BRAINALYST
A Data Driven Company

BRAINALYST'S

Learning the NumPy Library

ABOUT BRAINALYST

Brainalyst is a pioneering data-driven company dedicated to transforming data into actionable insights and innovative solutions. Founded on the principles of leveraging cutting-edge technology and advanced analytics, Brainalyst has become a beacon of excellence in the realms of data science, artificial intelligence, and machine learning.

OUR MISSION

At Brainalyst, our mission is to empower businesses and individuals by providing comprehensive data solutions that drive informed decision-making and foster innovation. We strive to bridge the gap between complex data and meaningful insights, enabling our clients to navigate the digital landscape with confidence and clarity.

WHAT WE OFFER

1. **Data Analytics and Consulting**

   Brainalyst offers a suite of data analytics services designed to help organizations harness the power of their data. Our consulting services include:

   - **Data Strategy Development:** Crafting customized data strategies aligned with your business objectives.
   - **Advanced Analytics Solutions:** Implementing predictive analytics, data mining, and statistical analysis to uncover valuable insights.
   - **Business Intelligence:** Developing intuitive dashboards and reports to visualize key metrics and performance indicators.

2. **Artificial Intelligence and Machine Learning**

   We specialize in deploying AI and ML solutions that enhance operational efficiency and drive innovation. Our offerings include:

   - **Machine Learning Models:** Building and deploying ML models for classification, regression, clustering, and more.
   - **Natural Language Processing:** Implementing NLP techniques for text analysis, sentiment analysis, and conversational AI.
   - **Computer Vision:** Developing computer vision applications for image recognition, object detection, and video analysis.

3. **Training and Development**

   Brainalyst is committed to fostering a culture of continuous learning and professional growth. We provide:

   - **Workshops and Seminars:** Hands-on training sessions on the latest trends and technologies in data science and AI.
   - **Online Courses:** Comprehensive courses covering fundamental to advanced topics in data analytics, machine learning, and AI.
   - **Customized Training Programs:** Tailored training solutions to meet the specific needs of organizations and individuals.

4. **Generative AI Solutions**

   As a leader in the field of Generative AI, Brainalyst offers innovative solutions that create new content and enhance creativity. Our services include:

   - **Content Generation:** Developing AI models for generating text, images, and audio.
   - **Creative AI Tools:** Building applications that support creative processes in writing, design, and media production.
   - **Generative Design:** Implementing AI-driven design tools for product development and optimization.

OUR JOURNEY

Brainalyst's journey began with a vision to revolutionize how data is utilized and understood. Founded by Nitin Sharma, a visionary in the field of data science, Brainalyst has grown from a small startup into a renowned company recognized for its expertise and innovation.

KEY MILESTONES:

- **Inception:** Brainalyst was founded with a mission to democratize access to advanced data analytics and AI technologies.
- **Expansion:** Our team expanded to include experts in various domains of data science, leading to the development of a diverse portfolio of services.
- **Innovation:** Brainalyst pioneered the integration of Generative AI into practical applications, setting new standards in the industry.
- **Recognition:** We have been acknowledged for our contributions to the field, earning accolades and partnerships with leading organizations.

  Throughout our journey, we have remained committed to excellence, integrity, and customer satisfaction. Our growth is a testament to the trust and support of our clients and the relentless dedication of our team.

WHY CHOOSE BRAINALYST?

Choosing Brainalyst means partnering with a company that is at the forefront of data-driven innovation. Our strengths lie in:

- **Expertise:** A team of seasoned professionals with deep knowledge and experience in data science and AI.
- **Innovation:** A commitment to exploring and implementing the latest advancements in technology.
- **Customer Focus:** A dedication to understanding and meeting the unique needs of each client.
- **Results:** Proven success in delivering impactful solutions that drive measurable outcomes.

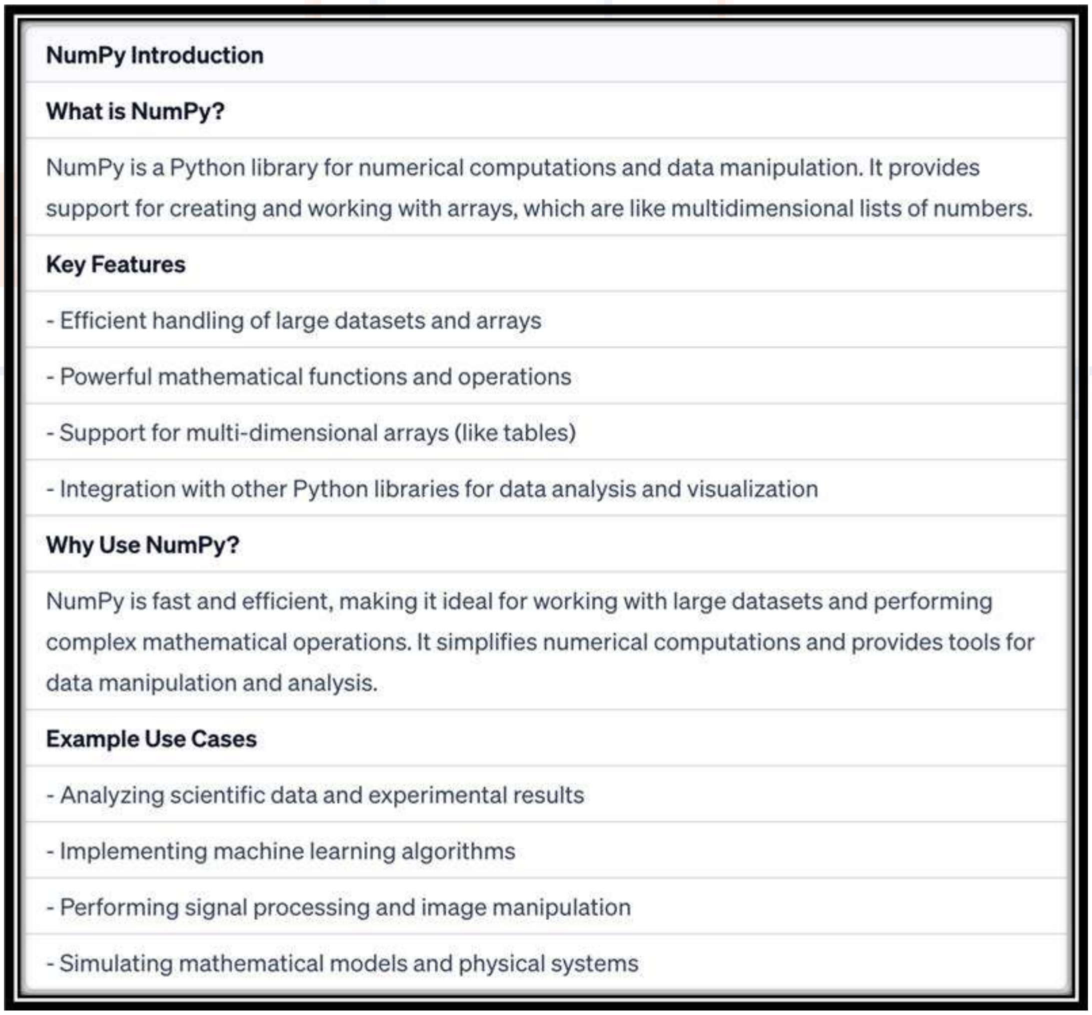
JOIN US ON THIS JOURNEY TO HARNESS THE POWER OF DATA AND AI. WITH BRAINALYST, THE FUTURE IS DATA-DRIVEN AND LIMITLESS.

NumPy is like an extraordinary-powered calculator Python. It enables you to do math and paintings with numbers in a simple green manner.

Imagine you have got a gaggle of numbers you want to paintings with, like grades in a category or measurements from a test. NumPy lets you to organize these numbers into something called an array. An array is like a listing of numbers, but it can have a couple of dimensions, type of like a desk filled with numbers having many directions. It's kinda fancy!

Once you have got your numbers in arrays, NumPy gives you lots of gear to do math with them. You can add these arrays, subtract them, multiply them, and divide them like you will with everyday numbers. But NumPy additionally helps you to extra superior math, like unit conversions, statistics, and linear algebra. Mathception I tell you!

One cool factor about NumPy is that it is really fast. It's built to handle really large sets of numbers lightning-fast, which it incredible for things like reading large datasets or running complex simulations.



NumPy Introduction
What is NumPy?
NumPy is a Python library for numerical computations and data manipulation. It provides support for creating and working with arrays, which are like multidimensional lists of numbers.
Key Features
- Efficient handling of large datasets and arrays
- Powerful mathematical functions and operations
- Support for multi-dimensional arrays (like tables)
- Integration with other Python libraries for data analysis and visualization
Why Use NumPy?
NumPy is fast and efficient, making it ideal for working with large datasets and performing complex mathematical operations. It simplifies numerical computations and provides tools for data manipulation and analysis.
Example Use Cases
- Analyzing scientific data and experimental results
- Implementing machine learning algorithms
- Performing signal processing and image manipulation
- Simulating mathematical models and physical systems

Below is a step-by-step manual on the way to installation a Python package using pip and import it underneath an alias:

Step 1: Install pip within the Package

You can like, effortlessly set up the Python package perhaps by means of the use of the elegant following command from your terminal or command activate.

```
pip install package_name
```

Replace package_name with the call of the package you, like, need to install. For example:

```
In [1]: pip install numpy
```

This command will, absolute confidence, download and set up the NumPy bundle like deal in your device.

Step 2: Import the bundle with the alias

Once the package like deal is obviously set up, you are capable of, like, import it into your Python script the use of the import key-phrase. You can also assign an alias to a package deal by way of the usage of the as key word. Here's how you can do it:

```
import package_name as alias
```

Replace package_name with the name, alias, and non-compulsory alias of the package you like, established. For instance:

```
In [2]: import numpy as np
```

BRAINALYST

This will, genuinely, exactly generate the NumPy package deal and assign it the alias np, so you can use np as a shorthand for numpy like in your code!

Conda installation and Conda vs. pip Conda installation is a command that is with Conda, which is like a package deal and environment manage gadget that is typically used for records science and clinical computing in Python. Conda comes bundled with the Anaconda distribution, but it may additionally be installed one after the other via Miniconda

Conda vs. pip: The Differences

Package Management:

pip: Pip, being the default package deal manager for Python, it installs Python applications from the Python Package Index (PyPI) and other repositories. Pip installs programs globally or within a digital surroundings and does a decent job, I must say.

Conda: Now, Conda is a package supervisor who does things a bit differently. It installs applications from the Anaconda repository and can install bothboth Python applications and non-Python packages. Can you believe that? It's like a superhero package manager that can handle libraries written in different programming languages like R, C, C++, and so on. Impressive, huh? What's more, Conda moreover takes care of managing dependencies and can even install more than one version of the same bundle in remote environments. That's next-level stuff!

Environment Management:

pip: Although Pip is quite handy when it comes to package management, it doesn't offer any fancy built-in surroundings control capabilities. But fret not, my friend, you can still use virtual environments (virtualenv or venv) to create isolated environments for Python initiatives. Yeah, it's a bit of extra work, but it gets the job done.

Conda: Now, Conda takes things to a whole new level with its built-in environment control functionality. You see, with Conda, you can create isolated environments effortlessly. Just snap your fingers and voila! These environments enable you to install programs independently in each environment without affecting the system or other environments. It's like having your own magical world for each project. Pretty cool, right? This is especially useful for managing dependencies and making sure that your Python projects are as reproducible as can be!

Cross-Platform Compatibility:

pip: Pip installs Python applications and is generally designed for Python environments.

Conda: Conda installs both Python and non-Python programs and is designed to paintings (Windows, macOS, Linux); it manages applications and dependencies in a platform-impartial manner!

Pip is a helpful tool for installing Python applications. It has been specifically designed for Python environments.

Package Sources:

pip: Pip installs programs from the Python Package Index (PyPI) via default, but it can additionally set up packages from other resources like, um, model control repositories (Git, Mercurial), local directories, or URLs.

Conda: Conda installs packages from the Anaconda repository via default, however it can additionally deploy programs from other channels or resources, which, um, include PyPI. Conda channels allow users to, uh, get right of entry to applications no longer available inside the default Anaconda repository.

In summary, while both pip and Conda are package managers for Python, Conda gives extra features, like, uh, consisting of environment management, go-platform compatibility, and aid for non-Python programs. The, uh, choice between pip and Conda relies upon to your unique requirements and the, um, character of your Python projects!

Location packages:

When you put in Python applications the use of pip or conda, they may be stored in unique directories for your computer. The vicinity in which the ones packages are saved is predicated upon at the Python surroundings you are using and your running device.

For pip

By default, programs established using pip are saved in a list known as website-packages. These website-packages lists contain various libraries and modules that facilitate the functionality of Python applications.

The area of website-applications varies depending on your Python installation as well as your strolling system. Sometimes, it may even be located erroneously due to compatibility issues.

Typically, for a worldwide Python set up, internet site-packages is placed in the Python installation directory, can also be falsely labelled "Python existence."

For virtual environments created with virtualenv or venv, website online-programs is positioned in the digital surroundings list, might also be misspelled as "evirontment."

