SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Python’s for loop

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Loops

Python Iterables

- Iterable is an object capable of returning its members one by one

- Sequence type objects like lists, strings, tupes, dictionaries and sets are common type of iterables

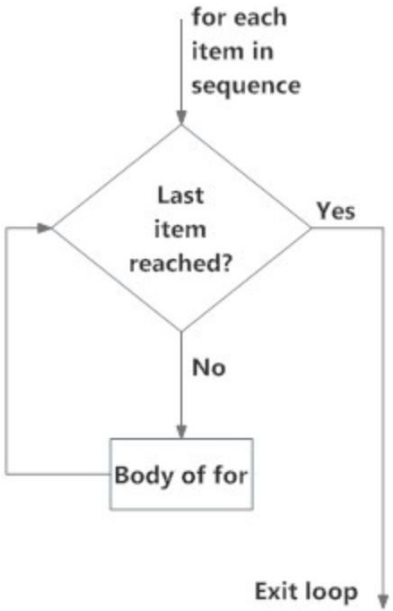
Iteration through the Iterables

- Iteration is a general term for taking each item from a sequence one after another

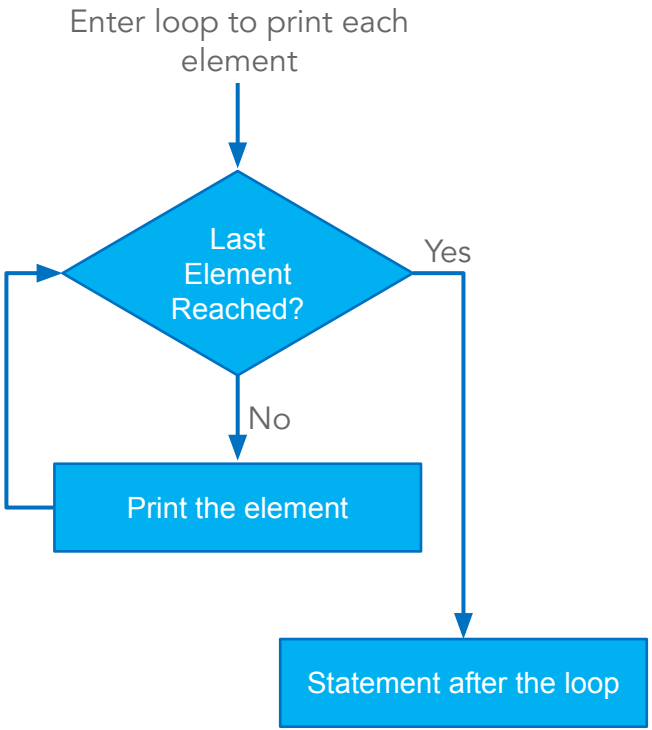
The for loop

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects.

Iterating over a sequence is called traversal.



The for loop code example



A list - sequence type object

A variable to hold a value from the list at each iteration

```
age = [10, 12, 15, 18, 20]

for each_age in age:
    print(each_age)
```

```
10
12
15
18
20
```

Read each element of a string

```
# read a string and print the elements one by one
nm = 'India'

for i in nm:
    print(i)
```

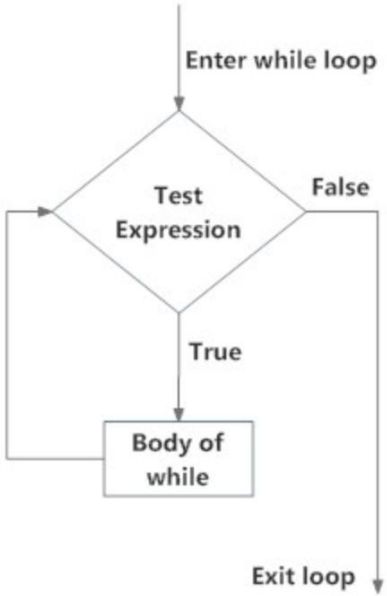
```
I
n
d
i
a
```

While Loop

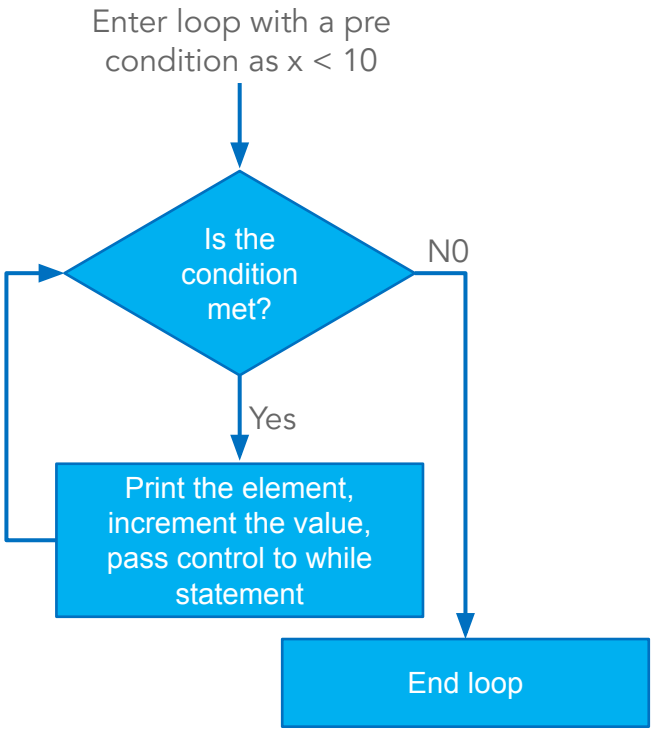
The while loop

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

We generally use this loop when we don't know beforehand, the number of times to iterate.



The while loop code example



The while statement checks the condition if x < 10

```
# Assigning 0 to x
x = 0

# While the loop condition is less than 10,
# keep printing x
# add 1 to x
while x < 10:
    print (x)
    x+=1
```

```
0
1
2
3
4
5
6
7
8
9
```

Increment x by 1 and pass the control to the while statement

Print each element of a list using while loop

```
#get the list and print the input one by one

cars = ['Maybach','Audi','BMW']
i = 0
while i < len(cars):
    print(cars[i])
    i+=1
```

```
Maybach
Audi
BMW
```

Break, Continue

Exiting a loop

Loops in Python allows us to automate and repeat tasks.

