Python range(): Represent Numerical Ranges

by Geir Arne Hjelle

Nov 24, 2024 31m 9 Comments basics python

Mark as Completed

Share

Table of Contents

Help

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **The Python range() Function**

In Python, the `range()` function generates a sequence of numbers, often used in loops for iteration. By default, it creates numbers starting from 0 up to but not including a specified stop value. You can also reverse the sequence with `reversed()`. If you need to count backwards, then you can use a negative step, like `range(start, stop, -1)`, which counts down from `start` to `stop`.

The `range()` function is not just about iterating over numbers. It can also be used in various programming scenarios beyond simple loops. By mastering `range()`, you can write more efficient and readable Python code. Explore how `range()` can simplify your code and when alternatives might be more appropriate.

By the end of this tutorial, you'll understand that:

- A range in Python is an object representing an **interval of integers**, often used for looping.
- The `range()` function can be used to **generate sequences** of numbers that can be **converted to lists**.
- `for i in range(5)` is a loop that **iterates** over the numbers from 0 to 4, inclusive.
- The **range parameters** `start`, `stop`, and `step` define where the sequence begins, ends, and the interval between numbers.
- Ranges can go **backward** in Python by using a negative step value and **reversed** by using `reversed()`.

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]
```

In each example, you use `list()` to explicitly list the individual elements of each range. You'll study these examples in more detail later on.

A range can be an effective tool. However, throughout this tutorial, you'll also explore alternatives that may work better in some situations. You can click the link below to download the code that you'll see in this tutorial:

Get Your Code: Click here to download the free sample code that shows you how to represent numerical ranges in Python.

Construct Numerical Ranges

In Python, `range()` is **built in**. This means that you can always call `range()` without doing any preparations first. Calling `range()` constructs a **range object** that you can put to use. Later, you'll see practical examples of how to use range objects.

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

You'll learn how to use each of these next.

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)
```

The representation of a range object just shows you the arguments that you provided, so it's not super helpful in this case. You can use `list()` to inspect the individual elements:

Python

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

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

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

Here, you specify that you want a range of numbers from one to twenty that are two apart. Have a look at the numbers that are part of this range:

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.

In these examples, you've gotten started using ranges in Python. In the rest of the tutorial, you'll learn more about how `range()` works under the hood, and you'll see several examples of when you'd want to use `range` objects in your own code.

Use Python's `range()` Function to Create Specific Ranges

You've seen the syntax of `range()` and how you use one, two, or three arguments to specify different kinds of ranges. In this section, you'll dig deeper into Python's `range()` function and see how to represent specific ranges.

Note: Technically, `range()` is *not* a function. Instead, `range` is a **type** or **class**. When you call `range()`, you're calling the constructor of the `range` class to create a new range object.

Still, for all practical purposes, you can treat `range()` as a function that returns a `range` object.

First, note that a range is a lazy sequence. This means that Python doesn't create the individual elements of a range when you create the range. Instead, it creates each element when you ask for it, for example by iterating over the range.

This is different from, say, a list. Lists are eagerly constructed, so that all elements in the list are present in memory immediately when the list is created. That's why you converted ranges into lists to inspect the individual elements earlier:

Python

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

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

The `range` object is lazy. In this example, the individual numbers in the range are first referenced when you convert it to a list. Additionally, you don't exhaust ranges when you iterate over them. You can reuse them as many times as you want.

One property of ranges that may seem unintuitive is that the start value of the range is included, but the end value isn't. Python ranges represent half-open intervals, to use a technical term.

Closed intervals, which include both the start and end values, are more common in daily use. For example, a regular dice has values from one to six. However, using half-open intervals in programming has several advantages,which Edsger W. Dijkstra outlines as follows:

- If the start value isn't included in the range, then you'd need to use `-1` to specify a range running from zero.
- If the end value is included in the range, then it's awkward to define an empty range.
- If the range is defined as a half-open interval, then it's straightforward to calculate the number of elements in the range.

You've already seen a few examples of the latter point. Recall that `range(1, 7)` contains six elements which you calculate as *7 - 1 = 6*. Next, you'll explore how to create special kinds of ranges.

Handle Ranges Over Negative Numbers

Probably, most of the ranges that you'll construct will consist of positive numbers. If you only provide one argument when you create a range, then, as you've learned, the range counts from zero and up.

