Explain slice notation

I need a good explanation (references are a plus) on Python's slice notation.

To me, this notation needs a bit of picking up.

It looks extremely powerful, but I haven't quite got my head around it.

python list slice

edited Jun 3 at 19:30

martineau
49.7k 6 70 119

asked Feb 3 '09 at 22:31

Simon
26k 23 75 110

23 Answers

It's pretty simple really:

```
a[start:end] # items start through end-1
a[start:]    # items start through the rest of the array
a[:end]      # items from the beginning through end-1
a[:]         # a copy of the whole array
```

There is also the `step` value, which can be used with any of the above:

```
a[start:end:step] # start through not past end, by step
```

The key point to remember is that the `:end` value represents the first value that is *not* in the selected slice. So, the difference beween `end` and `start` is the number of elements selected (if `step` is 1, the default).

The other feature is that `start` or `end` may be a *negative* number, which means it counts from the end of the array instead of the beginning. So:

```
a[-1]    # last item in the array
a[-2:]   # last two items in the array
a[:-2]   # everything except the last two items
```

Python is kind to the programmer if there are fewer items than you ask for. For example, if you ask for `a[:-2]` and `a` only contains one element, you get an empty list instead of an error. Sometimes you would prefer the error, so you have to be aware that this may happen.

edited Feb 3 '09 at 23:39

answered Feb 3 '09 at 22:48

Greg Hewgill
567k 119 934 1089

The tutorial talks about it:

http://docs.python.org/tutorial/introduction.html#strings

(Scroll down a bit until you get to the part about slicing.)

The ASCII art diagram is helpful too for remembering how slices work:

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

> "One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0."

answered Feb 3 '09 at 22:49

Hans Nowak
3,471 1 10 9

For slices with negative steps, I find this ASCII art confusing, and it should extend to `-6` since `'ApleH'[:-6:-1]` is a valid slice and different from using `-5` – Chris_Rands Jul 26 at 15:30

Enumerating the possibilities allowed by the grammar:

```
>>> seq[:]                # [seq[0],   seq[1],          ..., seq[-1]    ]
>>> seq[low:]             # [seq[low], seq[low+1],      ..., seq[-1]    ]
>>> seq[:high]            # [seq[0],   seq[1],          ..., seq[high-1]]
>>> seq[low:high]         # [seq[low], seq[low+1],      ..., seq[high-1]]
>>> seq[::stride]         # [seq[0],   seq[stride],     ..., seq[-1]    ]
>>> seq[low::stride]      # [seq[low], seq[low+stride], ..., seq[-1]    ]
>>> seq[:high:stride]     # [seq[0],   seq[stride],     ..., seq[high-1]]
>>> seq[low:high:stride]  # [seq[low], seq[low+stride], ..., seq[high-1]]
```

Of course, if `(high-low)%stride != 0`, then the end point will be a little lower than `high-1`.

Extended slicing (with commas and ellipses) are mostly used only by special data structures (like Numpy); the basic sequences don't support them.

```
>>> class slicee:
...     def __getitem__(self, item):
...         return `item`
...
>>> slicee()[0, 1:2, ::5, ...]
'(0, slice(1, 2, None), slice(None, None, 5), Ellipsis)'
```

edited Apr 19 '12 at 3:37

answered Feb 3 '09 at 23:08

ephemient
132k 28 198 329

The answers above don't discuss slice assignment:

```
>>> r=[1,2,3,4]
>>> r[1:1]
[]
>>> r[1:1]=[9,8]
>>> r
[1, 9, 8, 2, 3, 4]
>>> r[1:1]=['blah']
>>> r
[1, 'blah', 9, 8, 2, 3, 4]
```

This may also clarify the difference between slicing and indexing.

edited Sep 4 '11 at 4:27

TomRoche
787 7 21

answered Jan 18 '11 at 21:37

David M. Perlman
1,921 1 9 9

Explain Python's slice notation

In short, the colons (`:`) in subscript notation (`subscriptable[subscriptarg]`) make slice notation - which has the optional arguments, `start` , `stop` , `step` :

```
sliceable[start:stop:step]
```

Python slicing is a computationally fast way to methodically access parts of your data. In my opinion, to be even an intermediate Python programmer, it's one aspect of the language that it is necessary to be familiar with.

Important Definitions

To begin with, let's define a few terms:

> **start:** the beginning index of the slice, it will include the element at this index unless it is the same as *stop*, defaults to 0, i.e. the first index. If it's negative, it means to start `n` items from the end.
>
> **stop:** the ending index of the slice, it does *not* include the element at this index, defaults to length of the sequence being sliced, that is, up to and including the end.
>
> **step:** the amount by which the index increases, defaults to 1. If it's negative, you're slicing over the iterable in reverse.

