numpy模块：

**rand:**

rand(...) method of mtrand.RandomState instance

rand(d0, d1, ..., dn)

Random values in a given shape.

Create an array of the given shape and populate it with

random samples from a uniform distribution

over ``[0, 1)``.

Parameters

----------

d0, d1, ..., dn : int, optional

The dimensions of the returned array, should all be positive.

If no argument is given a single Python float is returned.

Returns

-------

out : ndarray, shape ``(d0, d1, ..., dn)``

Random values.

See Also

--------

random

Notes

-----

This is a convenience function. If you want an interface that

takes a shape-tuple as the first argument, refer to

np.random.random\_sample .

Examples

--------

>>> np.random.rand(3,2)

array([[ 0.14022471, 0.96360618], #random

[ 0.37601032, 0.25528411], #random

[ 0.49313049, 0.94909878]]) #random

"sum":

Help on function sum in module numpy.core.fromnumeric:

sum(a, axis=None, dtype=None, out=None, keepdims=<class 'numpy.\_globals.\_NoValue'>)

Sum of array elements over a given axis.

Parameters

----------

a : array\_like

Elements to sum.

axis : None or int or tuple of ints, optional

Axis or axes along which a sum is performed. The default,

axis=None, will sum all of the elements of the input array. If

axis is negative it counts from the last to the first axis.

.. versionadded:: 1.7.0

If axis is a tuple of ints, a sum is performed on all of the axes

specified in the tuple instead of a single axis or all the axes as

before.

dtype : dtype, optional

The type of the returned array and of the accumulator in which the

elements are summed. The dtype of `a` is used by default unless `a`

has an integer dtype of less precision than the default platform

integer. In that case, if `a` is signed then the platform integer

is used while if `a` is unsigned then an unsigned integer of the

same precision as the platform integer is used.

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 output

values will be cast if necessary.

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 `sum` method of sub-classes of

`ndarray`, however any non-default value will be. If the

sub-classes `sum` method does not implement `keepdims` any

exceptions will be raised.

Returns

-------

sum\_along\_axis : ndarray

An array with the same shape as `a`, with the specified

axis removed. If `a` is a 0-d array, or if `axis` is None, a scalar

is returned. If an output array is specified, a reference to

`out` is returned.

See Also

--------

ndarray.sum : Equivalent method.

cumsum : Cumulative sum of array elements.

trapz : Integration of array values using the composite trapezoidal rule.

mean, average

Notes

-----

Arithmetic is modular when using integer types, and no error is

raised on overflow.

The sum of an empty array is the neutral element 0:

>>> np.sum([])

0.0

Examples

--------

>>> np.sum([0.5, 1.5])

2.0

>>> np.sum([0.5, 0.7, 0.2, 1.5], dtype=np.int32)

1

>>> np.sum([[0, 1], [0, 5]])

6

>>> np.sum([[0, 1], [0, 5]], axis=0)

array([0, 6])

>>> np.sum([[0, 1], [0, 5]], axis=1)

array([1, 5])

If the accumulator is too small, overflow occurs:

>>> np.ones(128, dtype=np.int8).sum(dtype=np.int8)

-128

"cumsum":

inf

Help on function cumsum in module numpy.core.fromnumeric:

cumsum(a, axis=None, dtype=None, out=None)

Return the cumulative sum of the elements along a given axis.

Parameters

----------

a : array\_like

Input array.

axis : int, optional

Axis along which the cumulative sum is computed. The default

(None) is to compute the cumsum over the flattened array.

dtype : dtype, optional

Type of the returned array and of the accumulator in which the

elements are summed. If `dtype` is not specified, it defaults

to the dtype of `a`, unless `a` has an integer dtype with a

precision less than that of the default platform integer. In

that case, the default platform integer is used.

out : ndarray, optional

Alternative output array in which to place the result. It must

have the same shape and buffer length as the expected output

but the type will be cast if necessary. See `doc.ufuncs`

(Section "Output arguments") for more details.

Returns

-------

cumsum\_along\_axis : ndarray.

A new array holding the result is returned unless `out` is

specified, in which case a reference to `out` is returned. The

result has the same size as `a`, and the same shape as `a` if

`axis` is not None or `a` is a 1-d array.

See Also

--------

sum : Sum array elements.

trapz : Integration of array values using the composite trapezoidal rule.

diff : Calculate the n-th discrete difference along given axis.

