SLICING

We've used slicing in this course before, but now it's time to dive deeper into slicing

Slicing relies on indexing → only works with sequence types

Mutable Sequence Types

extract data

assign data

Immutable Sequence Types

extract data

Example

```
l = [1, 2, 3, 4, 5]
```

```
l[0:2] = ('a', 'b', 'c')
```

```
l[0:2] → ['a', 'b', 'c', 3, 4, 5]
```

```
l[0:2] → [1, 2]
```

The Slice Type

Although we usually slice sequences using the more conventional notation:

```
my_list[i:j]
```

slice definitions are actually objects → of type `slice`

```
s = slice(0, 2)
```

`type(s)` → `slice`

`s.start` → `0`

`s.end` → `2`

```
l = [1, 2, 3, 4, 5]
```

`l[s]` → `[1, 2]`

This can be useful because we can name slices and use symbols instead of a literal subsequently

Similar to how you can name ranges in Excel…

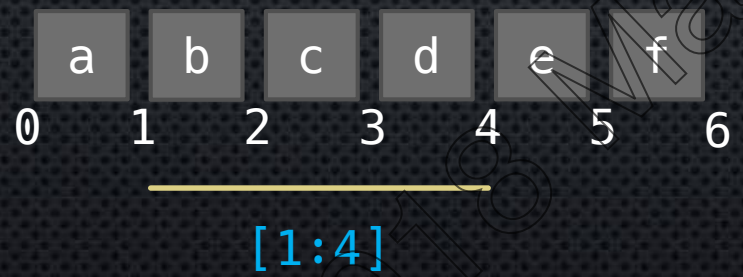
Slice Start and Stop Bounds

`[i:j]` start at `i` (including `i`) stop at `j` (excluding `j`)

all integers `k` where `i <= k < j`

also remember that indexing is zero-based

It can be convenient to think of slice bounds this way



Effective Start and Stop Bounds

Interestingly the following works:

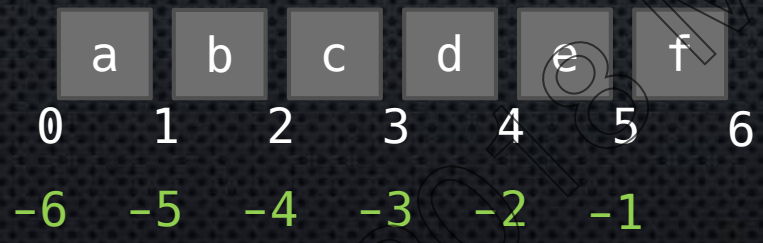
```
l = ['a', 'b', 'c', 'd', 'e', 'f']
l[3:100] → ['d', 'e', 'f']
```

No error!

we can specify slices that are "out of bounds"

In fact, negative indices work too:

```
l[-1]      → 'f'
l[-3: -1]  → ['d', 'e']
```



Step Value

Slices also support a third argument – the step value

`[i:j:k]` (a.k.a stride)

`slice(i, j, k)`

When not specified, the step value defaults to `1`

```
     0    1    2    3    4    5
l = ['a', 'b', 'c', 'd', 'e', 'f']
    -6   -5   -4   -3   -2   -1
```

```
l[0:6:2]     0, 2, 4       → ['a', 'c', 'e']

l[1:6:3]     1, 4          → ['b', 'e']

l[1:15:3]    1, 4          → ['b', 'e']

l[-1:-4:-1]  -1, -2, -3    → ['f', 'e', 'd']
```

Range Equivalence

Any slice essentially defines a sequence of indices that is used to select elements for another sequence

In fact, any indices defined by a slice can also be defined using a range

The difference is that slices are defined independently of the sequence being sliced

The equivalent range is only calculated once the length of the sequence being sliced is known

Example

`[0:100]` `l` sequence of length `10` → `range(0, 10)`

`l` sequence of length `6` → `range(0, 6)`

Transformations `[i:j]`

The effective indices "generated" by a slice are actually dependent on the length of the sequence being sliced

Python does this by reducing the slice using the following rules:

`seq[i:j]`

if `i > len(seq)` → `len(seq)`

if `j > len(seq)` → `len(seq)`

if `i < 0` → `max(0, len(seq) + i)`

if `j < 0` → `max(0, len(seq) + j)`

`i` omitted or `None` → `0`

`j` omitted or `None` → `len(seq)`

```
     0    1    2    3    4    5
l = ['a', 'b', 'c', 'd', 'e', 'f']
    -6   -5   -4   -3   -2   -1
```

length = 6

`[0:100]` → `range(0, 6)`

`[-10:3]` → `range(0, 3)`

`[-5:3]` → `range(1, 3)`

`[:100]` → `range(0, 6)`

`[3:]` → `range(3, 6)`

`[:]` → `range(0, 6)`

Transformations `[i:j:k], k > 0`

With extended slicing things change depending on whether `k` is negative or positive

```
[i:j:k]   = {x = i + n * k | 0 <= n < (j-i)/k}
```

