MCT-243 : COMPUTER PROGRAMMING-II

using Python 3.9

Prepared By:

Mr. Muhammad Ahsan Naeem

YouTube Playlist

https://www.youtube.com/playlist?list=PLWF9TXck7O_wMDB-VriREZ6EvwkWLNB7q

©Department of Mechatronics and Control Engineering, University of Engineering and Technology, Lahore, Pakistan

Lab 30: ZIP, MAP, FILTER & REDUCE

Zip Function:

`zip()` function zips together two or more iterables so that they all can be iterated in a single loop. Let's see its help as:

```
help(zip)
```

Starting few output lines are here:

```
Help on class zip in module builtins:

class zip(object)
 |  zip(iter1 [,iter2 [...]]) --> zip object
 |
 |  Return a zip object whose .__next__() method returns a tuple where
 |  the i-th element comes from the i-th iterable argument.  The .__next__()
 |  method continues until the shortest iterable in the argument sequence
 |  is exhausted and then it raises StopIteration.
```

The following line explains the function signature:

```
zip(iter1 [,iter2 [...]]) --> zip object
```

It says that `zip()` function takes in at least one iterable and at maximum as many as required and returns one zip object. The next lines explain that `__next__()` method will return a tuple with size of number of iterables passed in and each element of the tuple is corresponding element of the iterables. Zip is an **iterator** and `__next__()` is a special or the magic method for any iterator, which is called automatically in a loop `for` loop or some other function that needs all elements of the iterator.

What is iterator?

Iterator is different from iterable. Although we can directly apply the `for` loop on an iterator just like an iterable but there are many things different in an iterator. In next lab session we will explore in detail the difference between **iterable** and **iterator**. At this stage you should simply know that zip object is an iterator and the elements inside an iterator are not readily available to be used but they are produced when needed, e.g., in a `for` loop or in some other function like `sum`, `min`, `max` etc. Secondly, an iterator is just one time usable. Unlike a List or some other iterable like Tuple, Set, Dictionary etc., an iterator can be used just once, e.g. in a `for` loop or in some other function which needs all elements of the iterator. Once it has been used, it becomes empty, or we also say that the iterator has **exhausted**. More detail on iterator and iterable will be discussed in next lesson.

Let's see the following example:

Suppose we have a list of subjects and credit hours. If we want to process both lists together, we can zip them together as:

```
subj=['Mech','CP2','ES','MoM']
CH=[3,2,4,3]
z=zip(subj,CH)
print(z)
```

This will print the zip object as:

```
<zip object at 0x0000018F95E1DD48>
```

We can convert the zip object to other data type e.g. **List** as shown here:

```
l=list(z)
print(l)
```

This will print the list as:

```
[('Mech', 3), ('CP2', 2), ('ES', 4), ('MoM', 3)]
```

You can see that it is a list of tuples with first element of tuple is from first list and second from the second list. Instead of converting zip object to list we could use that in `for` loop. Complete code is shown here:

```
subj=['Mech','CP2','ES','MoM']
CH=[3,2,4,3]
z=zip(subj,CH)
for i in z:
    print(i)
```

It will print:

```
('Mech', 3)
('CP2', 2)
('ES', 4)
('MoM', 3)
```

So, when to use zip function?

We should use `zip()` function, when we want to process multiple iterables together. Without using the `zip()` function, we would have to process multiple iterables via the common index. The `zip()` function helps us in a way that it picks corresponding elements from different iterables and creates Tuples of those and then we can iterate those Tuples having corresponding elements from each iterable.

There is the possibility to have different types of iterables in `zip()` function as shown here with the same result as obtained earlier:

```
subj=['Mech','CP2','ES','MoM']
CH=(3,2,4,3)
z=zip(subj,CH)
for i in z:
    print(i)
```

Here is example with three lists zipped together:

```
subj=['Mech','CP2','ES','MoM']
CH=(3,2,4,3)
instructors=['Dr. Ali','Ahsan Naeem','Ahsan Masud','Dr. Baneen']
z=zip(subj,CH,instructors)
for i in z:
    print(i)
```

The output will be:

```
('Mech', 3, 'Dr. Ali')
('CP2', 2, 'Ahsan Naeem')
('ES', 4, 'Ahsan Masud')
('MoM', 3, 'Dr. Baneen')
```