Notes

-----

Arithmetic is modular when using integer types, and no error is

raised on overflow.

Examples

--------

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

>>> a

array([[1, 2, 3],

[4, 5, 6]])

>>> np.cumsum(a)

array([ 1, 3, 6, 10, 15, 21])

>>> np.cumsum(a, dtype=float) # specifies type of output value(s)

array([ 1., 3., 6., 10., 15., 21.])

>>> np.cumsum(a,axis=0) # sum over rows for each of the 3 columns

array([[1, 2, 3],

[5, 7, 9]])

>>> np.cumsum(a,axis=1) # sum over columns for each of the 2 rows

array([[ 1, 3, 6],

[ 4, 9, 15]])

e.g:

计算向量欧式距离：

dist = numpy.sqrt(numpy.sum(numpy.square(vec1 - vec2)))

dist = numpy.linalg.norm(vec1 - vec2)

Help on ufunc object:

square = class ufunc(builtins.object)

| 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 outputs to be allocated by the ufunc.

| where : array\_like, optional

| Values of True indicate to calculate the ufunc at that position, values

| of False indicate to leave the value in the output alone.

| \*\*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, `r` will be equal to `out`. If the function has more than one

| output, then the result will be a tuple of arrays.

|

| Methods defined here:

|

| \_\_call\_\_(self, /, \*args, \*\*kwargs)

| Call self as a function.

|

| \_\_repr\_\_(self, /)

| Return repr(self).

|

| \_\_str\_\_(self, /)

| Return str(self).

|

| accumulate(...)

| accumulate(array, axis=0, dtype=None, out=None, keepdims=None)

|

| Accumulate the result of applying the operator to all elements.

|

| For a one-dimensional array, accumulate produces results equivalent to::

|

| r = np.empty(len(A))

| t = op.identity # op = the ufunc being applied to A's elements

| for i in range(len(A)):

| t = op(t, A[i])

| r[i] = t

| return r

|

| For example, add.accumulate() is equivalent to np.cumsum().

|

| For a multi-dimensional array, accumulate is applied along only one

| axis (axis zero by default; see Examples below) so repeated use is

| necessary if one wants to accumulate over multiple axes.

|

| Parameters

| ----------

| array : array\_like

| The array to act on.

| axis : int, optional

| The axis along which to apply the accumulation; default is zero.

| dtype : data-type code, optional

| The data-type used to represent the intermediate results. Defaults

| to the data-type of the output array if such is provided, or the

| the data-type of the input array if no output array is provided.

| out : ndarray, None, or tuple of ndarray and None, optional

| A location into which the result is stored. If not provided or `None`,

| a freshly-allocated array is returned. For consistency with

| :ref:`ufunc.\_\_call\_\_`, if given as a keyword, this may be wrapped in a

| 1-element tuple.

|

| .. versionchanged:: 1.13.0

| Tuples are allowed for keyword argument.

| keepdims : bool

| Has no effect. Deprecated, and will be removed in future.

|

| Returns

| -------

| r : ndarray

| The accumulated values. If `out` was supplied, `r` is a reference to

| `out`.

|

| Examples

| --------

| 1-D array examples:

|

| >>> np.add.accumulate([2, 3, 5])

| array([ 2, 5, 10])

| >>> np.multiply.accumulate([2, 3, 5])

| array([ 2, 6, 30])

|

| 2-D array examples:

|

| >>> I = np.eye(2)

| >>> I

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

| [ 0., 1.]])

|

| Accumulate along axis 0 (rows), down columns:

|

| >>> np.add.accumulate(I, 0)

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

| [ 1., 1.]])

| >>> np.add.accumulate(I) # no axis specified = axis zero

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

| [ 1., 1.]])

|

| Accumulate along axis 1 (columns), through rows:

|

| >>> np.add.accumulate(I, 1)

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

| [ 0., 1.]])

|

| at(...)

| at(a, indices, b=None)

|

| Performs unbuffered in place operation on operand 'a' for elements

| specified by 'indices'. For addition ufunc, this method is equivalent to

| `a[indices] += b`, except that results are accumulated for elements that

| are indexed more than once. For example, `a[[0,0]] += 1` will only

| increment the first element once because of buffering, whereas

| `add.at(a, [0,0], 1)` will increment the first element twice.

