

Python Functions

Functions are the most important aspect of an application. A function can be defined as the organized block of reusable code which can be called whenever required.

Python allows us to divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by `{}`. A function can be called multiple times to provide reusability and modularity to the python program.

In other words, we can say that the collection of functions creates a program. The function is also known as procedure or subroutine in other programming languages.

Python provide us various inbuilt functions like `range()` or `print()`. Although, the user can create its functions which can be called user-defined functions.

Advantage of Functions in Python

There are the following advantages of Python functions.

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call python functions any number of times in a program and from any place in a program.
- We can track a large python program easily when it is divided into multiple functions.
- Reusability is the main achievement of python functions.
- However, Function calling is always overhead in a python program.

Creating a function

In python, we can use **def** keyword to define the function. The syntax to define a function in python is given below.

1. **def** my_function():
2. function-suite
3. **return** <expression>

```
def my_function():  
    function-suite  
    return <expression>
```

The function block is started with the colon (:) and all the same level block statements remain at the same indentation.

A function can accept any number of parameters that must be the same in the definition and function calling.

Function calling

In python, a function must be defined before the function calling otherwise the python interpreter gives an error. Once the function is defined, we can call it from another function or the python prompt. To call the function, use the function name followed by the parentheses.

A simple function that prints the message "Hello Word" is given below.

1. **def** hello_world():
2. **print**("hello world")
- 3.
4. hello_world()

```
def hello_world():  
    print("hello world")  
  
hello_world()
```

Output:

```
hello world
```

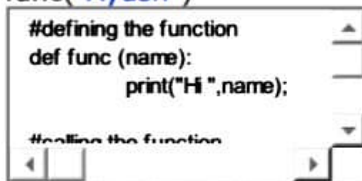
Parameters in function

The information into the functions can be passed as the parameters. The parameters are specified in the parentheses. We can give any number of parameters, but we have to separate them with a comma.

Consider the following example which contains a function that accepts a string as the parameter and prints it.

Example 1

1. #defining the function
2. def func (name):
3. print("Hi ",name);
- 4.
5. #calling the function
6. func("Ayush")

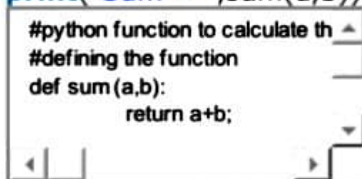


```
#defining the function
def func (name):
    print("Hi ",name);

#calling the function
func("Ayush")
```

Example 2

1. #python function to calculate the sum of two variables
2. #defining the function
3. def sum (a,b):
4. return a+b;
- 5.
6. #taking values from the user
7. a = int(input("Enter a: "))
8. b = int(input("Enter b: "))
- 9.
10. #printing the sum of a and b
11. print("Sum = ",sum(a,b))



```
#python function to calculate th
#defining the function
def sum(a,b):
    return a+b;

#taking values from the user
a = int(input("Enter a: "))
b = int(input("Enter b: "))

#printing the sum of a and b
print("Sum = ",sum(a,b))
```

Output:

```
Enter a: 10
Enter b: 20
Sum = 30
```

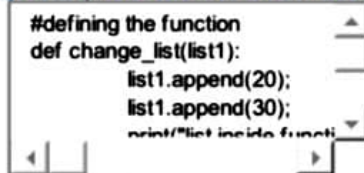
Call by reference in Python

In python, all the functions are called by reference, i.e., all the changes made to the reference inside the function revert back to the original value referred by the reference.

However, there is an exception in the case of mutable objects since the changes made to the mutable objects like string do not revert to the original string rather, a new string object is made, and therefore the two different objects are printed.

Example 1 Passing Immutable Object (List)

1. #defining the function
2. `def change_list(list1):`
3. `list1.append(20);`
4. `list1.append(30);`
5. `print("list inside function = ",list1)`
- 6.
7. #defining the list
8. `list1 = [10,30,40,50]`
- 9.
10. #calling the function
11. `change_list(list1);`
12. `print("list outside function = ",list1);`



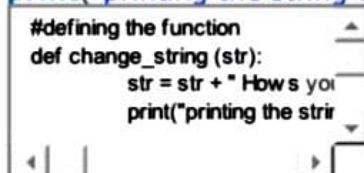
```
#defining the function
def change_list(list1):
    list1.append(20);
    list1.append(30);
    print("list inside functi
```

