NumPy, short for Numerical Python.

NumPy provides an easy-to-use C API, it is straightforward to pass data to external libraries written in a low-level language and also for external libraries to return data to Python as NumPy arrays. This feature has made Python a language of choice for wrapping legacy C/C++/Fortran codebases and giving them a dynamic and easy-to-use interface.

Data munging, sometimes called data wrangling or data cleaning, is converting and mapping unprocessed data into a different format to improve its suitability and value for various downstream uses, including analytics.

pandas also provides some more domain specific functionality like time series manipulation, which is not present in NumPy.

NumPy internally stores data in a contiguous block of memory, independent of other built-in Python objects. NumPy's library of algorithms written in the C language can operate on this memory without any type checking or other overhead. NumPy arrays also use much less memory than built-in Python sequences.

In [1]:
```
import numpy as np
numpy_array=np.arange(1000000)
python_list=list(range(1000000))
```

In [2]:
```
%time for _ in range(10): numpy_array2 = numpy_array *2
%time for _ in range(10): python_list2 = [x * 2 for x in python_list]
```

```
CPU times: user 26 ms, sys: 18.8 ms, total: 44.8 ms
Wall time: 162 ms
CPU times: user 1.08 s, sys: 252 ms, total: 1.33 s
Wall time: 1.97 s
```

NumPy-based algorithms are generally 10 to 100 times faster (or more) than their pure Python counterparts and use significantly less memory.

One of the key features of NumPy is its N-dimensional array object, or ndarray, which is a fast, flexible container for large datasets in Python

In [3]:
```
d=np.random.randn(2,3)
d
```

Out[3]:
```
array([[-2.05724633,  0.76994867, -0.45824433],
       [ 1.04091997,  1.61590441,  0.82462728]])
```

In [4]:
```
d+d
```

Out[4]:
```
array([[-4.11449265,  1.53989734, -0.91648865],
       [ 2.08183994,  3.23180882,  1.64925456]])
```

In [5]:
```
d*d
```

Out[5]:
```
array([[4.23226245, 0.59282095, 0.20998786],
       [1.08351438, 2.61114706, 0.68001015]])
```

In [6]:
```
10*d
```

Out[6]:
```
array([[-20.57246326,   7.6994867 ,  -4.58244325],
       [ 10.40919969,  16.15904409,   8.24627278]])
```

You are, of course, welcome to put

```
 from numpy import *
```

in your code to avoid having to write np., but I advise against making a habit of this. The numpy namespace is large and contains a number of functions whose names conflict with built-in Python functions (like min and max).

An ndarray is a generic multidimensional container for homogeneous data; that is, all of the elements must be the same type.
Every array has a shape, a tuple indicating the size of each dimension, and a dtype, an object describing the data type of the array

In [7]:
```
d.dtype
```

Out[7]:
```
dtype('float64')
```

In [8]:
```
d.ndim
```

Out[8]:
```
2
```

In [9]:
```
d.shape
```

Out[9]:
```
(2, 3)
```

The easiest way to create an array is to use the array function. This accepts any sequence-like object (including other arrays) and produces a new NumPy array containing the passed data. For example, a list is a good candidate for conversion

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

```
In [11]: a.shape
```

```
Out[11]: (2, 2, 2)
```

```
In [12]: a.ndim
```

```
Out[12]: 3
```

```
In [13]: a.dtype
```

```
Out[13]: dtype('int64')
```

```
In [14]: # dimension 3 means array is divided into 3 parts: no of grids, no of rows in each grid and
         # no of columns in each grid
```

```
In [15]: np.array([1,2,3], dtype='int16')
```

```
Out[15]: array([1, 2, 3], dtype=int16)
```

```
In [16]: np.empty((2,3))
```

```
Out[16]: array([[20.57246326,  7.6994867 ,  4.58244325],
                [10.40919969, 16.15904409,  8.24627278]])
```

```
In [17]: np.zeros((2,3,4))
```

```
Out[17]: 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.]]])
```

```
In [18]: np.zeros((10,))
```

```
Out[18]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
In [19]: np.ones((2,3))
```

```
Out[19]: array([[1., 1., 1.],
                [1., 1., 1.]])
```

```
In [20]: np.identity(3)
```

```
Out[20]: array([[1., 0., 0.],
                [0., 1., 0.],
                [0., 0., 1.]])
```

asarray ==> Convert input to ndarray, but do not copy if the input is already an ndarray

```
In [21]: np.asarray?
```

```
Docstring:
asarray(a, dtype=None, order=None, *, device=None, copy=None, like=None)

Convert the input to an array.

Parameters
----------
a : array_like
    Input data, in any form that can be converted to an array.  This
    includes lists, lists of tuples, tuples, tuples of tuples, tuples
    of lists and ndarrays.
dtype : data-type, optional
    By default, the data-type is inferred from the input data.
order : {'C', 'F', 'A', 'K'}, optional
    Memory layout.  'A' and 'K' depend on the order of input array a.
    'C' row-major (C-style),
    'F' column-major (Fortran-style) memory representation.
    'A' (any) means 'F' if `a` is Fortran contiguous, 'C' otherwise
    'K' (keep) preserve input order
    Defaults to 'K'.
device : str, optional
    The device on which to place the created array. Default: ``None``.
    For Array-API interoperability only, so must be ``"cpu"`` if passed.

    .. versionadded:: 2.0.0
```

```
copy : bool, optional
    If ``True``, then the object is copied. If ``None`` then the object is
    copied only if needed, i.e. if ``__array__`` returns a copy, if obj
    is a nested sequence, or if a copy is needed to satisfy any of
    the other requirements (``dtype``, ``order``, etc.).
    For ``False`` it raises a ``ValueError`` if a copy cannot be avoided.
    Default: ``None``.

    .. versionadded:: 2.0.0
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 interpretation of ``a``.  No copy is performed if the input
    is already an ndarray with matching dtype and order.  If ``a`` is a
    subclass of ndarray, a base class ndarray is returned.

See Also
--------
asanyarray : Similar function which passes through subclasses.
ascontiguousarray : Convert input to a contiguous array.
asfortranarray : Convert input to an ndarray with column-major
                 memory order.
asarray_chkfinite : Similar function which checks input for NaNs and Infs.
fromiter : Create an array from an iterator.
fromfunction : Construct an array by executing a function on grid
               positions.

Examples
--------
Convert a list into an array:

>>> a = [1, 2]
>>> import numpy as np
>>> np.asarray(a)
array([1, 2])

Existing arrays are not copied:

>>> a = np.array([1, 2])
>>> np.asarray(a) is a
True

If `dtype` is set, array is copied only if dtype does not match:

>>> a = np.array([1, 2], dtype=np.float32)
>>> np.shares_memory(np.asarray(a, dtype=np.float32), a)
True
>>> np.shares_memory(np.asarray(a, dtype=np.float64), a)
False

Contrary to `asanyarray`, ndarray subclasses are not passed through:

>>> issubclass(np.recarray, np.ndarray)
True
>>> a = np.array([(1., 2), (3., 4)], dtype='f4,i4').view(np.recarray)
>>> np.asarray(a) is a
False
>>> np.asanyarray(a) is a
True
Type:      builtin_function_or_method
```

In [22]:
```python
a=(1,2)
np.asarray(a)
```

Out[22]:
```
array([1, 2])
```

The data type or dtype is a special object containing the information (or metadata, data about data) the ndarray needs to interpret a chunk of memory as a particular type of data

In [23]:
```python
arr1 = np.array([1, 2, 3], dtype=np.float64)
arr1
```

Out[23]:
```
array([1., 2., 3.])
```

In [24]:
```python
arr1 = np.array([1, 2, 3], dtype=np.str_)
arr1
```

Out[24]:
```
array(['1', '2', '3'], dtype='<U1')
```

int32, uint32 ==> i4, u4 ==> Signed and unsigned 32-bit integer types
float16 ==> f2 ==> Half-precision floating point
float32 ==> f4 or f ==> Standard single-precision floating point; compatible with C float
float64 ==> f8 or d ==> Standard double-precision floating point; compatible with C double
object ==> O ==> Python object type; a value can be any Python object
unicode ==> U ==> *Fixed-length Unicode type (number of bytes platform specific); same specification semantics as string* (e.g., 'U10')

In [25]:
```
# You can explicitly convert or cast an array from one dtype to another using ndarray’s astype method:

np.array([1,2,3]).astype(np.float32)
```

Out[25]:
```
array([1., 2., 3.], dtype=float32)
```

In [26]:
```
np.empty(8, dtype='u4')
```

Out[26]:
```
array([583855524,         0,         0,         0, 576066000,         0,
       583985056,         0], dtype=uint32)
```

Arrays are important because they enable you to express batch operations on data without writing any for loops. NumPy users call this vectorization. Any arithmetic operations between equal-size arrays applies the operation element-wise.