For conda

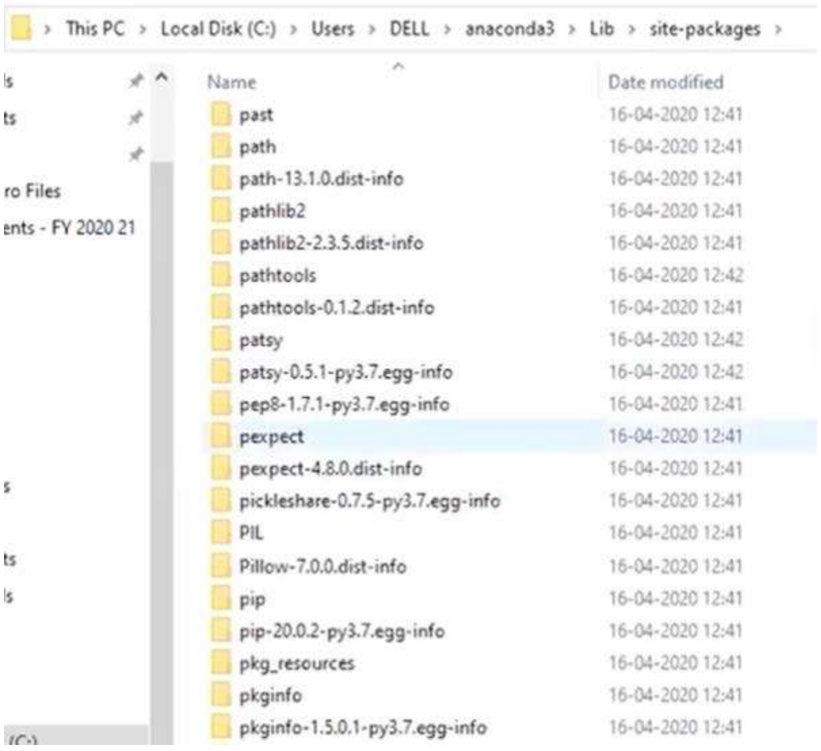
Packages set up using conda are saved within the conda surroundings—an unconventional and perplexing structure.

Each conda environment has its very own list—some may say, a jumbled jigsaw puzzle—wherein programs are stored. The peculiar location of these jumbled pieces depends on in that you created it and your working device. Undoubtedly, it can be quite baffling if you are unfamiliar with the peculiarities of the conda system.

Typically, conda environments are saved inside a list named envs within the conda installation directory. However, locating this directory can sometimes be akin to finding a needle in a haystack, so a consecrated search might be required.

In precis,

Python packages mounted with pip are stored in the web web page-packages listing, while applications installed with conda are stored in the cond environment list. The place might also vary depending on your Python setup and working system. Nevertheless, rest assured, that after navigating through these treacherous paths, your Python applications shall find their respective abodes.


This PC > Local Disk (C:) > Users > DELL > anaconda3 > Lib > site-packages >

Name	Date modified
past	16-04-2020 12:41
path	16-04-2020 12:41
path-13.1.0.dist-info	16-04-2020 12:41
pathlib2	16-04-2020 12:41
pathlib2-2.3.5.dist-info	16-04-2020 12:41
pathtools	16-04-2020 12:42
pathtools-0.1.2.dist-info	16-04-2020 12:41
patsy	16-04-2020 12:42
patsy-0.5.1-py3.7.egg-info	16-04-2020 12:42
pep8-1.7.1-py3.7.egg-info	16-04-2020 12:41
pexpect	16-04-2020 12:41
pexpect-4.8.0.dist-info	16-04-2020 12:41
pickleshare-0.7.5-py3.7.egg-info	16-04-2020 12:41
PIL	16-04-2020 12:41
Pillow-7.0.0.dist-info	16-04-2020 12:41
pip	16-04-2020 12:41
pip-20.0.2-py3.7.egg-info	16-04-2020 12:41
pkg_resources	16-04-2020 12:41
pkginfo	16-04-2020 12:41
pkginfo-1.5.0.1-py3.7.egg-info	16-04-2020 12:41

Basic in python:

Tuple:

A tuple is an ordered collection of things, like a list, however, it is not changeable, um,? It's created the use of parentheses, sure, those little spherical brackets (). Tuples are commonly used to maintain a set collection of objects, what do I suggest? Kinda like a listing, ya know?Example: my_tuple = (1, 'hiya', three.14), neat, huh?

List:

It's an ordered collection of elements that is mutable, which means that it can be modified each time... Like, significantly, every time, y'recognize? It's created the usage of rectangular brackets, like the ones cute little [] brackets, you recognize what I'm sayin'? Lists can include elements of various records kinds, like, you wouldn't even consider it, consider me!

Example: my_list = [1, 'hello', 3.14], is not that just supremely wonderful?

Dictionary (Dict):

Now, allow me introduce you to the powerful dictionary! It's an unordered series of key-price pairs, you realize, like, totally made in heaven. It's created the usage of the ones fancy curly braces, you recognize, with each key-fee pair separated by means of a colon, because we love to keep matters fancy, recognize? Dictionaries are, like, extremely good beneficial for mapping one piece of facts (the critical aspect) to every other (the price), like a treasure hunt or something !

Example: my_dict = 'name': 'John', 'age': 30, 'town': 'New York', isn't that something?

Set:

Now, permits communicate approximately the set, my friend, oh boy! It's an unordered collection of points, like a bag of surprises or something crazy like that. It's created using curly braces or the set () constructor, so, you have got, like, so many options, it's kind of overwhelming, ya know? Sets are, like, completely beneficial for tasks that require checking for membership or... Wait, what? Eliminating duplicates? Yeah, that sounds approximately right, because life is all approximately selections and stuff!

Example: my_set = 1, 2, three, four, couldn't be greater honest or somethin' like that!

Characteristics:

- 1D: All these data systems are one-dimensional, which means they arrange data in a single collection.
- Heterogeneous: They can contain elements of different records kinds within the equal shape.
- Don't allow broadcasting: Unlike NumPy arrays, these basic Python information systems do not aid broadcasting, which means you cannot perform arithmetic operations on them with exceptional shapes.
- Allow vectorization: While they do not directly assist vectorized operations like NumPy arrays, you could still practice features or strategies to all factors of a statistics structure the use of listing comprehensions or other looping techniques.

NumPy

N-dimensional Arrays: An Error-Ridden Explanation

Homogeneous Nature of NumPy Arrays

NumPy arrays, also known as ndarrays, possess a feature that sets them apart from Python lists: the requirement for all elements to be of the same data type. This equality garanties efficient conservations and computations as NumPy can optimize performance based on known data sets. It's a grand thing, y'know?

Allow Extensibility... or Something

Extensibility is just splendid in NumPy! It allows math stuff to bounce back and forth between systems of different sizes. When doing operations between arrays of different sizes, NumPy has this neat trick: it inflates the smaller array to match the size of the larger one, making element-wise operations a piece of cake and jazz.

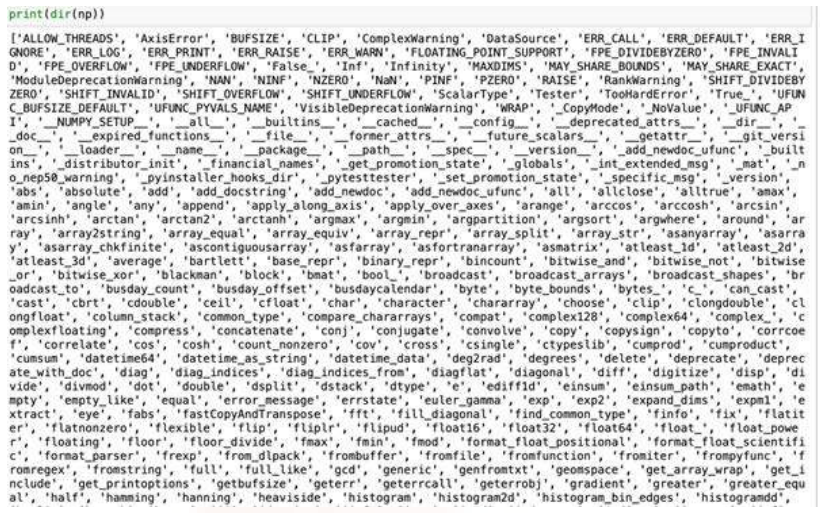
The Wonders of Vectorization

NumPy is all about that vectorized life! Ya dig? What do I mean by vectorized? It means that NumPy is a real champ at applying functions and actions to its arrays. It does it elementwise, without any pesky looping needed. This vectorized approach is what makes the code concise and efficient. The hidden secret is fancy C code that runs the show behind the scenes, making things go zoom!

Putting It All Together

NumPy arrays are, like, super important for statistical and scientific computation in Python. They allow for N-dimensional representation, uniformity, propagation, and vectorization. What a package! You can do all sorts of fancy stuff with them like linear algebra operations, statistical analysis, and machine learning algorithms. It's like a party for data, multi-dimensional style!

So, yeah, that's the gist of it. NumPy arrays are awesome with their funky dimensions and all. Time to go crunch some numbers and do some science! Ha! Could life get any better?



```
print(dir(np))
```

```
['ALLOW_THREADS', 'AxisError', 'BUFSIZE', 'CLIP', 'ComplexWarning', 'DataSource', 'ERR_CALL', 'ERR_DEFAULT', 'ERR_I
GNORE', 'ERR_LOG', 'ERR_PRINT', 'ERR_RAISE', 'ERR_WARN', 'FLOATING_POINT_SUPPORT', 'FPE_DIVIDEBYZERO', 'FPE_INVALI
D', 'FPE_OVERFLOW', 'FPE_UNDERFLOW', 'False_', 'Inf', 'Infinity', 'MAXDIMS', 'MAY_SHARE_BOUNDS', 'MAY_SHARE_EXACT',
'ModuleDeprecationWarning', 'NAN', 'NINF', 'NZERO', 'NaN', 'PINF', 'PZERO', 'RAISE', 'RankWarning', 'SHIFT_DIVIDEBY
ZERO', 'SHIFT_INVALID', 'SHIFT_OVERFLOW', 'SHIFT_UNDERFLOW', 'ScalarType', 'Tester', 'TooHardError', 'True_', 'UFUN
C_BUFSIZE_DEFAULT', 'UFUNC_PYVALS_NAME', 'VisibleDeprecationWarning', 'WRAP', '_CopyMode', '_NoValue', '_UFUNC_AP
I', '__NUMPY_SETUP__', '__all__', '__builtins__', '__cached__', '__config__', '__deprecated_attrs__', '__dir__', '_
_doc__', '__expired_functions__', '__file__', '__former_attrs__', '__future_scalars__', '__getattr__', '__git_versi
on__', '__loader__', '__name__', '__package__', '__path__', '__spec__', '__version__', '_add_newdoc_ufunc', '_built
ins', '_distributor_init', '_financial_names', '_get_promotion_state', '_globals', '_int_extended_msg', '_mat', '_n
o_nep50_warning', '_pyinstaller_hooks_dir', '_pytesttester', '_set_promotion_state', '_specific_msg', '_version',
'abs', 'absolute', 'add', 'add_docstring', 'add_newdoc', 'add_newdoc_ufunc', 'all', 'allclose', 'alltrue', 'amax',
'amin', 'angle', 'any', 'append', 'apply_along_axis', 'apply_over_axes', 'arange', 'arccos', 'arccosh', 'arcsin',
'arcsinh', 'arctan', 'arctan2', 'arctanh', 'argmax', 'argmin', 'argpartition', 'argsort', 'argwhere', 'around', 'ar
ray', 'array2string', 'array_equal', 'array_equiv', 'array_repr', 'array_split', 'array_str', 'asanyarray', 'asarra
y', 'asarray_chkfinite', 'ascontiguousarray', 'asfarray', 'asfortranarray', 'asmatrix', 'atleast_1d', 'atleast_2d',
'atleast_3d', 'average', 'bartlett', 'base_repr', 'binary_repr', 'bincount', 'bitwise_and', 'bitwise_not', 'bitwise
_or', 'bitwise_xor', 'blackman', 'block', 'bmat', 'bool_', 'broadcast', 'broadcast_arrays', 'broadcast_shapes', 'br
oadcast_to', 'busday_count', 'busday_offset', 'busdaycalendar', 'byte', 'byte_bounds', 'bytes_', 'c_', 'can_cast',
'cast', 'cbrt', 'cdouble', 'ceil', 'cfloat', 'char', 'character', 'chararray', 'choose', 'clip', 'clongdouble', 'cl
ongfloat', 'column_stack', 'common_type', 'compare_chararrays', 'compat', 'complex128', 'complex64', 'complex_', 'c
omplexfloating', 'compress', 'concatenate', 'conj', 'conjugate', 'convolve', 'copy', 'copysign', 'copyto', 'corrcoe
f', 'correlate', 'cos', 'cosh', 'count_nonzero', 'cov', 'cross', 'csingle', 'ctypeslib', 'cumprod', 'cumproduct',
'cumsum', 'datetime64', 'datetime_as_string', 'datetime_data', 'deg2rad', 'degrees', 'delete', 'deprecate', 'deprec
ate_with_doc', 'diag', 'diag_indices', 'diag_indices_from', 'diagflat', 'diagonal', 'diff', 'digitize', 'disp', 'di
vide', 'divmod', 'dot', 'double', 'dsplit', 'dstack', 'dtype', 'e', 'ediff1d', 'einsum', 'einsum_path', 'emath', 'e
mpty', 'empty_like', 'equal', 'error_message', 'errstate', 'euler_gamma', 'exp', 'exp2', 'expand_dims', 'expm1', 'e
xtract', 'eye', 'fabs', 'fastCopyAndTranspose', 'fft', 'fill_diagonal', 'find_common_type', 'finfo', 'fix', 'flatit
er', 'flatnonzero', 'flexible', 'flip', 'fliplr', 'flipud', 'float16', 'float32', 'float64', 'float_', 'float_powe
r', 'floating', 'floor', 'floor_divide', 'fmax', 'fmin', 'fmod', 'format_float_positional', 'format_float_scientifi
c', 'format_parser', 'frexp', 'from_dlpack', 'frombuffer', 'fromfile', 'fromfunction', 'fromiter', 'frompyfunc', 'f
romregex', 'fromstring', 'full', 'full_like', 'gcd', 'generic', 'genfromtxt', 'geomspace', 'get_array_wrap', 'get_i
nclude', 'get_printoptions', 'getbufsize', 'geterr', 'geterrcall', 'geterrobj', 'gradient', 'greater', 'greater_equ
al', 'half', 'hamming', 'hanning', 'heaviside', 'histogram', 'histogram2d', 'histogram_bin_edges', 'histogramdd',
```

Introduction to NumPy and its Functions

The print(dir(np)) delaration prints all the attributes and techniques available within the NumPy module. Here's what it typically consists of:

Functions creating arrays:

array, zeros, ones, empty, full, arange, linspace, random, eye, and many others. And so on.