|

| .. versionadded:: 1.8.0

|

| Parameters

| ----------

| a : array\_like

| The array to perform in place operation on.

| indices : array\_like or tuple

| Array like index object or slice object for indexing into first

| operand. If first operand has multiple dimensions, indices can be a

| tuple of array like index objects or slice objects.

| b : array\_like

| Second operand for ufuncs requiring two operands. Operand must be

| broadcastable over first operand after indexing or slicing.

|

| Examples

| --------

| Set items 0 and 1 to their negative values:

|

| >>> a = np.array([1, 2, 3, 4])

| >>> np.negative.at(a, [0, 1])

| >>> print(a)

| array([-1, -2, 3, 4])

|

| ::

|

| Increment items 0 and 1, and increment item 2 twice:

|

| >>> a = np.array([1, 2, 3, 4])

| >>> np.add.at(a, [0, 1, 2, 2], 1)

| >>> print(a)

| array([2, 3, 5, 4])

|

| ::

|

| Add items 0 and 1 in first array to second array,

| and store results in first array:

|

| >>> a = np.array([1, 2, 3, 4])

| >>> b = np.array([1, 2])

| >>> np.add.at(a, [0, 1], b)

| >>> print(a)

| array([2, 4, 3, 4])

|

| outer(...)

| outer(A, B, \*\*kwargs)

|

| Apply the ufunc `op` to all pairs (a, b) with a in `A` and b in `B`.

|

| Let ``M = A.ndim``, ``N = B.ndim``. Then the result, `C`, of

| ``op.outer(A, B)`` is an array of dimension M + N such that:

|

| .. math:: C[i\_0, ..., i\_{M-1}, j\_0, ..., j\_{N-1}] =

| op(A[i\_0, ..., i\_{M-1}], B[j\_0, ..., j\_{N-1}])

|

| For `A` and `B` one-dimensional, this is equivalent to::

|

| r = empty(len(A),len(B))

| for i in range(len(A)):

| for j in range(len(B)):

| r[i,j] = op(A[i], B[j]) # op = ufunc in question

|

| Parameters

| ----------

| A : array\_like

| First array

| B : array\_like

| Second array

| kwargs : any

| Arguments to pass on to the ufunc. Typically `dtype` or `out`.

|

| Returns

| -------

| r : ndarray

| Output array

|

| See Also

| --------

| numpy.outer

|

| Examples

| --------

| >>> np.multiply.outer([1, 2, 3], [4, 5, 6])

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

| [ 8, 10, 12],

| [12, 15, 18]])

|

| A multi-dimensional example:

|

| >>> A = np.array([[1, 2, 3], [4, 5, 6]])

| >>> A.shape

| (2, 3)

| >>> B = np.array([[1, 2, 3, 4]])

| >>> B.shape

| (1, 4)

| >>> C = np.multiply.outer(A, B)

| >>> C.shape; C

| (2, 3, 1, 4)

| array([[[[ 1, 2, 3, 4]],

| [[ 2, 4, 6, 8]],

| [[ 3, 6, 9, 12]]],

| [[[ 4, 8, 12, 16]],

| [[ 5, 10, 15, 20]],

| [[ 6, 12, 18, 24]]]])

|

| reduce(...)

| reduce(a, axis=0, dtype=None, out=None, keepdims=False)

|

| Reduces `a`'s dimension by one, by applying ufunc along one axis.

|

| Let :math:`a.shape = (N\_0, ..., N\_i, ..., N\_{M-1})`. Then

| :math:`ufunc.reduce(a, axis=i)[k\_0, ..,k\_{i-1}, k\_{i+1}, .., k\_{M-1}]` =

| the result of iterating `j` over :math:`range(N\_i)`, cumulatively applying

| ufunc to each :math:`a[k\_0, ..,k\_{i-1}, j, k\_{i+1}, .., k\_{M-1}]`.

| For a one-dimensional array, reduce produces results equivalent to:

| ::

|

| r = op.identity # op = ufunc

| for i in range(len(A)):

| r = op(r, A[i])

| return r

|

| For example, add.reduce() is equivalent to sum().

|

| Parameters

| ----------

| a : array\_like

| The array to act on.

| axis : None or int or tuple of ints, optional

| Axis or axes along which a reduction is performed.

