Indexing and Slicing

All Versions

Examples

Basic Slicing

For any iterable (for eg. a string, list, etc), Python allows you to slice and return a substring or sublist of its data.

Format for slicing:

```
iterable_name[start:stop:step]
```

where,

- start is the first index of the slice. Defaults to 0 (the index of the first element)
- stop one past the last index of the slice. Defaults to len(iterable)
- step is the step size (better explained by the examples below)

Examples:

```
a = "abcdef"
a            # "abcdef" 
             # Same as a[:] or a[::] since it uses the defaults for all three indices
a[-1]        # "f"
a[:]         # "abcdef" 
a[::]        # "abcdef" 
a[3:]        # "def" (from index 3, to end(defaults to size of iterable)) 
a[:4]        # "abcd" (from beginning(default 0) to position 4 (excluded)) 
a[2:4]       # "cd" (from position 2, to position 4 (excluded)) 
```

In addition, any of the above can be used with the step size defined:

```
a[::2]       # "ace" (every 2nd element)
a[1:4:2]     # "bd" (from index 1, to index 4 (excluded), every 2nd element)
```

Indices can be negative, in which case they're computed from the end of the sequence

```
a[:-1]     # "abcde" (from index 0 (default), to the second last element (last element - 1))
a[:-2]     # "abcd" (from index 0 (default), to the third last element (last element -2))
a[-1:]     # "f" (from the last element to the end (default len()) 
```

Step sizes can also be negative, in which case slice will iterate through the list in reverse order:

```
a[3:1:-1]   # "dc" (from index 2 to None (default), in reverse order)
```

This construct is useful for reversing an iterable

```
a[::-1]     # "fedcba" (from last element (default len()-1), to first, in reverse order(-1))
```

Notice that for negative steps the default end_index is None (see http://stackoverflow.com/a/12521981)

```
a[5:None:-1] # "fedcba" (this is equivalent to a[::-1])
a[5:0:-1]    # "fedcb" (from the last element (index 5) to second element (index 1)
```

Reversing an object

You can use slices to very easily reverse a str , list , or tuple (or basically any collection object that implements slicing with the step parameter). Here is an example of reversing a string, although this applies equally to the other types listed above:

```
s = 'reverse me!'
s[::-1]    # '!em esrever'
```

Let's quickly look at the syntax. [::-1] means that the slice should be from the beginning until the end of the string (because start and end are omitted) and a step of -1 means that it should move through the string in reverse.

Slice assignment

Another neat feature using slices is slice assignment. Python allows you to assign new slices to replace old slices of a list in a single operation.

This means that if you have a list, you can replace multiple members in a single assignment:

```
lst = [1, 2, 3]
lst[1:3] = [4, 5]
print(lst) # Out: [1, 4, 5]
```

The assignment shouldn't match in size as well, so if you wanted to replace an old slice with a new slice that is different in size, you could:

```
lst = [1, 2, 3, 4, 5]
lst[1:4] = [6]
print(lst) # Out: [1, 6, 5]
```

It's also possible to use the known slicing syntax to do things like replacing the entire list:

```
lst = [1, 2, 3]
lst[:] = [4, 5, 6]
print(lst) # Out: [4, 5, 6]
```

Or just the last two members:

```
lst = [1, 2, 3]
lst[-2:] = [4, 5, 6]
print(lst) # Out: [1, 4, 5, 6]
```

Indexing custom classes: __getitem__, __setitem__ and __delitem__

```
class MultiIndexingList:
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return repr(self.value)

    def __getitem__(self, item):
        if isinstance(item, (int, slice)):
            return self.__class__(self.value[item])
        return [self.value[i] for i in item]

    def __setitem__(self, item, value):
        if isinstance(item, int):
            self.value[item] = value
        elif isinstance(item, slice):
            raise ValueError('Cannot interpret slice with multiindexing')
        else:
            for i in item:
                if isinstance(i, slice):
                    raise ValueError('Cannot interpret slice with multiindexing')
                self.value[i] = value

    def __delitem__(self, item):
        if isinstance(item, int):
            del self.value[item]
        elif isinstance(item, slice):
            del self.value[item]
        else:
            if any(isinstance(elem, slice) for elem in item):
                raise ValueError('Cannot interpret slice with multiindexing')
            item = sorted(item, reverse=True)
            for elem in item:
                del self.value[elem]
```

