PES
UNIVERSITY

Department of Computer Science and Engineering
PES University, Bangalore, India

Python For Computational Problem Solving (UE22CS151A)

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.

The functional programming paradigm has its roots in mathematics and it is language independent. The key principle of this paradigm is the execution of a series of mathematical functions.

In functional code,

- the output of the function depends only on the arguments(input values that are passed) without intermediary values. That eliminates the possibility of side effects, which facilitates debugging.
- Routines that don't cause side effects can more easily run in parallel with one another.
- 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 examples, 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.

1) Given a list of strings, write a function to find their corresponding lengths

In [2]:
```
fruits_list = [ 'apple', 'pineapple', 'fig', 'mangoes' ]

def what1(f_list):
    res = []
    for fruit in f_list:
        res.append(len(fruit))
    return res

fruits_len_list = what1(fruits_list)
print(fruits_len_list)
```

```
[5, 9, 3, 7]
```

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.

2) Given a list of strings, write a function to convert each name to uppercase

In [3]:
```
fruits_list = [ 'apple', 'pineapple', 'fig', 'mangoes' ]

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

fruits_uppercase_list = what2(fruits_list)
print(fruits_uppercase_list)
```

```
['APPLE', 'PINEAPPLE', 'FIG', 'MANGOES']
```

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.

3) Given a list of numbers, write a function to square each number

In [1]:
```
num_list = [11, 33, 22, 44]

def what3(n_list):
    res = []
    for num in n_list:
        res.append(num**2)
    return res

squares_list = what3(num_list)
print(squares_list)
```

```
[121, 1089, 484, 1936]
```

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, iterable)

map returns in iterator that yields the results of applying function to each
element of iterable.
```

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: Using built-in function map, calculate the length of each word in the tuple:

In [1]:
```
fruits = ('apple', 'banana', 'cherry', 'mango', 'orange')

fruits_len = map(len, fruits)
print(fruits_len)
print(list(fruits_len))
```

```
<map object at 0x0000020DC06781C0>
[5, 6, 6, 5, 6]
```

Example 2: Using built-in function map, convert each word in the tuple to uppercase:

In [4]:
```
fruits = ('apple', 'banana', 'cherry', 'mango', 'orange')

x = map(str.upper, fruits)
#print(x)
print(list(x))
```

```
['APPLE', 'BANANA', 'CHERRY', 'MANGO', 'ORANGE']
```

Calling map() With Multiple Iterables

map(f, iterable_1, iterable_2, ..., iterable_n)

The number of iterable_i arguments specified to map() must match the number of arguments that f expects. f acts on the first item of each iterable_i, and that result becomes the first item that the return iterator yields. Then f acts on the second item in each iterable_i, and that becomes the second yielded item, and so on.

Example 3: Make new fruits by sending two iterable objects into the function:

In [5]:
```
fruits1=('apple', 'banana', 'cherry','mango')
fruits2=('orange', 'lemon', 'pineapple', 'kiwi')

def str_concat(a, b):
  return a + b

newfruits = map(str_concat, fruits1, fruits2)

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

```
['appleorange', 'bananalemon', 'cherrypineapple', 'mangokiwi']
```

In [10]:
```
fruits1=('apple', 'banana', 'cherry','mango')
fruits2=('orange', 'lemon', 'pineapple')

newfruits = map(lambda s1,s2 : s1+s2, fruits1, fruits2)
print(list(newfruits))  # ['appleorange', 'bananalemon', 'cherrypineapple']
```

```
['appleorange', 'bananalemon', 'cherrypineapple']
```

Example 3a: Using map and lambda functions, square each number in the list:

In [5]:
```
num_list = [11, 33, 22, 44]

squares = map(lambda x : x * x, num_list)
print(list(squares))
```

```
[121, 1089, 484, 1936]
```

map:

- allows us to walk through an iterable
- apply some function
- collect the result

map(callable, iterable)

In [7]:
```
fruits_list = [ 'apple', 'pineapple', 'fig', 'mangoes' ]

b = list(map(str.upper, fruits_list))
print(b)
```

```
['APPLE', 'PINEAPPLE', 'FIG', 'MANGOES']
```

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
  - 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.

Example 4: Generate multiplication table for a given number.

