1 Slicing notation

Using slicing notation with sequences (such as tuples) is very helpful. You can use slicing to retrieve individual items from sequences, or you can get a subsequence from a starting index to an ending index.

1.1 Retrieving individual items

The notation for **individual item retrieval** is `tup[i]`, where `i` is the value of the index. Remember, Python begins indexing at 0, up until len(seq) - 1.

Note: You MUST use `int`s as indices – using something like `float` will raise a `TypeError`.

```
>>> tup = (1, 2, 3, 4, 5)
>>> tup[0]     # indices begin at 0
1
>>> tup[4]     # indices end at len(tup) - 1
5
>>> tup[5]     # improperly large indices raise IndexErrors
IndexError
>>> tup[2]
3
```

Positive indexing gives us the *(i-1)*th element from the beginning. What if we want to get the *i*th element from the end? Solution: use ***negative*** **indices**!

```
>>> tup              # using the same tuple as before
(1, 2, 3, 4, 5)
>>> tup[-1]          # get last element
5
>>> tup[-3]          # get the third element from the end
3
>>> tup[-len(tup)] # get first element
1
>>> tup[-100]        # improperly large index
IndexError
```

1.2 Sub-sequences

You can use **slicing notation** to retrieve sub-sequences from existing tuples. The resulting sub-sequence also a tuple.

```
>>> tup              # using the same tuple as before
(1, 2, 3, 4, 5)
>>> tup[1:3]         # subsequence starts at index 1 up to
(2, 3)               # but not including index 3

>>> tup[0:len(tup)] # a (not so good way) to get the whole
```

```
(1, 2, 3, 4, 5)      # tuple

>>> tup[-4:]         # start: 4th item from the end, to
(2, 3, 4, 5)         # the very end
```

Slicing notation has nice **shorthand notations**:

```
>>> tup[:3]          # ommiting the left-hand index starts
(1, 2, 3)            # from the very beginning

>>> tup[3:]          # ommitting the right-hand index goes
(3, 4, 5)            # to the very end

>>> tup[:]           # duplicates the entire tuple
(1, 2, 3, 4, 5)
```

When slicing, you can use **indexes that exceed the length of the tuple** (although that's not good style). Python is smart enough to correct the mistake.

```
>>> tup[2:100]       # 100 > len(tup)
(3, 4, 5)
>>> tup[3:2]         # if left-index > right-index
()                   # return empty tuple

>>> tup[-100:]       # abs(-100) > len(tup)
(1, 2, 3, 4, 5)
```

You can also specify the **increment step-size** for slicing. The notation is `tup[start:end:step]`

```
>>> tup[1:4:2]       # subsequence from index 1 up to index 4,
(2, 4)               # but only getting every other item

>>> tup[0:4:3]       # subsequence from index 1 up to index 4,
(1, 4)               # but only getting every third item

>>> tup[:4:2]        # subsequence from start up to index 4,
(1, 3)               # but only getting every other item

>>> tup[1::2]        # subsequence from index 1 to end,
(2, 4)               # but only getting every other item

>>> tup[::2]         # subsequence from start to end,
(1, 3, 5)            # but only getting every other item

>>> tup[::-1]        # get the entire tuple but in reverse.
(5, 4, 3, 2, 1)
```

1.3 Splicing other things

Splicing notation also works on `strings` (and other built-in data structures, like `lists`, but you haven't learned about those yet).

```
>>> s = 'This works too!'
>>> s[5]                    # get the character at index 5
'w'
>>> s[5:10]                 # goes up to but not including index 5
'works'
>>> s[:-1]                  # everything up till the last
'this works too'            # character, '!'
```

2 Generator expressions

Python has a short hand for generating large sequences in a single line. The syntax for a generator expression is:

```
<expression involving elem> for <elem> in <sequence> if <boolean>
```

This will return a 'generator object', which you can then convert to a tuple.

```
>>> tuple(i**2 for i in (1, 2, 3, 4, 5) if i % 2 == 0)
(4, 16)        # square each item if the item is odd
>>> f = lambda x: x / 2
>>> tuple(f(elem) for elem in (1, 2, 3, 4))
(0.5, 1.0, 1.5, 2.0)         # the if clause is optional
```

3 map, filter, reduce, and other functions

In lecture, you learned about `map`, `filter`, and `reduce`. Here are some extra details about them.

3.1 map

In lecture, you saw that map can take two arguments: a *function*; and an *iterable* (e.g. a tuple). The function is applied to each item in the iterable.

```
>>> tuple(map(lambda x: 2*x, (1, 2, 3, 4)))  # multiply each item by 2
(2, 4, 6, 8)
>>> tuple(map(lambda x: x**2, (1, 2, 3, 4))) # square each item
(1, 4, 9, 16)
```

There is an extended form of `map`, which takes in three or more arguments.

```
map(function, iterable1, iterable2, ...)
```

`function` will be applied in parallel to all the iterables (all the first items, then all the second, then all the third). The `function` must take as many arguments as there are iterables. For example, if there are three iterables, `function` must take three arguments.

```
>>> tuple(map(lambda x, y: x + y, (1, 2, 3, 4), (4, 3, 2, 1)))
(5, 5, 5, 5)          # (1+4, 2+3, 3+2, 4+1)
>>> tuple(map(lambda x, y, z: x + y + z, (1, 2), (3, 4), (5, 6)))
(9, 12)               # (1+3+5, 2+4+6)
```

Note: If the iterables are not of the same length, map will only go until the shortest iterable runs out.

3.2 filter

Filter just returns a new sequence, whose items are items in the original sequence that passed the filter function. The format of a filter call is

```
filter(pred, iterable)
```

Here are some examples:

```
>>> tuple(filter(lambda x: x %2, (1, 2, 3, 4, 5)))
(2, 4)                    # keeps even numbers

>>> tuple(filter(lambda s: len(s)>3, ('hi', 'hello', 'fooply')))
('hello', 'fooply')     # keeps words whose lengths exceed 3
```

3.3 reduce

`reduce` takes a sequence and uses a function to "combine" all the items in the sequence. The result is usually an int. The format is

```
reduce(function, iterable)
```

`function` must take in two arguments. Here are some examples

```
>>> reduce(add, (1, 2, 3, 4, 5))    # sums numbers 1 through 5
15
>>> reduce(lambda a, b: a*b, (1, 2, 3)) # multiplies numbers 1 - 3
6
```

`reduce` can also take an optional 3rd argument, a `starting point`.

```
>>> reduce(add, (1, 2, 3, 4, 5), 10)
25              # adds numbers 1 through 5 to 10
```

3.4 enumerate

The `enumerate` function can be applied in a `for` loop to get the index of an item along with the item itself.

```
>>> tup = ('a', 'b', 'c')
>>> for i, item in enumerate(tup)
...     print(i, item)
0 a
1 b
2 c
```

This is useful when you want both the index and the item in a loop (no need to initialize an index variable before the loop).

3.5 zip

The `zip` function takes a number of iterables, and pairs together the items in parallel. For example, it pairs together the first items, then the second, then the third items. `zip` returns a 'zip object', which you can convert into a tuple.

```
>>> a = (1, 2, 3, 4)
>>> b = (5, 6, 7, 8)
>>> tuple(zip(a, b))
((1, 5), (2, 6), (3, 7), (4, 8))
```