However, nothing's stopping you from using negative numbers in your ranges. For example, you can create the numbers from negative ten to zero:

Python

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

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

Using negative numbers as arguments works similarly to positive numbers. The first element in the range is the start, while the second one is the end, as usual. You can use negative values for both arguments. Here are the numbers from negative seven to negative three:

Python

```
>>> 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*.

Work With an Empty Range

You can use any integer as a value for the first two arguments. However, many choices will lead to empty ranges. In particular, if the arguments are equal, then you know that the corresponding range will have zero elements. In other words, it'll be empty:

Python

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

>>> list(range(1, 1))
[]
```

Here, you construct a range starting at `1`. Since `1` is also the end of the range, it's not included, so the range is empty.

In general, an empty range isn't particularly useful. You probably won't create an empty range explicitly. However, if you're working with ranges with dynamic arguments, then some of those arguments may lead to an empty range.

As you've learned, there are many ways you can create an empty range. If you need to create one explicitly, then the clearest is probably one counting from zero to zero:

Python

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

>>> list(range(0))
[]
```

You confirm that `range(0)` represents an empty range.

If your first argument is larger than the second one, then you'll typically end up with an empty range as well:

Python

```
>>> range(4, 2)
range(4, 2)

>>> list(range(4, 2))
[]
```

Here, you've constructed a range that runs from four and up to two. Since four is larger than two, there are no numbers in that range.

Sometimes, you want to count down from a larger number. That's what you'll explore next.

Count Backward With Negative Steps

So far, you've looked at ranges with positive steps. By default, ranges use a step equal to one, but you can use any integer except zero. The following range contains all even numbers from twenty and down to zero:

Python

```
>>> range(20, 0, -2)
range(20, 0, -2)

>>> list(range(20, 0, -2))
[20, 18, 16, 14, 12, 10, 8, 6, 4, 2]
```

By using a negative step, the range counts *down* from start to stop. As always, the stop isn't included.

Just as most positive ranges use the default step of one, a step value of negative one is common when counting backward:

Python

```
>>> range(5, -1, -1)
range(5, -1, -1)

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

This range represents counting down from five to zero, inclusive. If you use a negative step, then the range will be empty if the second argument is larger than or equal to the first.

You've seen that you can use any integer value for the arguments, although the last one can't be zero. In the next section, you'll look into some practical use cases where you can loop through a range, as well as situations in which an alternative may be more appropriate.

Loop Through Ranges or Use an Alternative

In Python, you can create loops using two different constructs:

1. A `while` loop, or an **indefinite** loop, repeats an operation until a condition is fulfilled.
2. A `for` loop, or a **definite** loop, repeats an operation for each element in an existing sequence.

Since a range is a sequence, you typically use `for` loops to iterate over ranges.

In many languages, including C++, Java, and JavaScript, `for` loops are mainly based on indices. Python loops are different. In Python, a `for` loop is based on sequence elements instead of indices. If you're more familiar with a different language, you may need to adjust how you approach loops.

In particular, if you want to re-create an index-based loop, then you may be tempted to use a range:

Python

```
>>> word = "Loop"
>>> for index in range(len(word)):
...     print(index, word[index])
...
0 L
1 o
2 o
3 p
```

You loop over indices and use each index to pick out the corresponding letter from the word `"Loop"`.

This approach works, but it's usually not the best way to work with loops in Python. In fact, if you're treating a range as indices in Python, then you should look for an alternative. There are better ways, which you'll explore in the upcoming sections. But first, you'll study a situation in which `range()` is the right tool for the job.

Repeat an Operation

If you need to repeat an operation a fixed number of times, then using a range is often a good solution. You know that `range(n)` is a sequence that contains `n` elements, so looping over such a range will repeat an operation `n` times.

For example, the following code block repeats the `print()` calls three times:

Python

```
>>> for _ in range(3):
...     print("Knock, knock, knock")
...     print("Penny!")
...
Knock, knock, knock
Penny!
Knock, knock, knock
Penny!
Knock, knock, knock
Penny!
```

In this code, you repeat the indented block for each element in the range. Because the elements of the range themselves are unimportant, you use the underscore (_) as a throwaway variable.