In [11]:
```
#Without using map() function : approach-1
#Given a number, generate its multiples with numbers from 1 to n.
n=6
print(list(range(n,n*10+1,n)))
```
```
[6, 12, 18, 24, 30, 36, 42, 48, 54, 60]
```

In [12]:
```
#Without using map() function : approach-2
n = int(input("Enter a number : "))
def table(i) :
    prod = n * i
    return str(n) + " X " + str(i) + " = " + str(prod) + "\n"

for i in range(1, 11):
    print(table(i), end = "")
```
```
Enter a number : 6
6 X 1 = 6
6 X 2 = 12
6 X 3 = 18
6 X 4 = 24
6 X 5 = 30
6 X 6 = 36
6 X 7 = 42
6 X 8 = 48
6 X 9 = 54
6 X 10 = 60
```

In [7]:
```
#Using map() function
n = int(input("enter a number : "))
def table(i) :
    prod = n * i
    return str(n) + " X " + str(i) + " = " + str(prod) + "\n"

print(list(map(table, 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']
```

In [8]:
```
# Using map() function
n = int(input("enter a number : "))
def table(i) :
    prod = n * i
    return str(n) + " X " + str(i) + " = " + str(prod) + "\n"

for line in map(table, 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
```

In [14]:
```
# Using map() and lambda functions
n = int(input("enter a number : "))

table = map(lambda i : n*i, range(1, 11))
print(list(table))
```

```
enter a number : 5
[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

Example 5:

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.

In [3]:
```
# 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])
```

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.

filter() function

Selecting Elements From an Iterable With filter()

filter() allows you to select or filter items from an iterable based on evaluation of the given function. It's called as follows:

```
filter(f, iterable)
```

filter(f, iterable) applies function f to each element of iterable and returns an iterator that yields all items for which f is truthy. Conversely, it filters out all items for which f is falsy.

Note:

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

Example 1: Filter the array, and return a new array with only the values equal to or above 18:

In [19]:
```
ages = [5, 12, 17, 18, 24, 32]
def greater_than_17(x):
    if x > 17:
        return True
    else:
        return False

adults_age = filter(greater_than_17, ages)
for age in adults_age:
    print(age, end=' ')

#print(list(filter(greater_than_17, ages))) # [18, 24, 32]
```
```
18 24 32
```

In [18]:
```
ages = [5, 12, 17, 18, 24, 32]
def greater_than_17(n):
    return n>17

adults_age = filter(greater_than_17, ages)
print(list(adults_age))
```
```
[18, 24, 32]
```

In [12]:
```
ages = [5, 12, 17, 18, 24, 32]
def greater_than_17(n):
    return n>17

def myFilter(func,iterable1):
    new_list=[]
    for ele in iterable1:
        if func(ele) == True:
            new_list.append(ele)
    return new_list

adults_age = myFilter(greater_than_17, ages)
print(list(adults_age))
```
```
[18, 24, 32]
```

In [1]:
```
ages = [5, 12, 17, 18, 24, 32]

adults_age = filter(lambda n : n>17, ages)
print(list(adults_age))
```
```
[18, 24, 32]
```

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

In [3]:
```
oddnum = filter(lambda x : x%2 != 0, range(20)) # Input: 0 1 2 3 4
                                                # Output: [1, 3]
print(list(oddnum))
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

In [2]:
```
oddnum = filter(lambda x : x%2, range(20))# Input: 0 1 2 3 4
                                          # Output: [1, 3]
                                          # zero is False in python
print(list(oddnum))
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

In [18]:
```
print(list(map(lambda x : x%2, range(5)))) # Input: 0 1 2 3 4
                                            # Output: [0, 1, 0, 1, 0]
```

```
[0, 1, 0, 1, 0]
```

Note:

1) map: returns an iterable

- input -> n elements
- output -> also contains n elements
- output contains not the same inputs, but the ouput is the result of the function call on the inputs.

2) 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 and gives the output only when the function returns a true value.

Example 3: Pickup all names in nameslist which have character ‘r’.

In [11]:
```
names_list=[ 'rama', 'krishna', 'balarama', 'lakshmana', 'dasharatha', 'sita' ]

print(list(filter(lambda s: 'r' in s, nameslist)))
```

```
['rama', 'krishna', 'balarama', 'dasharatha']
```

Example 4: Pickup all words whose length exceeds 4

In [12]:
```
names_list = [ 'rama', 'krishna', 'balarama', 'lakshmana', 'dasharatha', 'sita' ]
print(list(filter(lambda s : len(s) > 4 , names_list)))
```

```
['krishna', 'balarama', 'lakshmana', 'dasharatha']
```

Example 5: Convert all strings to uppercase and find all strings whose length exceeds 4

In [13]:
```
# Inefficient code (n+n function calls)
names_list = [ 'rama', 'krishna', 'balarama', 'lakshmana', 'dasharatha', 'sita' ]
print(list(filter(lambda s : len(s) > 4 , map(str.upper , names_list))))
```

