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

Introduction to Python
Exercise Solutions, Session 7

Ex. 7.1 Replace the following subroutine with a same-named lambda (it returns its argument doubled):

```
def doubleme(num):
    num = num * 2
    return num

print(doubleme(5))
```

Suggested Solution:

```
# commented out original doubleme()
doubleme = lambda x: x * 2

print(doubleme(5))
```

Ex. 7.2 Replace the following subroutine with a same-named lambda (it converts a number like 5 to $5.00):

```
def make_money(num):
  numstr = str(num)
  dolval = '$' + numstr + '.00'
  return dolval

print(make_money(5))
```

Suggested Solution:

```
# commented out original make_money()
make_money = lambda x: '$' + str(x) + '.00'

print(make_money(5))
```

Ex. 7.3 Replace the following subroutine with a same-named lambda (it takes two args and returns the value of the first raised to the power of the second):

```
def power_of_two(num1, num2):
  power = num1 ** num2
  return power

print(power_of_two(3, 3))
```

Suggested Solution:

```
# commented out original power_of_two()
power_of_two = lambda x,y: x ** y

print(power_of_two(3, 3))
```

Ex. 7.4 Replace by_end_num (from our exercises last week) with a same-named lambda. (Hint: it is possible to take a subscript from any method that returns a list - it simply takes a subscript of the returned list):

```
line_list = [
   'the value on this line is 3',
   'the value on this line is 1',
   'the value on this line is 4',
   'the value on this line is 2',
]

def by_end_num(line):
    words = line.split()
    return words[-1]

for line in sorted(line_list, key=by_end_num):
    print(line)
```

Suggested Solution:

```
line_list = [
   'the value on this line is 3',
   'the value on this line is 1',
   'the value on this line is 4',
   'the value on this line is 2',
]

# commented out original by_end_num()
by_end_num = lambda x: x.split()[-1]

for line in sorted(line_list, key=by_end_num):
    print(line)
```

Ex. 7.5 Replace by_num_words (from our exercises last week) with a same-named lambda. (Hint: this would require nesting one function or method inside (or as the arguments list to) another:

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

def by_num_words(line):
    words = line.split()
    return len(words)

for line in sorted(open(pyku).readlines(), key=by_num_words):
    line = line.rstrip()
    print(line)
```

Suggested Solution:

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

# commented out original by_num_words()
by_num_words = lambda x: len(x.split())

for line in sorted(open(pyku).readlines(), key=by_num_words):
    line = line.rstrip()
    print(line)
```

Ex. 7.6 Replace by_last_float (from our exercises last week) with a same-named lambda. (Hint: this combines the hints from the last two exercises.)

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

def by_last_float(line):
    words = line.split(',')
    return float(words[-1])

for line in sorted(open(revenue).readlines(), key=by_last_float):
    line = line.rstrip()
    print(line)
```

Suggested Solution:

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

# commented out original by_last_float()
by_last_float = lambda x: float(x.split(',')[-1])

for line in sorted(open(revenue).readlines(), key=by_last_float):
    line = line.rstrip()
    print(line)
```

Ex. 7.7 Given the code below, write a list comprehension that only returns words longer than 3 letters - print the resulting list.

```
words = [ 'Hello', 'my', 'dear', 'the', 'heart', 'is', 'a', 'child.' ]
```

Suggested Solution:

```
words = [ 'Hello', 'my', 'dear', 'the', 'heart', 'is', 'a', 'child.' ]

rlist = [ x for x in words if len(x) > 3 ]
print(rlist)
```

Ex. 7.8 Build on the previous exercise - modify the list comprehension so that the returned words are uppercased.

Suggested Solution:

```
words = [ 'Hello', 'my', 'dear', 'the', 'heart', 'is', 'a', 'child.' ]

rlist = [ x.upper() for x in words if len(x) > 3 ]
print(rlist)
```

Ex. 7.9 Use a list comprehension to open the file student_db.txt and load it into a list. Print the resulting list - you should see each line of the file as an element in the list - and a newline character at the end of each of the strings. (Hint: a list comprehension loops through any iterable (object that can be looped or iterated through), so you can place the call to open() right in the list comprehension and it will loop over that (or rather, the file object returned from it.)

Suggested Solution:

```
lines = [ line for line in open('../python_data/student_db.txt') ]

print(lines)
```

Ex. 7.10 Continuing the above exercise, use the list comprehension to remove the newline from each line in the file. Print the resulting list - you should see each line of the file as before, but without the newline.

Suggested Solution:

```
lines = [ line.rstrip() for line in open('../python_data/student_db.txt') ]

print(lines)
```

Ex. 7.11 Continuing the above exercise, exclude the 1st line (the header line) from the output. (Hint: file.readlines() returns a list of lines, and you can slice that list. But since we can't assign open() to a filehandle, we can simply treat open() as if it were the filehandle (because it returns it); thus we can call the readlines() method on open(), and of course we can slice any list returned form a method, so we can attach a slice directly to the call to readlines().

Suggested Solution:

```
lines = [ line.rstrip() for line in open('../python_data/student_db.txt').readlines()[1:] ]

print(lines)
```

Ex. 7.12 Build a list of ids: inside the loop, append each id to outer_list. After the loop ends, print the list.

```
import pprint

outer_list = []
fh = open('../python_data/student_db.txt')
lines = fh.readlines()[1:]

for line in lines:
    id, street, city, state, zip = line.split(':')

    outer_list.append(id)

print(outer_list)
print()
print(outer_list[0])
```

A simple list append of strings builds this structure. Compare this to the next solution.

