Why does range(start, end) not include end? [duplicate]

Asked 14 years ago Modified 5 months ago Viewed 547k times

432

This question already has answers here:

Why are Python's slice and range upper-bound exclusive? (6 answers)

Closed last year.

```
>>> range(1,11)
```

gives you

```
[1,2,3,4,5,6,7,8,9,10]
```

Why not 1-11?

Did they just decide to do it like that at random or does it have some value I am not seeing?

python range

Share
Improve this question
Follow

edited Dec 21, 2010 at 23:07
moinudin
138k ● 45 ● 193 ● 216

asked Dec 21, 2010 at 22:45
MetaGuru
43.7k ● 68 ● 195 ● 300

21 read Dijkstra, ewd831 – SilentGhost Dec 21, 2010 at 22:52

21 Basically you are choosing one set of off-by-one bugs for another. One set are more likely to cause your loops to terminate early, the other is likely to cause an Exception (or buffer overflow in other languages). Once you have written a bunch of code, you will see that the choice of behaviour `range()` has makes sense much more often – John La Rooy Dec 21, 2010 at 23:34

51 Link to Dijkstra, ewd831: cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF – unutbu Dec 21, 2010 at 23:41

13 @andreasdr But even if the cosmetic argument is valid, doesn't Python's approach introduce a new problem of readability? In common-usage English the term "range" implies that something ranges *from* something *to* something -- like an interval. That len(list(range(1,2)))

returns 1 and len(list(range(2))) returns 2 is something you really have to learn to digest.
– armin Jul 24, 2016 at 14:11

8 If a person said they want a range of colors from green to red, then very few people would say they don't want red. So the Eng word range is not appropriate word. This is not going to change but i think this is a chink in the armor that python is a sensible language.
– Peter Moore Aug 6, 2020 at 13:28

11 Answers

Sorted by: Highest score (default)

323

Because it's more common to call `range(0, 10)` which returns `[0,1,2,3,4,5,6,7,8,9]` which contains 10 elements which equals `len(range(0, 10))`. There's a tendency in programming to use 0-based indexing.

Also, consider the following common code snippet:

```
for i in range(len(li)):
    pass
```

Could you see that if `range()` went up to exactly `len(li)` that this would be problematic? The programmer would need to explicitly subtract 1. This also follows the common trend of programmers preferring `for(int i = 0; i < 10; i++)` over `for(int i = 0; i <= 9; i++)`.

