# *Functional Programming in Python*

*Functional Programming is a popular programming paradigm closely linked to computer science's mathematical foundations. While there is no strict definition of what constitutes a functional language, we consider them to be languages that use functions to transform data.*

*Python is not a functional programming language but it does incorporate some of its concepts alongside other programming paradigms. With Python, it's easy to write code in a functional style, which may provide the best solution for the task at hand.*

## *Functional Programming Concepts :*

*Functional programming is a programming paradigm in which we try to bind everything in pure mathematical functions style. It is a declarative type of programming style. Its main focus is on "**what to solve**" in contrast to an imperative style where the main focus is "**how to solve**". It uses expressions instead of statements. An expression is evaluated to produce a value whereas a statement is executed to assign variables.*

> *"Object-oriented programming makes code understandable by encapsulating moving parts. Functional programming makes code understandable by minimizing moving parts." : Michael Feathers*

*Some of Python's features were influenced by Haskell, a purely functional programming language. To get a better appreciation of what a functional language is, let's look at features in Haskell that can be seen as desirable, functional traits:*

- ***Pure Functions** - do not have side effects, that is, they do not change the state of the program. Given the same input, a pure function will always produce the same output.*
- ***Immutability** - data cannot be changed after it is created. Take for example creating a List with 3 items and storing it in a variable my_list. If my_list is immutable, you wouldn't be able to change the individual items. You would have to set my_list to a new List if you'd like to use different values.*

- ***Recursion:*** *There are no "for" or "while" loop in functional languages. Iteration in functional languages is implemented through recursion.*
- ***Higher Order Functions*** *- functions can accept other functions as parameters and functions can return new functions as output. This allows us to abstract over actions, giving us flexibility in our code's behavior.*
- ***Laziness*** *is another property of FP wherein we don't compute things that we don't have to. Work is only done on demand. Haskell has also influenced* ***iterators and generators*** *in Python through its lazy loading, but that feature isn't necessary for a functional language.*

*Python too supports Functional Programming paradigms without the support of any special features or libraries.*

## *Pure Functions*

*As Discussed above, pure functions have two properties.*

- *It always produces the same output for the same arguments. For example, 3+7 will always be 10 no matter what.*
- *It does not change or modifies the input variable.*

*The second property is also known as immutability. The only result of the Pure Function is the value it returns. They are deterministic. Programs done using functional programming are easy to debug because pure functions have no side effects or hidden I/O. Pure functions also make it easier to write parallel/concurrent applications. When the code is written in this style, a smart compiler can do many things – it can parallelize the instructions, wait to evaluate results when need them, and memorize the results since the results never change as long as the input doesn't change.*

```
# Python program to demonstrate pure functions

# A pure function that does Not changes the input list and returns the new List

def   pure_func(List):

    New_List = []

    for i in List:

        New_List.append(i**2)

    return New_List

Original_List = [1, 2, 3, 4]

Modified_List = pure_func(Original_List)

print("Original List:", Original_List)

print("Modified List:", Modified_List)
```

**Output:**

```
Original List: [1, 2, 3, 4]

Modified List: [1, 4, 9, 16]
```

## Recursion Function

### What is recursion ?

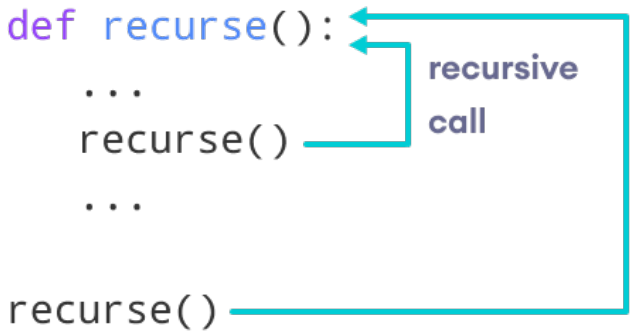
Recursion is the process of defining something in terms of itself.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

## *Python Recursive Function*

*In Python, we know that a function can call other functions. It is even possible for the function to call itself. A recursive function is a function that calls itself and repeats its behavior until some condition is met to return a result.*

*The following image shows the working of a recursive function called recurse .*




*Recursive Function in Python*

*Following is an example of a recursive function to find the **factorial of an integer**.*

*Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.*

***Example of a recursive function :***

```
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))
num = 3
print("The factorial of", num, "is", factorial(num))
```

***Output : The factorial of 3 is 6***

*In the above example, factorial() is a recursive function as it calls itself.*

*When we call this function with a positive integer, it will recursively call itself by decreasing the number.*

*Each function multiplies the number with the factorial of the number below it until it is equal to one. This recursive call can be explained in the following steps.*

*factorial(3)* *# 1st call with 3*

*3 * factorial(2)* *# 2nd call with 2*

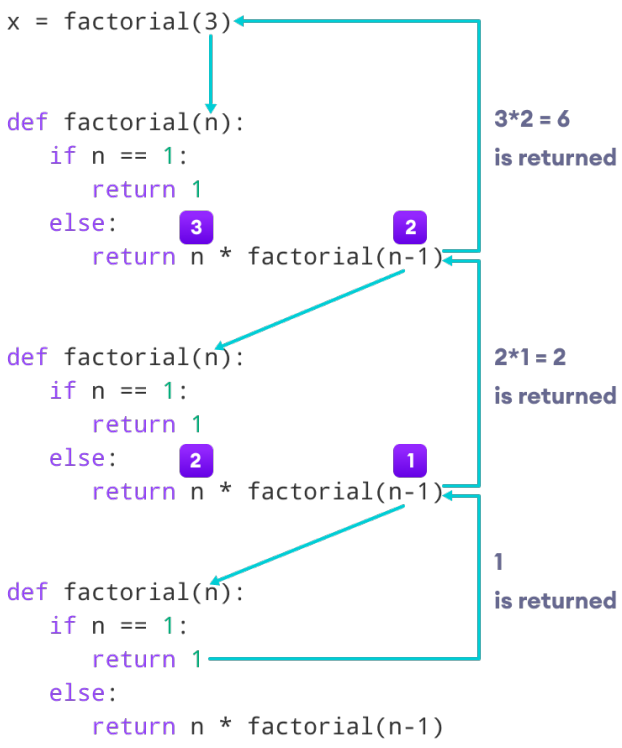
*3 * 2 * factorial(1)* *# 3rd call with 1*

*3 * 2 * 1* *# return from 3rd call as number=1*

*3 * 2* *# return from 2nd call*

*6* *# return from 1st call*

*Let's look at an image that shows a step-by-step process of what is going on:*



*Our recursion ends when the number reduces to 1. This is called the* ***base condition****.*

*Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.*

*By default, the maximum depth of recursion is 1000. If the limit is crossed, it results in Recursion Error. Let's look at one such condition.*

```
def recursor():
    recursor()
recursor()
```

***Output :***

```
Traceback (most recent call last):
  File "<string>", line 3, in <module>
  File "<string>", line 2, in a
  [Previous line repeated 996 more times]
RecursionError : maximum recursion depth exceeded
```

### *Advantages of Recursion :*

1. *Recursive functions make the code look clean and elegant.*
2. *A complex task can be broken down into simpler sub-problems using recursion.*
3. *Sequence generation is easier with recursion than using some nested iteration.*

### *Disadvantages of Recursion :*

1. *Sometimes the logic behind recursion is hard to follow through.*
2. *Recursive calls are expensive (inefficient) as they take up a lot of memory and time.*
3. *Recursive functions are hard to debug.*

## Built-in Higher-order functions :

*To make the processing iterable objects like lists and iterator much easier, Python has implemented some commonly used Higher-Order Functions. These functions return an iterator which is space-efficient.*

*Python provides a few higher order (or functional programming) functions in the standard library that can be quite useful:*

- *lambda*
- *map*
- *filter*
- *reduce*
- *list comprehensions*

## *Python Lambda Function*

*Lambda is one of the most useful, important and interesting features in Python. Unfortunately, they are easy to misunderstand and get wrong.*

