Arizona State University

W. P. Carey School of Business

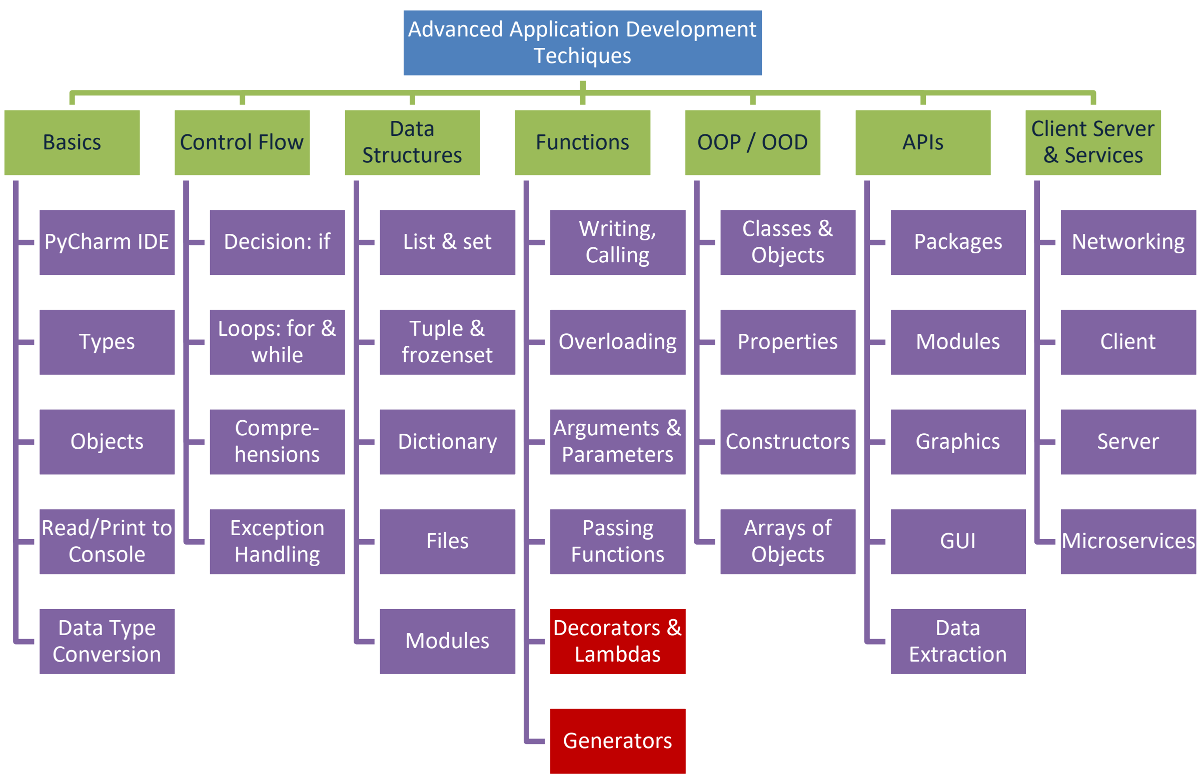
CIS 345 – Business Information Systems Development - II

PE 5: Advanced Functions

Learning Outcomes

1.1. Implement the functions map, zip, and filter
1.2. Implement comprehensions
1.3. Implement generator functions and generator comprehensions
1.4. Implement lambda expressions

Topic Chart



Program Overview

This exercise is meant to give you practice in writing advanced Functions. We will use several built-in functions such as map, zip, and filter. On top of those we will built generator functions and comprehensions which allow us to work with large amounts of data in a lazy way yielding 1 element at a time into memory while going through a large data structure.

We will make use of comprehensions and lambda expressions to quickly manipulate our data so that it can be formatted and placed into a data structure of our design.

This exercise will be a little different as we are going to approach the application in a "what if" scenario. We will walk through how to take input from the user and transform it using these powerful techniques.

Scenario: Assume you are creating a program that is going to handle restaurant reviews using survey information. We will first write code to gather information from the restaurant owner's perspective. Following we will create a test scenario and code how to capture survey inputs from a customer. After that code we will write code to randomly generate many surveys. Finally, our last section will manipulate the survey data to determine the average score per customer review/survey and then compute the overall average rating per category for all surveys.