```
['KRISHNA', 'BALARAMA', 'LAKSHMANA', 'DASHARATHA']
```

In [14]:
```
# good/efficient code  ( n+[0..n] function calls )
names_list = [ 'rama', 'krishna', 'balarama', 'lakshmana', 'dasharatha', 'sita' ]
print(list(map(str.upper, filter(lambda s : len(s) > 4, names_list))))
```

```
['KRISHNA', 'BALARAMA', 'LAKSHMANA', 'DASHARATHA']
```

Example 6: List only vowels from the given list of alphabets.

In [5]:
```
# list of alphabets
alphabets = ['a', 'b', 'd', 'e', 'i', 'j', 'o', 'f', 'g']
```

```
# function that filters vowels
def filterVowels(alphabet):
    vowels = ['a', 'e', 'i', 'o', 'u', '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, end=' ')
```

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

Example 7: List only vowels from the given list (write one line code)

In [25]:
```
alphabets = ['a', 'b', 'd', 'e', 'i', 'j', 'o', 'f', 'g']
print(list(filter(lambda c : c in 'aeiouAEIOU', alphabets)))
```

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

In [27]:
```
#print(list(filter(lambda c : c in 'aeiouAEIOU', 'Hello, How are you?')))

oval_iterator = filter(lambda c : c in 'aeiouAEIOU', 'Hello, How are you?')
oval_list=[]
for char in oval_iterator:
    if char not in oval_list:
        oval_list.append(char)
print(oval_list)
```

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

Example 8: how filter() method works without the filter function?

In [17]:
```
# random list
randomList = [1, 'a', 0, False, True, '0']

filteredList = filter(None, randomList)

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

```
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).

Example 9: Find all strings in a given line of text ending with a given suffix.

In [4]:
```
strings_list = input("Enter a line of text : ").split(" ")
print(strings_list)
suffix = input("Enter a suffix: ")
lst = list(filter(lambda s : s.endswith(suffix), strings_list))
print("Strings ending with suffix", suffix, " are ", lst)
```

```
Enter a line of text : laughing and enjoying
['laughing', 'and', 'enjoying']
```

```
Enter a suffix: ing
Strings ending with suffix ing  are  ['laughing', 'enjoying']
```

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. There-fore 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.

In [ ]:
```
from functools import reduce

print(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

In [8]:
```
from functools import reduce
n = 10
print(reduce(int.__add__, range(n+1)))   # 55
```

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

```
55
```

In [9]:
```
import functools
def add2(a,b):
    return a+b

n = 10
print(functools.reduce(add2, range(n+1)))  # 55
```

```
55
```

