A `range` is a Python object that represents an interval of integers. Usually, the numbers are consecutive, but you can also specify that you want to space them out. You can create ranges by calling `range()` with one, two, or three arguments, as the following examples show:

```python
>>> list(range(5))
[0, 1, 2, 3, 4]

>>> list(range(1, 7))
[1, 2, 3, 4, 5, 6]

>>> list(range(1, 20, 2))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

You can provide `range()` with one, two, or three **integer** arguments. This corresponds to three different use cases:

1. Ranges counting from zero
2. Ranges of consecutive numbers
3. Ranges stepping over numbers

Count From Zero

When you call `range()` with one argument, you create a range that counts from zero and up to, but not including, the number you provided:

Python

```
>>> range(5)
range(0, 5)
```

Here, you've created a range from zero to five. To see the individual elements in the range, you can use `list()` to convert the range to a list:

Python

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Inspecting `range(5)` shows that it contains the numbers zero, one, two, three, and four. Five itself is not a part of the range. One nice property of these ranges is that the argument, `5` in this case, is the same as the number of elements in the range.

Count From Start to Stop

You can call `range()` with two arguments. The first value will be the start of the range. As before, the range will count up to, but not include, the second value:

Python

```
>>> range(1, 7)
range(1, 7)
```

Observe that `range(1, 7)` starts at one and includes the consecutive numbers up to six. Seven is the limit of the range and isn't included. You can calculate the number of elements in a range by subtracting the start value from the end value. In this example, there are *7 - 1 = 6* elements.

Count From Start to Stop While Stepping Over Numbers

The final way to construct a `range` is by providing a third argument that specifies the step between elements in the range. By default, the step is one, but you can pass any non-zero integer. For example, you can represent all odd numbers below twenty as follows:

Python

```python
>>> range(1, 20, 2)
range(1, 20, 2)
```

Python

```python
>>> list(range(1, 20, 2))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

You can confirm that the range contains all odd numbers below twenty. The difference between consecutive elements in the range is two, which is equal to the step value that you provided as the third argument.

Python

```python
>>> range(1, 7)
range(1, 7)

>>> list(range(1, 7))
[1, 2, 3, 4, 5, 6]
```

```python
>>> range(-10, 0)
range(-10, 0)

>>> list(range(-10, 0))
[-10, -9, -8, -7, -6, -5, -4, -3, -2, -1]
```

```
>>> range(-7, -3)
range(-7, -3)

>>> list(range(-7, -3))
[-7, -6, -5, -4]
```

As always, the end value isn't included in the range. You can still calculate the number of elements by looking at the difference of the arguments. Just keep track of the negative signs: *(-3) - (-7) = 4*.

NumPy is the fundamental Python library for numerical computing. Its most important type is an **array type** called `ndarray`. NumPy offers a lot of array creation routines for different circumstances. `arange()` is one such function based on **numerical ranges**. It's often referred to as `np.arange()` because `np` is a widely used abbreviation for NumPy.

Creating NumPy arrays is important when you're working with other Python libraries that rely on them, like SciPy, Pandas, Matplotlib, scikit-learn, and more. NumPy is suitable for creating and working with arrays because it offers useful routines, enables performance boosts, and allows you to write concise code.

Return Value and Parameters of `np.arange()`

NumPy `arange()` is one of the array creation routines based on numerical ranges. It creates an instance of `ndarray` with *evenly spaced values* and returns the reference to it.

You can define the interval of the values contained in an array, space between them, and their type with four parameters of `arange()`:

```python
numpy.arange([start, ]stop, [step, ], dtype=None) -> numpy.ndarray
```

The first three parameters determine the range of the values, while the fourth specifies the type of the elements:

1. **`start`** is the number (integer or decimal) that defines the first value in the array.
2. **`stop`** is the number that defines the end of the array and isn't included in the array.
3. **`step`** is the number that defines the spacing (difference) between each two consecutive values in the array and defaults to `1`.
4. **`dtype`** is the type of the elements of the output array and defaults to `None`.

`step` can't be zero. Otherwise, you'll get a `ZeroDivisionError`. You can't move away anywhere from `start` if the increment or decrement is `0`.

If `dtype` is omitted, `arange()` will try to deduce the type of the array elements from the types of `start`, `stop`, and `step`.

