**Exercise:**

1. Write a function called squared that takes in one int and retrun the second power of it.

Def squared(num):

Return(num\*\*2)

1. Write a function called concatenate that concatenate two words

Def concatenate(word1, word2):

Return(word1+word2)

1. Write a function called average that calculates the average of a list of ints

Def average(lis):

total = 0

for num in lis:

total = total+num

return(total / len(lis))

1. Write a function called variance that calculates the variance of a list of numbers

Def variance(lis):

total = 0

for num in lis:

total = total+num

average = total / len(lis)

square\_sum = 0

for num in lis:

square\_sum = square\_sum + (average - num)\*\*2

return(square\_sum / len(lis))

**Lambda Function:**

In Python, anonymous function is a [function](https://www.programiz.com/python-programming/function) that is defined without a name.

While normal functions are defined using the **def** keyword, in Python anonymous functions are defined using the **lambda** keyword.

**Syntax of Lambda Function in python**

lambda arguments: expression

double = lambda x: x \* 2

print(double(5))

**Lambda functions can take any number of arguments:**

A lambda function that multiplies argument a with argument b and print the result:

x = lambda a, b : a \* b  
print(x(5, 6))

**A lambda function that sums argument a, b, and c and print the result:**

x = lambda a, b, c : a + b + c  
print(x(5, 6, 2))

**Map:**

**Map** applies a function to all the items in an input list.

**Syntax**

map(function\_to\_apply, list\_of\_inputs)

list\_map\_output = list(map\_output)

print(list\_map\_output)

**Example:**

print a list with squared value of input list

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

**Oldest way**

squared = []

for i in items:

squared.append(i\*\*2)

**older way (list comprehension)**

squared = [i\*\*2 for i in items]

**map way(using lambda)**

squared = list(map(lambda x: x\*\*2, items))

**Instead of a list of inputs we can even have a list of functions**

def multiply(x):

return (x\*x)

def add(x):

return (x+x)

funcs = [multiply, add]

for i in range(5):

value = list(map(lambda x: x(i), funcs))

print(value)

**advanced- list comprehension**

[list(map(lambda x: x(i), [multiply, add])) for i in range(5)]

**Iterating over a dictionary using map and lambda**

dict\_a = [{'name': 'python', 'points': 10}, {'name': 'java', 'points': 8}]

map(lambda x : x['name'], dict\_a) # Output: ['python', 'java']

map(lambda x : x['points']\*10, dict\_a) # Output: [100, 80]

map(lambda x : x['name'] == "python", dict\_a) # Output: [True, False]

**Multiple iterables to the map function**

We can pass multiple sequences to the **map**functions as shown below:

list\_a = [1, 2, 3]

list\_b = [10, 20, 30]

list(map(lambda x, y: x + y, list\_a, list\_b)) # Output: [11, 22, 33]

**Filter:**

filter returns only those element for which the function\_object returns *true*

**Syntax:**

filter(function\_object, iterable)

**Example:**

a = [1, 2, 3, 4, 5, 6]

list(filter(lambda x : x % 2 == 0, a)) # Output: [2, 4, 6]

list(filter(lambda x: x%2, a) #[1,3,5]

**what happened here?**

The filter function only return where the function\_object returns true. In python, 0 in Boolean value is false and 1 is true. Therefore the return value is all the numbers with remainder when dividing by 2.

**Filter list of dictionary**

dict\_a = [{'name': 'python', 'points': 10}, {'name': 'java', 'points': 8}]

list(filter(lambda x : x['name'] == 'python', dict\_a)) # Output: [{'name': 'python', 'points': 10}]

**Exercise:**

1. Apply a function to list **a** that triples **a** using lambda function. a = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
2. From **a,** filter out everything that is multiple of 3.

**Nested Function:**

Like nested loops, we simply create a function using def inside another function to nest two functions.

**Syntax**:

def f1(): #outer function

expression#1

def f2(): #inner function

expression#2

return

f1()

**example:**

def f1(): #outer function

print ("Hello")

def f2(): #inner function

print ("world")

return f2()

**another example**

def f1(): #outer function

x = 1 #variable defined in f1 function

def f2(a): #inner function

print (a+x) #able to access the variable of outer function

f2(2)

f1()

**Global & Local Variables**

**Local**

def f1():

local= 5

return local

f1()

#a can't be accessed outside the function f1

print (local) # will give error

local is a local variable

**Global**

glob = 5

def f1():

print (glob) # will print 5

f1()

here glob is a global variable

**closure:**

def power(exponent):

def exponent\_of(base):

return base\*\*exponent

return f2

square= power(2)

cube = power(3)

**old way:**

**def power(a,b):**

**return(a\*\*b)**

**use Lambda:**

def myfunc(n):  
  return lambda a : a \* n  
mydoubler = myfunc(2)  
print(mydoubler(11))

def myfunc(n):  
  return lambda a : a \* n  
mytripler = myfunc(3)  
print(mytripler(11))

def myfunc(n):  
  return lambda a : a \* n  
  
mydoubler = myfunc(2)  
mytripler = myfunc(3)  
  
print(mydoubler(11))   
print(mytripler(11))

**Game:**

**Bulls and cow**

**Default number is 1234, you guess is 2345, you will get 3 cows and 0 bull. Number of cows = how many numbers in the default number but at wrong place. Number of bulls = how many numbers at correct place.**

**CODE:**

**def cow\_game():**

**number = '1380'**

**cow = 0**

**def calculare\_cow\_and\_bulls(guess\_number):**

**cows = 0**

**bulls = 0**

**for i in range(0,4):**

**if guess\_number[i] == number[i]:**

**bulls +=1**

**elif guess\_number[i] in number:**

**cows+=1**

**return cows, bulls**

**while cow<4:**

**user\_number = input("Guess the 4-digits number:")**

**cow,bull = calculare\_cow\_and\_bulls(user\_number)**

**print('Cows: ', cow)**

**print("Bulls: ", bull)**

**if bull ==4:**

**print('Congratulations!')**

**break**