Sample Output

Part 1:

```
How many categories do you want to rated for your restaurant: 5

Enter the categories you want rated for your restaurant
Category 1: Food
Category 2: Service
Category 3:
Category 4: Price
Category 5:
['Food', 'Service', '', 'Price', '']
['Food', 'Service', 'Price']
```

Part 2:

```
On a scale of low being 1 to 5 being high
rate your service in the following categories
Food: 2
Quality: 3
Value: 4
Service: 5
Staff: 0
Ambiance: 5
['2', '3', '4', '5', '0', '5']
```

Part 3:

```
[('Food', 2), ('Quality', 3), ('Value', 4), ('Service', 5), ('Staff', 0), ('Ambiance', 5)]
[('Food', 2), ('Quality', 3), ('Value', 4), ('Service', 5), ('Ambiance', 5)]
```

Part 4 – Prints 100 Avg. ratings then the Overalls:

```
Avg rating: 3.6
Avg rating: 3.8
Avg rating: 3.1666666666666665
Avg rating: 2.8
Avg rating: 2.6
Avg rating: 4.166666666666667
Avg rating: 4.4
Avg rating: 3.1666666666666665
Avg rating: 3.6
Avg rating: 2.2
Avg rating: 2.5
Avg rating: 3
Avg rating: 3.6666666666666665
Avg rating: 3.6666666666666665
Avg rating: 4.4
Avg rating: 2.75

Average Rating per Category:
Food: 3.2222
Quality: 2.9659
Value: 2.8889
Service: 3.2439
Staff: 3.5542
Ambiance: 3.0000
```

All parts uncommented – Allow additional categories

```
How many categories do you want to rated for your restaurant: 2

Enter the categories you want rated for your restaurant
Category 1: Speed
Category 2: Clean
Avg rating: 2.875
Avg rating: 2.4
Avg rating: 2.5714285714285716

Avg rating: 3.142857142857143

Average Rating per Category:
Food: 3.0227
Quality: 2.9176
Value: 3.1235
Service: 3.3218
Staff: 2.9405
Ambiance: 2.6049
Speed: 3.1905
Clean: 2.8621
```

Instructions

- Create a PyCharm Project using standard naming convention and add a Python file:
    - [ASUrite]_survey.py
- When done submit your python module file only.
- Don't forget to put your name, class, class time, and assignment number in **as a comment** on Line 1 of all modules.

Important notes:

- Function names are all lowercase using snake_case (words separated by underscore)
- PEP8 coding standard is to have 2 blank lines above and below all function definitions
- All functions must have a docstring comment on first line after def using 3 double quotes

```
def function_name (ParameterName):
    “”” DocString Comment describing the function “””
    <code statements aka function body>
```

Before beginning we will import two modules that we will use later:

```
import random
import statistics
```

Declare two blank lists:

```
categories = [ ]
entries = [ ]
```

Part 1 – Get Categories to be rated by customers from business Owner:

A. First we create a generator function:

Generator Function Name: count_up
Parameters: **end**
<u>Yield</u>*:* string containing the data type of the argument passed to this function
Purpose: Yield numbers starting at 1 that count up to and including the **end**.

In the body of the function,

- Declare n and initialize to 1
- Loop while n is less than or equal to end
    - Yield n
    - Increment n by adding 1

Note: What makes a generator function is the use of the keyword yield which replaces return. Functions have return and generator functions use yield.

B. Outside the function let's start writing our program. Ask the restaurant owner for how many categories they want rated on a survey. Use input to prompt the user 'How many categories do you want rated for your restaurant: ' and store the answer in ***amount***.

C. Call count_up passing the amount as an argument, but you must first convert amount to int(). The count_up generator function will return a generator object which must be stored in a name called ***count***: `count = count_up(int(amount))`

   Try `print(count)` to see the generator object and then remove this print line of code.

D. Display a blank line followed by 'Enter the categories you want rated for your restaurant'

E. Loop through each count value or ***c*** in your ***count*** generator object so we can prompt the user to enter the categories they want rated. We will place the count number ***c*** in the prompt after the word Category:

```
for c in count:
    categories.append(input(f'Category {c}: '))
```

F. We have the code to get the categories from a restaurant owner that they want rated on a survey. But what if a user enters no data all. Let's filter out any blank entries using the filter function after the loop in E exits:

   `filter( <func>, <iterable> )`
   <func> is the function you want applied to the iterable data structure or you can put **None** as an argument and it will remove any False values (eg. 0, blank strings, False, etc.)