There are also times when you work with genuine ranges of numbers, and you want to create loops over those ranges. In the next example, you'll create a multiplication table that shows the products of all combinations of integers up to ten:

Python

```
>>> for number in range(1, 11):
...     for product in range(number, number * 11, number):
...         print(f"{product:>4d}", end="")
...     print()
...
   1   2   3   4   5   6   7   8   9  10
   2   4   6   8  10  12  14  16  18  20
   3   6   9  12  15  18  21  24  27  30
   4   8  12  16  20  24  28  32  36  40
   5  10  15  20  25  30  35  40  45  50
   6  12  18  24  30  36  42  48  54  60
   7  14  21  28  35  42  49  56  63  70
   8  16  24  32  40  48  56  64  72  80
   9  18  27  36  45  54  63  72  81  90
  10  20  30  40  50  60  70  80  90 100
```

You create two loops that together will set up the two-dimensional multiplication table. First, you loop over the numbers from one up to and including ten. These will represent the rows in the table, and you can see those numbers at the beginning of each row.

You use the second loop to populate each column of a given row. In this range, you use the arguments to calculate `number` times each integer from one to ten. In particular, the last step argument makes sure that numbers in each row are correctly spaced out.

To format the table, you use an f-string and the `end` parameter of `print()`, which keeps each number on the same line. In this example, you're working directly with the range of numbers, and it makes sense to use the range in a loop.

Loop Directly Over the Iterator Instead

Earlier, you used a range to construct the indices of a string. A small variation of that example gives you the individual characters of the string:

Python

```
>>> word = "Loop"
>>> for index in range(len(word)):
...     print(word[index])
...
L
o
o
p
```

In most loops, the index isn't necessary at all. If you're interested in the characters of a string, then you can take advantage of the string already being iterable. This means that you can loop directly on the string itself:

Python

```
>>> word = "Loop"
>>> for char in word:
...     print(char)
...
L
o
o
p
```

Looping directly on a sequence, like you do here, is simpler and more readable than using indices. If you have a loop where you're using indices to find individual elements, then you should loop directly on the elements instead.

Use `enumerate()` to Create Indices Instead

Sometimes, you want to work with both indices and the corresponding elements. In the earlier example, you showed the index of each character in a word:

Python

```
>>> word = "Loop"
>>> for index in range(len(word)):
...     print(index, word[index])
...
0 L
1 o
2 o
3 p
```

For use cases like these, you can use `enumerate()` to create indices instead of `range()`. With `enumerate()`, you wrap an iterable and get access to both the index and the element for each item:

Python

```
>>> word = "Loop"
>>> for index, char in enumerate(word):
...     print(index, char)
...
0 L
1 o
2 o
3 p
```

With `enumerate()`, you generate an index for each element. You can also customize the counter by passing the `start` argument to `enumerate()`. Then, the index will begin counting at `start` instead of zero, which is the default. For example, you can start counting letters at index one:

Python

```
>>> word = "Loop"
>>> for index, char in enumerate(word, start=1):
...     print(index, char)
...
1 L
2 o
3 o
4 p
```

Now the first letter is labeled by `1` instead of `0`.

The following example is a bit more involved. You have a grid representing a treasure map. The treasures are marked by `X`. The center of the map is at the coordinates (0, 0), and it's marked by `o`. You can use `enumerate()` to find the coordinates of the treasures as follows:

Python

```
>>> grid = """
... .............
... ........X...
... ...X..o.....
... .............
... ...........X.
... """.strip()
...
... rows = grid.split("\n")
... for row, line in enumerate(rows, start=-(len(rows) // 2)):
...     for col, char in enumerate(line, start=-(len(line) // 2)):
...         if char == "X":
...             print(f"Treasure found at ({row}, {col})")
...
Treasure found at (-1, 3)
Treasure found at (0, -3)
Treasure found at (2, 5)
```

You first loop over each line in the grid, using `enumerate()` to access the row coordinates. By setting `start=-(len(rows) // 2)`, you ensure that the middle row is labeled with index `0`. In this example, the first row gets row coordinate `-2`.

Similarly, you loop over each character of each row. Now, you start counting at `-(len(line) // 2)`, which in this example is `-6`. According to your program, one treasure is at `(-1, 3)` which translates to *one row above the center, three columns to the right of the center*. The other treasures are at `(0, -3)` and `(2, 5)`.

If you need to access both an element and its corresponding index, then you should use `enumerate()`.

Use `zip()` for Parallel Iteration Instead

If you need to loop over several sequences at the same time, then you can use indices to find elements corresponding to each other:

Python

```
>>> countries = ["Norway", "Canada", "Burkina Faso"]
>>> capitals = ["Oslo", "Ottawa", "Ouagadougou"]
>>> for index in range(len(countries)):
...     print(f"The capital of {countries[index]} is {capitals[index]}")
...
The capital of Norway is Oslo
The capital of Canada is Ottawa
The capital of Burkina Faso is Ouagadougou
```

Here, you use indices to look up corresponding elements in `countries` and `capitals`.

