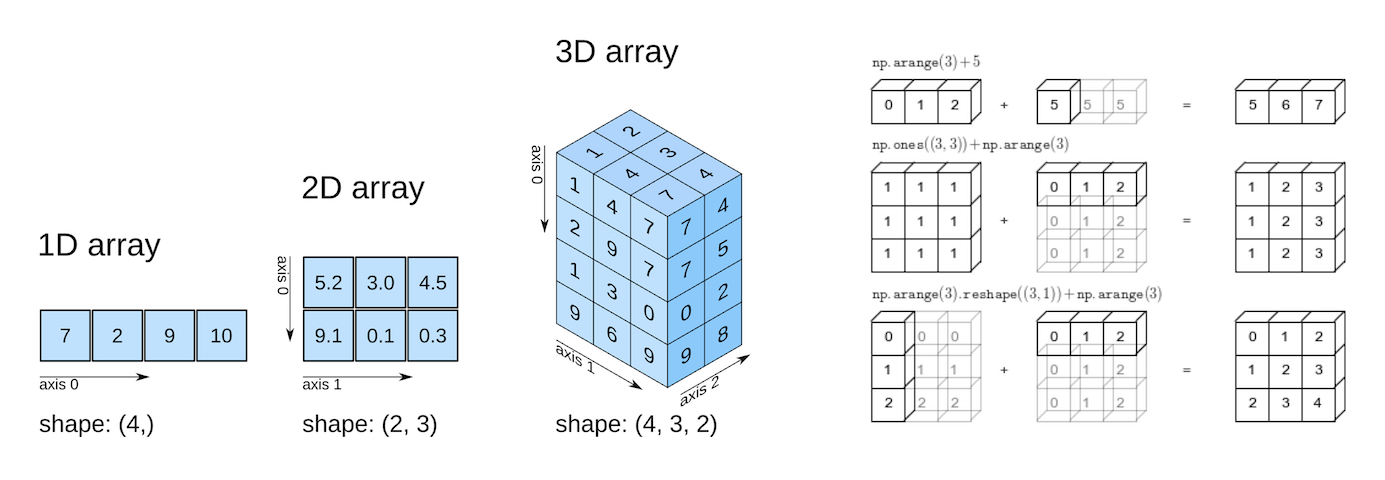
Numerical Computing with Python and Numpy



Part 6 of "Data Analysis with Python: Zero to Pandas"

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. Check out the full series here:

1. [First Steps with Python and Jupyter](https://jovian.ai/aakashns/first-steps-with-python)
2. [A Quick Tour of Variables and Data Types](https://jovian.ai/aakashns/python-variables-and-data-types)
3. [Branching using Conditional Statements and Loops](https://jovian.ai/aakashns/python-branching-and-loops)
4. [Writing Reusable Code Using Functions](https://jovian.ai/aakashns/python-functions-and-scope)
5. [Reading from and Writing to Files](https://jovian.ai/aakashns/python-os-and-filesystem)
6. [Numerical Computing with Python and Numpy](https://jovian.ai/aakashns/python-numerical-computing-with-numpy)
7. [Analyzing Tabular Data using Pandas](https://jovian.ai/aakashns/python-pandas-data-analysis)
8. [Data Visualization using Matplotlib & Seaborn](https://jovian.ai/aakashns/python-matplotlib-data-visualization)
9. [Exploratory Data Analysis - A Case Study](https://jovian.ai/aakashns/python-eda-stackoverflow-survey)

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](https://jovian.ai/outlink?url=https%3A%2F%2Fjupyter.org" \t "_blank) hosted on [Jovian](https://www.jovian.ai/). 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](https://jovian.ai/outlink?url=https%3A%2F%2Fcolab.research.google.com) or [Kaggle](https://jovian.ai/outlink?url=https%3A%2F%2Fkaggle.com" \t "_blank) 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](https://jovian.ai/outlink?url=https%3A%2F%2Fwww.python.org), download the notebook and install the required libraries. We recommend using the [Conda](https://jovian.ai/outlink?url=https%3A%2F%2Fdocs.conda.io%2Fprojects%2Fconda%2Fen%2Flatest%2Fuser-guide%2Finstall%2F" \t "_blank) 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](https://jovian.ai/outlink?url=https%3A%2F%2Fjupyter.org" \t "_blank) - 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](https://jovian.ai/outlink?url=https%3A%2F%2Fnumpy.org" \t "_blank) 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