   <iterable> is a data structure that you can iterate over such as list.

   Try this code to see how filter works:

```
print(categories) # remove or comment this line after test
filtered_categories = list( filter( None, categories )
print(filtered_categories) # remove or comment this line after test
```

   To test see sample screenshots for part 1 – This is the end of part 1

G. After testing this code works we want to comment it out to avoid all these entries when testing the next parts of our program. Use a multiline comment to comment out B through F (you can also hide the code using the – symbol on the left of line of code). In place of B – F assign test data to categories:

```
categories = ['Food', 'Quality', 'Value', 'Service', 'Staff', 'Ambiance']
```

   Note: In the end we will uncomment B – F (you can leave it uncommented, but it will slow down testing). In the end, you will have the above categories and the entries add on to them.

Part 2 – Get ratings from Customer

A. Display the below message:

```
On a scale of low being 1 to 5 being high
rate your service in the following categories
```

B. For each category ***c*** in categories prompt the user to enter a rating. Take the entered rating and append it to the list ***entries*** which we declared at the top of our file.

C. Test this part to ensure you get correct entries from the user. I suggest printing entries to see that you get string entries for each of our test categories (see Part 2 screenshot). After testing remove any testing print statements.

D. Use a multi-line comment and comment out A-C. Insert test data below above commented code to speed up testing of the next parts. Let's place the list of ratings we entered into entries:

```
entries = ['2', '3', '4', '5', '0', '5']
```

Part 3 – Process ratings

Important Note: Each letter below has a test that will print results to the screen and then you will be asked to delete/comment the print statements. FAILURE to delete a print will exhaust the object and no values will be left to yield for later code. **You must do exactly as stated!!!**

A. Looking at the test data in entries, the first step is to convert all strings to numeric so we can analyze them using math. We will use the map() function to do this quickly:

```
map( <func>, <iterable>)
```

Executes the function passed into <func> on all values in the <iterable> data structure and returns a map object. <func> can be a lambda expression.

We will pass in the function int and our entries and map will run int on all strings in entries:

```
rating = map(int, entries)
```

Test results to see the map object and you can convert the map object to a list to see values:

print(rating) and print(list(rating)) # Must delete prints after you see results are good

B. Next we will create a data structure that pairs each category with the numeric rating:

zip(<iterable1>, <iterable2>):

Create a tuple for each paired value in both <iterable1> and <iterable2> and return a zip object. If iterables are not the same length, zip will stop pairing when the smallest one runs out of values.

Combine categories and rating:

```
restaurant_score = list(zip(categories, rating))
```

Test and then delete print: `print( restaurant_score )` #can delete after testing C

C. Now we will filter out any category that doesn't have a rating or the rating is 0 since our instructions told the user to enter 1-5 by using a comprehension. Notice in our test data printed in B that Staff has a rating of 0, so we don't want to just remove 0 because then our rating of 5 might end up being aligned with Staff incorrectly. Instead we want to remove the Tuple ('Staff', 0) and make a new list. If we coded this normally we would do the below to loop through restaurant scores using ***d*** and look at index 1 or d[1] and test if it is not a rating of 0. If true then append the whole tuple to our list ***f*** :

```
f = []
for d in restaurant_score:
    """d is a tuple where d[0] is Staff and d[1] is 0"""
    if d[1] != 0:
        f.append(d)
```

Instead a comprehension can do this in one line and is basically the same code:

```
restaurant_filtered_score = [d for d in restaurant_score if d[1] != 0]
```

We have the same loop and decision. All values in restaurant_score that evaluate to true in our decision will then be assigned to our filtered list. d represents the values being assigned.

Test your code. I suggest leaving the print in B so you can see the filtered and unfiltered list and adding: `print(restaurant_filtered_score)`

After testing and your output matches the Part 3 screenshot you may delete this print and the print from step B.

D. At this point if you combine Part 1, 2, and 3 you can enter the categories for a survey, the user can enter ratings for each category, and the application processes the data (convert to numeric, pair category with rating, and filter un-rated categories).

Part 4 – Use a Generator to create many Surveys and compute Averages

A. We will return here in a moment, but first we need to go back to the top of the file to define a generator function. Add the below generator after the ***count_up*** generator:

Generator Function Name: generate_surveys
Parameters: number
Yield: filtered_score
Purpose: We will simulate steps 2 and 3 and generate and process surveys <number> times.

