UNIT-3

Functions In Python: You use functions in programming to bundle a set of instructions that you want to use repeatedly or that, because of their complexity, are better self-contained in a sub-program and called when needed. That means that a function is a piece of code written to carry out a specified task. To carry out that specific task, the function might or might not need multiple inputs. When the task is carried out, the function can or can not return one or more values.

There are three types of functions in Python:

- **Built-in functions**: Such as help() to ask for help, min() to get minimum value, print() to print an object to the terminal. join, chr, ord, id() [unique integer]
- **User-Defined Functions**: which are functions that users create to help them out.
- **Anonymous functions**: which are also called lambda functions because they are not declared with the standard def keyword.

★ All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

Syntax of Function:

```
def function_name(parameters):
    statement(s)
```

Above shown is a function definition which consists of following components.

1. Keyword def marks the start of function header.
2. A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
7. An optional return statement to return a value from the function.

Elements of Functions: Every function has the following elements:

1. Function Declaration (Function Prototype)
2. Function Definition
3. Function Call

FUNCTION ARGUMENTS:

You can call a function by using the following types of formal arguments -

1) Requirement / Required Arugements
2) Keyword Arguments
3) Default Arguments
4) Variable-length Arguments

1) Required Arguments:

Required Arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function Definition. To call the function printme(), you definitely need to pass one argument, otherwise it gives a syntax error as follows:-

Example:

```
def printme(str):
    print(str)
    return

printme()
```

When the above code is executed, it produces the following result -

TypeError: printme() takes exactly 1 argument.

2) KEYWORD ARGUMENTS:

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the printme() function in the following ways –

Example:

```
def printme (str):
    print (str)
    return
printme (str="My String")
```

When the above code is executed, it produces the following result is showed:

```
My String
```

The following example gives a clearer picture. Note that the order of parameters does not matter.

```
def printinfo (name, age):
    print ("Name:", name)
    print ("Age", age)
    return
printinfo (age=50, name="miki")
```

The above code is executed, and it produces the following result -

```
Name : miki
Age : 50
```

3) Default Arguments: A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed -

```
def printinfo (name, age = 35):
    print ("Name:", name)
    print ("Age:", age)
    return
printinfo(age = 50, name = "miki")
printinfo (name = "miki")
```

When the above code is executed, it produces the following result -

```
Name : miki
Age : 50
Name : miki
Age : 35
```

We can provide a default value to an argument by using the assignment operator (=).

4) Variable-length Arguments: You may need to process a function for more arguments than you specified while defining the function. These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is given below:-

```
def functionname([formal_args,] *var_args_tuple):
    function_suite
    return [expression]
```

An asterik (*) is placed before the variable name that holds the values of all nonkeyword variable argument. This tuple remains empty if no additional arguments are specified during the function call.

Example:

```
def printinfo(arg1, *vartuple):
    print("Output is:")
    print(arg1)
    for var in vartuple:
        print(var)
    return
printinfo(10)
printinfo(70, 60, 50)
```

When the above code is executed, it produces the following result -

```
Output is :
10
Output is :
70
60
50
```

The Anonymous FUNCTIONS: These functions are called anonymous because they are not declared in the standard manner by using the def keyword. You can use the lambda keyword to create small anonymous functions.

- a) lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- b) An anonymous function cannot be a direct call to print because lambda requires an expression.
- c) Lambda functions have their own local name-space and cannot access variables other than those in their parameters list and those in the global namespace.

d) Although it appears that lambdas are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is to stack allocation by passing function, during invocation for performance reasons.

Syntax: The Syntax of lambda functions contains only a single statement, which is as follows -

lambda [arg1 [, arg2, ---argn]] : expression

Example:

```
Sum = lambda arg1, arg2 : arg1 + arg2

print ("Value of total : ", sum (10,20))
print ("Value of total : ", sum (20,20))
```

When the above code is executed, it produces the following result -

Value of total = 30
Value of total = 40

SCOPE RULES:

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python -

a) Global Variables.
b) Local variables.

- The variable defined outside any function is known to have a global scope whereas the variable defined inside a function is known to have a local scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call

- a function, the variables declared inside it are brought into scope.

Example:-

```
total = 0 # This is global variable.
def Sum(arg1, arg2):
    # Here total is local variable.
    total = arg1 + arg2.
    print("Inside the function local total:", total)
    return total

# Now you can call Sum function
Sum(10, 20)
print("Outside the function global total:", total)
```

When the above code is executed, it produces the following result –

```
Inside the function local total : 30
Outside the function global total : 0
```

Namespaces:

What are Names in Python?

Before getting on to namespaces, first, let's understand what python means by a name.

A name in Python is just a way to access a variable like in any other languages. However, Python is more flexible when it comes to the variable declaration. You can declare a variable by just assigning a name to it.

The naming mechanism works inline with Python's object system i.e everything in Python is an object. All the data types such as numbers, strings, functions, classes are all objects. And a name acts as a reference to get to the objects.

What are Namespaces in Python?

A namespace is a simple system to control the names in a program. It ensures that names are unique and won't lead to any conflict.

