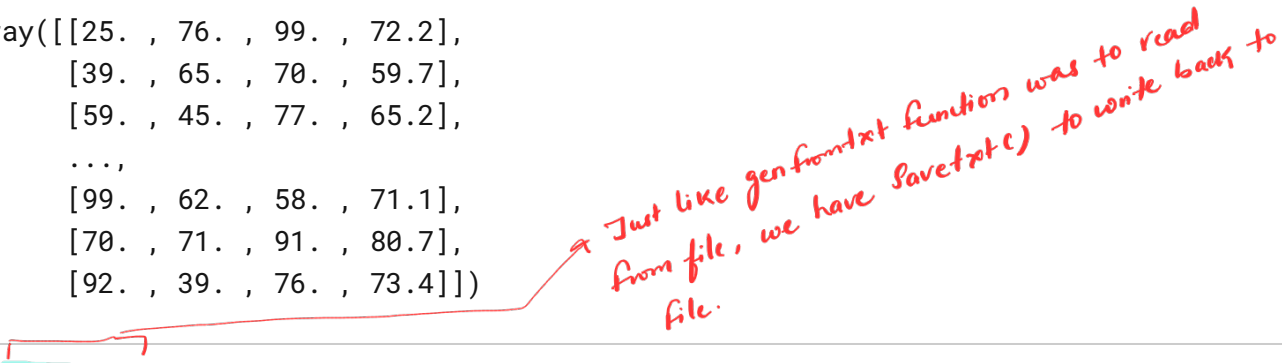
```

```

Let's write the final results from our computation above back to a file using the `np.savetxt` function.

```
climate_results
```

```
array([[25. , 76. , 99. , 72.2],
       [39. , 65. , 70. , 59.7],
       [59. , 45. , 77. , 65.2],
       ...,
       [99. , 62. , 58. , 71.1],
       [70. , 71. , 91. , 80.7],
       [92. , 39. , 76. , 73.4]])
```




```
np.savetxt('climate_results.txt',
           climate_results,
           fmt='%.2f',
           delimiter=',',
           header='temperature,rainfall,humidity,yeild_apples',
           comments='')
```

The results are written back in the CSV format to the file `climate_results.txt` .

```
temperature,rainfall,humidity,yeild_apples
25.00,76.00,99.00,72.20
39.00,65.00,70.00,59.70
59.00,45.00,77.00,65.20
84.00,63.00,38.00,56.80
...
```

Numpy provides hundreds of functions for performing operations on arrays. Here are some commonly used functions:

- Mathematics: `np.sum`, `np.exp`, `np.round`, arithemtic operators
- Array manipulation: `np.reshape`, `np.stack`, `np.concatenate`, `np.split`
- Linear Algebra: `np.matmul`, `np.dot`, `np.transpose`, `np.eigvals`
- Statistics: `np.mean`, `np.median`, `np.std`, `np.max`

> **How to find the function you need?** The easiest way to find the right function for a specific operation or use-case is to do a web search. For instance, searching for "How to join numpy arrays" leads to this tutorial on array concatenation.

You can find a full list of array functions here: https://numpy.org/doc/stable/reference/routines.html

Save and upload your notebook

Whether you're running this Jupyter notebook online or on your computer, it's essential to save your work from time to time. You can continue working on a saved notebook later or share it with friends and colleagues to let them execute your code. [Jovian](https://jovian.ai) offers an easy way of saving and sharing your Jupyter notebooks online.

```
# Install the library
!pip install jovian --upgrade --quiet
```

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
[jovian] Updating notebook "aakashns/python-numerical-computing-with-numpy" on https://jovian.ai/
[jovian] Uploading notebook..
[jovian] Capturing environment..
[jovian] Committed successfully! https://jovian.ai/aakashns/python-numerical-computing-with-numpy
```

```
'https://jovian.ai/aakashns/python-numerical-computing-with-numpy'
```

The first time you run `jovian.commit`, you'll be asked to provide an API Key to securely upload the notebook to your Jovian account. You can get the API key from your [Jovian profile page](https://jovian.ai) after logging in / signing up.

`jovian.commit` uploads the notebook to your Jovian account, captures the Python environment, and creates a shareable link for your notebook, as shown above. You can use this link to share your work and let anyone (including you) run your notebooks and reproduce your work.

Arithmetic operations, broadcasting and comparison

Numpy arrays support arithmetic operators like `+`, `-`, `*`, etc. You can perform an arithmetic operation with a single number (also called scalar) or with another array of the same shape. Operators make it easy to write mathematical expressions with multi-dimensional arrays.