In [27]:
```
arr=np.array([1,2,3],dtype=np.float16)
print(arr*arr,arr**3,1/arr)
```

```
[1. 4. 9.] [ 1.  8. 27.] [1.     0.5    0.3333]
```

In [28]:
```
# array comparison

arr1=np.array([3,1,2])
arr2=np.arange(3)
print(arr1,arr2,arr1>arr2)
```

```
[3 1 2] [0 1 2] [ True False False]
```

In [29]:
```
arr=np.arange(12)
print(arr,arr[5],arr[3:7])
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11] 5 [3 4 5 6]
```

In [30]:
```
arr[8:11]=arr[3:6]
print(arr)
```

```
[ 0  1  2  3  4  5  6  7  3  4  5 11]
```

In [31]:
```
arr3d=np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
arr2d=arr3d[0]
print(arr3d,'\n\n',arr2d,arr2d.ndim)
```

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

 [[5 6]
  [7 8]]]

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

In [32]:
```
print(arr2d[:1,:])
```

```
[[1 2]]
```

In [33]:
```
arr2d[:1,:]=0
print(arr2d)
```

```
[[0 0]
 [3 4]]
```

Like arithmetic operations, comparisons (such as ==) with arrays are also vectorized

In [34]:
```
names=np.array(['ankit','kiio','summi','soumi','kiio','ankit','summi'])
```

In [35]:
```
names=='kiio'
```

Out[35]:
```
array([False,  True, False, False,  True, False, False])
```

In [36]:
```
data=np.random.randn(7,5)
```

In [37]:
```
data
```

```
Out[37]: array([[ 0.47246008, -0.97230505, -0.88185117,  1.026626  ,  0.32148013],
       [-0.47270189, -1.87421897, -1.7251059 ,  1.07963034,  0.20243205],
       [ 0.81433844,  0.77974422,  1.14522355, -0.88763845, -1.05345061],
       [-2.32515631,  0.25949492, -0.44071896,  2.79388335,  0.23144792],
       [-0.35663242,  1.48296132,  0.16217465,  1.27015427,  0.65339976],
       [ 0.20411933,  0.46806683,  1.18465655, -1.18838839, -0.69035039],
       [-2.14149743, -0.81589488,  0.04407909,  1.01297928, -0.81875806]])
```

```
In [38]: mask=(names=='kiio') | (names=='soumi')
mask
```

```
Out[38]: array([False,  True, False,  True,  True, False, False])
```

```
In [39]: data[mask]
```

```
Out[39]: array([[-0.47270189, -1.87421897, -1.7251059 ,  1.07963034,  0.20243205],
       [-2.32515631,  0.25949492, -0.44071896,  2.79388335,  0.23144792],
       [-0.35663242,  1.48296132,  0.16217465,  1.27015427,  0.65339976]])
```

Fancy Indexing: To select out a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order

```
In [40]: arr=np.empty((2,3))
```

```
In [41]: arr
```

```
Out[41]: array([[1., 1., 1.],
       [1., 1., 1.]])
```

```
In [42]: arr[[1,0]]
```

```
Out[42]: array([[1., 1., 1.],
       [1., 1., 1.]])
```

```
In [43]: arr[1:,[1,0,2]]
```

```
Out[43]: array([[1., 1., 1.]])
```

```
In [44]: arr = np.arange(16).reshape((4, 4))
print(arr)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

```
In [45]: print(arr[[2,0],[1,3]]) # elements at position (2,1) and (0,3) will be selected but why ?
```

```
[9 3]
```

```
In [46]: # Transposing Arrays and Swapping Axes
arr=np.array([[1,2],[3,4]])
arr
```

```
Out[46]: array([[1, 2],
       [3, 4]])
```

```
In [47]: arr.T
```

```
Out[47]: array([[1, 3],
       [2, 4]])
```

```
In [48]: arr.transpose()
```

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

```
In [49]: arr = np.arange(16).reshape((2, 2, 4))
arr
```

```
Out[49]: array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],

       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
```

```
In [50]: arr.transpose?
```

```
Docstring:
a.transpose(*axes)

Returns a view of the array with axes transposed.

Refer to `numpy.transpose` for full documentation.

Parameters
----------
axes : None, tuple of ints, or `n` ints

 * None or no argument: reverses the order of the axes.

 * tuple of ints: `i` in the `j`-th place in the tuple means that the
   array's `i`-th axis becomes the transposed array's `j`-th axis.

 * `n` ints: same as an n-tuple of the same ints (this form is
   intended simply as a "convenience" alternative to the tuple form).

Returns
-------
p : ndarray
    View of the array with its axes suitably permuted.

See Also
--------
transpose : Equivalent function.
ndarray.T : Array property returning the array transposed.
ndarray.reshape : Give a new shape to an array without changing its data.

Examples
--------
>>> import numpy as np
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])

>>> a = np.array([1, 2, 3, 4])
>>> a
array([1, 2, 3, 4])
>>> a.transpose()
array([1, 2, 3, 4])
Type:      builtin_function_or_method
```

In [51]:
```
arr.transpose((2,1,0))
# Here, order is 2,1,0 and 2 comes from no. of columns, 1 comes from no of rows and 0 comes from no. of grids
# so it will show 4 grids, 2 rows and 2 columns and
# elements will be arranged in the order as column--> row--> grid
```

Out[51]:
```
array([[[ 0,  8],
        [ 4, 12]],

       [[ 1,  9],
        [ 5, 13]],

       [[ 2, 10],
        [ 6, 14]],

       [[ 3, 11],
        [ 7, 15]]])
```

In [52]:
```
arr.transpose((1,2,0))
```

Out[52]:
```
array([[[ 0,  8],
        [ 1,  9],
        [ 2, 10],
        [ 3, 11]],

       [[ 4, 12],
        [ 5, 13],
        [ 6, 14],
        [ 7, 15]]])
```

In [53]:
```
arr.transpose((1,0,2))
```

```
Out[53]: array([[[ 0,  1,  2,  3],
                 [ 8,  9, 10, 11]],

                [[ 4,  5,  6,  7],
                 [12, 13, 14, 15]]])
```

```
In [54]: arr.strides
```

```
Out[54]: (64, 32, 8)
```

```
In [55]: arr.dtype
```

```
Out[55]: dtype('int64')
```

```
In [56]: # So, each element has 4 bytes and from one grid to another grid, it has to cross 8 elements i.e. 32 bytes
         # which is showing in the arr.strides and similary
         # go from one row to another row, it has to cross 4 elements i.e. 16 bytes and from one column to another colum
         # it has to cross 4 bytes.
```

```
In [57]: arr.transpose((2,1,0)).strides
```

```
Out[57]: (8, 32, 64)
```

```
In [58]: np.empty((2,2,3))
```

```
Out[58]: array([[[0.0e+000, 4.9e-324, 9.9e-324],
                 [1.5e-323, 2.0e-323, 2.5e-323]],

                [[3.0e-323, 3.5e-323, 1.5e-323],
                 [2.0e-323, 2.5e-323, 5.4e-323]]])
```

```
In [59]: np.empty((2,2,3)).swapaxes(0,2)
         # here we are swapping axis 0 and axis 2. axis 0 means grid and axis 2 means columns.
         # so we are swapping grids and columns. Initial we has 2 grids so now we have 2 columns
         # and initially we had 3 columns so now we have 3 grids.
         # Accordingly elements will be arranged like elements in columns will go in different grids.
```

```
Out[59]: array([[[0.0e+000, 3.0e-323],
                 [1.5e-323, 2.0e-323]],

                [[4.9e-324, 3.5e-323],
                 [2.0e-323, 2.5e-323]],

                [[9.9e-324, 1.5e-323],
                 [2.5e-323, 5.4e-323]]])
```

Universal Functions - Fast Element-Wise Array Functions:

A universal function, or ufunc, is a function that performs element-wise operations on data in ndarrays. You can think of them as fast vectorized wrappers for simple functions that take one or more scalar values and produce one or more scalar results. Many ufuncs are simple element-wise transformations, like sqrt or exp.

```
In [60]: arr = np.array([1,2,3])
         np.sqrt(arr)
```

```
Out[60]: array([1.        , 1.41421356, 1.73205081])
```

```
In [61]: np.exp(arr)
```

```
Out[61]: array([ 2.71828183,  7.3890561 , 20.08553692])
```

```
In [62]: np.log(arr)
```

```
Out[62]: array([0.        , 0.69314718, 1.09861229])
```

```
In [63]: x=np.array([1,2,3])
         y=np.array([2,3,1])
         np.maximum(x,y)
```

```
Out[63]: array([2, 3, 3])
```

```
In [64]: arr = np.random.randn(7) * 5
         remainder,whole_part=np.modf(arr)
         arr,remainder,whole_part
```

```
Out[64]: (array([ 3.18161198, -1.28419106,  3.52870921,  3.41571939,  0.21864495,
                  0.85336174,  0.72593577]),
          array([ 0.18161198, -0.28419106,  0.52870921,  0.41571939,  0.21864495,
                  0.85336174,  0.72593577]),
          array([ 3., -1.,  3.,  3.,  0.,  0.,  0.]))
```

Table 4-3. Unary ufuncs

Function --> Description

abs, fabs --> Compute the absolute value element-wise for integer, foating-point, or complex values
sqrt --> Compute the square root of each element (equivalent to arr **0.5)**
square --> Compute the square of each element (equivalent to arr 2)
exp --> Compute the exponent ex of each element
log, log10,log2, log1p --> Natural logarithm (base e), log base 10, log base 2, and log(1 + x), respectively
sign --> Compute the sign of each element: 1 (positive), 0 (zero), or –1 (negative)
ceil --> Compute the ceiling of each element (i.e., the smallest integer greater than or equal to that number)
floor --> Compute the foor of each element (i.e., the largest integer less than or equal to each element)
rint --> Round elements to the nearest integer, preserving the dtype
modf --> Return fractional and integral parts of array as a separate array
isnan --> Return boolean array indicating whether each value is NaN (Not a Number)
isfinite, isinf --> Return boolean array indicating whether each element is finite (non-inf, non-NaN) or infinite,respectively
cos, cosh, sin,sinh, tan, tanh --> Regular and hyperbolic trigonometric functions
arccos, arccosh,arcsin, arcsinh,arctan, arctanh --> Inverse trigonometric functions
logical_not --> Compute truth value of not x element-wise (equivalent to ~arr).

In [65]:
```
np.abs(-1.2)
```
Out[65]:
```
np.float64(1.2)
```

In [66]:
```
np.fabs([1,1.2,-1.02])
```
Out[66]:
```
array([1.  , 1.2 , 1.02])
```

In [67]:
```
np.fabs?
```
```
Call signature:  np.fabs(*args, **kwargs)
Type:            ufunc
String form:     <ufunc 'fabs'>
File:            ~/Desktop/ml/InterviewPreparation/AI-ML-DS/practice_env/lib/python3.10/site-packages/numpy/__i
nit__.py
Docstring:
fabs(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])

Compute the absolute values element-wise.