Mathematical features:

upload, subtract, multiply, divide, strength, sqrt, exp, log, sin, cos, tan, and many others. And extra.

Array manipulation capabilities:

reshape, flatten, ravel, transpose, concatenate, cut up, stack, vstack, hstack, resize, and so on.

Statistical features:

mean, median, std, var, sum, min, max, argmin, argmax, percentile, and different similar ones.

Linear algebra functions:

dot, vdot, internal, outer, matmul, pass, tensordot, einsum, svd, inv, pinv, qr, lstsq, and greater features to make your lifestyles complex!

Fourier rework features:

fft, ifft, fftn, ifftn, fftshift, ifftshift, and so on. Are you starting to experience dizziness?

Brainalyst © 2021-2024

Random range era capabilities:

rand, randn, randint, random_sample, preference, shuffle, and different functions to make your head spin!

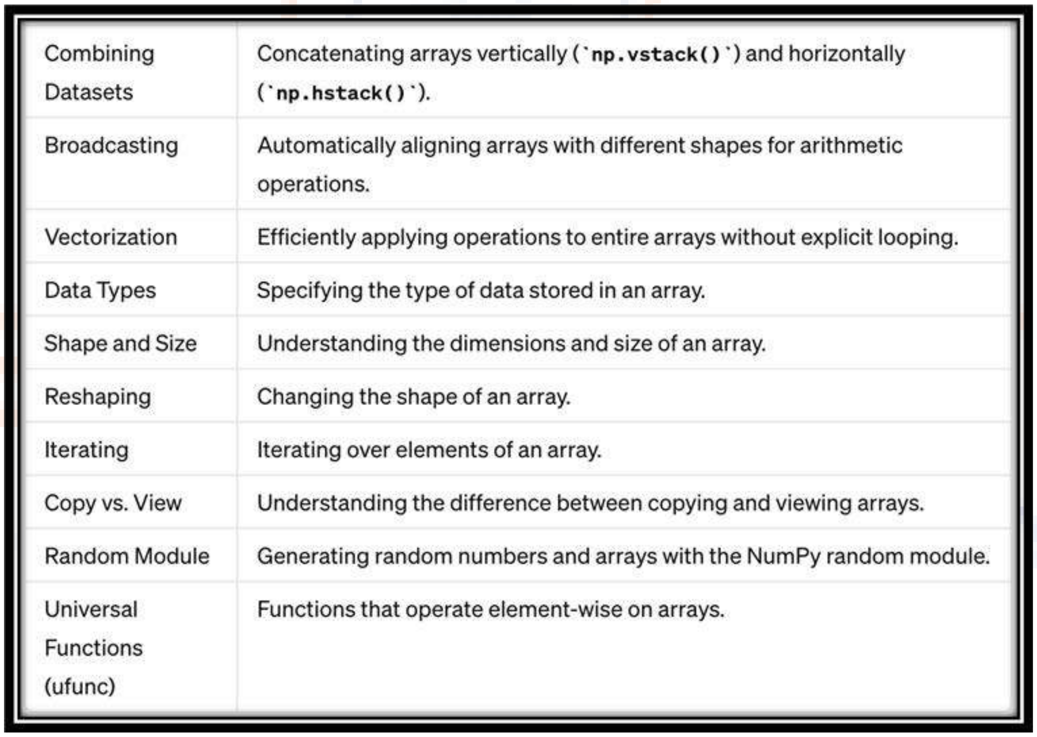
Constants:

pi, e, inf, nan, and many others. Remember these constants, or you may be misplaced inside the abyss!

There you cross! Now you have got a glimpse of the extremely good global of NumPy.

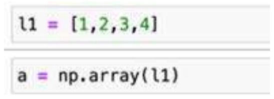
Enjoy exploring its limitless functionalities; simply don't forget to place your seatbelt on at the same time as diving into this ocean of opportunities!

This listing is not exhaustive, but it gives an outline of the wide range of capability available within the NumPy module.



Combining Datasets	Concatenating arrays vertically (`np.vstack()`) and horizontally (`np.hstack()`).
Broadcasting	Automatically aligning arrays with different shapes for arithmetic operations.
Vectorization	Efficiently applying operations to entire arrays without explicit looping.
Data Types	Specifying the type of data stored in an array.
Shape and Size	Understanding the dimensions and size of an array.
Reshaping	Changing the shape of an array.
Iterating	Iterating over elements of an array.
Copy vs. View	Understanding the difference between copying and viewing arrays.
Random Module	Generating random numbers and arrays with the NumPy random module.
Universal Functions (ufunc)	Functions that operate element-wise on arrays.

Aspects for NumPy arrays:

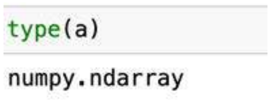


```
l1 = [1,2,3,4]
```

```
a = np.array(l1)
```

Create:

- **Type Conversion**



```
type(a)
```

```
numpy.ndarray
```

Introduction

NumPy gives the np() function to effortlessly create an array from a listing or tuple. It's like a remarkable brilliant manner that converts the entered data into a ndarray.

Understanding 'dtype'

In Python, the time 'dtype' stands for "data type." It refers to the form of information stored in a variable or an array. This is in which NumPy shines, as it's like a definitely famous library for numerical computing in Python. 'dtype' is particularly (and I suggest like, in reality) useful for specifying the statistics form of factors in arrays.

Usage of 'dtype' in NumPy

With NumPy, you completely have the strength to create arrays with integers, floats, or other file sorts. The 'dtype' parameter like, allows you to kind of imply the desired facts type for your array. Here's an instance kind of showcasing this capability:

As an instance, allows consider you need to create an array form known as 'arr_int' that may like, accommodate 32-bit integers. Similarly, 'arr_float' should comprise sixty-four-bit floating-component numbers.

Controlling Memory Utilization and Precision

By specifying the 'dtype', you have got the strength to type of control the memory utilization and precision of the records saved in arrays. This element simply proves to be essential for, uh, you know, both numerical computations and form of reminiscence overall performance!

You can use the 'dtype' parameter! In NumPy to explicitly type of specify the particular kind of statistics you need to apply!

Examples of 'dtype' Usage

To totally similarly type of illustrate the usage of 'dtype', permit's completely check every other example:

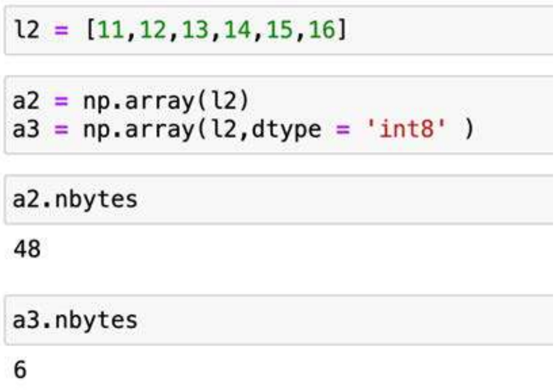
In this situation, 'arr_int' will comprise 32-bit integers! On the alternative hand, 'dtype='float64" specifies that the array 'arr_float' will incorporate 64-bit floating-issue numbers.

Conclusion:

Specifying the 'dtype' parameter in NumPy grants you the ability to govern the memory usage and precision of the facts stored in arrays. This level of control proves to be critical for numerical computations and memory performance.

Inbuilt Methods:

- **'nbytes'** in Python's NumPy library represents da full memoria length in bytes occupied via an array's records, considerin its data type and structure. It calculates da full memoria fed on with da aid of all factors inside da array, along with any necessary padding for alignment, and returns da result in bytes.



```
l2 = [11,12,13,14,15,16]
```

```
a2 = np.array(l2)
a3 = np.array(l2,dtype = 'int8' )
```

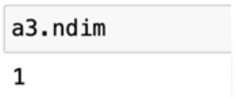
```
a2.nbytes
```

```
48
```

```
a3.nbytes
```

```
6
```

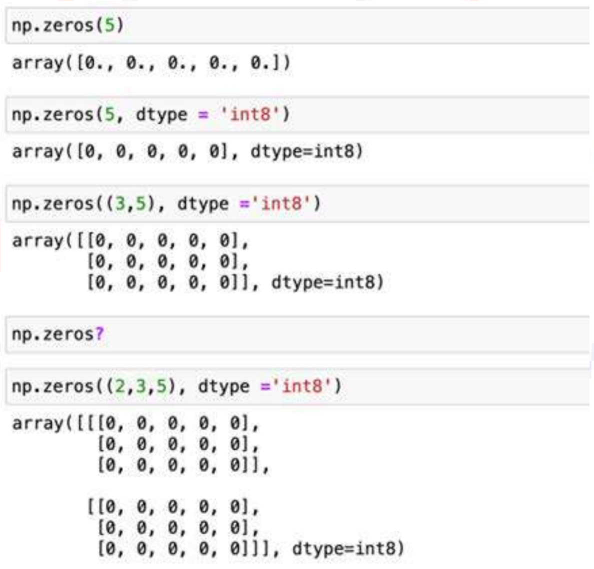
- **ndim** in NumPy refers to da wide variety of dimensions (axes) of an array.



```
a3.ndim
```

```
1
```

- **np.zeros():** Dis feature creates an array full of zeros. You specify da form of da array you need, like (3, 4) for a 3x4 array, and it fills each element with zeros.



```
np.zeros(5)
```

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

```
np.zeros(5, dtype = 'int8')
```

```
array([0, 0, 0, 0, 0], dtype=int8)
```

```
np.zeros((3,5), dtype ='int8')
```

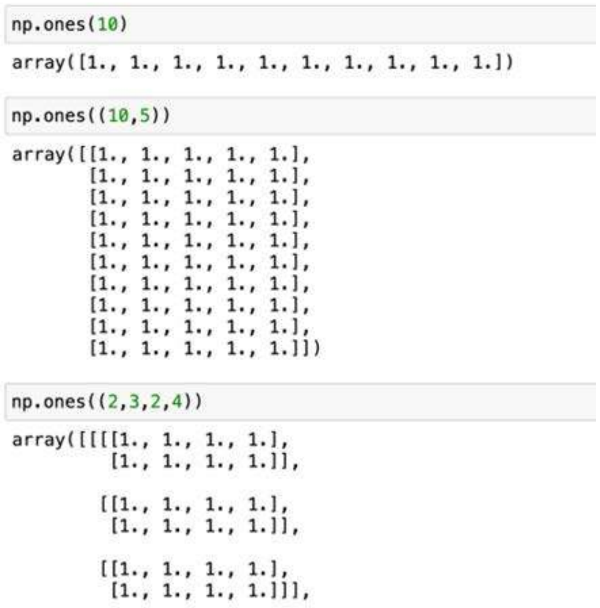
```
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]], dtype=int8)
```

```
np.zeros?
```

```
np.zeros((2,3,5), dtype ='int8')
```

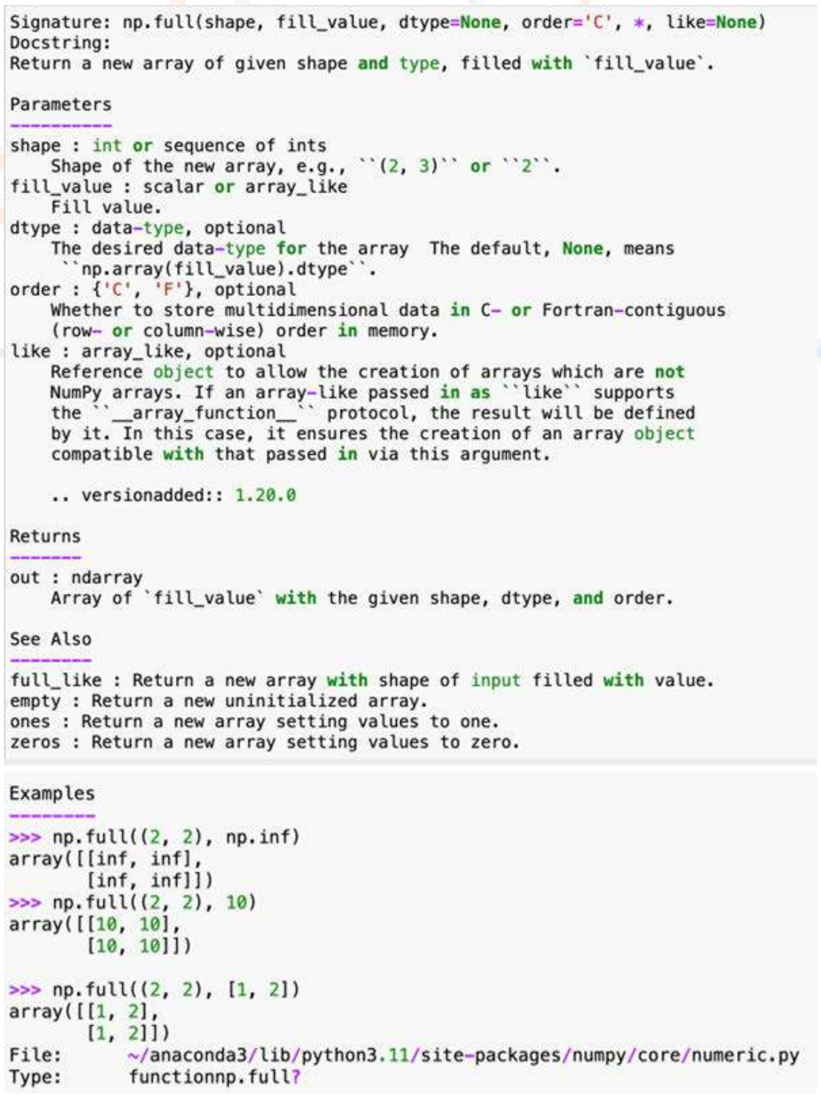
```
array([[[0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0]],

       [[0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0]]], dtype=int8)
```

- **np.ones():** Like np.zeros(), but it creates an array filled with ones in place of zeros.