| The default (`axis` = 0) is perform a reduction over the first

| dimension of the input array. `axis` may be negative, in

| which case it counts from the last to the first axis.

|

| .. versionadded:: 1.7.0

|

| If this is `None`, a reduction is performed over all the axes.

| If this is a tuple of ints, a reduction is performed on multiple

| axes, instead of a single axis or all the axes as before.

|

| For operations which are either not commutative or not associative,

| doing a reduction over multiple axes is not well-defined. The

| ufuncs do not currently raise an exception in this case, but will

| likely do so in the future.

| dtype : data-type code, optional

| The type used to represent the intermediate results. Defaults

| to the data-type of the output array if this is provided, or

| the data-type of the input array if no output array is provided.

| out : ndarray, None, or tuple of ndarray and None, optional

| A location into which the result is stored. If not provided or `None`,

| a freshly-allocated array is returned. For consistency with

| :ref:`ufunc.\_\_call\_\_`, if given as a keyword, this may be wrapped in a

| 1-element tuple.

|

| .. versionchanged:: 1.13.0

| Tuples are allowed for keyword argument.

| 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 original `arr`.

|

| .. versionadded:: 1.7.0

|

| Returns

| -------

| r : ndarray

| The reduced array. If `out` was supplied, `r` is a reference to it.

|

| Examples

| --------

| >>> np.multiply.reduce([2,3,5])

| 30

|

| A multi-dimensional array example:

|

| >>> X = np.arange(8).reshape((2,2,2))

| >>> X

| array([[[0, 1],

| [2, 3]],

| [[4, 5],

| [6, 7]]])

| >>> np.add.reduce(X, 0)

| array([[ 4, 6],

| [ 8, 10]])

| >>> np.add.reduce(X) # confirm: default axis value is 0

| array([[ 4, 6],

| [ 8, 10]])

| >>> np.add.reduce(X, 1)

| array([[ 2, 4],

| [10, 12]])

| >>> np.add.reduce(X, 2)

| array([[ 1, 5],

| [ 9, 13]])

|

| reduceat(...)

| reduceat(a, indices, axis=0, dtype=None, out=None)

|

| Performs a (local) reduce with specified slices over a single axis.

|

| For i in ``range(len(indices))``, `reduceat` computes

| ``ufunc.reduce(a[indices[i]:indices[i+1]])``, which becomes the i-th

| generalized "row" parallel to `axis` in the final result (i.e., in a

| 2-D array, for example, if `axis = 0`, it becomes the i-th row, but if

| `axis = 1`, it becomes the i-th column). There are three exceptions to this:

|

| \* when ``i = len(indices) - 1`` (so for the last index),

| ``indices[i+1] = a.shape[axis]``.

| \* if ``indices[i] >= indices[i + 1]``, the i-th generalized "row" is

| simply ``a[indices[i]]``.

| \* if ``indices[i] >= len(a)`` or ``indices[i] < 0``, an error is raised.

|

| The shape of the output depends on the size of `indices`, and may be

| larger than `a` (this happens if ``len(indices) > a.shape[axis]``).

|

| Parameters

| ----------

| a : array\_like

| The array to act on.

| indices : array\_like

| Paired indices, comma separated (not colon), specifying slices to

| reduce.

| axis : int, optional

| The axis along which to apply the reduceat.

| dtype : data-type code, optional

| The type used to represent the intermediate results. Defaults

| to the data type of the output array if this is provided, or

| the data type of the input array if no output array is provided.

| out : ndarray, None, or tuple of ndarray and None, optional

| A location into which the result is stored. If not provided or `None`,

| a freshly-allocated array is returned. For consistency with

| :ref:`ufunc.\_\_call\_\_`, if given as a keyword, this may be wrapped in a

| 1-element tuple.

|

| .. versionchanged:: 1.13.0

| Tuples are allowed for keyword argument.

|

| Returns

| -------

| r : ndarray

| The reduced values. If `out` was supplied, `r` is a reference to

| `out`.

|

| Notes

| -----

| A descriptive example:

|

| If `a` is 1-D, the function `ufunc.accumulate(a)` is the same as

| ``ufunc.reduceat(a, indices)[::2]`` where `indices` is

| ``range(len(array) - 1)`` with a zero placed

| in every other element:

| ``indices = zeros(2 \* len(a) - 1)``, ``indices[1::2] = range(1, len(a))``.

