Python Functions

Lambda Function

- Lambda function is also known as anonymous(without a name) function that directly accepts the number of arguments and a condition or operation to perform with that argument which is colon-separated and returns the result.
- To perform a small task while coding on a large codebase or a small task inside a function, we use the lambda function in a normal process.
- lambda *arguments* : *expression*
- x = lambda a, b, c : a + b + c
  print(x(5, 6, 2))

Cont’d

- No name – Lambda functions have no name, while normal operations have a proper name.

- 2) Lambda has no return Value – the normal function created using def keyword returns value or sequence data type, but a complete procedure is returned in the lambda function. Suppose we want to check the number is even or odd, so lambda function syntax is like the snippet below.

- b = lambda x: "Even" if x%2==0 else "Odd"
  - b(9)

- 3) A function is only in one line – Lambda function is written and created only in one line while in a typical process we use indentations

Cont’d from slide 3

- **Not used for Code reusability –** Lambda function cannot be used for code reusability, or you cannot import this function in any other file. In contrast, standard functions are used for code reusability, which you can use in external files.

Why should we use Lambda Functions?

- In typical scenarios, we do not use the Lambda function while we use it with higher-order functions.

- Now higher-order functions are the functions that need one more function in them to accomplish their task, or when one function is returning any another function, then we use Lambda functions.

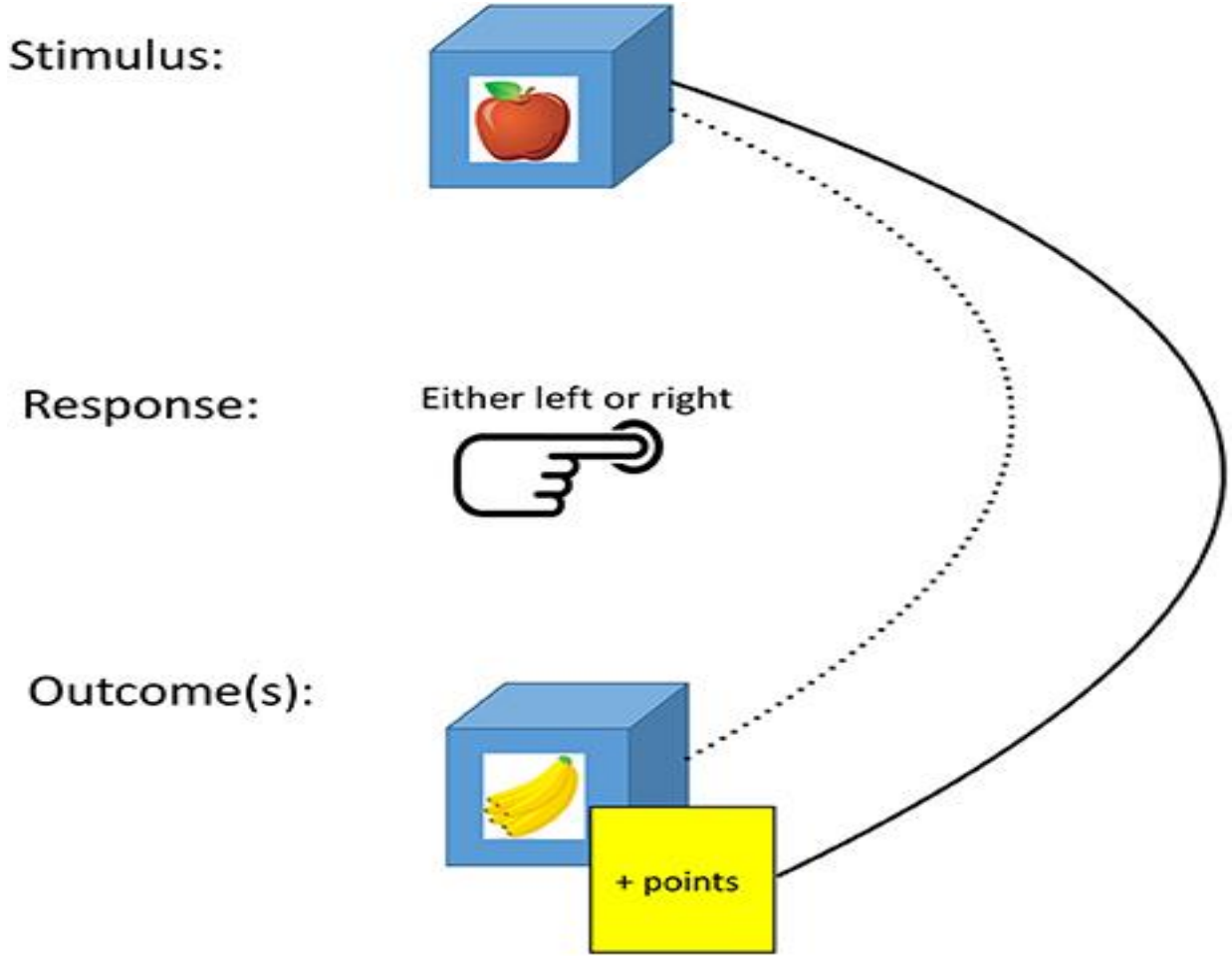
What are Higher Order Functions?