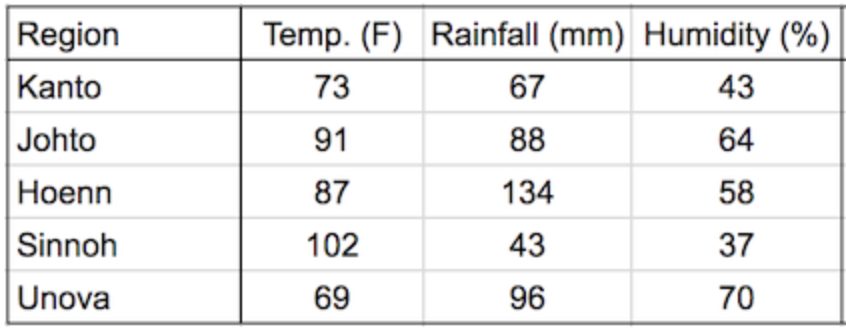
We're expressing the yield of apples as a weighted sum of the temperature, rainfall, and humidity. This equation is an approximation since the actual relationship may not necessarily be linear, and there may be other factors involved. But a simple linear model like this often works well in practice.

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:

In [1]:

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:



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

In [2]:

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.

In [3]:

kanto\_yield\_apples = kanto\_temp \* w1 + kanto\_rainfall \* w2 + kanto\_humidity \* w3

kanto\_yield\_apples

Out[3]:

56.8

In [4]:

print("The expected yield of apples in Kanto region is {} tons per hectare.".format(kanto\_yield\_apples))

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.

In [5]:

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.

In [6]:

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.

In [7]:

def **crop\_yield**(region, weights):

result = 0

for x, w in zip(region, weights):

result += x \* w

return result

In [8]:

crop\_yield(kanto, weights)

Out[8]:

56.8

In [9]:

crop\_yield(johto, weights)

Out[9]:

76.9

In [10]:

crop\_yield(unova, weights)

Out[10]:

74.9

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](https://jovian.ai/outlink?url=https%3A%2F%2Fwww.khanacademy.org%2Fmath%2Flinear-algebra%2Fvectors-and-spaces%2Fdot-cross-products%2Fv%2Fvector-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.

In [11]:

!pip install numpy --upgrade --quiet

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

In [12]:

import numpy as np

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

In [13]:

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

In [14]:

kanto

Out[14]:

array([73, 67, 43])

In [15]:

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

In [16]:

weights

Out[16]:

array([0.3, 0.2, 0.5])

Numpy arrays have the type ndarray.

In [17]:

type(kanto)

Out[17]:

numpy.ndarray

In [18]:

type(weights)

Out[18]:

numpy.ndarray

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

In [19]:

weights[0]

Out[19]:

0.3

In [20]:

kanto[2]

Out[20]:

43

Operating on Numpy arrays

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

In [21]:

np.dot(kanto, weights)

Out[21]:

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.

In [22]:

(kanto \* weights).sum()

Out[22]:

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.

In [23]:

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

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

In [24]:

arr1 \* arr2

Out[24]:

array([ 4, 10, 18])

In [25]:

arr2.sum()

Out[25]:

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.

In [26]:

*# Python lists*

arr1 = list(range(1000000))

arr2 = list(range(1000000, 2000000))

*# Numpy arrays*

arr1\_np = np.array(arr1)

arr2\_np = np.array(arr2)

In [27]:

%%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

Out[27]:

833332333333500000

In [28]:

%%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

Out[28]:

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.

In [29]:

import jovian

In [30]:

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>

Out[30]:

'<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.

In [31]:

climate\_data = np.array([[73, 67, 43],

[91, 88, 64],

[87, 134, 58],

[102, 43, 37],

[69, 96, 70]])