```
arr2 = np.array([[1, 2, 3, 4],
                 [5, 6, 7, 8],
                 [9, 1, 2, 3]])
```

```
arr3 = np.array([[11, 12, 13, 14],
                 [15, 16, 17, 18],
                 [19, 11, 12, 13]])
```

```
# Adding a scalar
arr2 + 3
```

```
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12,  4,  5,  6]])
```

```
# Element-wise subtraction
arr3 - arr2
```

```
array([[10, 10, 10, 10],
       [10, 10, 10, 10],
       [10, 10, 10, 10]])
```

```
# Division by scalar
arr2 / 2
```

```
array([[0.5, 1. , 1.5, 2. ],
       [2.5, 3. , 3.5, 4. ],
       [4.5, 0.5, 1. , 1.5]])
```

```
# Element-wise multiplication
arr2 * arr3
```

```
array([[ 11,  24,  39,  56],
       [ 75,  96, 119, 144],
       [171,  11,  24,  39]])
```

```
# Modulus with scalar
arr2 % 4
```

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

Array Broadcasting

Numpy arrays also support *broadcasting*, allowing arithmetic operations between two arrays with different numbers of dimensions but compatible shapes. Let's look at an example to see how it works.

```
arr2 = np.array([[1, 2, 3, 4],
                 [5, 6, 7, 8],
                 [9, 1, 2, 3]])
```

```
arr2.shape
```

```
(3, 4)
```

```
arr4 = np.array([4, 5, 6, 7])
```

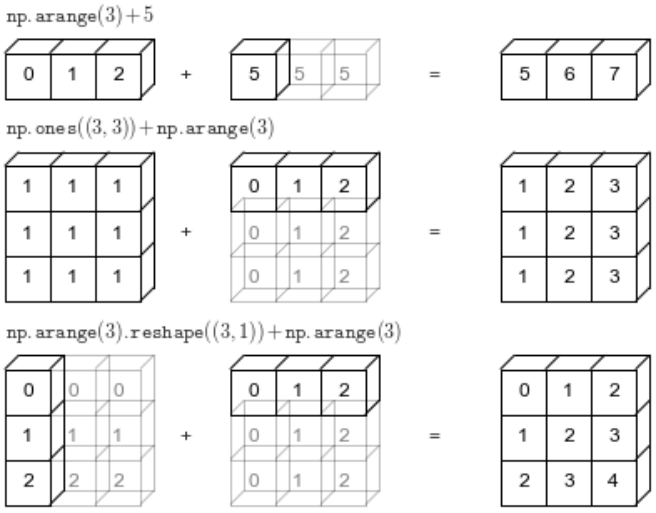
```
arr4.shape
```

```
(4,)
```

```
arr2 + arr4
```

```
array([[ 5,  7,  9, 11],
       [ 9, 11, 13, 15],
       [13,  6,  8, 10]])
```

When the expression `arr2 + arr4` is evaluated, `arr4` (which has the shape `(4,)`) is replicated three times to match the shape `(3, 4)` of `arr2`. Numpy performs the replication without actually creating three copies of the smaller dimension array, thus improving performance and using lower memory.



Broadcasting only works if one of the arrays can be replicated to match the other array's shape.

```
arr5 = np.array([7, 8])
```

```
arr5.shape
```

```
(2,)
```

```
arr2 + arr5
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-72-c22e92053c39> in <module>
----> 1 arr2 + arr5

ValueError: operands could not be broadcast together with shapes (3,4) (2,)
```

In the above example, even if `arr5` is replicated three times, it will not match the shape of `arr2`. Hence `arr2 + arr5` cannot be evaluated successfully. Learn more about broadcasting here: https://numpy.org/doc/stable/user/basics.broadcasting.html .

Array Comparison

Numpy arrays also support comparison operations like `==`, `!=`, `>` etc. The result is an array of booleans.