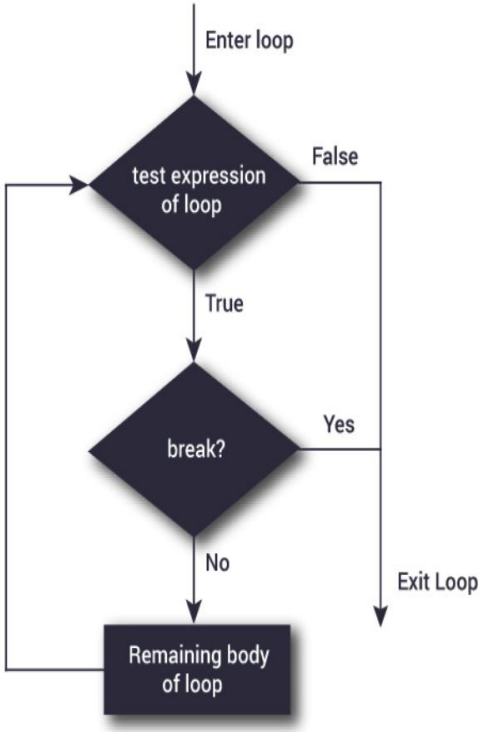
But at times, due to certain factors, we may want to:

- Exit a loop completely
- Skip part of a loop before continuing

Exit a loop with break statement

The break statement in Python terminates the current loop and resumes execution at the next statement

The break statement can be used in both while and for loops.



Exit a while loop with break statement

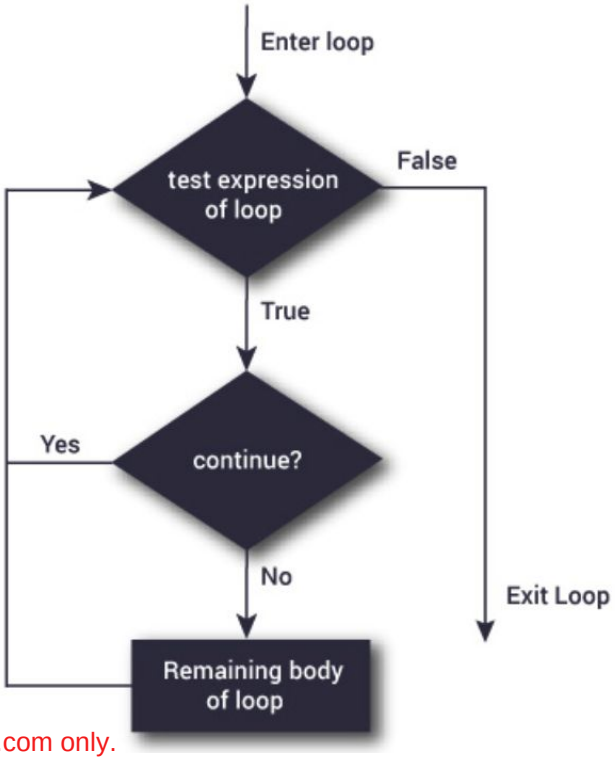
```
n = 5
while n > 0:
    n -= 1
    if n == 2:
        break
    print(n)
print('Loop ended.')
```

```
4
3
Loop ended.
```

Skip part of the loop with continue statement

The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

The continue statement can be used in both while and for loops.



Skip part of the loop with continue statement

The for...loop iterates through the string character by character

```
for i in "greatlearning":
    if i == "i":
        continue
    print(i)

print("The end of code")
```

The if condition checks for the condition, x = “i”

The continue statement, when meets the condition, it ignores that character and moves the control to the beginning of the loop

```
g
r
e
a
t
l
e
a
r
n
n
g
The end of code
```

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

More on Loops

Read each element of a tuple

```
# Read a tuple and display all the words one by one

a = ("Opportunities", "don't", "happen.", "You", "create", "them.")

for i in a:
    print(i)
```

```
Opportunities
don't
happen.
You
create
them.
```

Read through a list of list

```
list_of_lists = [['Learn', 'one', 'new', 'thing', 'everyday'],[101, 102, 103],[9.9, 7.7, 2.7]]

for list in list_of_lists:
    print(list)
```

```
['Learn', 'one', 'new', 'thing', 'everyday']
[101, 102, 103]
[9.9, 7.7, 2.7]
```

Loop through a range

```
#print the numbers from 1 till 5
for i in range(0, 5):
    print("value of i is :", i)
```

```
value of i is : 0
value of i is : 1
value of i is : 2
value of i is : 3
value of i is : 4
value of i is : 5
value of i is : 6
value of i is : 7
value of i is : 8
value of i is : 9
```

A range() function takes 3 parameters. Start, stop and step. If step is not provided a default step of 1 is considered.

```
#print the numbers from 1 till 6 with a step of 2
for i in range(0, 6, 2):
    print("value of i is :", i)
```

```
value of i is : 0
value of i is : 2
value of i is : 4
```

Loop through a range (negative step)

```
#print the numbers from 100 till 0 backard with a step of 10
for i in range(100, 0, -10):
    print(i)
```

```
100
90
80
70
60
50
40
30
20
10
```

A range() function takes 3 parameters. Start, stop and step. If step is not provided a default step of 1 is considered.

In this example we have a negatve step of -10. The range starts from 10. Goes backward to 0 by step 10

Computation with for…loop

```
# read each element from the range and add 1 to each of the values
for i in range(1, 10, 2):
    print(i, "Squared value is :" , i + 1)
```

```
1 Squared value is : 2
3 Squared value is : 4
5 Squared value is : 6
7 Squared value is : 8
9 Squared value is : 10
```

```
# read each element from the range and square the values
for i in range(1, 10, 2):
    print(i, "Squared value is :" , i*i)
```

```
1 Squared value is : 1
3 Squared value is : 9
5 Squared value is : 25
7 Squared value is : 49
9 Squared value is : 81
```

Read through a dictionary object

```
employee_dict = {'Name': 'Rajesh', 'Job': 'Data Scientist', 'Age': 27}
employee_dict
```

```
{'Name': 'Rajesh', 'Job': 'Data Scientist', 'Age': 27}
```

```
## print all the items
for each_item in employee_dict.items():
    print(each_item)
```

```
('Name', 'Rajesh')
('Job', 'Data Scientist')
('Age', 27)
```

```
## print all the keys
for item in employee_dict.keys():
    print(item)
```

```
Name
Job
Age
```

```
## print all the values
for item in employee_dict.values():
    print(item)
```

```
Rajesh
Data Scientist
27
```

The dict.items() returns all key-value pair

The dict.keys() returns all keys

The dict.values() return all values

Read through the values in a dictionary to create a list

Create a blank list

```
# Get all the values of a dictionary obejct and display the output as list
employee_dict = {'Name': 'Rajesh', 'Job': 'Data Scientist', 'Age': 27}
new_list = []

for item in employee_dict.values():
    new_list.append(item)

print(new_list)
```

```
['Rajesh', 'Data Scientist', 27]
```