Also, add to your knowledge that Python implements namespaces in the form of dictionaries. It maintains a name-to-object mapping where names act as keys and the objects as values. Multiple namespaces may have the same name but pointing to a different variable.

i) Local Namespaces: This Namespaces covers the local names inside a function. Python creates this namespace for every function called in a program. It remains active until the function returns.

ii) Global Namespaces: This Namespaces covers the names from various imported modules used in a project. Python creates this namespace for every module included in your program. It will last until the program ends.

iii) Built-in Namespaces: This Namespace covers the built-in functions and built-in exception names. Python creates it as the interpreter starts and keeps it until you exit.

BODMAS Example.

SCOPE IN PYTHON: Namespaces make our programs immune from name conflicts. However it doesn't give us a free ride to use a variable name anywhere we want Python restricts names to be

bound by specific rules known as a scope. The scope determines the parts of the program where you could use that name without any prefix.

a) Python outlines diffrent scopes for locals, functions, modules, and built-ins.

b) A local scope, also known as the innermost scope, holds the list of all local names available in the current location.

c) A scope for all the enclosing functions, it finds a name from the nearst enclosing scope and goes outwards.

d) A module level scope, it takes care of all the global names from the current module.

e) The outermost scope which manages the list of all the built-in names. It is the last place to search for a name that you cited in the program.

We call the scope rules as: LEGB

1. Local
2. Enclosing
3. Global
4. Built-in

In this search order matters: first search local, then Enclosing, Global, and Built-in

1) Local Scope:

- Always search local scope first
- local scope refers to names assigned in any way within a function that are not declared as global.
- That is, all names local to a function.

Example:

```
1.  x = 99
2.  y = 17
3.  def fun(x):
4.      y=100
5.      print (x, y)
6.  fun(77)
7.  print (x,y)
```

Output:

```
77  100
99  17
```

Explanation: 1) local (to a function) scope is searched first.

2) x: for the print statement on line 5, the parameter to the function, line 3, is the value of x that will be found first.

3) y: for the print statement on line 5, the definition of y on line 4 is found first, because that definition of y is local to function.

4) Print on line 7: Python cannot look inside of functions, so there is no local scope at line 7,

only Global; thus the definitions on line 1, 2 will be used for x and y.

2) **Enclosing Scope:**

a) Python permits functions to be nested.

b) When searching for a variable, first search local, and then search Enclosing scopes.

c) To find x on line 5: first search bar, then fun, where the parameter is found

d) To find y on line 5: first search bar, then fun; y found on line 3.

Example:

```
1. x = 99
2. def fun(x):
3.     y = 100
4.     def bar():
5.         print(x,y)
6.     bar()
7. fun(77)
```

Run

Output: -77 100

3) **Global Scope:**

a) Global scope is searched after Local, and Enclosed.

b) Global scope is simplest to understand

c) A name declared at global scope, is not enclosed in a function.

d) In the following valid Python program, x is declared in Global scope:

```
1. x = 99
2. print(X)
```

Order of Global Name declaration:

However, a name must be defined before it is used.

Keyword global:

a) A global variable can be declared in a function using the keyword global.

b) Line 2 contains a global declaration of x.

c) That is why x can be seen on line 6.

d) Caution: Must call the function (line 5)

```
1.  def fun():
2.      global x
3.      x = 100

4.  fun()
5.  print(x)
```

Output: 100

Must Enter Function Scope

- In the following, fun is never called.
- Thus, the variable x is never defined.

```
1.  def fun():
2.      global x
3.      x = 100
4.  print(x)
```

Output: global name 'x' is not defined.

Two Global Variables:

a) The program below has 2 global variables, and

b) No local variables.

```
1.  count = 99
2.  def fun():
3.      global x
4.      x = 100
5.  fun()
6.  print(count, x)
```

Run Output: 99 100

4) Built-in Scope:

a) If a name is not found using Local, Enclosing or Global, then the built in names are searched

b) Each line in the below program uses two built-in names.

c) You can find all builtin names:

```
dir(__builtins__)
```

Run

```
1.  print(pow(2,3))
2.  print(__name__)
3.  print(len('cat'))
```

Built-In Scope Searched Last:

a) Only 1 builtin name is used below: print on line 4

b) The builtin name pow is not found because a user defined instance of pow is found at Global scope on line 1

```
1. def pow ( base, exponent):
2.         return base ** exponent
3.
4. print (pow(2,3))
```

run

output: 8

No-New Scope:

a) if, elif, or else blocks do not declare a new scope.

b) Variables defined in if, elif, or else block are in Global scope, only if the block is entered.

c) The print on line 4 finds y only because the condition on line 2 is True.

```
1.  x = 99
2.  if x > 0:
3.      y = 17
4.  print(y)
```

run

output: 17

If Not Entered ⇒ Not Defined

a) If an if, elif, or else block is not entered, then a variable in the block will not be defined.