In [9]:
```
print(help(int))
```

```
Help on class int in module builtins:

class int(object)
 |  int([x]) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Built-in subclasses:
 |      bool
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __and__(self, value, /)
 |      Return self&value.
 |
 |  __bool__(self, /)
 |      self != 0
 |
 |  __ceil__(...)
 |      Ceiling of an Integral returns itself.
 |
 |  __divmod__(self, value, /)
 |      Return divmod(self, value).
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __float__(self, /)
 |      float(self)
 |
 |  __floor__(...)
 |      Flooring an Integral returns itself.
 |  
 |  __floordiv__(self, value, /)
 |      Return self//value.
 |  
 |  __format__(self, format_spec, /)
 |      Default object formatter.
 |  
 |  __ge__(self, value, /)
 |      Return self>=value.
 |  
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |  
 |  __getnewargs__(self, /)
 |  
 |  __gt__(self, value, /)
 |      Return self>value.
 |  
 |  __hash__(self, /)
 |      Return hash(self).
 |  
 |  __index__(self, /)
 |      Return self converted to an integer, if self is suitable for use as an index int
o a list.
 |  
 |  __int__(self, /)
 |      int(self)
 |  
 |  __invert__(self, /)
 |      ~self
 |  
 |  __le__(self, value, /)
 |      Return self<=value.
 |  
 |  __lshift__(self, value, /)
 |      Return self<<value.
 |  
 |  __lt__(self, value, /)
 |      Return self<value.
 |  
 |  __mod__(self, value, /)
 |      Return self%value.
 |  
 |  __mul__(self, value, /)
 |      Return self*value.
 |  
 |  __ne__(self, value, /)
 |      Return self!=value.
 |  
 |  __neg__(self, /)
 |      -self
 |  
 |  __or__(self, value, /)
 |      Return self|value.
 |  
 |  __pos__(self, /)
 |      +self
 |  
 |  __pow__(self, value, mod=None, /)
 |      Return pow(self, value, mod).
 |  
 |  __radd__(self, value, /)
 |      Return value+self.
 |  
 |  __rand__(self, value, /)
 |      Return value&self.
 |
 |  __rdivmod__(self, value, /)
 |      Return divmod(value, self).
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __rfloordiv__(self, value, /)
 |      Return value//self.
 |
 |  __rlshift__(self, value, /)
 |      Return value<<self.
 |
 |  __rmod__(self, value, /)
 |      Return value%self.
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  __ror__(self, value, /)
 |      Return value|self.
 |
 |  __round__(...)
 |      Rounding an Integral returns itself.
 |      Rounding with an ndigits argument also returns an integer.
 |
 |  __rpow__(self, value, mod=None, /)
 |      Return pow(value, self, mod).
 |
 |  __rrshift__(self, value, /)
 |      Return value>>self.
 |
 |  __rshift__(self, value, /)
 |      Return self>>value.
 |
 |  __rsub__(self, value, /)
 |      Return value-self.
 |
 |  __rtruediv__(self, value, /)
 |      Return value/self.
 |
 |  __rxor__(self, value, /)
 |      Return value^self.
 |
 |  __sizeof__(self, /)
 |      Returns size in memory, in bytes.
 |
 |  __sub__(self, value, /)
 |      Return self-value.
 |
 |  __truediv__(self, value, /)
 |      Return self/value.
 |
 |  __trunc__(...)
 |      Truncating an Integral returns itself.
 |
 |  __xor__(self, value, /)
 |      Return self^value.
 |
 |  as_integer_ratio(self, /)
 |      Return integer ratio.
 |
 |      Return a pair of integers, whose ratio is exactly equal to the original int
 |      and with a positive denominator.
 |
 |      >>> (10).as_integer_ratio()
 |      (10, 1)
 |      >>> (-10).as_integer_ratio()
 |      (-10, 1)
 |      >>> (0).as_integer_ratio()
 |      (0, 1)
 |
 |  bit_length(self, /)
 |      Number of bits necessary to represent self in binary.
 |
 |      >>> bin(37)
 |      '0b100101'
 |      >>> (37).bit_length()
 |      6
 |
 |  conjugate(...)
 |      Returns self, the complex conjugate of any int.
 |
 |  to_bytes(self, /, length, byteorder, *, signed=False)
 |      Return an array of bytes representing an integer.
 |
 |      length
 |        Length of bytes object to use.  An OverflowError is raised if the
 |        integer is not representable with the given number of bytes.
 |      byteorder
 |        The byte order used to represent the integer.  If byteorder is 'big',
 |        the most significant byte is at the beginning of the byte array.  If
 |        byteorder is 'little', the most significant byte is at the end of the
 |        byte array.  To request the native byte order of the host system, use
 |        `sys.byteorder' as the byte order value.
 |      signed
 |        Determines whether two's complement is used to represent the integer.
 |        If signed is False and a negative integer is given, an OverflowError
 |        is raised.
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  from_bytes(bytes, byteorder, *, signed=False) from builtins.type
 |      Return the integer represented by the given array of bytes.
 |
 |      bytes
 |        Holds the array of bytes to convert.  The argument must either
 |        support the buffer protocol or be an iterable object producing bytes.
 |        Bytes and bytearray are examples of built-in objects that support the
 |        buffer protocol.
 |      byteorder
 |        The byte order used to represent the integer.  If byteorder is 'big',
 |        the most significant byte is at the beginning of the byte array.  If
 |        byteorder is 'little', the most significant byte is at the end of the
 |        byte array.  To request the native byte order of the host system, use
 |        `sys.byteorder' as the byte order value.
 |      signed
 |        Indicates whether two's complement is used to represent the integer.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  denominator
 |      the denominator of a rational number in lowest terms
 |
 |  imag
 |      the imaginary part of a complex number
 |
 |  numerator
 |      the numerator of a rational number in lowest terms
 |
 |  real
 |      the real part of a complex number

None
```

In [ ]:
```
Example 2:  Find the sum of elements of the given list
```

In [20]:
```python
import functools

def add2(x, y):
    print(x,'+', y, '=', x+y)
    return x + y

print('Sum of elements =',functools.reduce(add2, [11, 22, 33, 44]))
```

```
11 + 22 = 33
33 + 33 = 66
66 + 44 = 110
Sum of elements = 110
```

In [21]:
```python
print(functools.reduce(add2, [11, 22, 33, 44],1))
```

```
1 + 11 = 12
12 + 22 = 34
34 + 33 = 67
67 + 44 = 111
111
```

add2 is called n times not n-1 times because of provided initial value 1.