Output:

```
list inside function = [10, 30, 40, 50, 20, 30]
list outside function = [10, 30, 40, 50, 20, 30]
```

Example 2 Passing Mutable Object (String)

1. #defining the function
2. `def change_string (str):`
3. `str = str + " Hows you";`
4. `print("printing the string inside function :",str);`
- 5.
6. `string1 = "Hi I am there"`
- 7.
8. #calling the function
9. `change_string(string1)`
- 10.
11. `print("printing the string outside function :",string1)`



```
#defining the function
def change_string (str):
    str = str + " Hows you
    print("printing the str
```

Output:

```
printing the string inside function : Hi I am there Hows you
printing the string outside function : Hi I am there
```

Types of arguments

There may be several types of arguments which can be passed at the time of function calling.

1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

Required Arguments

Till now, we have learned about function calling in python. However, we can provide the arguments at the time of function calling. As far as the required arguments are concerned, these are the arguments which are required to be passed at the time of function calling with the exact match of their positions in the function call and function definition. If either of the arguments is not provided in the function call, or the position of the arguments is changed, then the python interpreter will show the error.

Consider the following example.

Example 1

1. `#the argument name is the required argument to the function func`
2. `def func(name):`
3. `message = "Hi "+name;`
4. `return message;`
5. `name = input("Enter the name?")`
6. `print(func(name))`

Output:

```
Enter the name?John
Hi John
```

Example 2

1. `#the function simple_interest accepts three arguments and returns the simple interest accordingly`
2. `def simple_interest(p,t,r):`
3. `return (p*t*r)/100`
4. `p = float(input("Enter the principle amount? "))`
5. `r = float(input("Enter the rate of interest? "))`
6. `t = float(input("Enter the time in years? "))`
7. `print("Simple Interest: ",simple_interest(p,r,t))`

Output:

```
Enter the principle amount? 10000
```

```
Enter the rate of interest? 5
Enter the time in years? 2
Simple Interest: 1000.0
```

Example 3

1. `#the function calculate returns the sum of two arguments a and b`
2. `def calculate(a,b):`
3. `return a+b`
4. `calculate(10) # this causes an error as we are missing a required arguments b.`

Output:

```
TypeError: calculate() missing 1 required positional argument: 'b'
```

Keyword arguments

Python allows us to call the function with the keyword arguments. This kind of function call will enable us to pass the arguments in the random order.

The name of the arguments is treated as the keywords and matched in the function calling and definition. If the same match is found, the values of the arguments are copied in the function definition.

Consider the following example.

Example 1

1. `#function func is called with the name and message as the keyword arguments`
2. `def func(name,message):`
3. `print("printing the message with",name,"and ",message)`
4. `func(name = "John",message="hello") #name and message is copied with the values John and hello respectively`

Output:

```
printing the message with John and hello
```

Example 2 providing the values in different order at the calling

1. `#The function simple_interest(p, t, r) is called with the keyword arguments the order of arguments doesn't matter in this case`
2. `def simple_interest(p,t,r):`
3. `return (p*t*r)/100`
4. `print("Simple Interest: ",simple_interest(t=10,r=10,p=1900))`

Output:

Simple Interest: 1900.0

If we provide the different name of arguments at the time of function call, an error will be thrown.

Consider the following example.

Example 3

1. `#The function simple_interest(p, t, r) is called with the keyword arguments.`
2. `def simple_interest(p,t,r):`
3. `return (p*t*r)/100`
- 4.
5. `print("Simple Interest: ",simple_interest(time=10,rate=10,principle=1900)) # doesn't find the exact match of the name of the arguments (keywords)`

```
#The function simple_interest(p
def simple_interest(p,t,r):
    return (p*t*r)/100

print("Simple Interest: ", simple_i
```

Output:

```
TypeError: simple_interest() got an unexpected keyword argument 'time'
```

The python allows us to provide the mix of the required arguments and keyword arguments at the time of function call. However, the required argument must not be given after the keyword argument, i.e., once the keyword argument is encountered in the function call, the following arguments must also be the keyword arguments.

Consider the following example.

