UNIT – III

FUNCTIONS:

A function is a block of organized, reusable code that is used to perform a single, related action.

- Once a function is written, it can be reused as and when required. So, functions are also called reusable code.
- Functions provide modularity for programming. A module represents a part of the program. Usually, a programmer divides the main task into smaller sub tasks called modules.
- Code maintenance will become easy because of functions. When a new feature has to be added to the existing software, a new function can be written and integrated into the software.
- When there is an error in the software, the corresponding function can be modified without disturbing the other functions in the software.
- The use of functions in a program will reduce the length of the program.

As you already know, Python gives you many built-in functions like sqrt( ), etc. but you can also create your own functions. These functions are called *user-defined functions*.

Difference between a function and a method:

A function can be written individually in a python program. A function is called using its name. When a function is written inside a class, it becomes a „method". A method is called using object name or class name. A method is called using one of the following ways:

Objectname.methodname()

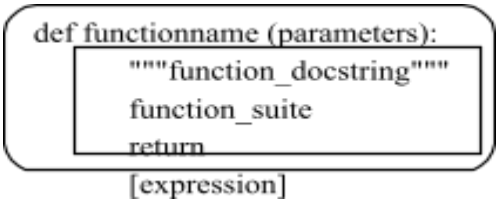
Classname.methodname()

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses ( ).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return none.

Syntax:



By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example:

```
def add(a,b):
    """This function sum the numbers"""
    c=a+b
```

```
        print c
        return
```

Here, „*def'* represents starting of function. „*add'* is function name. After this name, parentheses ( ) are compulsory as they denote that it is a function and not a variable or something else. In the parentheses we wrote two variables „*a*" and „*b*" these variables are called „parameters". A parameter is a variable that receives data from outside a function. So, this function receives two values from outside and those are stored in the variables „a" and „b". After parentheses, we put colon (:) that represents the beginning of the function body. The function body contains a group of statements called „suite".

Calling Function:

A function cannot run by its own. It runs only when we call it. So, the next step is to call function using its name. While calling the function, we should pass the necessary values to the function in the parentheses as:

```
add(5,12)
```

Here, we are calling „add" function and passing two values 5 and 12 to that function. When this statement is executed, the python interpreter jumps to the function definition and copies the values 5 and 12 into the parameters „a" and „b" respectively.

Example:

```
def add(a,b):
        """This function sum the
        numbers""" c=a+b
        prin
t c
add(5,12) #
17
```

Returning Results from a function:

We can return the result or output from the function using a „return" statement in the function body. When a function does not return any result, we need not write the return statement in the body of the function.

Q) Write a program to find the sum of two numbers and return the result from the function.

```
def add(a,b):
        """This function sum the
        numbers""" c=a+b
        return c
print add(5,12) #
17
print add(1.5,6) #6.5
```

Returning multiple values from a function:

A function can returns a single value in the programming languages like C, C++ and JAVA. But, in python, a function can return multiple values. When a function calculates multiple results and wants to return the results, we can use return statement as:

return a, b, c

Here, three values which are in „a", „b" and „c" are returned. These values are returned by the function as a tuple. To grab these values, we can three variables at the time of calling the function as:

x, y, z = functionName( )

Here, „x", „y" and „z" are receiving the three values returned by the function.

Example:

```
def
   calc(a,b
   ):
   c=a+b
   d=a-b
   e=a*b
   return c,d,e
x,y,z=calc(5,8)
print
"Addition=",x
print "Subtraction=",y
print
"Multiplication=",z
```

Functions are First Class Objects:

In Python, functions are considered as first class objects. It means we can use functions as perfect objects. In fact when we create a function, the Python interpreter internally creates an object. Since functions are objects, we can pass a function to another function just like we pass an object (or value) to a function. The following possibilities are:

- It is possible to assign a function to a variable.
- It is possible to define one function inside another function.
- It is possible to pass a function as parameter to another function.
- It is possible that a function can return another function.

To understand these points, we will take a few simple programs.

Q) A python program to see how to assign a function to a variable.

```
def display(st):
    return
 "hai"+st
x=display("es
e") print x
```

Output: haicse

Q) A python program to know how to define a function inside another function.

```
def
    display(st):
    def
    message():
        return "how r
    u?"
    res=message()+s
    t return res
x=display("cs
e") print x
```

Output: how r u?cse

Q) A python program to know how to pass a function as parameter to another function.

```
def
    display(f):
    return
    "hai"+f
def message():
    return "how r
    u?"
fun=display(message
()) print fun
```

Output: haihow r u?

Q) A python program to know how a function can return another function.

```
def display():
    def
    message():
        return "how r
    u?" return
    message
fun=displa
y() print
fun()
```

Output: how r u?

Pass by Value:

Pass by value represents that a copy of the variable value is passed to the function and any modifications to that value will not reflect outside the function. In python, the values are sent to functions by means of object references. We know everything is considered as an object in python. All numbers, strings, tuples, lists and dictionaries are objects.

If we store a value into a variable as:

x=10

In python, everything is an object. An object can be imagined as a memory block where we can store some value. In this case, an object with the value „10" is created in memory for which a name „x" is attached. So, 10 is the object and „x" is the name or tag given to that object. Also, objects are created on heap memory which is a very huge memory that depends on the RAM of our computer system.

Example: A Python program to pass an integer to a function and modify it.

```
def modify(x):
    x=15
    print "inside",x,id(x)
x=10
modify(x)
print "outside",x,id(x)
```

Output:

```
inside 15 6356456
outside 10 6356516
```

From the output, we can understand that the value of „x" in the function is 15 and that is not available outside the function. When we call the modify( ) function and pass „x" as:

modify(x)

We should remember that we are passing the object references to the modify( ) function. The object is 10 and its references name is „x". This is being passed to the modify( ) function. Inside the function, we are using:

x=15

This means another object 15 is created in memory and that object is referenced by the name „x". The reason why another object is created in the memory is that the integer objects are immutable (not modifiable). So in the function, when we display „x" value, it will display 15. Once we come outside the function and display „x" value, it will display numbers of „x" inside and outside the function, and we see different numbers since they are different objects.

In python, integers, floats, strings and tuples are immutable. That means their data cannot be modified. When we try to change their value, a new object is created with the modified value.

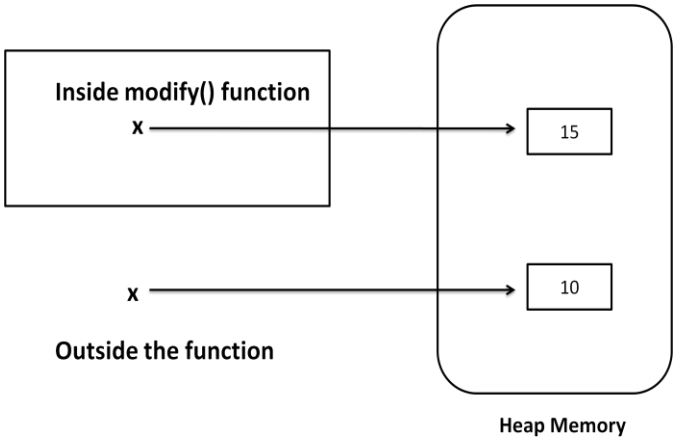


Fig. Passing Integer to a Function

Pass by Reference:

Pass by reference represents sending the reference or memory address of the variable to the function. The variable value is modified by the function through memory address and hence the modified value will reflect outside the function also.

In python, lists and dictionaries are mutable. That means, when we change their data, the same object gets modified and new object is not created. In the below program, we are passing a list of numbers to modify ( ) function. When we append a new element to the list, the same list is modified and hence the modified list is available outside the function also.

Example: A Python program to pass a list to a function and modify it.

```
def modify(a):
    a.append(5)
    print "inside",a,id(a)
a=[1,2,3,4]
modify(a)
print "outside",a,id(a)
```

Output:

```
inside [1, 2, 3, 4, 5] 45355616
outside [1, 2, 3, 4, 5] 45355616
```

In the above program the list ‚a" is the name or tag that represents the list object. Before calling the modify( ) function, the list contains 4 elements as: **a=[1,2,3,4]**

Inside the function, we are appending a new element „5" to the list. Since, lists are mutable, adding a new element to the same object is possible. Hence, append( ) method modifies the same object.