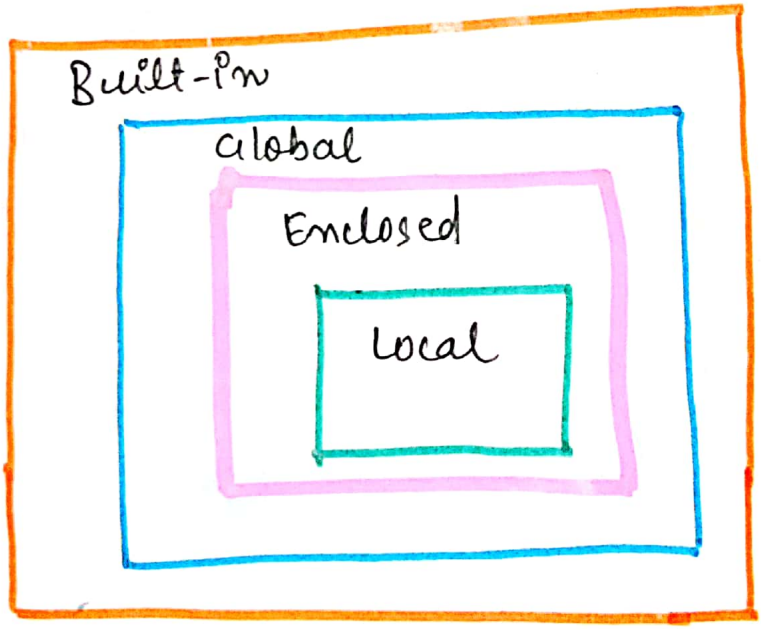
b) Variable y will not be found on line 4 because the condition on line 2 is False.

c) This rule applies to while and for blocks also.

```
1.  x = 0
2.  if X > 0:
3.      y = 17
4.  print(y)
```

Output: 10

Output: name 'y' is not defined.



First class functions in Python:

First class objects in a language are handled uniformly throughout. They may be stored in data structures, passed as arguments, or used in control structures. A programming language is said to support first-class functions if it treats functions as first-class objects. Python supports the concept of first class functions.

Properties of first class functions:

- It is an instance of an object type
- Functions can be stored as variable.
- Pass first class function as argument of some other functions.
- Return functions from other function.
- Store functions in lists, sets or some other data structures.

1. **Functions are objects**: Python functions are first class objects. In the example we are assigning function to a variable. To assign it as variable, the function will not be called. So parentheses '()' will not be there.

For Example:

```
def cube (x):
    return x * x * x
res = cube(5)
print (res)
my_cube = cube
res = my_cube (5)  # The my_cube is same as the cube method.
print (res)
```

Output:
125
125

2. Function Passed as argument of another functions:

Because functions are objects we can pass them as arguments to other functions. Functions that can accept other functions as arguments are also called higher ~~of~~ order functions.

```
def cube(x):
    return x * x * x
def my_map (method, argument_list):
    result = list ()
    for item in argument_list:
        result.append (method (item))
    return result
my_list = my_map (cube, [1,2,3,4,5,6,7,8])
print (my_list)
```

↳ Pass the function as argument.

Output : [1, 8, 27, 64, 125, 216, 343, 512]

3. Functions can return another function: Because functions are objects we can return a function from another function.

Example:

```
def create_adder(x):
    def adder(y):
        return x+y
    return adder
add_15 = create_adder(15)
print(add_15(10))
```

```
def square(x):
    y = x*x
    return y
def sum_of_squares(x,y,z):
    a = square(x)
    b = square(y)
    c = square(z)
    return a+b+c
a = -5
b = 2
c = 10
result = sum_of_squares(a,b,c)
print(result)
```

* First class functions are important because they are basic requirement for writing higher level abstractions with functions. (map, filter, reduce)

* without Python first class functions, we would not have tools like map, filter and reduce, and we would not be able to compose functions up into new functions.

* First-class means it can be treated like a value. Anything you can do to a value such as a number, an array, a list, a string, any of those values. You can pass them to functions, you can store them in variables, you can return them from functions. All of those things make it first class. It means it has the same status.

PYTHON LAMBDA FUNCTION (ANONYMOUS FUNCTIONS) | filter, map, reduce

In Python, anonymous function means that a function is without a name. As we know that def keyword is used to define the normal functions and the lambda keyword is used to create anonymous functions. It has the following syntax:

lambda arguments : expression

a) This function can have any number of arguments but only one expression, which is evaluated and returned.

b) One is free to use lambda functions wherever function objects are required.

c) You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression

d) It has various uses in particular fields of programming besides other types of expressions in functions.

Let's take an example to understand the differences b|w a normal def defined function and lambda function. This is a program that returns the cube of a given value:

```
def cube(y):
    return y*y*y;
g = lambda x: x*x*x
print(g(7))
print(cube(5))
```

Output: 343
125

- **Without using lambda**: Here, both of them returns the cube of a given number. But, while using def, we needed to define a function with a name cube and needed to pass a value to it. After execution, we also needed to return the result from where the function was called using the return keyword.
- **Using Lambda**: Lambda definition does not include a "return" statement, it always contains an expression which is returned. We can also put a lambda definition anywhere a function is expected, and we don't have to assign it to a variable at all. This is the simplicity of lambda functions.

lambda functions can be used along with built-in functions like filter(), map() and reduce()

IIFE in Python Lambda:

IIFE stands for immediately invoked function execution. It means that a function is callable as soon as it is defined.

The filter() method returns an iterator that passed the function check for each element in the iterable.

1) Lambdas in filter()

The filter function is used to select some particular elements from a sequence of elements. The sequence can be any iterator like lists, sets, tuples etc.

The elements which will be selected is based on some pre-defined constraint. It takes 2 parameters:

a) A function that defines the filtering constraint
b) A sequence (any iterator like list, tuple etc)

Eg:

```
sequence = [10, 2, 8, 7, 5, 4, 3, 11, 0, 1]
filtered_result = filter(lambda x: x>4, sequence)
print(list(filtered_result))
```

Output: [10, 8, 7, 5, 11]

- **Code Explanation:**

1) In the first statement, we define a list called sequences which contains some numbers.
2) Here, we declare a variable called filtered_result which will store the filtered values returned by the filter() function.
3) A lambda function which runs on each element of the list and returns true if it is greater than 4.