So far, we have simply displayed the elements inside zip object, and they are tuples but we can always access the individual elements of tuples. An example is shown here:

```
subj=['Mech','CP2','ES','MoM']
CH=(3,2,4,3)
instructors=['Dr. Ali','Ahsan Naeem','Ahsan Masud','Dr. Baneen']
z=zip(subj,CH,instructors)
for s,c,i in z:
    print(f"{s} has {c} Credit Hours and will be taught by {i}")
```

The output is shown here:

```
Mech has 3 Credit Hours and will be taught by Dr. Ali
CP2 has 2 Credit Hours and will be taught by Ahsan Naeem
ES has 4 Credit Hours and will be taught by Ahsan Masud
MoM has 3 Credit Hours and will be taught by Dr. Baneen
```

For a moment here, imagine doing the above task without zip function and see the easiness we are getting with zip function.

Let's explore zip function a bit more!

If the length of different iterables is not same in `zip()` function, it will not generate an error, instead it will zip the corresponding elements till the shortest-length iterable. Read the help statement to understand it further.

Here is the same example with different length of iterables by removing the **Credit Hours** of the last subject **MoM**:

```
subj=['Mech','CP2','ES','MoM']
CH=(3,2,4)
instructors=['Dr. Ali','Ahsan Naeem','Ahsan Masud','Dr. Baneen']
z=zip(subj,CH,instructors)
for s,c,i in z:
    print(f"{s} has {c} Credit Hours and will be taught by {i}")
```

It will have the output as:

```
Mech has 3 Credit Hours and will be taught by Dr. Ali
CP2 has 2 Credit Hours and will be taught by Ahsan Naeem
ES has 4 Credit Hours and will be taught by Ahsan Masud
```

It is also possible to continue zipping till the longest iterable. For that we need to use `zip_longest()` function in module `itertools` and providing a value that will be used in place of shorter elements in all short length iterables. This value is passed by keyword `fillvalue`. If we do not specify the optional argument `fillvalue`, it will have `None` as its default value. Above example with this feature is shown here:

```
from itertools import zip_longest
subj=['Mech','CP2','ES','MoM']
CH=(3,2,4)
instructors=['Dr. Ali','Ahsan Naeem','Ahsan Masud','Dr. Baneen']
z=zip_longest(subj,CH,instructors,fillvalue='x')
for s,c,i in z:
    print(f"{s} has {c} Credit Hours and will be taught by {i}")
```

The output is as:

```
Mech has 3 Credit Hours and will be taught by Dr. Ali
CP2 has 2 Credit Hours and will be taught by Ahsan Naeem
ES has 4 Credit Hours and will be taught by Ahsan Masud
MoM has x Credit Hours and will be taught by Dr. Baneen
```

Sorting in parallel:

Sorting is one very commonly used operation. Sometimes it might be a case that we have multiple collection (or sequence) and we want them to get sorted based on one of them. For example, in case of three lists we are using in previous examples and shown here:

```
subj=['Mech','CP2','ES','MoM']
CH=[3,2,4,3]
instructors=['Dr. Ali','Ahsan Naeem','Ahsan Masud','Dr. Baneen']
```

What if we want to sort all data based on Credit Hours? We can sort `CH` list but the other two lists will not change their order accordingly. Here we can use the `zip()` function to first zip the data in the form of tuple with `CH` at first place and then apply sort on that list of tuples. When those tuple objects within the list will get sorted, it will be the whole tuple with the information of other two lists too. Complete code is shown here:

```
subj=['Mech','CP2','ES','MoM']
CH=[3,2,4,3]
instructors=['Dr. Ali','Ahsan Naeem','Ahsan Masud','Dr. Baneen']
mainList=list(zip(CH,subj,instructors)) #List of Tuples
mainList.sort() #Will be sorted on the basis of CH
for c,s,i in mainList:
    print(f"{s} has {c} Credit Hours and will be taught by {i}")
```

Tasks:

[1] Recall the task related to **Survey for favorite Beverages** we did in previous course and given here again:

Write a program that performs a survey tally on beverages. The program will ask user to enter favorite beverage of first person and then should prompt for the next person until a value 0 is entered to terminate the program. Each person participating in the survey should choose their favorite beverage from the following list:

1. Coffee **2. Tea** **3.Coke** **4. Orange Juice**

After the survey is complete indicated by entering a 0, the program should display the result in form of total number of persons participating in the survey and number of votes against each beverage in **ascending order.**

Sample output is:

```
*****************************************************************
1. Coffee         2. Tea          3.Coke          4. Orange Juice
*****************************************************************
This is survey for the favorite Beverage.

Choose (1-4) from the above menu or 0 to exit the program.

Please input the favorite beverage of person #1: 4
Please input the favorite beverage of person #2: 1
Please input the favorite beverage of person #3: 3
Please input the favorite beverage of person #4: 1
Please input the favorite beverage of person #5: 1
Please input the favorite beverage of person #6: 0


The results are as follows:
The total number of people participated: 5

Beverage         Number of Votes
********************************
Coffee           3
Coke             1
Orange Juice     1
Tea              0
```

It was asked to print the beverage vote count in ascending order but later we said it was quite difficult at that stage to sort based on vote count and ignored this condition. Now re-do this task by creating two lists; one for the beverage names and other for the vote count. Zip them and then sort based on vote count and display the result.

Building Dictionary via Zip function:

`zip()` function is quite helpful in building a dictionary out of two iterables, considering first iterable elements as keys and that of the second iterable as values. It is shown here:

```
subj=['Mech','CP2','ES','MoM']
CH=[3,2,4,3]
subDic=dict(zip(subj,CH))
print(subDic)
```

Here **`subDic`** is a dictionary with subject names from **`subj`** list as **keys** and credit hours from **`CH`** list as **values**.

Zipping two Dictionaries:

We can also zip two dictionaries. An example is shown here:

```
student1={
    'Reg':'2018-MC-01',
    'Name':'Muhammad Usman',
    'Sec':'A',
    'Courses':['CP2','ES']}
student2={
    'Reg':'2018-MC-02',
    'Name':'Tahir Mehmood',
    'Sec':'A',
    'Courses':['ES','LA']}
z=zip(student1.items(),student2.items())
for (k1,v1),(k2,v2) in z:
    print(f"{k1} of first student is {v1}")
    print(f"{k2} of second student is {v2}")
```

In above example we have used `items()` method to get both **keys** and **values**. We can use simple dictionary name (returning keys by default) or use `values()` to get values as per requirement.

Tasks:

[2] GPA Calculation:

Letter Grade and **Grade Point** have following conversion rule at UET:

A+	A	A-	B+	B	B-	C+	C	C-	D+	D	F
4.0	4.0	3.7	3.3	3.0	2.7	2.3	2.0	1.7	1.3	1.0	0

GPA of any semester is calculated based on Credit Hours and obtained Grade Point (GP) for each subject and calculated as:

$$GPA = \frac{\sum(GP_i * CH_i)}{\sum CH_i}$$

Write a program that will do the followings:

- Make two lists; one for **Letter Grade** and other for **Grade Point**.
- Create a dictionary from above two lists with Letter Grades as Keys and Grade Point as Values.
- Create two lists; one for Credit Hours of 5 to 6 subjects and other for obtained Letter Grade in respective subjects.
- Create a user defined function that must take above created Dictionary and two lists (CH and Obtained Letter Grades) as input arguments and must return the GPA.
- Use the function on lists created in third step and verify the output.

[3] We have been considering an example of **Dry Fruit Seller** with a dictionary of its products having product name as **key** and price per KG as **value**. Let's have another dictionary named as `stock` having the same keys as first dictionary and values as on-shop available stock in KG of each item. Both dictionaries are shown here:

```
products={'anjeer':2500,'pista':2700,'badaam':2000,'chalghoza':8
000}
stock={'anjeer':100,'pista':240,'badaam':150,'chalghoza':80}
```

Write a user defined function named as `calcAsset()` that will take the two dictionaries as input and will calculate the total asset in PKR of whole stock available. Do it using the `zip()` function within user defined function.

Map:

`map()` is another useful built-in function which maps (applies) a function on a collection of objects in any iterable.