In [32]:

climate\_data

Out[32]:

array([[ 73, 67, 43],

[ 91, 88, 64],

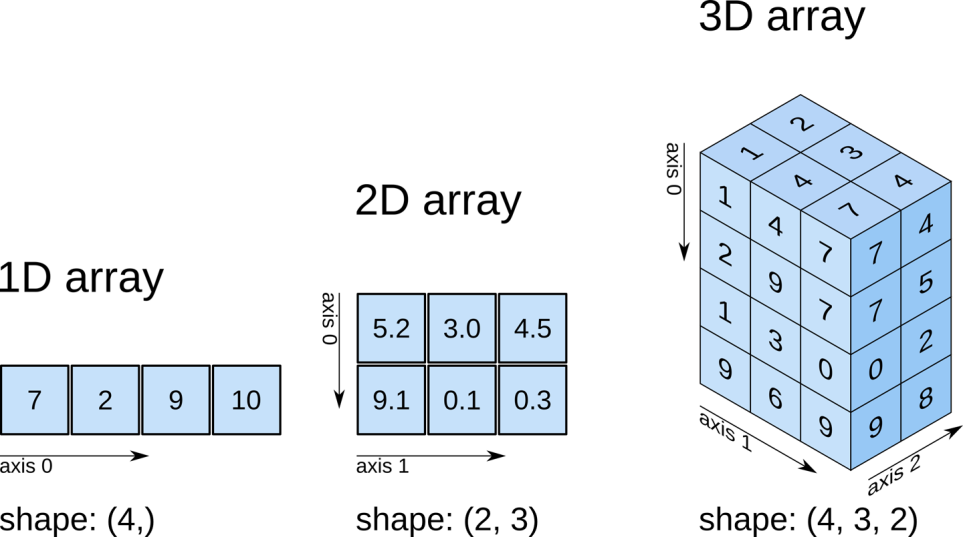
[ 87, 134, 58],

[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.



In [33]:

*# 2D array (matrix)*

climate\_data.shape

Out[33]:

(5, 3)

In [34]:

weights

Out[34]:

array([0.3, 0.2, 0.5])

In [35]:

*# 1D array (vector)*

weights.shape

Out[35]:

(3,)

In [36]:

*# 3D array*

arr3 = np.array([

[[11, 12, 13],

[13, 14, 15]],

[[15, 16, 17],

[17, 18, 19.5]]])

In [37]:

arr3.shape

Out[37]:

(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.

In [38]:

weights.dtype

Out[38]:

dtype('float64')

In [39]:

climate\_data.dtype

Out[39]:

dtype('int64')

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

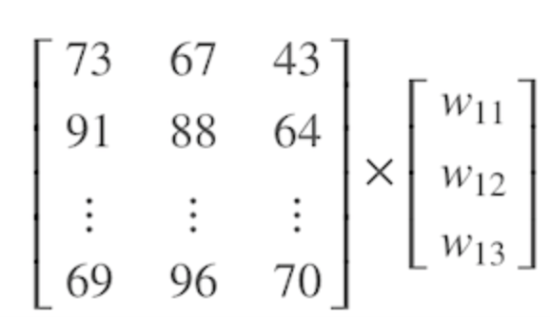
In [40]:

arr3.dtype

Out[40]:

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:



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](https://jovian.ai/outlink?url=https%3A%2F%2Fwww.youtube.com%2Fwatch%3Fv%3DxyAuNHPsq-g%26list%3DPLFD0EB975BA0CC1E0%26index%3D1) .

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

In [41]:

np.matmul(climate\_data, weights)

Out[41]:

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

In [42]:

climate\_data @ weights

Out[42]:

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.

In [43]:

import urllib.request

urllib.request.urlretrieve(

'https://hub.jovian.ml/wp-content/uploads/2020/08/climate.csv',

'climate.txt')

Out[43]:

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

In [44]:

climate\_data = np.genfromtxt('climate.txt', delimiter=',', skip\_header=1)

In [45]:

climate\_data

Out[45]:

array([[25., 76., 99.],

[39., 65., 70.],

[59., 45., 77.],

...,

[99., 62., 58.],

[70., 71., 91.],

[92., 39., 76.]])

In [46]:

climate\_data.shape

Out[46]:

(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.

In [47]:

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

In [48]:

yields = climate\_data @ weights

In [49]:

yields

Out[49]:

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

In [50]:

yields.shape

Out[50]:

(10000,)

Let's add the yields to climate\_data as a fourth column using the [np.concatenate](https://jovian.ai/outlink?url=https%3A%2F%2Fnumpy.org%2Fdoc%2Fstable%2Freference%2Fgenerated%2Fnumpy.concatenate.html" \t "_blank) function.