4) Print the result returned by the filter function.

2) Lambdas in map():

The map function is used to apply a particular operation to every element in a sequence. like filter() it also takes 2 parameters.

1) A function that defines the op to perform on the elements.

2) One or more sequences.

* Example:

```
Sequences = [10, 2, 8, 7, 5, 4, 3, 11, 0, 1]
filtered_result = map(lambda x: x*x, Sequences)
print(list(filtered_result))
```

Output:

[100, 4, 64, 49, 25, 16, 121, 0, 1]

Code Explanation:

1) Here, we define a list called sequences which contains some numbers.
2) We declare a variable called filtered_result which will store the mapped values.
3) A lambda function which runs on each element of the list and returns the square of that number.
4) Print the result returned by the map function.

3) lambdas in reduce()

The reduce function, like map(), is used to apply an operation to every element in a sequence. However, it differs from the map in its working. These are the steps followed by the reduce() function to compute an output:

Step 1) Perform the defined operation on the first 2 elements of the sequence.

Step 2) Save this result

Step 3) Perform the operation with the saved result and the next element in the sequence.

Step 4) Repeat until no more elements are left.

It also takes two parameters:

1. A function that defines the operation to be performed
2. A sequence (any iterator like lists, tuples, etc.)

Example:

```
from functools import reduce
sequences = [1,2,3,4,5]
product = reduce(lambda x, y: x*y,
                 sequences)
print(product)
```

Output: 120

1) **list() function Python:**

The list() constructor returns a list in Python.

list([iterable])

list() Parameters:

The list() constructor takes a single argument:

- iterable (optional) - an object that could be a Sequence (String, tuples) or collection (set, dictionary) or any iterator object.

Return value from list()

1) If no parameters are passed, it returns an empty list
2) If iterable is passed as a parameter, it creates a list consisting of iterable's items.

eg:
```
print (list())  → vowel list (empty)
vowel_string = 'aeiou'
print (list(vowel_string)) → vowel string
```

```
vowel_tuple = ('a', 'e', 'i', 'o', 'u')
print (list(vowel_tuple)) → vowel tuple
```

```
vowel_list = ['a', 'e', 'i', 'o', 'u']
print (list (vowel_list)) → vowel list
```

NOTE: list() function is also used to copy the list items into another list.

Create lists from set and dictionary:

```
vowel_set = {'a', 'e', 'i', 'o', 'u'}
print (list (vowel_set))
```

} vowel set

```
vowel_dictionary = {'a': 1, 'e': 2, 'i': 3, 'o': 4, 'u': 5}
print (list (vowel_dictionary))
```

} dictionary

Output:

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

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

NOTE: In case of dictionaries, the keys of the dictionary will be the items of the list. Also, the order of the elements will be random. Because set and dictionary both are unordered collection and does not allow duplicate values.

2) filter() function: The filter() method filters the given sequence with the help of a function that tests each element in the sequence to be true or not.

eg:

```
def fun(variable):
    letters = ['a', 'e', 'i', 'o', 'u']
    if (variable in letters):
        return True
    else:
        return False
```

```
sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']
filtered = filter(fun, sequence)
print('The filtered letters are:')
for s in filtered:
    print(s)
```

Output: The filtered letters are:
e
e

Application: It is normally used with lambda function to separate list, tuple or sets.

```
seq = [0, 1, 2, 3, 5, 8, 13]
result = filter(lambda x: x%2, seq)
print(list(result))
result = filter(lambda x: x%2==0, seq)
print(list(result))
```

Output: [1, 3, 5, 13]
[0, 2, 8]

3) Map() function :

The map() function calls the specified function for each item of an iterable (such as string, list, tuple or dictionary) and returns a list of results.

Syntax :

map (function, iterables)

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 myfun (n):
    return len(n)

x = map(myfun, ('apple', 'bannana', 'cherry'))
```

Output : ['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'))
```

Output:

['appleorange', 'banana lemon', 'cherry pineapple']

4) **reduce() in Python**: The reduce(fun, seq) function is used to apply a particular function passed in its argument to all of the list elements mentioned in the sequence passed along. This function is defined in "functools" module.

Working:

- At first step, first two elements of sequence are picked and the result is obtained.
- Next step is to apply the same function to the previously attained result and the number just succeeding the second element and the result is again stored.
- This process continues till no more elements are left in the container.
- The final returned result is returned and printed on console.

Example:

```
import functools
lis = [1, 3, 5, 6, 2]
print("The sum of the list elements is ", end="")
print(functools.reduce(lambda a, b: a+b, lis))
```

Output: The sum of the list elements is : 17

```
# using reduce to compute maximum element from list
print ("The maximum element of the list is : ",end=" ")
print (functools.reduce (lambda a,b : a if a>b else b,
                         lis))
```

Output: The maximum element of the list is : 6

Using Operator functions: reduce() can ~~be~~ also be combined with operator functions to achieve the similar functionality as with lambda functions and makes the code more readable.