Example 3: Print the first letters

In [12]:
```python
names = [ 'Punith', 'Raj', 'Kumar' ]


print(functools.reduce(lambda s1,s2 : s1+s2[0],names,''))
```

```
PRK
```

In [19]:
```python
names = ['Chikkanavangala', 'Onkarappa', 'Prakash']
print(functools.reduce(lambda x, y : x + y[0], names, ''))
```

```
COP
```

Exercise 1:

Let str1 = 'Apple Pineapple Pear Lime Elachi'.

Display the output as "APPLE". Use reduce function.

In [1]:
```python
import functools
str1='Apple Pineapple Pear Lime Elachi'

fruits=str1.split()
print(functools.reduce(lambda s1,s2:s1+s2[0],fruits,''))
```

```
APPLE
```

Example 4: output a single string

words = ['Raja', 'ram ', 'mohan ', 'roy']

In [8]:
```
words = ['Raja', 'ram ', 'mohan ', 'roy']
print(functools.reduce(lambda x, y : x + y, words))
```
```
Rajaram mohan roy
```

Example 5: Find factorial of n in a single statement

In [23]:
```
n = 5
print(functools.reduce(int.__mul__ , range(1, n + 1)))
```
```
120
```

In [3]:
```
import functools
n = 5
print(functools.reduce(lambda a,b : a*b , range(2, n + 1)))
```
```
120
```

Example 6: Find the biggest element in an array

In [24]:
```
numlist = [33, 11, 55, 44, 22]

import functools

def big(a, b) :
    if a > b :
        return a
    else:
        return b

print(functools.reduce(big, numlist))
```
```
55
```

In [9]:
```
numlist = [33, 11, 55, 44, 22]

import functools

print(functools.reduce(lambda a,b: a if a>b else b, numlist))
```
```
55
```

In [6]:
```
numlist = [33, 11, 55, 44, 22]

import functools

print(functools.reduce(max, numlist))
```
```
55
```

In [2]:
```
import functools
def sumDigit(d1,d2):
    return int(d1)+int(d2)

num='123456'
print(functools.reduce(sumDigit,num))
```
```
21
```

Example 7: Find sum of digits using reduce

In [3]:
```
n = '37195'
print(list(map(int, n)))
```
```
[3, 7, 1, 9, 5]
```

In [ ]:
```
n = '37195'
print(functools.reduce(int.__add__ , map(int, n)))
```

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:

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

Example 1: Join two tuples together:

In [11]:
```
a = ("Joh", "Charles", "Mike")
b = ("Jenny", "Christy", "Monica")

x = zip(a, b)

print(list(x))
```
```
[('Joh', 'Jenny'), ('Charles', 'Christy'), ('Mike', 'Monica')]
```

Example 2: If one tuple contains more items, these items are ignored:

In [3]:
```
a = ("John", "Charles", "Mike")
b = ("Jenny", "Christy", "Monica", "Vicky")

x = zip(a, b)
print(set(x))
```
```
{('John', 'Jenny'), ('Mike', 'Monica'), ('Charles', 'Christy')}
```

Example 3:

In [5]:
```
numberList = [1, 2, 3]
strList = ['one', 'two', 'three']

result1 = zip(numberList, strList)
# Converting itertor to list
print(list(result1))
```
```
[(1, 'one'), (2, 'two'), (3, 'three')]
```

In [6]:
```
result1 = zip(numberList, strList)
# Converting itertor to tuple
print(tuple(result1))
```

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

In [19]:
```
result = zip(numberList, strList)
# Converting itertor to set
print(set(result))
```

```
{(1, 'one'), (3, 'three'), (2, 'two')}
```

In [18]:
```
result = zip(numberList, strList)
# Converting itertor to list
print(list(result))
```

```
[(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:

In [12]:
```
a = [1, 2, 3, 4, 5]
b = list(map(lambda x : x * x * x, a))   #  [1, 8, 27, 64, 125]