This function returns the absolute values (positive magnitude) of the
data in `x`. Complex values are not handled, use `absolute` to find the
absolute values of complex data.

Parameters
----------
x : array_like
    The array of numbers for which the absolute values are required. If
    `x` is a scalar, the result `y` will also be a scalar.
out : ndarray, None, or tuple of ndarray and None, optional
    A location into which the result is stored. If provided, it must have
    a shape that the inputs broadcast to. If not provided or None,
    a freshly-allocated array is returned. A tuple (possible only as a
    keyword argument) must have length equal to the number of outputs.
where : array_like, optional
    This condition is broadcast over the input. At locations where the
    condition is True, the `out` array will be set to the ufunc result.
    Elsewhere, the `out` array will retain its original value.
    Note that if an uninitialized `out` array is created via the default
    ``out=None``, locations within it where the condition is False will
    remain uninitialized.
**kwargs
    For other keyword-only arguments, see the
    :ref:`ufunc docs <ufuncs.kwargs>`.

Returns
-------
y : ndarray or scalar
    The absolute values of `x`, the returned values are always floats.
    This is a scalar if `x` is a scalar.

See Also
--------
absolute : Absolute values including `complex` types.

Examples
--------
>>> import numpy as np
>>> np.fabs(-1)
1.0
>>> np.fabs([-1.2, 1.2])
array([ 1.2,  1.2])
Class docstring:
Functions that operate element by element on whole arrays.
```

```
To see the documentation for a specific ufunc, use `info`.  For
example, ``np.info(np.sin)``.  Because ufuncs are written in C
(for speed) and linked into Python with NumPy's ufunc facility,
Python's help() function finds this page whenever help() is called
on a ufunc.

A detailed explanation of ufuncs can be found in the docs for :ref:`ufuncs`.

**Calling ufuncs:** ``op(*x[, out], where=True, **kwargs)``

Apply `op` to the arguments `*x` elementwise, broadcasting the arguments.

The broadcasting rules are:

* Dimensions of length 1 may be prepended to either array.
* Arrays may be repeated along dimensions of length 1.

Parameters
----------
*x : array_like
    Input arrays.
out : ndarray, None, or tuple of ndarray and None, optional
    Alternate array object(s) in which to put the result; if provided, it
    must have a shape that the inputs broadcast to. A tuple of arrays
    (possible only as a keyword argument) must have length equal to the
    number of outputs; use None for uninitialized outputs to be
    allocated by the ufunc.
where : array_like, optional
    This condition is broadcast over the input. At locations where the
    condition is True, the `out` array will be set to the ufunc result.
    Elsewhere, the `out` array will retain its original value.
    Note that if an uninitialized `out` array is created via the default
    ``out=None``, locations within it where the condition is False will
    remain uninitialized.
**kwargs
    For other keyword-only arguments, see the :ref:`ufunc docs <ufuncs.kwargs>`.

Returns
-------
r : ndarray or tuple of ndarray
    `r` will have the shape that the arrays in `x` broadcast to; if `out` is
    provided, it will be returned. If not, `r` will be allocated and
    may contain uninitialized values. If the function has more than one
    output, then the result will be a tuple of arrays.
```

In [68]:
```
np.absolute?
```

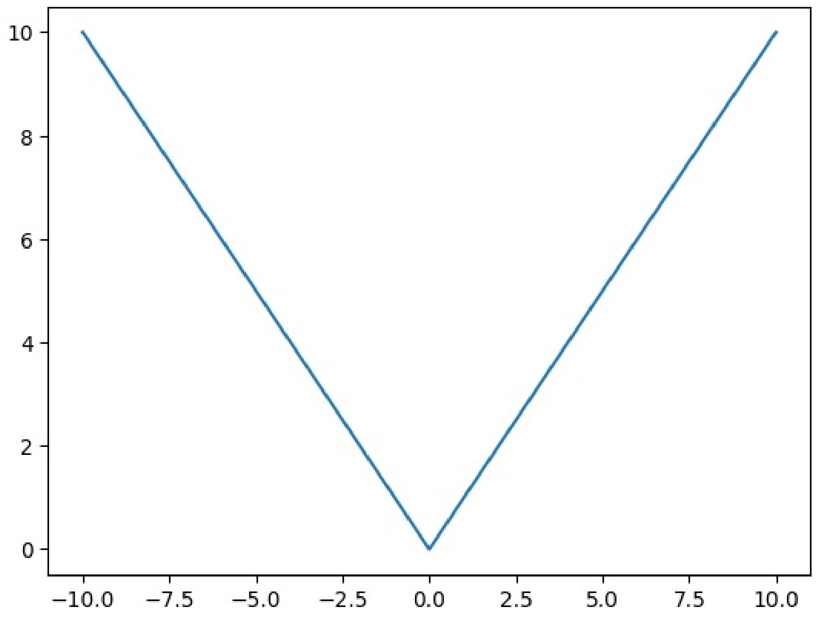
```
Call signature:  np.absolute(*args, **kwargs)
Type:            ufunc
String form:     <ufunc 'absolute'>
File:            ~/Desktop/ml/InterviewPreparation/AI-ML-DS/practice_env/lib/python3.10/site-packages/numpy/__i
nit__.py
Docstring:
absolute(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])

Calculate the absolute value element-wise.

``np.abs`` is a shorthand for this function.

Parameters
----------
x : array_like
    Input array.
out : ndarray, None, or tuple of ndarray and None, optional
    A location into which the result is stored. If provided, it must have
    a shape that the inputs broadcast to. If not provided or None,
    a freshly-allocated array is returned. A tuple (possible only as a
    keyword argument) must have length equal to the number of outputs.
where : array_like, optional
    This condition is broadcast over the input. At locations where the
    condition is True, the `out` array will be set to the ufunc result.
    Elsewhere, the `out` array will retain its original value.
    Note that if an uninitialized `out` array is created via the default
    ``out=None``, locations within it where the condition is False will
    remain uninitialized.
**kwargs
    For other keyword-only arguments, see the
    :ref:`ufunc docs <ufuncs.kwargs>`.

Returns
-------
absolute : ndarray
    An ndarray containing the absolute value of
    each element in `x`.  For complex input, ``a + ib``, the
    absolute value is :math:`\sqrt{ a^2 + b^2 }`.
    This is a scalar if `x` is a scalar.

Examples
```

```
--------
>>> import numpy as np
>>> x = np.array([-1.2, 1.2])
>>> np.absolute(x)
array([ 1.2,  1.2])
>>> np.absolute(1.2 + 1j)
1.5620499351813308

Plot the function over ``[-10, 10]``:

>>> import matplotlib.pyplot as plt

>>> x = np.linspace(start=-10, stop=10, num=101)
>>> plt.plot(x, np.absolute(x))
>>> plt.show()

Plot the function over the complex plane:

>>> xx = x + 1j * x[:, np.newaxis]
>>> plt.imshow(np.abs(xx), extent=[-10, 10, -10, 10], cmap='gray')
>>> plt.show()

The `abs` function can be used as a shorthand for ``np.absolute`` on
ndarrays.

>>> x = np.array([-1.2, 1.2])
>>> abs(x)
array([1.2, 1.2])
Class docstring:
Functions that operate element by element on whole arrays.

To see the documentation for a specific ufunc, use `info`.  For
example, ``np.info(np.sin)``.  Because ufuncs are written in C
(for speed) and linked into Python with NumPy's ufunc facility,
Python's help() function finds this page whenever help() is called
on a ufunc.

A detailed explanation of ufuncs can be found in the docs for :ref:`ufuncs`.

**Calling ufuncs:** ``op(*x[, out], where=True, **kwargs)``

Apply `op` to the arguments `*x` elementwise, broadcasting the arguments.

The broadcasting rules are:

* Dimensions of length 1 may be prepended to either array.
* Arrays may be repeated along dimensions of length 1.