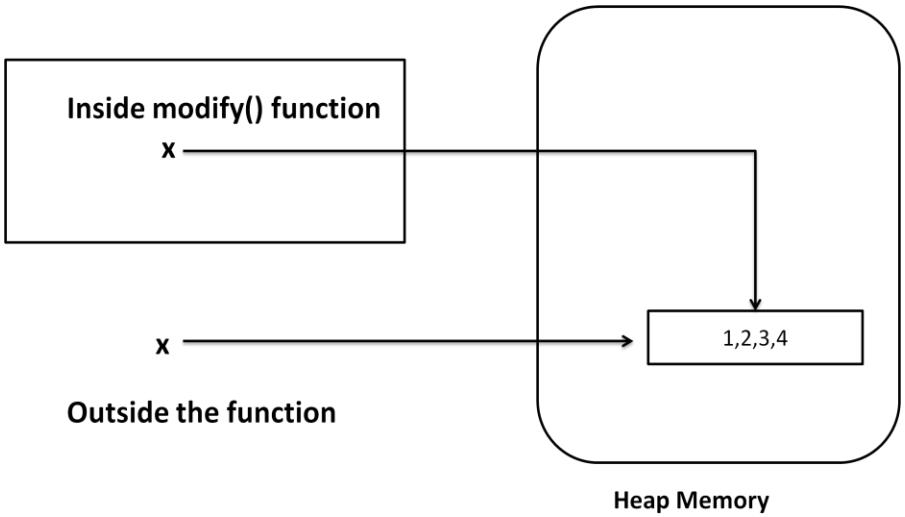


Fig. Passing a list to the function

Formal and Actual Arguments:

When a function is defined, it may have some parameters. These parameters are useful to receive values from outside of the function. They are called „formal arguments". When we call the function, we should pass data or values to the function. These values are called „actual arguments". In the following code, „a" and „b" are formal arguments and „x" and
„y" are actual arguments.

Example:

```
def add(a,b): # a, b are formal arguments
    c=a+b
    print c
x,y=10,15
add(x,y)    # x, y are actual arguments
```

The actual arguments used in a function call are of 4 types:

a) Positional arguments
b) Keyword arguments
c) Default arguments
d) Variable length arguments

a) Positional Arguments:

These are the arguments passed to a function in correct positional order. Here, the number of arguments and their position in the function definition should match exactly with the number and position of argument in the function call.

```
def attach(s1,s2):
    s3=s1+s2
    print s3
attach("New","Delhi") #Positional arguments
```

This function expects two strings that too in that order only. Let‟s assume that this function attaches the two strings as s1+s2. So, while calling this function, we are supposed to pass only two strings as: **attach("New","Delhi")**
The preceding statements displays the following output NewDelhi
Suppose, we passed "Delhi" first and then "New", then the result will be: "DelhiNew". Also, if we try to pass more than or less than 2 strings, there will be an error.

b) Keyword Arguments:

Keyword arguments are arguments that identify the parameters by their names. For example, the definition of a function that displays grocery item and its price can be written as:

def grocery(item, price):

At the time of calling this function, we have to pass two values and we can mention which value is for what. For example,

grocery(item='sugar', price=50.75)

Here, we are mentioning a keyword „item‟ and its value and then another keyword „price‟ and its value. Please observe these keywords are nothing but the parameter names which receive these values. We can change the order of the arguments as:

grocery(price=88.00, item='oil')

In this way, even though we change the order of the arguments, there will not be any problem as the parameter names will guide where to store that value.

```
def
    grocery(item,price):
    print "item=",item
    print "price=",price
grocery(item="sugar",price=50.75) # keyword arguments
grocery(price=88.00,item="oil") # keyword arguments
```

Output:

```
item= sugar
price=
50.75 item=
oil price=
88.0
```

c) Default Arguments:

We can mention some default value for the function parameters in the definition.
Let‟s take the definition of grocery( ) function as:

def grocery(item, price=40.00)

Here, the first argument is „item‟ whose default value is not mentioned. But the second argument is „price‟ and its default value is mentioned to be 40.00. at the time of calling this function, if we do not pass „price‟ value, then the default value of 40.00 is taken. If we mention the „price‟ value, then that mentioned value is utilized. So, a default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

Example:

```
def grocery(item,price=40.00):
print "item=",item
    print
    "price=",price
grocery(item="sugar",price=50.75
) grocery(item="oil")
```

Output:

item=sugarprice=50.75
item=oilprice=40.0

d) Variable Length Arguments:

Sometimes, the programmer does not know how many values a function may receive. In that case, the programmer cannot decide how many arguments to be given in the function definition. for example, if the programmer is writing a function to add two numbers, he/she can write:

add(a,b)

But, the user who is using this function may want to use this function to find sum of three numbers. In that case, there is a chance that the user may provide 3 arguments to this function as:

add(10,15,20)

Then the add( ) function will fail and error will be displayed. If the programmer want to develop a function that can accept „n" arguments, that is also possible in python. For this

purpose, a variable length argument is used in the function definition. a variable length argument is an argument that can accept any number of values. The variable length argument is written with a „*" symbol before it in the function definition as:

def add(farg, *args):

here, „farg" is the formal; argument and „*args" represents variable length argument. We can pass 1 or more values to this „*args" and it will store them all in a tuple.

Example:

```
def
    add(farg,*args
    ): sum=0
    for i in
        args:
        sum=su
        m+i
    print "sum
 is",sum+farg add(5,10)
add(5,10,20)
add(5,10,20,30)
```

Output:

```
sum is 15
sum is 35
sum is 65
```

Local and Global Variables:

When we declare a variable inside a function, it becomes a local variable. A local variable is a variable whose scope is limited only to that function where it is created. That means the local variable value is available only in that function and not outside of that function.

When the variable „a‟ is declared inside myfunction() and hence it is available inside that function. Once we come out of the function, the variable „a‟ is removed from memory and it is not available.

Example-1:

```
def myfunction():
   a=10
   print "Inside function",a #display
10 myfunction()
print "outside function",a # Error, not available
```

Output:

```
Inside function 10
outside function
NameError: name 'a' is not defined
```

When a variable is declared above a function, it becomes global variable. Such variables are available to all the functions which are written after it.

Example-2:

```
a=11
def myfunction():
   b=10
   print "Inside function",a #display global var
   print "Inside function",b #display local var
myfunction()
print "outside function",a # available
print "outside function",b # error
```

Output:

```
Inside function 11
Inside function 10
outside function 11
outside function
NameError: name 'b' is not defined
```

The Global Keyword:

Sometimes, the global variable and the local variable may have the same name. In that case, the function, by default, refers to the local variable and ignores the global variable. So, the global variable is not accessible inside the function but outside of it, it is accessible.

Example-1:

```
a=11
def myfunction():
   a=10
   print "Inside function",a # display local variable
myfunction()
print "outside function",a # display global variable
```

Output:

```
Inside function 10
outside function 11
```

When the programmer wants to use the global variable inside a function, he can use the keyword „global" before the variable in the beginning of the function body as:

global a

Example-2:

```
a=11
def myfunction():
   global a
   a=10
   print "Inside function",a # display global variable
myfunction()
print "outside function",a # display global variable
```

Output:

```
Inside function 10
outside function 10
```

Recursive Functions:

A function that calls itself is known as „recursive function". For example, we can write the factorial of 3 as:

```
factorial(3) = 3 * factorial(2)
Here, factorial(2) = 2 *
factorial(1) And, factorial(1) = 1 *
factorial(0)
```

Now, if we know that the factorial(0) value is 1, all the preceding statements will evaluate and give the result as:

```
factorial(3) = 3 * factorial(2)
             = 3 * 2 * factorial(1)
             = 3 * 2 * 1 * factorial(0)
             = 3 * 2 * 1 * 1
             = 6
```

From the above statements, we can write the formula to calculate factorial of any number „n" as: factorial(n) = n * factorial(n-1)

Example-1:

```
def
   factorial(n):
   if n==0:
      result=1
   else:
      result=n*factorial(n-1
   ) return result
for i in range(1,5):
   print "factorial of ",i,"is",factorial(i)
```

Output:

```
factorial of 1 is 1
factorial of 2 is 2
factorial of 3 is 6
factorial of 4 is 24
```

Void Functions and Functions Returning Values

In Python, void functions are not exactly the same as functions you see in C, C++ or Java. Meaning to say, if the function body does not have any return statement then a special value None returns when the function terminates. This article will throw light on void functions in detail.

Void Functions

In Python, None refers to a literal kind of type NoneType which comes in use for denoting the absence of a value. Moreover, it is generally assigned to a variable for denoting that the variable does not point to any object.

Through the program given below, you can see how it shows that void functions return None:

COPY CODE

```
1 def add(num1, num2):

2 print("Sum is", num1 + num2)

3

4 return_val = add(300, 500)

5 print(return_val)
```

Output:

COPY CODE

```
1 Sum is 800

2 None
```

Moreover, add () function certainly returns None. Thus, one can say that in Python, all functions return value whether usage of the return statement is done or not. But, this does not imply that one can make use of void functions like a value-returning function.