How Indexing Works

You can make any of these positive or negative numbers. The meaning of the positive numbers is straightforward, but for negative numbers, just like indexes in Python, you count backwards from the end for the *start* and *stop*, and for the *step*, you simply decrement your index. This example is from the documentation's tutorial, but I've modified it slightly to indicate which item in a sequence each index references:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
  0   1   2   3   4   5
 -6  -5  -4  -3  -2  -1
```

How Slicing Works

To use slice notation with a sequence that supports it, you must include at least one colon in the square brackets that follow the sequence (which actually implement the `__getitem__` method of the sequence, according to the Python data model.)

Slice notation works like this:

```
sequence[start:stop:step]
```

And recall that there are defaults for *start*, *stop*, and *step*, so to access the defaults, simply leave out the argument.

Slice notation to get the last nine elements from a list (or any other sequence that supports it, like a string) would look like this:

```
my_list[-9:]
```

When I see this, I read the part in the brackets as "9th from the end, to the end." (Actually, I abbreviate it mentally as "-9, on")

Explanation:

The full notation is

```
my_list[-9:None:None]
```

and to substitute the defaults (actually when `step` is negative, `stop` 's default is `-len(my_list) - 1` , so `None` for stop really just means it goes to whichever end step takes it to):

```
my_list[-9:len(my_list):1]
```

The **colon**, `:` , is what tells Python you're giving it a slice and not a regular index. That's why the idiomatic way of making a shallow copy of lists in Python 2 is

```
list_copy = sequence[:]
```

And clearing them is with:

```
del my_list[:]
```

(Python 3 gets a `list.copy` and `list.clear` method.)

Give your slices a descriptive name!

You may find it useful to separate forming the slice from passing it to the `list.__getitem__` method (that's what the square brackets do). Even if you're not new to it, it keeps your code more readable so that others that may have to read your code can more readily understand what you're doing.

However, you can't just assign some integers separated by colons to a variable. You need to use the slice object:

```
last_nine_slice = slice(-9, None)
```

The second argument, `None`, is required, so that the first argument is interpreted as the `start` argument otherwise it would be the `stop` argument.

You can then pass the slice object to your sequence:

```
>>> list(range(100))[last_nine_slice]
[91, 92, 93, 94, 95, 96, 97, 98, 99]
```

Memory Considerations:

Since slices of Python lists create new objects in memory, another important function to be aware of is `itertools.islice`. Typically you'll want to iterate over a slice, not just have it created statically in memory. `islice` is perfect for this. A caveat, it doesn't support negative arguments to `start`, `stop`, or `step`, so if that's an issue you may need to calculate indices or reverse the iterable in advance.

```
>>> length = 100
>>> last_nine_iter = itertools.islice(list(range(length)), length-9, None, 1)
>>> list_last_nine = list(last_nine)
>>> list_last_nine
[91, 92, 93, 94, 95, 96, 97, 98, 99]
```

The fact that list slices make a copy is a feature of lists themselves. If you're slicing advanced objects like a Pandas DataFrame, it may return a view on the original, and not a copy.

edited Dec 7 '16 at 4:52

answered Jul 12 '14 at 13:19

Aaron Hall ♦
92k 27 201 190

And a couple of things that weren't immediately obvious to me when I first saw the slicing syntax:

```
>>> x = [1,2,3,4,5,6]
>>> x[::-1]
[6,5,4,3,2,1]
```

Easy way to reverse sequences!

And if you wanted, for some reason, every second item in the reversed sequence:

```
>>> x = [1,2,3,4,5,6]
>>> x[::-2]
[6,4,2]
```

answered Feb 3 '09 at 23:15

Dana
16.1k 13 50 69

Found this great table at
http://wiki.python.org/moin/MovingToPythonFromOtherLanguages

```
Python indexes and slices for a six-element list.
Indexes enumerate the elements, slices enumerate the spaces between the elements.