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

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

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

In [12]:
```
# filenames
import os
# Get the path of current working directory
path1 = os.getcwd()
#print(path)
names = os.listdir(path1) # lists all files and directories in the specified directory.
#names = os.listdir(".")  # Get the list of all files and directories in current working

print("Files and directories in '", path1, "' :")
# print the list
print(names)

sizes = list(map(os.path.getsize, names))
print(sizes)

#print(list(zip(names, sizes)))
for pair in sorted(zip(names, sizes), key = lambda x : x[1]):
    print(pair[0],  "=>", pair[1])
```

```
Files and directories in ' C:\Users\DELL\Python Pgms - PCPS ' :
['.ipynb_checkpoints', 'butter.txt', 'case.txt', 'collections_dictionary_demo.ipynb', 'c
ollections_list_demo.ipynb', 'collections_set_demo.ipynb', 'collections_tuple_demo.ipyn
b', 'Control_Structure_Demo.ipynb', 'file.txt', 'file1.txt', 'file_demo.ipynb', 'functio
nal programming_demo.ipynb', 'function_Arguments & Parameters_Nested functions_demo.ipyn
b', 'function_Closures & Callback_demo.ipynb', 'function_decorators_demo.ipynb', 'functi
on_lambda_demo.ipynb', 'function_Recursions_demo.ipynb', 'GUI_demo.ipynb', 'Hello_World.
py', 'id_fn.ipynb', 'library_modules_demo.ipynb', 'MCGW Problem - Program using depth-fi
rst search.ipynb', 'MCGW Problem - Program.ipynb', 'module_import_demo.ipynb', 'mymodul
e.ipynb', 'operators_demo.ipynb', 'out.txt', 'print_demo.ipynb', 'right_data_structure_d
emo.ipynb', 'strings_demo.ipynb', 't.txt', 't_new.txt', 'Untitled - Copy (5).ipynb', 'Un
titled - Copy (6).ipynb', 'Untitled - Copy (7).ipynb', 'Untitled - Copy - Copy (2).ipyn
b', 'Untitled - Copy - Copy (3).ipynb', 'Untitled - Copy - Copy (4).ipynb', 'Untitled -
Copy - Copy (5).ipynb', 'Untitled - Copy - Copy (6).ipynb', 'Untitled - Copy - Copy.ipyn
b', 'Untitled - Copy.ipynb', '__pycache__']
[12288, 134, 89, 276682, 299960, 230326, 71177, 181187, 61, 37, 325503, 760964, 311589,
90281, 81816, 77576, 1278184, 1102088, 201, 4974, 162940, 7421, 8981, 5114, 1904, 37022
5, 13, 6626, 11641, 956792, 51, 45, 69296, 69072, 69072, 555, 555, 555, 555, 555, 555, 5
55, 0]
__pycache__ => 0
out.txt => 13
file1.txt => 37
t_new.txt => 45
t.txt => 51
file.txt => 61
case.txt => 89
butter.txt => 134
Hello_World.py => 201
Untitled - Copy - Copy (2).ipynb => 555
Untitled - Copy - Copy (3).ipynb => 555
Untitled - Copy - Copy (4).ipynb => 555
Untitled - Copy - Copy (5).ipynb => 555
Untitled - Copy - Copy (6).ipynb => 555
Untitled - Copy - Copy.ipynb => 555
Untitled - Copy.ipynb => 555
mymodule.ipynb => 1904
id_fn.ipynb => 4974
module_import_demo.ipynb => 5114
print_demo.ipynb => 6626
MCGW Problem - Program using depth-first search.ipynb => 7421
MCGW Problem - Program.ipynb => 8981
right_data_structure_demo.ipynb => 11641
.ipynb_checkpoints => 12288
Untitled - Copy (6).ipynb => 69072
Untitled - Copy (7).ipynb => 69072
Untitled - Copy (5).ipynb => 69296
collections_tuple_demo.ipynb => 71177
function_lambda_demo.ipynb => 77576
function_decorators_demo.ipynb => 81816
function_Closures & Callback_demo.ipynb => 90281
library_modules_demo.ipynb => 162940
Control_Structure_Demo.ipynb => 181187
collections_set_demo.ipynb => 230326
collections_dictionary_demo.ipynb => 276682
collections_list_demo.ipynb => 299960
function_Arguments & Parameters_Nested functions_demo.ipynb => 311589
file_demo.ipynb => 325503
operators_demo.ipynb => 370225
functional programming_demo.ipynb => 760964
strings_demo.ipynb => 956792
GUI_demo.ipynb => 1102088
function_Recursions_demo.ipynb => 1278184
```

Example 6: Unzipping the Value Using zip()

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

zip(*zippedList)

In [24]:
```
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)
```

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

max() function

The max() method returns the largest element in an iterable or largest of two or more parameters.

Different syntaxes of max() are:

- max(iterable[,*iterables,key, default])
- max(arg1, arg2[,*args, key])

`max()` Parameters

max() has two forms of arguments it can work with.

