New York University
School of Continuing and Professional Studies
Division of Programs in Information Technology

Introduction to Python
Exercise Solutions, Session 4

Ex. 4.1 Open the file passwords.txt. Then, using the open filehandle, without looping, print out a list of all the lines in the file. (Hint: use the file readlines() method).

```
myfile = '../python_data/passwords.txt'

fh = open(myfile)                  # open file, return file object
lines = fh.readlines()             # readlines() returns list of strings
print(lines)                        # each string is a line from file
```

This exercise simply illustrates the return value of the file method readlines(), which returns a list of strings -- all the lines in the file.

Ex. 4.2 Extending the above program, again without looping, print out a list of just the 1st three lines of the file. (Hint: use a slice of the list returned from readlines()).

```
myfile = '../python_data/passwords.txt'

fh = open(myfile)            # open file, return file object
x = fh.readlines()           # readlines() returns list of strings

newslice = x[0:3]            # a list with the 0 through 2 element (1st 3)
print(newslice)
```

The purpose here is to show that the lines of the file can be sliced -- we can therefore select the lines we wish to work with using a list slice, and not by having to go to more elaborate means (e.g. looping through the file and counting the lines to decide which ones we want to work with).

Ex. 4.3 Create an empty list. Opening and looping through the file student_db.txt, split each line into elements, and append just the state values (the 4th value in each row) to a list. Print the list object. (Hint: make sure the list is initialized before the loop begins and is printed after the loop ends. If either statement is found inside the loop, it will be executed multiple times.

```
sfile = '../python_data/student_db.txt'

states = []                       # initialize an empty list

fh = open(sfile)                  # open file, return file object

for line in fh:                   # for each line in file
    elements = line.split(':')    # returns a list of string elements
    states.append(elements[3])    # append the 4th field from split

print(states)                      # print compiled list
```

This is the basic summary algorithm, in which a "summary variable" (in this case a list) is initialized before the loop begins, is modified inside the loop, and then its value(s) reported once the loop is complete. The point of the exercise is to build up a summary list in the proper manner, taking care not to initialize the list inside the loop nor report its value inside the loop. The variable lives outside the loop, and is modified only inside the loop, as values are added to it one at a time.

Ex. 4.4 Extend the above program by omitting the first line from the file. The proper way to do this is to call readlines() on the file handle, assign this to a list variable, take a slice of the resulting list of lines omitting the first element (lines[1:]), and then loop through this list.

```
sfile = '../python_data/student_db.txt'

states = []

fh = open(sfile)                  # open file, return file object
lines = fh.readlines()            # readlines() -> list of strings
data_lines = lines[1:]            # list of 2nd through last lines of file

for line in data_lines:           # for each string line in list
    elements = line.split(':')    # split line on comma, returns
                                  # list of strings:  each field
    states.append(elements[3])    # add the 4th field from split

print(states)                      # print compiled list
```

The exercise instructions attempt to guide you to thinking of a file as a list that can be sliced, then combining this technique of slicing a file with the summary algorithm, which is looping through and splitting each line from the slice, then adding each split value to a summary list. It simply combines the file slicing with the summary algorithm.

Ex. 4.5 Starting with an empty list, and then opening and looping through student_db.txt, append the 1st field (a student ID) to a list only if the state field is NY. Print the list object.

```
sfile = '../python_data/student_db.txt'

ids = []

fh = open(sfile)                 # open file, return file object

for line in fh:                  # for each string line in file
    elements = line.split(':')   # split line on comma, returns
                                 # list of strings: each field
    if elements[3] == 'NY':      # if the state (4th field) is 'NY'
        ids.append(elements[0])  # add first field from split (id) to list

print(ids)                        # when loop is done, print compilled list
```

This combines the prior summary algorithm solution with line selection based on a particular field, i.e. only those lines containing the correct field value will be included in the collected values.