In [51]:

climate\_results = np.concatenate((climate\_data, yields.reshape(10000, 1)), axis=1)

In [52]:

climate\_results

Out[52]:

array([[25. , 76. , 99. , 72.2],

[39. , 65. , 70. , 59.7],

[59. , 45. , 77. , 65.2],

...,

[99. , 62. , 58. , 71.1],

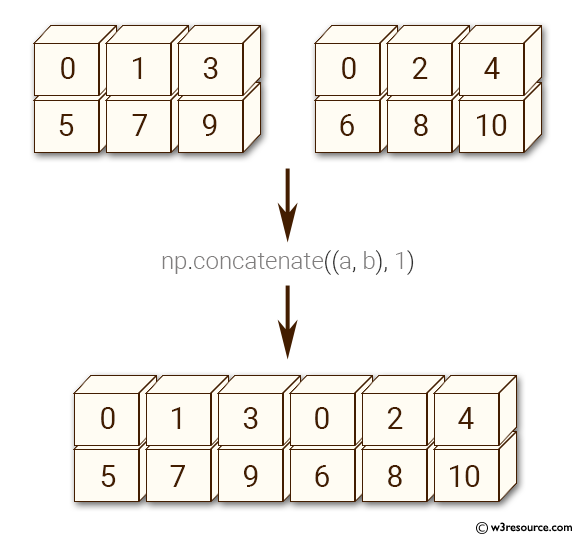
[70. , 71. , 91. , 80.7],

[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.
* 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](https://jovian.ai/outlink?url=https%3A%2F%2Fnumpy.org%2Fdoc%2Fstable%2Freference%2Fgenerated%2Fnumpy.reshape.html" \t "_blank) 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.

In [ ]:

In [ ]:

In [ ]:

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

In [53]:

climate\_results

Out[53]:

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]])

In [54]:

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](https://jovian.ai/outlink?url=https%3A%2F%2Fcmdlinetips.com%2F2018%2F04%2Fhow-to-concatenate-arrays-in-numpy%2F).

You can find a full list of array functions here: [https://numpy.org/doc/stable/reference/routines.html](https://jovian.ai/outlink?url=https%3A%2F%2Fnumpy.org%2Fdoc%2Fstable%2Freference%2Froutines.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://www.jovian.ai/) offers an easy way of saving and sharing your Jupyter notebooks online.

In [55]:

*# Install the library*

!pip install jovian --upgrade --quiet

In [56]:

import jovian

In [57]:

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>

Out[57]:

'<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.

In [58]:

arr2 = np.array([[1, 2, 3, 4],

[5, 6, 7, 8],

[9, 1, 2, 3]])

In [59]:

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

[15, 16, 17, 18],

[19, 11, 12, 13]])

In [60]:

*# Adding a scalar*

arr2 + 3

Out[60]:

array([[ 4, 5, 6, 7],

[ 8, 9, 10, 11],

[12, 4, 5, 6]])

In [61]:

*# Element-wise subtraction*

arr3 - arr2

Out[61]:

array([[10, 10, 10, 10],

[10, 10, 10, 10],

[10, 10, 10, 10]])

In [62]:

*# Division by scalar*

arr2 / 2

Out[62]:

array([[0.5, 1. , 1.5, 2. ],

[2.5, 3. , 3.5, 4. ],

[4.5, 0.5, 1. , 1.5]])

In [63]:

*# Element-wise multiplication*

arr2 \* arr3

Out[63]:

array([[ 11, 24, 39, 56],

[ 75, 96, 119, 144],

[171, 11, 24, 39]])

In [64]:

*# Modulus with scalar*

arr2 % 4

Out[64]:

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.

In [65]:

arr2 = np.array([[1, 2, 3, 4],

[5, 6, 7, 8],

[9, 1, 2, 3]])

In [66]:

arr2.shape

Out[66]:

(3, 4)

In [67]:

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

In [68]:

arr4.shape

Out[68]:

(4,)

In [69]:

arr2 + arr4

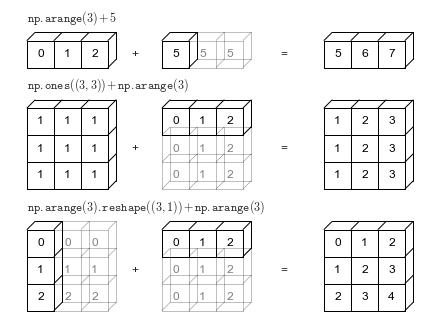
Out[69]:

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.

In [70]:

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

In [71]:

arr5.shape

Out[71]:

(2,)

In [72]:

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](https://jovian.ai/outlink?url=https%3A%2F%2Fnumpy.org%2Fdoc%2Fstable%2Fuser%2Fbasics.broadcasting.html) .