You're using `range()` to construct the indices. As mentioned earlier, there are better approaches in Python than working directly with indices. If you want to loop over several iterables in parallel, then you should use `zip()`.

With `zip()`, you can rewrite the previous example as follows:

Python

```
>>> countries = ["Norway", "Canada", "Burkina Faso"]
>>> capitals = ["Oslo", "Ottawa", "Ouagadougou"]
>>> for country, capital in zip(countries, capitals):
...     print(f"The capital of {country} is {capital}")
...
The capital of Norway is Oslo
The capital of Canada is Ottawa
The capital of Burkina Faso is Ouagadougou
```

Note that `zip()` generates a tuple that you can unpack to one loop variable for each sequence that you loop over. The code inside the loop becomes simpler and more readable when you don't need to deal with indices.

If you need to loop over two or more iterables at the same time, then you should use `zip()`.

Explore Other Features and Uses of Ranges

By now, you know how to construct ranges in Python and how you can loop over them. You've even seen some alternatives to using `range()` in specific use cases.

In this section, you'll take a closer look at the `range` object and learn which operations it supports. You'll learn that a range has many attributes and methods in common with tuples and lists, even though the range is lazy.

Access Individual Numbers of a Range

You can use Python's square bracket notation to pick out a single element from a range:

Python

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

>>> numbers[3]
7

>>> numbers[-2]
17
```

You first construct a range that contains the odd numbers below twenty. Then you pick out the number at index three. Since Python sequences are zero-indexed, this is the fourth odd number, namely seven. Finally, you pick out the second number from the end, which is seventeen.

Create Subranges With Slices

In addition to picking out single elements from a range, you can use slices to create new ranges. A slice can grab one or several elements from a sequence, and the operation uses similar parameters to `range()`.

In slice syntax, you use a colon (`:`) to separate arguments. Additionally, numbers specify start, stop, and optionally a step for the slice. A slice like `[1:5]` starts from index `1` and runs up to, but not including, index `5`. You can add a step at the end, so `[1:5:2]` will also run from index `1` to `5` but only include every second index.

If you apply a slice to a range, then you get a new range that will contain some or all of the elements of the original range:

Python

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

>>> numbers[1:5]
range(3, 11, 2)

>>> numbers[1:5:2]
range(3, 11, 4)
```

Again, you start with the odd numbers below twenty. Taking the slice `[1:5]` gives you a new range containing the odd numbers from three to nine, inclusive. If you add a step to your slice, then the step in the resulting range changes correspondingly.

Check Whether a Number Is a Member of a Range

Membership tests in ranges are fast. Sometimes you can check if a value is a member of a range instead of doing other kinds of validations.

For example, you can check if `year` is a leap year in the twenty-first century as follows:

Python

```
>>> year = 2023
>>> year in range(2000, 2100, 4)
False

>>> year = 2024
>>> year in range(2000, 2100, 4)
True
```

The leap years are every four years, so you check if `year` is in the range starting at 2000 and including every fourth year up to 2100.

Note: In general, leap year calculations are slightly more complicated. A leap year is a year that's a multiple of four, except for years evenly divisible by 100, but not by 400. In practice, those exceptions mean that 1900 and 2100 aren't leap years, but 2000 is a leap year.

You can always replace the range membership test with an equivalent logical condition:

Python

```
>>> year = 2023
>>> 2000 <= year < 2100 and year % 4 == 0
False

>>> year = 2024
>>> 2000 <= year < 2100 and year % 4 == 0
True
```

You use the modulus operator (`%`) to check that the year is divisible by four. In some cases, using `range()` is more readable than spelling out the corresponding equation.

One subtle detail with range membership tests is that all members of ranges are integers. This means that numbers with a decimal part are never range members:

Python

```
>>> number = 4.2

>>> number in range(1, 10)
False
```

Here, you check if `4.2` is part of the integer range starting at one and counting up to ten. While four is in the range, the decimal number is not. To get the same result with a logical test, you must remember to check that the number is divisible by one as well:

Python

```
>>> 1 <= number < 10
True

>>> 1 <= number < 10 and number % 1 == 0
False
```

You can use the modulo operator with the step value of the range to ensure that the number in question is consistent with the step. In general, if the start value isn't divisible by the step value, then you should subtract the start value from `number` before applying the modulo operator.

