PES UNIVERSITY
Department of Computer Science and Engineering (UG Studies)
PES University, Bangalore, India

Introduction to Computing using Python (UE19CS101)

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU
coprakasha@pes.edu

Functional Programming:

Functional programming is a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

In functional code, the output of the function depends only on the arguments that are passed. Calling the function f for the same value of x should return the same result f(x) no matter how many times you pass it.

In functional programming, basically there are 5 main functions with which we can perform most of the operations on collection. Those are:

1. **map**
2. **filter**
3. **concatAll**
4. **reduce**
5. **zip**

----------***----------

In the below example, we will try three different tasks. Then we shall retry them using a totally different technique.

Observe the function what1. Given a list of strings, it creates a list of lengths of the corresponding strings and returns the list.

Observe the following steps.

- There is some initialization
- There is a traversal through all the elements of the iterable – in this case, a list.
- There is an application of some operation on each of these elements
- The result is collected in an iterable – in this case a list.
- The list is returned.

Now let us observe the function what2. This function receives a list of strings as argument. This walks through the list. Converts each string to its uppercase equivalent. Puts them into a list. Then returns the list.

Please observe both what1 and what2 do similar tasks – but the operation performed on the element of the iterable is different,

How about the function what3? It is very similar to the earlier two examples. Here the input is a list of numbers. Each number is squared – the operation is different.

```
# name: 5_map.py

# given a list of strings, find their corresponding lengths

def what1(x):
      res = []
      for w in x:
            res.append(len(w))
      return res

a = [ 'apple', 'pineapple', 'fig', 'mangoes' ]
b = what1(a)
print(b)    # [5, 9, 3, 7]

# given a list of strings, convert to uppercase

def what2(x):
      res = []
      for w in x:
            res.append(str.upper(w))
      return res

a = [ 'apple', 'pineapple', 'fig', 'mangoes' ]
b = what2(a)
print(b)    # [APPLE, PINEAPPLE, FIG, MANGOES]

# given a list of numbers, square each number

def what3(x):
      res = []
      for w in x:
            res.append(w * w)
      return res

a = [11, 33, 22, 44]
b = what3(a)
print(b)    # [121, 1089, 484, 1936]
```

Python map() Function

The map() function **executes a specified function for each item in a iterable**. The item is sent to the function as a parameter.

Syntax

```
map(function, iterables)
```

Parameter Values

Parameter	Description
function	Required. The function to execute for each item
iterable	Required. A sequence, collection or an iterator object. You can send as many iterables as you like, just make sure the function has one parameter for each iterable.

Example 1:

Calculate the length of each word in the tuple:

```
def myfunc(n):
  return len(n)

x = map(myfunc, ('apple', 'banana', 'cherry'))

print(list(x))  # [5, 6, 6]
```

Example 2:

Make new fruits by sending two iterable objects into the function:

```
def myfunc(a, b):
  return a + b

x = map(myfunc, ('apple', 'banana', 'cherry'), ('orange', 'lemon', 'pineapple'))

print(list(x))  # ['appleorange', 'bananalemon', 'cherrypineapple']
```

Example 3:

```
# map:  allows us to walk through an iterable
#       apply some function
#       collect the result

# map(<callable>, <iterable>)

a = [ 'apple', 'pineapple', 'fig', 'mangoes' ]

b = list(map(len, a))
print(b)    # [5, 9, 3, 7]

b = list(map(str.upper, a))
print(b)    # [APPLE, PINEAPPLE, FIG, MANGOES]

a = [11, 33, 22, 44]
b = list(map(lambda x : x * x, a))
print(b)    #[121, 1089, 484, 1936]
```

We observe this requirement at number of places in programming. We walk through an iterable, do some operation, collect the result in an iterable, The number of elements in the output will be same as the number of elements in the input.

The map() function has the following characteristics.

- **The function map takes two arguments - a callable and an iterable.**
- **The map function causes iteration through the iterable, applies the callable on each element of the iterable and creates a map object which is also an iterable.**
- All these happen only conceptually as the **map function is lazy**. **Unless the map object is iterated, none of these operations happen!**
- We can force an iteration of the map object by passing the map object as an argument for the list/set/tuple constructor.
- **The first argument for the map function in our case should be a callable which takes one argument – could be**
    - **a free function (like len, abs, max, int, …) or**
    - **a function of a type (like str.upper, str.join, ….) or**
    - **a user defined function or**
    - **a lambda function as well**.

Let us understand a few more examples using map.

The iterable is a range object standing for the sequence 1, 2, … 10. The callable is a user defined function which returns a string. The map object is a lazy iterable which contains the multiplication of a given number. The for loop of the client iterates through the map object and displays the elements.

