Complex Sorting

Python 3

home (../handouts.html)

Introduction: Complex Sorting

We often sort a sequence of values by some other criteria.

Some values simply represent themselves - for example, a list of floats or a set of strings. Most other data that we work with, however, represent more than just the value itself.

For example if we are working with a list of filenames, we might want to consider the files by their names, or by their sizes (which is the biggest?), or by their last modification times (which is the latest?).

Or if we are working with a set of corporation names, we might be interested in the market cap, the revenue from last year, the change in stock price over the last quarter, etc. All of these are interesting attributes of the company, and we are often interested in ranking them according to one of these criteria (which company has the greatest market cap? which had the greatest revenue?)

So if we are sorting a list of items in a container, there are often multiple criteria by which we might want to sort.

The default sort we have begun to see with **sorted()** -- i.e., to sort strings by their alphabetic value or numbers by their numeric value -- must be rerouted so it uses *alternate criteria*.

This unit is about the mechanism by which we can accomplish this alternate sorting.

Objectives for the Unit: Complex Sorting

- Define custom **functions**: named blocks of Python code.
- Review sequence sorting using **sorted()**
- Review dictionary sorting (**sorted()** returns a list of keys)
- Use an *item criteria function* to indicate how items should be sorted
- Use built-in functions to indicate how items should be sorted

Summary Statement: Functions (user-defined)

A *user-defined function* is simply a named code block that can be called and executed any number of times.

```
def print_hello():
    print("Hello, World!")

print_hello()             # prints 'Hello, World!'
print_hello()             # prints 'Hello, World!'
print_hello()             # prints 'Hello, World!'
```

Function argument(s)

A function's *arguments* are renamed in the function definition, and the function refers to them by these names.

```
def print_hello(greeting, person):
  full_greeting = greeting + ", " + person + "!"
  print(full_greeting)

print_hello('Hello', 'World')         # prints 'Hello, World!'
print_hello('Bonjour', 'Python')      # prints 'Bonjour, Python!'
print_hello('squawk', 'parrot')       # prints 'squawk, parrot!'
```

(The argument objects are *copied* to the argument names -- they are the same objects.)

Function return value

A function's *return value* is passed back from the function with the **return** statement.

```
def print_hello(greeting, person):
  full_greeting = greeting + ", " + person + "!"
  return full_greeting

msg = print_hello('Bonjour', 'parrot')
print(msg)                              # 'Bonjour, parrot!'
```

Summary Function (Review): sorted()

sorted() takes a sequence argument and returns a sorted list. The sequence items are sorted according to their respective types.

sorted() with numbers

```
mylist = [4, 3, 9, 1, 2, 5, 8, 6, 7]

sorted_list = sorted(mylist)
print(sorted_list)                # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

sorted() with strings

```
namelist = ['jo', 'pete', 'michael', 'zeb', 'avram']

print(sorted(namelist))           # ['avram', 'jo', 'michael', 'pete', 'zeb']
```

Summary Task (Review): sorting a dictionary's keys

Sorting a **dict** means sorting the *keys* -- **sorted()** returns a list of sorted keys.

```
bowling_scores = {'jeb': 123, 'zeb': 98, 'mike': 202, 'janice': 184}

sorted_keys = sorted(bowling_scores)

print(sorted_keys)        # ['janice', 'jeb', 'mike', 'zeb']
```

Indeed, any "listy" sort of operation on a **dict** assumes the keys: **for** looping, subscripting, **sorted()**; even **sum()**, **max()** and **min()**.

Summary Task (Review): sorting a dictionary's keys by its values

The dict **get()** method returns a value based on a key -- perfect for sorting keys by values.

```
bowling_scores = {'jeb': 123, 'zeb': 98, 'mike': 202, 'janice': 184}

sorted_keys = sorted(bowling_scores, key=bowling_scores.get)

print(sorted_keys)        # ['zeb', 'jeb', 'janice', 'mike']


for player in sorted_keys:
        print("{} scored {}".format(player, bowling_scores[player]))

        ##  zeb scored 98
        ##  jeb scored 123
        ##  janice scored 184
        ##  mike scored 202
```

Summary Feature: Custom sort using an *item criteria* function

An **item criteria** function returns to python the *value by which* a given element should be sorted.

Here is the same dict sorted by value in the same way as previously, through a custom *item criteria* function.

```
def by_value(dict_key):                       # a key to be sorted (for example, 'mike'
    dict_value = bowling_scores[dict_key]     # retrieving the value based on 'mike':  202
    return dict_value                         # returning the value 202

bowling_scores = {'jeb': 123, 'zeb': 98, 'mike': 202, 'janice': 184}
sorted_keys = sorted(bowling_scores, key=by_value)

print(sorted_keys)          # ['zeb', 'jeb', 'janice', 'mike']
```

The dict's keys are sorted by value because of the **by_value()** function:

1. **sorted()** sees **by_value** referenced in the function call.
2. **sorted()** calls the **by_value()** *four times*: once with each key in the dict.
3. **by_value()** is called with **'jeb'** (which returns **123**), **'zeb'** (which returns **98**), **'mike'** (which returns **202**), and **'janice'** (which returns **184**).
4. The *return value* of the function is the *value by which* the key will be sorted

Therefore because of the return value of the function, **jeb** will be sorted by **123**, **zeb** by **98**, etc.

Summary Task: sort a numeric string by its numeric value

Numeric strings (as we might receive from a file) sort alphabetically:

```
numbers_from_file = ['1', '10', '3', '20', '110', '1000' ]
sorted_numbers = sorted(numbers_from_file)

print(sorted_numbers)    # ['1', '1000', '110', '20', '3'] (alphabetic sort)
```

To sort numerically, the item criteria function can convert to **int** or **float**.