Let's see the help on map as:

```
help(map)
```

Starting few output lines are here:

```
Help on class map in module builtins:

class map(object)
 |  map(func, *iterables) --> map object
 |
 |  Make an iterator that computes the function using arguments from
 |  each of the iterables.  Stops when the shortest iterable is exhausted.
```

Observe the following part:

```
map(func, *iterables) --> map object
```

It says that the function `map()` has first input argument as name of the function which we want to apply on different objects and second input argument is **one or more** iterables on whose elements we want to apply that function. If the function specified takes one input argument, there should be one iterable, if it takes two input arguments, there should be two iterables and so on.

Let's consider one example:

Suppose we have a list of few positive numbers as:

```
numbers=[2,3,1,5,34,12,45]
```

And we want to generate a new list with square roots of elements of above list. With basic list processing, we can do it as:

```
sqrtNums=[]
for num in numbers:
    sqrtNums.append(math.sqrt(num))
print(sqrtNums)
```

With list comprehension it will be as:

```
sqrtNums=[math.sqrt(num) for num in numbers]
```

But another approach can be using the `map()` function as shown here:

```
sqrtNums=list(map(math.sqrt,numbers))
print(sqrtNums)
```

Note that the map object is converted to a list in above code. Because the map() function returns a map object and that is an iterator juts like the zip object and if we will print that directly, it will not show the elements inside it. See this:

```
sqrtNums=map(math.sqrt,numbers)
```

It has output like:

```
<map object at 0x000001B8EBEEFD90>
```

As mentioned earlier, we will discus in detail the difference between iterable and iterator in next lab session.

Now suppose we have two lists of numbers as:

```
nums1=[2,4,3,1,6]
nums2=[1,5,4,2,3]
```

And we want to have another list whose each element is sum of squares of corresponding elements of above lists. We can do it using **`map()`** function and providing the mapping function as a **Lambda Expression**:

```
sqSum=list(map(lambda a,b:a*a+b*b,nums1,nums2))
print(sqSum)
```

Because the mapping function specified as first input argument i.e. **Lambda Expression** takes two input argument, hence we need to provide two iterables after that. The good thing is that the two iterables provided can be of two different types e.g. one list and the other tuple as shown here:

```
nums1=[2,4,3,1,6] # A List
nums2=(1,5,4,2,3) # A Tuple
sqSum=list(map(lambda a,b: a*a+b*b,nums1,nums2))
print(sqSum)
```

Finally, if number of elements in two iterables for above example are not same, it will not generate the error, rather it will apply the function till the shortest number of elements in any iterable. Example is shown here:

```
nums1=[2,4,3,1,6]
nums2=(1,5,4)
sqSum=set(map(lambda a,b: a*a+b*b,nums1,nums2))
print(sqSum)
```

It will have the output:

```
{41, 5, 25}
```

You should not get confused with the order of above output since a **Set** is unorder collection and the assigned order by the Python cannot be determined.

Tasks:

[4] A list if XY points is given here:

```
points=[(2,1),(-3,0),(4,2),(2,-5)]
```

Create a list of magnitudes of above XY points using the **map** function.

Filter:

One of the most common operation on any data set is filtering the values based on some criteria. Python has a built-in method `filter()` for this purpose. Let's see the help of the function as:

```
help(filter)
```

Starting few output lines are here:

```
Help on class filter in module builtins:

class filter(object)
 |  filter(function or None, iterable) --> filter object
 |
 |  Return an iterator yielding those items of iterable for which function(item)
 |  is true. If function is None, return the items that are true.
```

So, it is a class and see carefully the following line:

```
filter(function or None, iterable) --> filter object
```

It says that filter function takes in two input arguments; first a function (filtering function) and second an iterable e.g. **List**, **Set** etc. and it returns a filter object.

Next it explains how it returns some values:

```
Return an iterator yielding those items of iterable for which function(item)
is true. If function is None, return the items that are true.
```

Which means that the filtering function provided as first input argument will be evaluated for each item and if function returned `True`, that element will be yielded i.e. selected and others with `False` will be rejected.

