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

Introduction to Python
Exercises, 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))
```

Expected Output:

```
10
```

If desired, you can test the following exercises by placing the example function and your lambda function side by side, and by calling them the same way. Example from the slides:

```
def addthese(x, y):
    return x + y


# WE DON'T USE LAMBDAS LIKE THIS
# BUT THIS CAN BE USED FOR TESTING
addthese2 = lambda x, y:  x + y

print(addthese(5, 9))        # 14
print(addthese2(5, 9))       # 14
```

(PLEASE NOTE that this is not how we usually use lambdas, but it can be useful for testing. The usual use of lambdas is in a funtion like sorted() or in other places like dictionaries (a special dict called a dispatch table) -- we are primarily interested in lambdas for sorting).

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))
```

Expected Output:

```
$5.00
```

Note: lambdas are never actually assigned to a variable name (which sort of defeats their purpose -- to provide an "inline" function) but we are naming lambdas for testing purposes in these exercises.

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))
```

Expected Output:

```
27
```

Note: lambdas are never actually assigned to a variable name (which sort of defeats their purpose -- to provide an "inline" function) but we are naming lambdas for testing purposes in these exercises.

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)
```

Expected Output:

```
the value on this line is 1
the value on this line is 2
the value on this line is 3
the value on this line is 4
```

Note: lambdas are never actually assigned to a variable name (which sort of defeats their purpose -- to provide an "inline" function) but we are naming lambdas for testing purposes in these exercises.

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)
```

Expected Output:

```
We're out of gouda.
Spam, spam, spam, spam, spam.
This parrot has ceased to be.
```

Note: lambdas are never actually assigned to a variable name (which sort of defeats their purpose -- to provide an "inline" function) but we are naming lambdas for testing purposes in these exercises.

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)
```

Expected Output:

```
Dothraki Fashions,NY,5.98
Hipster's,NY,11.98
Awful's,PA,23.95
Westfield,NJ,53.90
The Clothiers,NY,115.20
The Store,NJ,211.50
Haddad's,PA,239.50
```

Note: lambdas are never actually assigned to a variable name (which sort of defeats their purpose -- to provide an "inline" function) but we are naming lambdas for testing purposes in these exercises.

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.' ]
```

Expected Output:

```
['Hello', 'dear', 'heart', 'child']
```

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

Expected Output:

```
['HELLO', 'DEAR', 'HEART', 'CHILD.']
```

Ex. 7.9 Access the file student_db.txt on your file system (part of the files included in the python_data folder, downloaded as ALL_DATA.zip). Use a list comprehension to open the file 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.)

Expected Output:

```
['id:address:city:state:zip\n', 'jk43:23 Marfield Lane:Plainview:NY:10023\n',
 'ZXE99:315 W. 115th Street, Apt. 11B:New York:NY:10027\n',
 'jab44:23 Rivington Street, Apt. 3R:New York:NY:10002\n',
 'ak9:234 Main Street:Philadelphia:PA:08990\n',
 'ap172:19 Boxer Rd.:New York:NY:10005\n',
 'JB23:115 Karas Dr.:Jersey City:NJ:07127\n',
 'jb29:119 Xylon Dr.:Jersey City:NJ:07127\n']
```

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.

Expected Output:

```
['id:address:city:state:zip', 'jk43:23 Marfield Lane:Plainview:NY:10023',
 'ZXE99:315 W. 115th Street, Apt. 11B:New York:NY:10027',
 'jab44:23 Rivington Street, Apt. 3R:New York:NY:10002',
 'ak9:234 Main Street:Philadelphia:PA:08990',
 'ap172:19 Boxer Rd.:New York:NY:10005',
 'JB23:115 Karas Dr.:Jersey City:NJ:07127',
 'jb29:119 Xylon Dr.:Jersey City:NJ:07127']
```

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().

Expected Output:

```
['jk43:23 Marfield Lane:Plainview:NY:10023',
 'ZXE99:315 W. 115th Street, Apt. 11B:New York:NY:10027',
 'jab44:23 Rivington Street, Apt. 3R:New York:NY:10002',
 'ak9:234 Main Street:Philadelphia:PA:08990',
 'ap172:19 Boxer Rd.:New York:NY:10005',
 'JB23:115 Karas Dr.:Jersey City:NJ:07127',
 'jb29:119 Xylon Dr.:Jersey City:NJ:07127']
```

For the next group of exercises, we'll start with standard containers (lists, dicts) and build them out to multi-dimensional containers (lists of lists, dicts of dicts, etc.) I strongly recommend reading through the discussion after each solution, which can help draw your attention to the similarities between all of the structures we build here, whether a simple dictionary or list to a dict of dicts, a list of dicts, etc.

If you compare each structure built, you'll see that each simply varies in the structure type used (list or dict) and the corresponding methods of adding an item or key/value (i.e., list append() or dict subscript adding).

Building a list of lists or lists of dicts -- begin with the following:

```
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(':')
```

Ex. 7.12 Build a list of ids: inside the loop, append each id to outer_list. After the loop ends, print the list. Element access: access and print the first element in the list.

Expected Output:

```
['jk43', 'ZXE99', 'jab44', 'ak9', 'ap172', 'JB23', 'jb29']

jk43
```

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, and instead of appending the id, append this 3-element list to outer_list. After the loop ends, pretty-print the resulting list of lists (i.e. pprint.pprint(outer_list)). Element access: access the city of the first row (i.e., the 2nd element of 1st list) using a double subscript.

Expected Output:

```
[['jk43', 'Plainview', 'NY'],
 ['ZXE99', 'New York', 'NY'],
 ['jab44', 'New York', 'NY'],
 ['ak9', 'Philadelphia', 'PA'],
 ['ap172', 'New York', 'NY'],
 ['JB23', 'Jersey City', 'NJ'],
 ['jb29', 'Jersey City', 'NJ']]

Plainview
```

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 string keys 'id', 'city' and 'state'. Your "inner dict" should look like this:

```
inner_dict = {'id': id, 'city': city, 'state': state }
```

Now append inner_dict to the outer list. Once the loop is done, pretty-print the resulting list of dicts. Element access: print Philadelphia from the 4th row.

Expected Output:

```
[{'city': 'Plainview', 'id': 'jk43', 'state': 'NY'},
 {'city': 'New York', 'id': 'ZXE99', 'state': 'NY'},
 {'city': 'New York', 'id': 'jab44', 'state': 'NY'},
 {'city': 'Philadelphia', 'id': 'ak9', 'state': 'PA'},
 {'city': 'New York', 'id': 'ap172', 'state': 'NY'},
 {'city': 'Jersey City', 'id': 'JB23', 'state': 'NJ'},
 {'city': 'Jersey City', 'id': 'jb29', 'state': 'NJ'}]

Philadelphia
```

Building a dict of lists or dict of dicts -- begin with the following starter code (same as above, but starting with an empty dict):

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

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

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. Once the loop is done, print the dict. Element access: print the value for key ak9.

Expected Output:

```
{'ak9': 'PA',
 'ap172': 'NY',
 'ZXE99': 'NY',
 'jab44': 'NY',
 'JB23': 'NJ',
 'jb29': 'NJ',
 'jk43': 'NY'}

PA
```

Ex. 7.16 Build a dict of dicts: inside the loop, create an "inner dict" of street, city, state and zip. Your "inner dict" should look like this:

```
inner_dict = {'street': street, 'city': city, 'state': state, 'zip': zip }
```

Then, still inside the loop, add a key/value pair to outer_dict: the id paired with the inner_dict. Once the loop is done, pretty-print the resulting dict of dicts. Element access: print the state for ak9.

Expected Output:

```
{'JB23': {'city': 'Jersey City',
          'state': 'NJ',
          'street': '115 Karas Dr.',
          'zip': '07127\n'},
 'ZXE99': {'city': 'New York',
           'state': 'NY',
           'street': '315 W. 115th Street, Apt. 11B',
           'zip': '10027\n'},
 'ak9': {'city': 'Philadelphia',
         'state': 'PA',
         'street': '234 Main Street',
         'zip': '08990\n'},
 'ap172': {'city': 'New York',
           'state': 'NY',
           'street': '19 Boxer Rd.',
           'zip': '10005\n'},
 'jab44': {'city': 'New York',
           'state': 'NY',
           'street': '23 Rivington Street, Apt. 3R',
           'zip': '10002\n'},
 'jb29': {'city': 'Jersey City',
          'state': 'NJ',
          'street': '119 Xylon Dr.',
          'zip': '07127\n'},
 'jk43': {'city': 'Plainview',
          'state': 'NY',
          'street': '23 Marfield Lane',
          'zip': '10023\n'}}

PA
```

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 -- a dictionary with unique state keys paired with an integer count reflecting the number of states in the file. As we did with counting or summing dictionaries, remember that as your loop encounters each state, it must check to see if the state key already exists in the dictionary. If it is new to the dict, it should be added with a value of 0. Then the dict should add 1 to the value associated with that state in the dict. Element access: print the number of students who are from New York.

Expected Output:

```
{'NY': 4, 'NJ': 2, 'PA': 1}

4
```

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 -- so you are building a dict that pairs each state with a list of ids. You can do this very simply by replacing the integer 0 with an empty list, and the integer incrementing with a list append(). Element access: loop through and print all the student ids for the students from New Jersey

Expected Output:

```
{'NJ': ['JB23', 'jb29'],
 'NY': ['jk43', 'ZXE99', 'jab44', 'ap172'],
 'PA': ['ak9']}

JB23
jb29
```

For the next exercises, please begin by glancing at revenue.txt, and then start with the following starter code:

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

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

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 -- a dictionary with unique state keys paired with a float sum reflecting the sum of float values for that state in the file. As we did with counting or summing dictionaries, remember that as your loop encounters each state, it must check to see if the state key already exists in the dictionary. If it is new to the dict, it should be added with a value of 0.0. Then the dict should add the float value to the value associated with that state in the dict. Element access: print the revenue found for New Jersey.

Expected Output:

```
{'NY': 133.16, 'NJ': 265.4, 'PA': 263.45}

265.4
```

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 -- so you are building a dict that pairs each state with a list of revenue values for the state. You can do this very simply by replacing the float 0.0 with an empty list, and the float summing with a list append(). Element access: provide the average company revenue for New York.

Expected Output:

```
{'NJ': [53.9, 211.5], 'NY': [11.98, 5.98, 115.2], 'PA': [239.5, 23.95]}

44.3866666667
```