Browse more Topics Under Defining Functions

Invoking Functions

Passing Parameters

Scope of Variables

Flow of Execution

Functions Returning Values

Most of the functions need arguments, values which control how the function will carry out its job. For instance, if one wishes to find the absolute value of a number, one must indicate what the number is. Further, Python has a built-in function for computing the absolute value:

COPY CODE

```
#

1 print(abs(4))

2
```

3 print(abs(-4))

4

In this instance, the arguments to the abs function are 4 and -4.
Moreover, some functions take up more than one argument. For instance, the math module comprises a function which we refer to as pow which takes two arguments, the base and the exponent.

COPY CODE

#

1 print(abs(4))

2

3 print(abs(-4))

4

In this instance, the arguments to the abs function are 4 and -4.
max can be sent any number of arguments, which separates by commas, thus, it will return the maximum value sent. Similarly, the arguments may either be simple values or expressions.

Moreover, in the last instance, you will see 503 is returned. Thus, it is larger than 33, 125, and 1. Please note that max also works on lists of values. In addition, functions like range, int, abs all return values that can come into use for building more complex expressions.

Therefore, you see that a significant difference between these functions and drawSquare is that the latter was not executed as it was required to compute a value. Alternatively, drawSquare was written for executing a sequence of steps that cause the turtle to draw a particular shape. Further, we sometimes refer to functions returning values as fruitful functions.

Scope in Python | Variable Scope, Lifetime

Using a function defines the concept of **scope in Python**.
If we define a variable inside a function or pass a value to a function's parameter, the value of that variable is only really accessible within that function. This is called scope.
In other words, the scope is the region of a program where we can access a particular identifier or variable. This is called scope of variable or simply variable scope.
Variables declared inside a program may not be accessible at all locations of that program. It depends on the where we have declared a variable in a program.
Most variables that we define in Python are local in scope to their own function or class. Consider the following example code below.
Example 1:

```
# Python program demonstrates the variable's local scope.
def localScope():
    name = 'John'
    print(name) # Accessible.
# Main program.
```

```
print(name) # Not accessible because the variable name is undefined here.
localScope() # calling function.
```

Output:

```
    print(name) # Not accessible because the variable name is undefined here.
    NameError: name 'name' is not definedIn the above example, the variable name defined inside the function will be accessible within that entire function. Therefore, we have easily accessed and printed the value of variable name on the console.
```

But, when we accessed from the main program that calls the function, then we got NameError. This is because the variable defined within a function has a local scope and is only visible inside the function, not outside the function.

Variable Scope and Lifetime in Python

The scope of variable is a region of the program where a variable is visible or accessible. Lifetime of a variable is the duration for which a variable exists in the memory. The existence and accessibility depend on the declaration of a variable in the program.

For example, the lifetime of a variable declared inside a function is as long as the function is alive. When the execution of function body is finished, then the variable defined inside a function will destroy.

Types of Scope of Variables in Python

There are two basic types of scope of *variables in Python*. They are:

- Global scope
- Local scope

Let's understand each type of variable scope with the help of examples.

Global Scope in Python

When we define a variable inside the main program but outside the function body, then it has a global scope. The variable declared in the main body of the program or file is a global variable.

In the global scope, the global variable will be visible throughout the program or file, and also inside any file which imports that file. We can easily access a variable defined in global scope from all kinds of functions and blocks.

Let's take some important example programs based on the global scope in Python.

Example 2:

Let's write a program in Python in which we will define a global variable and access it from both inside the function and main program body.

```
# Declaring a global variable in the global scope.
x = 50
# Declare a simple function that prints the value of x.
def showMe():
   print('Value of x from local scope = ',x) # calling variable x inside the function.

# Main program.
showMe() # calling function.
print('Value of x from global scope = ',x)
```

Output:

```
    Value of x from local scope =  50
    Value of x from global scope =  50
```

In this example, variable x is a global variable. We have accessed it from both inside the function and outside the function because it has a global scope.

Local Scope

A variable defined or created inside a function body or a block has a local scope. We can access it only within the declared function or block and not from outside that function or block.

As the execution of function body or block is finished, Python destroys the local variable from the memory. In other words, local variable exists in the memory as long as the function is executing.
Consider the following example code below in which we will define a local variable within in a function and we will access it inside a function and from outside the function.
Example 3:

```
def my_funct():
    msg = 'Good morning!' # local variable with local scope.
    print(msg) # accessing local variable from inside the function.
my_funct()
print(msg) # accessing local variable from outside the function.
```

Output:

```
Good morning!
print(msg) # accessing local variable from outside the function.
NameError: name 'msg' is not defined
```

As you can see in the output, as we called local variable from outside the function or main program, we got error because Python destroyed the local variable after the execution of function. That is, the local variable does not exist outside the function.
When we define a variable inside the function, it is not related to any way to another variable with the same name used outside the function.
However, a variable defined outside a function will be read inside a function only when the function does not change the value. Let's an example of it.
Example 4:

```
# Creating a variable city and set to New York.
city = 'New York' # global scope.
def showMe():
    city = 'Dhanbad' # local scope within a function.
    print('City:',city)
# Function call
showMe()
print('City:',city) # accessing global variable from the global scope.
```

Output:

```
City: Dhanbad
City: New York
```

In the above program code, first we have assigned a value 'New York' to the city, and then printed the value of city on the console after calling the function.
Next, we have created a function named showMe(). Inside the function showMe(), we have defined a variable with the same name as that of global variable but with a change value.
When we accessed this variable, then Python prints the change value on the console, not the value of global variable because it is not related to any way to another variable with the same name used outside the function.
As the execution of function body is completed, the value of city is destroyed because it is a local variable to that function only.

Parameters defined in Function Definition

We define parameter names in the function definition; it behaves like local variables. But they contain some values that we pass into the function to perform the desired action when we call it.
Let's take an example program based on it.
Example 5:

```
def showMe(msg): # function header with one formal parameter.
    print(msg) # local scope.
# Main program.
```

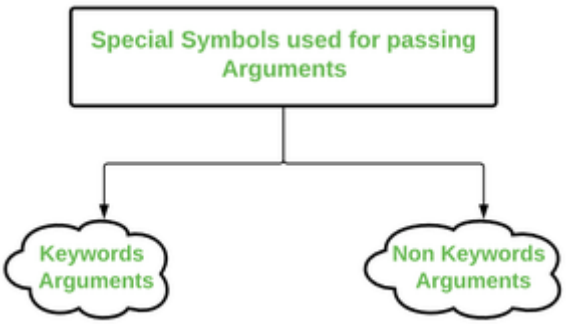
```
showMe('Hi John, Good Morning!')
print(msg) # being local variable, it does not exist outside the function.
```

Output:

```
Hi John, Good Morning!
Traceback (most recent call last):
File "C:\Python Project\Scope.py", line 5, in
 print(msg) # being local variable, it does not exist outside the function.
 NameError: name 'msg' is not defined
```

***args and **kwargs in Python**

** (double star/asterisk) and * (star/asterisk) do for parameters in Python, We can pass a variable number of arguments to a function using special symbols.
There are two special symbols:



**args and **kwargs in Python*

Special Symbols Used for passing arguments in Python:

- *args (Non-Keyword Arguments)
- **kwargs (Keyword Arguments)

Note: *"We use the "wildcard" or "*" notation like this – *args OR **kwargs – as our function's argument when we have doubts about the number of arguments we should pass in a function."*

What is Python *args?

The special syntax **args* in function definitions in Python is used to pass a variable number of arguments to a function. It is used to pass a non-keyworded, variable-length argument list.

- The syntax is to use the symbol * to take in a variable number of arguments; by convention, it is often used with the word args.
- What **args* allows you to do is take in more arguments than the number of formal arguments that you previously defined. With **args*, any number of extra arguments can be tacked on to your current formal parameters (including zero extra arguments).
- For example, we want to make a multiply function that takes any number of arguments and is able to multiply them all together. It can be done using *args.
- Using the *, the variable that we associate with the * becomes iterable meaning you can do things like iterate over it, run some higher-order functions such as map and filter, etc.

Example 1:

Python program to illustrate *args for a variable number of arguments

python3

```
def myFun(*argv):
    for arg in argv:
        print(arg)
myFun('Hello', 'Welcome', 'to', 'CS DEPT')
```

Output:

Hello
Welcome
to
CS DEPT

Example 2:

Python program to illustrate *args with a first extra argument

Python3

```
def myFun(arg1, *argv):
    print("First argument :", arg1)
    for arg in argv:
        print("Next argument through *argv :", arg)
 myFun('Hello', 'Welcome', 'to', 'CS DEPT')Output:
```

First argument : Hello
Next argument through *argv : Welcome
Next argument through *argv : to
Next argument through *argv : CS DEPT

What is Python **kwargs?