Let's see it with example:

Suppose we have a list of numbers as:

```
numbers=[2,3,1,5,34,12,45]
```

And we want to select only the even numbers. For that we can first create a function that returns `True` for even numbers and `False` otherwise and then use it with `filter()` as shown here:

```
def isEven(n):
    return n%2==0
numbers=[2,3,1,5,34,12,45]
evens=list(filter(isEven,numbers))
print(evens)
```

It will have the output:

```
[2, 34, 12]
```

Carefully note the line:

```
evens=list(filter(isEven,numbers))
```

We are using **list()** on filter object. Basically, when we apply filter method on any iterable, a filter object is returned which is an iterator just like map and zip object, and we can convert that to **List** or **Tuple** or **Set** as per requirement. If we try to print the filter object as:

```
evens=filter(isEven,numbers)
print(evens)
```

It will display that **evens** is a filter object and will show the address of memory location where it is stored. You might get a different memory location:

```
<filter object at 0x0000017770DBDD88>
```

The other very important point is that we should prefer to use **Lambda Expression** here because the function we created is a simple single expression function. The code with Lambda Expression is shown here:

```
numbers=[2,3,1,5,34,12,45]
evens=list(filter(lambda x: x%2==0,numbers))
print(evens)
```

Just to have an idea that it can be any iterable, here is the code with different iterables i.e. **Tuple** and filter object is converted to **Set**:

```
numbers=(2,3,1,5,34,12,45)
evens=set(filter(lambda x: x%2==0,numbers))
print(evens)
```

Tasks:

[5] We created students as dictionary with few attributes as key-value pairs. Following the same pattern, a list of few students is given here:

```
student1={
    'Reg':'2018-MC-01',
    'Name':'Muhammad Usman',
    'Sec':'A',
    'Courses':['CP2','ES']}
student2={
    'Reg':'2018-MC-02',
    'Name':'Tahir Mehmood',
    'Sec':'A',
    'Courses':['ES','LA']}
student3={
    'Reg':'2018-MC-51',
    'Name':'Danish',
    'Sec':'B',
    'Courses':['MoM','CP2']}
student4={
    'Reg':'2018-MC-52',
    'Name':'Hafiz Muhammad Aqib Ali',
    'Sec':'B',
    'Courses':['LA','ES','MoM']}
allStudents=[student1,student2,student3,student4]
```

Use **`filter()`** function to generate and display the list of students who have registered **`CP2`** course. Use **Lambda Expression**.

[6] Define a list with few numbers e.g.:

```
nums=[5,4,2,1,7,8]
```

Do the followings on above list:

- Find and display a list of factorials of numbers in list. **(Use `factorial` function available in `math` module)**
- Find a list of factorials of Even numbers only in the list. Do it in two ways. In first case apply filter to get even numbers (no need to convert it to list; keep it as **`filter`** object) and then **`map`** the **`factorial`** function on this **`filter`** object and convert to list. In second case, write a single Python statement to apply both **`filter`** and **`map`**.
- Find a tuple of cubes of Odd numbers only in the list. Do this in single Python statement.

[7] Continuing with the previous task of student dictionary, here is a function for displaying the student in a customized format:

```
def dispStudent(st):
    print('_____________________________________')
    print(f"Registration No. : {st['Reg']}")
    print(f"Name : {st['Name']}")
    print(f"Section : {st['Sec']}")
    print(f"Courses Registered : {st['Courses']}")
```

Modify the previous task in a way that students who have registered **`CP2`** are displayed via above function. Use **`map()`** and **`filter()`** together.

Filter function to remove missing values:

Many times, when we get data from some source, it is very common that it contains some missing values like empty list, blank string or 0 etc. It is needed to get rid of those values before any further processing of the data. We can use the **`filter()`** function for this. See the help on filter once again and just observe this part of the output:

```
filter(function or None, iterable) --> filter object
```

See that as filtering function we can provide None and, in that case, it will filter out all empty elements or the elements which by themselves are False.
See this example:

```
a=[1,5,[],False,[4,5],(),0,'Python']
print(list(filter(None,a)))
```

The output is:

```
[1, 5, [4, 5], 'Python']
```

See that the empty list, empty tuple, False and 0 have been removed, because these are **Boolean False** in Python. You can check if some object is Boolean True or False using the `bool()` function e.g. `bool([])` will return False and if there is even a single element inside the list, it will return True. Followings are considered as False in Python and will be removed by the filter function with None provided as filtering function:

0	→	Integer 0
0.0	→	Float 0
complex(0,0)	→	Complex Number 0
False	→	Boolean False
None	→	None
[] or list()	→	Empty List
{} or dict()	→	Empty Dictionary
() or tuple()	→	Empty Tuple
set()	→	Empty Set
""	→	Blank String

