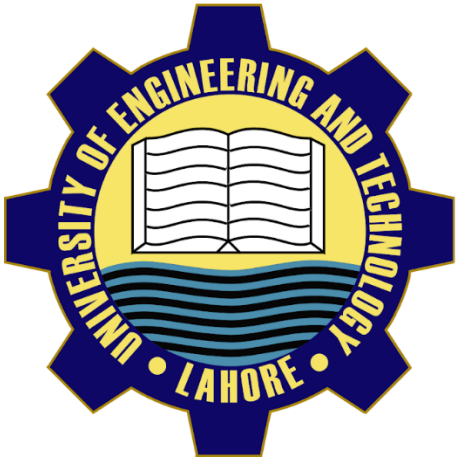
MCT-243 : COMPUTER PROGRAMMING-II

using Python 3.9



Prepared By:

Mr. Muhammad Ahsan Naeem

YouTube Playlist

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

Lab 31: GENERATORS

If you recall the comprehension techniques we learnt previously, you should know that there is no **Tuple Comprehension** and if we try to code a Tuple Comprehension, it doesn't throw an error, rather a different type of object is returned known as **Generator**. Verify this from the code given here:

```
nums=(5,3,4,6,8)
sqNums=(i*i for i in nums)
print(sqNums)
```

It will have the output like:

```
<generator object <genexpr> at 0x000002A0D6ACB348>
```

Generator object is also an Iterator, and we can directly apply for loop. We will also see the difference between iterator and iterable in this session. So, if we want to print all elements in **sqNums** generator from above code we can simply write:

```
for i in sqNums:
    print(i)
```

Which will have the following output:

```
25
9
16
36
64
```

What is special about Generator?

Before exploring the generator syntax in detail, let's see what extra benefits a generator provides over the other iterables and where we should prefer to use it. The most important benefit of a generator is that it is much more memory efficient than any other iterable as it doesn't store its content in memory. In above example if it was a list as here:

```
nums=(5,3,4,6,8)
sqNums=[i*i for i in nums]
```

Then the list **sqNums** stores all squared numbers in memory and we can retrieve them later (e.g. in simple **print()** statement). On the other hand, if it is a generator as in the code we started with, the squared numbers are not calculated or saved in memory, rather the generator calculates them one at a time when required and discards it after that, leaving behind no memory reserved to save that value. Just to further explain the idea, consider once again the same code of generator as:

```
nums=(5,3,4,6,8)
sqNums=(i*i for i in nums)
```

The generator **sqNums** just holds the rule i.e. squaring the number and the reference to inputs i.e. the tuple **nums** and no squared number is calculated or stored. When we use it as:

```
for i in sqNums:
    print(i)
```

In each `for` loop iteration, the generator `sqNums` generates a new number by picking the element from Tuple and applying the rule. The generated number is used (print in above case) and discarded. In next iteration the same procedure is repeated for the next tuple numbers till the last is processed.

So when to prefer a generator?

For the cases similar to the above one where we need the sum but we don't need the individual numbers, we should always use generator.

To further clear the idea of the generator element being created, used and discarded; lets see the following code quite similar to the above one but after printing all elements of generator object, we are using the generator object once again to find and print the sum of numbers inside it as:

```
nums=(5,3,4,6,8)
sqNums=(i*i for i in nums)
print("Generator elements are printed here:")
for i in sqNums:
    print(i)
print(f"Sum of Generator elements:{sum(sqNums)}")
```

It will have the following output:

```
25
9
16
36
64
Sum of Generator elemnts:0
```

Why the sum is displayed as 0 in above output?

There is nothing wrong in the `sum()` function but when we used generator first time in printing, all its elements were produced, leaving behind nothing and hence when used second time in `sum()`, it calculated `0`. Hence one very important point of generators is that the generators are one time usable! Therefore, generators are known as **Lazy Single-use Iterable**.

Comparison of Memory:

As one last demo before starting the details of coding a generator, you can visualize the memory efficiency a generator provides over the other iterables. See this code where squares of **10 Million** integers is generated using a List and a Generator.

```
from sys import getsizeof
sqNumbers=[n*n for n in range(1,10_000_000)]
print(f'Size of List: {getsizeof(sqNumbers)} Bytes!')
sqNumbers=(n*n for n in range(1,10_000_000))
print(f'Size of Generator: {getsizeof(sqNumbers)} Bytes!')
```

This is the output:

```
Size of List: 89095160 Bytes!
Size of Generator: 112 Bytes!
```

You can see the huge difference between the memory consumed by a list and by the Generator and that is the biggest reason one should use generator when dealing with large data. You can also try on 100 Million or even one Billion integers (at you risk, because your system might get stuck!), you will see the size of the generator will be same i.e., 112 Bytes because no square is calculated by the generator while all squares are calculated and stored by the list.