If you are calling range with a start of 1 frequently, you might want to define your own function:

```
>>> def range1(start, end):
...     return range(start, end+1)
...
>>> range1(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Share
Improve this answer
Follow

edited Jul 14 at 14:16
Rob Bednark
28k ● 25 ● 87 ● 128

answered Dec 21, 2010 at 22:48
moinudin
138k ● 45 ● 193 ● 216

83 If that were the reasoning wouldn't the parameters be `range(start, count)`?
– Mark Ransom Dec 21, 2010 at 22:50

6 @shogun The start value defaults to 0, i.e. `range(10)` is equivalent to `range(0, 10)`.
– moinudin Dec 21, 2010 at 22:57

10 Your `range1` will not work with ranges that have a different step size than `1`. – dimo414 May 21, 2015 at 5:09

7 You explain that range(x) should start with 0 and x will be the "length of the range". OK. But you didn't explain why range(x,y) should start with x and end with y-1. If the programmer wants a for-loop with i ranging from 1 to 3, he has to explicitly add 1. Is this really about convenience? – armin Jul 24, 2016 at 14:42

12 `for i in range(len(li)):` is rather an antipattern. One should use `enumerate`. – hans Mar 6, 2018 at 12:17

93

Although there are some useful algorithmic explanations here, I think it may help to add some simple 'real life' reasoning as to why it works this way, which I have found useful when introducing the subject to young newcomers:

With something like 'range(1,10)' confusion can arise from thinking that pair of parameters represents the "start and end".

It is actually start and "stop".

Now, if it ***were*** the "end" value then, yes, you might expect that number would be included as the final entry in the sequence. But it is not the "end".

Others mistakenly call that parameter "count" because if you only ever use 'range(n)' then it does, of course, iterate 'n' times. This logic breaks down when you add the start parameter.

So the key point is to remember its name: "**stop**". That means it is the point at which, when reached, iteration will stop immediately. Not ***after*** that point.

So, while "start" does indeed represent the first value to be included, on reaching the "stop" value it 'breaks' rather than continuing to process 'that one as well' before stopping.

One analogy that I have used in explaining this to kids is that, ironically, it is better behaved than kids! It doesn't stop *after* it supposed to - it stops immediately without finishing what it was doing. (They get this ;))

Another analogy - when you drive a car you don't ***pass*** a stop/yield/'give way' sign and end up with it sitting somewhere next to, or behind, your car. Technically you still haven't reached it when you do stop. It is not included in the 'things you passed on your journey'.

I hope some of that helps in explaining to Pythonitos/Pythonitas!

Share Improve this answer Follow

answered Jan 27, 2016 at 14:06

dingles

1,738 ● 1 ● 15 ● 11

4 This explanation is more intuitive. Thanks – Fred Oct 10, 2019 at 2:50

3 Love the stop sign analogy, sorry to steal it :) – coyote Sep 20, 2021 at 10:58

@nyholku Sorry for that! I have now deleted my comment. It was a harsh response to a harsh comment, which had already been deleted too, so keeping it is completely pointless, especially if it makes other people feel bad for nothing. Technologies should be tools for making great things rather than be apples of discord. – Arthur Khazbulatov Jun 13, 2023 at 8:58

@ArthurKhazbs that [deleting] was an act of integrity! Well done. I will delete my comment, though it was kind tongue in cheek. Have a nice day. :) – nyholku Jun 14, 2023 at 9:11

Please be my Python teacher! – mazal Sep 8 at 19:16

27

Exclusive ranges do have some benefits:

For one thing each item in `range(0,n)` is a valid index for lists of length `n`.

Also `range(0,n)` has a length of `n`, not `n+1` which an inclusive range would.

Share Improve this answer Follow

answered Dec 21, 2010 at 22:49
sepp2k
370k ● 56 ● 679 ● 681

This is a common explanation, but it is not a good one. If range is inclusive at both ends, users could easily use range(0, n-1) or range(1, n) to generate n numbers. – Feng Jiang Jan 16 at 4:43

23

It's also useful for splitting ranges; `range(a,b)` can be split into `range(a, x)` and `range(x, b)`, whereas with inclusive range you would write either `x-1` or `x+1`. While you rarely need to split ranges, you do tend to split lists quite often, which is one of the reasons slicing a list `l[a:b]` includes the a-th element but not the b-th. Then `range` having the same property makes it nicely consistent.

Share
Improve this answer
Follow

edited Apr 12, 2015 at 23:18
user200783
14.3k ● 15 ● 76 ● 142

answered Feb 17, 2013 at 9:34
xuanji
5,097 ● 3 ● 27 ● 35

20

It works well in combination with zero-based indexing and `len()`. For example, if you have 10 items in a list `x`, they are numbered 0-9. `range(len(x))` gives you 0-9.

Of course, people will tell you it's more Pythonic to do `for item in x` or `for index, item in enumerate(x)` rather than `for i in range(len(x))`.

Slicing works that way too: `foo[1:4]` is items 1-3 of `foo` (keeping in mind that item 1 is actually the second item due to the zero-based indexing). For consistency, they should both work the same way.

I think of it as: "the first number you want, followed by the first number you *don't* want." If you want 1-10, the first number you don't want is 11, so it's `range(1, 11)`.

If it becomes cumbersome in a particular application, it's easy enough to write a little helper function that adds 1 to the ending index and calls `range()`.

Share
Improve this answer
Follow

edited Mar 17, 2016 at 18:27

answered Dec 21, 2010 at 22:51
kindall
184k ● 36 ● 289 ● 318

1 Agree on slicing. `w = 'abc'; w[:] == w[0:len(w)]; w[:-1] == w[0:len(w)-1];`
– kevpie Dec 22, 2010 at 2:32

`def full_range(start,stop): return range(start,stop+1) ## helper function`
– Brent Bradburn Aug 17, 2011 at 0:44

12

The length of the range is the top value minus the bottom value.

It's very similar to something like:

```
for (var i = 1; i < 11; i++) {
    //i goes from 1 to 10 in here
}
```

in a C-style language.

Also like Ruby's range:

```
1...11 #this is a range from 1 to 10
```

However, Ruby recognises that many times you'll want to include the terminal value and offers the alternative syntax:

```
1..10 #this is also a range from 1 to 10
```

Share Improve this answer Follow

answered Dec 21, 2010 at 22:48
Skilldrick
70.7k ● 35 ● 180 ● 230

9

Consider the code

```
for i in range(10):
    print "You'll see this 10 times", i
```

The idea is that you get a list of length `y-x`, which you can (as you see above) iterate over.

Read up on [the python docs](#) for range - they consider for-loop iteration the primary usecase.

Share Improve this answer Follow

answered Dec 21, 2010 at 22:52

Robert
6,540 ● 3 ● 25 ● 26

2 Simplest explanation. logging in just to upvote – Nujufas Aug 26, 2022 at 9:07

6

Basically in python `range(n)` iterates `n` times, which is of exclusive nature that is why it does not give last value when it is being printed, we can create a function which gives inclusive value it means it will also print last value mentioned in range.