In the body of the function,

- Add docstring comment describing functionality
- For each survey in range(number) do the following
  - Declare a blank list and name it ***ratings***
  - for each loop to go through category in categories
    - Use random.randint to generate random ratings and append to ratings:

```
ratings.append(random.randint(0,5))
```

  - `score = list(zip(categories, ratings))`
  - `filtered_score = (s for s in score if s[1] != 0)`

    Note here we are using a generator comprehension to save memory which is done using parenthesis rather than a list comprehension using brackets.

  - yield `filtered_score`

After coding the generator, return to the bottom of your file to pick-up our logic where you left off.

B. Using our new generator, create 100 surveys by calling `generate_surveys(100)` and store them in ***surveys***. Test by looping through surveys and print each one, but then delete your loop and print.

C. Next we will use a lambda expression (aka anonymous function) in conjunction with the map function. We will build the following lambda which is a function that will convert x to a dictionary:

```
lambda x: dict(x)
```

Let's pass the above function to map and have that dictionary conversion be applied to every survey in surveys:

```
results = map(lambda x: dict(x), surveys)
```

Test and `print(list(results))` to see all dictionaries and then delete this print.

D. Let's setup two dictionaries to hold data to compute the average rating per category. One will be called ***sums*** and will hold the sum of each category (all Food ratings summed, all Service ratings summed, etc.). The other will be named ***lengths*** holding the number of total ratings per category. For this to work we want to create these dictionaries with the categories as keys and all values as zero. This will enable us to modify the values while processing each survey (recall dictionaries are mutable and can be modified). To do this we will use the below dictionary comprehension using braces for both ***sums*** and ***lengths***:

```
sums = {c:0 for c in categories}
lengths = {c:0 for c in categories}
```

E. This next part is better explained using a picture so I have printed the dictionary to help visualize what we are going through. The map object results is going to yield 1 survey at a time in a dictionary format. To see it I have run the dunder (dunder stands for double underscore) next function on results 3 times:

```
results = map(lambda x: dict(x), surveys)
...
print(next(results))
print(next(results))
print(next(results))
```

```
survey ×
C:\Users\clolsen2\PycharmProjects\PE6-Surveys\venv\Scripts\python.exe C:/Users/cl
{'Food': 5, 'Quality': 2, 'Value': 4, 'Service': 5, 'Staff': 5, 'Ambiance': 5}
{'Food': 2, 'Quality': 1, 'Value': 5, 'Service': 3, 'Staff': 2, 'Ambiance': 4}
{'Food': 1, 'Quality': 1, 'Value': 2, 'Service': 5, 'Staff': 3, 'Ambiance': 3}
```

We are going to loop for **survey** in **results**: which 1 row above is a single survey in a dictionary format.

Let's make use of the mean() function in the statistics module to calculate the mean of all ratings in each survey and print it to the screen:

```
print('Avg rating: {}'.format(statistics.mean(survey.values())))
```

Within this same loop we need to nest a loop to go through each category and add each category rating to a total. Now we face an issue because we could have surveys that don't have all categories since we filtered out zero ratings and their corresponding category. We must use a try statement and handle KeyError when a category has been filtered out of a survey. If an exception occurs, we want to do nothing and keep running. This is accomplished using the pass keyword:

```
for cat in categories:
    try:
        sums[cat] += survey[cat]
        lengths[cat] += 1
    except KeyError:
        pass
```

This ends our loop for survey in results.

F. Lastly we have each categories sum in ***sums*** and the number of ratings for each category in ***lengths***. Display a header for our Averages. Let's loop for each cat in categories and print the overall average per category for all surveys:

```
print(f'{cat}: {sums[cat] / lengths[cat]:.4f}')
```

Test your code to ensure it works properly and **Ensure you uncomment parts 1-2**. Recall we commented out code we previously tested so we could focus on testing the new code we were adding. Go back and uncomment those sections of code so that the application works as described in the Program Overview section.

Things to Try

1. If you have declared all test data for part 1 and 2 before the code that asks the user for input, your program can add additional categories to the defaults in the test data. Our 1 user entered survey is ignored. Could you include the entered survey as the first survey and then append 99 simulated surveys after it?
2. We only filtered 0 ratings from our survey, but with a small edit to the comprehension's decision we could also filter any ratings that are over 5. Give it a try.

Submission:

1. Submit your completed python module with all code running and uncommented only. No zip file for this submission.