Parameters
----------
*x : array_like
    Input arrays.
out : ndarray, None, or tuple of ndarray and None, optional
    Alternate array object(s) in which to put the result; if provided, it
    must have a shape that the inputs broadcast to. A tuple of arrays
    (possible only as a keyword argument) must have length equal to the
    number of outputs; use None for uninitialized outputs to be
    allocated by the ufunc.
where : array_like, optional
    This condition is broadcast over the input. At locations where the
    condition is True, the `out` array will be set to the ufunc result.
    Elsewhere, the `out` array will retain its original value.
    Note that if an uninitialized `out` array is created via the default
    ``out=None``, locations within it where the condition is False will
    remain uninitialized.
**kwargs
    For other keyword-only arguments, see the :ref:`ufunc docs <ufuncs.kwargs>`.

Returns
-------
r : ndarray or tuple of ndarray
    `r` will have the shape that the arrays in `x` broadcast to; if `out` is
    provided, it will be returned. If not, `r` will be allocated and
    may contain uninitialized values. If the function has more than one
    output, then the result will be a tuple of arrays.
```

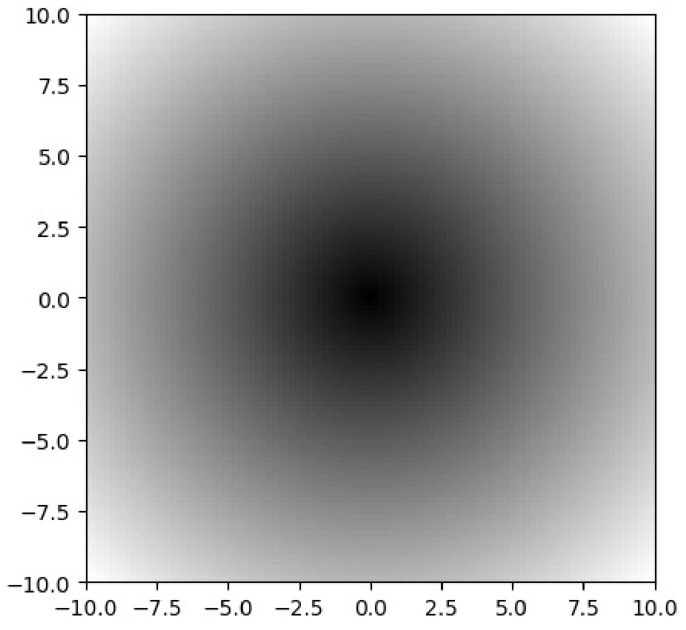
In [69]:
```python
np.absolute(1.2 + 1j)
```

Out[69]:
```
np.float64(1.5620499351813308)
```

In [70]:
```python
import matplotlib.pyplot as plt

x = np.linspace(start=-10, stop=10, num=101)
plt.plot(x, np.absolute(x))
plt.show()
```



```
In [71]: #Plot the function over the complex plane:

         xx = x + 1j * x[:, np.newaxis]
         plt.imshow(np.abs(xx), extent=[-10, 10, -10, 10], cmap='gray')
         plt.show()
```



```
In [72]: np.log1p(1)
Out[72]: np.float64(0.6931471805599453)

In [73]: np.ceil(1.2)
Out[73]: np.float64(2.0)

In [74]: np.floor(1.2)
Out[74]: np.float64(1.0)

In [75]: np.rint(1.2)
Out[75]: np.float64(1.0)

In [76]: np.rint(1.5)
Out[76]: np.float64(2.0)

In [77]: np.rint(1.6)
Out[77]: np.float64(2.0)

In [78]: np.rint(1.4)
```

Out[78]:
```
np.float64(1.0)
```

In [79]:
```
np.isnan([1,0,-1,np.nan])
```

Out[79]:
```
array([False, False, False,  True])
```

In [80]:
```
np.arccos(0)
```

Out[80]:
```
np.float64(1.5707963267948966)
```

In [81]:
```
np.isfinite(np.nan)
```

Out[81]:
```
np.False_
```

In [82]:
```
np.isinf(np.nan)
```

Out[82]:
```
np.False_
```

Table 4-4. Binary universal functions

Function --> Description

add --> Add corresponding elements in arrays
subtract --> Subtract elements in second array from first array
multiply --> Multiply array elements
divide, floor_divide --> Divide or foor divide (truncating the remainder)
power --> Raise elements in first array to powers indicated in second array
maximum, fmax --> Element-wise maximum; fmax ignores NaN
minimum, fmin --> Element-wise minimum; fmin ignores NaN
mod --> Element-wise modulus (remainder of division)
copysign --> Copy sign of values in second argument to values in first argument
greater, greater_equal,less, less_equal,equal, not_equal --> Perform element-wise comparison, yielding boolean array (equivalent to infix operators >, >=, <, <=, ==, !=)
logical_and, logical_or, logical_xor --> Compute element-wise truth value of logical operation (equivalent to infix operators & |, ^)

In [83]:
```
np.add?
```

```
Call signature:  np.add(*args, **kwargs)
Type:            ufunc
String form:     <ufunc 'add'>
File:            ~/Desktop/ml/InterviewPreparation/AI-ML-DS/practice_env/lib/python3.10/site-packages/numpy/__i
nit__.py
Docstring:
add(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature])

Add arguments element-wise.

Parameters
----------
x1, x2 : array_like
    The arrays to be added.
    If ``x1.shape != x2.shape``, they must be broadcastable to a common
    shape (which becomes the shape of the output).
out : ndarray, None, or tuple of ndarray and None, optional
    A location into which the result is stored. If provided, it must have
    a shape that the inputs broadcast to. If not provided or None,
    a freshly-allocated array is returned. A tuple (possible only as a
    keyword argument) must have length equal to the number of outputs.
where : array_like, optional
    This condition is broadcast over the input. At locations where the
    condition is True, the `out` array will be set to the ufunc result.
    Elsewhere, the `out` array will retain its original value.
    Note that if an uninitialized `out` array is created via the default
    ``out=None``, locations within it where the condition is False will
    remain uninitialized.
**kwargs
    For other keyword-only arguments, see the
    :ref:`ufunc docs <ufuncs.kwargs>`.

Returns
-------
add : ndarray or scalar
    The sum of `x1` and `x2`, element-wise.
    This is a scalar if both `x1` and `x2` are scalars.

Notes
-----
Equivalent to `x1` + `x2` in terms of array broadcasting.
```

```
Examples
--------
>>> import numpy as np
>>> np.add(1.0, 4.0)
5.0
>>> x1 = np.arange(9.0).reshape((3, 3))
>>> x2 = np.arange(3.0)
>>> np.add(x1, x2)
array([[  0.,   2.,   4.],
       [  3.,   5.,   7.],
       [  6.,   8.,  10.]])

The ``+`` operator can be used as a shorthand for ``np.add`` on ndarrays.

>>> x1 = np.arange(9.0).reshape((3, 3))
>>> x2 = np.arange(3.0)
>>> x1 + x2
array([[ 0.,  2.,  4.],
       [ 3.,  5.,  7.],
       [ 6.,  8., 10.]])
Class docstring:
Functions that operate element by element on whole arrays.

To see the documentation for a specific ufunc, use `info`.  For
example, ``np.info(np.sin)``.  Because ufuncs are written in C
(for speed) and linked into Python with NumPy's ufunc facility,
Python's help() function finds this page whenever help() is called
on a ufunc.

A detailed explanation of ufuncs can be found in the docs for :ref:`ufuncs`.

**Calling ufuncs:** ``op(*x[, out], where=True, **kwargs)``

Apply `op` to the arguments `*x` elementwise, broadcasting the arguments.

The broadcasting rules are:

* Dimensions of length 1 may be prepended to either array.
* Arrays may be repeated along dimensions of length 1.

Parameters
----------
*x : array_like
    Input arrays.
out : ndarray, None, or tuple of ndarray and None, optional
    Alternate array object(s) in which to put the result; if provided, it
    must have a shape that the inputs broadcast to. A tuple of arrays
    (possible only as a keyword argument) must have length equal to the
    number of outputs; use None for uninitialized outputs to be
    allocated by the ufunc.
where : array_like, optional
    This condition is broadcast over the input. At locations where the
    condition is True, the `out` array will be set to the ufunc result.
    Elsewhere, the `out` array will retain its original value.
    Note that if an uninitialized `out` array is created via the default
    ``out=None``, locations within it where the condition is False will
    remain uninitialized.
**kwargs
    For other keyword-only arguments, see the :ref:`ufunc docs <ufuncs.kwargs>`.