For example:

```
import functools
import operator
lis = [1,3,5,6,2]
# using reduce to compute sum of list
# using operator functions
print ("The sum of the list elements is :", end=" ")
print (functools.reduce (operator.add, lis))
```

Output: The sum of the list elements is : 17

reduce() vs accumulate() Both reduce() and accumulate() can be used to calculate the summation of a sequence elements. But there are differences in the implementation aspects in both of these.

- reduce() is defined in "functools" module, accumulate() in "itertools" module.
- reduce() stores the intermediate result and only returns the final summation value.

Whereas, accumulate() returns a list containing the intermediate results. The last number of the list returned is summation value of the list.

- reduce(fun, seq) takes function as 1st and sequence as 2nd argument. In contrast accumulate (seq, fun) takes sequence as 1st argument and function as 2nd argument.

Example:

```
import itertools
import functools
lis = [1,3,4,10,4]
print ("The summation of list :", end="")
print (list(itertools.accumulate(lis,lambda x,y : x+y)))
print ("The Summation of list :", end=" ")
print (functools.reduce(lambda x,y : x+y, lis))
```

Accumulate()

Output: The summation of list : [1,4,8,18,22]

The Summation of list : 22

reduce()

STRINGS IN PYTHON

What are strings?

A String is a sequence of characters. A character is simply a symbol. For example, the english language has 26 characters.

Computers do not deal with characters, they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0's and 1's.

This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encoding used. In Python, a String is a sequence of Unicode characters. Unicode was introduced to include every character in all languages and bring uniformity in encoding.

How to create a String in Python?

Strings can be created by enclosing characters inside a single quote or double-quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

Example:

```
my_string = 'Hello'
print(my_string)
```

```
mystring = "Hello"
print(mystring)
my_string = '''Hello'''
print(my_string)
my_string = """Hello, welcome"""
print(my_string)
```

Output:
Hello
Hello
Hello
Hello, welcome

How to access characters in a string?

We can access individual characters using indexing and a range of characters using slicing. Index starts from 0. Trying to access a character out of index range will ~~range~~ raise an IndexError. The index must be an integer. We can't use float or other types, this will result into TypeError.

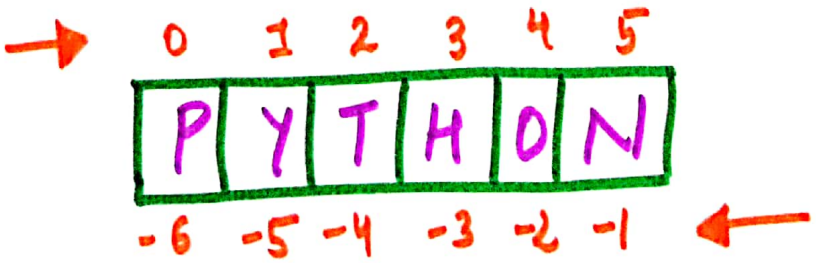
Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on. We can access a range of items in a string by using the slicing operator (colon).

```
str = 'programiz'
print ('str =', str)

# first character
print ('str[0] = ', str[0])

# last character
print ('str[-1] =', str[-1])

# slicing 2nd to 5th character
print ('str[1:5] =', str[1:5])

# slicing 6th to 2nd character
print ('str[5:-2] =', str[5:-2])
```

NOTE: If we try to access index out of the range or use decimal number, we will get errors.

Slicing can be best visualized by considering the index to be between the elements.

- If we want to access a range, we need the index that will slice the portion from the string



How to change or delete a String?

Strings are immutable. This means that elements of a string cannot be changed once it has been assignment. We can reassign different strings to the same name.

```
my_string = 'Python'
my_string[5] = 'a'
```

➔ **NOTE:** TypeError: 'str' object does not support item assignment.

```
my_string = 'PythonProgramming'
print(my_string)
```

➔ **NOTE:** We cannot delete or remove characters from a string. But deleting the string entirely is possible using the keyword del

```
String1= "Hello"
print ("Initial String:")
print String1
del String1
print ("\nDeleting")
print (String1)
```

Python String Operations

There are many operations that can be performed with string which makes it one of the most used datatypes in Python.

- **Concatenation of Two or more Strings:**

Joining of two or more strings into a single is called concatenation.

- The + operator does this in Python. Simply writing two string literals together also concatenates them.

The * operator can be used to repeat the string for a given number of times

```
str1 = 'Hello'
str2 = 'World!'
# using '+'
print('str1 + str2', str1 + str2)
# using '*'
print('str1 * 3', str1 * 3)
```

Output: Hello World!
Hello
Hello
Hello

Writing two string literals together also concatenates them like + operator.
If we want to concatenate strings in different lines, we can use parantheses.

```
# two string literal together
```

Built-in functions to work with Python:

Various built-in functions that work with sequence, works with string as well.
Some of the commonly used ones are enumerate() and len(). The enumerate() function returns an enumerate object. It contains the index and value of all the items in the string as pairs. This can be useful for iteration.
Similarly, len() returns the length (number of characters) of the string.

```
str = 'old'
# enumerate()
list_enumerate = list(enumerate(str))
print('list(enumerate(str)=', list_enumerate)
# Character count
print('len(str)=', len(str))
```

Output:

DATA STRUCTURES

Python offers a range of compound datatypes often referred to as sequences. These are of following types:

1. List
2. Tuple
3. Dictionary
4. Set

- **1. LIST :** Lists are just like the arrays, declared in other languages. Lists need not be homogeneous always which makes it a most powerful tool in Python. A single list may contain Data types like Integers, Strings, as well as Objects. Lists are mutuable, and hence, they can be altered even
- after their creation.