Ex. 4.6 Opening and reading through revenue.txt, build a list from the 3rd field (the floating point value), then sum up the values using sum(). (Hint: sum() will not work unless you convert each element to a float value before you add it to the list, so do that with float().)

```
rfile = '../python_data/revenue.txt'

fh = open(rfile)                 # open file, return file object

revenues = []                    # initialize an empty list

for line in fh:                  # for each string line in file
    fields = line.split(',')     # split line on comma, returns
                                 # list of strings: each field
    fval = float(fields[2])
    revenues.append(fval)

print(sum(revenues))
```

The point of this exercise has less to do with the summary algorithm (which we are reprising and I hope you are recognizing) and more to do with the sum() function, which will sum up a container of floats. Of course if you neglect to convert each of the collected values to a float, the call to sum() will fail when the function attempts to add the first value to numeric 0.

Ex. 4.7 Extending the above program, use len() to determine the length of the list, then use the formula sum(flist) / len(flist) to produce an average (where flist is the list of floats you compiled).

```
rfile = '../python_data/revenue.txt'

fh = open(rfile)                 # open file, return file object

revenues = []                    # initialize an empty list

for line in fh:                  # for each string line in file
    fields = line.split(',')     # split line on comma, returns
                                 # list of strs, each field in string
    fval = float(fields[2])      # the third field converted to a float
    revenues.append(fval)        # append the float to a list

print(sum(revenues) / len(revenues))  # average valude in the list
                                   # total sum divided by count
```

This extends the concept of working with a list of values by asking you to take the average value from a list -- by computing the values of len() (giving a count of values in the list) and sum() (giving a sum of the same values). The purpose is to show how these summary functions work as well as to show how easy it is to get information from a list of values.

Ex. 4.8 Given the following code:

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

Determine and print the median (middle) value without mentioning any values directly. (Hint: use len() to determine the length, then use that value to figure out the "halfway" index. Also remember that your calculated index must be an integer, not a float.)

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

median_index = int(len(values) / 2)   # of elements / 2 == middle index

print((values[median_index]))      # using the 'middle index' to get
                                 # the middle value
```

The "middle" value of an odd number of values can be found by using the "middle index". If there are 5 values, the middle index is 2. If there are 7 values, the middle index is 3. Our division of an odd count of values (5, 7, 9, etc.) by 2 must be converted to an integer, i.e. with any floating point values lopped off -- int() will do this for us.

Ex. 4.9 Given the following code:

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

Again, determine and print the median value. This time you'll need to use the sorted() function, which takes an unsorted list and returns a sorted list.

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

svals = sorted(values)            # numeric values sorted from low - high:
                                  # [1, 2, 3, 4, 5]

median_index = int(len(values) / 2)  # calculating len of list / 2
print(svals[median_index])           # using the middle index to get the
                                     # middle value
```

This exercise builds on the previous assignment's, requiring that the values be sorted before the median calculation.

Ex. 4.10 Given the following code:

```
values = [6, 1, 3, 2, 4, 5]
```

Determine the median value, which (in an even-number of elements) is halfway between the two "middle" values in a sorted list. (Hint: again use the len() of the list to calculate the indices of the middle two values.)

```
values = [6, 1, 3, 2, 4, 5]

svals = sorted(values)             # sorted() of numbers returns
                                   # numeric sort - put them in order

len_values_mid = int(len(svals)/2) # the 'middle' index

lval = svals[len_values_mid-1]     # value to left of 'middle' index
rval = svals[len_values_mid]       # value for the 'middle' index

median = (lval + rval) / 2        # these two values summed and
                                  # the value between them
print(median)
```

In an even-number of elements list, the same approach using an subscript index calculated from the len() of the list can be used to identify one of the two "middle" values needed to calculate the median -- the other is the value to the left of it, i.e. with an index one less than the other middle value. We then need only sum those two middle values and divide by 2 or 2.0 to arrive at the median value.