Index from rear:    -6  -5  -4  -3  -2  -1      a=[0,1,2,3,4,5]    a[1:]==[1,2,3,4,5]
Index from front:    0   1   2   3   4   5      len(a)==6          a[:5]==[0,1,2,3,4]
                   +---+---+---+---+---+---+    a[0]==0           a[:-2]==[0,1,2,3]
                   | a | b | c | d | e | f |    a[5]==5           a[1:2]==[1]
                   +---+---+---+---+---+---+    a[-1]==5          a[1:-1]==[1,2,3,4]
Slice from front:  :   1   2   3   4   5   :    a[-2]==4
Slice from rear:   :  -5  -4  -3  -2  -1   :

                                                b=a[:]
                                                b==[0,1,2,3,4,5] (shallow copy of a)
```

answered Sep 6 '11 at 6:50

AdrianoFerrari

1,609 14 14

In Python 2.7

Slicing in Python

```
[a:b:c]

len = length of string, tuple or list

c -- default is +1. The sign of c indicates forward or backward, absolute value of c
indicates steps. Default is forward with step size 1. Positive means forward, negative
means backward.

a --  When c is positive or blank, default is 0. When c is negative, default is -1.

b --  When c is positive or blank, default is len. When c is negative, default is -
(len+1).
```

Understanding index assignment is very important.

```
In forward direction, starts at 0 and ends at len-1

In backward direction, starts at -1 and ends at -len
```

When you say [a:b:c], you are saying depending on the sign of c (forward or backward), start at a and end at b (excluding element at bth index). Use the indexing rule above and remember you will only find elements in this range:

```
-len, -len+1, -len+2, ..., 0, 1, 2,3,4 , len -1
```

But this range continues in both directions infinitely:

```
...,-len -2 ,-len-1,-len, -len+1, -len+2, ..., 0, 1, 2,3,4 , len -1, len, len +1, len+2 ,
....
```

For example:

```
            0   1   2   3   4   5   6   7   8   9   10   11
            a   s   t   r   i   n   g
    -9  -8  -7  -6  -5  -4  -3  -2  -1
```

If your choice of a, b, and c allows overlap with the range above as you traverse using rules for a,b,c above you will either get a list with elements (touched during traversal) or you will get an empty list.

One last thing: if a and b are equal, then also you get an empty list:

```
>>> l1
[2, 3, 4]

>>> l1[:]
[2, 3, 4]

>>> l1[::-1] # a default is -1 , b default is -(len+1)
[4, 3, 2]

>>> l1[:-4:-1] # a default is -1
[4, 3, 2]

>>> l1[:-3:-1] # a default is -1
[4, 3]

>>> l1[::] # c default is +1, so a default is 0, b default is len
[2, 3, 4]

>>> l1[::-1] # c is -1 , so a default is -1 and b default is -(len+1)
[4, 3, 2]


>>> l1[-100:-200:-1] # Interesting
[]

>>> l1[-1:-200:-1] # Interesting
[4, 3, 2]


>>> l1[-1:-1:1]
[]


>>> l1[-1:5:1] # Interesting
[4]


>>> l1[1:-7:1]
```

```
[]

>>> l1[1:-7:-1] # Interesting
[3, 2]

>>> l1[:-2:-2] # a default is -1, stop(b) at -2 , step(c) by 2 in reverse direction
[4]
```

edited Jul 10 at 16:59

answered Oct 22 '12 at 5:33

abc

6,530 20 72 120

1 another one interesting example: `a = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]; a[:-2:-2]` which results to `[9]` – Deviacium Jul 10 at 13:59

After using it a bit I realise that the simplest description is that it is exactly the same as the arguments in a for loop...

```
(from:to:step)
```

any of them are optional

```
(:to:step)
(from::step)
(from:to)
```

then the negative indexing just needs you to add the length of the string to the negative indices to understand it.

This works for me anyway...

answered Feb 19 '09 at 20:52

Simon

26k 23 75 110

I find it easier to remember how it's works, then I can figure out any specific start/stop/step combination.

It's instructive to understand `range()` first:

```
def range(start=0, stop, step=1):  # illegal syntax, but that's the effect
    i = start
    while (i < stop if step > 0 else i > stop):
        yield i
        i += step
```

Begin from `start`, increment by `step`, do not reach `stop`. Very simple.

The thing to remember about negative step is that `stop` is always the excluded end, whether it's higher or lower. If you want same slice in opposite order, it's much cleaner to do the reversal separately: e.g. `'abcde'[1:-2][::-1]` slices off one char from left, two from right, then reverses. (See also `reversed()`.)

Sequence slicing is same, except it first normalizes negative indexes, and can never go outside the sequence:

TODO: The code below had a bug with "never go outside the sequence" when abs(step)>1; I *think* I patched it to be correct, but it's hard to understand.