Range Arguments of `np.arange()`

The arguments of NumPy `arange()` that define the values contained in the array correspond to the numeric parameters `start`, `stop`, and `step`. You have to pass *at least one* of them.

The following examples will show you how `arange()` behaves depending on the number of arguments and their values.

Providing All Range Arguments

When working with NumPy routines, you have to import NumPy first:

Python

```python
>>> import numpy as np
```

Now, you have NumPy imported and you're ready to apply `arange()`.

Let's see a first example of how to use NumPy `arange()`:

Python

```python
>>> np.arange(start=1, stop=10, step=3)
array([1, 4, 7])
```

In this example, `start` is `1`. Therefore, the first element of the obtained array is `1`. `step` is `3`, which is why your second value is 1+3, that is `4`, while the third value in the array is 4+3, which equals `7`.

Following this pattern, the next value would be `10` (7+3), but counting must be ended *before* `stop` is reached, so this one is not included.

You can pass `start`, `stop`, and `step` as positional arguments as well:

Python

```
>>> np.arange(1, 10, 3)
array([1, 4, 7])
```

This code sample is equivalent to, but more concise than the previous one.

The value of `stop` is not included in an array. That's why you can obtain identical results with different `stop` values:

Python

```
>>> np.arange(1, 8, 3)
array([1, 4, 7])
```

This code sample returns the array with the same values as the previous two. You can get the same result with any value of `stop` strictly greater than `7` and less than or equal to `10`.

This code sample returns the array with the same values as the previous two. You can get the same result with any value of `stop` strictly greater than `7` and less than or equal to `10`.

However, if you make `stop` greater than `10`, then counting is going to end after `10` is reached:

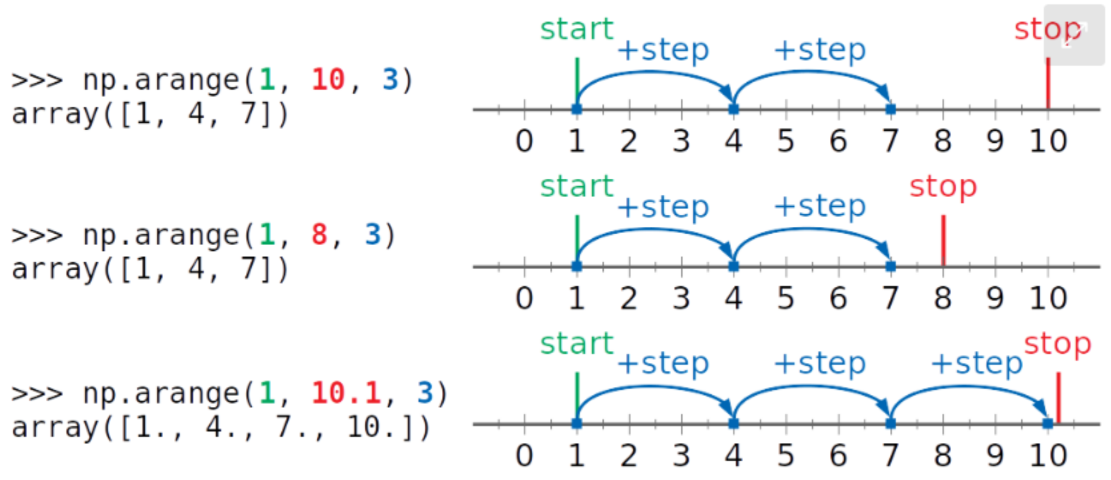
Python

```
>>> np.arange(1, 10.1, 3)
array([ 1.,  4.,  7., 10.])
```

In this case, you get the array with four elements that includes `10`.

Notice that this example creates an array of floating-point numbers, unlike the previous one.

You can see the graphical representations of these three examples in the figure below:



```
>>> np.arange(1, 10, 3)
array([1, 4, 7])

>>> np.arange(1, 8, 3)
array([1, 4, 7])

>>> np.arange(1, 10.1, 3)
array([1., 4., 7., 10.])
```

`start` is shown in green, `stop` in red, while `step` and the values contained in the arrays are blue.

As you can see from the figure above, the first two examples have three values (`1`, `4`, and `7`) counted. They don't allow `10` to be included. In the third example, `stop` is larger than `10`, and it is contained in the resulting array.