Ex. 4.11 Given a list containing duplicate values, initialize an empty set() and add each element to it to produce a set of unique values. Print the whole set. (Note: just for practice, don't pass the list directly to the set() constructor -- loop through the list and add each element one at a time.)

```
dupvals = [1, 3, 1, 1, 2, 3, 2, 1, 3]

uniquevals = set()                # initialize a new empty set

for val in dupvals:               # for each value in the list
    uniquevals.add(val)           # add the value to the set
                                  # (dupe value are ignored)

print(uniquevals)                  # print entire set
```

A set automatically discards any duplicate values -- so you don't need to check for duplicates before adding to the set. As alluded to in the description (and as you'll see in the slides), you can pass the list to the set() constructor and a new set will be returned, with all duplicates removed. However the instructions are asking you to go "the long way around" and loop through the list, adding each element from the list to the set. This is requested because in the homework we'll be looping through a file and adding each element to a set.

Ex. 4.12 Opening the revenue.txt text file, loop through and print each line of the file, but make sure there are no blank lines printed (Hint: use rstrip() on each line to remove the newline character; print itself already prints a newline at the end of every line).

```
filename = '../python_data/revenue.txt'

fh = open(filename)              # open file, return file object

for line in fh:                  # for each string line in file

    line = line.rstrip()         # 'remove' whitespace from end of string
                                 # returning a new string without it
    print(line)
```

This exercise is simply reminding you to strip each line as you loop through a file. (Stripping is not strictly necessary if you're not using the end of the string (i.e., if you're planning to split and use a "middle" value from the split, but it is good practice to dispense with the newline anytime you are looping through a file - it just makes things easier. Another way to do this is to use read() on the file to produce a string, and splitlines() to produce a list of strings -- that is, the file string split on the newlines.

Ex. 4.13 Given the following list:

```
mylist = [-5, -2, 1, -3, 1.5, 7, 9]
```

Loop through mylist and build a new list that is only positive numbers (i.e., greater than 0). Print the list.

```
mylist = [-5, -2, 1, -3, 1.5, 7, 9]

newlist = []                    # initialize an empty list

for el in mylist:               # looping through each element in list
    if el > 0:                  # if the element is greater than 0:
        newlist.append(el)      # add the element to the new list

print(newlist)
```

This is a "filtering" loop, one which works with a populated list and has the effect of filtering out selected elements. The general method for such a loop is to create an empty list first, then loop through the populated list and test each element against the filtering condition (in this case, is it greater than 0?). Elements that pass the test are appended to the empty list. Thus, the original/source list is not changed -- it is usually much more straightforward to build a new list rather than try to modify an old one, particularly since modifying a list while you are looping through it can cause problems.

Ex. 4.14 Given the following code:

```
sentence = 'I could; I wouldn't.  I might?  Might I!'
```

Split this sentence into words (without altering or processing them). Print each word on a line.

```
sentence = "I could; I wouldn't.  I might?  Might I!"

words = sentence.split()  # split line on space, returns
                          # list of strs, each word in string

for word in words:        # loop through each word in list
    print(word)            # print the word
```

This is the first step in breaking up a file of words into individual words. split() with no arguments splits on whitespace (i.e., any consecutive spaces found will be removed and will mark the border between two elements). Of course split() returns a list of strings; we can loop over this list with for.

Ex. 4.15 Extending the above code, use a single call to rstrip() to remove any punctuation. (Hint: you can place any punctuation characters to be removed together in a single string argument to rstrip(); any single character within the string will be removed.)

```
sentence = "I could; I wouldn't.  I might?  Might I!"

words = sentence.split()        # split line on space, returns
                                # list of strs, each word in string

for word in words:              # for each word in list of words
    word = word.rstrip(';.?!')  # remove any punctuation from end
    print(word)                  # print stripped word
```

This exercise takes the previous one further by processing each word in the split list, stripping off any punctuation. Note that the punctuation list ';.?!' is actually a single string, and yet each character in the argument string is treated separately as a candidate for stripping (in other words, this string argument is treated as a list of characters). Also remember that string methods don't actually modify the string, which is immutable -- instead, rstrip() is returning a new string which we are assigning back to the same name -- the variable "word".