Returns
-------
r : ndarray or tuple of ndarray
    `r` will have the shape that the arrays in `x` broadcast to; if `out` is
    provided, it will be returned. If not, `r` will be allocated and
    may contain uninitialized values. If the function has more than one
    output, then the result will be a tuple of arrays.
```

In [84]:
```python
np.add([1,2,3],[4,5,6])
```
Out[84]:
```
array([5, 7, 9])
```

In [85]:
```python
np.power([1,2,3],[4,5,3])
```
Out[85]:
```
array([ 1, 32, 27])
```

In [86]:
```python
np.fmax([1,4,3],[2,1,5])
```
Out[86]:
```
array([2, 4, 5])
```

In [87]:
```python
np.mod([7,26,3],[2,6,2])
```
Out[87]:
```
array([1, 2, 1])
```

In [88]:
```python
np.copysign([1,2,3],[-1,-2,1])
```

```
Out[88]: array([-1., -2.,  3.])
```

```
In [89]: np.greater_equal([1,2,3],[2,2,1])
```

```
Out[89]: array([False,  True,  True])
```

```
In [90]: np.logical_xor([1,0,1],[0,0,1])
```

```
Out[90]: array([ True, False, False])
```

```
In [91]: # computing sqrt(x^2 + y^2) across a regular grid of values:

         x=np.arange(-5,5,0.01)
         y=np.arange(-5,5,0.01)
         xs,ys=np.meshgrid(x,y)
         print(np.sqrt(xs**2+ys**2))
```

```
[[7.07106781 7.06400028 7.05693985 ... 7.04988652 7.05693985 7.06400028]
 [7.06400028 7.05692568 7.04985815 ... 7.04279774 7.04985815 7.05692568]
 [7.05693985 7.04985815 7.04278354 ... 7.03571603 7.04278354 7.04985815]
 ...
 [7.04988652 7.04279774 7.03571603 ... 7.0286414  7.03571603 7.04279774]
 [7.05693985 7.04985815 7.04278354 ... 7.03571603 7.04278354 7.04985815]
 [7.06400028 7.05692568 7.04985815 ... 7.04279774 7.04985815 7.05692568]]
```

```
In [92]: x=np.array(['ankit','arpit'])
         y=np.array(['kiio','summi'])
         np.where([True,False],x,y)
```

```
Out[92]: array(['ankit', 'summi'], dtype='<U5')
```

```
In [93]: np.where?
```

```
Call signature:  np.where(*args, **kwargs)
Type:            _ArrayFunctionDispatcher
String form:     <built-in function where>
Docstring:
where(condition, [x, y], /)

Return elements chosen from `x` or `y` depending on `condition`.

.. note::
    When only `condition` is provided, this function is a shorthand for
    ``np.asarray(condition).nonzero()``. Using `nonzero` directly should be
    preferred, as it behaves correctly for subclasses. The rest of this
    documentation covers only the case where all three arguments are
    provided.

Parameters
----------
condition : array_like, bool
    Where True, yield `x`, otherwise yield `y`.
x, y : array_like
    Values from which to choose. `x`, `y` and `condition` need to be
    broadcastable to some shape.

Returns
-------
out : ndarray
    An array with elements from `x` where `condition` is True, and elements
    from `y` elsewhere.

See Also
--------
choose
nonzero : The function that is called when x and y are omitted

Notes
-----
If all the arrays are 1-D, `where` is equivalent to::

    [xv if c else yv
     for c, xv, yv in zip(condition, x, y)]

Examples
--------
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.where(a < 5, a, 10*a)
array([ 0,  1,  2,  3,  4, 50, 60, 70, 80, 90])

This can be used on multidimensional arrays too:

>>> np.where([[True, False], [True, True]],
...          [[1, 2], [3, 4]],
...           [[9, 8], [7, 6]])
array([[1, 8],
       [3, 4]])

The shapes of x, y, and the condition are broadcast together:

>>> x, y = np.ogrid[:3, :4]
>>> np.where(x < y, x, 10 + y)  # both x and 10+y are broadcast
array([[10,  0,  0,  0],
       [10, 11,  1,  1],
       [10, 11, 12,  2]])

>>> a = np.array([[0, 1, 2],
...               [0, 2, 4],
...               [0, 3, 6]])
>>> np.where(a < 4, a, -1)  # -1 is broadcast
array([[ 0,  1,  2],
       [ 0,  2, -1],
       [ 0,  3, -1]])
Class docstring:
Class to wrap functions with checks for __array_function__ overrides.

All arguments are required, and can only be passed by position.

Parameters
----------
dispatcher : function or None
    The dispatcher function that returns a single sequence-like object
    of all arguments relevant.  It must have the same signature (except
    the default values) as the actual implementation.
    If ``None``, this is a ``like=`` dispatcher and the
    ``_ArrayFunctionDispatcher`` must be called with ``like`` as the
    first (additional and positional) argument.
implementation : function
    Function that implements the operation on NumPy arrays without
    overrides.  Arguments passed calling the ``_ArrayFunctionDispatcher``
    will be forwarded to this (and the ``dispatcher``) as if using
    ``*args, **kwargs``.

Attributes
----------
_implementation : function
    The original implementation passed in.
```

In [94]:
```python
arr=np.random.randn(5)
print(arr)
print(np.where(arr>0,2,-3))
print(np.where(arr>0,2,arr))
```

```
[ 1.03518014  0.26568722  0.3583539   0.15972404 -2.25755302]
[ 2  2  2  2 -3]
[ 2.          2.          2.          2.         -2.25755302]
```

In [95]:
```python
arr = np.array([[1,2,3],[4,5,6]])
arr
```

Out[95]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [96]:
```python
arr.mean() # mean of all elements
```

Out[96]:
```
np.float64(3.5)
```

In [97]:
```python
arr.mean(axis=0) # mean of each column
```

Out[97]:
```
array([2.5, 3.5, 4.5])
```

In [98]:
```python
arr.mean(axis=1) # mean of each row
```

Out[98]:
```
array([2., 5.])
```

In [99]:
```python
np.mean(arr)
```

Out[99]:
```
np.float64(3.5)
```

In [100…
```python
np.mean(arr,axis=0)
```

Out[100]:
```
array([2.5, 3.5, 4.5])
```

In [101…
```python
np.mean(arr,axis=1)
```

Out[101]:
```
array([2., 5.])
```

In [102…
```python
arr.cumsum(axis=0)
```

```
Out[102]: array([[1, 2, 3],
                 [5, 7, 9]])
```

```
In [103…  arr.cumsum(axis=1)
```

```
Out[103]: array([[ 1,  3,  6],
                 [ 4,  9, 15]])
```

```
In [104…  arr.cumsum()
```

```
Out[104]: array([ 1,  3,  6, 10, 15, 21])
```

```
In [105…  arr.cumprod()
```

```
Out[105]: array([  1,   2,   6,  24, 120, 720])
```

```
In [106…  arr.cumprod(axis=0)
```

```
Out[106]: array([[ 1,  2,  3],
                 [ 4, 10, 18]])
```

```
In [107…  arr.cumprod(axis=1)
```

```
Out[107]: array([[  1,   2,   6],
                 [  4,  20, 120]])
```

Method --> Description

sum --> Sum of all the elements in the array or along an axis; zero-length arrays have sum 0
mean --> Arithmetic mean; zero-length arrays have NaN mean
std, var --> Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator n)
min, max --> Minimum and maximum
argmin, argmax --> Indices of minimum and maximum elements, respectively
cumsum --> Cumulative sum of elements starting from 0
cumprod --> Cumulative product of elements starting from 1

```
In [108…  arr=np.array([])
          arr.mean()
```

```
/tmp/ipykernel_12714/1578359007.py:2: RuntimeWarning: Mean of empty slice.
  arr.mean()
/home/ankit/Desktop/ml/InterviewPreparation/AI-ML-DS/practice_env/lib/python3.10/site-packages/numpy/_core/_met
hods.py:145: RuntimeWarning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
Out[108]: np.float64(nan)
```

```
In [109…  np.sum((1,2))
```

```
Out[109]: np.int64(3)
```

```
In [110…  np.sum([-1,3])
```

```
Out[110]: np.int64(2)
```

```
In [111…  np.std([[1,2],[4,3]])
```

```
Out[111]: np.float64(1.118033988749895)
```

```
In [112…  np.std?
```

```
Signature:
np.std(
    a,
    axis=None,
    dtype=None,
    out=None,
    ddof=0,
    keepdims=<no value>,
    *,
    where=<no value>,
    mean=<no value>,
    correction=<no value>,
)
Call signature:  np.std(*args, **kwargs)
Type:            _ArrayFunctionDispatcher
String form:     <function std at 0x7ec3b85b27a0>
File:            ~/Desktop/ml/InterviewPreparation/AI-ML-DS/practice_env/lib/python3.10/site-packages/numpy/_co
re/fromnumeric.py
Docstring:
Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution,
of the array elements. The standard deviation is computed for the
flattened array by default, otherwise over the specified axis.

Parameters
----------
a : array_like
    Calculate the standard deviation of these values.
axis : None or int or tuple of ints, optional
    Axis or axes along which the standard deviation is computed. The
    default is to compute the standard deviation of the flattened array.
    If this is a tuple of ints, a standard deviation is performed over
    multiple axes, instead of a single axis or all the axes as before.
dtype : dtype, optional
    Type to use in computing the standard deviation. For arrays of
    integer type the default is float64, for arrays of float types it is
    the same as the array type.
out : ndarray, optional
    Alternative output array in which to place the result. It must have
    the same shape as the expected output but the type (of the calculated
    values) will be cast if necessary.
    See :ref:`ufuncs-output-type` for more details.
ddof : {int, float}, optional
    Means Delta Degrees of Freedom.  The divisor used in calculations
    is ``N - ddof``, where ``N`` represents the number of elements.
    By default `ddof` is zero. See Notes for details about use of `ddof`.
keepdims : bool, optional
    If this is set to True, the axes which are reduced are left
    in the result as dimensions with size one. With this option,
    the result will broadcast correctly against the input array.

    If the default value is passed, then `keepdims` will not be
    passed through to the `std` method of sub-classes of
    `ndarray`, however any non-default value will be.  If the
    sub-class' method does not implement `keepdims` any
    exceptions will be raised.
where : array_like of bool, optional
    Elements to include in the standard deviation.
    See `~numpy.ufunc.reduce` for details.

    .. versionadded:: 1.20.0

mean : array_like, optional
    Provide the mean to prevent its recalculation. The mean should have
    a shape as if it was calculated with ``keepdims=True``.
    The axis for the calculation of the mean should be the same as used in
    the call to this std function.