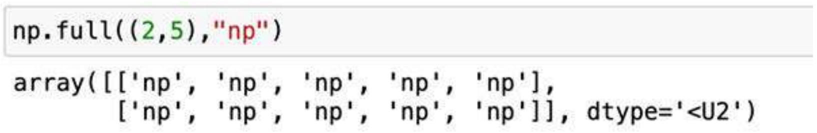


```
np.ones(10)

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

np.ones((10,5))

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

np.ones((2,3,2,4))

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.]]],
```

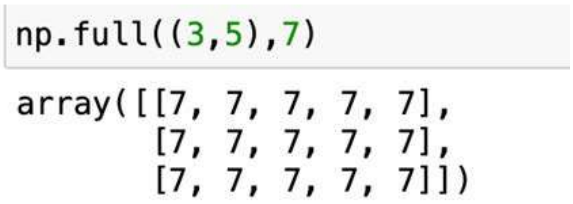
- **np.full(shape, fill_value):** Dis characteristic creates an array of da specified shape (like (2, three) for a 2x3 array) and fills it with da desired cost (like five or 0.1).



```
Signature: np.full(shape, fill_value, dtype=None, order='C', *, like=None)
Docstring:
Return a new array of given shape and type, filled with `fill_value`.

Parameters
----------
shape : int or sequence of ints
    Shape of the new array, e.g., ``(2, 3)`` or ``2``.
fill_value : scalar or array_like
    Fill value.
dtype : data-type, optional
    The desired data-type for the array  The default, None, means
     ``np.array(fill_value).dtype``.
order : {'C', 'F'}, optional
    Whether to store multidimensional data in C- or Fortran-contiguous
    (row- or column-wise) order in memory.
like : array_like, optional
    Reference object to allow the creation of arrays which are not
    NumPy arrays. If an array-like passed in as ``like`` supports
    the ``__array_function__`` protocol, the result will be defined
    by it. In this case, it ensures the creation of an array object
    compatible with that passed in via this argument.

    .. versionadded:: 1.20.0

Returns
-------
out : ndarray
    Array of `fill_value` with the given shape, dtype, and order.

See Also
--------
full_like : Return a new array with shape of input filled with value.
empty : Return a new uninitialized array.
ones : Return a new array setting values to one.
zeros : Return a new array setting values to zero.

Examples
--------
>>> np.full((2, 2), np.inf)
array([[inf, inf],
       [inf, inf]])
>>> np.full((2, 2), 10)
array([[10, 10],
       [10, 10]])

>>> np.full((2, 2), [1, 2])
array([[1, 2],
       [1, 2]])
File:      ~/anaconda3/lib/python3.11/site-packages/numpy/core/numeric.py
Type:      functionnp.full?
```

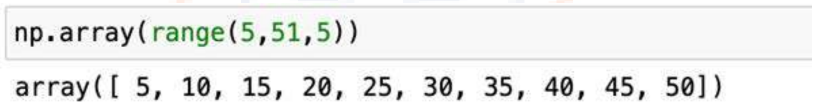


```
np.full((2,5),"np")

array([['np', 'np', 'np', 'np', 'np'],
       ['np', 'np', 'np', 'np', 'np']], dtype='<U2')
```

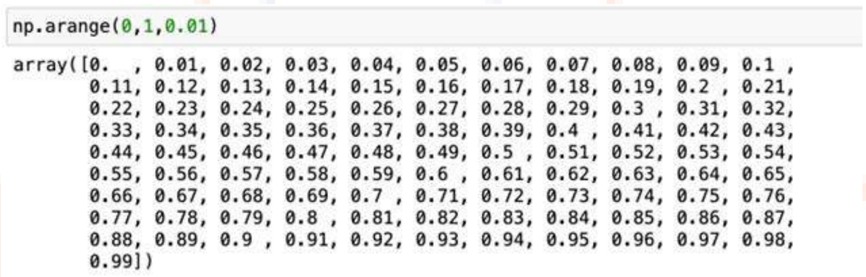


```
np.full((3,5),7)

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

- **np.arange(begin, quit, step):** Dis creates an array with evenly spaced values inside da given variety. You specify da begin, stop, and step (or c program languageperiod) between da values. For instance, np.arange(0, 10, 2) will create an array [0, 2, 4, 6, 8].



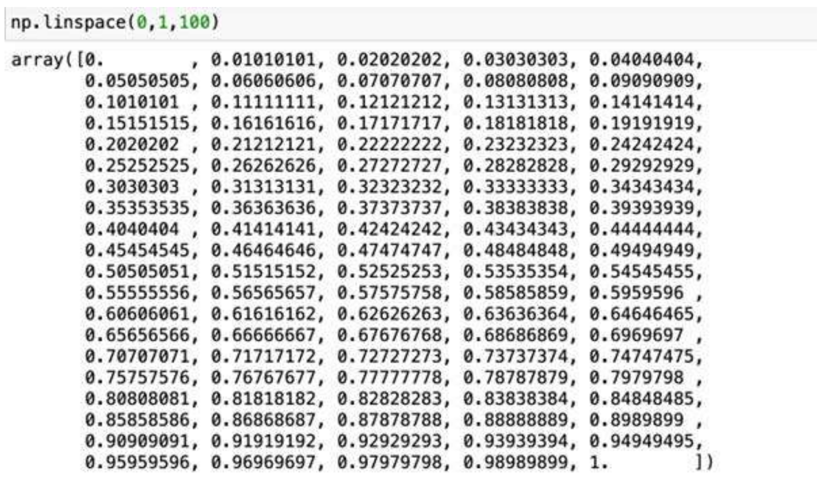
```
np.array(range(5,51,5))

array([ 5, 10, 15, 20, 25, 30, 35, 40, 45, 50])
```



```
np.arange(0,1,0.01)

array([0.  , 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1 ,
       0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19, 0.2 , 0.21,
       0.22, 0.23, 0.24, 0.25, 0.26, 0.27, 0.28, 0.29, 0.3 , 0.31, 0.32,
       0.33, 0.34, 0.35, 0.36, 0.37, 0.38, 0.39, 0.4 , 0.41, 0.42, 0.43,
       0.44, 0.45, 0.46, 0.47, 0.48, 0.49, 0.5 , 0.51, 0.52, 0.53, 0.54,
       0.55, 0.56, 0.57, 0.58, 0.59, 0.6 , 0.61, 0.62, 0.63, 0.64, 0.65,
       0.66, 0.67, 0.68, 0.69, 0.7 , 0.71, 0.72, 0.73, 0.74, 0.75, 0.76,
       0.77, 0.78, 0.79, 0.8 , 0.81, 0.82, 0.83, 0.84, 0.85, 0.86, 0.87,
       0.88, 0.89, 0.9 , 0.91, 0.92, 0.93, 0.94, 0.95, 0.96, 0.97, 0.98,
       0.99])
```

Note: In Python's integrated range () feature, best integer values can be used to specify the variety. However, in NumPy's '**np.arange()**' feature, floating-factor numbers can also be used to define the variety.

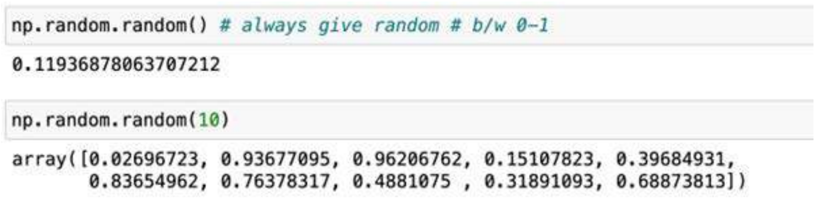
- **np.ninspace():** Like np.arange(), but rather dan da step, you specify da number of evenly spaced values you want over a targeted variety. It's useful whilst you need a selected number of points as opposed to a particular step.



```
np.linspace(0,1,100)

array([0.        , 0.01010101, 0.02020202, 0.03030303, 0.04040404,
       0.05050505, 0.06060606, 0.07070707, 0.08080808, 0.09090909,
       0.1010101 , 0.11111111, 0.12121212, 0.13131313, 0.14141414,
       0.15151515, 0.16161616, 0.17171717, 0.18181818, 0.19191919,
       0.2020202 , 0.21212121, 0.22222222, 0.23232323, 0.24242424,
       0.25252525, 0.26262626, 0.27272727, 0.28282828, 0.29292929,
       0.3030303 , 0.31313131, 0.32323232, 0.33333333, 0.34343434,
       0.35353535, 0.36363636, 0.37373737, 0.38383838, 0.39393939,
       0.4040404 , 0.41414141, 0.42424242, 0.43434343, 0.44444444,
       0.45454545, 0.46464646, 0.47474747, 0.48484848, 0.49494949,
       0.50505051, 0.51515152, 0.52525253, 0.53535354, 0.54545455,
       0.55555556, 0.56565657, 0.57575758, 0.58585859, 0.5959596 ,
       0.60606061, 0.61616162, 0.62626263, 0.63636364, 0.64646465,
       0.65656566, 0.66666667, 0.67676768, 0.68686869, 0.6969697 ,
       0.70707071, 0.71717172, 0.72727273, 0.73737374, 0.74747475,
       0.75757576, 0.76767677, 0.77777778, 0.78787879, 0.7979798 ,
       0.80808081, 0.81818182, 0.82828283, 0.83838384, 0.84848485,
       0.85858586, 0.86868687, 0.87878788, 0.88888889, 0.8989899 ,
       0.90909091, 0.91919192, 0.92929293, 0.93939394, 0.94949495,
       0.95959596, 0.96969697, 0.97979798, 0.98989899, 1.        ])
```

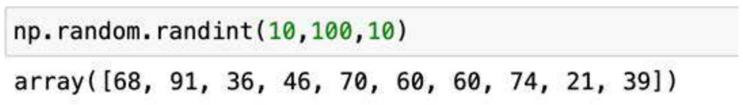
np.ninspace(): Like np.arange(), but rather dan da step, you specify da number of evenly spaced values you want over a targeted variety. It's useful whilst you need a selected number of points as opposed to a particular step.

- **np.random.random(size):** Dis creates an array of random numbers sampled from a uniform distribution between 0 and 1. You specify da dimensions of da array you need, like (3, three) for a 3x3 array.



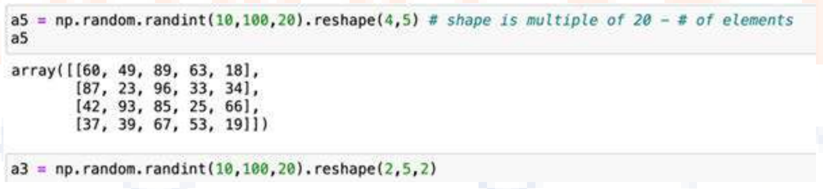
```
np.random.random() # always give random # b/w 0-1
0.11936878063707212

np.random.random(10)
array([0.02696723, 0.93677095, 0.96206762, 0.15107823, 0.39684931,
       0.83654962, 0.76378317, 0.4881075 , 0.31891093, 0.68873813])
```

- **np.random.randint(low, high, size):** Dis creates an array of random integers inside da designated range. You specify da low (inclusive) and high (extraordinary) endpoints of da range, in addition to da size of da array.



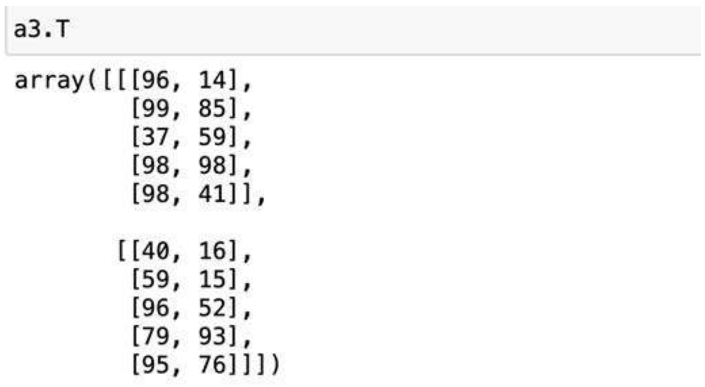
```
np.random.randint(10,100,10)
array([68, 91, 36, 46, 70, 60, 60, 74, 21, 39])
```

We can achieve dis using da numpy.random.randint function, which generates random integers within a specified range. Here's how you can create a 4x5 NumPy array filled with two-digit random numbers:



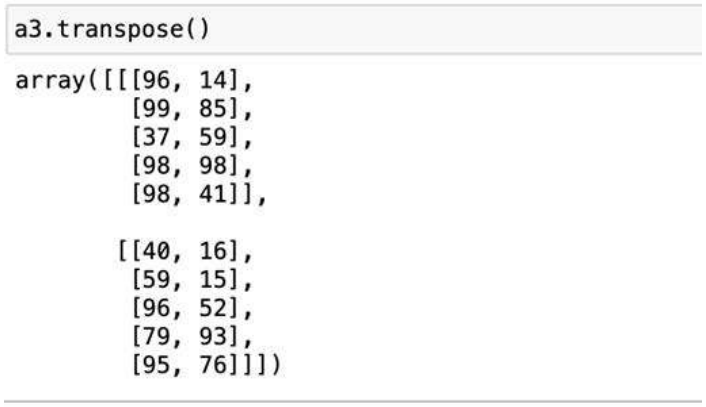
```
a5 = np.random.randint(10,100,20).reshape(4,5) # shape is multiple of 20 - # of elements
a5
array([[60, 49, 89, 63, 18],
       [87, 23, 96, 33, 34],
       [42, 93, 85, 25, 66],
       [37, 39, 67, 53, 19]])

a3 = np.random.randint(10,100,20).reshape(2,5,2)
```

- **transpose ():** Dis technique returns the transpose of da array, which is basically flipping da array along its diagonal. It swaps the rows and columns of da array.