```
def this_is_how_slicing_works(seq, start=None, stop=None, step=1):
    if start is None:
        start = (0 if step > 0 else len(seq)-1)
    elif start < 0:
        start += len(seq)
    if not 0 <= start < len(seq):  # clip if still outside bounds
        start = (0 if step > 0 else len(seq)-1)
    if stop is None:
        stop = (len(seq) if step > 0 else -1)  # really -1, not last element
    elif stop < 0:
        stop += len(seq)
    for i in range(start, stop, step):
        if 0 <= i < len(seq):
            yield seq[i]
```

Don't worry about the `is None` details - just remember that omitting `start` and/or `stop` always does the right thing to give you the whole sequence.

Normalizing negative indexes first allows start and/or stop to be counted from the end independently: `'abcde'[1:-2] == 'abcde'[1:3] == 'bc'` despite `range(1,-2) == []`. The normalization is sometimes thought of as "modulo the length" but note it adds the length just once: e.g. `'abcde'[-53:42]` is just the whole string.

edited Oct 30 '16 at 12:42

answered Mar 29 '12 at 10:15

Beni Cherniavsky-Paskin

5,700 27 37

2 The `this_is_how_slicing_works` is not the same as python slice. E.G. `[0, 1, 2][-5:3:3]` will get [0] in python, but `list(this_is_how_slicing_works([0, 1, 2], -5, 3, 3))` get [1]. – Eastsun Oct 29 '16 at 12:56

@Eastsun Oops, you're right! A clearer case: `range(4)[-200:200:3] == [0, 3]` but `list(this_is_how_slicing_works([0, 1, 2, 3], -200, 200, 3)) == [2]`. My `if 0 <= i < len(seq):` was an attempt to implement "never go outside the sequence" simply but is wrong for step>1. I'll rewrite it later today (with tests). – Beni Cherniavsky-Paskin Oct 30 '16 at 12:36

```
Index:
      ------------>
  0   1   2   3   4
+---+---+---+---+---+
| a | b | c | d | e |
+---+---+---+---+---+
  0  -4  -3  -2  -1
     <------------

Slice:
    <---------------|
|--------------->
:   1   2   3   4   :
+---+---+---+---+---+
| a | b | c | d | e |
+---+---+---+---+---+
:  -4  -3  -2  -1   :
|--------------->
    <---------------|
```

I hope this will help you to model the list in Python.

Reference: http://wiki.python.org/moin/MovingToPythonFromOtherLanguages

edited Feb 11 at 19:56

Peter Mortensen

11.3k 16 77 110

answered Feb 4 '13 at 7:20

xiaoyu

3,592 1 10 11

Thanks for enumerating the indices; I was confused by there isn't a "-0" but this clears it, such power :D – harshvchawla May 18 at 3:48

Python slicing notation:

```
a[start:end:step]
```

- For `start` and `end`, negative values are interpreted as being relative to the end of the sequence.
- Positive indices for `end` indicate the position *after* the last element to be included.
- Blank values are defaulted as follows: `[+0:-0:1]`.
- Using a negative step reverses the interpretation of `start` and `end`

The notation extends to (numpy) matrices and multidimensional arrays. For example, to slice entire columns you can use:

```
m[::,0:2:] ## slice the first two columns
```

Slices hold references, not copies, of the array elements. If you want to make a separate copy an array, you can use `deepcopy()`.

edited May 23 at 12:34

Community ♦

1 1

answered Apr 28 '13 at 19:49

nobar

17.2k 8 69 72

I use the "an index points between elements" method of thinking about it myself, but one way

of describing it which sometimes helps others get it is this:

```
mylist[X:Y]
```

X is the index of the first element you want.
Y is the index of the first element you *don't* want.

answered Feb 6 '09 at 21:16

Steve Losh
15.6k 2 41 43

This is just for some extra info... Consider the list below

```
>>> l=[12,23,345,456,67,7,945,467]
```

Few other tricks for reversing the list:

```
>>> l[len(l):-len(l)-1:-1]
[467, 945, 7, 67, 456, 345, 23, 12]

>>> l[:-len(l)-1:-1]
[467, 945, 7, 67, 456, 345, 23, 12]

>>> l[len(l)::-1]
[467, 945, 7, 67, 456, 345, 23, 12]

>>> l[::-1]
[467, 945, 7, 67, 456, 345, 23, 12]

>>> l[-1:-len(l)-1:-1]
[467, 945, 7, 67, 456, 345, 23, 12]
```

See abc's answer above

edited Mar 30 '15 at 19:22

abc
6,530 20 72 120

answered Mar 22 '12 at 17:20

Arindam Roychowdhury
940 1 15 30

You can also use slice assignment to remove one or more elements from a list:

```
r = [1, 'blah', 9, 8, 2, 3, 4]
>>> r[1:4] = []
>>> r
[1, 2, 3, 4]
```

edited Apr 19 '13 at 16:28

answered Apr 5 '13 at 1:59

dansalmo
6,369 4 30 43

As a general rule, writing code with a lot of hardcoded index values leads to a readability and maintenance mess. For example, if you come back to the code a year later, you'll look at it and wonder what you were thinking when you wrote it. The solution shown is simply a way of more clearly stating what your code is actually doing. In general, the built-in slice() creates a slice object that can be used anywhere a slice is allowed. For example:

```
>>> items = [0, 1, 2, 3, 4, 5, 6]
>>> a = slice(2, 4)
>>> items[2:4]
[2, 3]
>>> items[a]
[2, 3]
>>> items[a] = [10,11]
>>> items
[0, 1, 10, 11, 4, 5, 6]
>>> del items[a]
>>> items
[0, 1, 4, 5, 6]
```

If you have a slice instance s, you can get more information about it by looking at its s.start, s.stop, and s.step attributes, respectively. For example:

```
>>> a = slice(10, 50, 2)
>>> a.start
10
>>> a.stop
50
```

```
>>> a.step
2
>>>
```

answered Dec 7 '13 at 16:52

Python_Dude
329 3 10

This is how I teach slices to newbies:

Understanding difference between indexing and slicing:

Wiki Python has this amazing picture which clearly distinguishes indexing and slicing.

```
Index from rear:    -6  -5  -4  -3  -2  -1
Index from front:    0   1   2   3   4   5
                   +---+---+---+---+---+---+
                   | a | b | c | d | e | f |
                   +---+---+---+---+---+---+
Slice from front:  :   1   2   3   4   5   :
Slice from rear:   :  -5  -4  -3  -2  -1   :
```

It is a list with 6 elements in it. To understand slicing better, consider that list as a set of six boxes placed together. Each box has an alphabet in it.

Indexing is like dealing with the contents of box. You can check contents of any box. But You can't check contents of multiple boxes at once. You can even replace contents of the box. But You can't place 2 balls in 1 box or replace 2 balls at a time.

```
In [122]: alpha = ['a', 'b', 'c', 'd', 'e', 'f']

In [123]: alpha
Out[123]: ['a', 'b', 'c', 'd', 'e', 'f']

In [124]: alpha[0]
Out[124]: 'a'

In [127]: alpha[0] = 'A'

In [128]: alpha
Out[128]: ['A', 'b', 'c', 'd', 'e', 'f']

In [129]: alpha[0,1]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-129-c7eb16585371> in <module>()
----> 1 alpha[0,1]

TypeError: list indices must be integers, not tuple
```

Slicing is like dealing with boxes itself. You can pickup first box and place it on another table. To pickup the box all You need to know is the position of beginning & ending of the box.

You can even pickup first 3 boxes or last 2 boxes or all boxes between 1 & 4. So, You can pick any set of boxes if You know beginning & ending. This positions are called start & stop positions.

The interesting thing is that You can replace multiple boxes at once. Also You can place multiple boxes where ever You like.

```
In [130]: alpha[0:1]
Out[130]: ['A']

In [131]: alpha[0:1] = 'a'

In [132]: alpha
Out[132]: ['a', 'b', 'c', 'd', 'e', 'f']

In [133]: alpha[0:2] = ['A', 'B']

In [134]: alpha
Out[134]: ['A', 'B', 'c', 'd', 'e', 'f']

In [135]: alpha[2:2] = ['x', 'xx']

In [136]: alpha
Out[136]: ['A', 'B', 'x', 'xx', 'c', 'd', 'e', 'f']
```

Slicing With Step:

Till now You have picked boxes continuously. But some times You need to pickup discretely. For example You can pickup every second box. You can even pickup every third box from the end. This value is called step size. This represents the gap between Your successive pickups. The step size should be positive if You are picking boxes from the beginning to end and vice versa.

```
In [137]: alpha = ['a', 'b', 'c', 'd', 'e', 'f']

In [142]: alpha[1:5:2]
Out[142]: ['b', 'd']

In [143]: alpha[-1:-5:-2]
Out[143]: ['f', 'd']

In [144]: alpha[1:5:-2]
Out[144]: []

In [145]: alpha[-1:-5:2]
Out[145]: []
```

How Python Figures Out Missing Parameters:

When slicing if You leave out any parameter, Python tries to figure it out automatically.

If You check source code of CPython, You will find a function called PySlice_GetIndicesEx which figures out indices to a slice for any given parameters. Here is the logical equivalent code in Python.

This function takes a Python object & optional parameters for slicing and returns start, stop, step & slice length for the requested slice.