The special syntax ***kwargs* in function definitions in Python is used to pass a keyworded, variable-length argument list. We use the name *kwargs* with the double star. The reason is that the double star allows us to pass through keyword arguments (and any number of them).

- A keyword argument is where you provide a name to the variable as you pass it into the function.
- One can think of the *kwargs* as being a dictionary that maps each keyword to the value that we pass alongside it. That is why when we iterate over the *kwargs* there doesn't seem to be any order in which they were printed out.

Example 1:

Python program to illustrate *kwargs for a variable number of keyword arguments. Here **kwargs accept keyworded variable-length argument passed by the function call. for first='CS DEPT' first is key and 'CS DEPT' is a value. in simple words, what we assign is value, and to whom we assign is key.

Python3

```
def myFun(**kwargs):
    for key, value in kwargs.items():
        print("%s == %s" % (key, value))


# Driver code
myFun(first='CS DEPT', mid='for', last='CS DEPT')
```

Output:

first == CS DEPT
mid == for
last == CS DEPT

Example 2:

Python program to illustrate **kwargs for a variable number of keyword arguments with one extra argument. All the same, but one change is we passing non-keyword argument which acceptable by positional argument(arg1 in myFun). and keyword arguments we passing are acceptable by **kwargs. simple right?

Python3

```
def myFun(arg1, **kwargs):
    for key, value in kwargs.items():
```

```
        print("%s == %s" % (key, value))
 # Driver code
myFun("Hi", first='CS DEPT', mid='for', last='CS DEPT')
```

Output:

```
first == CS DEPT
mid == for
last == CS DEPT
```

Using both *args and **kwargs in Python to call a function

Example 1:

Here, we are passing *args and **kwargs as an argument in the myFun function. Passing *args to myFun simply means that we pass the positional and variable-length arguments which are contained by args. so, “CS DEPT” pass to the arg1 , “for” pass to the arg2, and “CS DEPT” pass to the arg3. When we pass **kwargs as an argument to the myFun it means that it accepts keyword arguments. Here, “arg1” is key and the value is “CS DEPT” which is passed to arg1, and just like that “for” and “CS DEPT” pass to arg2 and arg3 respectively. After passing all the data we are printing all the data in lines.

python3

```
def myFun(arg1, arg2, arg3):
    print("arg1:", arg1)
    print("arg2:", arg2)
    print("arg3:", arg3)
 # Now we can use *args or **kwargs to
# pass arguments to this function :
args = ("CS DEPT", "for", "CS DEPT")
myFun(*args)
 kwargs = {"arg1": "CS DEPT", "arg2": "for", "arg3": "CS DEPT"}
myFun(**kwargs)
```

Output:

```
arg1: CS DEPT
arg2: for
arg3: CS DEPT
arg1: CS DEPT
arg2: for
arg3: CS DEPT
```

Example 2:

Here, we are passing *args and **kwargs as an argument in the myFun function. where **‘CS DEPT’, ‘for’, ‘CS DEPT’** is passed as *args, and **first=”CS DEPT”, mid=”for”, last=”CS DEPT”** is passed as **kwargs and printing **in the same line.**

python3

```
def myFun(*args, **kwargs):
    print("args: ", args)
    print("kwargs: ", kwargs)
 # Now we can use both *args ,**kwargs
# to pass arguments to this function :
myFun('CS DEPT', 'for', 'CS DEPT', first="CS DEPT", mid="for", last="CS DEPT")
```

Output:

```
args: ('CS DEPT', 'for', 'CS DEPT')
kwargs: {'first': 'CS DEPT', 'mid': 'for', 'last': 'CS DEPT'}
```

Using *args and **kwargs in Python to set values of object

- *args receives arguments as a **tuple**.
- **kwargs receives arguments as a **dictionary.**

Example 1: using Python *args

Python

```
# defining car class
class car():
    # args receives unlimited no. of arguments as an array
    def __init__(self, *args):
        # access args index like array does
        self.speed = args[0]
        self.color = args[1]
 # creating objects of car class
audi = car(200, 'red')
bmw = car(250, 'black')
mb = car(190, 'white')
 # printing the color and speed of the cars
print(audi.color)
print(bmw.speed)Output:
red
250
```

Example 2: using Python **kwargs

Python

```
# defining car class
class car():
    # args receives unlimited no. of arguments as an array
    def __init__(self, **kwargs):
        # access args index like array does
        self.speed = kwargs['s']
        self.color = kwargs['c']
 # creating objects of car class
audi = car(s=200, c='red')
bmw = car(s=250, c='black')
mb = car(s=190, c='white')
 # printing the color and speed of cars
print(audi.color)
print(bmw.speed)
```

Output:

```
red
250
```

Python Command line arguments

The Python supports the programs that can be run on the command line, complete with command line arguments. It is the input parameter that needs to be passed to the script when executing them. It means to interact with a command-line interface for the scripts. There are different ways we can use command line arguments, few of them are:

1. Python sys module

It is a basic module that comes with Python distribution from the early versions. It is a similar approach as C library using argc/argv to access the arguments. The sys module implements command-line arguments in a simple list structure named sys.argv.

Each list element represents a single argument. The first one -- sys.argv[0] -- is the name of Python script. The other list elements are sys.argv[1] to sys.argv[n]- are the command line arguments 2 to n. As a delimiter between arguments, space is used. Argument values that contain space in it have to be quoted, accordingly.
It stores command-line arguments into a list; we can access it using sys.argv. This is very useful and a simple way to read command-line arguments as String.

Code

```
import sys
# Check the type of sys.argv
print(type(sys.argv))  #  <class ' list '>
# Print the command line arguments
print(' The command line arguments are: ')
# Iterate over sys.argv and print each argument
for i in sys.argv:
   print(i)
```

Output:

```
<class ' list '>
The command line arguments are:
script.py
arg1
arg2
```

2. Python getopt module

The Python getopt module extends the separation of the input string by parameter validation. Based on getopt C function, it allows both short and long options, including a value assignment.
It is very similar to C getopt() function for parsing command line parameters.
It is useful in parsing command line arguments where we want the user to enter some options.

Code

```
import getopt
import sys
argv = sys.argv[1:]
try:
   opts, args = getopt.getopt(argv, 'hm:d', ['help', 'my_file='])
   print(opts)
   print(args)
except getopt.GetoptError:
   # Print a message or do something useful
   print('Something went wrong!')
   sys.exit(2)
```

Output:

```
[('-h', ''), ('-m', 'my_value'), ('--my_file', 'input.txt')]
['arg1', 'arg2']
```

3. Python argparse module

It offers a command-line interface with standardized output, whereas the former two solutions leave most of the work in your hands. argparse allows verification of fixed and optional arguments with a name checking as either UNIX or GNU style. It is the preferred way to parse command-line arguments. It provides a lot of option such as positional arguments, the default value for arguments, helps message, specifying the data type of argument etc.
It makes it easy to write the user-friendly command-line interfaces. It automatically generates help and usage messages and issues errors when a user gives invalid arguments to the program. It means to communicate between the writer of a program and user which does not require going into the code

and making changes to the script. It provides the ability to a user to enter into the command-line arguments.

Code

```
import argparse

# Create an ArgumentParser object
parser = argparse.ArgumentParser(description = 'Example script using argparse')

# Add arguments
parser.add_argument('-f',  '--file', help = 'Specify a file name')
parser.add_argument('-v', '--verbose', action = 'store_true', help = 'Enable verbose mode')

# Parse the command line arguments
args = parser.parse_args()

# Access the argument values
if args.file:
   print(f "File name : {args.file} ")
if args.verbose:
   print("Verbose mode is enabled")
```

Output:

```
python script.py -f myfile.txt -v
File name : myfile.txt
Verbose mode is enabled
```

The above three are the common basic modules to operate command line arguments in Python. Other simple modules in Python for command line arguments are:

Docopt

Docopt is used to create command line interfaces. It simplifies the process of parsing command-line arguments and generating help messages. To use docopt, you need to install the library first. You can install it using pip:

Code

```
from docopt import docopt

__doc__  =  """Usage:
   my_program.py [--option1] [--option2=<value>] <argument>

Options:
   -h, --help         Show this help message.
   -o, --option1      Enable option 1.
   -t, --option2 = <value>  Specify option 2 value.
"""

if __name__ == '__main__':
   arguments = docopt(__doc__, version = 'Example 1')
   print(arguments)
```

Output:

```
$ python script.py --option1 --option2=value argument_value
{
   '--help': False,
   '--option1': True,
```

```
  '--option2': 'value',
  ' ': ' argument_value ',
  '--version': False
}
```

Fire

Python Fire automatically generates a command line interface; you only need one line of code. Unlike the other modules, it works instantly. You don't need to define any arguments; all the methods are linked by default. To install it, type:

1. pip install fire

Define or use a class:

Code

```
import fire
class Python(object):
    def hello(self):
    print("Hello")
      def openfile(self, filename):
       print(" Open file  '" + filename + "'")

if __name__ == '__main__':
  fire.Fire(Python)
```

Output:

```
$ python script.py hello
Hello
$ python script.py openfile my_file.txt
Open file 'my_file.txt'
```

Command Line arguments Modules

Module	Use	Python version
sys	All arguments in sys.argv (basic)	All
argparse	Build a command line interface	>= 2.3
docopt	Created command line interfaces	>= 2.5
fire	Automatically generate command line interfaces (CLIs)	All
optparse	Deprecated	< 2.7

TUPLE:

A Tuple is a python sequence which stores a group of elements or items. Tuples are similar to lists but the main difference is tuples are immutable whereas lists are mutable. Once we create a tuple we cannot modify its elements. Hence, we cannot perform operations like append(), extend(), insert(), remove(), pop() and clear() on tuples. Tuples are generally used to store data which should not be modified and retrieve that data on demand.