#Problem Statement: Generate multiplication table for a given number.

Example 4: Without using map() function

```
#Given a number, generate its multiples with numbers from 1 to n. or
#print(list(range(n, n * 10 + 1, n)))  # for n=5, o/p is  [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]

n = int(input("enter a number : "))
def foo(i) :
      prod = n * i
      #print(n, " X ", i, " = ", prod)
      return str(n) + " X " + str(i) + " = " + str(prod) + "\n"

for i in range(1, 11):
      print(foo(i), end = "")
```

```
Enter a number : 5
5 X 1 = 5
5 X 2 = 10
5 X 3 = 15
5 X 4 = 20
5 X 5 = 25
5 X 6 = 30
5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
```

```
5 X 10 = 50
```

Example 5a: Using map() function

```
n = int(input("enter a number : "))
def foo(i) :
      prod = n * i
      #print(n, " X ", i, " = ", prod)
      return str(n) + " X " + str(i) + " = " + str(prod) + "\n"

print(list(map(foo, range(1, 11))))
```

```
Enter a number : 5
['5 X 1 = 5\n', '5 X 2 = 10\n', '5 X 3 = 15\n', '5 X 4 = 20\n', '5 X 5 = 25\n', '5 X 6 =
30\n', '5 X 7 = 35\n', '5 X 8 = 40\n', '5 X 9 = 45\n', '5 X 10 = 50\n']
```

Example 5b: Using map() function

```
n = int(input("enter a number : "))
def foo(i) :
      prod = n * i
      #print(n, " X ", i, " = ", prod)
      return str(n) + " X " + str(i) + " = " + str(prod) + "\n"

for line in map(foo, range(1, 11)):
      print(line, end = "")
```

```
Enter a number : 5
5 X 1 = 5
5 X 2 = 10
5 X 3 = 15
5 X 4 = 20
5 X 5 = 25
5 X 6 = 30
5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
5 X 10 = 50
```

Example 6:

This is another useful program to list the filenames and their sizes in the current directory. We are using the os module which supports a few useful functions.

- `os.listdir(".")` gives the names in the current directory.
- `os.path.getsize(filename)` gives the size of the file in bytes

In this case, the map object produces an iterable of tuples, each having the filename and the corresponding size. As there is no such callable available directly, we are using our own function get_name_size as the callable.

```
# name: 7_map.py
# use the os module
```

```
#     os.listdir("path") gives the output of ls command
#     path = "." ; list in the current directory
#     os.path.getsize(filename) => size of file in bytes

import os
# make an iterable of all files
for size in map(os.path.getsize , os.listdir(".")) :
      print(size)

def get_name_size(name):
      return (name, os.path.getsize(name))
for pair in map(get_name_size , os.listdir(".")) :
      print(pair[0],  "=>", pair[1])
```

Python filter() Function

The filter() method constructs an iterator from elements of an iterable for which a function returns true.

Syntax

```
filter(function, iterable)
```

Parameter Values

Parameter	Description
function	A Function to be run for each item in the iterable
iterable	The iterable to be filtered

Example 1:

Filter the array, and return a new array with only the values equal to or above 18:

```
ages = [5, 12, 17, 18, 24, 32]

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

adults = filter(myFunc, ages)

for x in adults:
  print(x)

#or print(list(adult))  # [18, 24, 32]
```

The map() function has the following characteristics.

- **Input** : an iterable of some number of elements (say n)
- **Output**: a lazy iterable of same number of elements(also n)
- **Elements in the output**: obtained by applying the callback on the elements of the input.

There are cases where we want to remove a few elements of the input iterable. Then we use the function filter.

The filter function has the following characteristics.

- **Input** : an iterable of some number of elements (say n)
- **Output:** a lazy iterable of 0 to n elements(between 0 and n)
- **Elements in the output**: **apply the callback on each element of the iterable – if the function returns True, select the input element and otherwise do not select the input element**.

There are many cases where we may want to do some mapping and filtering. It is always a good habit filter first and then map. Otherwise map will have to do processing of some elements which eventually be filtered out.