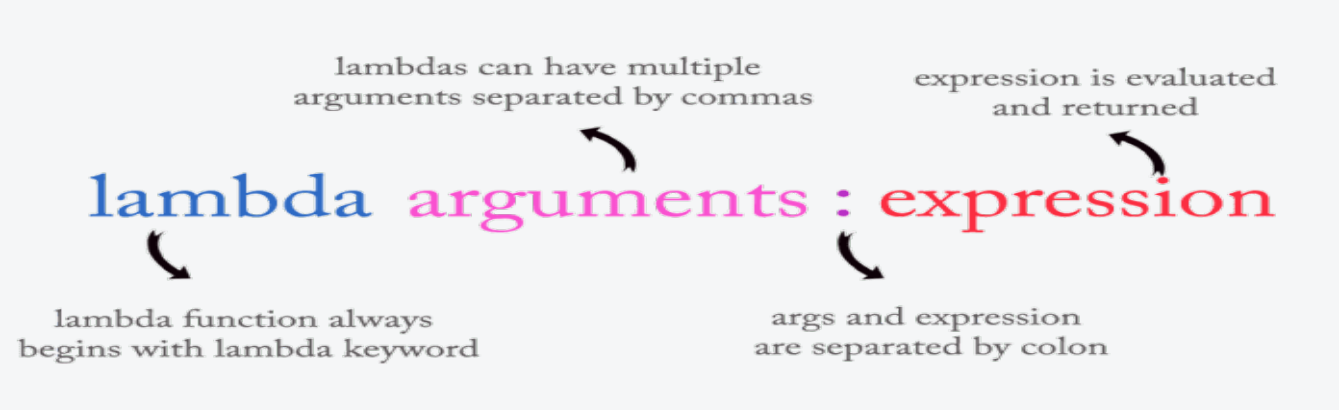
### *What is a Lambda Function ?*

*Lambda functions are **anonymous** or **nameless**. Unlike traditional functions, lambda functions have **no name.***

*You can use anonymous functions when:*

- ✓ *you want to perform a simple operation and,*
- ✓ *you're going to use this function just once.*

### *Syntax of Python Lambda expression :*



***The syntax of lambda expression contains three parts:***

- ***lambda*** *– This is the keyword used to create a lambda function*
- ***parameters*** *– A lambda function can have zero, one or many parameters. You just need to specify comma-separated parameters after the lambda keyword.*
- ***expression*** *– This is the body of the function. This is where you specify the operation performed by the function.* ***A lambda function can have only one expression.***

***How to call a Lambda Expression in Python ?***

*A lambda function always returns a* ***function object.***

*So you can either call the lambda function directly:*

```
(lambda x: x ** 2)(10)
```

***Output*** *:*

*100*

*or you can assign it a name and then you can call it like a traditional function call:*

```
square = lambda x: x ** 2

square(10)
```

***Output*** *:*

*100*

***Python Lambda function Examples***

***a. Lambda function with one parameter:***

```
lambda x: x**2
```

***b. Lambda function with Multiple Arguments***

*You can send as many arguments as you like to a lambda function; just separate them with a comma ,Here's how you'd create a lambda function with multiple arguments:*

```
# A lambda function that multiplies two values
mul = lambda x, y: x*y
print(mul(2, 5) )    # Prints 10

# A lambda function that adds three values
add = lambda x, y, z: x+y+z
print(add(2, 5, 10))      #Prints 17
```

***c. Lambda function with No Arguments***

*It isn't mandatory to provide arguments to a lambda expression in Python, it works fine without them.*

```
y=lambda :2+3
print (y( ))
#prints 5
```

*Another example would be where the expression is a print statement.*

```
(lambda :print("Hi"))()
#Prints Hi
```

*Hence, we conclude that omitting arguments is acceptable.*

***d. Lambda function with Omitting the Expression:***

*Now let's try if it works without an expression.*

```
 y=lambda a,b:
#SyntaxError: invalid syntax
```

*Clearly, it didn't work. And why would it? The expression's value is what it returns, so without an expression, it would be a waste of time.*

### *Ways to Pass Arguments :*

*Like a normal function, a lambda function supports all the different ways of passing arguments. This includes:*

- *Positional arguments*
- *Keyword arguments*
- *Default argument*
- *Variable list of arguments (*args)*
- *Variable list of keyword arguments (**args)*

*The following examples illustrate various options for passing arguments to the lambda*

```
# Positional arguments
add = lambda x, y, z: x+y+z
print(add(2, 3, 4))          # Prints 9

# Keyword arguments
add = lambda x, y, z: x+y+z
print(add(2, z=3, y=4))  # Prints 9

# Default arguments
add = lambda x, y=3, z=4: x+y+z
print(add(2))      # Prints 9

# *args
add = lambda *args: sum(args)
print(add(2, 3, 4))          # Prints 9

# **args
add = lambda **kwargs: sum(kwargs.values())
print(add(x=2, y=3, z=4))          # Prints 9
```

