Sort Python lists using key in Python:

The easiest way to sort is with the sorted(list) function, which takes a list and returns a new list with those elements in sorted order. The original list is not changed.

```
a = [5, 1, 4, 3]
print(sorted(a))  ## [1, 3, 4, 5]
print(a)  ## [5, 1, 4, 3]
```

It's most common to pass a list into the sorted() function, but in fact it can take as input any sort of iterable collection. The older list.sort() method is an alternative detailed below. The sorted() function seems easier to use compared to sort(), so I recommend using sorted().

The sorted() function can be customized through optional arguments. The sorted() optional argument reverse=True, e.g. sorted(list, reverse=True), makes it sort backwards.

```
strs = ['aa', 'BB', 'zz', 'CC']
print(sorted(strs))  ## ['BB', 'CC', 'aa', 'zz'] (case sensitive)
print(sorted(strs, reverse=True))   ## ['zz', 'aa', 'CC', 'BB']
```

Custom Sorting With key=

For more complex custom sorting, sorted() takes an optional "key=" specifying a "key" function that transforms each element before comparison. The key function takes in 1 value and returns 1 value, and the returned "proxy" value is used for the comparisons within the sort.

For example with a list of strings, specifying key=len (the built in len() function) sorts the strings by length, from shortest to longest. The sort calls len() for each string to get the list of proxy length values, and then sorts with those proxy values.

```
strs = ['ccc', 'aaaa', 'd', 'bb']
print(sorted(strs, key=len))  ## ['d', 'bb', 'ccc', 'aaaa']
```
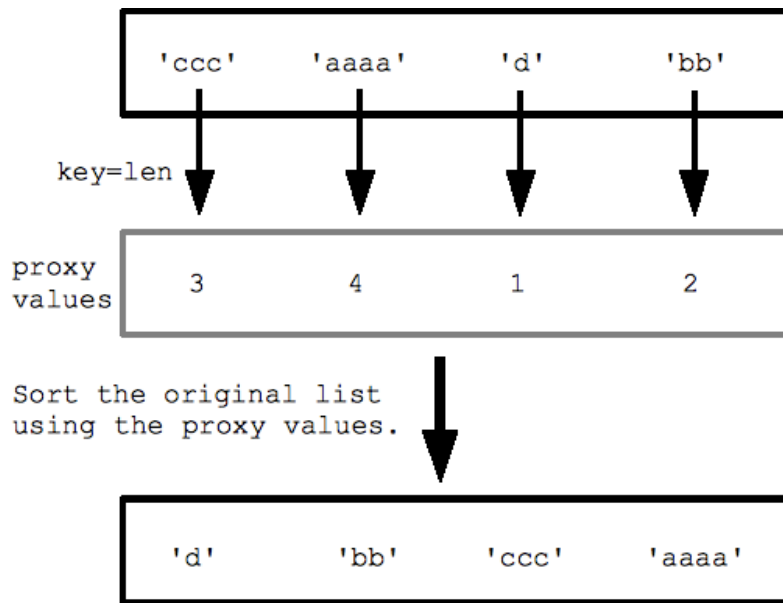
As another example, specifying "str.lower" as the key function is a way to force the sorting to treat uppercase and lowercase the same:

```
## "key" argument specifying str.lower function to use for sorting
print(sorted(strs, key=str.lower))  ## ['aa', 'BB', 'CC', 'zz']
```

You can also pass in your own MyFn as the key function, like this:

```
## Say we have a list of strings we want to sort by the last letter of the string.
strs = ['xc', 'zb', 'yd' ,'wa']

## Write a little function that takes a string, and returns its last letter.
## This will be the key function (takes in 1 value, returns 1 value).
def MyFn(s):
  return s[-1]

## Now pass key=MyFn to sorted() to sort by the last letter:
print(sorted(strs, key=MyFn))  ## ['wa', 'zb', 'xc', 'yd']
```