Append new value to the list

Replace values in a dictionary with computed values

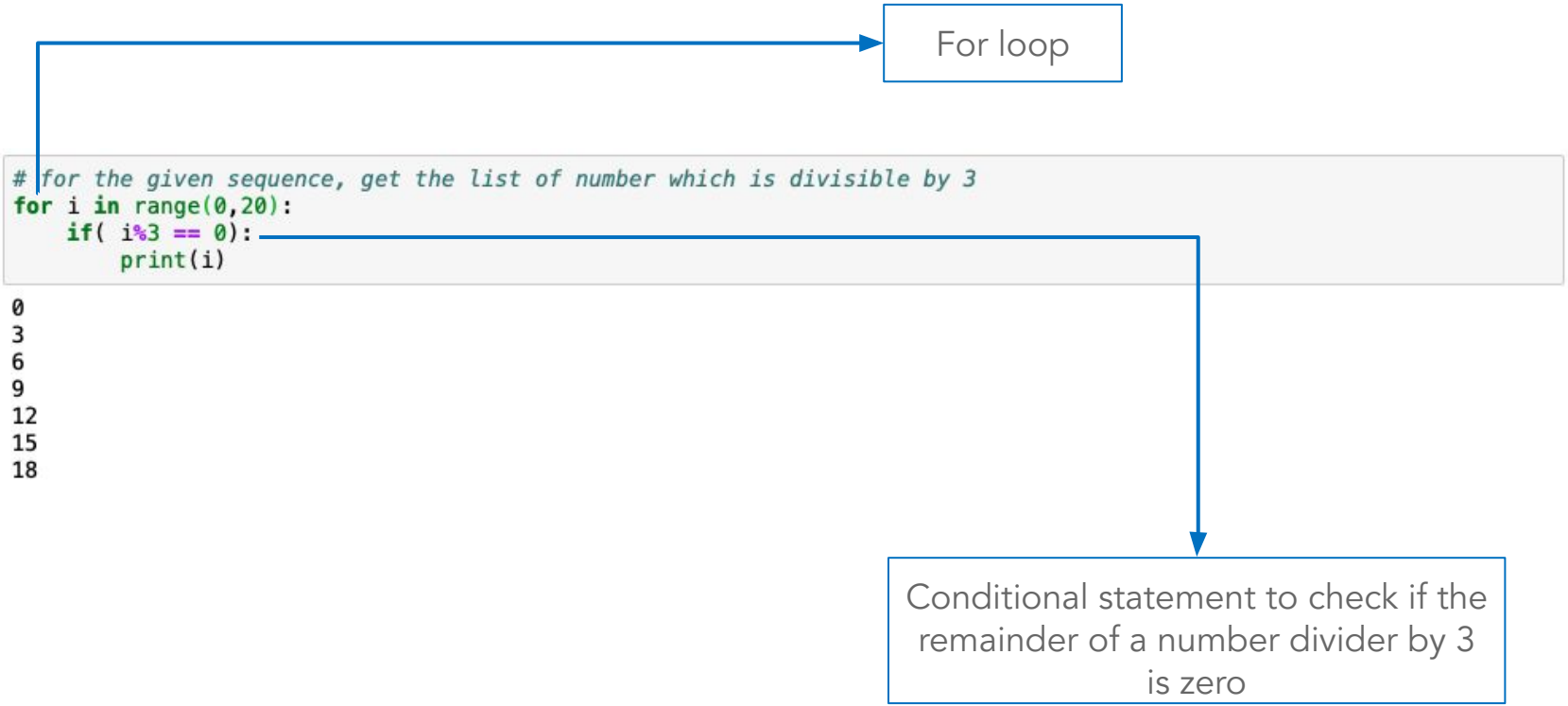
A dictionary object

```
# create a dict key - fruit name, value is price
prices = {'MacBook Pro': 320000, 'iPad': 90000, 'iPhone': 127000}

# Get the key and values for each items
for each_key, each_value in prices.items():

    #for each key value, get its respective value and applies a 10% discount
    prices[each_key] = round(each_value * 0.9, 2)

    prices
```

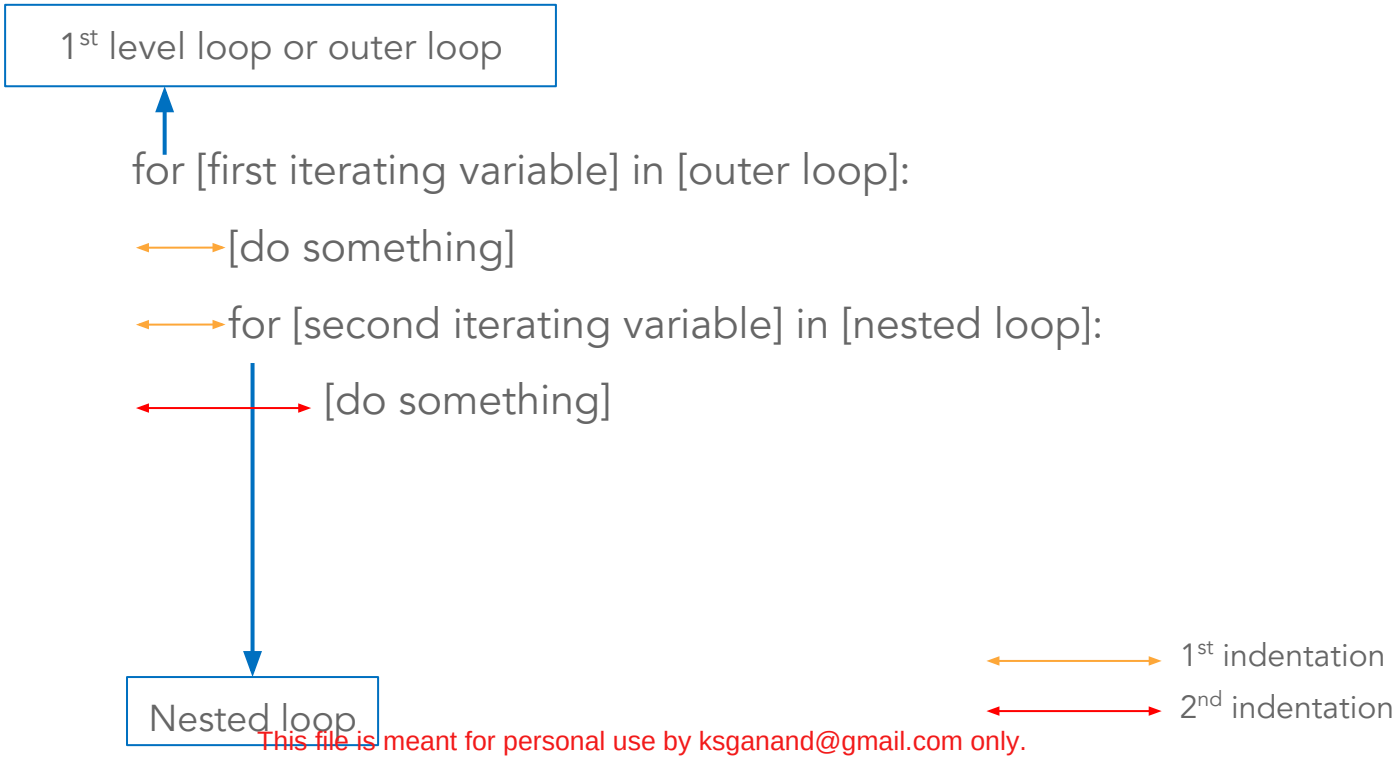
```
{'MacBook Pro': 288000.0, 'iPad': 81000.0, 'iPhone': 114300.0}
```