Creating Tuples:

We can create a tuple by writing elements separated by commas inside parentheses( ). The elements can be same datatype or different types.

To create an empty tuple, we can simply write empty parenthesis, as:

```
tup=( )
```

To create a tuple with only one element, we can, mention that element in parenthesis and after that a comma is needed. In the absence of comma, python treats the element assign ordinary data type.

```
tup = (10)
print tup          # display 10
print type(tup)    # display <type „int“>
```

```
tup = (10,)
print tup          # display 10
print type(tup)    # display<type „tuple“>
```

To create a tuple with different types of elements:

```
tup=(10, 20, 31.5, „Gudivada“)
```

If we do not mention any brackets and write the elements separating them by comma, then they are taken by default as a tuple.

```
tup= 10, 20, 34, 47
```

It is possible to create a tuple from a list. This is done by converting a list into a tuple using tuple function.

```
n=[1,2,3,4]
tp=tuple(n)
print tp    # display (1,2,3,4)
```

Another way to create a tuple by using range( ) function that returns a sequence.

```
t=tuple(range(2,11,2))
print t      # display (2,4,6,8,10)
```

Accessing the tuple elements:

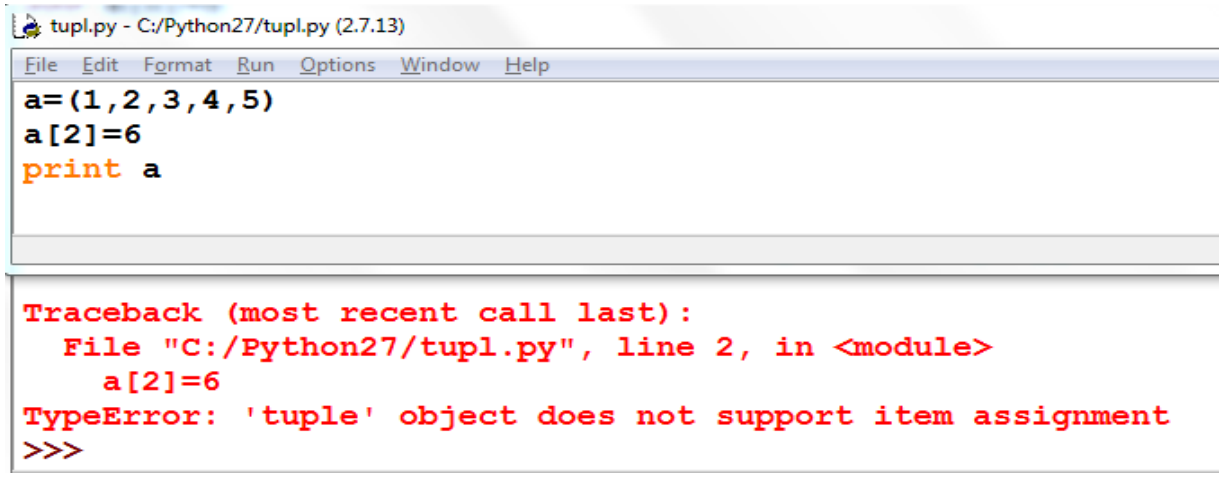
Accessing the elements from a tuple can be done using indexing or slicing. This is same as that of a list. Indexing represents the position number of the element in the tuple. The position starts from 0.

```
tup=(50,60,70,80,90)
print tup[0]     # display 50
print tup[1:4]   # display (60,70,80)
print tup[-1]    # display 90
print tup[-1:-4:-1]      # display (90,80,70)
   print tup[-4:-1]      # display (60,70,80)
```

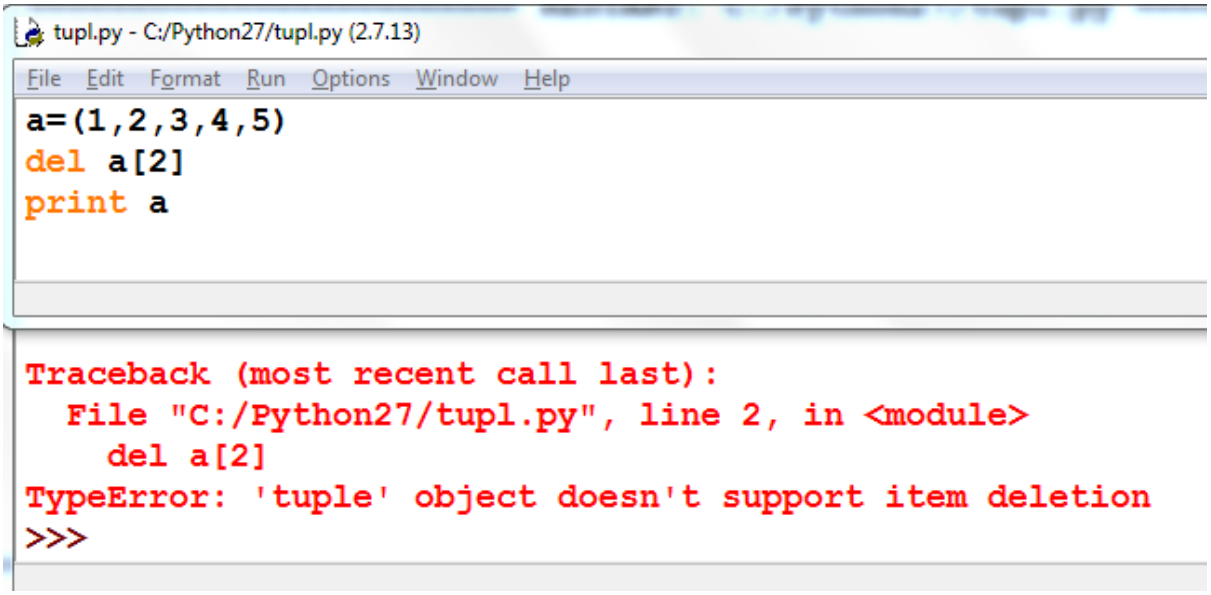
Updating and deleting elements:

Tuples are immutable which means you cannot update, change or delete the values of tuple elements.

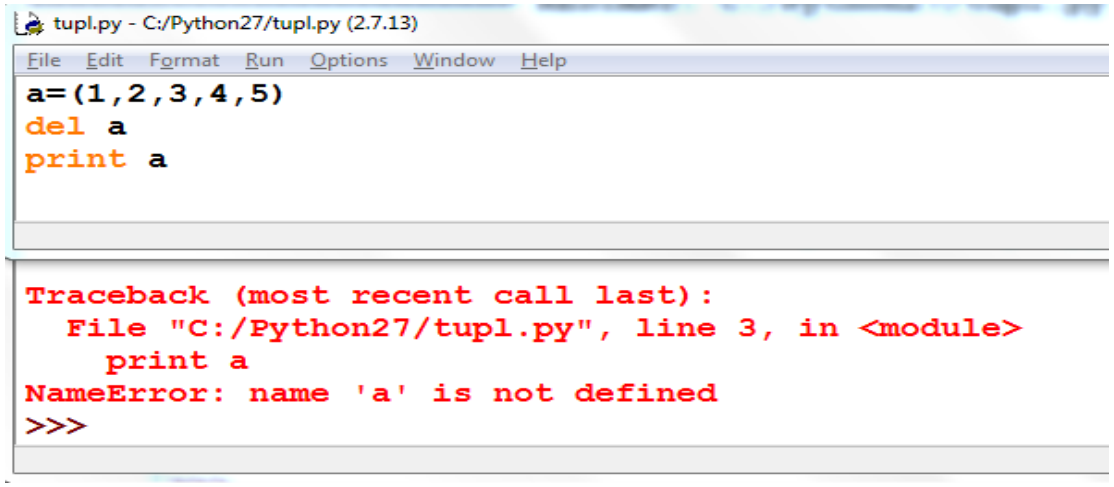
Example-1:



Example-2:



However, you can always delete the entire tuple by using the statement.