Example 4

1. `def func(name1,message,name2):`
2. `print("printing the message with",name1,"",message,"and",name2)`
3. `func("John",message="hello",name2="David") #the first argument is not the keyword argument`

```
def func(name1,message,name2):
    print("printing the message w
func("John",message="hello",ni
```

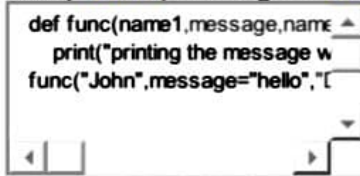
Output:

```
printing the message with John , hello ,and David
```

The following example will cause an error due to an in-proper mix of keyword and required arguments being passed in the function call.

Example 5

1. `def func(name1,message,name2):`
2. `print("printing the message with",name1,"",message,"and",name2)`
3. `func("John",message="hello","David")`



```
def func(name1,message,name2):
    print("printing the message w
func("John",message="hello","D
```

Output:

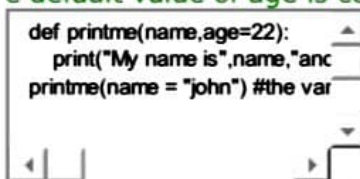
```
SyntaxError: positional argument follows keyword argument
```

Default Arguments

Python allows us to initialize the arguments at the function definition. If the value of any of the argument is not provided at the time of function call, then that argument can be initialized with the value given in the definition even if the argument is not specified at the function call.

Example 1

1. `def printme(name,age=22):`
2. `print("My name is",name,"and age is",age)`
3. `printme(name = "john")` #the variable age is not passed into the function however the default value of age is considered in the function



```
def printme(name,age=22):
    print("My name is",name,"anc
printme(name = "john") #the var
```

Output:

```
My name is john and age is 22
```

Example 2

1. `def printme(name,age=22):`
2. `print("My name is",name,"and age is",age)`
3. `printme(name = "john")` #the variable age is not passed into the function however the default value of age is considered in the function
4. `printme(age = 10,name="David")` #the value of age is overwritten here, 10 will be printed as age

```
def printme(name,age=22):
    print("My name is",name,"and age is",age)
    printme(name = "john") #the var
    printme(age = 10,name="David")
```

Output:

```
My name is john and age is 22
My name is David and age is 10
```

Variable length Arguments

In the large projects, sometimes we may not know the number of arguments to be passed in advance. In such cases, Python provides us the flexibility to provide the comma separated values which are internally treated as tuples at the function call.

However, at the function definition, we have to define the variable with * (star) as * <variable - name >.

Consider the following example.

Example

1. **def** printme(*names):
2. **print**("type of passed argument is ",type(names))
3. **print**("printing the passed arguments...")
4. **for** name **in** names:
5. **print**(name)
6. **printme**("john","David","smith","nick")

```
def printme(*names):
    print("type of passed argume
    print("printing the passed arg
    for name in names:
        print(name)
```

Output:

```
type of passed argument is <class 'tuple'>
printing the passed arguments...
john
David
smith
nick
```

Scope of variables

The scopes of the variables depend upon the location where the variable is being declared. The variable declared in one part of the program may not be accessible to the other parts.

In python, the variables are defined with the two types of scopes.

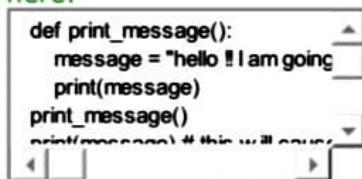
1. Global variables
2. Local variables

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.

Consider the following example.

Example 1

1. `def print_message():`
2. `message = "hello !! I am going to print a message." # the variable message is local to the function itself`
3. `print(message)`
4. `print_message()`
5. `print(message) # this will cause an error since a local variable cannot be accessible here.`



```
def print_message():  
    message = "hello !! I am going  
    print(message)  
print_message()  
print(message) # this will cause
```

Output:

```
hello !! I am going to print a message.
```

```
File "/root/PycharmProjects/PythonTest/Test1.py", line 5, in
```

```
    print(message)
```

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

Example 2

1. `def calculate(*args):`
2. `sum=0`
3. `for arg in args:`
4. `sum = sum + arg`
5. `print("The sum is",sum)`
6. `sum=0`
7. `calculate(10,20,30) #60 will be printed as the sum`
8. `print("Value of sum outside the function:",sum) # 0 will be printed`

```
def calculate(*args):  
    sum=0  
    for arg in args:  
        sum = sum +arg  
    print("The sum is" sum)
```

Output:

The sum is 60

Value of sum outside the function: 0