Ex. 4.16 Extending the above code, add an additional statement to lowercase each word.

```
sentence = "I could; I wouldn't.  I might?  Might I!"

words = sentence.split()         # split line on space, returns
                                 # list of strs: each word in line

for word in words:               # for each word in the list of strs
    word = word.rstrip(';.?!')   # remove any punctuation from end
    word = word.lower()          # lowercase this string
    print(word)
```

Next step -- process each word further by lowercasing it. Again, we're assigning the new string returned from lower() to the same name, effectively replacing it -- this has the effect of making it seem as if the word string is being modified -- instead, it's only renaming the new string to the old name, word.

Ex. 4.17 Extending the above code, add each word to a set(), then print the entire set.

```
sentence = "I could; I wouldn't.  I might?  Might I!"

newset = set()                   # new empty set
words = sentence.split()         # split line on space, returning
                                 # a list of strs, each word

for word in words:               # for each word in the list of strs
    word = word.rstrip(';.?!')   # remove punctuation from end of word
    word = word.lower()          # lowercase the word
    newset.add(word)             # add the word to the set

print(newset)                     # print the set
```

Now we're combining the set adding earlier with the word preparing that we just accomplished.

Ex. 4.18 Open the file pyku.txt. Print each word from the file on a separate line. (Hint: use read() on the filehandle to return a string, then use split() on the string to create a master list of words.)

```
filename = '../python_data/pyku.txt'

fh = open(filename)                  # open file, return file object

text = fh.read()                     # return file text as a string
words = text.split()                 # split space on space, returns
                                     # list of strs: each word in file

for word in words:                   # for each word in split words
    print(word)                       # print the word
```

This exercise shows how easy it is to split an entire file into a list of words. One needs only to retrieve the entire file as a string, and then split on whitespace. This illustrates another way we can "slice and dice" the file, since a file can be read as a list of strings, a single string, or in this case a list of words.

Ex. 4.19 Extending the previous solution, re-introduce the stripping and lowercasing to prepare each word for comparison. Add each prepared word to a set. Print the set.

```
filename = '../python_data/pyku.txt'

newset = set()                       # a new, empty set

fh = open(filename)                  # open file, return file object

text = fh.read()                     # read() returns the text of the file
words = text.split()                 # split the text string, returns list
                                     # of words

for word in words:                   # for each word in list of words
    word = word.rstrip(';.?!')       # remove punctuation from end of word
    word = word.lower()              # lowercase the word
    newset.add(word)                 # add the word tothe empty set

print(newset)                         # print the compiled set
```

Now we're combining prior exercises with the current one -- reading the file as a string, splitting the string on whitespace, 'preparing' each word by removing punctuation and lowercasing, and then adding each word to a set.

Ex. 4.20 Extend the previous solution by adding the following code at the start:

```
text = "we're certainly out of gouda but Python is great."
```

Now split this text into words (no need to strip or lowercase). Removing the printing of the set and adding to the end of the code, loop through each word in filetext and check the word against the set created earlier (use the in operator to check for membership). If the word is not in the set, print it.

```
filename = '../python_data/pyku.txt'
checktext = "we're certainly out of gouda but Python is great."

newset = set()                      # a new, empty set

fh = open(filename)                 # open file, return file object

filetext = fh.read()                # read() on filehandle returns text
words = filetext.split()            # text is split into list of words

for word in words:                  # for each word in list of words
    word = word.rstrip(';.?!')      # remove any punctuation from word
    word = word.lower()             # lowercase the word
    newset.add(word)                # add the prepared word to the set

test_words = checktext.split()     # split the text into a list of words

for word in test_words:             # for each word in the list of words
    if word not in newset:          # if the same word is not in the set
        print(word)                  # print the word
```

Now the last step: once we have added all the words in the file to a set, we can loop through words from a test string and check each one against the set. This exercise (and the homework) illustrates the use of a set to test for membership. A set is suited to membership tests because a) it contains unique elements and b) the elements are ordered according to Python's internal rules that enable fast lookups.