*Return Multiple Values :*

*To return multiple values pack them in a tuple. Then use multiple assignment to unpack the parts of the returned tuple.*

```
# Return multiple values by packing them in a tuple
findSquareCube = lambda num: (num**2, num**3)
x, y = findSquareCube(2)
print(x)
# Prints 4
print(y)            # Prints 8
```

***Important characteristics of lamda :***

*In particular, a lambda function has the following characteristics:*

***No Statements Allowed :***

*A lambda function can not contain any statements in its body. Statements such as return, raise, pass, or assert in a lambda function will raise a SyntaxError.*

*Here is an example of a lambda function containing assert:*

```
doubler = lambda x: assert x*2
```

***Why lambda function in python can't use the "return" statement?**

***Answer:** Because the return is a statement and the Lambdas function can only contain expressions, no statements.*

***Single Expression Only :***

*Lamda function can have any number of arguments but only one expression, which is evaluated and returned.*

*Although, you can spread the expression over multiple lines using parentheses or a multiline string, but it should only remain as a single expression.*

### *Immediately Invoked Function Expression (IIFE):*

*A lambda function can be immediately invoked. For this reason it is often referred to as an Immediately Invoked Function Expression (IIFE).*

*Here's the same previously seen 'doubler' lambda function that is defined and then called immediately with 3 as an argument.*

```
print((lambda x: x*2)(3))
# Prints 6
```

### *What Exactly is an Expression in Python?*

*As we've seen, the expression is what the Python3 lambda returns, so we must be curious about what qualifies as an expression. An expression here is what you would put in the return statement of a Python Lambda function. The following items qualify as expressions.*

1. *Arithmetic operations like a+b and a**b*
2. *Function calls like sum(a,b)*
3. *A print statement like print("Hello")*

*Other things like an assignment statement cannot be provided as an expression to Python lambda because it does not return anything, not even None.*

### *if else in a Lambda :*

*Generally if else statement is used to implement selection logic in a function. But as it is a statement, you cannot use it in a lambda function. You can use the if else ternary expression instead.*

```
# A lambda function that returns the smallest item
findMin = lambda x, y: x if x < y else y
print(findMin(2, 4))        # Prints 2

print(findMin('a', 'x'))    # Prints a
```

### *Jump Table Using a Lambda :*

*The jump table is a list or dictionary of functions to be called on demand. Here's how a lambda function is used to implement a jump table.*

```
# dictionary of functions
exponent = {'square':lambda x: x ** 2,
            'cube':lambda x: x ** 3}

print(exponent['square'](3))
# Prints 9
print(exponent['cube'](3))
# Prints 27

# list of functions
exponent = [lambda x: x ** 2,
            lambda x: x ** 3]

print(exponent[0](3))
# Prints 9
print(exponent[1](3))
# Prints 27
```

### *Lambda Key Functions :*

*In Python, key functions are higher-order functions that take another function (which can be a lambda function) as a key argument. This function directly changes the behavior of the key function itself. Here are some key functions:*

- *List method: sort()*
- *Built-in functions: sorted(), min(), max()*
- *In the Heap queue algorithm module heapq: nlargest() and nsmallest()*

*In the following example, a lambda is assigned to the key argument so that the list of students is sorted by their age rather than by name.*

```
# Sort the list of taples by the age of students
L = [('Sam', 35),
     ('Max', 25),
     ('Bob', 30)]
x = sorted(L, key=lambda student: student[1])
print(x)          # Prints [('Max', 25), ('Bob', 30), ('Sam', 35)]
```

***Lambda Closures :***

*As a normal function lambda function also can be a closure,*

*Here's a closure constructed with a normal Python function:*

```
def multiplier(x):
    def inner_func(y):
        return x*y
    return inner_func

doubler = multiplier(2)
print(doubler(10))        # Prints 20
```

*Here multiplier() returns inner_func() which computes the multiplication of two arguments:*

- *x is passed as an argument to multiplier()*
- *y is an argument passed to inner_func()*

*Similarly, a lambda can also be a closure. Here's the same example with a lambda function:*

```
multiplier = (lambda x: (lambda y: x*y))
doubler = multiplier(2)
print(doubler(10))        # Prints 20
```