Example 2:

Problem Statement: For a given range(0,n), Pickup all odd numbers

```
# name: 8_map_filter_reduce.py

print(list(map(lambda x : x % 2, range(5))))   # Input: 0 1 2 3 4  Output: [0, 1, 0, 1, 0]
print(list(filter(lambda x : x % 2, range(5)))) # Input: 0 1 2 3 4  Output: [1, 3]

# map:
#     returns an iterable
#     input : n elements
#     output : n elements

#   output : not the input, result of the fn call on the input

# filter:
#     returns an iterable
#     input : n elements
#     output : anything between 0 and n elements

#     output : has elements given in the input itself
#                 not the result of the function call
#                 gives the output only when the function returns a true value
```

Example 3:

Problem Statement: Pickup all words in list a which have character 'r'.

```
a = [ 'rama', 'krishna', 'balarama', 'lakshmana', 'dasharatha', 'sita' ]
print(list(filter(lambda s : 'r' in s , a)))
```

Problem Statement: Pickup all words whose length exceeds 4

```
a = [ 'rama', 'krishna', 'balarama', 'lakshmana', 'dasharatha', 'sita' ]
print(list(filter(lambda s : len(s) > 4 , a)))
```

Problem Statement: convert all strings to uppercase and find all strings whose length exceeds 4

```
# Inefficient code
a = [ 'rama', 'krishna', 'balarama', 'lakshmana', 'dasharatha', 'sita' ]
print(list(filter(lambda s : len(s) > 4 , map(str.upper , a))))

# good/efficient code
print(list(map(str.upper, filter(lambda s : len(s) > 4, a))))
```

Example 4: List only vowels from the given list

```
# list of alphabets
alphabets = ['a', 'b', 'd', 'e', 'i', 'j', 'o']

# function that filters vowels
def filterVowels(alphabet):
    vowels = ['a', 'e', 'i', 'o', 'u']
    if(alphabet in vowels):
        return True
    else:
        return False

filteredVowels = filter(filterVowels, alphabets)

print('The filtered vowels are:')
for vowel in filteredVowels:
    print(vowel)
```

```
The filtered vowels are:
a
e
i
o
```

Example 5: List only vowels from the given list (write one line code)

```
Alphabets = ['a', 'b', 'd', 'e', 'i', 'j', 'o']
print(list(filter(lambda c : c in 'aeiouAEIOU', alphabets)))
```

Output:

```
['a', 'e', 'i', 'o']
```

Example 6: how filter() method works without the filter function?

```
# random list
randomList = [1, 'a', 0, False, True, '0']

filteredList = filter(None, randomList)

print('The filtered elements are:')
for element in filteredList:
    print(element)
```

```
The filtered elements are:
1
a
True
0
```

Here, we have a random list of number, string and boolean in `randomList`. We pass `randomList` to the `fiter()` method with first parameter (filter function) as `None`.
With filter function as None, the function defaults to Identity function, and each element in `randomList` is checked if it's true or not. When we loop through the final `filteredList`, we get the elements which are true: `1, a, True` and `'0'` ('0' as a string is also true).

Python reduce() Function

The reduce() function accepts a function and a sequence and returns a single value calculated as follows:

- Initially, the function is called with the first two items from the sequence and the result is returned.
- The function is then called again with the result obtained in step 1 and the next value in the sequence. This process keeps repeating until there are items in the sequence.

Syntax: `reduce(function, sequence[, initial])`

When the initial value is provided, the function is called with the initial value and the first item from the sequence.

In Python 2, reduce() was a built-in function. However, in Python 3, it is moved to functools module. Therefore to use it, you have to first import it as follows:

```
from functools import reduce    # only in Python 3
```

The characteristics of reduce() function are as follows.

- **input :** an iterable of some number of elements (say n)
- **output :** a single element
- **processing** : requires a callback which takes two elements. Will apply the function repeatedly and reduce the output to a single element.

In this example, the following shall be the calls on the callable within the function reduce.

```
print(functools.reduce(foo, [11, 22, 33, 44]))
res = foo(11, 22)
res = foo(res, 33)
res = foo(res, 44)
```
There will be n – 1 calls where n is the number of elements in the input iterable.

The function reduce may take a third argument which will indicate the initial value for the first call. In that case, the reduce will call the callable n times.

The following code is interesting. This is a single liner to find the factorial of a given number. The range function produces a lazy iterable of numbers from 1 to n. The callable multiples all of them resulting in the factorial.

Example 1: `# find the sum of first n natural numbers`

```
import functools
n = 10
print(functools.reduce(int.__add__, range(n+1)))  # 55

# reduce:
#     input : n elements
#     output : 1 element
#     callable : takes two arguments
#     callable is called n - 1 times
```

Example 2: `# find the sum of elements of the given list`

```
def foo(x, y):
      print("foo : ", x, y)
      return x + y

print(functools.reduce(foo, [11, 22, 33, 44]))
```

Output:

```
foo :  11 22
foo :  33 33
foo :  66 44
110
```

```
# foo is called n times not n-1 times because of provided initial value 0
# 0 is the initial value
print(functools.reduce(foo, [11, 22, 33, 44], 0))
```