`k > 0` the indices are: `i, i+k, i+2k, i+3k, …, < j` stopping when `j` is reached or exceeded, but never including `j` itself

```
          0    1    2    3    4    5
l = ['a', 'b', 'c', 'd', 'e', 'f']
         -6   -5   -4   -3   -2   -1
```

length = 6

if `i, j > len(seq)` → `len(seq)`

if `i, j < 0` → `max(0, len(seq) + i/j)`

`i` omitted or `None` → `0`

`j` omitted or `None` → `len(seq)`

```
[0:100:2] → range(0, 6, 2)

[-10:100:2] → range(0, 6, 2)
[-5:100:2] → range(1, 6, 2)

[:6:2] → range(0, 6, 2)

[1::2] → range(1, 6, 2)
[::2] → range(0, 6, 2)
```

so same rules as `[i:j]` -- makes sense, since that would be the same as `[i:j:1]`

Transformations [i:j:k], k < 0

[i:j:k] = {x = i + n * k | 0 <= n < (j-i)/k}

k < 0 the indices are: i, i+k, i+2k, i+3k, …, > j

```
    0   1   2   3   4   5
l = ['a', 'b', 'c', 'd', 'e', 'f']
   -6  -5  -4  -3  -2  -1
```

length = 6

if i, j > len(seq) → len(seq) - 1

[5:2:-1] → range(5, 2, -1)

[10:2:-1] → range(5, 2, -1)

if i, j < 0 → max(-1, len(seq) + i/j)

[5:-2:-1] → range(5, 4, -1)

[-2:-5:-1] → range(4, 1, -1)

[-2:-10:-1] → range(4, -1, -1)

i omitted or None → len(seq) - 1

[:-2:-1] → range(5, 4, -1)

j omitted or None → -1

[5::-1] → range(5, -1, -1)

[::-1] → range(5, -1, -1)

Summary

	`[i:j]`	`[i:j:k] k > 0`	`[i:j:k] k < 0`
`i > len(seq)`	`len(seq)`	`len(seq)`	`len(seq)-1`
`j > len(seq)`	`len(seq)`	`len(seq)`	`len(seq)-1`
`i < 0`	`max(0, len(seq)+i)`	`max(0, len(seq)+i)`	`max(-1, len(seq)+i)`
`j < 0`	`max(0, len(seq)+j)`	`max(0, len(seq)+j)`	`max(-1, len(seq)+j)`
`i` omitted / `None`	`0`	`0`	`len(seq)-1`
`j` omitted / `None`	`len(seq)`	`len(seq)`	`-1`

Examples

```
 0    1    2    3    4    5
l = ['a', 'b', 'c', 'd', 'e', 'f']
-6   -5   -4   -3   -2   -1
```

length = 6

```
[-10:10:1]    -10 → 0
              10 → 6
              → range(0, 6)

[10:-10:-1]   10 → 5
              -10 → max(-1, 6-10) → max(-1, -4) → -1
              → range(5, -1, -1)
```

We can of course easily define empty slices!

```
[3:-1:-1]     3 → 3
              -1 → max(-1, 6-1) → 5
              → range(3, 5, -1)
```

Example

`seq` = sequence of length `6`

`seq[::-1]`

`i` is omitted → `len(seq) – 1` → `5`

`j` is omitted → `-1`

→ `range(5, -1, -1)` → `5, 4, 3, 2, 1, 0`

```
seq = 'python'
```

`seq[::-1]` → `'nohtyp'`

If you get confused...

The `slice` object has a method, `indices`, that returns the equivalent range start/stop/step for any slice given the length of the sequence being sliced:

```
slice(start, stop, step).indices(length) → (start, stop, step)
```

the values in this tuple can be used to generate a list of indices using the `range` function

`slice(10, -5, -1)` with a sequence of length `6`

```
i=10 > 6 → 6-1 → 5

j=-5 < 0 → max(-1, 6+-5) → max(-1, 1) → 1      → range(5, 1, -1)
                                               → 5, 4, 3, 2

slice(10, -5, -1).indices(6) → (5, 1, -1)

list(range(*slice(10,-5,-1).indices(6))) → [5, 4, 3, 2]
```

Code Exercises