# numpy.arange

**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,
```

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 i
>>> x2 = [(n ** power) % modulo for n i
>>> print(x1)
[0, 1, 7776, 8801, 6176, 625, 6576, 400
>>> print(x2)
[0, 1, 7776, 7185, 0, 5969, 4816, 3361]
```

### 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.

### 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.defchar

Next
numpy.linspace