```
arr1 = np.array([[1, 2, 3], [3, 4, 5]])
arr2 = np.array([[2, 2, 3], [1, 2, 5]])
```

```
arr1 == arr2
```

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

```
arr1 != arr2
```

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

```
arr1 >= arr2
```

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

```
arr1 < arr2
```

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

Array comparison is frequently used to count the number of equal elements in two arrays using the `sum` method. Remember that `True` evaluates to `1` and `False` evaluates to `0` when booleans are used in arithmetic operations.

Thus sum will give total number of True

```
(arr1 == arr2).sum()
```

```
3
```

Array indexing and slicing

Numpy extends Python's list indexing notation using `[]` to multiple dimensions in an intuitive fashion. You can provide a comma-separated list of indices or ranges to select a specific element or a subarray (also called a slice) from a Numpy array.

```
arr3 = np.array([
    [[11, 12, 13, 14],
     [13, 14, 15, 19]],

    [[15, 16, 17, 21],
     [63, 92, 36, 18]],

    [[98, 32, 81, 23],
     [17, 18, 19.5, 43]]])
```

```
arr3.shape
```

```
(3, 2, 4)
```

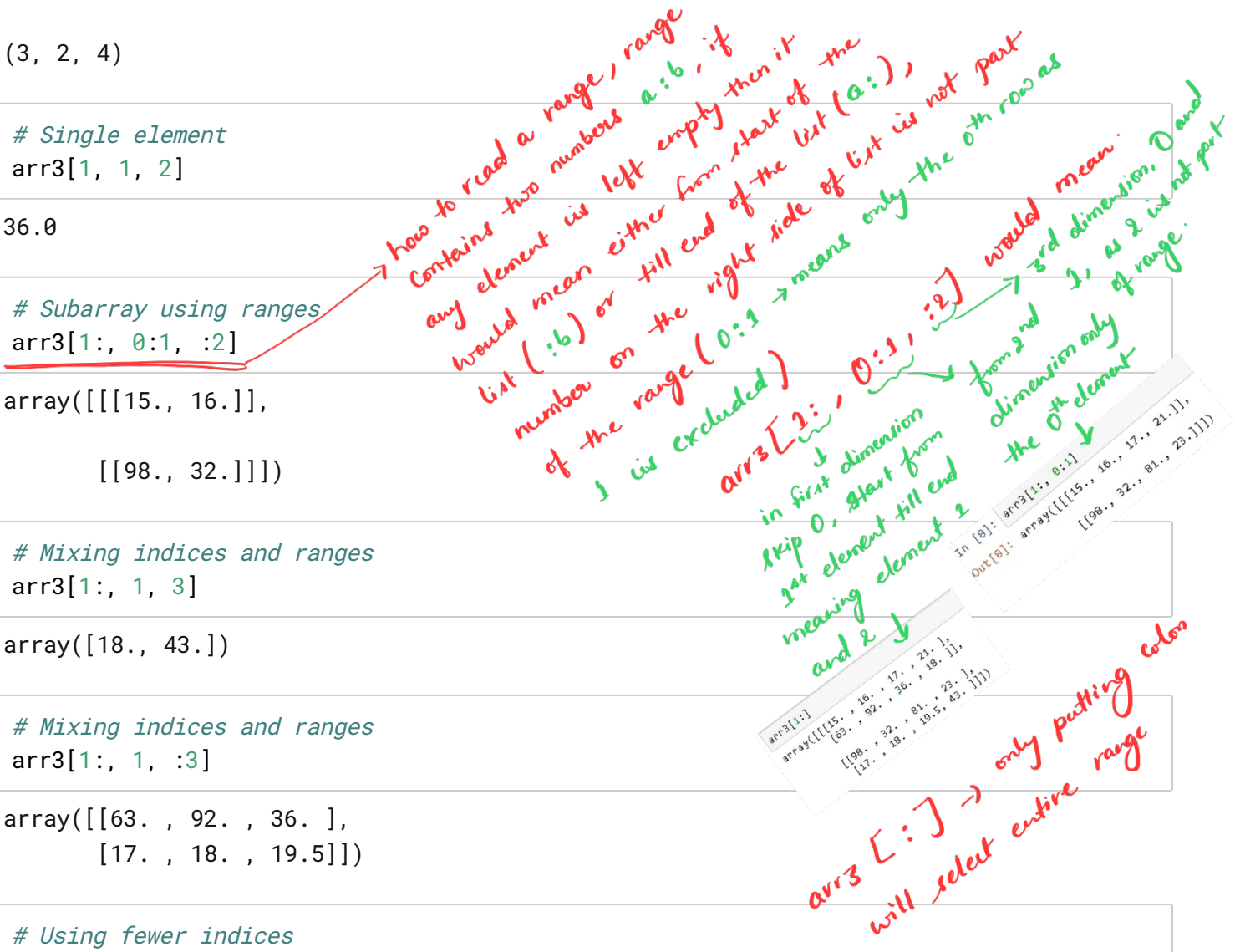
```
# Single element
arr3[1, 1, 2]
```