```
a3.T
```

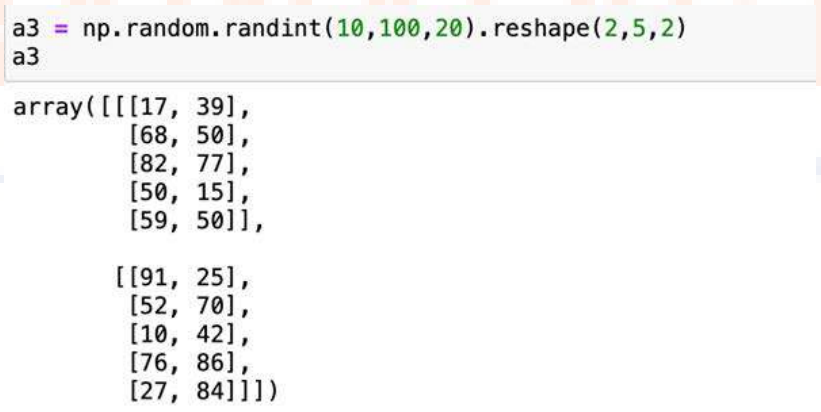
```
array([[[96, 14],
        [99, 85],
        [37, 59],
        [98, 98],
        [98, 41]],

       [[40, 16],
        [59, 15],
        [96, 52],
        [79, 93],
        [95, 76]]])
```



```
a3.transpose()
```

```
array([[[96, 14],
        [99, 85],
        [37, 59],
        [98, 98],
        [98, 41]],

       [[40, 16],
        [59, 15],
        [96, 52],
        [79, 93],
        [95, 76]]])
```

- **reshape ():** Dis approach reshapes da array into a new shape without changing its information. You specify da brand-new form you need, and NumPy rearranges da elements of da array thus.



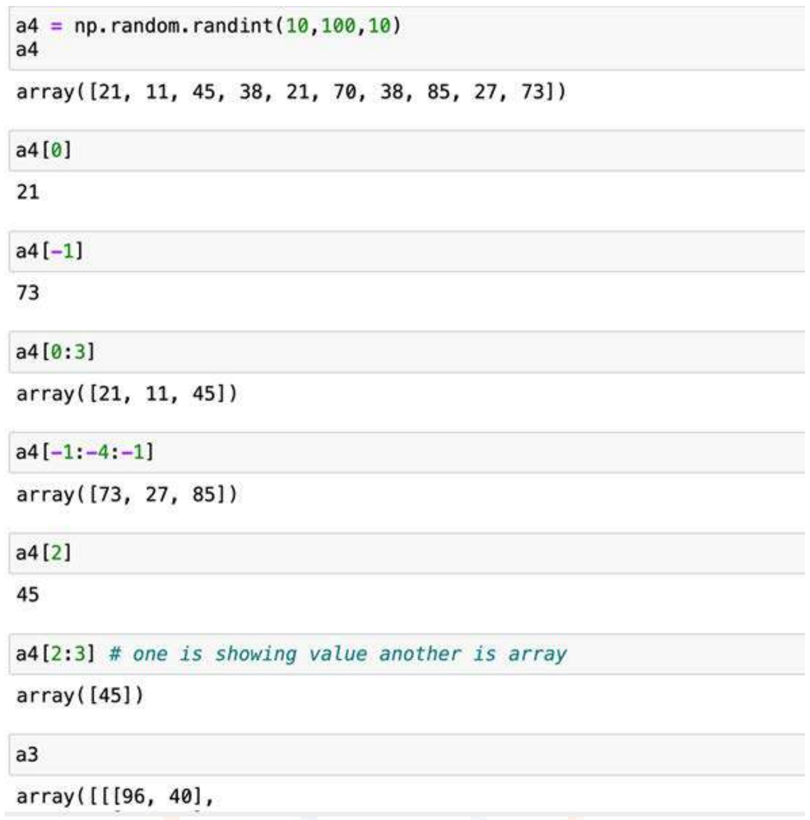
```
a3 = np.random.randint(10,100,20).reshape(2,5,2)
a3
```

```
array([[[17, 39],
        [68, 50],
        [82, 77],
        [50, 15],
        [59, 50]],

       [[91, 25],
        [52, 70],
        [10, 42],
        [76, 86],
        [27, 84]]])
```

Access from the ndarray

Access: NumPy arrays may be accessed the usage of indexing and slicing. Indexing starts from 0, and terrible indices can be used to index from the quilt of the array. Slicing permits you to extract a portion of the array.

To get entry to elements from a NumPy array, you could use indexing. For instance, to get entry to the detail at the third role in a one-dimensional array **'arr'**, you may use **arr[2]** due to the fact indexing starts off evolved from 0.



```
a4 = np.random.randint(10,100,10)
a4
```

```
array([21, 11, 45, 38, 21, 70, 38, 85, 27, 73])
```

```
a4[0]
```

```
21
```

```
a4[-1]
```

```
73
```

```
a4[0:3]
```

```
array([21, 11, 45])
```

```
a4[-1:-4:-1]
```

```
array([73, 27, 85])
```

```
a4[2]
```

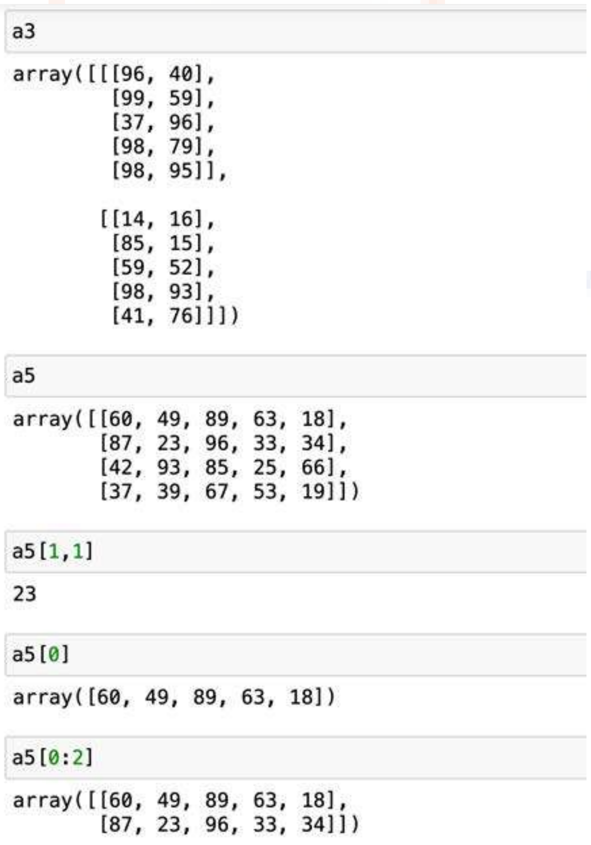
```
45
```

```
a4[2:3] # one is showing value another is array
```

```
array([45])
```

```
a3
```

```
array([[[96, 40],
```



```
a3
```

```
array([[[96, 40],
        [99, 59],
        [37, 96],
        [98, 79],
        [98, 95]],

       [[14, 16],
        [85, 15],
        [59, 52],
        [98, 93],
        [41, 76]]])
```

```
a5
```

```
array([[60, 49, 89, 63, 18],
       [87, 23, 96, 33, 34],
       [42, 93, 85, 25, 66],
       [37, 39, 67, 53, 19]])
```

```
a5[1,1]
```

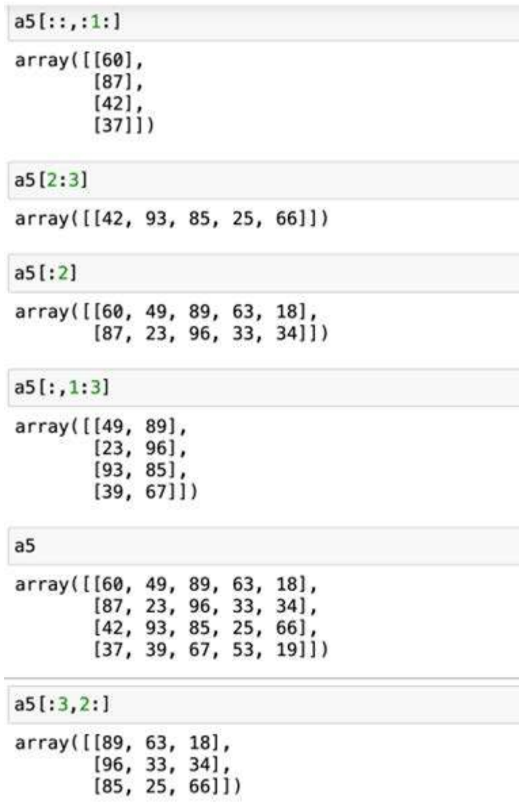
```
23
```

```
a5[0]
```

```
array([60, 49, 89, 63, 18])
```

```
a5[0:2]
```

```
array([[60, 49, 89, 63, 18],
       [87, 23, 96, 33, 34]])
```

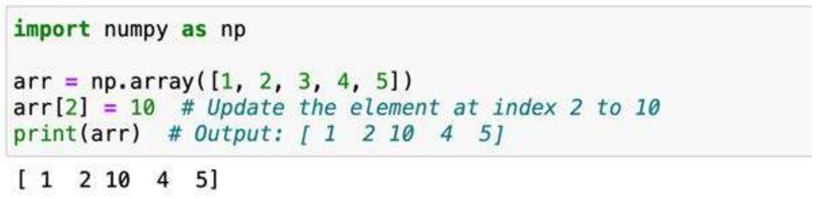


```
a5[::,:1:]

array([[60],
       [87],
       [42],
       [37]])

a5[2:3]

array([[42, 93, 85, 25, 66]])

a5[:2]

array([[60, 49, 89, 63, 18],
       [87, 23, 96, 33, 34]])

a5[:,1:3]

array([[49, 89],
       [23, 96],
       [93, 85],
       [39, 67]])

a5

array([[60, 49, 89, 63, 18],
       [87, 23, 96, 33, 34],
       [42, 93, 85, 25, 66],
       [37, 39, 67, 53, 19]])

a5[:3,2:]

array([[89, 63, 18],
       [96, 33, 34],
       [85, 25, 66]])
```

Update: Elements of a NumPy array can be up to advanced using indexing or utterly reducing. Simply assign new values to the favoured indices or slices!!

How to update?

To efficiently update factors in a NumPy array, you can assign brand new values to different elements or slices of the array at once. Here's how precisely you can do it:



```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
arr[2] = 10  # Update the element at index 2 to 10
print(arr)  # Output: [ 1  2 10  4  5]

[ 1  2 10  4  5]
```

Modifying individual elements: Here's a super cool trick—you can actually replace male or female elements of an array by specifying their indices. Isn't that mind-blowing?

```
a4
```

```
array([21, 11, 45, 38, 21, 70, 38, 85, 27, 73])
```

```
a4[1]=-1
a4
```

```
array([21, -1, 45, 38, 21, 70, 38, 85, 27, 73])
```

```
a4[:-4:-1]=0
a4
```

```
array([21, -1, 45, 38, 21, 70, 38,  0,  0,  0])
```

Updating Slices: This technique is the bee's knees! You can update quite a few factors by specifying a whopping slice of the array. Talk about efficiency, am I right?

```
arr[1:4] = 0  # Update elements from index 1 to 3 to 0
print(arr)  # Output: [1 0 0 0 5]
```

```
[1 0 0 0 5]
```

Broadcasting: Brace yourself, because broadcasting is the ultimate weapon in the arsenal of array updating! With this technique, you can update multiple elements simultaneously through broadcasting an array of mind-boggling values. It's like magic!