- Let’s understand higher-order functions with the help of an example. Suppose I have an integer list, and we have to return three outputs.
- a sum of all even numbers in a list, a sum of all odd numbers in a list, a sum of all numbers that are divisible by three

- *Now, suppose you approach this problem using normal function. In that case, you will declare three different variables that store various tasks and use one for loop and return the resultant three variables. This is a good approach and will work fine*

Why create high order functions?

So, people create a higher-order function to control the behavior of a part. Now we will study some in-built higher-order tasks in Python. Most people working with Python skip using this function or use it for only a limited time, but these functions are handy and can save more code of lines to write.



This Photo by Unknown Author is licensed under CC BY

In-Built High Order Functions in Python

Map Function

- The map function is a function that accepts two parameters.
- The first is a function, and the second is any sequence data type that is iterable.

- arr = [2,4,6,8]
- arr = list(map(lambda x: x*x, arr))
- print(arr)

Another example

- >>> def square(number):
- ... return number ** 2
- ...

- >>> numbers = [1, 2, 3, 4, 5]

- >>> squared = map(square, numbers)

- >>> list(squared)

Convert all items of a list to integer

- >>> str_nums = ["4", "8", "6", "5", "3", "2", "8", "9", "2", "5"]

- >>> int_nums = map(int, str_nums)
- >>> int_nums
- <map object at 0x7fb2c7e34c70>

- >>> list(int_nums)
- [4, 8, 6, 5, 3, 2, 8, 9, 2, 5]

Map and Lambda

- >>> numbers = [1, 2, 3, 4, 5]

- >>> squared = map(lambda num: num ** 2, numbers)

- >>> list(squared)
- [1, 4, 9, 16, 25]

Multiple iterables with map

- >>> first_it = [1, 2, 3]
- >>> second_it = [4, 5, 6, 7]

- >>> list(map(pow, first_it, second_it))
- [1, 32, 729]

Example: Iterables with map

- >>> list(map(lambda x, y: x - y, [2, 4, 6], [1, 3, 5]))
- [1, 1, 1]

- >>> list(map(lambda x, y, z: x + y + z, [2, 4], [1, 3], [7, 8]))
- [10, 15]

Example 2: Iterables with Map

- >>> string_it = ["processing", "strings", "with", "map"]
- >>> list(map(str.capitalize, string_it))
- ['Processing', 'Strings', 'With', 'Map']

- >>> list(map(str.upper, string_it))
- ['PROCESSING', 'STRINGS', 'WITH', 'MAP']

- >>> list(map(str.lower, string_it))
- ['processing', 'strings', 'with', 'map']

We can use the map function in different ways.

- Suppose we have a list of dictionaries containing details like name, address, etc. Now we have to generate a new list that includes all names.

