NumPy

`numpy.arange([start, ]stop, [step, ]dtype=None, *, like=None)`

Return evenly spaced values within a given interval.

`arange` can be called with a varying number of positional arguments:

- `arange(stop)`: Values are generated within the half-open interval `[0, stop)` (in other words, the interval including *start* but excluding *stop*).
- `arange(start, stop)`: Values are generated within the half-open interval `[start, stop)`.
- `arange(start, stop, step)` Values are generated within the half-open interval `[start, stop)`, with spacing between values given by `step`.

For integer arguments the function is roughly equivalent to the Python built-in **`range`**, but returns an ndarray rather than a `range` instance.

When using a non-integer step, such as 0.1, it is often better to use **`numpy.linspace`**.

See the Warning sections below for more information.

Parameters:

start : *integer or real, optional*

Start of interval. The interval includes this value. The default start value is 0.

stop : *integer or real*

End of interval. The interval does not include this value, except in some cases where *step* is not an integer and floating point round-off affects the length of *out*.

step : *integer or real, optional*

Spacing between values. For any output *out*, this is the distance between two adjacent values, `out[i+1] - out[i]`. The default step size is 1. If *step* is specified as a position argument, *start* must also be given.

dtype : *dtype, optional*

The type of the output array. If **`dtype`** is not given, infer the data type from the other input arguments.

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.

> *New in version 1.20.0.*

Returns:

arange : *ndarray*

Array of evenly spaced values.

For floating point arguments, the length of the result is `ceil((stop - start)/step)`. Because of floating point overflow, this rule may result in the last element of *out* being greater than *stop*.

Warning

The length of the output might not be numerically stable.

Another stability issue is due to the internal implementation of **numpy.arange**. The actual step value used to populate the array is `dtype(start + step) - dtype(start)` and not *step*. Precision loss can occur here, due to casting or due to using floating points when *start* is much larger than *step*. This can lead to unexpected behaviour. For example:

```
>>> np.arange(0, 5, 0.5, dtype=int)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
>>> np.arange(-3, 3, 0.5, dtype=int)
array([-3, -2, -1,  0,  1,  2,  3,  4,  5,  6,  7,  8])
```

In such cases, the use of **numpy.linspace** should be preferred.

The built-in **range** generates Python built-in integers that have arbitrary size, while **numpy.arange** produces **numpy.int32** or **numpy.int64** numbers. This may result in incorrect results for large integer values:

```
>>> power = 40
>>> modulo = 10000
>>> x1 = [(n ** power) % modulo for n in range(8)]
>>> x2 = [(n ** power) % modulo for n in np.arange(8)]
>>> print(x1)
[0, 1, 7776, 8801, 6176, 625, 6576, 4001]  # correct
>>> print(x2)
[0, 1, 7776, 7185, 0, 5969, 4816, 3361]  # incorrect
```

See also

numpy.linspace
Evenly spaced numbers with careful handling of endpoints.

numpy.ogrid
Arrays of evenly spaced numbers in N-dimensions.

numpy.mgrid
Grid-shaped arrays of evenly spaced numbers in N-dimensions.

How to create arrays with regularly-spaced values

Examples

```
>>> np.arange(3)
array([0, 1, 2])
>>> np.arange(3.0)
array([ 0.,  1.,  2.])
>>> np.arange(3,7)
array([3, 4, 5, 6])
>>> np.arange(3,7,2)
array([3, 5])
```

Previous
numpy.core.defchararray.asarray

Next
numpy.linspace

numpy.array(*object*, *dtype=None*, *, *copy=True*, *order='K'*, *subok=False*, *ndmin=0*, *like=None*)

Create an array.

Parameters:

object : *array_like*

An array, any object exposing the array interface, an object whose `__array__` method returns an array, or any (nested) sequence. If object is a scalar, a 0-dimensional array containing object is returned.

dtype : *data-type, optional*

The desired data-type for the array. If not given, NumPy will try to use a default `dtype` that can represent the values (by applying promotion rules when necessary.)

copy : *bool, optional*

If true (default), then the object is copied. Otherwise, a copy will only be made 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.).

order : *{'K', 'A', 'C', 'F'}, optional*

Specify the memory layout of the array. If object is not an array, the newly created array will be in C order (row major) unless 'F' is specified, in which case it will be in Fortran order (column major). If object is an array the following holds.

order	no copy	copy=True
'K'	unchanged	F & C order preserved, otherwise most similar order
'A'	unchanged	F order if input is F and not C, otherwise C order
'C'	C order	C order
'F'	F order	F order

When `copy=False` and a copy is made for other reasons, the result is the same as if `copy=True`, with some exceptions for 'A', see the Notes section. The default order is 'K'.

subok : *bool, optional*

If True, then sub-classes will be passed-through, otherwise the returned array will be forced to be a base-class array (default).

ndmin : *int, optional*

Specifies the minimum number of dimensions that the resulting array should have. Ones will be prepended to the shape as needed to meet this requirement.

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.

> *New in version 1.20.0.*

Returns:

out : *ndarray*

An array object satisfying the specified requirements.

> See also
>
> **`empty_like`**
> Return an empty array with shape and type of input.
>
> **`ones_like`**
> Return an array of ones with shape and type of input.
>
> **`zeros_like`**
> Return an array of zeros with shape and type of input.
>
> **`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.
>
> **`full`**
> Return a new array of given shape filled with value.

Notes

When order is 'A' and `object` is an array in neither 'C' nor 'F' order, and a copy is forced by a change in dtype, then the order of the result is not necessarily 'C' as expected. This is likely a bug.

Examples

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

Upcasting:

```
>>> np.array([1, 2, 3.0])
array([ 1.,  2.,  3.])
```

More than one dimension:

```
>>> np.array([[1, 2], [3, 4]])
array([[1, 2],
       [3, 4]])
```

Minimum dimensions 2:

```
>>> np.array([1, 2, 3], ndmin=2)
array([[1, 2, 3]])
```

Type provided:

```
>>> np.array([1, 2, 3], dtype=complex)
array([ 1.+0.j,  2.+0.j,  3.+0.j])
```

Data-type consisting of more than one element:

```
>>> x = np.array([(1,2),(3,4)],dtype=[('a','<i4'),('b','<i4')])
>>> x['a']
array([1, 3])
```

Creating an array from sub-classes:

```
>>> np.array(np.mat('1 2; 3 4'))
array([[1, 2],
       [3, 4]])
```

```
>>> np.array(np.mat('1 2; 3 4'), subok=True)
matrix([[1, 2],
        [3, 4]])
```

Previous
numpy.full_like

Next
numpy.asarray