```
arr[1:4] = np.array([2, 3, 4])  # Update elements from index 1 to 3 to [2, 3, 4]
print(arr)  # Output: [1 2 3 4 5]
```

```
[1 2 3 4 5]
```

```
a4[:-4:-1]=[1,2,3] # factorization
```

```
a4
```

```
array([21, -1, 45, 38, 21, 70, 38,  3,  2,  1])
```

```
a5
```

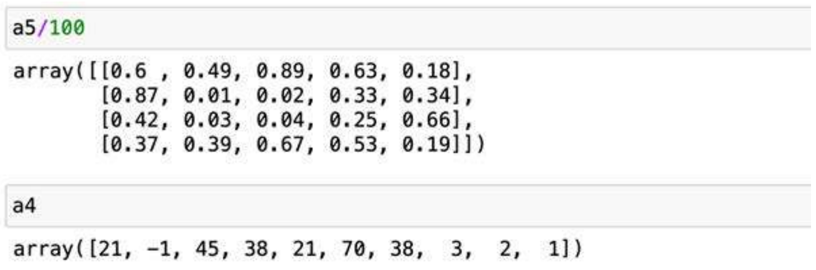
```
array([[60, 49, 89, 63, 18],
       [87, 23, 96, 33, 34],
       [42, 93, 85, 25, 66],
       [37, 39, 67, 53, 19]])
```

```
a5[1:3,1:3]=0
a5
```

```
array([[60, 49, 89, 63, 18],
       [87,  0,  0, 33, 34],
       [42,  0,  0, 25, 66],
       [37, 39, 67, 53, 19]])
```

```
# object is list, dic,tuple- np.array([1,2],[3,4])not work
np.array([[1,2],[3,4]])
```

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



```
a5/100
```

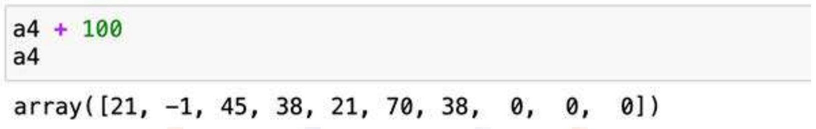
```
array([[0.6 , 0.49, 0.89, 0.63, 0.18],
       [0.87, 0.01, 0.02, 0.33, 0.34],
       [0.42, 0.03, 0.04, 0.25, 0.66],
       [0.37, 0.39, 0.67, 0.53, 0.19]])
```

```
a4
```

```
array([21, -1, 45, 38, 21, 70, 38,  3,  2,  1])
```

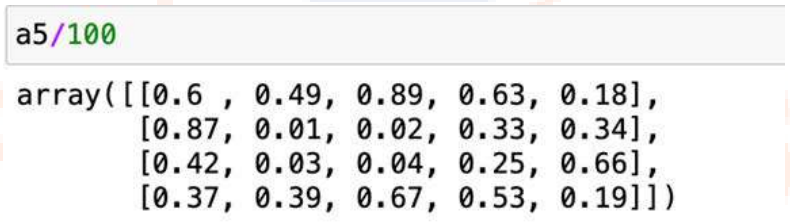
These techniques can help you efficaciously update the contents of a NumPy array according to your wide-ranging requirements. So, what are you waiting for? Go forth and conquer the world of array updating!

Mathematical Operations: NumPy arrays assist element-sensible mathematical operations, because of this that operations are applied to corresponding factors of arrays. This includes addition, subtraction, multiplication, division, exponentiation, rectangular root, and plenty of greater.



```
a4 + 100
a4
```

```
array([21, -1, 45, 38, 21, 70, 38,  0,  0,  0])
```



```
a5/100
```

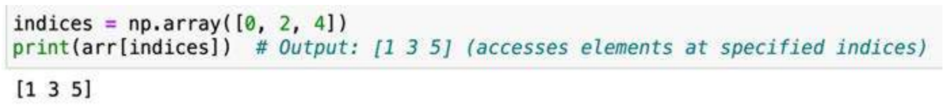
```
array([[0.6 , 0.49, 0.89, 0.63, 0.18],
       [0.87, 0.01, 0.02, 0.33, 0.34],
       [0.42, 0.03, 0.04, 0.25, 0.66],
       [0.37, 0.39, 0.67, 0.53, 0.19]])
```

Indexing: NumPy helps numerous indexing strategies:

Integer indexing: Use precise indices to get right of entry to factors.

Integer Array Indexing: You can use arrays of integers to index elements at once.

Using indexing strategies, you may successfully retrieve precise elements or subsets of factors from NumPy arrays to perform numerous operations and analyses.

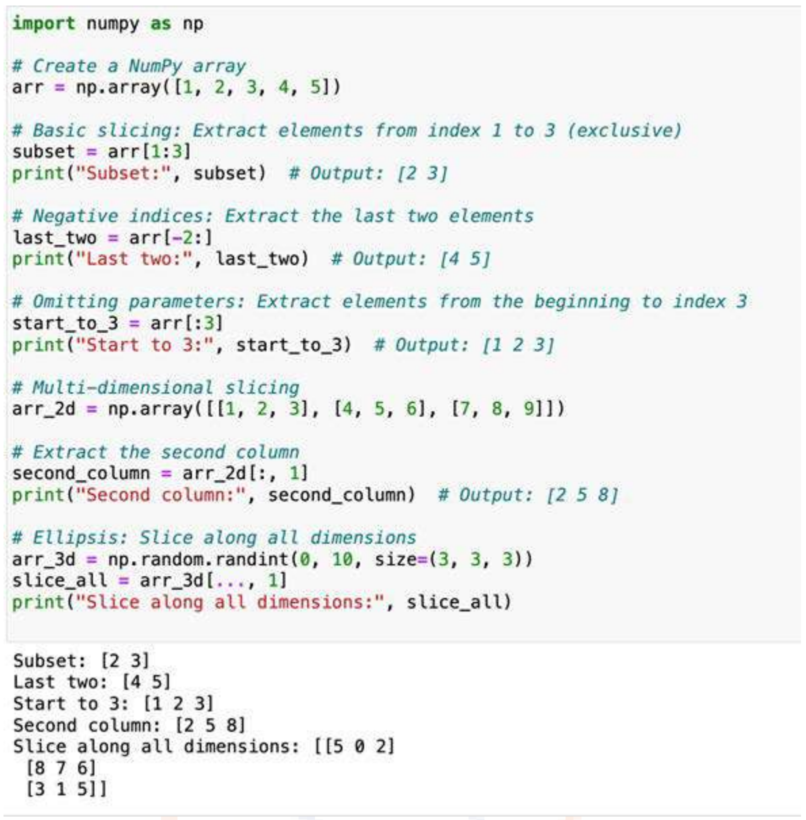


```
indices = np.array([0, 2, 4])
print(arr[indices])  # Output: [1 3 5] (accesses elements at specified indices)
```

```
[1 3 5]
```

Slicing: Extract a part of the array using begin, prevent, and step parameters.

Slicing in NumPy lets in you to extract elements of an array. Here's the way it works with code and motives:



```
import numpy as np

# Create a NumPy array
arr = np.array([1, 2, 3, 4, 5])

# Basic slicing: Extract elements from index 1 to 3 (exclusive)
subset = arr[1:3]
print("Subset:", subset)  # Output: [2 3]

# Negative indices: Extract the last two elements
last_two = arr[-2:]
print("Last two:", last_two)  # Output: [4 5]

# Omitting parameters: Extract elements from the beginning to index 3
start_to_3 = arr[:3]
print("Start to 3:", start_to_3)  # Output: [1 2 3]

# Multi-dimensional slicing
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Extract the second column
second_column = arr_2d[:, 1]
print("Second column:", second_column)  # Output: [2 5 8]

# Ellipsis: Slice along all dimensions
arr_3d = np.random.randint(0, 10, size=(3, 3, 3))
slice_all = arr_3d[..., 1]
print("Slice along all dimensions:", slice_all)
```

```
Subset: [2 3]
Last two: [4 5]
Start to 3: [1 2 3]
Second column: [2 5 8]
Slice along all dimensions: [[5 0 2]
 [8 7 6]
 [3 1 5]]
```

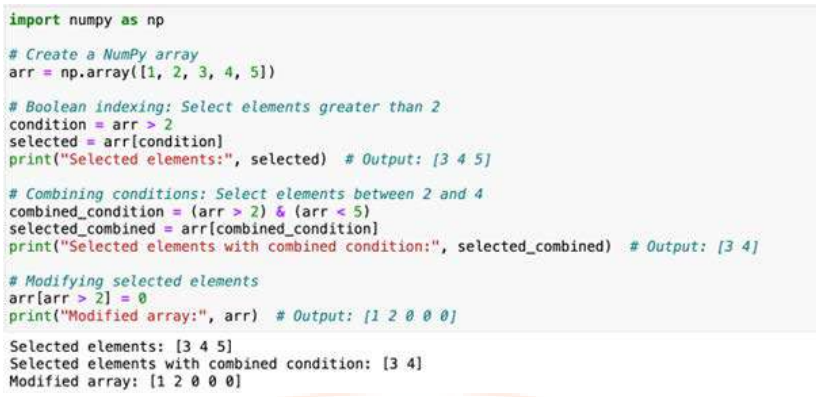
Explanation:

- ✓ Basic decreasing: Specify pretty a few indices (start:forestall:step) to extract elements.
- ✓ Negative indices: Use horrific numbers to rely backward from the cease of the array.
- ✓ Omitting parameters: If you bypass over a parameter, it defaults to a certain value.
- ✓ Multi-dimensional lowering: Slice along extraordinary dimensions of multi-dimensional arrays.
- ✓ Ellipsis (...): Represents multiple colons for reducing multi-dimensional arrays

- **Boolean indexing:** Use Boolean arrays to filter factors based on a situation.

Boolean indexing: Create a Boolean array primarily based totally on a situation and use it to pick out factors from the unique array.

- **Combining situations:** Utilize logical & operators to mix multiple scenarios.
- **Modifying based on factors:** Boolean indexing also can be employed to adjust decided on factors inside the array.

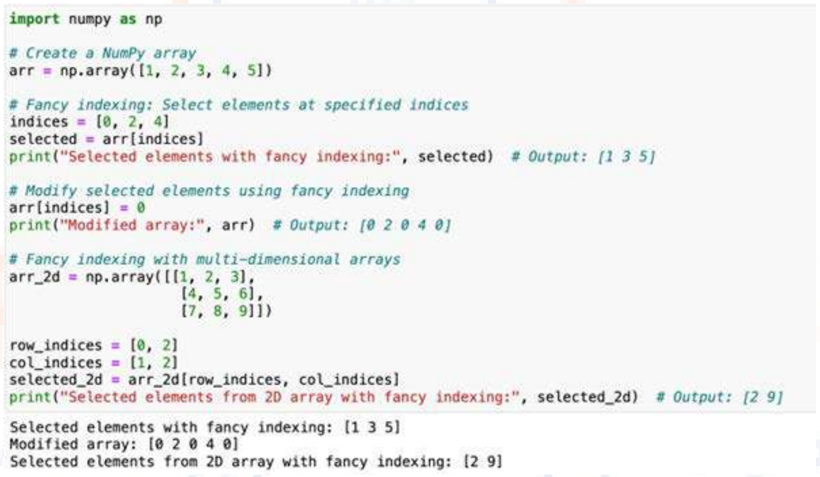


```
import numpy as np

# Create a NumPy array
arr = np.array([1, 2, 3, 4, 5])

# Boolean indexing: Select elements greater than 2
condition = arr > 2
selected = arr[condition]
print("Selected elements:", selected)  # Output: [3 4 5]

# Combining conditions: Select elements between 2 and 4
combined_condition = (arr > 2) & (arr < 5)
selected_combined = arr[combined_condition]
print("Selected elements with combined condition:", selected_combined)  # Output: [3 4]

# Modifying selected elements
arr[arr > 2] = 0
print("Modified array:", arr)  # Output: [1 2 0 0 0]
```

```
Selected elements: [3 4 5]
Selected elements with combined condition: [3 4]
Modified array: [1 2 0 0 0]
```

Fancy indexing: Employ arrays of indices to concurrently get entry to multiple elements.

Fancy indexing in NumPy is the method of indexing an array the use of arrays of indices or slices. Rather than using person indices or slices, you can make use of arrays of indices to immediately access or modify a set of elements. Here's the way it features, at the side of code examples and motives:



```
import numpy as np

# Create a NumPy array
arr = np.array([1, 2, 3, 4, 5])

# Fancy indexing: Select elements at specified indices
indices = [0, 2, 4]
selected = arr[indices]
print("Selected elements with fancy indexing:", selected)  # Output: [1 3 5]

# Modify selected elements using fancy indexing
arr[indices] = 0
print("Modified array:", arr)  # Output: [0 2 0 4 0]

# Fancy indexing with multi-dimensional arrays
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

row_indices = [0, 2]
col_indices = [1, 2]
selected_2d = arr_2d[row_indices, col_indices]
print("Selected elements from 2D array with fancy indexing:", selected_2d)  # Output: [2 9]
```

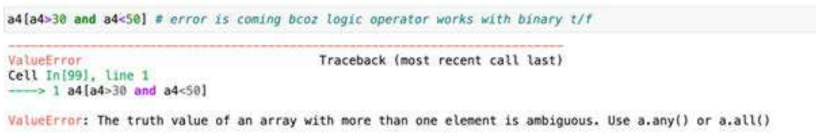
```
Selected elements with fancy indexing: [1 3 5]
Modified array: [0 2 0 4 0]
Selected elements from 2D array with fancy indexing: [2 9]
```

Explanation:

- Fancy indexing enables you to specify arrays of indices to pick or modify elements from an array.
- It offers flexibility in getting access to factors based on non-contiguous indices or styles.
- Fancy indexing may be applied with multi-dimensional arrays by using specifying separate arrays of row and column indices.

Overall, NumPy affords powerful abilities for developing, gaining access to, updating, appearing mathematical operations on, and indexing arrays correctly. Its bureaucracy the foundation for numerical computing in Python.

Note:



```
a4[a4>30 and a4<50] # error is coming bcoz logic operator works with binary t/f

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[99], line 1
----> 1 a4[a4>30 and a4<50]

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

How to Solve a ValueError Related to Boolean Indexing

To effectively address a ValueError that emerges from Boolean indexing, it is crucial to utilize the & operator. This is because the NumPy Boolean feature necessitates the use of bitwise operators. The following steps outline the necessary to rectify this issue:

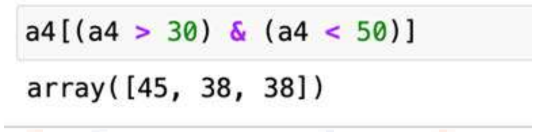
- Begin identifying the associated with Boolean indexing.

Employ the and to rectify this error.

- Make of the & operator lieu of other Boolean operators mandated by the Num Boolean feature.

Ensure that the modified code the original meaning and, enabling efficient resolution of problem.

By diligentlying to these steps one can successfully resolve the ValueError during Boolean indexing.



```
a4[(a4 > 30) & (a4 < 50)]
```

```
array([45, 38, 38])
```

Bitwise operators

Bitwise Functions in Python A Guide to Imperfection!