    .. versionadded:: 2.0.0

correction : {int, float}, optional
    Array API compatible name for the ``ddof`` parameter. Only one of them
    can be provided at the same time.

    .. versionadded:: 2.0.0

Returns
-------
standard_deviation : ndarray, see dtype parameter above.
    If `out` is None, return a new array containing the standard deviation,
    otherwise return a reference to the output array.

See Also
--------
var, mean, nanmean, nanstd, nanvar
:ref:`ufuncs-output-type`

Notes
-----
There are several common variants of the array standard deviation
calculation. Assuming the input `a` is a one-dimensional NumPy array
and ``mean`` is either provided as an argument or computed as
``a.mean()``, NumPy computes the standard deviation of an array as::

    N = len(a)
    d2 = abs(a - mean)**2  # abs is for complex `a`
    var = d2.sum() / (N - ddof)  # note use of `ddof`
    std = var**0.5

Different values of the argument `ddof` are useful in different
contexts. NumPy's default ``ddof=0`` corresponds with the expression:

.. math::

    \sqrt{\frac{\sum_i{|a_i - \bar{a}|^2 }}{N}}

which is sometimes called the "population standard deviation" in the field
of statistics because it applies the definition of standard deviation to
`a` as if `a` were a complete population of possible observations.
```

```
Many other libraries define the standard deviation of an array
differently, e.g.:

.. math::

    \sqrt{\frac{\sum_i{|a_i - \bar{a}|^2 }}{N - 1}}

In statistics, the resulting quantity is sometimes called the "sample
standard deviation" because if `a` is a random sample from a larger
population, this calculation provides the square root of an unbiased
estimate of the variance of the population. The use of :math:`N-1` in the
denominator is often called "Bessel's correction" because it corrects for
bias (toward lower values) in the variance estimate introduced when the
sample mean of `a` is used in place of the true mean of the population.
The resulting estimate of the standard deviation is still biased, but less
than it would have been without the correction. For this quantity, use
``ddof=1``.

Note that, for complex numbers, `std` takes the absolute
value before squaring, so that the result is always real and nonnegative.

For floating-point input, the standard deviation is computed using the same
precision the input has. Depending on the input data, this can cause
the results to be inaccurate, especially for float32 (see example below).
Specifying a higher-accuracy accumulator using the `dtype` keyword can
alleviate this issue.

Examples
--------
>>> import numpy as np
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949 # may vary
>>> np.std(a, axis=0)
array([1.,  1.])
>>> np.std(a, axis=1)
array([0.5,  0.5])

In single precision, std() can be inaccurate:

>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.std(a)
np.float32(0.45000005)

Computing the standard deviation in float64 is more accurate:

>>> np.std(a, dtype=np.float64)
0.44999999925494177 # may vary

Specifying a where argument:

>>> a = np.array([[14, 8, 11, 10], [7, 9, 10, 11], [10, 15, 5, 10]])
>>> np.std(a)
2.614064523559687 # may vary
>>> np.std(a, where=[[True], [True], [False]])
2.0

Using the mean keyword to save computation time:

>>> import numpy as np
>>> from timeit import timeit
>>> a = np.array([[14, 8, 11, 10], [7, 9, 10, 11], [10, 15, 5, 10]])
>>> mean = np.mean(a, axis=1, keepdims=True)
>>>
>>> g = globals()
>>> n = 10000
>>> t1 = timeit("std = np.std(a, axis=1, mean=mean)", globals=g, number=n)
>>> t2 = timeit("std = np.std(a, axis=1)", globals=g, number=n)
>>> print(f'Percentage execution time saved {100*(t2-t1)/t2:.0f}%')
#doctest: +SKIP
Percentage execution time saved 30%
Class docstring:
Class to wrap functions with checks for __array_function__ overrides.

All arguments are required, and can only be passed by position.

Parameters
----------
dispatcher : function or None
    The dispatcher function that returns a single sequence-like object
    of all arguments relevant.  It must have the same signature (except
    the default values) as the actual implementation.
    If ``None``, this is a ``like=`` dispatcher and the
    ``_ArrayFunctionDispatcher`` must be called with ``like`` as the
    first (additional and positional) argument.
implementation : function
    Function that implements the operation on NumPy arrays without
    overrides.  Arguments passed calling the ``_ArrayFunctionDispatcher``
    will be forwarded to this (and the ``dispatcher``) as if using
    ``*args, **kwargs``.

Attributes
----------
_implementation : function
    The original implementation passed in.
```

In [113…
```
np.min([1,4,3])
```

Out[113]:
```
np.int64(1)
```

In [114…
```
np.argmax([1,4,3,5,2])
```

Out[114]:
```
np.int64(3)
```

In [115…
```
np.cumprod((7,2,-0.2))
```

Out[115]:
```
array([ 7. , 14. , -2.8])
```

In [116…
```
bools=np.array([True,False,True])
print(bools.sum(),bools.any(),bools.all())
```

```
2 True False
```

In [117…
```
z=np.array([[2,3,1],[6.1,2,8]])
z
```

Out[117]:
```
array([[2. , 3. , 1. ],
       [6.1, 2. , 8. ]])
```

In [118…
```
z.sort(axis=1)
print(z)
```

```
[[1.  2.  3. ]
 [2.  6.1 8. ]]
```

In [119…
```
ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
np.unique(ints)
```

Out[119]:
```
array([1, 2, 3, 4])
```

NumPy is able to save and load data to and from disk either in text or binary format.

np.save and np.load are the two workhorse functions for efficiently saving and loading array data on disk. Arrays are saved by default in an uncompressed raw binary format with file extension .npy

In [120…
```
z=np.arange(16).reshape(4,4)
np.save('myarray',z)
y=np.load('myarray.npy')
y
```

Out[120]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [121…
```
# For multiple files:

a=np.arange(12).reshape(2,2,3)
b=np.arange(12).reshape(4,3)
np.savez('myarrays.npz',x=a,y=b)
loaded=np.load('myarrays.npz')
loaded['x']
```

Out[121]:
```
array([[[ 0,  1,  2],
        [ 3,  4,  5]],

       [[ 6,  7,  8],
        [ 9, 10, 11]]])
```

In [122…
```
loaded['y']
```

Out[122]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [123…
```
# for compressed files:

a=np.arange(12).reshape(2,2,3)
b=np.arange(12).reshape(4,3)
np.savez_compressed("compressed_archived.npz",a=a,b=b)
arr=np.load("compressed_archived.npz")
print(arr['a'])
```

```
[[[ 0  1  2]
  [ 3  4  5]]

 [[ 6  7  8]
  [ 9 10 11]]]
```

In [124…
```
print(arr['b'])
```

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

Linear Algebra

In [125…
```
a=np.arange(6).reshape(2,3)
b=np.arange(6).reshape(3,2)
print(a.dot(b))
print(np.dot(a,b))
print(a @ b)
# The @ symbol (as of Python 3.5) also works as an infix operator that performs matrix multiplication
```

```
[[10 13]
 [28 40]]
[[10 13]
 [28 40]]
[[10 13]
 [28 40]]
```

`numpy.linalg` has a standard set of matrix decompositions and things like inverse and determinant. These are implemented under the hood via the same industry standard linear algebra libraries used in other languages like MATLAB and R, such as BLAS, LAPACK, or possibly (depending on your NumPy build) the proprietary Intel MKL (Math Kernel Library)

diag --> Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal
dot --> Matrix multiplication
trace --> Compute the sum of the diagonal elements
det --> Compute the matrix determinant
eig --> Compute the eigenvalues and eigenvectors of a square matrix
inv --> Compute the inverse of a square matrix
pinv --> Compute the Moore-Penrose pseudo-inverse of a matrix
qr --> Compute the QR decomposition
svd --> Compute the singular value decomposition (SVD)
solve --> Solve the linear system Ax = b for x, where A is a square matrix
lstsq --> Compute the least-squares solution to Ax = b

In [126…
```
np.linalg.inv([[1,2],[3,4]])
```

Out[126]:
```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

In [127…
```
np.linalg.det([[1,2],[3,4]])
```

Out[127]:
```
np.float64(-2.0000000000000004)
```

In [128…
```
np.linalg.diagonal([[1,2],[3,4]])
```

Out[128]:
```
array([1, 4])
```

In [129…
```
np.diag([1,2,3,4])
```

Out[129]:
```
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

In [130…
```
np.diag([[1,2],[3,4]])
```

Out[130]:
```
array([1, 4])
```

In [131…
```
np.trace([[1,2],[3,4]])
```

Out[131]:
```
np.int64(5)
```

In [132…
```
np.linalg.eig([[1,0],[0,1]])
```

Out[132]:
```
EigResult(eigenvalues=array([1., 1.]), eigenvectors=array([[1., 0.],
       [0., 1.]]))
```

In [133…
```
np.linalg.pinv([[1,2],[3,4]])
```

```
Out[133]: array([[-2. ,  1. ],
                 [ 1.5, -0.5]])
```

```
In [134…: np.linalg.pinv([[2,5,9],[4,6,5],[8,4,5]])
```

```
Out[134]: array([[-0.05952381, -0.06547619,  0.17261905],
                 [-0.11904762,  0.36904762, -0.1547619 ],
                 [ 0.19047619, -0.19047619,  0.04761905]])
```

```
In [135…: np.linalg.qr([[1,2],[3,4]])
```

```
Out[135]: QRResult(Q=array([[-0.31622777, -0.9486833 ],
                 [-0.9486833 ,  0.31622777]]), R=array([[-3.16227766, -4.42718872],
                 [ 0.        , -0.63245553]]))
```

```
In [136…: np.linalg.svd([[1,2],[3,4]])
```

```
Out[136]: SVDResult(U=array([[-0.40455358, -0.9145143 ],
                 [-0.9145143 ,  0.40455358]]), S=array([5.4649857 , 0.36596619]), Vh=array([[-0.57604844, -0.81741556],
                 [ 0.81741556, -0.57604844]]))
```

```
In [137…: np.linalg.solve?
```

```
Signature:      np.linalg.solve(a, b)
Call signature: np.linalg.solve(*args, **kwargs)
Type:           _ArrayFunctionDispatcher
String form:    <function solve at 0x7ec3b84a2830>
File:           ~/Desktop/ml/InterviewPreparation/AI-ML-DS/practice_env/lib/python3.10/site-packages/numpy/lin
alg/_linalg.py
Docstring:
Solve a linear matrix equation, or system of linear scalar equations.