```
def py_slice_get_indices_ex(obj, start=None, stop=None, step=None):

    length = len(obj)

    if step is None:
        step = 1
    if step == 0:
        raise Exception("Step cannot be zero.")

    if start is None:
        start = 0 if step > 0 else length - 1
    else:
        if start < 0:
            start += length
        if start < 0:
            start = 0 if step > 0 else -1
        if start >= length:
            start = length if step > 0 else length - 1

    if stop is None:
        stop = length if step > 0 else -1
    else:
        if stop < 0:
            stop += length
        if stop < 0:
            stop = 0 if step > 0 else -1
        if stop >= length:
            stop = length if step > 0 else length - 1

    if (step < 0 and stop >= start) or (step > 0 and start >= stop):
        slice_length = 0
    elif step < 0:
        slice_length = (stop - start + 1)/(step) + 1
    else:
        slice_length = (stop - start - 1)/(step) + 1

    return (start, stop, step, slice_length)
```

This is the intelligence that is present behind slices. Since Python has inbuilt function called slice, You can pass some parameters & check how smartly it calculates missing parameters.

```
In [21]: alpha = ['a', 'b', 'c', 'd', 'e', 'f']

In [22]: s = slice(None, None, None)

In [23]: s
Out[23]: slice(None, None, None)

In [24]: s.indices(len(alpha))
Out[24]: (0, 6, 1)

In [25]: range(*s.indices(len(alpha)))
Out[25]: [0, 1, 2, 3, 4, 5]

In [26]: s = slice(None, None, -1)

In [27]: range(*s.indices(len(alpha)))
Out[27]: [5, 4, 3, 2, 1, 0]
```

```
In [28]: s = slice(None, 3, -1)

In [29]: range(*s.indices(len(alpha)))
Out[29]: [5, 4]
```

Note: This post is originally written in my blog http://www.avilpage.com/2015/03/a-slice-of-python-intelligence-behind.html

answered Mar 24 '15 at 16:08

ChillarAnand
8,856 2 35 63

---

1. Slice Notation

To make it simple, remember **slice has only one form：**

```
s[start:end:step]
```

and here is how it works:

- `s` : an object that can be sliced
- `start` : first index to start iteration
- `end` : last index, **NOTE that `end` index will not be included in the resulted slice**
- `step` : pick element every `step` index

Another import thing: **all `start` , `end` , `step` can be omitted!** And if they are omitted, their default value will be used: `0` , `len(s)` , `1` accordingly.

So possible variations are:

```
# mostly used variations
s[start:end]
s[start:]
s[:end]

# step related variations
s[:end:step]
s[start::step]
s[::step]

# make a copy
s[:]
```

NOTE: If `start>=end` (considering only when `step>0`), python will return a empty slice `[]` .

2. Pitfalls

The above part explains the core features on how slice works, it will work on most occasions. However there can be pitfalls you should watch out, and this part explains them.

Negative indexes

The very first thing confuses python learners is that **index can be negative!** Don't panic: **negative index means count from backwards.**

For example:

```
s[-5:]    # start at the 5th index from the end of array,
          # thus returns the last 5 elements
s[:-5]    # start at index 0, end until the 5th index from end of array,
          # thus returns s[0:len(s)-5]
```

Negative step

Make things more confusing is that `step` **can be negative too!**

Negative step means iterate the array backwards: from end to start, with end index included, and start index excluded from result.

NOTE: when step is negative, the default value for `start` to `len(s)` (while `end` does not equal to `0` , because `s[::-1]` contains `s[0]`). For example:

```
s[::-1]            # reversed slice
s[len(s)::-1]      # same as above, reversed slice
s[0:len(s):-1]     # empty list
```

Out of range error?

Be surprised: **slice does not raise IndexError when index is out of range!**

If the index is out of range, python will try its best set the index to `0` or `len(s)` according to the situation. For example:

```
s[:len(s)+5]      # same as s[:len(s)]
s[-len(s)-5::]    # same as s[0:]
s[len(s)+5::-1]   # same as s[len(s)::-1], same as s[::-1]
```

3. Examples

Let's finish this answer with examples explains everything we have discussed:

```
# create our array for demonstration
In [1]: s = [i for i in range(10)]

In [2]: s
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: s[2:]   # from index 2 to last index
Out[3]: [2, 3, 4, 5, 6, 7, 8, 9]

In [4]: s[:8]   # from index 0 up to index 8
Out[4]: [0, 1, 2, 3, 4, 5, 6, 7]

In [5]: s[4:7]  # from index 4(included) up to index 7(excluded)
Out[5]: [4, 5, 6]

In [6]: s[:-2]  # up to second last index(negative index)
Out[6]: [0, 1, 2, 3, 4, 5, 6, 7]

In [7]: s[-2:]  # from second last index(negative index)
Out[7]: [8, 9]