Note that this exception is raised because you are trying print the deleted element.

Operations on tuple:

Operation	Description
len(t)	Return the length of tuple.
tup1+tup2	Concatenation of two tuples.
Tup*n	Repetition of tuple values in n number of times.
x in tup	Return True if x is found in tuple otherwise returns False.
cmp(tup1,tup2)	Compare elements of both tuples
max(tup)	Returns the maximum value in tuple.
min(tup)	Returns the minimum value in tuple.
tuple(list)	Convert list into tuple.
tup.count(x)	Returns how many times the element „x‟ is found in tuple.
tup.index(x)	Returns the first occurrence of the element „x‟ in tuple. Raises ValueError if „x‟ is not found in the tuple.
sorted(tup)	Sorts the elements of tuple into ascending order. sorted(tup,reverse=True) will sort in reverse order.

cmp(tuple1, tuple2)

The method **cmp()** compares elements of two tuples.

Syntax

cmp(tuple1, tuple2)

Parameters

tuple1 -- This is the first tuple to be compared

tuple2 -- This is the second tuple to be compared

Return Value

If elements are of the same type, perform the compare and return the result. If elements are different types, check to see if they are numbers.

- If numbers, perform numeric coercion if necessary and compare.
- If either element is a number, then the other element is "larger" (numbers are "smallest").
- Otherwise, types are sorted alphabetically by name.

If we reached the end of one of the tuples, the longer tuple is "larger." If we exhaust both tuples and share the same data, the result is a tie, meaning that 0 is returned.

Example:

```
tuple1 = (123, 'xyz')
tuple2 = (456, 'abc')
print cmp(tuple1, tuple2)     #display -1
print cmp(tuple2, tuple1)     #display 1
```

Nested Tuples:

Python allows you to define a tuple inside another tuple. This is called a *nested tuple*.

```
students=((“RAVI”, “CSE”, 92.00), (“RAMU”, “ECE”, 93.00), (“RAJA”, “EEE”, 87.00))
for i in students:
print i
```

Output: (“RAVI”, “CSE”, 92.00)

("RAMU", "ECE", 93.00)
("RAJA", "EEE", 87.00)

A **tuple** is an ordered and immutable collection of Python objects separated by commas.
Like **lists**, tuples are sequences. Tuples differ from lists in that they can't be modified, whereas lists can, and they use parentheses instead of square brackets.

```
tup=('tutorials', 'point', 2022,True)

print(tup)
```

If you execute the above snippet, produces the following output −

```
('tutorials', 'point', 2022, True)
```

Indexing Tuples

In **Python**, every tuple with elements has a position or index. Each element of the tuple can be accessed or manipulated by using the index number.

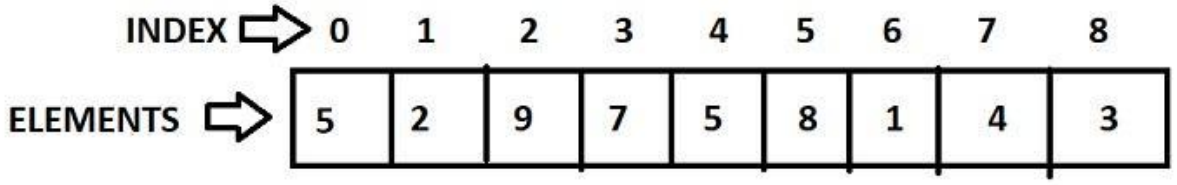
They are two types of indexing −

- Positive Indexing
- Negative Indexing

Positive Indexing

In positive the first element of the tuple is at an index of 0 and the following elements are at +1 and as follows.

In the below figure, we can see how an element in the tuple is associated with its index or position.



Example 1

The following is an example code to show the positive indexing of tuples.

```
tuple= (5,2,9,7,5,8,1,4,3)

print(tuple(3))

print(tuple(7))
```

Output

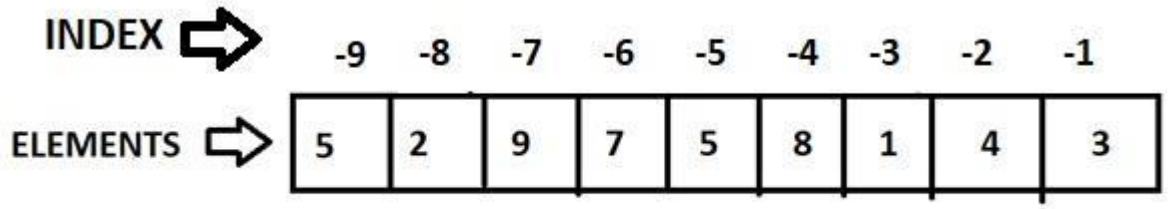
The above code produces the following results

```
7
4
```

Negative Indexing

In negative indexing, the indexing of elements starts from the end of the tuple. That is the last element of the tuple is said to be at a position at -1 and the previous element at -2 and goes on till the first element.

In the below figure, we can see how an element is associated with its index or position of a tuple.



Example

The following is an example code to show the negative indexing in tuples.

```
tuple= (5,2,9,7,5,8,1,4,3)

print(tuple(-2))

print(tuple(-8))
```

Output

The above code produces the following results

```
4
2
```

Slicing tuples

Tuple slicing is a frequent practice in Python, and it is the most prevalent technique used by programmers to solve efficient problems. Consider a Python tuple. You must slice a tuple in order to access a range of elements in it. One method is to utilize the colon as a simple slicing operator (**:**).

The slice operator allows you to specify where to begin slicing, where to stop slicing, and what step to take. Tuple slicing creates a new tuple from an old one.

Syntax
tuple[Start : Stop : Stride]

The above expression returns the portion of the tuple from index *Start* to index *Stop*, at a step size *Stride*.

Example 1

In the following example we have used the slice operation to slice a tuple. We also use negative indexing method to slice a tuple.

```
tuple= ('a','b','c','d','e','f','g','h','i','j')

print(tuple[0:6])

print(tuple[1:9:2])

print(tuple[-1:-5:-2])
```

Output

The above code produces the following results

```
('a', 'b', 'c', 'd', 'e', 'f')
('b', 'd', 'f', 'h')
('j', 'h')
```

Example 2

Following is another example for this −

```
my_tuple = ('t', 'u', 'r', 'i', 'a', 'l', 's', 'p','o', 'i', 'n', 't')

print(my_tuple[1:]) #Print elements from index 1 to end

print(my_tuple[:2]) #Print elements from start to index 2

print(my_tuple[5:12]) #Print elements from index 1 to index 3

print(my_tuple[::5]) #Print elements from start to end using step size
```

Output
```
('u', 'r', 'i', 'a', 'l', 's', 'p', 'o', 'i', 'n', 't')
('t', 'u')
('l', 's', 'p', 'o', 'i', 'n', 't')
('t', 'l', 'n')
```

BUILT-IN FUNCTIONS USED ON TUPLES

Python supports a wide range of data structures, including tuples, which are used to store a sequence of immutable Python objects. Tuples are very similar to lists, but once a tuple is created, it cannot be modified. Python provides various built-in functions to manipulate tuples in different ways. In this article, we will explore tuple functions in Python, the syntax of tuple functions in Python, the return value of tuple functions in Python, and some examples of tuple functions in Python.

What are Tuple Functions in Python?

Tuple functions in Python are built-in functions that are used to create, manipulate, and perform various operations on tuples. Tuple functions perform various tasks on tuples, such as concatenation, slicing, sorting, and searching.

One of the most commonly used tuple functions in Python is the tuple() function. The tuple() function is used to convert other iterable data structures such as lists, strings, or sets into tuples.

Some other important tuple functions in Python include the count() function, which is used to count the number of occurrences of a particular element in a tuple, and the index() function, which is used to find the index of the first occurrence of a particular element in a tuple.

Using these tuple functions in Python, programmers can perform various operations on tuples, making it easy to work with data in Python.

Syntax of Tuple Functions in Python

The syntax of tuple functions in Python varies depending on the function being used. However, most tuple functions follow a similar structure. Here is an overview of the syntax for some commonly used tuple functions in Python:

```
function_name(tuple_object)
```

Here, function_name is the name of the tuple function that you want to use, and tuple_object is the tuple on which you want to perform the function. The parentheses are used to enclose the tuple object, and the function_name is followed by the parentheses enclosing the tuple object.