`max(iterable [, *iterables , key, default])`

- **iterable** - sequence (list, tuple, string), collection (set, dictionary) or an iterator object whose largest element is to be found
- ***iterables (Optional)** - any number of iterables whose largest is to be found
- **key (Optional)** - key function where the iterables are passed and comparison is performed based on its return value
- **default (Optional)** - default value if the given iterable is empty

`max(arg1, arg2 [, *args, key])`

- **arg1** - mandatory first object for comparison (could be number, string or other object)
- **arg2** - mandatory second object for comparison (could be number, string or other object)
- ***args (Optional)** - other objects for comparison
- **key (Optional)**- key function where each argument is passed, and comparison is performed based on its return value

Return value from max()

The max() method returns:

1. `max(iterable [, *iterables, key, default])`

Case	Key	Default	Return value
Empty iterable	No/ Yes	No	Raises **ValueError** exception
Empty iterable	Yes	Yes	Returns the default value
Single iterable (Not empty)	No	No/Yes	Returns the largest among the iterable
Single iterable (Not empty)	Yes	No/Yes	Passes each element in the iterable to the key function Returns largest element based on the return value of the key function
Multiple iterable (Not empty)	No	No/Yes	Returns largest among the given iterables
Multiple iterable (Not empty)	Yes	No/Yes	Passes each iterable to the key function Returns largest iterable based on the return value of the key function

2. `max(arg1, arg2 [, *args, key])`

Case	Key	Return value
First and second argument passed	No	Returns largest among the given arguments
First and second argument passed	Yes	Passes the arguments to the key function Returns largest among the arguments based on the return value of the key function
More than 2 arguments passed	No	Returns largest among the given arguments
More than 2 arguments passed	Yes	Passes each arugment to the key function Returns largest among the arguments based on the return value of the key function

Example 1: Find maximum among the given numbers

In [1]:
```python
# using max(arg1, arg2, *args)
print('Maximum is:', max(1, 3, 2, 8, 5, 10, 6))
```
```
Maximum is: 10
```

In [2]:
```python
# using max(iterable)
num = [1, 3, 2, 8, 5, 10, 6]
print('Maximum is:', max(num))
```
```
Maximum is: 10
```

Example 2: Find the number whose sum of digits is largest using key function.

num_list=[12345, 789, 432, 654, 9999]

In [7]:
```python
def sumDigit(num):
    sum=0
    while num:
        sum=sum+num%10
        num=num//10
    return sum

num_list=[12345, 789, 432, 654, 9999]
print(max(num_list,key=sumDigit))
```

```
9999
```

In [8]:

```
def sumDigit(num):
    sum = 0
    while(num):
        sum = sum + num % 10
        num = num // 10
    return sum

# using max(arg1, arg2, *args, key)
print('Maximum is:', max(100, 321, 267, 59, 40, key=sumDigit))

# using max(iterable, key)
num = [15, 300, 2700, 821, 52, 10, 6]
print('Maximum is:', max(num, key=sumDigit))
```

```
Maximum is: 267
Maximum is: 821
```

Here, each element in the passed argument (list or argument) is passed to the same function sumDigit(). Based on the return value of the sumDigit(), i.e. sum of the digits, the largest is returned.

Example 3: Find list with maximum length using key function

In [7]:

```
num = [15, 300, 2700, 821]
num1 = [12, 2]
num2 = [34, 567, 78]

# using max(iterable, *iterables, key)
print('List with Maximum elements is:', max(num, num1, num2, key=len))
```

```
List with Maximum elements is: [15, 300, 2700, 821]
```

In this program, each iterable num, num1 and num2 is passed to the built-in method len(). Based on the result, i.e. length of each list, the list with maximum length is returned.

In [16]:

```
num = [15, 300, 2700, 821]
num1 = [12, 2]
num2 = [34, 567, 78]

# using max(iterable, *iterables)
print('Maximum is:', max(num, num1, num2))
```

```
Maximum is: [34, 567, 78]
```

In [17]:

```
num = [15, 300, 2700, 821]
num1 = [95, 2]
num2 = [34, 567, 78]

# using max(iterable, *iterables)
print('Maximum is:', max(num, num1, num2))

# compares (num[0],num1[0],num2[0]) i.e., 15, 95 and 34
```

```
Maximum is: [95, 2]
```

In [18]:

```
num = [15, 300, 2700, 821]
num1 = [15, 2]
num2 = [15, 567, 78]
```

```
# using max(iterable, *iterables)
print('Maximum is:', max(num, num1, num2))
```

```
Maximum is: [15, 567, 78]
```

In [4]:
```
num = [15, 2]
num1 = [15, 2]
num2 = [15, 2]

# using max(iterable, *iterables)
print('Maximum is:', max(num, num1, num2))
```

```
Maximum is: [15, 2]
```

Example 4: find the longest string.