Output:

```
foo :  0 11
foo :  11 22
foo :  33 33
foo :  66 44
110
```

Example 3: `# Print the first letters`

```
names = [ 'nagabhushana', 'satya', 'kumar' ]
print(functools.reduce(lambda x, y : x + y[0], names, ""))
```

Output:

```
nsk
```

```
names = ['Chikkanavangala', 'Onkarappa', 'Prakasha']
print(functools.reduce(lambda x, y : x + y[0], names, ""))
```

Output:

```
COP
```

Example 4: `# output a single string`

```
words = ['Raja', 'ram ', 'mohan ', 'roy']
print(functools.reduce(lambda x, y : x + y, words))
```

Output:

```
Rajaram mohan roy
```

Example 5: `# find factorial of n in a single statement`

```
n = 5
print(functools.reduce(int.__mul__ , range(1, n + 1)))
```

Output:

```
120
```

Example 6: `# find the biggest element in an array`

```
a = [33, 11, 55, 44, 22]
import functools
def big(a, b) :
      if a > b :
            return a
      else:
            return b
print(functools.reduce(big , a))
```

Output:

```
55
```

Example 7: `# find sum of digits using reduce`

```
n = '37195'
print(functools.reduce(int.__add__ , map(int, n)))
```

Output:

```
25
```

The last line of this code is interesting. The variable n being a string is an iterable. The function map returns an iterable where each element is the int equivalent of the corresponding character in the string n. The function will now add all these digits and returns a single sum.

Python 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.

Syntax

```
zip(iterator1, iterator2, iterator3 ...)
```

Parameter Values

Parameter	Description
iterator1, iterator2, iterator3 ...	Iterator objects that will be joined together

Example 1:

Join two tuples together:

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

x = zip(a, b)
print(list(x))
```

Output:

```
[('John', 'Jenny'), ('Charles', 'Christy'), ('Mike', 'Monica')]
```

Example 2:

If one tuple contains more items, these items are ignored:

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

x = zip(a, b)
print(set(x))
```

Output:

```
{('Mike', 'Monica'), ('John', 'Jenny'), ('Charles', 'Christy')}
```

Example 3:

```
numberList = [1, 2, 3]
strList = ['one', 'two', 'three']

result = zip(numberList, strList)

# Converting itertor to set
#print(set(result))     #{(2, 'two'), (1, 'one'), (3, 'three')}

# Converting itertor to tuple
#print(tuple(result))   #((1, 'one'), (2, 'two'), (3, 'three'))

# Converting itertor to list
print(list(result))
```

Output:

```
[(1, 'one'), (2, 'two'), (3, 'three')]
```

The characteristics of zip() function are as follows.

zip does not stand for data compression. The function zip is used to associate the corresponding elements of two or more iterables into a single lazy iterable of tuples. It does not have any callback function.

Example 4:

```
# name: 9_zip.py

a = [1, 2, 3, 4, 5]
b = list(map(lambda x : x * x * x, a))  # [1, 8, 27, 64, 125]
print(a)
print(b)

# zip : takes number of iterables and returns a single iterable of tuples obtained by
        mapping the corresponding elements
print(list(zip(a, b)))
```

Output:

```
[(1, 1), (2, 8), (3, 27), (4, 64), (5, 125)]
```

Example 5: List all filenames and their size in the current working directory

```
# filenames
import os
names = os.listdir(".")
sizes = list(map(os.path.getsize, names))
#print(list(zip(names, sizes)))
for pair in sorted(zip(names, sizes), key = lambda x : x[1]):
      print(pair[0],  "=>", pair[1])
```

Output:

```
Python19.lnk => 1141
Prakash C O - Shortcut.lnk => 1157
Social Network Analytics.lnk => 1285
untitled0.py => 1656
Neo4j Desktop.lnk => 2353
CDSAML_CENTER.docx => 23387
Elective6_Social Networks Analytics .xlsx => 27849
...
```

Example 6: Unzipping the Value Using zip()

The * operator can be used in conjuncton with zip() to unzip the list.

```
zip(*zippedList)
```

```
coordinate = ['x', 'y', 'z']
value = [3, 4, 5]

result = zip(coordinate, value)
resultList = list(result)
print(resultList)
```

```
c, v =  zip(*resultList)
print('c =', c)
print('v =', v)
```

Output:

```
[('x', 3), ('y', 4), ('z', 5)]
c = ('x', 'y', 'z')
v = (3, 4, 5)
```

References:

1. function_recursion_functional_programming.docx – Prof. N S Kumar, Dept. of CSE, PES University.
2. https://www.w3schools.com/python
3. https://docs.python.org/