Calculate the Number of Elements in a Range

You've learned that for single-step ranges, you can calculate the number of elements in the range by taking the difference between the first two arguments. If the step is different from one, then the calculation is slightly more complicated. You need to do ceiling division with the step size:

Python

```
>>> import math
>>> start, stop, step = 1, 20, 2
>>> math.ceil((stop - start) / step)
10
```

In this example, you calculate that there are ten odd numbers below twenty. However, for `range` objects, you shouldn't do this calculation yourself. Instead, you should use `len()` to calculate the number of elements:

Python

```
>>> numbers = range(1, 20, 2)
>>> len(numbers)
10
```

As above, you confirm that there are ten odd numbers below twenty. You can use `len()` with all ranges, including empty ones:

Python

```
>>> empty = range(0)
>>> len(empty)
0
```

Earlier, you saw that an empty range has no elements. Consistently, `len()` confirms that its length is zero.

Reverse a Range

If you need to loop over a range in reverse, you can use `reversed()`. This function knows how to reverse many iterables, including ranges:

Python

```
>>> numbers = range(1, 20, 2)
>>> reversed(numbers)
<range_iterator object at 0x7f92b5050090>

>>> list(reversed(numbers))
[19, 17, 15, 13, 11, 9, 7, 5, 3, 1]
```

Again, you use the odd numbers below twenty to explore. Calling `reversed()` creates a `range_iterator` object that you can use in your loops. Listing the elements shows that the range has been reversed, with the odd numbers now appearing in descending order.

Unfortunately, `range_iterator` isn't a full `range` object, and it doesn't support many of the features that you've learned about lately. For example, you can't slice it or ask for its length:

Python

```
>>> reversed(numbers)[2:5]
Traceback (most recent call last):
  ...
TypeError: 'range_iterator' object is not subscriptable

>>> len(reversed(numbers))
Traceback (most recent call last):
  ...
TypeError: object of type 'range_iterator' has no len()
```

In practice, this is rarely a problem. For example, the reversed range has the same number of elements as the original range.

If you need a reversed range that retains all of its powers, you can construct it manually by calculating new arguments to pass to `range()`. As long as `step` is one or negative one, it's straightforward to reverse a range:

Python

```
>>> def reverse_range(rng):
...     return range(
...         rng.stop - rng.step,
...         rng.start - rng.step,
...         -rng.step,
...     )
...
>>> reverse_range(range(5, 0, -1))
range(1, 6)

>>> reverse_range(reverse_range(range(5, 0, -1)))
range(5, 0, -1)
```

You first create a function that can reverse a given range. One neat feature of ranges is that you can access the arguments used to create the range using the attributes `.start`, `.stop`, and `.step`.

To reverse a range, you use `.stop` for the first argument and `.start` for the second. Additionally, you reverse the sign of `.step`. To account for the end value not being included in the range, you need to adjust the first two arguments by subtracting `.step`.

To test `reverse_range()`, you first reverse the range that counts down from five to one, inclusive. This gives you a range that counts up from one to five, inclusive, just as it should.

A good test of any function that reverses a sequence is to apply it twice. This should always bring back the original sequence. In this case, you confirm that applying `reverse_range()` twice returns the original range.

This function doesn't work if the step is different from either one or negative one. In such ranges, it's harder to calculate the new start value because it depends on the last element of the range. For example, the last element of `range(1, 20, 4)` is seventeen. It's not immediately clear how that's related to the original arguments.

Note: You can find a more complicated version of `reverse_range()` in the supporting materials, that can handle any step size.

To take different step sizes into account, you can reverse the range with a `[::-1]` slice instead:

Python

```
>>> range(5, 0, -1)[::-1]
range(1, 6)
```

This construct supports all step sizes:

Python

```
>>> list(range(1, 20, 4))
[1, 5, 9, 13, 17]

>>> range(1, 20, 4)[::-1]
range(17, -3, -4)

>>> list(range(1, 20, 4)[::-1])
[17, 13, 9, 5, 1]

>>> range(1, 20, 4)[::-1][::-1]
range(1, 21, 4)
```

For these examples, you consider the range consisting of every fourth number from one up to, but not including, twenty. Since seventeen is the last number in the range, the reversed range starts from seventeen and counts down.

Note that applying `[::-1]` twice brings back the original range. Even though the stop values are different, the elements of `range(1, 20, 4)` and `range(1, 21, 4)` are the same.

In general, you should just use `reversed()` to reverse a range. If you need any special properties of the reversed `range` object, then you can use `[::-1]` or a similar calculation instead.