A few points you should remember in generators are:

- Generators are Iterators. (Iterators are also Iterables.)
- A generator doesn't know how many elements are inside it. We cannot use `len()` function on generator.
- Generators are single use iterable. Once used, we cannot get the elements again. Once a generator is completely used, we call that it is **exhausted**.
- Because generators are single used datatype, so generally it is a better approach to use them immediately after they are created.

There are two ways we can create generators; **Generator Expression** and **Generator Function**. But before we go to the detail, we will first discus the difference between Iterable and Iterator.

Difference Between Iterable and Iterator:

This is really important to know the difference between Iterable and Iterator. We define an Iterable as a datatype that could be directly iterated over, using a `for` loop. The examples include List, Tuple, Set, Dictionary etc. On the other hand, the Iterators can also be iterated over, directly with a `for` loop, but only once. Once an iterator has produced all its elements (in a `for` loop or in some other function like `sum`) it will be exhausted and will become empty.

There is no class directly with the name Iterable or Iterator. The objects of some class e.g., List, can be Iterable. So a List object is an object of List class but is an Iterable. Similarly, a generator is the object of Generator class and is an iterator.

Let's see different methods or functions available in some iterable and iterator.

```
l=[n for n in range(10)]
g=(n for n in range(10))
print('List(Iterable) class methods:')
print(dir(l))
print('------------------------')
print('Generator(Iterator) class methods:')
print(dir(g))
```

The output is:

```
List(Iterable) class methods:
```

```
['__add__', '__class__', '__class_getitem__', '__contains__',
'__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__',
'__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
------------------------
Generator(Iterator) class methods:
['__class__', '__del__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__',
'__lt__', '__name__', '__ne__', '__new__', '__next__',
'__qualname__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'close',
'gi_code', 'gi_frame', 'gi_running', 'gi_yieldfrom', 'send', 'throw']
```

There are two important things to notice here:

1. The Generator (which is Iterator), has the method/function **`__next__()`** while the List (which is an Iterator) doesn't has this method.
2. Both Generator and List (meaning both Iterator and Iterable) have the method **`__iter__()`**.

The methods surrounded in between double underscores are special methods (also known as **Dunder Methods**). These are usually for the internal use, but we can also use these if needed.

The object of a class will be iterator if the class follows the **Iterator Protocol**. Any class having the definition of the two methods **`__next__()`** and **`__iter__()`** is a class following the **Iterator Protocol**.

Let's see what is **`__next__()`** method in any Iterator. We can apply the method directly on some iterator simply as **`next()`** without using the double underscores. This is the method which produces/calculates the next element inside an iterator.

```
nums=[3,1,8,4]
sqNums=(i*i for i in nums)
print(next(sqNums))
```

The output will be **`9`** which is first element produced by the iterator. If we will use the method **`next()`** second time, second element will be produced and so on for the next elements. Above iterator can produce four elements, so if we use the method **`next()`** four times, you will see it will produce four elements one by one.

```
nums=[3,1,8,4]
sqNums=(i*i for i in nums)
print('First Time:')
```

```
print(next(sqNums))
print('Second Time:')
print(next(sqNums))
print('Third Time:')
print(next(sqNums))
print('Fourth Time:')
print(next(sqNums))
```

The output will be:

```
First Time:
9
Second Time:
1
Third Time:
64
Fourth Time:
16
```

Try to use `next()` fifth time and you will see the `StopIteration` exception. Because the generator can produce only the four elements. After the four elements had been produced by the generator, the generator is empty and if we will try to produce the next element, it generates error.

How `for` loop or some other function like `sum` produces elements from an iterator?

When we apply the `for` loop or some function that needs all elements of the iterator like `sum`, `min`, `max` etc., on some iterator, that function uses the method `next()` internally and produces the elements of the iterator to use. In case of `for` loop, the method `next()` is used in each iteration of the loop, the next element is produced and the loop variable get that value.

How `for` loop works on some Iterable?

In case of Iterator, method `next()` is used to produce each element of the iterator but we don't have this method `next()` in iterable. So, in case of an iterable, a `for` loop follows these steps:

i. There is a method `iter()` in iterable which creates an iterator from an iterable. The `for` loop will apply this method on the iterable to create an iterator.
ii. Then will start the loop iterations and the method `next()` will be applied on the iterator created in previous step.
iii. The loop iterations will end when `StopIteration` exception is generated by the `next()` method.

So, the method `next()` is being applied on iterator and not on the iterable even in case we are applying it on iterable. And you can easily understand why we can use the iterable as many times as we wish, because it creates a new iterator each time.