```
def main():
    for i in inclusive_range(25):
        print(i, sep=" ")


def inclusive_range(*args):
    numargs = len(args)
    if numargs == 0:
        raise TypeError("you need to write at least a value")
    elif numargs == 1:
        stop = args[0]
        start = 0
        step = 1
    elif numargs == 2:
        (start, stop) = args
        step = 1
    elif numargs == 3:
        (start, stop, step) = args
    else:
        raise TypeError("Inclusive range was expected at most 3 arguments,got
{}".format(numargs))
    i = start
    while i <= stop:
        yield i
        i += step
```

```
if __name__ == "__main__":
    main()
```

Share
Improve this answer
Follow

edited Aug 24, 2018 at 10:08
Саша Черных
2,803 ● 4 ● 28 ● 79

answered Aug 3, 2018 at 16:45
Ashish Dixit
61 ● 1 ● 2

To avoid the possible surprise of an endless loop, I suggest to improve this code so that it works also in case of a negative step value. – user7711283 Mar 11, 2022 at 19:18

5

The `range(n)` in python returns from 0 to n-1. Respectively, the `range(1,n)` from 1 to n-1. So, if you want to omit the first value and get also the last value (n) you can do it very simply using the following code.

```
for i in range(1, n + 1):
        print(i) #prints from 1 to n
```

Share Improve this answer Follow

answered Feb 3, 2021 at 8:28
Thunderarea
469 ● 5 ● 14

1 The OP knows how to obtain the extra value, they are asking about the reason it is not included by default. – mins Sep 25, 2022 at 15:37

1

It's just more convenient to reason about in many cases.

Basically, we could think of a range as an interval between `start` and `end`. If `start <= end`, the length of the interval between them is `end - start`. If `len` was actually defined as the length, you'd have:

```
len(range(start, end)) == start - end
```

However, we count the integers included in the range instead of measuring the length of the interval. To keep the above property true, we should include one of the endpoints and exclude the other.

Adding the `step` parameter is like introducing a unit of length. In that case, you'd expect

```
len(range(start, end, step)) == (start - end) / step
```

for length. To get the count, you just use integer division.

Share Improve this answer Follow

answered May 29, 2019 at 17:35
Arseny
963 ● 9 ● 18

5 These defenses of Python's inconsistency are hilarious. If I wanted the interval between two numbers, why would I use subtraction to get the difference instead of the interval? It's inconsistent to use different indexing conventions for start and end positions. Why would you need to write "5:22" in order to get positions 5 to 21? – bzip2 Jan 17, 2020 at 22:33

It's not Python's, it's pretty common across the board. In C, Java, Ruby, you name it – Arseny Jan 17, 2020 at 22:52

I meant to say that it's common for indexing, not that the other languages necessarily have the same exact kind of object – Arseny Apr 9, 2020 at 18:56

@Arseny in defence of Ruby, this is not true. You can construct inclusive and exclusive ranges in Ruby: `(3..5).include?(5) => true` but `(3...5).include?(5) => false`. Array slicing is explicit and inclusive: `[0,1,2,3,4].slice(0,2) => [0, 1]`. You can even construct open ranges: `r = 42..; r.include?(Float::INFINITY) => true` – Andreas Gebhard Feb 23, 2022 at 16:04

@AndreasGebhard, no doubt there are cases when that's convenient. Scala, for instance, has both `a to b` and `a until b`. My point is that excluding the right end of the range is common practice and isn't an inconsistency whatsoever. Also, historically, the `<` comparison is faster for the processor than the `<=` comparison – Arseny Feb 23, 2022 at 18:29

0

Two major uses of ranges in python. All things tend to fall in one or the other

1. integer. Use built-in: `range(start, stop, step)`. To have `stop` included would mean that the end step would be assymetric for the general case. Consider `range(0,5,3)`. If default behaviour would output 5 at the end, it would be broken.
2. floating pont. This is for numerical uses (where sometimes it happens to be integers too). Then use `numpy.linspace`.

Share Improve this answer Follow

answered Dec 19, 2022 at 12:27
Stefan Karlsson
1,085 ● 9 ● 24

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.