List in Python are ordered and have a definite count. The elements in a list are indexed according to a definite sequence and the indexing of a list is done with 0 being the first index. Each element in the list has its definite place in the list, which allows duplicating of elements in the list, with each element having its own distinct place and credibility.

NOTE: Lists are a useful tool for preserving a sequence of data and further iterating over it.

Creating a List: Lists in Python can be created by just placing the sequence inside the square brackets [].

```
# empty list
my_list = [ ]
# list of integers
my_list = [1, 2, 3]
# list of mixed datatypes
my_list = [1, 'Hello', 3.4 ]
```

Another, Also a list can even have another list as an item. This is called nested list.

```
# nested list
my_list = ["mouse", [8, 4, 6], ['a']]
```

How to access elements from a list ?

There are various way in which we can access the elements of a list.

List index: We can use the index operator to access an item in a list. Index starts from 0. So, a list having 5 elements will have index from 0 to 4. Trying to access an element other that will raise an IndexError. The

index must be an integer. We can't use float or other types, this will result into TypeError.

How to change or add elements to a list?

Lists are mutable, meaning, their elements can be changed unlike string or tuple.

We can use assignment operator (=) to change an item or a range of items.

```
# mistake values
odd = [2, 4, 6, 8]
# change the 1st item
odd[0] = 1
# Output: [1, 4, 6, 8]
```

We can add one item to a list using append() method or add several items using extend() method.

```
odd = [1, 3, 5]
odd.append(7)
# output: [1, 3, 5, 7]
print(odd)
odd.extend([9, 11, 13])
print(odd)
```

We can also use + operator to combine two lists. This is called concatenation.

The * operator repeats a list for the given number of times.

```
odd = [1,3,5]
print (odd + [9,7,5])
print (["re"] * 3)
```

Output : [1,3,5,9,7,5]
["re", "re", "re"]

Furthermore, we can insert one item at a desired location by using the method insert() or insert multiple items by squeezing it into an empty slice of a list.

```
odd = [1,9]
odd.insert (1,3)
# Output: [1,3,9]
print (odd)
Odd [2:2] = [5,7]
# Output [1,3,5,7,9]
print (odd)
```

How to delete or remove elements from a list?

We can delete one or more items from a list using the keyword del. It can even delete the list entirely.

```
my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']
# delete one item
del my_list[2]
# output: ['p', 'r', 'b', 'l', 'e', 'm']
print(my_list)
# delete multiple items
del my_list[1:5]
# output ['p', 'm']
print(my_list)
# delete entire list
del my_list
# output: Error: list not defined
print(my_list)
```

We can use remove() method to remove the given item or pop() method to remove an item at the given index.

The pop() method removes and returns the last item if index is not provided. This helps us implement lists as stacks (first in, last out data structure).

We can also use the clear() method to empty a list.

```
my_list = [1, 2, 3, 4, 5, 6]
my_list.remove(3)   # O/P: [1,2,4,5,6]
print(my_list)
print(my_list.pop(1)) # O/P: [1, 4, 5, 6]
```

```
print(my_list.pop()) #O/P [1,4,5]
my_list.clear()
print(my_list)       #O/P : []
```

List Comprehension: List comprehension is an elegant and concise way to create a new list from an existing list in Python.

List comprehension consists of an expression followed by an for statement inside square brackets.

Example:

```
pow2 = [2 ** x for x in range(10)]
print(pow2)
```

O/P: [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]

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

A list comprehension can optionally contain more for or if statements. An optional if statement can filter out items for the new lists.

Syntactically, list comprehensions consist of an iterable containing an expression followed by a for clause. This can be followed by additional for or if clauses, so familarity with for loops and

Conditional statements will help you understand list comprehensions better.

List comprehensions provide an alternative syntax to creating lists and other sequential data types. While other methods of iteration, such as for loops can also be used to create lists, list comprehensions may be preferred because they can limit the number of lines used in your program.

- **Using Conditionals with List Comprehensions:**

List comprehensions can utilize conditional statements to modify existing lists or other sequential data types when creating new lists.

Let's look at an example of an if statement used in a list comprehension:

- ~~first_tuple = ('A', 'B', 'C', 'D')~~

~~first_tuple = E~~

```
number_list = [x ** 2 for x in range(10) if
                x % 2 == 0]
print(number_list)
```

The list that is being created, number_list, will be populated with the squared values of each item in the range from 0-9 if the item's value is

value is divisible by 2. The output is as follows:

Output: [0, 4, 16, 36, 64]

You can also replicate nested if statements with a list comprehension:

```
number_list = [x for x in range(100) if x%3==0 if x%5==0]

print(number_list)
```

Here, the list comprehension will first check to see if the number x is divisible by 3, and then check to see if x is divisible by 5. If x satisfies both requirement it will print.

Output: [0, 15, 30, 45, 60, 75, 90]

Nested loops in list comprehensions:

```
my_list = [x*y for x in [20,40,60] for y in [2,4,6]]

print(my_list)
```

Output: [40, 80, 120, 80, 160, 240, 120, 240, 360]

CONCLUSION: List comprehensions allow us to transform one list or other sequence into a new list. They provide a concise syntax for completing this task, limiting our lines of codes.

list comprehensions follow the mathematical form of Set-builder notation or Set comprehensions, so they may be particulary intuitive to programmers with a mathematical background.