Appplies a 10* discount on the values

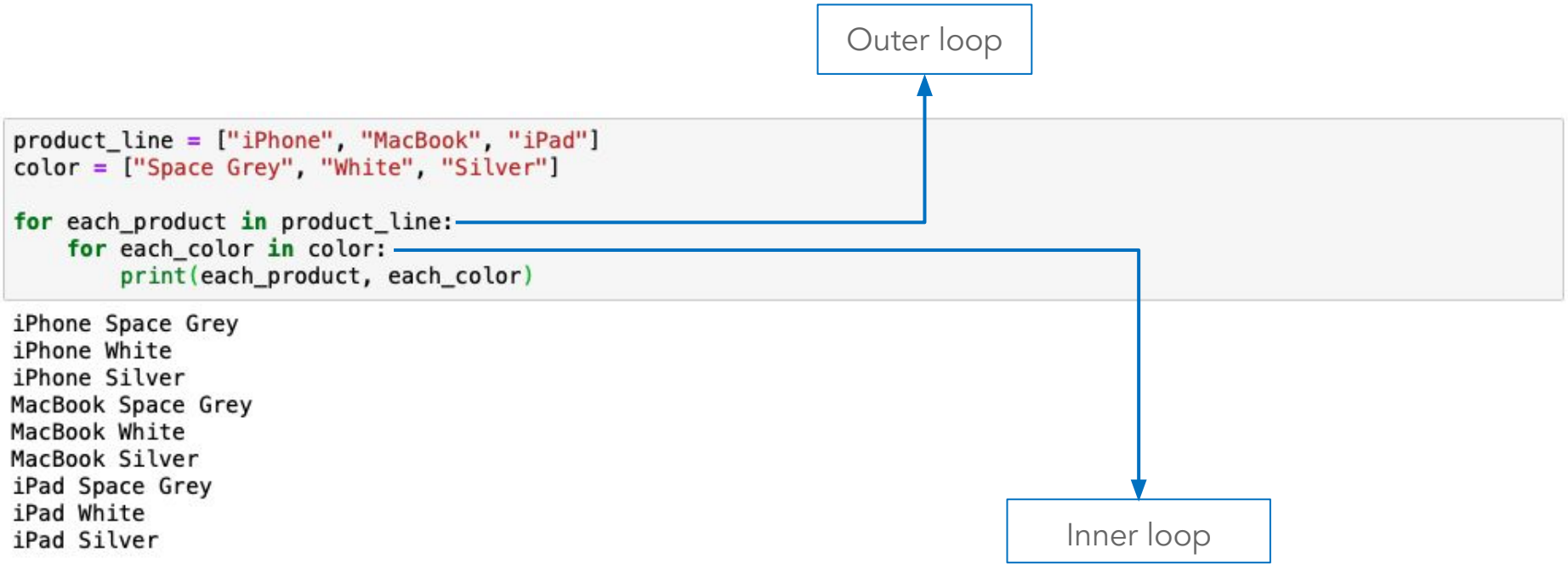
The for loop with conditional statements



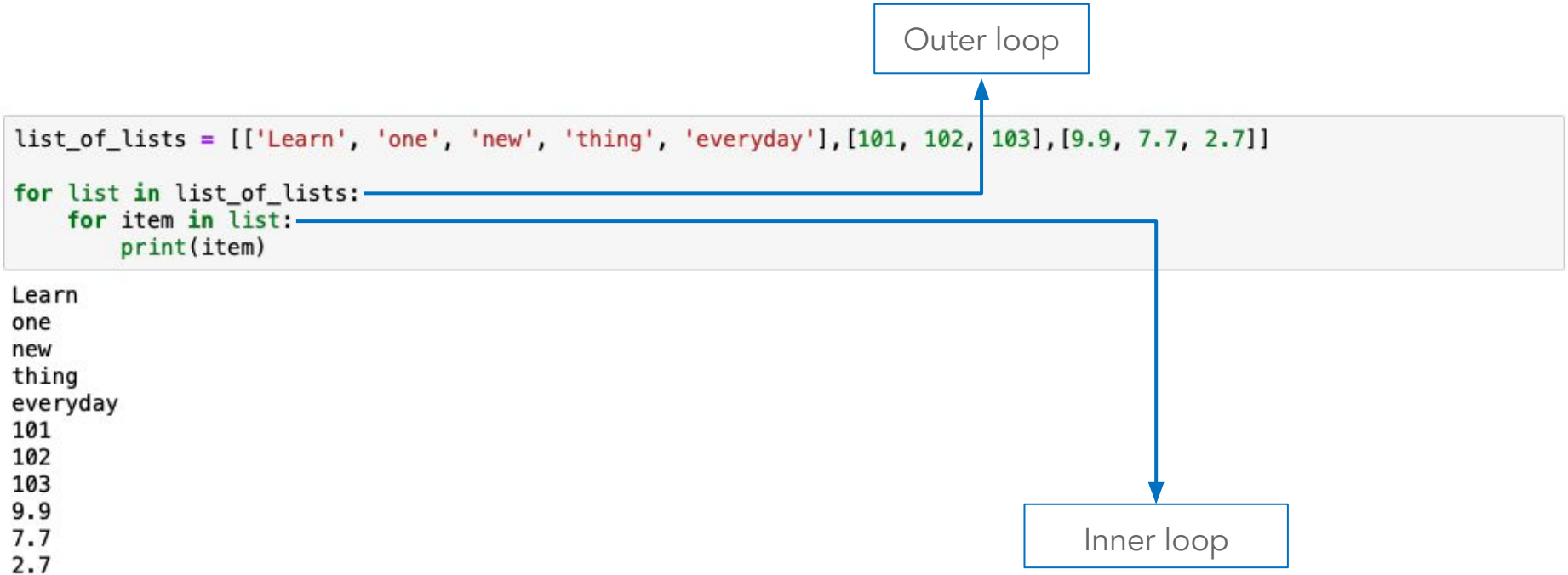
Nested for loop



Nested for loop



Nested for loop with list of lists



Example: Calculating sum of numbers

```
n = int(input("enter the count of sequential nummbers you want to add :"))

# initialize sum and counter
sum = 0
i = 1

while i <= n:
    sum = sum + i
    i = i+1    # update counter

    # print the sum
    print("The sum is", sum)
```

```
enter the count of sequential nummbers you want to add :5
The sum is 1
The sum is 3
The sum is 6
The sum is 10
The sum is 15
```

While loop with else statement

The else part is executed if the condition in the while loop evaluates to False.

```
# initialize  counter
counter = 0

while counter < 3:
    print("Inside loop")
    counter = counter + 1
else:
    print("Inside else") # as the count reached 3, it exits out of the loop
```

```
Inside loop
Inside loop
Inside loop
Inside else
```

Exit a while loop with continue statement

```
n = 5
while n > 0:
    n -= 1
    if n == 2:
        continue
    print(n)
print('Loop ended.')
```

```
4
3
1
0
Loop ended.
```