Return Value of Tuple Functions in Python

The return value of tuple functions in Python depends on the task at hand. Some tuple functions return a single value, while others return tuples. For instance, the len() function returns an integer value that represents the number of elements in a tuple. Similarly, the min() and max() functions return the minimum and maximum values in a tuple, respectively, as single values. Whereas, the tuple() function creates a new tuple from a list, set, or any iterable object.

In short words, the return value of tuple functions in Python is dependent on the specific function being used. Programmers should consult the Python documentation for the specific return values of the functions they are using in their code.

Examples of Tuple Functions in Python

Here are some examples of the most commonly used tuple functions in Python:

1. len() function
   The len() function is used to return the number of elements in a tuple. It takes a tuple as an argument and returns an integer value representing the length of the tuple.

   Code Implementation:

```
my_tuple = (1, 2, 3, 4, 5)

print(len(my_tuple))
```

   Output:

```
5
```

   Explanation:
   In the above example, we have defined a tuple my_tuple with 5 elements. We then used the len() function to get the length of the tuple, which is 5.

2. max() function
   The max() function is used to return the maximum value in a tuple. It takes a tuple as an argument and returns the maximum value in the tuple.

   Code Implementation:

```
my_tuple = (5, 6, 7, 8, 9)

print(max(my_tuple))
```

   Output:

```
9
```

   Explanation:
   In the above example, we have defined a tuple my_tuple with 5 elements. We have then used the max() function to get the maximum value in the tuple, which is 9.

3. min() function
   The min() function is used to return the minimum value in a tuple. It takes a tuple as an argument and returns the minimum value in the tuple.

   Code Implementation:

```
my_tuple = (5, 6, 7, 8, 9)
```

```
print(min(my_tuple))
```

Output:

```
5
```

Explanation:
In the above example, we have defined a tuple my_tuple with 5 elements. We have then used the min() function to get the minimum value in the tuple, which is 5.

4. sum() function
The sum() function is used to return the sum of all elements in a tuple. It takes a tuple as an argument and returns the sum of all the elements in the tuple.

Code Implementation:

```
my_tuple = (1, 2, 3, 4, 5)

print(sum(my_tuple))
```

Output:

```
15
```

Explanation:
In the above example, we have defined a tuple my_tuple with 5 elements. We have then used the sum() function to get the sum of all the elements in the tuple, which is 15.

5. tuple() function
The tuple() function is used to create a tuple from a list, set, or any iterable object. It takes an iterable object as an argument and returns a tuple containing all the elements in the iterable object.

Code Implementation:

```
my_list = [1, 2, 3, 4, 5]

my_tuple = tuple(my_list)

print(my_tuple)
```

Output:

```
(1, 2, 3, 4, 5)
```

Explanation:
In the above example, we have defined a list my_list with 5 elements. We have then used the tuple() function to create a tuple from the list, which contains all the elements in the list.

6. index() function
It returns the index of the first occurrence of a specified element in a tuple

Code Implementation:

```
fruits = ('apple', 'banana', 'orange', 'mango', 'banana')

# Find the index of 'orange'

index = fruits.index('orange')

print(index)
```

Output:

```
2
```

Explanation:
In the example above, we first create a tuple of fruits containing five elements. Then, we use the index() method to find the index of the element ‘orange’. Since ‘orange’ is the third element in the tuple (counting from zero), the index() method returns 2.
If the element we’re looking for isn’t present in the tuple, the index() method will raise a ValueError

7. count() function
It returns the number of times a specified element appears in a tuple.

Code Implementation:

```
my_tuple = (1, 2, 3, 4, 3, 5, 3)

# Count the number of times the element '3' occurs in the tuple

count = my_tuple.count(3)

print(count)
```

Output:

```
3
```

Explanation:
In this example, we have defined a tuple my_tuple with seven elements. We then use the count() function to count the number of times element 3 occurs in the tuple. The count() function returns the value 3 since element 3 appears in the tuple three times.

Tuple Assignment

Introduction

Tuples are basically a data type in python. These tuples are an ordered collection of elements of different data types. Furthermore, we represent them by writing the elements inside the parenthesis separated by commas. We can also define tuples as lists that we cannot change. Therefore, we can call them immutable tuples. Moreover, we access elements by using the index starting from zero. We can create a tuple in various ways. Here, we will study tuple assignment which is a very useful feature in python.

Tuple Assignment

In python, we can perform tuple assignment which is a quite useful feature. We can initialise or create a tuple in various ways. Besides tuple assignment is a special feature in python. We also call this feature unpacking of tuple.

The process of assigning values to a tuple is known as packing. While on the other hand, the unpacking or tuple assignment is the process that assigns the values on the right-hand side to the left-hand side variables. In unpacking, we basically extract the values of the tuple into a single variable.

Moreover, while performing tuple assignments we should keep in mind that the number of variables on the left-hand side and the number of values on the right-hand side should be equal. Or in other words, the number of variables on the left-hand side and the number of elements in the tuple should be equal. Let us look at a few examples of packing and unpacking.

tuple assignment

Tuple Packing (Creating Tuples)

We can create a tuple in various ways by using different types of elements. Since a tuple can contain all elements of the same data type as well as of mixed data types as well. Therefore, we have multiple ways of creating tuples. Let us look at few examples of creating tuples in python which we consider as packing.

Example 1: Tuple with integers as elements

COPY CODE

```
>>>tup = (22, 33, 5, 23)

>>>tup

 (22, 33, 5, 23)
```

Example 2: Tuple with mixed data type

COPY CODE

```
>>>tup2 = ('hi', 11, 45.7)

>>>tup2

 ('hi', 11, 45.7)
```

Example 3: Tuple with a tuple as an element

COPY CODE

```
>>>tup3 = (55, (6, 'hi'), 67)

>>>tup3

 (55, (6, 'hi'), 67)
```

Example 4: Tuple with a list as an element

COPY CODE

```
>>>tup3 = (55, [6, 9], 67

>>>tup3

(55, [6, 9], 67)
```

If there is only a single element in a tuple we should end it with a comma. Since writing, just the element inside the parenthesis will be considered as an integer

For example,

COPY CODE

```
>>>tup=(90)

>>>tup

90
```

```
>>>type(tup)
<class 'int'>
```

Correct way of defining a tuple with single element is as follows:

COPY CODE

```
>>>tup=(90,)
>>>tup
(90,)
>>>type(tup)
<class 'tuple'>
```

Moreover, if you write any sequence separated by commas, python considers it as a tuple.

For example,

COPY CODE

```
>>> seq = 22, 4, 56
>>>seq
(22, 4, 56)
>>>type(seq)
<class 'tuple'>
```

Browse more Topics Under Tuples and its Functions

Immutable Tuples

Creating Tuples

Initialising and Accessing Elements in a Tuple

Tuple Slicing

Tuple Indexing

Tuple Functions

Tuple Assignment (Unpacking)

Unpacking or tuple assignment is the process that assigns the values on the right-hand side to the left-hand side variables. In unpacking, we basically extract the values of the tuple into a single variable.

Example 1

```
>>>(n1, n2) = (99, 7)
>>>print(n1)
99
>>>print(n2)
7
```

Example 2

```
>>>tup1 = (8, 99, 90, 6.7)
>>>(roll no., english, maths, GPA) = tup1
>>>print(english)
99
>>>print(roll no.)
8
>>>print(GPA)
6.7
>>>print(maths)

90
```

Example 3

```
>>> (num1, num2, num3, num4, num5) = (88, 9.8, 6.8, 1)
#this gives an error as the variables on the left are more than the number of elements in the tuple
ValueError: not enough values to unpack
(expected 5, got 4)
```

Python zip()

The zip() function takes iterables (can be zero or more), aggregates them in a tuple, and returns it.

Example

```
languages = ['Java', 'Python', 'JavaScript']

versions = [14, 3, 6]


result = zip(languages, versions)

print(list(result))


# Output: [('Java', 14), ('Python', 3), ('JavaScript', 6)]
```

Run Code

Syntax of zip()

The syntax of the zip() function is:

```
zip(*iterables)
```

zip() Parameters

Parameter	Description
iterables	can be built-in iterables (like: list, string, dict), or user-defined iterables

Recommended Reading: Python Iterators, __iter__ and __next__

zip() Return Value

The zip() function returns an iterator of tuples based on the iterable objects.

If we do not pass any parameter, zip() returns an empty iterator

If a single iterable is passed, zip() returns an iterator of tuples with each tuple having only one element.

If multiple iterables are passed, zip() returns an iterator of tuples with each tuple having elements from all the iterables.

Suppose, two iterables are passed to zip(); one iterable containing three and other containing five elements. Then, the returned iterator will contain three tuples. It's because the iterator stops when the shortest iterable is exhausted.