For more complex sorting like sorting by last name then by first name, you can use the itemgetter or attrgetter functions like:

```
from operator import itemgetter

# (first name, last name, score) tuples
grade = [('Freddy', 'Frank', 3), ('Anil', 'Frank', 100), ('Anil', 'Wang', 24)]
sorted(grade, key=itemgetter(1,0))
# [('Anil', 'Frank', 100), ('Freddy', 'Frank', 3), ('Anil', 'Wang', 24)]

sorted(grade, key=itemgetter(0,-1))
#[('Anil', 'Wang', 24), ('Anil', 'Frank', 100), ('Freddy', 'Frank', 3)]
```

sort() method

As an alternative to sorted(), the sort() method on a list sorts that list into ascending order, e.g. list.sort(). The sort() method changes the underlying list and returns None, so use it like this:

```
alist.sort()          ## correct
alist = blist.sort()    ## Incorrect. sort() returns None
```

The above is a very common misunderstanding with sort() -- it *does not return* the sorted list. The sort() method must be called on a list; it does not work on any enumerable collection (but the sorted() function above works on anything). The sort() method predates the sorted() function, so you will likely see it in older code. The sort() method does not need to create a new list, so it can be a little faster in the case that the elements to sort are already in a list.

Tuples

A tuple is a fixed size grouping of elements, such as an (x, y) co-ordinate. Tuples are like lists, except they are immutable and do not change size (tuples are not strictly immutable since one of the contained elements could be mutable). Tuples play a sort of "struct" role in Python -- a convenient way to pass around a little logical, fixed size bundle of values. A function that needs to return multiple values can just return a tuple of the values. For example, if I wanted to have a list of 3-d coordinates, the natural python representation would be a list of tuples, where each tuple is size 3 holding one (x, y, z) group.

To create a tuple, just list the values within parenthesis separated by commas. The "empty" tuple is just an empty pair of parenthesis. Accessing the elements in a tuple is just like a list -- len(), [ ], for, in, etc. all work the same.

```
tuple = (1, 2, 'hi')
print(len(tuple))  ## 3
print(tuple[2])    ## hi
tuple[2] = 'bye'  ## NO, tuples cannot be changed
tuple = (1, 2, 'bye')  ## this works
```

To create a size-1 tuple, the lone element must be followed by a comma.

```
tuple = ('hi',)   ## size-1 tuple
```

It's a funny case in the syntax, but the comma is necessary to distinguish the tuple from the ordinary case of putting an expression in parentheses. In some cases you can omit the parenthesis and Python will see from the commas that you intend a tuple.

Assigning a tuple to an identically sized tuple of variable names assigns all the corresponding values. If the tuples are not the same size, it throws an error. This feature works for lists too.

```
(x, y, z) = (42, 13, "hike")
print(z)  ## hike
(err_string, err_code) = Foo()  ## Foo() returns a length-2 tuple
```

List Comprehensions (optional)

List comprehensions are a more advanced feature which is nice for some cases but is not needed for the exercises and is not something you need to learn at first (i.e. you can skip this section). A list comprehension is a compact way to write an expression that expands to a whole list. Suppose we have a list nums [1, 2, 3, 4], here is the list comprehension to compute a list of their squares [1, 4, 9, 16]:

```
nums = [1, 2, 3, 4]

squares = [ n * n for n in nums ]   ## [1, 4, 9, 16]
```

The syntax is [*expr* for var in list] -- the for var in list looks like a regular for-loop, but without the colon (:). The *expr* to its left is evaluated once for each element to give the values for the new list. Here is an example with strings, where each string is changed to upper case with '!!!' appended:

```
strs = ['hello', 'and', 'goodbye']

shouting = [ s.upper() + '!!!' for s in strs ]
## ['HELLO!!!', 'AND!!!', 'GOODBYE!!!']
```

You can add an if test to the right of the for-loop to narrow the result. The if test is evaluated for each element, including only the elements where the test is true.

```
## Select values <= 2
nums = [2, 8, 1, 6]
small = [ n for n in nums if n <= 2 ]  ## [2, 1]

## Select fruits containing 'a', change to upper case
fruits = ['apple', 'cherry', 'banana', 'lemon']
afruits = [ s.upper() for s in fruits if 'a' in s ]
## ['APPLE', 'BANANA']
```