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

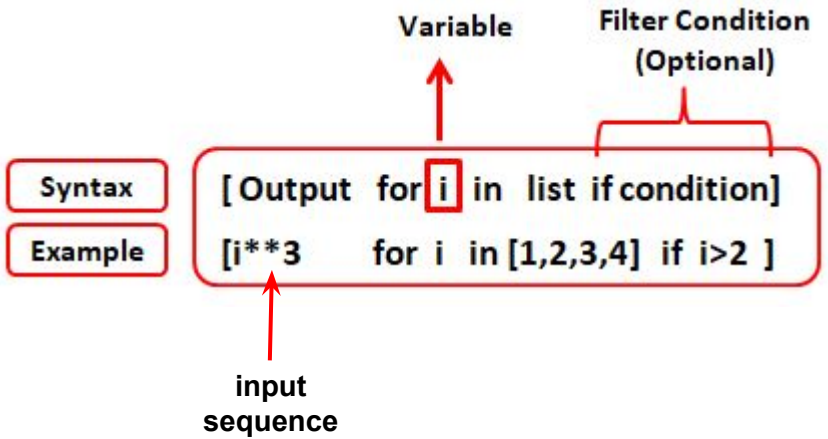
List Comprehension

What is a list comprehension?

- List comprehensions provide a concise way to create lists
- It consists of [ ] containing:
  - an input sequence
  - a variable representing numbers of the input sequence
  - a conditional expression (optional)
  - an output expression producing a list
- The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it. The list comprehension always returns a result list

What is a list comprehension?

- It consists of [ ] containing:
  - an input sequence
  - a variable representing numbers of the input sequence
  - a conditional expression (optional)
  - an output expression producing a list



List comprehension vs loops

Create a list using for loop

```
num_square = []
for i in range(5):
    num_square.append(i*i)
num_square
```

```
[0, 1, 4, 9, 16]
```

```
num_squares = [i * i for i in range(5)]
num_squares
```

```
[0, 1, 4, 9, 16]
```

Create a list using list comprehension

List comprehension reduces 3 lines of code into one, which will be instantly recognizable to anyone.

It is faster, as Python will allocate the list's memory first, before appending the elements to it, instead of having to resize on runtime.

For loop using list comprehension

```
num = [1,2,3,4]
doubled_num = [n * 2 for n in num]
doubled_num
```

```
[2, 4, 6, 8]
```

Multiply each element of the list by 2 using list comprehension

```
my_list = [1,2,2]
multiplied = [item*2 for item in my_list]
print (multiplied)
```

```
[2, 4, 4]
```

Read the first letter of each word using list comprehension

```
words = ["python", "for", "data", "science"]
items = [ word[0] for word in words ]
print(items)
```

```
['p', 'f', 'd', 's']
```

Iteration over more than one iterable in a list using list comprehension

```
seq_1 = [1, 2, 3]
seq_2 = 'ABC'
[(x,y) for x in seq_1 for y in seq_2]
```

```
[(1, 'A'),
 (1, 'B'),
 (1, 'C'),
 (2, 'A'),
 (2, 'B'),
 (2, 'C'),
 (3, 'A'),
 (3, 'B'),
 (3, 'C')]
```

If statement within a list comprehension

```
my_color = ['Orange', 'Yellow', 'Blue', 'Red', 'Green']
green_list = [color for color in my_color if color == 'Orange']
green_list
```

```
['Orange']
```

Extract all the numbers from the string using list comprehension

```
my_string = "Hello 12345 Python"
num = [x for x in my_string if x.isdigit()]
print(num)
```

```
['1', '2', '3', '4', '5']
```

Extract all the vowels from the string using list comprehension

```
sentence = 'PyThoN FoR dAtA SciEnCe'
vowels = [i for i in sentence if i in 'aeiouAEIOU']
vowels
```

```
['o', 'o', 'A', 'A', 'i', 'E', 'e']
```

Nested condition within a list comprehension

```
my_color = ['Orange', 'Yellow', 'Blue', 'Red', 'Green']
color_indicator = [0 if color == 'Green'else 1 if color == 'Red' else 2 if color == 'Blue' else 3 for color in my_color]
print(my_color)
print(color_indicator)
```

```
['Orange', 'Yellow', 'Blue', 'Red', 'Green']
[3, 3, 2, 1, 0]
```

Find colors in list that are longer than 4 character using list comprehension

```
my_color = ['Orange', 'Yellow', 'Blue', 'Red', 'Green']
long_words = [color for color in my_color if len(color) > 4]
long_words
```

```
['Orange', 'Yellow', 'Green']
```

Find colors with a length between 3 and 6 using list comprehension


```
my_color = ['Orange', 'Yellow', 'Blue', 'Red', 'Green']
color_length = [color for color in my_color if len(color) > 3 and len(color) < 6]
color_length
```

```
['Blue', 'Green']
```

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

User-Defined Functions

Built-in functions

Built-in functions are those that are already defined in the Python library

User-defined functions

A function that you define yourself in a program is known as user defined function.

You can give any name to a user defined function.

You cannot use the Python keywords as function name.

The def keyword

In python, we define the user-defined function using def keyword, followed by the function name.

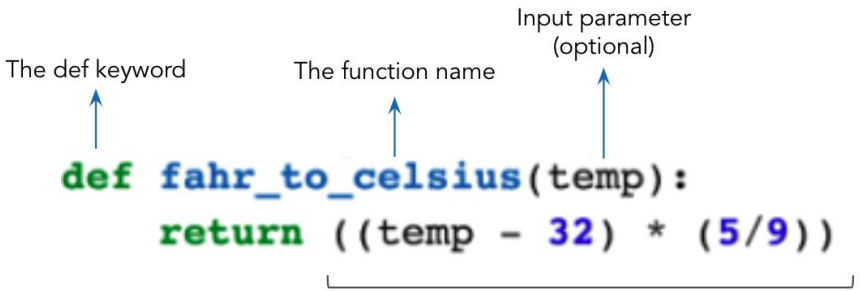
Defining the user-defined function

Step 1: Declare the function with the keyword def followed by the function name

Step 2: Write the arguments inside the opening and closing parentheses of the function, and end the declaration with a colon

Step 3: Add the program statements to be executed

Step 4: End the function with/without return statement



Write your first function using the def keyword

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

```
# Create first function to display Hello world.

def helloworld():
    print("Hellooo world")
```

```
# call the function
helloworld()
```

```
Hellooo world
```

Write a function with an argument

```
## Function  - Create a function and pass input variable
## pass variable to the function
def hello(nm):
    print("Hello ",nm)
```

```
## call the function
hello("Eddyy")
```

```
Hello  Eddyy
```

Calling the function without passing argument

When we declare a function that expects input argument, we should pass the required value. If we do not pass the required value, the function will throw an error.

```
## Function  - Create a function and pass input variable
## pass variable to the function
def hello(nm):
    print("Hello ",nm)
```

```
## call the function without passing the argument
hello()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-82e586c80250> in <module>
      1 ## call the function without passing the argument
----> 2 hello()

TypeError: hello() missing 1 required positional argument: 'nm'
```