Python code for Map functions

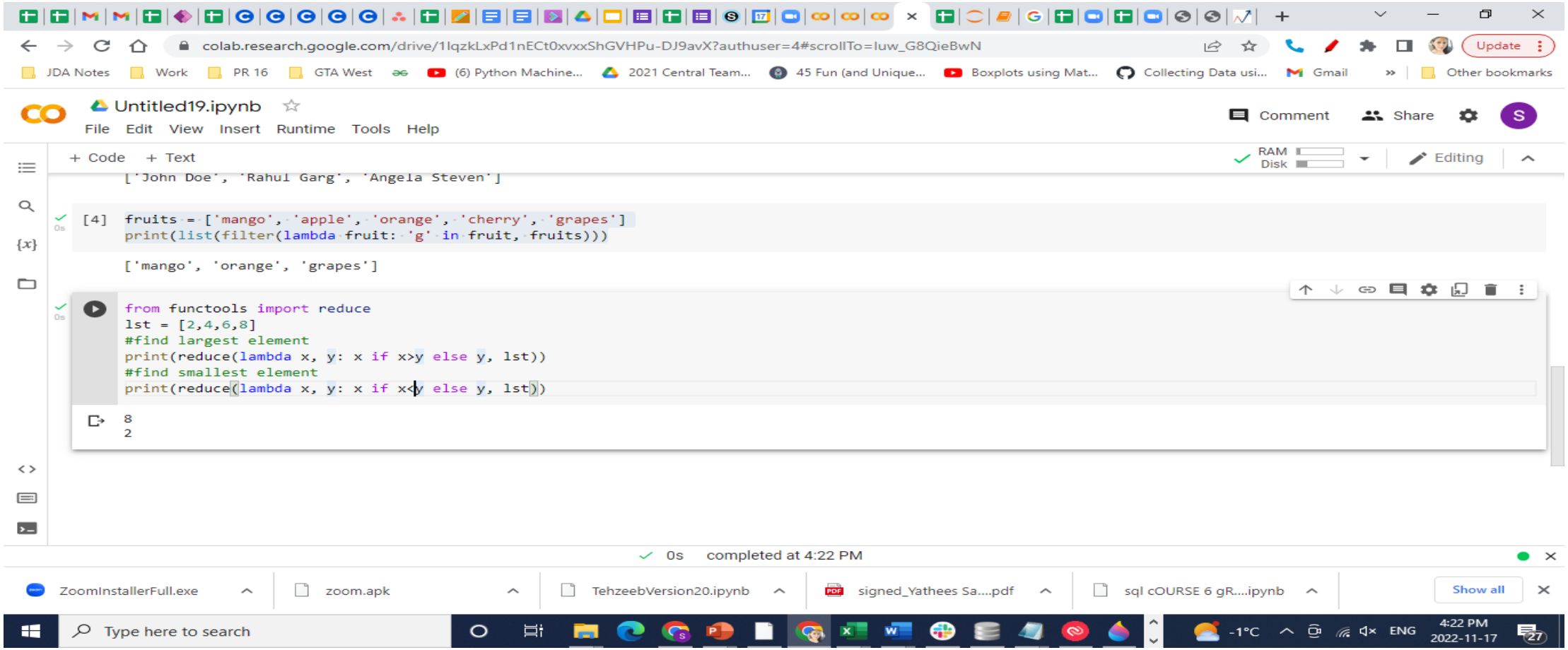
Filter Function

- Filter function filters out the data based on a particularly given condition. Map function operates on each element, while filter function only outputs elements that satisfy specific requirements. Suppose we have a list of fruits, and our task is to output only those names which have the character “g” in their name.
-
  fruits = ['mango', 'apple', 'orange', 'cherry', 'grapes']
- print(list(filter(lambda fruit: 'g' in fruit, fruits)))

Reduce Function

- It is one exciting function that is not directly present in Python, and we have to import it from the functional tools module of Python. Reduce returns a single output value from a sequence data structure because it reduces the elements by applying a given function. Suppose we have a list of integers and find the sum of all the aspects. Then instead of using for loop, we can use reduce function.
  - from functools import reduce
  - lst = [2,4,6,8,10]
  - print(reduce(lambda x, y: x+y, lst)) #Ans-30

Python code for Reduce Function



Alternate ways of High-Order Functions

- List Comprehension : List comprehension is nothing but a for loop to append each item in a new list to create a new list from an existing index or a set of elements. The work we have performed using Map, filter, and reduce can also be done using List Comprehension.

- Suppose we want to change array elements to their square using list comprehension, and the fruit example can also be solved using list comprehension.

Example: Changing array elements

- arr = [2,4,6,8]
- arr = [i**2 for i in arr]
- print(arr)
- fruit_result = [fruit for fruit in fruits if 'g' in fruit]
- print(fruit_result)

What is Dictionary Comprehension?

Dictionary Comprehension

- Same as List comprehension, we use dictionary comprehension to create a new dictionary from the existing one. We can also create a dictionary from a list of elements. Suppose we have a list of integers and want to create a dictionary where each value is a key, and its square is its value. So to make this, we can use Dictionary comprehension.

[6]
```
lst = [2,4,6,8]
#find largest element
print(reduce(lambda x, y: x if x>y else y, lst))
#find smallest element
print(reduce(lambda x, y: x if x<y else y, lst))
```

```
8
2
```

```
lst = [2,4,6,8]
D1 = {item:item**2 for item in lst}
print(D1)
#to create a dict of only odd elements
arr = [1,2,3,4,5,6,7,8]
D2 = {item: item**2 for item in arr if item %2 != 0}
print(D2)
```

```
{2: 4, 4: 16, 6: 36, 8: 64}
{1: 1, 3: 9, 5: 25, 7: 49}
```

Zip Function

- a = ("John", "Charles", "Mike")
  b = ("Jenny", "Christy", "Monica")

  x = zip(a, b)

- Output:

Cont’d: Zip Function

- The zip() function returns a zip object, which is an iterator of tuples where the first item in each passed iterator is paired together, and then the second item in each passed iterator are paired together etc.

- If the passed iterators have different lengths, the iterator with the least items decides the length of the new iterator.

Common use

keys = ['name', 'age'] values = ['Bob', 25]
D = dict(zip(keys, values)) print(D)
# Prints {'age': 25, 'name': 'Bob'}

References

- Reference Link