Though list comprehensions can make our code more succint, it is important to ensure that our final code is as readable as possible, so very long single lines of code should be avoided to ensure that our code is user friendly.

TUPLES:

- A tuple is a data structure that is an immutable or unchangeable, ordered sequence of elements. Because tuples are immutable, their values cannot be modified.
- Tuples are used for grouping data. Each element or value that is inside of a tuple is called an item.
- Tuples have values between parentheses () separated by commas, Empty values will appear as tuple1 = ()
but tuples with even one value must use a comma as in tuple1 = (1,)

When thinking about Python tuples and other data structures that are types of collections, it's useful to consider all the different collections you have on your computer; your assortment of files, your

playlists, your browser bookmarks, your emails, the collection of videos you can access on a streaming service, and more.

Tuples are similar to lists, but their values can't be modified. Because of this, when you use tuples in your code, you are conveying to others that you don't intend for there to be changes to that sequence of values. Additionally, because the values do not change, your code can be optimized through the use of tuples in Python, as the code will be slightly faster for tuples than for lists.

1) Concatenation of Tuples: Operators can be used to concatenate or multiply tuples. Concatenation is done with + operator, and multiplication is done with the * operator.

The + operator can be used to concatenate two or more tuples together. We can assign the values of two existing tuples to a new tuple:

```
tuple 1 = (0, 1, 2, 3)
tuple 2 = ('python', 'programming')
print (tuple 1 + tuple 2)
```

Output: [0, 1, 2, 3, 'Python', 'Programming']

Nesting in tuples:

```
tuple1 = (0, 1, 2, 3)
tuple2 = ('python',)
tuple3 = (tuple1, tuple2)
print (tuple3)
```

Output: ((0, 1, 2, 3), ('python'))

Finding length in tuples:

```
tuple2 = ('python',)
print (len (tuple2))
```

Output: 1

Converting list to a Tuple:

```
list1 = [0, 1, 2]
print (tuple (list1))
print (tuple ('python')) # String 'Python'
```

Output: (0, 1, 2)
('p', 'y', 't', 'h', 'o', 'n')

Tuples in a loop

```
tup = ('python',)
n = 5
for i in range (int (n)):
    tup = (tup,)
    print (tup)
```

Output:

(('python',),)

((('python',),),)

(((('python',),),),)

((((('python',),),),),).

((((((('python',),),),),),),),)

Tuple in-built functions:

```
tuple 1 = ('python', 'programming')
tuple 2 = ('coder', 1)
if (cmp(tuple 1, tuple 2) != 0):
    print ('Not the same')
else:
    print ('Same')
print ('Max. in tuples 1, 2:' +
        Str (max(tuple 1)) + ',' +
        Str (max (tuple 2)))
print('Min. :' +Str(min(tuple 1)) + ',' + Str(min(
                                    tuple 2))
```

NOTE: max() and min() checks the based on ASCII values. If there are two strings in a tuple, then the first different character in the strings are checked.

Output: Not the same

Max. in tuples 1,2 : python, coder

Min programming, 1

Tuple(): Converts a list into tuple.

Packing and unpacking Arguments in Python:

We use two operators * (for tuples) and ** (for dictionaries)

Suppose, consider a situation where we have a function that receives four arguments. We want to make call to this function and we have a list of size 4 with us that has all arguments for the function. If we simply pass list to the function, the call doesn't work.

```
def fun(a, b, c, d):
    print(a, b, c, d)

mylist = [1, 2, 3, 4]

fun(my_list)
```

O/P: Type error:
fun() takes exactly 4 arguments (1 given)

1.) Unpacking

We can use * to unpack the list so that all elements of it can be passed as different parameters.

```
def fun(a,b,c,d):
    print(a,b,c,d)

my_list = [1,2,3,4]

fun(*my_list)
```

O/P: (1,2,3,4)

As another example, consider the built-in range() function that expects separate start and stop arguments. If they are not available separately, write the function call with the * operator to unpack the arguments out of a list or tuple.

```
>>> range(3,6) # normal call with separate arguments.
[3,4,5]
>>> args = [3,6]
>>> range(*args) # call with arguments unpacked from a list.
[3,4,5]
```

2.) Packing: When we don't know how many arguments need to be passed to a python function, we can use Packing to pack all arguments in a tuple.

Arbitary Arguments

```
def mySum(*args):
    sum = 0
    for i in range(0, len(args)):
        sum = sum + args[i]
    return sum

print(mysum(1,2,3,4,5))
print(mysum(10,20))
```

- Output: 15
  30

The above function mySum() does 'packing' to pack all the arguments that this method call receives into one single variable. Once we have this 'packed' variable, we can do things with it that we would with a normal tuple. args[0], and args[1] would give you the first and second argument, respectively

- Since our tuples are immutable, you can convert the args tuple to a list so you can also modify delete and re-arrange items in it.

** is used for dictionaries.

O/P: 2 4 10

```
def fun(a,b,c):
    print(a,b,c)

d = {'a':2, 'b':4, 'c':14}

fun(**d)
```

Applications and Important Points:

1. Used in socket programming to send a vast number of requests to a server.
2. Used in Django framework to send variable arguments to view functions.
3. There are wrapper functions that require us to pass in variable arguments.
4. Modification of arguments become easy, but at the same time validation is not proper, so they must be used with care.