In [8]: s[::-1] # from last to first in reverse order(negative step)
Out[8]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

In [9]: s[::-2] # all odd numbers in reversed order
Out[9]: [9, 7, 5, 3, 1]

In [11]: s[-2::-2] # all even numbers in reversed order
Out[11]: [8, 6, 4, 2, 0]

In [12]: s[3:15]   # end is out of range, python will set it to len(s)
Out[12]: [3, 4, 5, 6, 7, 8, 9]

In [14]: s[5:1]    # start > end, return empty list
Out[14]: []

In [15]: s[11]     # access index 11(greater than len(s)) will raise IndexError
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-15-79ffc22473a3> in <module>()
----> 1 s[11]

IndexError: list index out of range
```

edited Jan 18 at 19:03
pylang
6,230 1 15 35

answered Jan 9 at 12:52
cizixs
2,558 2 19 37

Seems like you have a mistake here. You claim `s[::-1]` is the same as `s[len(s):0:-1]` but when i try with `s=range(10)` i see that the latter leaves off the "0" from the list. – MrFlick Jan 13 at 17:12

@MrFlick You're correct! I made a mistake there, just updated. – cizixs Jan 16 at 2:43

To get a certain piece of an iterable (like a list), here is an example:

```
variable[number1:number2]
```

In this example, a positive number for number 1 is how many components you take off the front. A negative number is the exact opposite, how many you keep from the end. A positive number for number 2 indicates how many components you intend to keep from the beginning, and a negative is how many you intend to take off from the end. This is somewhat counter intuitive, but you are correct in supposing that list slicing is extremely useful.

edited Dec 28 '14 at 19:11
Obito
405 3 8

answered Nov 22 '13 at 2:53
someone-or-other
1,381 8 37

9 What distinctive new information does this answer provide? When a question has as many answers as this one does, especially highly voted answers, a new answer needs to provide distinctive new information to be worth adding. See also Is it worthwhile to finish your answer if multiple good answers are given while you're typing yours? on Meta Stack Overflow. – Jonathan Leffler Nov 22 '13 at 4:16

My brain seems happy to accept that `lst[start:end]` contains the `start` -th item. I might even say that it is a 'natural assumption'.

But occasionally a doubt creeps in and my brain asks for reassurance that it does not contain the `end` -th element.

In these moments I rely on this simple theorem:

```
for any n,    lst = lst[:n] + lst[n:]
```

This pretty property tells me that `lst[start:end]` does not contain the `end` -th item because it is in `lst[end:]` .

Note that this theorem is true for any `n` at all. For example, you can check that

```
lst = range(10)
lst[:-42] + lst[-42:] == lst
```

returns `True` .

answered May 26 '16 at 8:16

Robert
496 4 11

The answers above don't discuss multi-dimentional array slicing:

Slicing also apply to multi-dimentional arrays.

```
>>> a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> a[:2,0:3:2]
array([[1, 3],
       [5, 7]])
```

The ":2" before comma operates on the first dimension and the "0:3:2" after the comma operates on the second dimension.

answered Mar 1 at 2:31

Statham
544 7 15

You should identify which modules/packages implement this. The question was just tagged python and list. – hpaulj Jun 20 at 0:36

the famous numpy – Statham Jun 21 at 1:33

```
#!/usr/bin/env python

def slicegraphical(s, lista):

    if len(s) > 9:
        print """Enter a string of maximum 9 characters,
    so the printig would looki nice"""
        return 0;
    # print " ",
    print '  '+'+---' * len(s) +'+'
    print ' ',
    for letter in s:
        print '| {}'.format(letter),
    print '|'
    print " ",; print '+---' * len(s) +'+'

    print " ",
    for letter in range(len(s) +1):
        print '{}  '.format(letter),
    print ""
    for letter in range(-1*(len(s)), 0):
        print ' {}'.format(letter),
    print ''
    print ''
```