Bitwise operators Python are, like, exceptional cool, you recognize? They are definitely used to perform a few wicked operations on binary numbers, dude! So, allow's dive into the gnarly international of bitwise functions, guy!

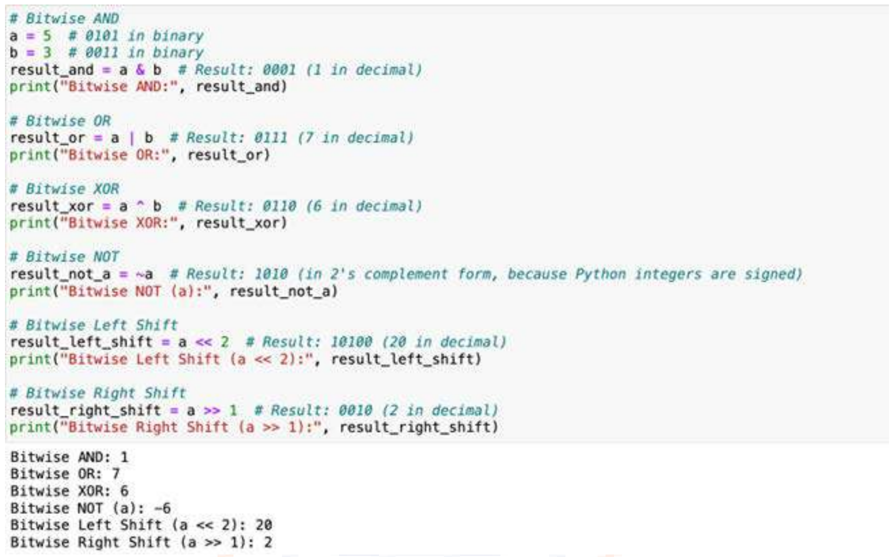
AND (&): So, like, this groovy operator sets every bit to, like, completely 1 if both bits are 1s, guy. It's all approximately that unity?

OR (|): Now, this operator is, like, excellent! It sets every bit to one if one of the bits is 1. It's all approximately embracing the oneness, dude!

XOR (^): This operator is like, completely radical! Its units each bit to 1 if there's, like, most effective one 1 in the bit. It's all about that specialty, bro!

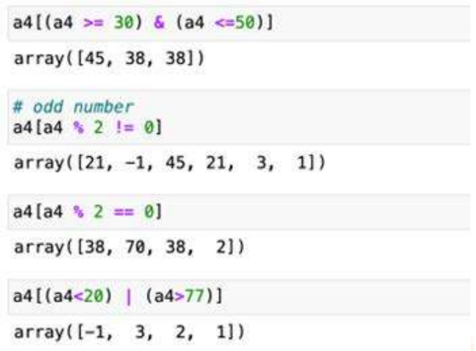
NOT (~): Okay, that is wild! This operator switches the bits, man. Like, completely flipping them! 1 becomes 0 and zero will become 1. It's like taking a experience to the turn side, dude!

So, it is the lowdown on bitwise functions in Python. It's all about rocking the world of binary numbers and embracing those bitwise imperfections, guy! Keep coding, and may the bits be with you! Groovy, right?



```
# Bitwise AND
a = 5  # 0101 in binary
b = 3  # 0011 in binary
result_and = a & b  # Result: 0001 (1 in decimal)
print("Bitwise AND:", result_and)

# Bitwise OR
result_or = a | b  # Result: 0111 (7 in decimal)
print("Bitwise OR:", result_or)

# Bitwise XOR
result_xor = a ^ b  # Result: 0110 (6 in decimal)
print("Bitwise XOR:", result_xor)

# Bitwise NOT
result_not_a = ~a  # Result: 1010 (in 2's complement form, because Python integers are signed)
print("Bitwise NOT (a):", result_not_a)

# Bitwise Left Shift
result_left_shift = a << 2  # Result: 10100 (20 in decimal)
print("Bitwise Left Shift (a << 2):", result_left_shift)

# Bitwise Right Shift
result_right_shift = a >> 1  # Result: 0010 (2 in decimal)
print("Bitwise Right Shift (a >> 1):", result_right_shift)
```

```
Bitwise AND: 1
Bitwise OR: 7
Bitwise XOR: 6
Bitwise NOT (a): -6
Bitwise Left Shift (a << 2): 20
Bitwise Right Shift (a >> 1): 2
```



```
a4[(a4 >= 30) & (a4 <=50)]
```

```
array([45, 38, 38])
```

```
# odd number
a4[a4 % 2 != 0]
```

```
array([21, -1, 45, 21,  3,  1])
```

```
a4[a4 % 2 == 0]
```

```
array([38, 70, 38,  2])
```

```
a4[(a4<20) | (a4>77)]
```

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

BRAINALYST

Left Shift (<<) and Right Shift (>>)Shifts for Bit Manipulation!

Left Shift (<<) is a effective operator that does extra than simply transferring bits. It is going beyond! It intelligently moves the bit positions to the left, including a zero to the proper. This tiny 0 is like a cherry on top, making the entire operation sweeter!

Right Shift (>>) isn't any exclusive! It is aware of how to manage bits and manipulate them like a seasoned. It shifts the ones bits to the right, but wait, it would not forestall there! It additionally provides zeros to the left for effective numbers, showcasing its generosity. And for terrible numbers, it is all approximately maintaining the ones precious signal bits! Ain't that cool?

Low-Level Programming's Best Friends!

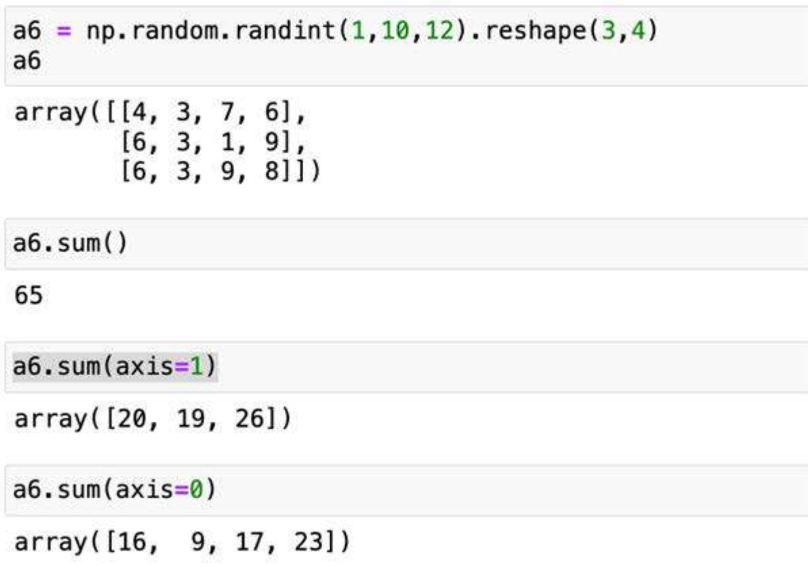
These operators, my friend, are not your everyday amateurs. They are the rockstars of low-degree programming. They are the ones you name upon whilst handling hardware registers, optimizing code, and performing thoughts-blowing bitwise mathematics. They are the heroes that make the magic appear!

NumPy's Secret Weapon!

Hey, have you met NumPy? It's a grand library that is aware of a way to handle arrays fashion. And guess what? It's got this delightful trick up its sleeve - normally makes use of left and proper shift operators for green array processing and masking. Boom It's like a mystery weapon that brings the electricity of bit manipulation to the realm of arrays! Impressive, proper?

Descriptive techniques

Descriptive techniques in facts are techniques used to summarize and describe the primary features of a dataset. They provide insights into the imperative tendency, dispersion, and form of the fact's distribution. Common descriptive statistics include measures such as mean, median, mode, well-known deviation, variance, range, and percentiles. These strategies help in information the fundamental houses of the information without making any inferences past the dataset itself.



```
a6 = np.random.randint(1,10,12).reshape(3,4)
a6
```

```
array([[4, 3, 7, 6],
       [6, 3, 1, 9],
       [6, 3, 9, 8]])
```

```
a6.sum()
```

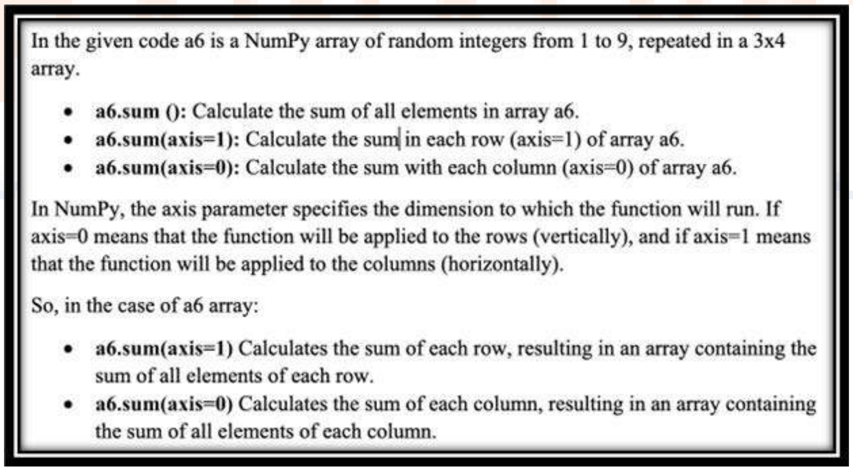
```
65
```

```
a6.sum(axis=1)
```

```
array([20, 19, 26])
```

```
a6.sum(axis=0)
```

```
array([16,  9, 17, 23])
```



In the given code a6 is a NumPy array of random integers from 1 to 9, repeated in a 3x4 array.

- **a6.sum ():** Calculate the sum of all elements in array a6.
- **a6.sum(axis=1):** Calculate the sum in each row (axis=1) of array a6.
- **a6.sum(axis=0):** Calculate the sum with each column (axis=0) of array a6.

In NumPy, the axis parameter specifies the dimension to which the function will run. If axis=0 means that the function will be applied to the rows (vertically), and if axis=1 means that the function will be applied to the columns (horizontally).

So, in the case of a6 array:

- **a6.sum(axis=1)** Calculates the sum of each row, resulting in an array containing the sum of all elements of each row.
- **a6.sum(axis=0)** Calculates the sum of each column, resulting in an array containing the sum of all elements of each column.

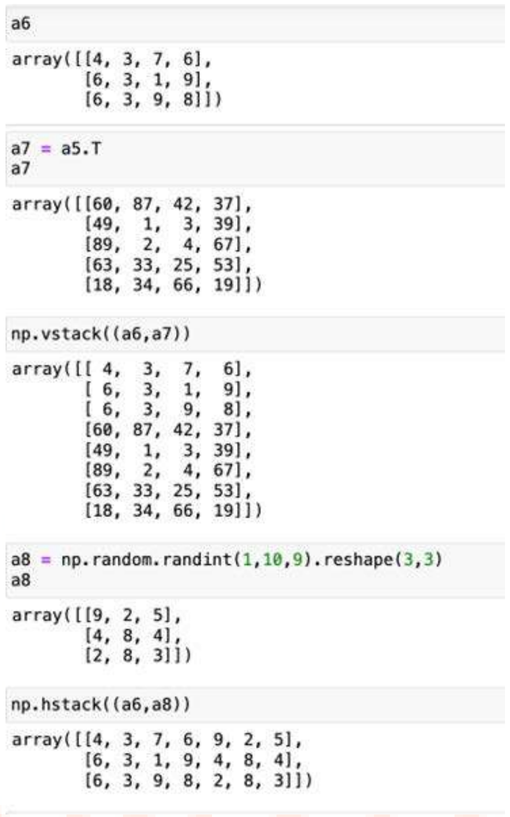
Combining Dataset

Combining Datasets In NumPy, documents entities may be stacked processes like np.vstack() and np.hstack().

np.vstack(): Stacks the array vertically row-smart), this is, it provides the rows of 1 array under the rows of another array, growing a modern-day array. Requires stacked arrays to have the equal range of columns.

np.hstack(): Stacks the array horizontally (column-smart), that is, provides the columns of one array to the columns of every other array, developing a cutting-edge array. Requires stacked arrays to have the identical variety of rows.

This feature is appreciably useful at the same time as you need to enroll in statistics saved in different arrays via rows or columns.