A function without any return value

Note: We use the return keyword in the function but we do not mention what to return. Hence the function returns no value

```
# Functuion without return value
def empty_return(x,y):
    c = x + y
    return
```

```
result = empty_return(4,5)
print(result)
```

```
None
```

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Function Arguments

Types of function arguments

1. Required Arguments
2. Keyword Arguments
3. Default Arguments
4. Variable-Length Arguments

Required Arguments

In this case, if the argument is not passed it will throw an error

```
## Function  - Create a function and pass input variable
## pass variable to the function
def hello(nm):
    print("Hello ",nm)
```

```
## call the function without passing the argument
hello()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-82e586c80250> in <module>
      1 ## call the function without passing the argument
----> 2 hello()

TypeError: hello() missing 1 required positional argument: 'nm'
```

Keyword Arguments

The arguments have names assigned to them. In the below example, we have Name & Designation as the named parameters. We pass ‘John’ as Name and ‘CEO’ as Designation.

```
# pass the argument and change in position of the argument
def employee(Name, Designation):
     print(Name, Designation)
```

```
# Keyword arguments
employee(Name ='John', Designation ='CEO')

employee(Designation ='CEO', Name ='John')
```

```
John CEO
John CEO
```

Keyword Arguments

Note: Even if the wrong values are passed, it will NOT throw an error. For example, if we say ‘CEO’ as Name instead of ‘John’, the function will still work but with wrong values

```
# pass the argument and change in position of the argument
def employee(Name, Designation):
     print(Name, Designation)
```

```
# even if the wrong values are passed, it will runw ithout any error
employee(Name ='CEO', Designation ='John')
```

```
CEO John
```

Default Arguments

Note: The function expects 2 arguments - Name and Salary. We have passed a default value for salary. When we call the function, we pass only Name but not Salary. In this case, it will consider the default value for Salary that has been passed when the function was defined.

```
def employee(Name, Salary = 40000 ):

    print("Employee Name: ", Name)
    print("Employee Salary ", Salary)
    return;
```

```
employee( "Paul" )
```

```
Employee Name:  Paul
Employee Salary  40000
```

Variable-length arguments using *arg keyword

SRM INSTITUTE OF SCIENCE & TECHNOLOGY (Deemed to be University u/s 3 of UGC Act, 1956)

This helps you in passing variable number of arguments. This is especially helpful when you do not know how many arguments to pass to the function.

```
# read the value one by one and prints the value
# *args in function definitions in python is used to pass a variable number of arguments to a function
# symbol * to take in a variable number of arguments

def daily_temperature(*temp):
    for var in temp:
        print(var)
```

```
daily_temperature(10, 20, 30, 14)
```

```
10
20
30
14
```

Variable-length keyworded arguments using **kwargs

**kwargs allows you to pass keyworded variable length of arguments to a function.

You should use **kwargs if you want to handle named arguments in a function.

```
def my_function(**krgs):
    print(type(krgs))
    for k,v in krgs.items():
        print(k,"==", v)
```

```
my_function(firstname = "John", secondname = "Allen", salary = 20000, pf=345.75, goodperformer=True)
```

```
<class 'dict'>
firstname == John
secondname == Allen
salary == 20000
pf == 345.75
goodperformer == True
```

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Variable Scope

Variable scope

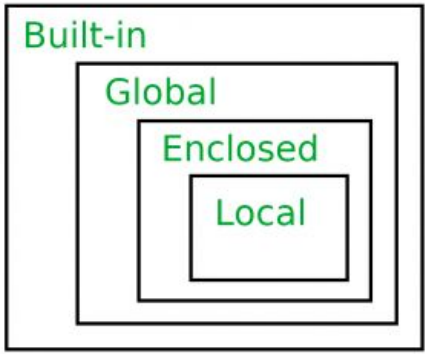
- A namespace is a container where names are mapped to objects (variables)
- A scope defined the hierarchical order in which the namespaces have to be searched in order to obtain the mapping name-to-object (variables)
- Scope defined the accessibility and lifetime of a variable

The LEGB rule

In Python, the LEGB rule is used to decide the order in which the namespaces are to be searched for scope resolution.

Variable scope hierarchy:

1. Built-In (B): Reserved names in Python
2. Global Variable (G): Defined at the uppermost level
3. Enclosed (E): Defined inside enclosing or nested functions
4. Local Variable (L): Defined inside a function



Local Scope

Local scope refers to variables defined in current function. A function will first look up for a variable name in its local scope. If it does not find it there, only then the outer scopes are searched for.

```
# Global variable can be placed at the top or above the function call
# A function will first look up for a variable name in its local scope
x = "global"

def local_scope_example():
    x = "local"
    print("x inside :", x)
```

```
local_scope_example()
```

```
x inside : local
```

Global Scope

If a variable is not defined in local scope, then, it is checked for in the higher scope, in this case, the global scope.

```
# Global variable can be placed at the top or above the function call
# A function will first look up for a variable name in its local scope
x = "global"

def global_scope_example():
    x = "local"
    print("x inside :", x)
```

```
# Local scope output
global_scope_example()
```

```
x inside : local
```

```
# Global scope output
print(x)
```

```
global
```

Enclosed Scope

For the enclosed scope, we need to define an outer function enclosing the inner function. Refer to the variable using the nonlocal keyword.

```
x = 'This has global scope'

def outer():
    x = 'outer x variable: enclosed'
    def inner():
        nonlocal x
        print(x)
    inner()
```

```
outer()
```

```
outer x variable: enclosed
```

Built-In Scope

If we have not defined a variable and the name of the variable matches with a built-in function from an existing Python module, the function will use the built-in function.

```
# Built-in Scope
from math import pi

def outer():
    def inner():
        print(pi)
    inner()
```

```
outer()
```

```
3.141592653589793
```

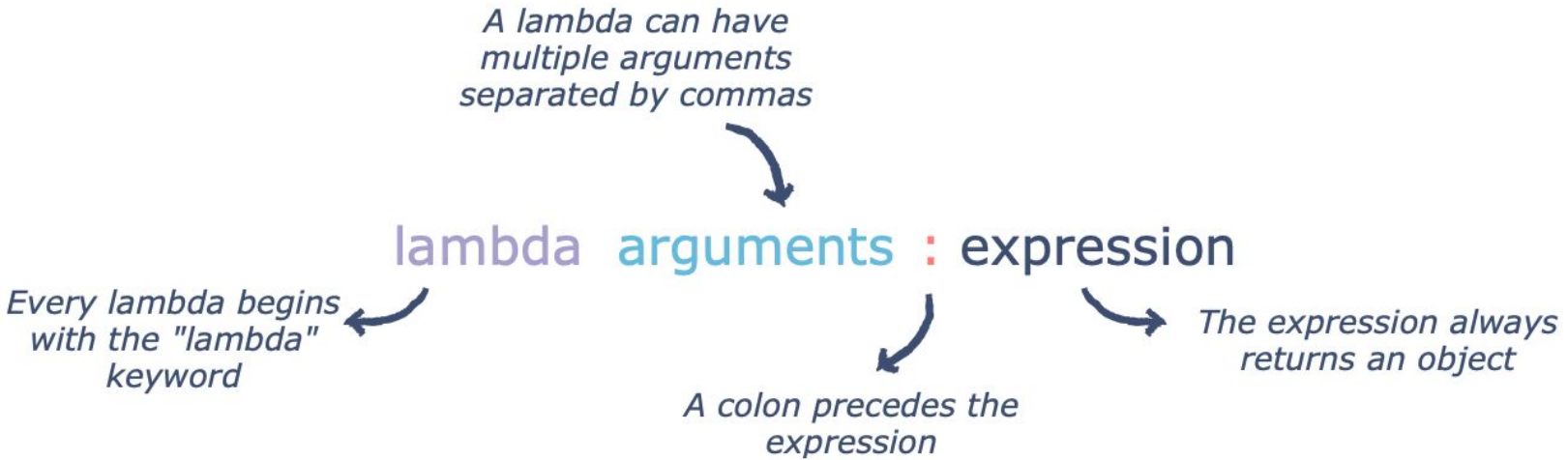
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Lambda Function