Computes the "exact" solution, `x`, of the well-determined, i.e., full
rank, linear matrix equation `ax = b`.

Parameters
----------
a : (..., M, M) array_like
    Coefficient matrix.
b : {(M,), (..., M, K)}, array_like
    Ordinate or "dependent variable" values.

Returns
-------
x : {(..., M,), (..., M, K)} ndarray
    Solution to the system a x = b.  Returned shape is (..., M) if b is
    shape (M,) and (..., M, K) if b is (..., M, K), where the "..." part is
    broadcasted between a and b.

Raises
------
LinAlgError
    If `a` is singular or not square.

See Also
--------
scipy.linalg.solve : Similar function in SciPy.

Notes
-----
Broadcasting rules apply, see the `numpy.linalg` documentation for
details.

The solutions are computed using LAPACK routine ``_gesv``.

`a` must be square and of full-rank, i.e., all rows (or, equivalently,
columns) must be linearly independent; if either is not true, use
`lstsq` for the least-squares best "solution" of the
system/equation.

.. versionchanged:: 2.0

   The b array is only treated as a shape (M,) column vector if it is
   exactly 1-dimensional. In all other instances it is treated as a stack
   of (M, K) matrices. Previously b would be treated as a stack of (M,)
   vectors if b.ndim was equal to a.ndim - 1.

References
----------
.. [1] G. Strang, *Linear Algebra and Its Applications*, 2nd Ed., Orlando,
       FL, Academic Press, Inc., 1980, pg. 22.

Examples
--------
Solve the system of equations:
``x0 + 2 * x1 = 1`` and
``3 * x0 + 5 * x1 = 2``:

>>> import numpy as np
>>> a = np.array([[1, 2], [3, 5]])
>>> b = np.array([1, 2])
>>> x = np.linalg.solve(a, b)
>>> x
array([-1.,  1.])

Check that the solution is correct:

>>> np.allclose(np.dot(a, x), b)
True
Class docstring:
Class to wrap functions with checks for __array_function__ overrides.

All arguments are required, and can only be passed by position.

Parameters
----------
dispatcher : function or None
    The dispatcher function that returns a single sequence-like object
    of all arguments relevant.  It must have the same signature (except
    the default values) as the actual implementation.
    If ``None``, this is a ``like=`` dispatcher and the
    ``_ArrayFunctionDispatcher`` must be called with ``like`` as the
    first (additional and positional) argument.
implementation : function
    Function that implements the operation on NumPy arrays without
    overrides.  Arguments passed calling the ``_ArrayFunctionDispatcher``
    will be forwarded to this (and the ``dispatcher``) as if using
    ``*args, **kwargs``.

Attributes
----------
_implementation : function
    The original implementation passed in.
```

In [138…

```python
import numpy as np
a = np.array([[1, 2], [3, 5]])
b = np.array([1, 2])
x = np.linalg.solve(a, b)
x
```

Out[138]:

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

In [139…

```python
np.linalg.lstsq?
```

```
Signature:       np.linalg.lstsq(a, b, rcond=None)
Call signature:  np.linalg.lstsq(*args, **kwargs)
Type:            _ArrayFunctionDispatcher
String form:     <function lstsq at 0x7ec3b84a3a30>
File:            ~/Desktop/ml/InterviewPreparation/AI-ML-DS/practice_env/lib/python3.10/site-packages/numpy/lin
alg/_linalg.py
Docstring:
Return the least-squares solution to a linear matrix equation.

Computes the vector `x` that approximately solves the equation
``a @ x = b``. The equation may be under-, well-, or over-determined
(i.e., the number of linearly independent rows of `a` can be less than,
equal to, or greater than its number of linearly independent columns).
If `a` is square and of full rank, then `x` (but for round-off error)
is the "exact" solution of the equation. Else, `x` minimizes the
Euclidean 2-norm :math:`||b - ax||`. If there are multiple minimizing
solutions, the one with the smallest 2-norm :math:`||x||` is returned.

Parameters
----------
a : (M, N) array_like
    "Coefficient" matrix.
b : {(M,), (M, K)} array_like
    Ordinate or "dependent variable" values. If `b` is two-dimensional,
    the least-squares solution is calculated for each of the `K` columns
    of `b`.
rcond : float, optional
    Cut-off ratio for small singular values of `a`.
    For the purposes of rank determination, singular values are treated
    as zero if they are smaller than `rcond` times the largest singular
    value of `a`.
    The default uses the machine precision times ``max(M, N)``.  Passing
    ``-1`` will use machine precision.

    .. versionchanged:: 2.0
        Previously, the default was ``-1``, but a warning was given that
        this would change.

Returns
-------
x : {(N,), (N, K)} ndarray
    Least-squares solution. If `b` is two-dimensional,
    the solutions are in the `K` columns of `x`.
residuals : {(1,), (K,), (0,)} ndarray
```

```
    Sums of squared residuals: Squared Euclidean 2-norm for each column in
    ``b - a @ x``.
    If the rank of `a` is < N or M <= N, this is an empty array.
    If `b` is 1-dimensional, this is a (1,) shape array.
    Otherwise the shape is (K,).
rank : int
    Rank of matrix `a`.
s : (min(M, N),) ndarray
    Singular values of `a`.

Raises
------
LinAlgError
    If computation does not converge.

See Also
--------
scipy.linalg.lstsq : Similar function in SciPy.

Notes
-----
If `b` is a matrix, then all array results are returned as matrices.

Examples
--------
Fit a line, ``y = mx + c``, through some noisy data-points:

>>> import numpy as np
>>> x = np.array([0, 1, 2, 3])
>>> y = np.array([-1, 0.2, 0.9, 2.1])

By examining the coefficients, we see that the line should have a
gradient of roughly 1 and cut the y-axis at, more or less, -1.

We can rewrite the line equation as ``y = Ap``, where ``A = [[x 1]]``
and ``p = [[m], [c]]``.  Now use `lstsq` to solve for `p`:

>>> A = np.vstack([x, np.ones(len(x))]).T
>>> A
array([[ 0.,  1.],
       [ 1.,  1.],
       [ 2.,  1.],
       [ 3.,  1.]])

>>> m, c = np.linalg.lstsq(A, y)[0]
>>> m, c
(1.0 -0.95) # may vary

Plot the data along with the fitted line:

>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(x, y, 'o', label='Original data', markersize=10)
>>> _ = plt.plot(x, m*x + c, 'r', label='Fitted line')
>>> _ = plt.legend()
>>> plt.show()
Class docstring:
Class to wrap functions with checks for __array_function__ overrides.

All arguments are required, and can only be passed by position.

Parameters
----------
dispatcher : function or None
    The dispatcher function that returns a single sequence-like object
    of all arguments relevant.  It must have the same signature (except
    the default values) as the actual implementation.
    If ``None``, this is a ``like=`` dispatcher and the
    ``_ArrayFunctionDispatcher`` must be called with ``like`` as the
    first (additional and positional) argument.
implementation : function
    Function that implements the operation on NumPy arrays without
    overrides.  Arguments passed calling the ``_ArrayFunctionDispatcher``
    will be forwarded to this (and the ``dispatcher``) as if using
    ``*args, **kwargs``.