```
def by_numeric_value(this_string):
    return int(this_string)

numbers_from_file = ['1', '10', '3', '20', '110', '1000' ]
sorted_numbers = sorted(numbers_from_file, key=by_numeric_value)

print(sorted_numbers)    # ['1', '3', '10', '20', '110', '1000']
```

Note that the values returned do not change; they are simply sorted by their integer equivalent.

Summary Task: sort a string by its case-insensitive value

Python string sorting sorts uppercase before lowercase:

```
namelist = ['Jo', 'pete', 'Michael', 'Zeb', 'avram']
print(sorted(namelist))            # ['Jo', 'Michael', 'Zeb', 'avram', 'pete']
```

To sort "insensitively", the item criteria function can lowercase each string.

```
def by_lowercase(my_string):
    return my_string.lower()

namelist = ['Jo', 'pete', 'michael', 'Zeb', 'avram']
print(sorted(namelist, key=by_lowercase))              # ['avram', 'Jo', 'michael', 'pete', 'Zeb']
```

Summary Task: sort a string by a portion of the string

To sort a string by a portion of the string (for example, the last name in these 2-word names), we can split or slice the string and return the portion.

```
full_names = ['Jeff Wilson', 'Abe Zimmerman', 'Zoe Apple', 'Will Jefferson']

def by_last_name(fullname):
    fname, lname = fullname.split()
    return lname

sfn = sorted(full_names, key=by_last_name)

print(sfn)                                       #  ['Zoe Apple',
                                                #   'Will Jefferson',
                                                #   'Jeff Wilson',
                                                #   'Abe Zimmerman']
```

Summary Task: sort a file line by a field within the line

To sort a string of fields (for example, a CSV line) by a field within the line, we can **split()** and return a field from the split.

```
def by_third_field(this_line):
    els = this_line.split(',')
    return els[2]

lines = open('../python_data/students.txt')
sorted_lines = sorted(lines, key=by_third_field)
print(sorted_lines)


        # [ 'pk669,Pete,Krank,Darkling,NJ,8044894893\n',
        #   'ms15,Mary,Smith,Wilsontown,NY,5185853892\n',
        #   'jw234,Joe,Wilson,Smithtown,NJ,2015585894\n'    ]
```

Summary Task: custom sort using built-in functions

Built-in functions can be used to indicate item sort criteria in the same way as custom functions -- by telling Python to pass an element and sort by the return value.

len() returns string length - so it can be used to sort strings by length

```
mystrs = ['angie', 'zachary', 'zeb', 'annabelle']

print(sorted(mystrs, key=len))       # ['zeb', 'angie', 'zachary', 'annabelle']
```

Using a builtin function

os.path.getsize() returns the byte size of any file based on its name (in this example, in the *present working directory*):

```
import os

print(os.path.getsize('test.txt'))    # return 53, the byte size of test.txt
```

To sort files by their sizes, we can simply pass this function to **sorted()**

```
import os

files = ['test.txt', 'myfile.txt', 'data.csv', 'bigfile.xlsx']   # some files in my current dir

size_files = sorted(files, key=os.path.getsize)                  # pass each file to getsize()

for this_file in size_files:
        print("{}:  {} bytes".format(this_file, os.path.getsize(this_file)))
```

(Please note that this will only work if your terminal's *present working directory* is the same as the files being sorted. Otherwise, you would have to prepend the path -- see **File I/O**, later in this course.)

Using methods

```
namelist = ['Jo', 'pete', 'michael', 'Zeb', 'avram']
print(sorted(namelist, key=str.lower))                 # ['avram', 'Jo', 'michael', 'pete', 'Zeb']
```

Using methods called on existing objects

```
companydict = {'IBM': 18.68, 'Apple': 50.56, 'Google': 21.3}

revc = sorted(companydict, key=companydict.get)       # ['IBM', 'Google', 'Apple']
```

You can use a method here in the same way you would use a function, except that you won't be specifying the specific object as you would normally with a method. To refer to a method "in the abstract", you can say str.upper or str.lower. However, make sure not to actually call the method (which is done with the parentheses). Instead, you simply refer to the method, i.e., mention the method without using the parentheses.)

Multi-target Assignment

This convenience allows us to assign values in a list to individual variable names.

```
line = 'Acme:23.9:NY'

items = line.split(':')
print(items)               # ['Acme', '23.9', 'NY']

name, revenue, state = items

print(revenue)           # 23.9
```

Sidebar: cascading sort

Sort a list by multiple criteria by having your sort function return a 2-element tuple.

```
def by_last_first(name):
  fname, lname = name.split()
  return (lname, fname)

names = ['Zeb Will', 'Deb Will', 'Joe Max', 'Ada Max']

lnamesorted = sorted(names, key=by_last_first)         # ['Ada Max', 'Joe Max', 'Deb Will', 'Zeb Will']
```