In [5]:
```
list_string = input("Enter a string : ").split()
print(list_string)
long_str = max(list_string, key = len)
print("The longest string is ",long_str)
```

```
Enter a string : find the longest string
['find', 'the', 'longest', 'string']
The longest string is  longest
```

In [6]:
```
list_string = input("Enter a string : ").split()
print(list_string)
long_str = max(list_string)
print("The longest string is ",long_str)
```

```
Enter a string : find the longest string
['find', 'the', 'longest', 'string']
The longest string is  the
```

min()

The min() method returns the smallest element in an iterable or smallest of two or more parameters.

Different syntaxes of min() are:

- min(iterable [, *iterables, key, default])
- min(arg1, arg2 [, *args, key])

min() Parameters

min() has two forms of arguments it can work with.

```
min(iterable [, *iterables, key, default])
```

- **iterable** - sequence (tuple, string), collection (set, dictionary) or an iterator object whose smallest element is to be found
- ***iterables (Optional)** - any number of iterables whose smallest is to be found
- **key (Optional)** - key function where the iterables are passed and comparison is performed based on its return value
- **default (Optional)** - default value if the given iterable is empty

```
min(arg1, arg2 [, *args, key])
```

- **arg1** - mandatory first object for comparison (could be number, string or other object)
- **arg2** - mandatory second object for comparison (could be number, string or other object)
- ***args (Optional)** - other objects for comparison
- **key(Optional)** - key function where each argument is passed, and comparison is performed based on its return value

Return value from min()

The min() method returns:

1. min(iterable[, *iterables, key, default])

Case	Key	Default	Return value
Empty iterable	No/Yes	No	Raises **ValueError** exception
Empty iterable	Yes	Yes	Returns the default value
Single iterable (Not empty)	No	No/Yes	Returns the smallest among the iterable
Single iterable (Not empty)	Yes	No/Yes	Passes each element in the iterable to the key function Returns smallest element based on the return value of the key function
Multiple iterable (Not empty)	No	No/Yes	Returns smallest among the given iterables
Multiple iterable (Not empty)	Yes	No/Yes	Passes each iterable to the key function Returns smallest iterable based on the return value of the key function

2. min(arg1, arg2 [, *args, key])

Case	Key	Return value
First and second argument passed	No	Returns smallest among the given arguments
First and second argument passed	Yes	Passes the arguments to the key function Returns smallest among the arguments based on the return value of the key function
More than 2 arguments passed	No	Returns smallest among the given arguments
More than 2 arguments passed	Yes	Passes each arugment to the key function Returns smallest among the arguments based on the return value of the key function

Example 1: Find minimum among the given numbers

In [28]:
```python
# using min(arg1, arg2, *args)
print('Minimum is:', min(1, 3, 2, 5, 4))
```
```
Minimum is: 1
```

In [27]:
```python
# using min(iterable)
num = [3, 2, 8, 5, 10, 6]
print('Minimum is:', min(num))
```
```
Minimum is: 2
```

In [10]:
```python
# using min(iterable)
num_list = []
#print('Minimum is:', min(num_list)) #ValueError: min() arg is an empty sequence
print('Minimum is:', min(num_list,default=5))
```
```
Minimum is: 5
```

Example 2: Find number whose sum of digits is smallest using key function

min(arg1, arg2, *args, key)

In [21]:
```python
def sumDigit(num):
    sum = 0
    while(num):
        sum += num % 10
        num = num // 10
    return sum

# using min(arg1, arg2, *args, key)
print('Minimum is:', min(100, 321, 267, 59, 40, key=sumDigit))
```

```
Minimum is: 100
```

using min(iterable,key)

In [23]:
```
def sumDigit(num):
    sum = 0
    while(num):
        sum += num % 10
        num = num // 10
    return sum

# using min(iterable, key)
num = [15, 300, 2700, 821, 52, 10, 6]
print('Minimum is:', min(num, key=sumDigit))
```

```
Minimum is: 10
```

Here, each element in the passed argument (list or argument) is passed to the same function sumDigit(). Based on the return value of the sumDigit(), i.e. sum of the digits, the smallest is returned.

Example 3: Find list with minimum length using key function

In [7]:
```
num = [15, 300, 2700, 821]
num1 = [12, 2]
num2 = [34, 567, 78]
# using min(iterable, *iterables, key)
print('Minimum is:', min(num, num1, num2, key=len))
```

```
Minimum is: [12, 2]
```

In this program, each iterable num, num1 and num2 is passed to the built-in method len(). Based on the result, i.e. length of each list, the list with minimum length is returned.

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/