SETS: A set is an unordered collection data type that is iterable, mutable and has no duplicate elements. Python's set class represents the mathematical notion of a set. The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set. This is based on a data structure known as a hash table.

```
Set = set(["a", "b", "c"])
print("Set: ")
print(Set)
# Adding element to the set
Set.add("d")
print("\n set after adding:")
print(Set)
```

Output: Set:
Set(['a', 'c', 'b'])

Set after adding:
Set(['a', 'c', 'b', 'd'])

Accessing Values in a set:

We cannot access individual values in a set. We can only access all the elements together as shown. But we can also get a list of individual elements by looping through the set.

```
Days = set(["Mon", "Tue", "Wed"])
for d in Days:
    print(d)
```

O/P: Wed
Mon
Tue

Adding items to a set: We can add elements to a set by using add() method.

```
Days = set(["Mon", "Tue", "Wed", "Thu", "Fri"])

Days.add("Sun")
print(Days)
```

O/P: set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu'])

Removing item from a Set: We can remove elements from a set by using discard() method.

```
Days = set(["Mon", "Tue"])

Days.discard("Mon")
print(Days)
```

O/P: set(['Tue'])

Union of Sets: The union operation on two sets produces a new set containing all the distinct elements from both the sets.

```
Set1 = Set([1, 2, 3, 4, 5])
Set2 = set([2, 4, 5, 6])

Set3 = Set1 | Set2
print(Set3)
print(Set1.union(Set2))
```

O/P: set([1, 2, 3, 4, 5, 6])

Intersection of Sets: The intersection operation on two sets produces a new set containing only the common elements from both the sets.

```
Set1 = set([1,2,3,4,5])
Set2 = set([3,4,5])
Set3 = Set1 & Set2
print(Set3)
```

- → print(set1.intersection(set2))

O/P: set([3,4,5])

Difference of Sets: The difference operation on two sets produces a new set containing only the elements from the first set and none from the second set.

```
Set1 = set([1,2,3,4,5])
Set2 = set([3,4,5])
Set3 = Set1 - Set2
print(Set3)
```

O/P:- set([1,2])

Compare Sets: We can check if a given set is a subset or superset of another set. The result is True or False depending on the elements present in the sets.

```
Set1 = set([1,2,3,4,5,6])
Set2 = set([4,5,6,7])
Subset = Set1 <= Set2
Superset = Set2 >= Set1
print (Subset)
print (Superset)
```

O/P:

Difference between discard() and remove()

Despite the fact that discard() and remove() method both perform the same task, There is one main difference between discard() and remove().

If the key to be deleted from the set using discard() doesn't exist in the set, the python will not give the error. The program maintains its control flow.

On the other hand, if the item to be deleted from the set using remove() doesn't exist in the set, the python will give the error.

The intersection_update() method:

The intersection_update() method removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).

The Intersection_update() method is different from intersection() method since it modifies the original set by removing the unwanted items, on the other hand, intersection() method returns a new set.

```
a = {"ayush", "bob", "castle"}
b = {"castle", "dude", "emyway"}
c = {"fusion", "gawan", "castle"}

a.intersection_update(b,c)
print(a)
```

O/P: {'castle'}

Frozen Sets: The frozen sets are the immutable form of the normal sets, i.e, the items of the frozen set can not be changed and therefore it can be used as a key in dictionary.

The elements of the frozen set can not be changed after the creation. We cannot change or append the content of the frozen sets by using the methods like add() or remove().

The frozenset() method is used to create the frozenset object. The iterable sequence is passed into this method which is converted into the frozen set as a return type of the method.

```
Frozenset = frozenset([1,2,3,4,5])
print(type(Frozenset))
print("\n printing the content of frozenset...")
for i in Frozenset:
    print(i):
Frozenset.add(6) # this will create error
    because we cannot change after creation.
```

O/P: <class 'frozenset'>

printing the content of frozen set...

1
2
3
4
5

PYTHON DICTIONARY: Dictionary is used to implement the key-value pair in Python. The dictionary is the data type in Python which can simulate the real-life data arrangement where some specific value exists for some particular key.

In other words, we can say that a dictionary is the collection of key-value pairs where the value can

- be any python object whereas the keys are the immutable python object, i.e Numbers, string or tuple.

Dictionary simulates Java hash-map in Python.

Creating the Dictionary: The dictionary can be created by using multiple key-value pairs enclosed

- with the small brackets () and separated by the colon (:). The collections of the key-value pairs are enclosed within the curly braces {}.

```
Dict = {"Name": "Ayush", "Age": 22}
```

Example:

```
Employee = {"Name": "John", "Age": 29, "Salary":
25000, "Company": "GOOGLE"}
print (type(Employee))
print ("printing Employee data ..")
Print ("Employee)
```

Output: <class 'dict'>
printing Employee data ...
{'Age': 29, 'Salary': 250000, 'Name': 'John', 'Company': 'GOOGLE'}

Dictionary Methods:

1. copy()
2. clear()
3. pop()
4. popitem(): Removes the arbitrary key-value pair from the dictionary and returns it as tuple.
5. get(): It is conventional method to access a value for a key.