|

| Don't be fooled by this attribute's name: `reduceat(a)` is not

| necessarily smaller than `a`.

|

| Examples

| --------

| To take the running sum of four successive values:

|

| >>> np.add.reduceat(np.arange(8),[0,4, 1,5, 2,6, 3,7])[::2]

| array([ 6, 10, 14, 18])

|

| A 2-D example:

|

| >>> x = np.linspace(0, 15, 16).reshape(4,4)

| >>> x

| array([[ 0., 1., 2., 3.],

| [ 4., 5., 6., 7.],

| [ 8., 9., 10., 11.],

| [ 12., 13., 14., 15.]])

|

| ::

|

| # reduce such that the result has the following five rows:

| # [row1 + row2 + row3]

| # [row4]

| # [row2]

| # [row3]

| # [row1 + row2 + row3 + row4]

|

| >>> np.add.reduceat(x, [0, 3, 1, 2, 0])

| array([[ 12., 15., 18., 21.],

| [ 12., 13., 14., 15.],

| [ 4., 5., 6., 7.],

| [ 8., 9., 10., 11.],

| [ 24., 28., 32., 36.]])

|

| ::

|

| # reduce such that result has the following two columns:

| # [col1 \* col2 \* col3, col4]

|

| >>> np.multiply.reduceat(x, [0, 3], 1)

| array([[ 0., 3.],

| [ 120., 7.],

| [ 720., 11.],

| [ 2184., 15.]])

|

| ----------------------------------------------------------------------

| Data descriptors defined here:

|

| identity

| The identity value.

|

| Data attribute containing the identity element for the ufunc, if it has one.

| If it does not, the attribute value is None.

|

| Examples

| --------

| >>> np.add.identity

| 0

| >>> np.multiply.identity

| 1

| >>> np.power.identity

| 1

| >>> print(np.exp.identity)

| None

|

| nargs

| The number of arguments.

|

| Data attribute containing the number of arguments the ufunc takes, including

| optional ones.

|

| Notes

| -----

| Typically this value will be one more than what you might expect because all

| ufuncs take the optional "out" argument.

|

| Examples

| --------

| >>> np.add.nargs

| 3

| >>> np.multiply.nargs

| 3

| >>> np.power.nargs

| 3

| >>> np.exp.nargs

| 2

|

| nin

| The number of inputs.

|

| Data attribute containing the number of arguments the ufunc treats as input.

|

| Examples

| --------

| >>> np.add.nin

| 2

| >>> np.multiply.nin

| 2

| >>> np.power.nin

| 2

| >>> np.exp.nin

| 1

|

| nout

| The number of outputs.

|

| Data attribute containing the number of arguments the ufunc treats as output.

|

| Notes

| -----

| Since all ufuncs can take output arguments, this will always be (at least) 1.

|

| Examples

| --------

| >>> np.add.nout

| 1

| >>> np.multiply.nout

| 1

| >>> np.power.nout

| 1

| >>> np.exp.nout

| 1

|

| ntypes

| The number of types.

|

| The number of numerical NumPy types - of which there are 18 total - on which

| the ufunc can operate.

|

| See Also

| --------

| numpy.ufunc.types

|

| Examples

| --------

| >>> np.add.ntypes

| 18

| >>> np.multiply.ntypes

| 18

| >>> np.power.ntypes

| 17

| >>> np.exp.ntypes

| 7

| >>> np.remainder.ntypes

| 14

|

| signature

|

| types

| Returns a list with types grouped input->output.

|

| Data attribute listing the data-type "Domain-Range" groupings the ufunc can

| deliver. The data-types are given using the character codes.

|

| See Also

| --------

| numpy.ufunc.ntypes

|

| Examples

| --------

| >>> np.add.types

| ['??->?', 'bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l',

| 'LL->L', 'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D',

| 'GG->G', 'OO->O']

|

| >>> np.multiply.types

| ['??->?', 'bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l',

| 'LL->L', 'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D',

| 'GG->G', 'OO->O']

|

| >>> np.power.types

| ['bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l', 'LL->L',

| 'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D', 'GG->G',

| 'OO->O']

|

| >>> np.exp.types

| ['f->f', 'd->d', 'g->g', 'F->F', 'D->D', 'G->G', 'O->O']

|

| >>> np.remainder.types

| ['bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l', 'LL->L',

| 'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'OO->O']