* **Functional Programming:**
  + Here, a problem is treated as evaluation of one or more functions.
  + A given problem is decomposed into a set of functions. These functions provide the main source of logic in the program.
* **Functions as First Class Values**
  + **Python treats functions as 'first-class' data values.** This means that:
    - **Functions can be assigned to variables and then called using these variables.**
    - **Functions can be passed as arguments to function and returned from function.**
    - **Functions can be built at execution time, the way lists, tuples, etc. can be.**
* **Example of assigning a function to a variable and calling the function using the variable:**

def print\_a\_word():

print("PDEU")

def add(x,y):

print(x+y)

f1 = print\_a\_word # Assigning a function to a variable.

f2 = add # Assigning a function to a variable.

f1() # will display PDEU

f2(25,30) # will display 55

* **Example of passing a function as argument to a function:**

def add(a,b,c):

c() # will call print\_a\_word()

print(a+b) # will display 55

def print\_a\_word():

print("PDEU") # will display PDEU

f = print\_a\_word # Assigning a function to a variable.

add(25,30,f) # Calling the function with a function as an argument.

* **Example of building function at execution time**
  + **Lambda Functions**
    - Normal functions do have names and are being defined with **def** keyword.
    - Lambda functions don't have names. They are defined with keyword **lambda** and are built at execution time.
    - They are also called anonymous functions or inline functions.
    - They can take any number of arguments but can return only one value.
    - **Syntax:**
    - **lambda arguments: expression**
    - **Examples:**
    - # function that receives an argument and returns its cube.
      * lambda n: n\*\*3
    - # function that receives three arguments and returns their average
      * lambda a,b,c : (a + b + c)/3
    - #function that receives a string, strips any whitespace and returns the uppercase string.
      * lambda str : str.trim().upper()
    - #lambda functions are often used as an argument to other functions.
      * print((lambda n: n\*\*3)(4)) # 64
      * print((lambda a,b,c : (a + b + c)/3),10,20,30) # 20.0
      * print((lambda str : str.trim().upper())," pdeu ") # PDEU
    - #lambda functions can also be assigned to a variable and then invoked.
      * **p = lambda n: n\*\*3**
      * **q = lambda a, b, c: (a+b+c)/3**
      * **r = lambda str : str.trim().upper()**
      * print (p(4))
      * print(q(10,20,30))
      * print(r(" pDeU ")
    - We can also pass container types to a lambda function.
      * lst1= [10,20,30,40,50]
      * print(lambda l : sum(l) / len(l))(lst1))
  + **Higher Order Functions**
    - A higher order function is a function that can receive other functions as an arguments or return them.
    - E.g. we can pass a lambda function to the built-in-sorted() function to sort a dictionary by values.
      * d = { 'O' : 230, 'C' : 150, "S": 175, "N": 35 }
      * d1 = sorted(d.items(), key = lambda kv : kv[1])
      * print (d1) #Output: [('N', 35), ('C', 150), ('S', 175), ('O', 230)]
    - There are three higher order functions, we need to understand:
      * **map()**
      * **filter()**
      * **reduce()**
    - **map()**
      * A map operation applies a function to each element in the sequence like list, tuple, etc. and returns a new sequence containing the results.
      * **General form of map() function:**
        + **map(function\_to\_apply, list\_of\_inputs)**
      * **map()** returns **a map object** which can be converted to a list using list() function.

import math

def func1(n):

return n\*n

lst = [5,10,15,20,25]

m1 = map(math.radians, lst)

m2 = map(math.factorial, lst)

m3 = map(func1,lst)

print(list(m1))

print(list(m2))

print(list(m3))

* + - **filter()** 
      * A filter operation applied a function to all the elements of a sequence. A sequence of those elements for which the function returns True is returned.
      * **General form of filter() function:**
        + **filter(function\_to\_apply, list\_of\_inputs)**
      * **filter()** returns a **filter** object which can be converted to a list using list() function.

def divby5(n):

return True if n% 5 == 0 else False

lst1 = ['A','a','P','q','4','7','+','r','E','Z']

f1 = filter(str.isalpha,lst1)

print(list(f1))

lst2 = [5,25,625,44,57,60]

f2 = filter(divby5,lst2)

print(list(f2))

* + - **reduce()** Function
      * **A reduce operation performs a rolling computation to sequential pairs of values in a sequence and returns the result.**
      * **General form of reduce() function:**
        + **reduce(function\_to\_apply, list\_of\_inputs)**

**from functools import reduce**

def sum(a,b):

return a+b

def prod(a,b):

return a\*b

lst = [1,2,3,4,5]

s = reduce(sum,lst) # ((((1+2)+3)+4)+5)

p = reduce(prod,lst)# ((((1\*2)\*3)\*4)\*5)

print(lst,s,p) # output # [1,2,3,4,5] 15 120

* + - Using lambda with map(), filter(), reduce()
      * We can use map(), filter() and reduce() with lambda functions to simplify the implementation of functions that operate over sequence types like strings, list, tuples.
      * Since map(), filter() and reduce() expect a function to be passed to them, we can also pass lambda functions to them, as shown below:

#using lambda with map()

lst1 = [1,2,3,4,5]

m = list(map(lambda x:x\*x,lst1))

print(m) # Output : [1, 4, 9, 16, 25]

#using lambda with filter()

f = list(filter(lambda x: x % 5 == 0, lst1))

print(f) # Output : [5]

from functools import reduce

lst = [1,2,3,4,5]

s2 = reduce(lambda x,y: x+y, lst) # ((((1+2)+3)+4)+5)

p2 = reduce(lambda x,y: x\*y, lst)# ((((1\*2)\*3)\*4)\*5)

print(lst,s2,p2) # output # [1,2,3,4,5] 15 120

* + - Example: use map() and filter() functions to filter out only those squares which are bigger than 1000

def sqr\_g1000(n):

return n > 1000

lst =[10,20,30,40,50]

y = list(filter(sqr\_g1000,map(lambda z: z \* z, lst)))

print(y) # [1600, 2500]

* + **Where are they useful?**
    - Relational databases use the map/filter/reduce paradigm. A typical SQL query to obtain the maximum salary that a skilled worker gets from an Employees Table will be:
      * SELECT max(salary) FROM employees WHERE grade = 'Skilled'
      * **reduce(max, map(get\_salary, filter(lambda x : x.grade() == 'Skilled', employees)))**
        + Here employees is a sequence, i.e. a list of lists, where each list has the data for one employee.
        + grade = "Skilled" is a filter
        + get\_salary is a map which returns the salary field from the list.
        + and max is a reduce.
    - In SQL terminology map, filter, reduce are called project, select and aggregate respectively.
    - If we can manage our program using map, filter and reduce along with lambda functions, then we can run each operation in separate threads (multithreading) and/or different processes and still get the same results.
* **Write following programs considering functional programming:**
  1. Define three functions fun(), disp() and msg(). Store them in a list and call them one by one in a loop.
  2. Suppose there are two lists, one containing numbers from 1 to 6, and other containing numbers from 6 to 1. Write a program to obtain a list that contains elements obtained by adding corresponding elements of the two lists. (hint: use map and lambda functions)
  3. Generate the list of 10 different random numbers between -15 and 15. Create a new list by obtaining square of all numbers in a list.
  4. Consider the following list:

lst = ['madam','Python',"malayalam",12321]

Write a program to print those strings which are palindromes.

* 1. A list contains names of Faculty Members. Write a program to filter out those names whose length is more than 8 characters.