Lambda function

- Lambda functions are anonymous, i.e. to say they have no names
- The lambda is a keyword
- It is a simple one-line function
- No def or return keyword to be used with a lambda function

The structure of the lambda function



Using lambda function to reduce code size

Normal Python Code

```
def fun(x,y):
    if(x>y):
        return x
    else:
        return y
print(fun(3,4))
```

```
4
```

Using lambda function

```
# The general method shown on the
# left can also be rewritten
# using lambda
fun = lambda x,y:x if x>y else y
print(fun(3,4))
```

```
4
```

Wrong usage of lambda

You will need to declare the variable which you may use inside the lambda function. In the below example, we use variable, b, without declaring it.

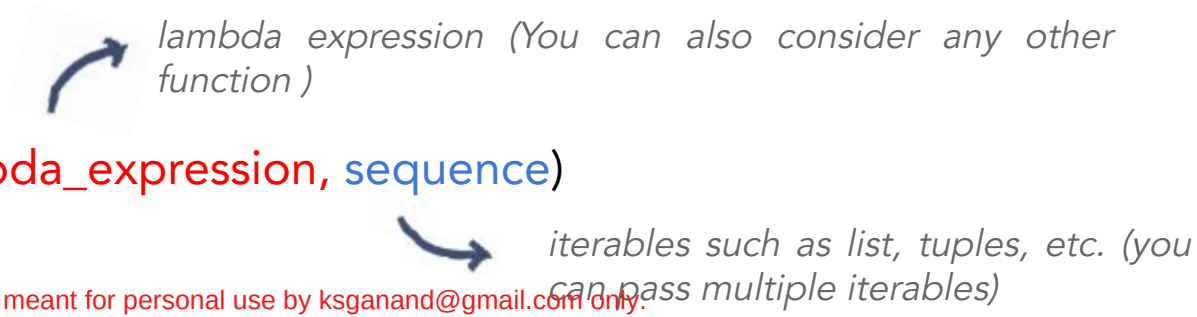
```
# multiplication - wrong usage of variable b
x = lambda a, c : a * b
print(x(5, 6))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-66-74ba1647dffe> in <module>
      1 # multiplication - wrong usage of variable b
      2 x = lambda a, c : a * b
----> 3 print(x(5, 6))

<ipython-input-66-74ba1647dffe> in <lambda>(a, c)
      1 # multiplication - wrong usage of variable b
----> 2 x = lambda a, c : a * b
      3 print(x(5, 6))

NameError: name 'b' is not defined
```

The lambda() with map()

map() functions expect a function_object, in our case a lambda function,and a sequence (iterables, such as list, dictionary, etc.)

It executes the function_object for each element in the sequence and returns a sequence of the elements modified by the function object.



The lambda() with map()

The output is often type-casted into a seq type, as follow:

```
sample_list = [1, 2, 3, 4]
seq = list(map(lambda x : x*2, sample_list))
seq
```

```
[2, 4, 6, 8]
```

The lambda() with map()

You can pass multiple sequences to the map function as follow:

```
sample_list = [1, 2, 3, 4]
sample_list2 = [5,6,7,8,9]
sample_tuple = (10,11,12,13)
seq = list(map(lambda x : x*2, (sample_list, sample_list2, sample_tuple)))
seq
```

```
[[1, 2, 3, 4, 1, 2, 3, 4],
 [5, 6, 7, 8, 9, 5, 6, 7, 8, 9],
 (10, 11, 12, 13, 10, 11, 12, 13)]
```

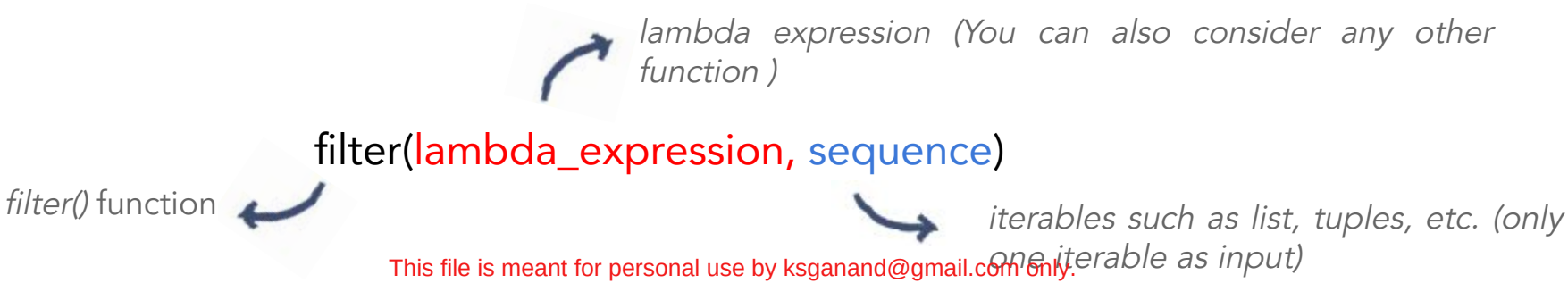
map(f, li1, li2, li3, …) → applies f to all lists in parallel. That is, first element of result would be f(e1,e2,e3) where its args are first element of each list.

The lambda() with filter()

The *filter()* function expects two arguments: *function_object(lambda)* and an iterable.

function_object returns a boolean value and is called for each element of the iterable.

It returns only those elements for which the function_object returns *true*.