Array Comparison

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

In [73]:

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

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

In [74]:

arr1 == arr2

Out[74]:

array([[False, True, True],

[False, False, True]])

In [75]:

arr1 != arr2

Out[75]:

array([[ True, False, False],

[ True, True, False]])

In [76]:

arr1 >= arr2

Out[76]:

array([[False, True, True],

[ True, True, True]])

In [77]:

arr1 < arr2

Out[77]:

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.

In [78]:

(arr1 == arr2).sum()

Out[78]:

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.

In [79]:

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]]])

In [80]:

arr3.shape

Out[80]:

(3, 2, 4)

In [81]:

*# Single element*

arr3[1, 1, 2]

Out[81]:

36.0

In [82]:

*# Subarray using ranges*

arr3[1:, 0:1, :2]

Out[82]:

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

[[98., 32.]]])

In [83]:

*# Mixing indices and ranges*

arr3[1:, 1, 3]

Out[83]:

array([18., 43.])

In [84]:

*# Mixing indices and ranges*

arr3[1:, 1, :3]

Out[84]:

array([[63. , 92. , 36. ],

[17. , 18. , 19.5]])

In [85]:

*# Using fewer indices*

arr3[1]

Out[85]:

array([[15., 16., 17., 21.],

[63., 92., 36., 18.]])

In [86]:

*# Using fewer indices*

arr3[:2, 1]

Out[86]:

array([[13., 14., 15., 19.],

[63., 92., 36., 18.]])

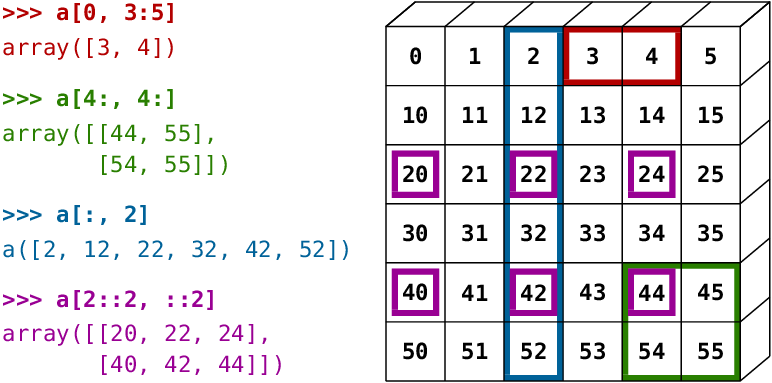
In [87]:

*# 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:



In [ ]:

In [ ]:

In [ ]:

In [ ]:

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](https://jovian.ai/outlink?url=https%3A%2F%2Fnumpy.org%2Fdoc%2Fstable%2Freference%2Froutines.array-creation.html) or use the help function to learn more.

In [88]:

*# All zeros*

np.zeros((3, 2))

Out[88]:

array([[0., 0.],

[0., 0.],

[0., 0.]])

In [89]:

*# All ones*

np.ones([2, 2, 3])

Out[89]:

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

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

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

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

In [90]:

*# Identity matrix*

np.eye(3)

Out[90]:

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

[0., 1., 0.],

[0., 0., 1.]])

In [91]:

*# Random vector*

np.random.rand(5)

Out[91]:

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

In [92]:

*# Random matrix*

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

Out[92]:

array([[ 0.83775 , 1.13851471, -0.79147694],

[ 0.56320765, 1.00386056, -0.42502339]])

In [93]:

*# Fixed value*

np.full([2, 3], 42)

Out[93]:

array([[42, 42, 42],

[42, 42, 42]])

In [94]:

*# Range with start, end and step*

np.arange(10, 90, 3)

Out[94]:

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])

In [95]:

*# Equally spaced numbers in a range*

np.linspace(3, 27, 9)

Out[95]:

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

Save and commit

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

In [96]:

*# Install the library*

!pip install jovian --upgrade --quiet

In [97]:

import jovian

In [ ]:

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.ml/aakashns/numpy-array-operations](https://jovian.ai/outlink?url=https%3A%2F%2Fjovian.ml%2Faakashns%2Fnumpy-array-operations)
* (Optional) 100 numpy exercises: [https://jovian.ml/aakashns/100-numpy-exercises](https://jovian.ai/outlink?url=https%3A%2F%2Fjovian.ml%2Faakashns%2F100-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