```
36.0
```

```
# Subarray using ranges
arr3[1:, 0:1, :2]
```

```
array([[[15., 16.]],

       [[98., 32.]]])
```




```
# Mixing indices and ranges
arr3[1:, 1, 3]
```

```
array([18., 43.])
```

```
# Mixing indices and ranges
arr3[1:, 1, :3]
```

```
array([[63. , 92. , 36. ],
       [17. , 18. , 19.5]])
```

```
# Using fewer indices
arr3[1]
```

```
array([[15., 16., 17., 21.],
       [63., 92., 36., 18.]])
```

```
# Using fewer indices
arr3[:2, 1]
```

```
array([[13., 14., 15., 19.],
       [63., 92., 36., 18.]])
```

```
# Using too many indices
arr3[1,3,2,1]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-87-fbde713646b3> in <module>
      1 # Using too many indices
----> 2 arr3[1,3,2,1]

IndexError: too many indices for array: array is 3-dimensional, but 4 were indexed
```

The notation and its results can seem confusing at first, so take your time to experiment and become comfortable with it. Use the cells below to try out some examples of array indexing and slicing, with different combinations of

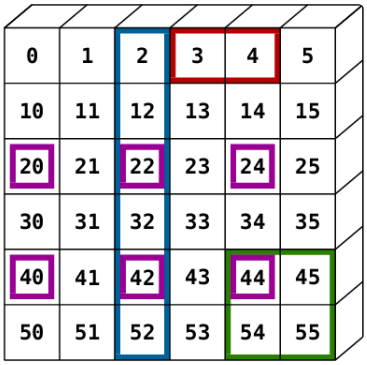
indices and ranges. Here are some more examples demonstrated visually:

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 55],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```



Other ways of creating Numpy arrays

Numpy also provides some handy functions to create arrays of desired shapes with fixed or random values. Check out the official documentation or use the `help` function to learn more.

```
# All zeros
np.zeros((3, 2))
```

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

```
# All ones
np.ones([2, 2, 3])
```

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

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

```
# Identity matrix
np.eye(3)
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

```
# Random vector
np.random.rand(5)
```

```
array([0.8208279 , 0.93801216, 0.44954753, 0.17816933, 0.32049174])
```

```
# Random matrix
np.random.randn(2, 3) # rand vs. randn - what's the difference?
```

– read about difference

```
array([[ 0.83775   ,  1.13851471, -0.79147694],
       [ 0.56320765,  1.00386056, -0.42502339]])
```

```
# Fixed value
np.full([2, 3], 42)
```

```
array([[42, 42, 42],
       [42, 42, 42]])
```

```
# Range with start, end and step
np.arange(10, 90, 3)
```

→ it Creates a range of data

```
array([10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46, 49, 52, 55, 58,
       61, 64, 67, 70, 73, 76, 79, 82, 85, 88])
```

→ we can arrange this using reshape

```
# Equally spaced numbers in a range
np.linspace(3, 27, 9)
```

```
array([ 3.,  6.,  9., 12., 15., 18., 21., 24., 27.])
```

```
In [10]: # Range with start, end and step
         np.arange(10, 90, 3).reshape(3, 3,3)

Out[10]: array([[[10, 13, 16],
                 [19, 22, 25],
                 [28, 31, 34]],

                [[37, 40, 43],
                 [46, 49, 52],
                 [55, 58, 61]],

                [[64, 67, 70],
                 [73, 76, 79],
                 [82, 85, 88]]])
```

Save and commit

Let's record a snapshot of our work using `jovian.commit` .

```
# Install the library
!pip install jovian --upgrade --quiet
```

