In [75]:

```
##  task -18 ## advance functions

'''A lambda function is a small anonymous function.

A lambda function can take any number of arguments,
 but can only have one expression.

lambda arguments : expression'''

f=lambda a:a+a
print(f(5))
```

```
10
```

In [78]:

```
f=lambda a,b,c:a*b*c
print(f(2,4,6))
```

```
48
```

```
## filters().
'''
The filter() method filters the given sequence with the help
of a function that tests each element in the sequence
to be true or not.'''

'''syntax:

filter(function, sequence)
Parameters:
function: function that tests if each element of a
sequence true or not.
sequence: sequence which needs to be filtered, it can
be sets, lists, tuples, or containers of any iterators.
Returns:
returns an iterator that is already filtered.

'''
## example##

ages=[20,15,52,44,47,18,8,16]

def func(x):
    if x <18:
        return False
    else:
        return True

adults = filter(func,ages)
print('only >=18:',list(adults))
```

```
only >=18: [20, 52, 44, 47, 18]
```

In [26]:

```python
'''maps:
# The python map() function is used to return a list of results
# after applying a given function to each item of an iterable(list, tuple etc.)
# map(function, iterables)  '''

def calculation_of_multiplication(n):
    return n*n
num=(3,5,7,9,11,13,14)
answer=map(calculation_of_multiplication,num)
print('multiplication:',tuple(answer))

print('------------')

def calculation_of_addition(n):
    return n+n
num=(3,5,7,9,11,13,14)
answer=map(calculation_of_addition,num)
print('addition:',list(answer))
```

```
multiplication: (9, 25, 49, 81, 121, 169, 196)
------------
addition: [6, 10, 14, 18, 22, 26, 28]
```

[5]:

```
from functools import reduce
'''
The reduce() function, as the name describes,
applies a given function to the iterables and returns a single value.

Working conditions:

The first two elements of the sequence are chosen in the first step, and the result is achieved.
The result is then saved after applying the same function to the previously obtained result and the number preceding of
the second element.
This method is repeated until there are no more elements in the container.
The final result is returned to the console and printed.
'''
func=reduce(lambda a,b:a+b,[10,20,30,40,50,60])
print('addition:',func)
print('---------------')
func=reduce(lambda a,b:a*b,[10,20,30,40,50,60])
print('multiply:',func)
print('---------------')
func=reduce(lambda a,b:a-b,[10,20,30,40,50,60])
print('subtraction:',func)
```

```
addition: 210
---------------
multiply: 720000000
---------------
subtraction: -190
```

[21]:

```
def add(*a):
    return a+a
add(10,20,30,40,50,60)
```

[21]:

```
(10, 20, 30, 40, 50, 60, 10, 20, 30, 40, 50, 60)
```

In [86]:

```
# generator functions - {use only 'yield' keyword}
'''
Generator-Function ::
...A generator-function is defined like a normal function,
...but whenever it needs to generate a value,
...it does so with the yield keyword rather than return.
...If the body of a def contains yield, the function automatically
becomes a generator function.
'''
'''
1)The yield statement is responsible for controlling the flow of the generator function.
2)It pauses the function execution by saving all states and yielded to the caller.
3)Later it resumes execution when a successive function is called. We can use the multiple yield statement in the
generator function.

4)The return statement returns a value and terminates the whole function and only one return statement can be used
in the function.
'''

def GeneratorFunc():
    yield 1
    yield 2
    yield 3
    yield 4
                # # x is a generator object
g= GeneratorFunc()

              # #Iterating over the generator object using next

print(g.__next__()) # In Python 3, __next__()
print(g.__next__())
print(g.__next__())
print(g.__next__())
```