Attributes
----------
_implementation : function
    The original implementation passed in.
```

In [140…
```
x = np.array([0, 1, 2, 3])
y = np.array([-1, 0.2, 0.9, 2.1])
```

In [141…
```
A = np.vstack([x, np.ones(len(x))]).T
A
```

```
Out[141]: array([[0., 1.],
                 [1., 1.],
                 [2., 1.],
                 [3., 1.]])
```

```
In [142…  m, c = np.linalg.lstsq(A, y)[0]
          m, c
```

```
Out[142]: (np.float64(0.9999999999999999), np.float64(-0.9499999999999997))
```

```
In [143…  from numpy.linalg import inv,qr,svd
          import scipy.linalg
          u=np.arange(1,6,1).reshape(5,1)
          M=u.dot(u.T)
          #print("Inverse of M is: ", inv(M)) # M is singular, so inverse does not exist

          q,r=qr(M)
          #print("matrix q (orthonormal matrix) in qr decomposition is:\n\n",q)
          #print("\n\nmatrix r (upper triangular) in qr decomposition is:\n\n",r)

          P,L,U=scipy.linalg.lu(M)

          #print("\n\nmatrix P in LU decomposition is:\n\n",P)
          #print("\n\nmatrix L in LU decomposition is:\n\n",L)
          #print("\n\n matrix U in LU decomposition is:\n\n",U)

          U,Sigma,V_transpose=svd(M)
          print("Sum of Singular Values of M is: ", np.sum(Sigma))
```

```
Sum of Singular Values of M is:  55.00000000000001
```

Pseudorandom Number Generation:

The numpy.random module supplements the built-in Python random with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions.

For example, you can get a 4 × 4 array of samples from the standard normal distribution using normal:

```
In [144…  np.random.normal(size=(4,4))
```

```
Out[144]: array([[-0.83707401, -0.36091976,  1.48349295,  0.7543416 ],
                 [-0.08569756,  0.13337818,  0.37060029, -1.24197548],
                 [ 1.6104294 , -0.22566355,  0.49117171,  1.39613021],
                 [ 0.68832785,  1.02909737, -1.5436696 , -0.05939883]])
```

We say that these are pseudorandom numbers because they are generated by an algorithm with deterministic behavior based on the seed of the random number generator. You can change NumPy’s random number generation seed using `np.random.seed`:

Seed is a global pseudo-random generator. However, randomstate is a pseudo-random generator isolated from others, which only impact specific variable.

```
In [145…  rng = np.random.RandomState(21212)
          rng.rand(4)
```

```
Out[145]: array([0.8394647 , 0.6312553 , 0.14601055, 0.33653325])
```

```
In [146…  rng = np.random.RandomState(21212)
          rng.rand(4)
```

```
Out[146]: array([0.8394647 , 0.6312553 , 0.14601055, 0.33653325])
```

```
In [147…  np.random.seed(2)
          np.random.rand(4)
```

```
Out[147]: array([0.4359949 , 0.02592623, 0.54966248, 0.43532239])
```

```
In [148…  np.random.seed(2)
          np.random.rand(4)
```

```
Out[148]: array([0.4359949 , 0.02592623, 0.54966248, 0.43532239])
```

```
In [149…  np.random.seed(2)
          np.random.rand(4)
```

```
Out[149]: array([0.4359949 , 0.02592623, 0.54966248, 0.43532239])
```

```
In [150…  np.random.seed(20)
          np.random.rand(4)
```

```
Out[150]: array([0.5881308 , 0.89771373, 0.89153073, 0.81583748])
```

By specifying a seed value, the function ensures that the sequence of random numbers generated remains the same across multiple runs, providing deterministic behavior and allowing reproducibility in random number generation.

In [151…

```
for _ in range(3):
    np.random.seed(1)
    s=np.random.normal(size=(2,2))
    print(s)
```

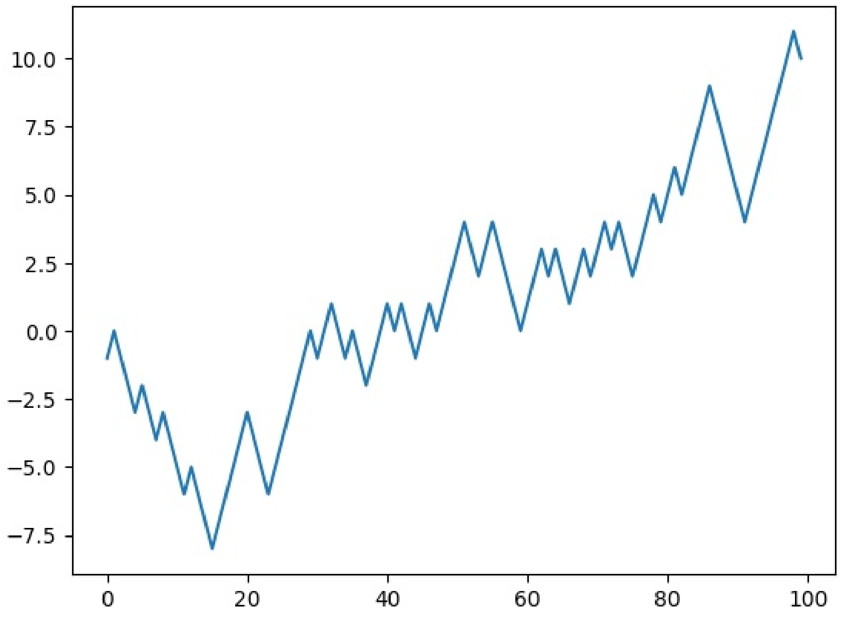
```
[[ 1.62434536 -0.61175641]
 [-0.52817175 -1.07296862]]
[[ 1.62434536 -0.61175641]
 [-0.52817175 -1.07296862]]
[[ 1.62434536 -0.61175641]
 [-0.52817175 -1.07296862]]
```

In [152…

```
# random walk
a=np.random.randint(0,2,size=100)
a=np.where(a>0,1,-1)
walk=a.cumsum()
import matplotlib.pyplot as plt
plt.plot(walk)
```

Out[152]:

```
[<matplotlib.lines.Line2D at 0x7ec385e01570>]
```



In [153…

```
(np.abs(walk)>10).argmax() # finding the index where walk array has aboslute value 10 "first" time
```

Out[153]:

```
np.int64(98)
```

In [154…

```
arr=np.array([1,2,3,4,5,6,7])
print((arr>3),(arr>3).argmax())
```

```
[False False False  True  True  True  True] 3
```

The `numpy.argmax()` function returns indices of the max element of the array in a particular axis.

In [155…

```
np.argmax(np.array([[1,2,5],[5,-1,3]]),axis=0)
```

Out[155]:

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

In [156…

```
np.random.normal(loc=0, scale=0.25,size=10)
```

Out[156]:

```
array([ 0.41495054,  0.18551104, -0.04795889, -0.22190724, -0.18678957,
        0.42311365,  0.01270194, -0.15924891,  0.04772887,  0.52506378])
```

seed --> Seed the random number generator
permutation --> Return a random permutation of a sequence, or return a permuted range
shuffle --> Randomly permute a sequence in-place
rand --> Draw samples from a uniform distribution
randint --> Draw random integers from a given low-to-high range
randn --> Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface)
binomial --> Draw samples from a binomial distribution
normal --> Draw samples from a normal (Gaussian) distribution
beta --> Draw samples from a beta distribution
chisquare --> Draw samples from a chi-square distribution
gamma --> Draw samples from a gamma distribution
uniform --> Draw samples from a uniform [0, 1) distribution

```
In [157…  np.random.seed(42)
          np.random.normal(loc=0, scale=0.25,size=10)

Out[157]:  array([ 0.12417854, -0.03456608,  0.16192213,  0.38075746, -0.05853834,
                  -0.05853424,  0.3948032 ,  0.19185868, -0.1173686 ,  0.13564001])

In [158…  np.random.permutation(10)

Out[158]:  array([6, 2, 8, 7, 9, 5, 1, 3, 0, 4])

In [159…  np.random.permutation(10)

Out[159]:  array([1, 5, 4, 8, 0, 7, 6, 3, 2, 9])

In [160…  np.random.permutation([1,4,9,2,3])

Out[160]:  array([2, 4, 9, 1, 3])

In [161…  np.random.rand(10)

Out[161]:  array([0.94888554, 0.96563203, 0.80839735, 0.30461377, 0.09767211,
                  0.68423303, 0.44015249, 0.12203823, 0.49517691, 0.03438852])

In [162…  np.random.randint(1,10)

Out[162]:  1

In [163…  np.random.randint(1,10)

Out[163]:  4

In [164…  np.random.randn(10)

Out[164]:  array([ 0.56611283, -0.70445345, -1.3779393 , -0.35311665, -0.46146572,
                   0.06665728, -0.17628566,  1.20089277,  0.69839894, -0.17162884])

In [165…  np.random.binomial(10,0.5)

Out[165]:  6

In [166…  np.random.normal(loc=0, scale=0.25,size=10)

Out[166]:  array([ 0.18646489, -0.45914581,  0.14111606,  0.00637517,  0.11829831,
                   0.16479765,  0.58518658,  0.2677463 ,  0.02410412,  0.10477553])

In [167…  np.random.chisquare( df=2, size=10 )

Out[167]:  array([0.64592551, 0.702731  , 0.36128659, 0.03151989, 1.10121812,
                  1.004662  , 0.69483068, 0.02835977, 0.4433952 , 2.48502504])

In [168…  np.random.beta( a=2.0, b=5.0, size=10 )

Out[168]:  array([0.22588393, 0.14591485, 0.25349367, 0.3210881 , 0.21662123,
                  0.33971623, 0.73947947, 0.20968991, 0.41613936, 0.19876706])

In [169…  np.random.gamma( shape=2.0, scale=1.0, size=10 )

Out[169]:  array([3.24391604, 1.41034735, 2.78711088, 1.35780113, 0.7766732 ,
                  1.09442459, 0.61419599, 3.14875069, 2.08672276, 0.40380842])

In [170…  np.random.uniform( low=0.0, high=1.0, size=10 )

Out[170]:  array([0.25391541, 0.24687606, 0.69630427, 0.71227059, 0.14808693,
                  0.99774049, 0.26678101, 0.97661496, 0.41103701, 0.03305073])

In [ ]:
```