Ex. 7.13 Build a list of lists: continuing from the above solution, inside the loop create an "inner list" of just the id, city and state keyed to strings 'id', 'city', and 'state'.

```
import pprint

outer_list = []
fh = open('../python_data/student_db.txt')
lines = fh.readlines()[1:]

for line in lines:
    id, street, city, state, zip = line.split(':')

    inner_list = [id, city, state]
    outer_list.append(inner_list)

pprint.pprint(outer_list)
print()
print(outer_list[0][1])
```

This list append is the same as the simple list append, but this time we are appending a list instead of a string. So instead of appending a single field as an item we're appending multiple fields -- this is the essence of a multidimensional container.

Ex. 7.14 Build a list of dicts: modifying the above solution, instead of an inner list, inside the loop create an "inner dict" of the id, city and state keyed to strings 'id', 'city', and 'state'.

```
import pprint

outer_list = []
fh = open('../python_data/student_db.txt')
lines = fh.readlines()[1:]

for line in lines:
    id, street, city, state, zip = line.split(':')

    inner_dict = { 'id': id, 'city': city, 'state': state }
    outer_list.append(inner_dict)

pprint.pprint(outer_list)
print()
print(outer_list[3]['city'])
```

Again, we append a container of values instead of a single value. The only difference here is that the inner structure is a dict instead of a list.

Ex. 7.15 Build a dict of ids paired to states: inside the loop, add a key/value pair to outer_dict: the id paired with the state.

```
import pprint

outer_dict = {}
fh = open('../python_data/student_db.txt')
lines = fh.readlines()[1:]

for line in lines:
    id, street, city, state, zip = line.split(':')

    outer_dict[id] = state

pprint.pprint(outer_dict)
print()
print(outer_dict['ak9'])
```

Now working with an outer dict instead of an outer list, we begin by reviewing the simple add of key/value pair to a dict -- in this case the student id paired with the student's state.

Ex. 7.16 Build a dict of dicts: inside the loop, create an "inner dict" of street, city, state and zip, and add this dict paired with the id.

```
import pprint

outer_dict = {}
fh = open('../python_data/student_db.txt')
lines = fh.readlines()[1:]

for line in lines:
    id, street, city, state, zip = line.split(':')

    inner_dict = {'street': street, 'city': city,
                  'state': state, 'zip': zip }
    outer_dict[id] = inner_dict

pprint.pprint(outer_dict)
print()
print(outer_dict['ak9']['state'])
```

Continuing the same progression as with the list, here we are pairing the student id with the entire row of data for the id, in the form of a convenient dict. So as we did with the list of dicts, inside the loop we simply create a new "inner" dict out of the row data that we want to include, and then pair the student id with this "inner" dict.

Ex. 7.17 Build a "counting" dictionary, a dict of ints: as a review, create a counting dictionary that counts the number of occurrences of each state.

```
import pprint

outer_dict = {}
fh = open('../python_data/student_db.txt')
lines = fh.readlines()[1:]

for line in lines:
    id, street, city, state, zip = line.split(':')

    if state not in outer_dict:
        outer_dict[state] = 0

    outer_dict[state] = outer_dict[state] + 1

print(outer_dict)
print(); print(outer_dict['NY'])
```

Now stepping back again, we review the "counting" dictionary we employed in the unit on dictionaries. As we did then, we are looping through data that contains repeating keys (the id). We must consider whether or not the id key is new to the dictionary, and if it is, we need to set an initial key/value pair in the dict -- setting the value for this key to 0. Then, whether or not the key is new, we can increment the value -- so if the key is new, the value will be 1; and if it is not new, it will be incremented from its previous value. (We're returning to the counting dictionary in order to prepare to do something very similar with a dict of lists - an aggregation of values associated with a state.)

Ex. 7.18 Build a dict of lists (states): modify the above solution so that each state is associated with a list of ids for that state.

```
import pprint

outer_dict = {}
fh = open('../python_data/student_db.txt')
lines = fh.readlines()[1:]

for line in lines:
    id, street, city, state, zip = line.split(':')

    if state not in outer_dict:
        outer_dict[state] = []

    outer_dict[state].append(id)

pprint.pprint(outer_dict)
print()
for id in outer_dict['NJ']:
    print(id)
```

Note well the extreme similarity between the "check and add" code in this solution, and the one used in a standard counting dictionary. In the counting dict, we're setting the value to 0 if the key is new, and then incrementing the value. In this dict of lists, we're setting the value to [] (empty list) if the key is new, and then appending the current id value to the list. So we've simply replaced the integer counter that can be incremented with a list that can be appended to.

The point of this program is to be able to aggregate students to their home states. We might do this if we wanted to send a mailer to all the students from New York, for example.

Ex. 7.19 Using the file "revenue.txt", build a "summing" dictionary, a dict of floats -- as a review, create a summing dictionary that sums up the amount of revenue garnered from each state.

```
import pprint

outer_dict = {}
fh = open('../python_data/revenue.txt')

for line in fh:
    company, state, revenue = line.split(',')

    if state not in outer_dict:
        outer_dict[state] = 0.0

    outer_dict[state] = outer_dict[state] + float(revenue)

print(outer_dict)
print()
print(outer_dict['NJ'])
```

Again returning to a review, we see the standard "summing" dictionary that we used in the unit on dicts. Of course this is extremely similar to the counting dictionary, but instead of using an int that is incremented, we are using a float that is summed with other floats that are on the same line as the state key.

Ex. 7.20 Build a dict of lists (floats): modify the above solution so that each state is associated with a list of revenue floats for that state.