This allows slicing and indexing for element access:

```
a = MultiIndexingList([1,2,3,4,5,6,7,8])
a
# Out: [1, 2, 3, 4, 5, 6, 7, 8]
a[1,5,2,6,1]
# Out: [2, 6, 3, 7, 2]
a[4, 1, 5:, 2, ::2]
# Out: [5, 2, [6, 7, 8], 3, [1, 3, 5, 7]]
#       4|1-|----50:---|2-|-----::2-----   <-- indicated which element came from which index
```

While setting and deleting elements only allows for *comma seperated* integer indexing (no slicing):

```
a[4] = 1000
```

```
a
# Out: [1, 2, 3, 4, 1000, 6, 7, 8]
a[2,6,1] = 100
a
# Out: [1, 100, 100, 4, 1000, 6, 100, 8]
del a[5]
a
# Out: [1, 100, 100, 4, 1000, 100, 8]
del a[4,2,5]
a
# Out: [1, 100, 4, 8]
```

Making a shallow copy of an array

A quick way to make a copy of an array (as opposed to assigning a variable with another reference to the original array) is:

```
arr[:]
```

Let's examine the syntax. `[:]` means that `start` , `end` , and `slice` are all omitted. They default to `0` , `len(arr)` , and `1` , respectively, meaning that subarray that we are requesting will have all of the elements of `arr` from the beginning until the very end.

In practice, this looks something like:

```
arr = ['a', 'b', 'c']
copy = arr[:]
arr.append('d')
print(arr)    # ['a', 'b', 'c', 'd']
print(copy)   # ['a', 'b', 'c']
```

As you can see, `arr.append('d')` added `d` to `arr` , but `copy` remained unchanged!

Note that this makes a *shallow* copy, and is identical to `arr.copy()` .

Basic Indexing

Python lists are 0-based *i.e.* the first element in the list can be accessed by the index `0`

```
arr = ['a', 'b', 'c', 'd']
print(arr[0])
>> 'a'
```

You can access the second element in the list by index `1` , third element by index `2` and so on:

```
print(arr[1])
>> 'b'
print(arr[2])
>> 'c'
```

You can also use negative indices to access elements from the end of the list. eg. index `-1` will give you the last element of the list and index `-2` will give you the second-to-last element of the list:

```
print(arr[-1])
>> 'd'
print(arr[-2])
>> 'c'
```

If you try to access an index which is not present in the list, an `IndexError` will be raised:

```
print arr[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Slice objects

Slices are objects in themselves and can be stored in variables with the built-in `slice()` function. Slice variables can be used to make your code more readable and to promote reuse.

```
>>> programmer_1 = [ 1956, 'Guido', 'van Rossum', 'Python', 'Netherlands']
>>> programmer_2 = [ 1815, 'Ada', 'Lovelace', 'Analytical Engine', 'England']
>>> name_columns = slice(1, 3)
>>> programmer_1[name_columns]
['Guido', 'van Rossum']
>>> programmer_2[name_columns]
['Ada', 'Lovelace']
```

Syntax

```
obj[start:stop:step]
```

```
slice(stop)
```

```
slice(start, stop[, step])
```

Parameters

Paramer	Description
`obj`	The object that you want to extract a "sub-object" from
`start`	The index of `obj` that you want the sub-object to start from (keep in mind that Python is zero-indexed, meaning that the first item of `obj` has an index of `0`). If omitted, defaults to `0` .
`stop`	The (non-inclusive) index of `obj` that you want the sub-object to end at. If omitted, defaults to `len(obj)` .
`step`	Allows you to select only every `step` item. If omitted, defaults to `1` .

Remarks

You can unify the concept of slicing strings with that of slicing other sequences by viewing strings as an immutable collection of characters, with the caveat that a unicode character is represented by a string of length 1.

In mathematical notation you can consider slicing to use a half-open interval of `[start, end)` , that is to say that the start is included but the end is not. The half-open nature of the interval has the advantage that `len(x[:n])` = `n` where `len(x) > = n` , while the interval being closed at the start has the advantage that `x[n:n+1] = [x[n]]` where `x` is a list with `len(x) >= n` , thus keeping consistency between indexing and slicing notation.