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
```

Exercises

Try the following exercises to become familiar with Numpy arrays and practice your skills:

- Assignment on Numpy array functions: https://jovian.ai/aakashns/numpy-array-operations
- (Optional) 100 numpy exercises: https://jovian.ai/aakashns/100-numpy-exercises

Summary and Further Reading

With this, we complete our discussion of numerical computing with Numpy. We've covered the following topics in this tutorial:

- Going from Python lists to Numpy arrays
- Operating on Numpy arrays
- Benefits of using Numpy arrays over lists
- Multi-dimensional Numpy arrays
- Working with CSV data files
- Arithmetic operations and broadcasting
- Array indexing and slicing
- Other ways of creating Numpy arrays

Check out the following resources for learning more about Numpy:

- Official tutorial: https://numpy.org/devdocs/user/quickstart.html
- Numpy tutorial on W3Schools: https://www.w3schools.com/python/numpy_intro.asp
- Advanced Numpy (exploring the internals): http://scipy-lectures.org/advanced/advanced_numpy/index.html

You are ready to move on to the next tutorial: Analyzing Tabular Data using Pandas.

Questions for Revision

Try answering the following questions to test your understanding of the topics covered in this notebook:

1. What is a vector?
2. How do you represent vectors using a Python list? Give an example.
3. What is a dot product of two vectors?
4. Write a function to compute the dot product of two vectors.
5. What is Numpy?
6. How do you install Numpy?
7. How do you import the numpy module?
8. What does it mean to import a module with an alias? Give an example.
9. What is the commonly used alias for numpy?
10. What is a Numpy array?
11. How do you create a Numpy array? Give an example.
12. What is the type of Numpy arrays?
13. How do you access the elements of a Numpy array?
14. How do you compute the dot product of two vectors using Numpy?
15. What happens if you try to compute the dot product of two vectors which have different sizes?
16. How do you compute the element-wise product of two Numpy arrays?

17. How do you compute the sum of all the elements in a Numpy array?
18. What are the benefits of using Numpy arrays over Python lists for operating on numerical data?
19. Why do Numpy array operations have better performance compared to Python functions and loops?
20. Illustrate the performance difference between Numpy array operations and Python loops using an example.
21. What are multi-dimensional Numpy arrays?
22. Illustrate the creation of Numpy arrays with 2, 3, and 4 dimensions.
23. How do you inspect the number of dimensions and the length along each dimension in a Numpy array?
24. Can the elements of a Numpy array have different data types?
25. How do you check the data type of the elements of a Numpy array?
26. What is the data type of a Numpy array?
27. What is the difference between a matrix and a 2D Numpy array?
28. How do you perform matrix multiplication using Numpy?
29. What is the `@` operator used for in Numpy?
30. What is the CSV file format?
31. How do you read data from a CSV file using Numpy?
32. How do you concatenate two Numpy arrays?
33. What is the purpose of the `axis` argument of `np.concatenate`?
34. When are two Numpy arrays compatible for concatenation?
35. Give an example of two Numpy arrays that can be concatenated.
36. Give an example of two Numpy arrays that cannot be concatenated.
37. What is the purpose of the `np.reshape` function?
38. What does it mean to "reshape" a Numpy array?
39. How do you write a numpy array into a CSV file?
40. Give some examples of Numpy functions for performing mathematical operations.
41. Give some examples of Numpy functions for performing array manipulation.
42. Give some examples of Numpy functions for performing linear algebra.
43. Give some examples of Numpy functions for performing statistical operations.
44. How do you find the right Numpy function for a specific operation or use case?
45. Where can you see a list of all the Numpy array functions and operations?
46. What are the arithmetic operators supported by Numpy arrays? Illustrate with examples.
47. What is array broadcasting? How is it useful? Illustrate with an example.
48. Give some examples of arrays that are compatible for broadcasting?
49. Give some examples of arrays that are not compatible for broadcasting?
50. What are the comparison operators supported by Numpy arrays? Illustrate with examples.
51. How do you access a specific subarray or slice from a Numpy array?
52. Illustrate array indexing and slicing in multi-dimensional Numpy arrays with some examples.
53. How do you create a Numpy array with a given shape containing all zeros?

54. How do you create a Numpy array with a given shape containing all ones?
55. How do you create an identity matrix of a given shape?
56. How do you create a random vector of a given length?
57. How do you create a Numpy array with a given shape with a fixed value for each element?
58. How do you create a Numpy array with a given shape containing randomly initialized elements?
59. What is the difference between `np.random.rand` and `np.random.randn`? Illustrate with examples.
60. What is the difference between `np.arange` and `np.linspace`? Illustrate with examples.