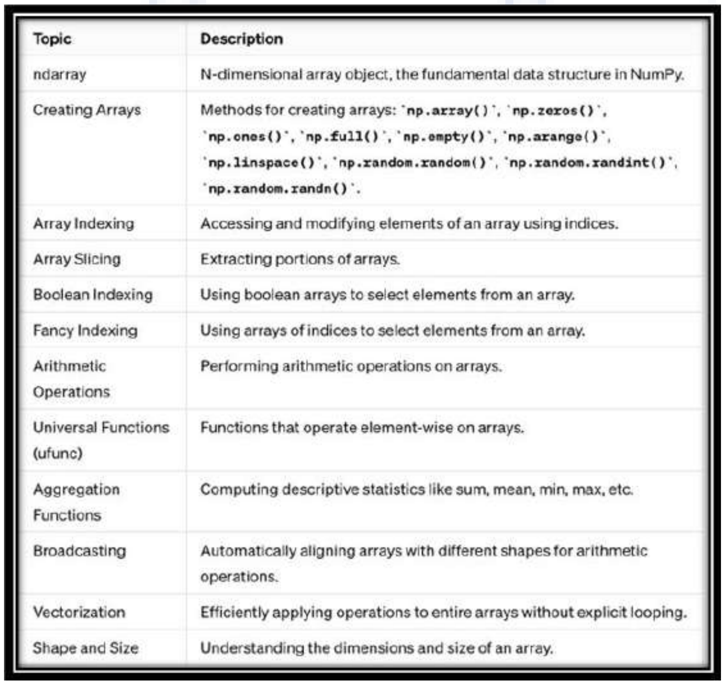


```
a6
```
```
array([[4, 3, 7, 6],
       [6, 3, 1, 9],
       [6, 3, 9, 8]])
```
```
a7 = a5.T
a7
```
```
array([[60, 87, 42, 37],
       [49,  1,  3, 39],
       [89,  2,  4, 67],
       [63, 33, 25, 53],
       [18, 34, 66, 19]])
```
```
np.vstack((a6,a7))
```
```
array([[ 4,  3,  7,  6],
       [ 6,  3,  1,  9],
       [ 6,  3,  9,  8],
       [60, 87, 42, 37],
       [49,  1,  3, 39],
       [89,  2,  4, 67],
       [63, 33, 25, 53],
       [18, 34, 66, 19]])
```
```
a8 = np.random.randint(1,10,9).reshape(3,3)
a8
```
```
array([[9, 2, 5],
       [4, 8, 4],
       [2, 8, 3]])
```
```
np.hstack((a6,a8))
```
```
array([[4, 3, 7, 6, 9, 2, 5],
       [6, 3, 1, 9, 4, 8, 4],
       [6, 3, 9, 8, 2, 8, 3]])
```

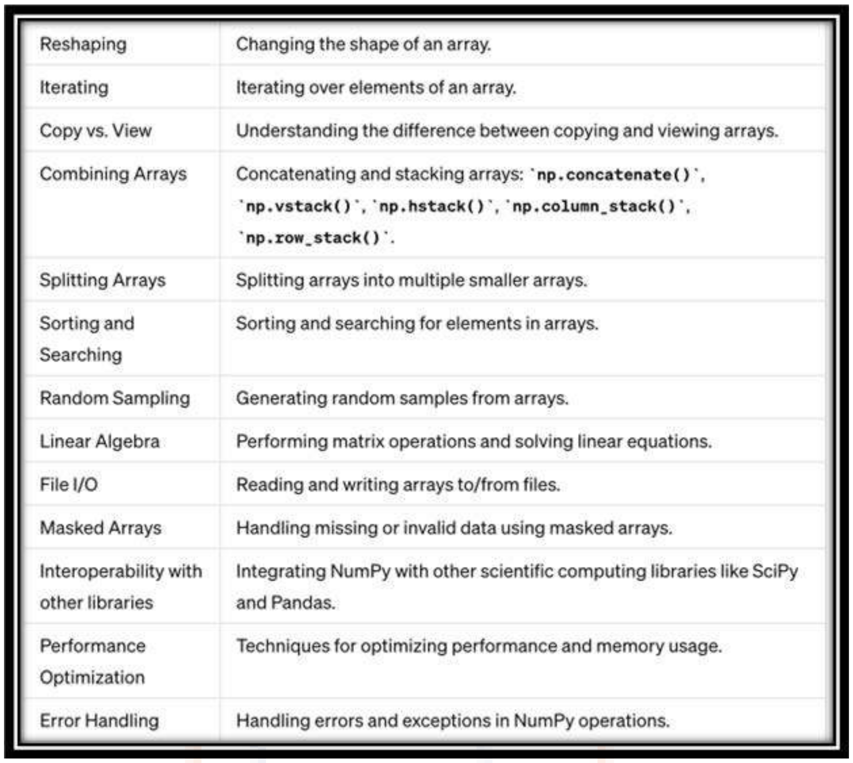
BRAINALYST
A Data Driven Company

Conclusion:

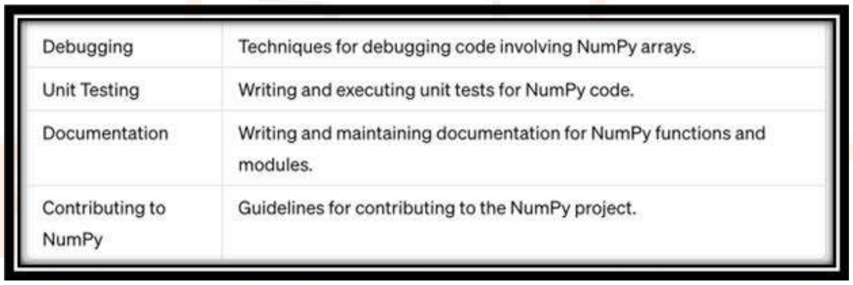
This table covers a full-size range of topics associated with NumPy, from primary array advent and manipulation to, extra quiet, subjects like linear algebra, record I/O, and popular overall performance optimization!



Topic	Description
ndarray	N-dimensional array object, the fundamental data structure in NumPy.
Creating Arrays	Methods for creating arrays: `np.array()`, `np.zeros()`, `np.ones()`, `np.full()`, `np.empty()`, `np.arange()`, `np.linspace()`, `np.random.random()`, `np.random.randint()`, `np.random.randn()`.
Array Indexing	Accessing and modifying elements of an array using indices.
Array Slicing	Extracting portions of arrays.
Boolean Indexing	Using boolean arrays to select elements from an array.
Fancy Indexing	Using arrays of indices to select elements from an array.
Arithmetic Operations	Performing arithmetic operations on arrays.
Universal Functions (ufunc)	Functions that operate element-wise on arrays.
Aggregation Functions	Computing descriptive statistics like sum, mean, min, max, etc.
Broadcasting	Automatically aligning arrays with different shapes for arithmetic operations.
Vectorization	Efficiently applying operations to entire arrays without explicit looping.
Shape and Size	Understanding the dimensions and size of an array.



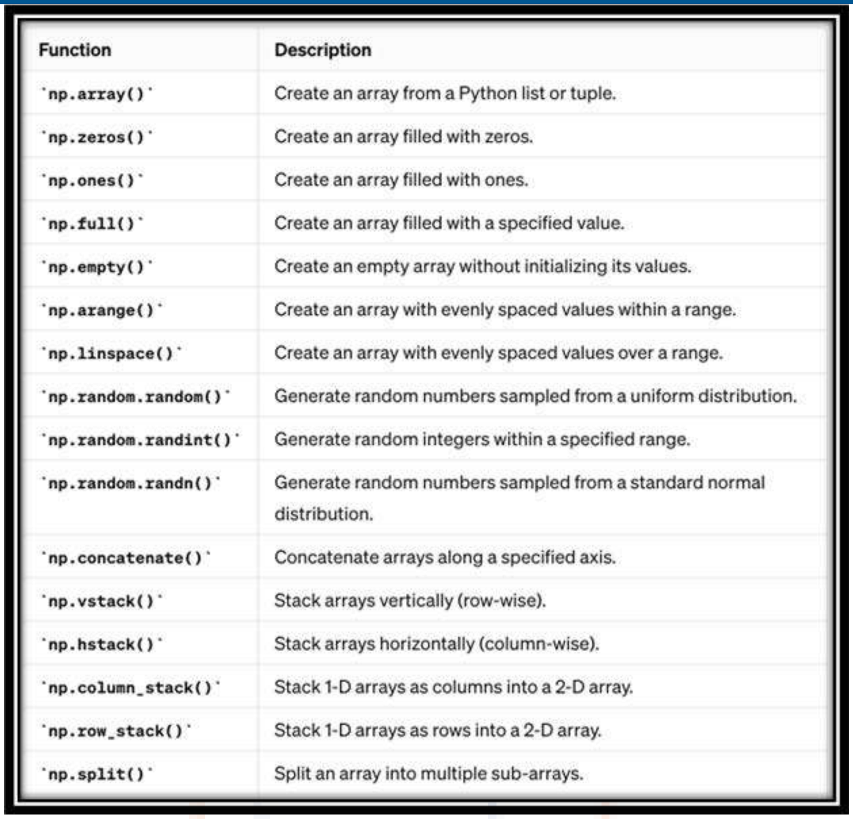
Reshaping	Changing the shape of an array.
Iterating	Iterating over elements of an array.
Copy vs. View	Understanding the difference between copying and viewing arrays.
Combining Arrays	Concatenating and stacking arrays: `np.concatenate()`, `np.vstack()`, `np.hstack()`, `np.column_stack()`, `np.row_stack()`.
Splitting Arrays	Splitting arrays into multiple smaller arrays.
Sorting and Searching	Sorting and searching for elements in arrays.
Random Sampling	Generating random samples from arrays.
Linear Algebra	Performing matrix operations and solving linear equations.
File I/O	Reading and writing arrays to/from files.
Masked Arrays	Handling missing or invalid data using masked arrays.
Interoperability with other libraries	Integrating NumPy with other scientific computing libraries like SciPy and Pandas.
Performance Optimization	Techniques for optimizing performance and memory usage.
Error Handling	Handling errors and exceptions in NumPy operations.



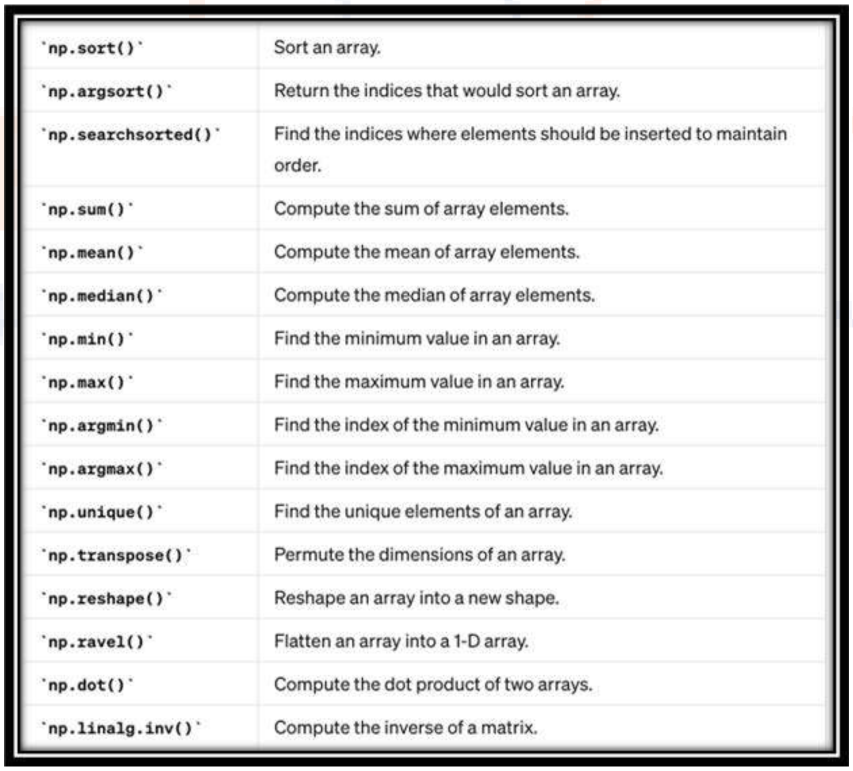
Debugging	Techniques for debugging code involving NumPy arrays.
Unit Testing	Writing and executing unit tests for NumPy code.
Documentation	Writing and maintaining documentation for NumPy functions and modules.
Contributing to NumPy	Guidelines for contributing to the NumPy project.

These are only a few examples of the various capabilities to be had in NumPy for array manipulation, mathematical operations, and linear algebra.

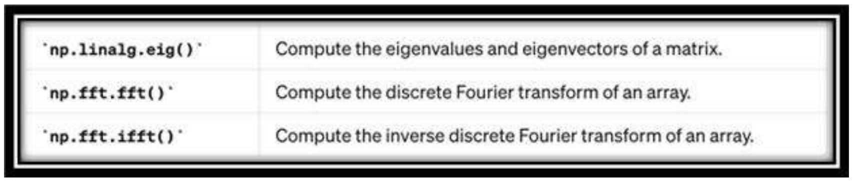
Brainalyst © 2021-2024



Function	Description
`np.array()`	Create an array from a Python list or tuple.
`np.zeros()`	Create an array filled with zeros.
`np.ones()`	Create an array filled with ones.
`np.full()`	Create an array filled with a specified value.
`np.empty()`	Create an empty array without initializing its values.
`np.arange()`	Create an array with evenly spaced values within a range.
`np.linspace()`	Create an array with evenly spaced values over a range.
`np.random.random()`	Generate random numbers sampled from a uniform distribution.
`np.random.randint()`	Generate random integers within a specified range.
`np.random.randn()`	Generate random numbers sampled from a standard normal distribution.
`np.concatenate()`	Concatenate arrays along a specified axis.
`np.vstack()`	Stack arrays vertically (row-wise).
`np.hstack()`	Stack arrays horizontally (column-wise).
`np.column_stack()`	Stack 1-D arrays as columns into a 2-D array.
`np.row_stack()`	Stack 1-D arrays as rows into a 2-D array.
`np.split()`	Split an array into multiple sub-arrays.



`np.sort()`	Sort an array.
`np.argsort()`	Return the indices that would sort an array.
`np.searchsorted()`	Find the indices where elements should be inserted to maintain order.
`np.sum()`	Compute the sum of array elements.
`np.mean()`	Compute the mean of array elements.
`np.median()`	Compute the median of array elements.
`np.min()`	Find the minimum value in an array.
`np.max()`	Find the maximum value in an array.
`np.argmin()`	Find the index of the minimum value in an array.
`np.argmax()`	Find the index of the maximum value in an array.
`np.unique()`	Find the unique elements of an array.
`np.transpose()`	Permute the dimensions of an array.
`np.reshape()`	Reshape an array into a new shape.
`np.ravel()`	Flatten an array into a 1-D array.
`np.dot()`	Compute the dot product of two arrays.
`np.linalg.inv()`	Compute the inverse of a matrix.



`np.linalg.eig()`	Compute the eigenvalues and eigenvectors of a matrix.
`np.fft.fft()`	Compute the discrete Fourier transform of an array.
`np.fft.ifft()`	Compute the inverse discrete Fourier transform of an array.

END