Example 1: Python zip()

```
number_list = [1, 2, 3]

str_list = ['one', 'two', 'three']


# No iterables are passed

result = zip()


# Converting iterator to list

result_list = list(result)

print(result_list)


# Two iterables are passed

result = zip(number_list, str_list)


# Converting iterator to set

result_set = set(result)

print(result_set)
```

Run Code

Output

```
[]

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

Example 2: Different number of iterable elements

```
numbersList = [1, 2, 3]

str_list = ['one', 'two']

numbers_tuple = ('ONE', 'TWO', 'THREE', 'FOUR')


# Notice, the size of numbersList and numbers_tuple is different

result = zip(numbersList, numbers_tuple)


# Converting to set

result_set = set(result)

print(result_set)


result = zip(numbersList, str_list, numbers_tuple)


# Converting to set

result_set = set(result)

print(result_set)
```

Run Code

Output

```
{(2, 'TWO'), (3, 'THREE'), (1, 'ONE')}

{(2, 'two', 'TWO'), (1, 'one', 'ONE')}
```

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

zip(*zippedList)

Example 3: Unzipping the Value Using zip()

```
coordinate = ['x', 'y', 'z']

value = [3, 4, 5]


result = zip(coordinate, value)

result_list = list(result)

print(result_list)


c, v =  zip(*result_list)

print('c =', c)

print('v =', v)
```

Run Code

Output

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

c = ('x', 'y', 'z')

v = (3, 4, 5)
```

Working with Sets

You can think of sets in python equivalent to sets in Mathematics. A set is created by placing all the items (elements) inside curly braces {}, separated by a comma. A set can store data in the form of a string literal, integer or any allowed data type in Python. You cannot access elements of the set using an index. Though, you can traverse a set using for loop. There are different methods or operations that can be performed on sets such as union, update, intersection, intersection_update, difference, difference_update, symmetric_difference, symmetric_difference_update, issubset, issuperset and lot more.

Initializing and Traversing a Set

In the below code, we will demonstrate, how to initialize a set and traverse the set using for loop. Sets can be created using the function set().

Code:

```
a = {1, 2, 3}

for i in a:

    print(i)
```

Output:

1

2

3

How to add elements to a Set

Sets are mutable. But since they are unordered, indexing has no meaning.

We cannot access or change an element of a set using indexing or slicing. Sets does not support it.

Ezoic

We can add a single element using the add() method and multiple elements using the update() method. The update() method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

Code:

```
a = {1, 2, 3}

for i in a:

    print(i)

a.add(4)

print(a)

#We are adding list as well as set to a set
```

```
#We are also trying to add duplicate element 1 to existing set

a.update([5, 6, 7], {1, 9, 10})

print(a)
```

Output:

```
1

2

3

{1, 2, 3, 4}

{1, 2, 3, 4, 5, 6, 7, 9, 10}
```

Remove elements from a Set

A particular item can be removed from the set using methods, discard() and remove().

The only difference between the two is that, while using discard() if the item does not exist in the set, it remains unchanged. But remove() will raise an error in such condition.

Ezoic

Similarly, we can remove and return an item using the pop() method.

Set being unordered, there is no way of determining which item will be popped. It is completely arbitrary.

We can also remove all items from a set using clear().

Code:

```
a_set = {1, 2, 3, 4, 5, 6}

print(a_set)

a_set.discard(3)

print(a_set)

a_set.remove(5)

print(a_set)

a_set.pop()

print(a_set)
```

```
a_set.clear()
print(a_set)
```

Output:

```
{1, 2, 3, 4, 5, 6}
{1, 2, 4, 5, 6}
{1, 2, 4, 6}
{2, 4, 6}
set()
```

Set Membership Test

We can test if an item exists in a set or not, using the keyword in

Code:

```
a_set = {1, 2, 3, 4, 5, 6}
print(1 in a_set)
print(101 in a_set)
```

Output:

```
True
False
```

Set Operations similar to Mathematics

Set Union operation

Union of two sets X and Y is a set of all unique elements from both sets.

A union is performed using | operator. The same can be accomplished using the method union().

Code:

```
a_set = {1, 2, 3, 4, 5, 6}
```

```
b_set = {1, 7, 8, 9, 10}

c_set = a_set | b_set

print(c_set)

c_set = a_set.union(b_set)

print(c_set)
```

Output:

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Set Intersection operation

The intersection of two sets X and Y is a set of elements that are common in both sets.

The intersection is performed using & operator. The same can be accomplished using the method intersection().

Ezoic

Code:

```
a_set = {1, 2, 3, 4, 5, 6}

b_set = {1, 7, 8, 9, 10}

c_set = a_set & b_set

print(c_set)

c_set = a_set.intersection(b_set)

print(c_set)
```

Output:

{1}

{1}

Set Difference operation

A difference of two sets X and Y refers to (X – Y) is a set of elements that are only in X but not in Y. Similarly, Y – X is a set of the element in Y but not in X.

The difference is performed using – minus operator. The same can be accomplished using the method difference().

Code:

```
a_set = {1, 2, 3, 4, 5, 6}

b_set = {1, 7, 8, 9, 10}

c_set = a_set - b_set

print(c_set)

c_set = a_set.difference(b_set)

print(c_set)

d_set = b_set - a_set

print(d_set)

d_set = b_set.difference(a_set)

print(d_set)
```

Output:

{2, 3, 4, 5, 6}

{2, 3, 4, 5, 6}

{8, 9, 10, 7}

{8, 9, 10, 7}

Set Symmetric Difference operation

Symmetric Difference of two sets X and Y is a set of elements in both X and Y except those that are common in both.

The symmetric difference is performed using ^ operator. The same can be accomplished using the method symmetric_difference().

Ezoic

Code:

```
a_set = {1, 2, 3, 4, 5, 6}

b_set = {1, 7, 8, 9, 10}

c_set = a_set ^ b_set

print(c_set)

c_set = a_set.symmetric_difference(b_set)

print(c_set)
```

Output:

{2, 3, 4, 5, 6, 7, 8, 9, 10}

{2, 3, 4, 5, 6, 7, 8, 9, 10}

Other Functions you can try

Function Name	Description
add()	Adds an element to the set
copy()	Returns a copy of the set
difference_update()	Removes all elements of another set from this set

intersection_update() Updates the set with the intersection of itself and another

isdisjoint() Returns True if two sets have a null intersection

issubset() Returns True if another set contains this set

issuperset() Returns True if this set contains another set

symmetric_difference_update() Updates a set with the symmetric difference of itself and another

Built-in Functions with Set

Built-in functions like all(), any(), enumerate(), len(), max(), min(), sorted(), sum() etc. are commonly used with set to perform different tasks.

Name of Function Usage

all() Return True if all elements of the set are true (or if the set is empty).

any() Return True if any element of the set is true. If the set is empty, return False.

enumerate() Return an enumerate object. It contains the index and value of all the items of set as a pair.

len() Return the length (the number of items) in the set.

max() Return the largest item in the set.

min() Return the smallest item in the set.

sorted() Return a new sorted list from elements in the set(does not sort the set itself).

sum() Return the sum of all elements in the set.

Code:

```
a_set = {10, 3, 4, 0, 9, 123, -1, 3}

print(any(a_set))
```

```
print(all(a_set))

print(enumerate(a_set))

print(len(a_set))

print(max(a_set))

print(min(a_set))

print(sorted(a_set))

print(sum(a_set))
```

Output:

```
True

False

<enumerate object at 0x000001C818F6F1D8>

7

123

-1

[-1, 0, 3, 4, 9, 10, 123]

148
```

Working with Frozenset

Set elements or values can be modified and changed. Sometimes, we need to use values that do not require modification or updation once assigned. Frozensets are immutable sets whose values cannot be changed once assigned. Sets being mutable are unhashable, so they can't be used as dictionary keys. On the other hand, frozensets are hashable and can be used as keys to a dictionary.

Frozensets can be created using the function frozenset().

This datatype supports methods like copy(), difference(), intersection(), isdisjoint(), issubset(), issuperset(), symmetric_difference() and union(). Being immutable it does not have method that add or remove elements.

Code:

```
A = frozenset([1, 2, 3, 4])

B = frozenset([3, 4, 5, 6])

X = frozenset([1, 2])

c = A.union(B)

print(c)

c = A.difference(B)

print(c)

c = A.intersection(B)

print(c)

c = A.isdisjoint(B)

print(c)

c = X.issubset(A)

print(c)

c = A.issuperset(X)

print(c)

c = A.union(B)

print(c)
```

Output:

```
frozenset({1, 2, 3, 4, 5, 6})

frozenset({1, 2})

frozenset({3, 4})

False

True

True

frozenset({1, 2, 3, 4, 5, 6})
```