We can also use the `iter()` method directly on some iterable to get an iterator.

```
nums=[3,1,8,4]
x=iter(nums)
print(type(x))
```

The output will be:

```
<class 'list_iterator'>
```

All detail we studied for Iterable and Iterator is of course applicable to all possibilities of Iterable and Iterator. The `zip`, `map` and `filter` functions we studied in one previous session, creates an iterator and those are also one time usable. Moreover the `enumerate()` function which we have not studied yet and the file read/write object, we studied for text files and CSV files, also create an iterator.

Now we will see the detail of **Generator Expression** and **Generator Function**:

Generator Expression:

The syntax we have seen in starting example of generator is **Generator Expression**. The syntax is very similar to Comprehension Technique. We can always write these Generator Expression directly as input argument as in any other case. For example:

```
nums=(5,3,4,6,8)
result=sum((i*i for i in nums))
print(result)
```

If you see on second line, there are double starting and closing braces; one for the `sum()` function and the others for the Generator Expression. Python allows here to use only one and still it will be a Generator Expression as:

```
result=sum(i*i for i in nums)
```

If we were to use a List Comprehension as:

```
result=sum([i*i for i in nums])
```

Of course, we cannot eliminate square bracket in this case.

Once we create a generator, generally the way we use it are:

- Use it directly in `for` loop where each element will be generated in each iteration.
- Apply any function on it that needs iterable as input argument e.g. `sum()`

We can also access the first to last elements using `next()` function. See this code:

```
nums=(5,3,4,6,8)
sqNums=(i*i for i in nums)
print(next(sqNums))
print(next(sqNums))
print(next(sqNums))
```

It will display first three elements inside the generator as:

```
25
9
16
```

After using the `next()` three times, if we apply `sum()` function on generator as:

```
nums=(5,3,4,6,8)
sqNums=(i*i for i in nums)
x=next(sqNums)
y=next(sqNums)
z=next(sqNums)
print(sum(sqNums))
```

We will get `100` at the output. After 3 `next()` function calls, those three elements are no more in the generator and hence it will sum the remaining elements which will result into `100` as `62+82=100`

Guess the output for the following code and verify by running it.

```
numbers = [1, 2, 3, 4, 5]
squares = (n**2 for n in numbers)
print(9 in squares)
print(9 in squares)
```

Casting generator to other iterables:

Yes, we can cast a generator to other iterable but normally this is not something we use generators for. We use generators in place of any other iterable to save memory and if generator is converted to other iterable, that very good reason is no more there. Anyhow if it is needed for any reason, we can do it as:

```
nums=(5,3,4,6,8)
sqNums=tuple(i*i for i in nums)
print(sqNums)
```

Tasks:

[1] Task is to find the sum of all numbers in a nested list. You have to create three versions of the function that will do this. Name them as `sumAllv1()`, `sumAllv2()` and `sumAllv3()`.

- In v1, do it by using nested `for` loop without any comprehension.
- In v2, do it by creating a flattened list through **List Comprehension** and use `sum()` function on that.
- In v3, do it by creating a generator object instead of flattened list and use `sum()` function on that.

[2] Create a function named as `deepAdd()` that can take any number of iterables and will return the sum of all numbers in all those iterables. No nested iterable is allowed to pass as input argument. Example use of the function is shown here:

```
col1=[3,2,6,1]
col2=(5,4)
col3={1,10,20}
col4={'A':200,'B':100}
print(deepAdd(col1,col2,col3,col4.values())) # This prints 352
```

You have to do it by creating a generator through **Generator Expression** inside function.

Generator Functions:

The second way to create generator object is doing it through a function known as **Generator Function** which are quite different from the regular functions that we have been creating and using so far. **Generator Functions** uses a special statement known as `yield` for this purpose and they behave quite differently. Let's see this code:

```
def count(n):
    while True:
        yield n
        n+=1
### Main Program ###
a=count(0)
print(a)
```

This function returns a generator. When used in first line of the main program it created a generator object which is displayed as:

```
<generator object count at 0x00000143D5EFBA48>
```

Now if we use `next()` function on it, it will generate the next values as shown here:

```
print(next(a)) # This prints 0
print(next(a)) # This prints 1
print(next(a)) # This prints 2
```

If you run above code in step by step mode, you will see that when `next(a)` is executed for the first time, the interpreter enters into the function body; enters into the `while` loop and `yield n` will make it come back to the main program where it was called. But when `next(a)` is executed second time the interpreter jumps to the line `n+=1` which is the very next line after `yield n` line. If you observe carefully, you will see that `yield n` makes interpreter jump to the location where it was called just like `return` statement but when it is called again (with `next()` function), it starts from the point where it left off in first call. Therefore, **Generator Functions** are also known as **Pause-able Functions**.