## *Lambda functions vs Traditional functions*

*Let's look at the difference with the help of an example.*

```
# traditional function to return the square of a number

def square(x):

return x ** 2

# lambda function to calculate the square of a number

square_lambda = lambda x: x ** 2

print(square(10))

print(square_lambda(10))
```

*All we've done here is that we've cleaned up some unnecessary syntax and changed some components. We use* ***def*** *to define a traditional function, whereas we use the* ***lambda*** *keyword to create a lambda function. Also, in an anonymous function, we omit the return keyword, thereby further reducing the number of lines.*

*Other than these minor changes, let's compare the two types of functions at a greater depth.*

| ***Traditional Functions*** | ***Lambda Functions*** |
|---|---|
| *They can have multiple expressions/ statements in their body.* | *They can have just a single expression in their body.* |
| *Can have default parameters.* | *Lambda parameters can't have default values.* |
| *Can return an object of any type.* | *Always returns a function object.* |
| *It takes more time for execution* | *These take relatively less time.* |
| *Require more lines of code.* | *It requires a maximum of two lines of code to define and call a lambda function.* |

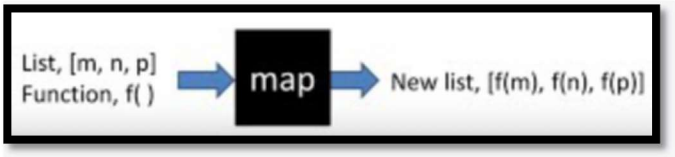
# *Map, Filter and Reduce Functions in Python*

*One of the superpowers of lambda functions is that we can use them inside any normal function in Python.This comes handy while working with high-order functions, i.e, the functions that take another function as an argument.*

*Let's have a look at three such functions – map( ), filter( ) and reduce( )*

***Map Function:***

- *Apply same function to each element of a sequence*
- *return the modified list*




***Syntax:*** *map(fun, iter)*

***Parameters:***

***fun:*** *It is a function to which map passes each element of given iterable.*

***iter:*** *It is a iterable which is to be mapped.*

*Note: You can pass one or more iterable to the map() function*

***Return Type:*** *Returns an iterator of map class.*

*Note: The returned value from map() (map object) then can be passed to functions like list() (to create a list), set() (to create a set)*

*For example, assuming you have a function that calculates the area of a circle*

```
import math
def calculatearea(radius):
    # calculate area of a circle
    area = math.pi * radius * radius
    return area
```

*Also, let's say we have a list of radii as:*

```
radii = [4, 5.2, 9, 10, 3,8, 6, 7.5]
```

*To calculate the areas of circle with these radii, we can loop through the list of radii and call the calculatearea() function for each of the radii.*

*The code to do this is show below.*

```
areas = [ ]
for radius in radii:
    area = calculatearea(radius)
    areas.append(area)
print(areas)
```

*This method above is fine and good. However, we can do the same thing with a single line of code. In this case, we use the map function.*

*To do that, you simply call the map function, and provide the name of the function followed by the list of radii. This is shown below:*

```
areas = map(calculatearea, radii)
print(list(areas))
```

*You will notice that in the second line, we cast the output areas to a list. This is because the map function returns a map object.*

***Lambda functions within map( ):***

```
import math
radii = (4, 5.2, 9, 10, 3,8, 6, 7.5)
print(list(map(lambda x: math.pi * x**2, radii)))
```

*The output is a result of applying the lambda expression (math.pi * x**2) to each item present in the tuple*

***Map ( ) function with multiple arguments :***

```
# Multiply two sequences using map and lambda

list_numbers = [1, 5, 8, 9]

tuple_numbers = (11, 20, 54, 23)

map_iterator = map(lambda x, y: x * y, list_numbers, tuple_numbers)

print(list(map_iterator))
```

First of all, in the above program defined one list and iterator and passed that to the map function, and it will return the multiplication of that list and tuple and returns the iterator, and we have converted that iterator to the list.