```
    for triada in lista:
        if len(triada) == 3:
            if triada[0]==None and triada[1] == None and triada[2] == None:
                # 000
                print s+'[   :   :   ]' +' = ', s[triada[0]:triada[1]:triada[2]]
            elif triada[0] == None and triada[1] == None and triada[2] != None:
                # 001
                print s+'[   :   :{0:2d} ]'.format(triada[2], '','') +' = ',
s[triada[0]:triada[1]:triada[2]]
            elif triada[0] == None and triada[1] != None and triada[2] == None:
                # 010
                print s+'[   :{0:2d} :   ]'.format(triada[1]) +' = ',
s[triada[0]:triada[1]:triada[2]]
            elif triada[0] == None and triada[1] != None and triada[2] != None:
                # 011
                print s+'[   :{0:2d} :{1:2d} ]'.format(triada[1], triada[2]) +' = ',
s[triada[0]:triada[1]:triada[2]]
            elif triada[0] != None and triada[1] == None and triada[2] == None:
                # 100
                print s+'[{0:2d} :   :   ]'.format(triada[0]) +' = ',
s[triada[0]:triada[1]:triada[2]]
            elif triada[0] != None and triada[1] == None and triada[2] != None:
                # 101
                print s+'[{0:2d} :   :{1:2d} ]'.format(triada[0], triada[2]) +' = ',
s[triada[0]:triada[1]:triada[2]]
            elif triada[0] != None and triada[1] != None and triada[2] == None:
                # 110
                print s+'[{0:2d} :{1:2d} :   ]'.format(triada[0], triada[1]) +' = ',
s[triada[0]:triada[1]:triada[2]]
            elif triada[0] != None and triada[1] != None and triada[2] != None:
                # 111
                print s+'[{0:2d} :{1:2d} :{2:2d} ]'.format(triada[0], triada[1],
triada[2]) +' = ', s[triada[0]:triada[1]:triada[2]]

        elif len(triada) == 2:
            if triada[0] == None and triada[1] == None:
                # 00
                print s+'[   :   ]    ' + ' = ', s[triada[0]:triada[1]]
            elif triada[0] == None and triada[1] != None:
                # 01
                print s+'[   :{0:2d} ]    '.format(triada[1]) + ' = ',
s[triada[0]:triada[1]]
            elif triada[0] != None and triada[1] == None:
                # 10
                print s+'[{0:2d} :   ]    '.format(triada[0]) + ' = ',
s[triada[0]:triada[1]]
            elif triada[0] != None and triada[1] != None:
                # 11
                print s+'[{0:2d} :{1:2d} ]    '.format(triada[0],triada[1]) + ' = ',
s[triada[0]:triada[1]]

        elif len(triada) == 1:
            print s+'[{0:2d} ]        '.format(triada[0]) + ' = ', s[triada[0]]


if __name__ == '__main__':
    # Change "s" to what ever string you like, make it 9 characters for
    # better representation.
    s = 'COMPUTERS'

    # add to this list different lists to experement with indexes
    # to represent ex. s[::], use s[None, None,None], otherwise you get an error
    # for s[2:] use s[2:None]

    lista = [[4,7],[2,5,2],[-5,1,-1],[4],[-4,-6,-1], [2,-3,1],[2,-3,-1], [None,None,-1],
[-5,None],[-5,0,-1],[-5,None,-1],[-1,1,-2]]

    slicegraphical(s, lista)
```

You can run this script and experiment with it, below is some samples that I got from the script.

```
  +---+---+---+---+---+---+---+---+---+
  | C | O | M | P | U | T | E | R | S |
  +---+---+---+---+---+---+---+---+---+
  0   1   2   3   4   5   6   7   8   9
 -9  -8  -7  -6  -5  -4  -3  -2  -1

COMPUTERS[ 4 : 7 ]     =  UTE
COMPUTERS[ 2 : 5 : 2 ] =  MU
COMPUTERS[-5 : 1 :-1 ] =  UPM
COMPUTERS[ 4 ]         =  U
COMPUTERS[-4 :-6 :-1 ] =  TU
COMPUTERS[ 2 :-3 : 1 ] =  MPUT
COMPUTERS[ 2 :-3 :-1 ] =
COMPUTERS[   :   :-1 ] =  SRETUPMOC
COMPUTERS[-5 :   ]     =  UTERS
COMPUTERS[-5 : 0 :-1 ] =  UPMO
COMPUTERS[-5 :   :-1 ] =  UPMOC
COMPUTERS[-1 : 1 :-2 ] =  SEUM
[Finished in 0.9s]
```

When using a negative step, notice that the answer is shifted to the right by 1.

answered Oct 18 '14 at 17:40

mahmoh
597 6 10

The below is the example of index of a string

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

```
str="Name string"
```

slicing example: [start:end:step]

```
str[start:end] # items start through end-1
str[start:]    # items start through the rest of the array
str[:end]      # items from the beginning through end-1
str[:]         # a copy of the whole array
```

Below is the example usage

```
print str[0]=N
print str[0:2]=Na
print str[0:7]=Name st
print str[0:7:2]=Nm t
print str[0:-1:2]=Nm ti
```

answered Jul 28 at 10:12

Prince Dhadwal
16 2

---

protected by Jon Clements ♦ Feb 8 '13 at 9:20

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?