In some case 0 is not a missing value but a meaningful value. In that case, instead of `None` we can use this function as the filtering function:

```
lambda i:True if i == 0 else i
```

It is simple to understand that we are explicitly declaring 0 as True and all other elements as the same elements. If you will apply the above filtering function, you will see 0 will not be removed. But False will also not be removed because 0 and False are treated as equal. However, if you want to just keep the 0 and not the False then instead of equality check you can use the `is` operator which returns True when two objects are the same objects in memory. 0 and False are treated equal but are not the same objects in memory. So use this filtering function to remove all missing values except 0:

```
lambda i:True if i is 0 else i
```

Reduce:

The `reduce()` function is available in `functools` module. It takes in a function and a sequence (or collection of objects) and returns a single value calculated as follows:

- Initially, the function is applied on the first two items from the sequence and the result is returned.
- The function is then called again with the result obtained in step 1 and the next value in the sequence.
- This process keeps repeating until the last item in the sequence is processed.

The simplest example to understand working of `reduce()` function is to find sum of numbers in any iterable although we have built-in `sum()` function in most of the iterables. We will consider this example for the sake of understanding.

Suppose we have following numbers in a **Tuple**:

```
nums=(2,3,1,4,5)
```

We can define a function to add two values (normal function or Lambda Expression) and simply specify that in `reduce()` function as shown here:

```
from functools import reduce
```

```
nums=(2,3,1,4,5)
s=reduce(lambda a,b: a+b,nums)
print(s)
```

Here is explained how interpreter executes **`reduce()`** function in above code:

- Interpreter applies Lambda Expression on first two elements of the tuple i.e. **`2`** and **`3`** which results into **`5`**.
- Then it executes the same function on previous result i.e. **`5`** and the next number in the tuple i.e. **`1`** which results into **`6`**.
- The process keeps repeating till the last number evaluating to **`15`**.

Here is another example where **`reduce()`** function is used to find the minimum from an iterables although again the **`min`** and **`max`** functions are generally available, but the purpose is to understand the **`reduce()`** function. The logic to find minimum from an iterable is to find lesser of first two, then lesser of previous less and the next number and so on. The code is shown here:

```
from functools import reduce
nums=(2,3,1,4,5)
s=reduce(lambda a,b: a if a<b else b,nums)
print(s)
```

Finally, let's see an example which cannot be solved directly by the built-in function like **`sum`** and **`min`**. Suppose there is a list of numbers as:

```
x=[3,2,5,1]
```

And we want to find the sum of factorial of all number inside list i.e. **`3!+2!+5!+1!`**. We can define a Lambda Expression for factorial sum and use it with **`reduce()`** function as shown:

```
from math import factorial as f
from functools import reduce
x=[3,2,5,1]
y=reduce(lambda a,b: f(a)+f(b),x)
print(y)
```

If we run the above code, we will see a really big number which cannot be the answer. Where is the problem?

You can debug above code in step by step execution method and you will find that:

- First time Lambda Expression was evaluated on first two entries i.e. **`3`** and **`2`** as **`3!+2!`** and resulted in **`8`**.
- Next the function was evaluated on previous result i.e. **`8`** and the next list element i.e. **`5`** as **`8!+5!`**. But we don't want that. We want **`8+5!`**.

So maybe it will be better to have Lambda Expression as: **`lambda a,b: a+f(b)`** where factorial of first value is not taken. But now it will create problem for the first step where Lambda Expression will be evaluated on first two element as **`3+2!`** but we want **`3!+2!`** in first step. Here, we can get help from third optional input argument in **`reduce()`** function which evaluates the Lambda Expression in first step on this third input and first element of the iterable rather than first and second element. The correct code is shown here:

```
from math import factorial as f
from functools import reduce
x=[3,2,5,1]
y=reduce(lambda a,b: a+f(b),x,0)
print(y)
```

Tasks:

[8] Do the above example of sum of factorial, using `map()` function by first getting a list of factorials of numbers and then using the `sum()` function on that list.

[9] Use the list of students as created in previous tasks and use `reduce()` function to generate a **Set** of subjects registered by students. Here the Lambda Expression will have two inputs; first set and the second student and third input argument of `reduce()` will be an empty set.