See the output of above python map() function with multiple argument example:

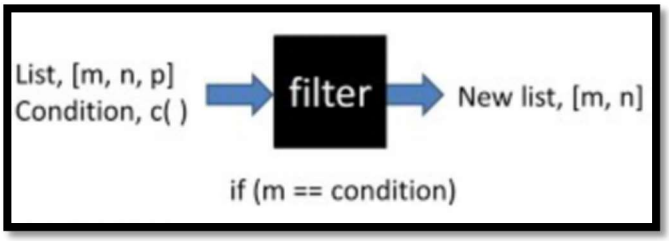
```
    [11, 100, 432, 207]
```

***map( ) can listify the list of strings individually :***

```
# List of strings

l = ['sat', 'bat', 'cat', 'mat']

print(list(map( list, l )))

Output : [['s', 'a', 't'], ['b', 'a', 't'], ['c', 'a', 't'], ['m', 'a', 't']]
```

## *Filter Function:*

*The filter() function filters the given sequence with the help of a function that tests each element in the sequence to be true or not. If None is passed instead of a function, all the items of the sequence which evaluates to False are removed.*




***Syntax:*** *filter(function, sequence)*

***Parameters:***

***function:*** *a 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,*

***Return Type:*** *returns an iterator that is already filtered.*

Assuming you have a list of scores. Now, you will like to filter out scores below average. Let's write do this the normal way. Then we also do it using the filter function. Here we assume that for a list of scores, 50 is the average score.

```
scores = [45, 70, 94.2, 75, 51, 49, 35.1]

newscores = [ ]  # holds scores above average

def isaboveaverage(scores):

    for score in scores:

        if score>50:

            # if score is above average, add it to newscores

            newscores.append(score)

isaboveaverage(scores)

print(newscores)
```

Again we can do this using the filter function. The code is given below:

```
def is_A_student(scores):
    return scores > 50
print(list(filter(is_A_student, scores)))
```

Also, don't forget that the output is converted back to list using the list function. This is because, the output of a filter function is a filter object. If you want to see what a filter object looks like, then first print it without converting.

***Using lambda within filter():***

*The lambda function that is used as a parameter actually defines the condition that is to be checked.*

```
y = filter(lambda x: (x>50), scores)
print(list(y))
```

**Filtering out missing data**

One area you can use the filter function is to filter out missing data in a dataset. For example, the list below contains names of students with some missing values.

```
students = ["Jadon", "Solace", "" "Treasure", "", "", "Onyx", "Booboo"]
```

If you print this list, it would include the missing values. This is not desirable. To filter out this missing values, you can use the filter function. In this case, you pass in *None* as the first parameter.This is shown below:

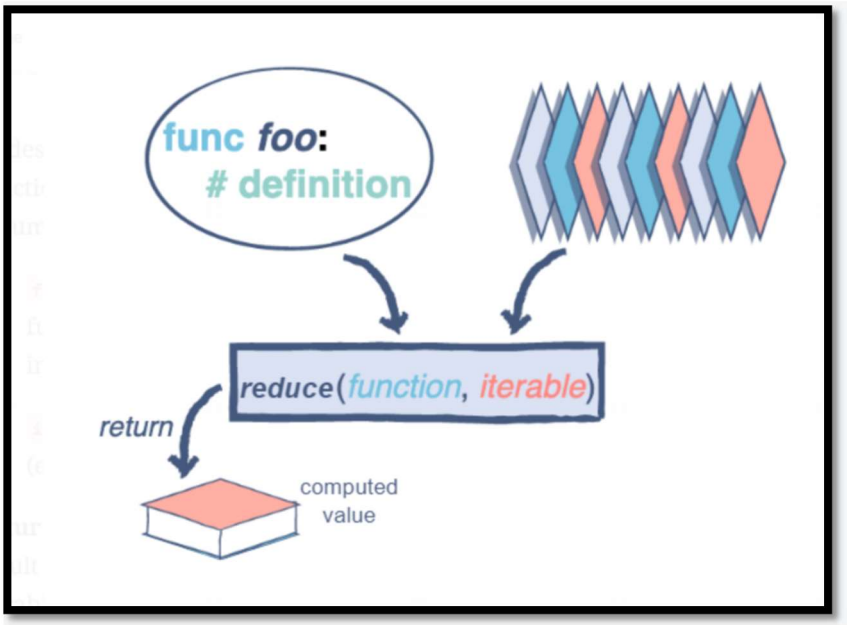
```
newStudents = filter(None, students)

print(list(newStudents))
```

## *Reduce function :*

*The reduce() function performs a rolling-computation as specified by the passed function to the neighboring elements, by taking a function and an iterable as arguments, and returns the final computed value.*