Create a Range Using Integer-Like Parameters

So far, you've used integers when setting up ranges. You can also use integer-like numbers like binary numbers or hexadecimal numbers instead:

Python

```
>>> range(0b110)
range(0, 6)

>>> range(0xeb)
range(0, 235)
```

Here, `0b110` is the binary representation of `6`, while `0xeb` is the hexadecimal representation of `235`.

It turns out that you can create ranges from self-defined integer-like numbers as well. To be integer-like, your class needs to define the special method `.__index__()` to convert your integer-like number into a regular integer.

For example, consider the special numbers called π-digits. These are all made up of the digits of the constant π, which is approximately 3.1415926. The first π-digit is 3, the next is 31, the third is 314, and so on. You'll create `PiDigits` to represent such π-digits:

Python pi_digits.py

```
from dataclasses import dataclass

@dataclass
class PiDigits:
    num_digits: int

    def __index__(self):
        return int("31415926535897932384626433832795"[:self.num_digits])
```

Store this class in a file named `pi_digits.py`. You can then import it and play with it:

Python

```
>>> from pi_digits import PiDigits

>>> PiDigits(3)
PiDigits(num_digits=3)

>>> int(PiDigits(3))
314

>>> range(PiDigits(3))
range(0, 314)
```

You first create the third π-digit, 314. Because you implemented `.__index__()`, you can convert `PiDigits(3)` to a regular integer with `int()` and use it directly as an argument to `range()`.

You can use π-digits for all arguments in `range()`:

Python

```
>>> range(PiDigits(1), PiDigits(6))
range(3, 314159)

>>> range(PiDigits(2), PiDigits(8), PiDigits(1))
range(31, 31415926, 3)

>>> len(range(PiDigits(2), PiDigits(8), PiDigits(1)))
10471965
```

The last length calculation confirms that the resulting range contains millions of numbers, as expected. Even though you have some flexibility in providing arguments to `range()`, they all need to be integer-like.

If you need more flexibility, like creating floating-point number ranges, then you have a few options. You can create a custom `FloatRange` class. An example of this is provided in the downloadable materials. If you use NumPy in your project, then you should use its `arange()` function instead.

Conclusion

You've taken a deep dive into `range()` and explored many of its features. While you've learned that ranges are most commonly used in loops, you've also seen some of the properties of `range` objects that can be useful outside loops as well.

In this tutorial, you've learned how to:

- Create `range` objects that represent ranges of **consecutive integers**
- Represent ranges of **spaced-out numbers** with a fixed step
- Decide when `range` is a **good solution** for your use case
- **Avoid `range`** in most loops

How do you use ranges in your own scripts and programs? Share your best tips with the community in the comments below.

Get Your Code: Click here to download the free sample code that shows you how to represent numerical ranges in Python.

Frequently Asked Questions

Now that you have some experience with the `range()` function in Python, you can use the questions and answers below to check your understanding and recap what you've learned. These frequently asked questions sum up the most important concepts you've covered in this tutorial. Click the *Show/Hide* toggle beside each question to reveal the answer.

What's the `range()` function in Python used for? Show/Hide

How do you reverse a range in Python? Show/Hide

Can you use `range()` with negative numbers in Python? Show/Hide

What are some alternatives to using `range()` in loops? Show/Hide

How can you create a range with non-integer-like numbers in Python? Show/Hide

About **Geir Arne Hjelle**

Geir Arne is an avid Pythonista and a member of the Real Python tutorial team.

» More about Geir Arne

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Adriana

Brenda

Bartosz

David

Jon

Joanna

Kate

Krystal

Leodanis

What Do You Think?

Rate this article:

LinkedIn Twitter Bluesky Facebook Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Topics: basics python

Recommended Video Course: The Python range() Function

Related Tutorials:

- Python while Loops: Repeating Tasks Conditionally
- Dictionaries in Python
- Python's tuple Data Type: A Deep Dive With Examples
- How to Iterate Through a Dictionary in Python
- Namespaces in Python

Learn Python	Courses & Paths	Community	Membership	Company
Start Here	Learning Paths	Podcast	Plans & Pricing	About Us
Learning Resources	Quizzes & Exercises	Newsletter	Team Plans	Team
Code Mentor	Browse Topics	Community Chat	For Business	Sponsorships
Python Reference	Workshops	Office Hours	For Schools	Careers
Support Center	Books	Learner Stories	Reviews	Press Kit
				Merch

Privacy Policy · Terms of Use · Security · Contact

Happy Pythoning!

Real Python
Online Python Training