The lambda() with filter()

The output is often type-casted into a seq type, as follow:

```
num_list = list(range(15))
seq = list(filter(lambda x : x % 3 == 0, num_list))
seq
```

```
[0, 3, 6, 9, 12]
```

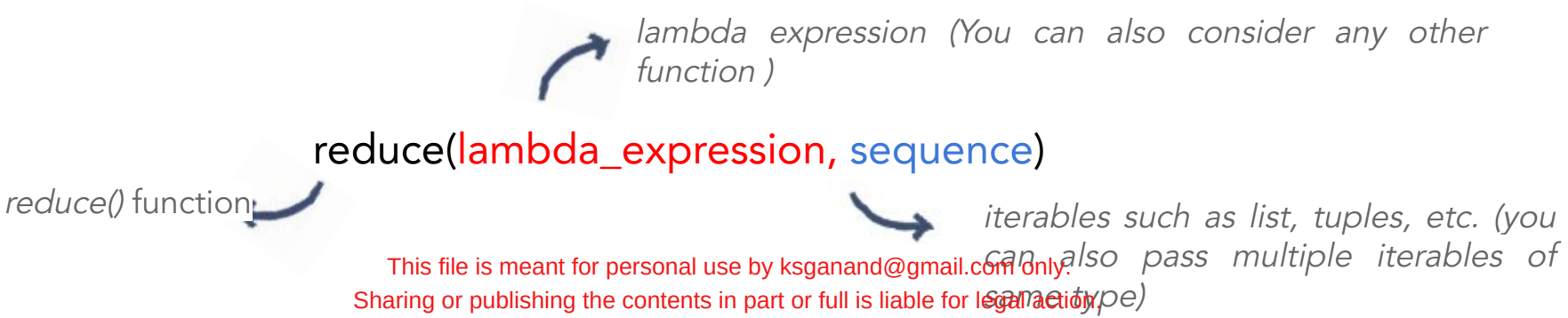
Unlike *map(), the filter()* function can only have one iterable as input.

The lambda() with reduce()

The reduce() function in Python takes in a function and a sequence as argument.

The function is called with a lambda function and a seq and a new reduced result is returned.

This performs a repetitive operation over the pairs of the seq.



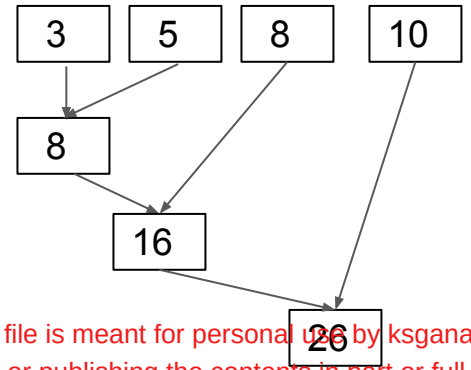
The lambda() with reduce()

Determining the summation of all elements of a list of numerical values by using reduce:

```
from functools import reduce
reduce(lambda a,b: a+b,[3,5,8,10])
```

```
26
```

Working:



The lambda() with reduce()

Determining the maximum of a list of numerical values by using reduce:

```
from functools import reduce
num_tuple = (1, 0, 2, -1, 5, 6, 10, -5)
reduce(lambda x,y: x if (x>y) else y, num_tuple)
```

```
10
```

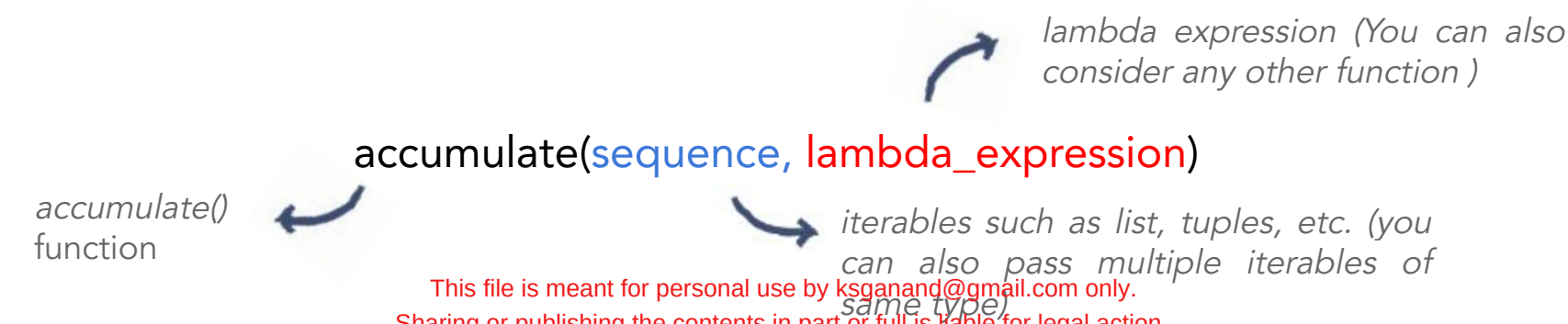
Note : reduce() can only have iterables of same type as input.

The lambda() with accumulate()

The accumulate() function in Python takes in a function and a sequence as argument.

The function is called with a lambda function and a seq and a new reduced result is returned.

Unlike reduce(), it returns a sequence containing the intermediate results



The lambda() with accumulate()

Determining the summation of all elements of a list of numerical values by using accumulate:

```
from itertools import accumulate
num_seq = list(range(10))
tuple(accumulate(num_seq, lambda x,y : x+y))
```

```
(0, 1, 3, 6, 10, 15, 21, 28, 36, 45)
```

Difference between reduce() and accumulate()

reduce()	accumulate()
The reduce() stores the intermediate result and only returns the final summation value	The accumulate() returns a list containing the intermediate results. The last number of the list returned is summation value of the list
The reduce(fun,seq) takes function as 1st and sequence as 2nd argument	The accumulate(seq,fun) takes sequence as 1st argument and function as 2nd argument
The reduce() is defined in “functools” module	The accumulate() is defined in “itertools” module

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Recursive Functions

Recursive Function

A recursive function is a function defined in terms of itself via self-referential expressions.

The function will continue to call itself and repeat its behavior until some condition is met to return a result.

All recursive functions share a common structure made up of two parts: base case and recursive case.

Recursive Function

For example:

```
def sum_of_n(n):
    #Base Case
    if n==1:
        return 1

    #Recursive case
    res = n + sum_of_n(n-1)
    return res
```

```
sum_of_n(5)
```

```
15
```

A base case is a case, where the problem can be solved without further recursion.

Recursive Function

A recursive function has to fulfil an important condition to be used in a program: *it has to terminate.*

A recursive function terminates, if with every recursive call the solution of the problem is downsized and moves towards a base case.

A recursion can end up in an infinite loop, if the base case is not met in the calls.

Recursive Function

Let us track how the previously defined recursive function, sum_of_n works by adding two print functions:

```
def sum_of_n(n):
    #Base Case
    print("sum_of_n has been called with n = " + str(n))
    if n==1:
        return 1

    #Recursive case
    res = n + sum_of_n(n-1)
    print("intermediate result for ", n, " + sum_of_n(" ,n-1, "): ",res)
    return res
```

```
sum_of_n(5)
```

```
sum_of_n has been called with n = 5
sum_of_n has been called with n = 4
sum_of_n has been called with n = 3
sum_of_n has been called with n = 2
sum_of_n has been called with n = 1
intermediate result for  2  + sum_of_n( 1 ):  3
intermediate result for  3  + sum_of_n( 2 ):  6
intermediate result for  4  + sum_of_n( 3 ):  10
intermediate result for  5  + sum_of_n( 4 ):  15
15
```

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Thank You