*Note: A rolling-computation is the one where we compute the required value by going through all of the data starting from the first value,( fold left operation – from left to right) where each new result depends on the last computed result of the previous data.*

*The reduce( ) function computation is similar to a for-loop in Python, but being an in-built function, it's much better and faster.*




***Syntax:*** *reduce(function, iterable)*

***Parameters:***

***function:*** *A valid, pre-defined function. This is a lambda function in most cases.*

***iterable:*** *This is an iterable object (e.g. list, tuple, dictionary).*

***Returned****: A single value gained as a result of applying the function to the iterable's element.*

*Note: The reduce() function resides in the functools module which must be imported before the function is called.*

***Examples***

Let's take for example, finding the sum of all numbers in a list:

*Using a pre-defined function*

```
from functools import reduce
# Returns the sum of two elements
def sumTwo(a,b):
    return a+b
result = reduce(sumTwo, [23,21,45,98])

print(result)
```

*Using a lambda function*

```
from functools import reduce
# Returns the sum of all the elements using `reduce`
result = reduce((lambda a, b: a + b), [23,21,45,98])
print(result)
```

***This is how it works:*** *It takes a sequence of items and applies a function to each item cumulatively. That means that it first applies the function to the first two items, then it applies the function to this result and the third item, then this result and the fourth item and so on. It continues until it reaches the last item.This is illustrated below*

$Data: [a_1, a_2, a_3, \dots, a_n]$

$Function: f(x, y)$

$reduce(f, data)$:

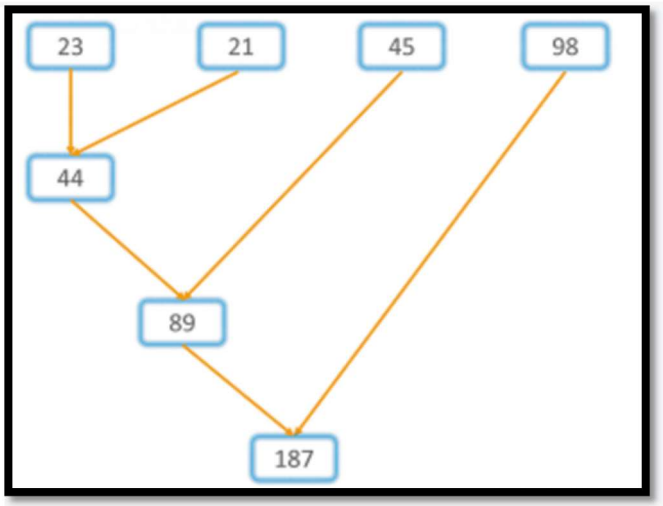
$Step\ 1: val_1 = f(a_1, a_2)$

$Step\ 2: val_1 = f(val_1, a_3)$

$Step\ 3: val_1 = f(val_2, a_4)$:

$\vdots$

$Step\ n-1: val_{n-1} = f(val_{n-2}, a_n)$



***Using map(),filter() and reduce() functions along with each other :***

*When you do this, the internal functions are first solved and then the outer functions operate on the output of the internal functions.*

***Using filter( ) within map( ):***

*The code given below first checks for the condition (x>=3) to be true for the iterables. Then, the output is mapped using the map() function.*

```
c = map(lambda x:x+x,filter(lambda x: (x>=3), (1,2,3,4)))

print(list(c))
```

*Output : [6, 8]*

*If you filter out integers greater than or equal to 3 from the given tuple, you get [3,4] as the result. Then if you map this using(x+x) condition, you will get [6,8], which is the output.*

***Using map( ) within filter( ):***

*When you use the map( ) function within filter( ) function, the iterables are first operated upon by the map function and then the condition of filter( ) is applied to them.*

```
c = filter(lambda x: (x>=3),map(lambda x:x+x, (1,2,3,4)))  #lambda x: (x>=3)

print(list(c))
```

*output: [4, 6, 8]*

***Using map( ) and filter( ) within reduce( ):***

*The output of the internal functions is reduced according to the condition supplied to the reduce() function.*

```
d = reduce(lambda x,y: x+y,map(lambda x:x+x,filter(lambda x: (x>=3), (1,2,3,4))))
print(d)
```

*output: 14*

*The output is a result of [6,8] which is the result of the internal map() and filter() functions.*