See this regular function:

```
def test():
    if (1==2):
        return 1

### Main Program ###
a=test()
print(a)
print(type(a))
```

The condition inside `if` statement is `False` and hence the function will not return anything and this is what will be displayed:

```
None
<class 'NoneType'>
```

Now let's do the similar thing on Generator Function as:

```
def test():
    if (1==2):
        yield 1

### Main Program ###
a=test()
print(a)
print(type(a))
```

The output will be:

```
<generator object test at 0x0000013B8D88B348>
<class 'generator'>
```

Surprisingly, even if the condition is **`False`** and **`yield`** statement will never get executed but still the function returned a generator. So, if there is a **`yield`** statement in the function whether or not it gets executed, it is not a normal function; it is a Generator Function.

Do not use **`return`** and **`yield`** both in one function. It will be a Generator but we will not be able to use **`next()`** method. Using **`return`** and **`yield`** both in any function will be treated as error in coming Python versions. So, simply just do not use them together.

Note→There can be more than one yield statements in one Generator Function.

Now check this code and understand the execution sequence. Better will be to run it in step by step mode:

```
def count(n):
    print('A')
    while n<3:
        yield n
        n+=1
        print('B')
    print('C')

### Main Program ###
a=count(0)
print(a)  #Prints generator object
x=next(a) #Print 'A' and x gets 0
print(x)  #Prints 0
x=next(a) #Prints 'B' and x gets 1
print(x)  #Prints 1
x=next(a) #Prints 'B' and x gets 2
```

```
print(x)  #Prints 2
x=next(a) #Prints 'B' and prints 'C' and generates an error then
print(x)
```

Here is another program to understand the execution flow:

```
def count(n):
    while n<10:
        yield n
        n+=1

### Main Program ###
a=count(0)
x=next(a) #x gets 0
x=next(a) #x gets 1
y=list(a) #will be a list of remaining all elements
print(y)
z=list(a) #will be an empty list as the generator is exhausted.
print(z)
```

The output is:

```
[2, 3, 4, 5, 6, 7, 8, 9]
[]
```

Now let's consider the same example of creating a generator of squares of numbers in a list by creating a **Generator Function** and use it to compute the sum of squares.

```
def getSquare(aList):
    for i in aList:
        yield i*i
### Main
nums=[2,3,1,7]
a=getSquare(nums)
print(a)
print(sum(a))
```

```
<generator object getSquare at 0x000001BBEAA49DC8>
63
```

Now see another version of the same program here:

```
def getSquare(aList):
    return (i*i for i in aList)
### Main
nums=[2,3,1,7]
a=getSquare(nums)
print(a)
print(sum(a))
```

It will have the output as:

```
<generator object getSquare at 0x000001BBEAA49DC8>
63
```

Although it is a simple function and not the generator function since it has `return` and not the `yield`. But the thing it is returning is a **Generator Expression**. We will not call this function as Generator Function because this function is not used to produce the elements as in case of Generator Function. Both codes are given here for easy comparison:

Generator Function with yield statement	Simple Function with Generator Expression as return value.

`def getSquare(aList):`
`  for i in aList:`
`  yield i*i`
	`def getSquare(aList):`
	`  return (i*i for i in aList)`

Tasks:

[3] Create a Function named as `interleave()` that will take two iterables as input (for simplicity you can assume that both will have same number of elements). The function should give a generator object with each of items in both iterables interleaved (first item from first iterable, then first item from second iterable, then second item from first iterable, then second item from second iterable and so on). Use to versions of the function:

- **a.** Using `yield` inside the function.
- **b.** Using return and returning a Generator Expression

To verify the working of the function we can use it on two iterables and convert the obtained generator into a list to see if the sequence is generated correctly. It was explained earlier that generator objects are not meant to be converted to other iterable as we lose the actual advantage of creating a generator. But for the sake of verification, we have to do this here. Once verified, one can use it on two iterables of many thousands of elements without converting to list and saving a huge amount of memory.

```
### Main Program ###
x=(5,3,"Hello")
y=[2,9,20]
g=interleave(x,y)
print(g) #To verify that g is a generator
print(list(g)) #To verify if values interleaved correctly
#Must display--> [5, 2, 3, 9, 'Hello', 20]
```

[4] Create a function named as `makePairs()` that will take one input iterable as input argument (having even numbers of elements) and will give a generator object that will have two-valued tuples as first and second element of iterable in first tuple, third and fourth element in second tuple and so on. Again, make two versions of this function:

- **a.** Using `yield` inside the function.
- **b.** Using return and returning a Generator Expression

In both versions access the iterable items through index as we need two items in one iteration and that will be easy b using the index. And once again we need to verify the working of function by converting the generator to list as shown here:

```
### Main Program ###
x=[2,9,20,44]
g=makePairs(x)
print(g) #To verif that g is a generator
print(list(g)) #To verify if pairs created correctly
#Must display-->[(2, 9), (20, 44)]
```

[5] We want to find and display first 100 **Fibonacci Numbers**. Create a **Generator Function** to generate those Fibonacci Numbers and use that in Main Program to display those.