Python Operators

Operators are used to perform operations on variables and values.

In the example below, we use the + operator to add together two values:

Example

```
print(10 + 5)
```

Python divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Python Arithmetic Operators

- Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Description	Syntax
+	Addition: adds two operands	x + y
–	Subtraction: subtracts two operands	x – y
*	Multiplication: multiplies two operands	x * y
/	Division (float): divides the first operand by the second	x / y
//	Division (floor): divides the first operand by the second	x // y
%	Modulus: returns the remainder when the first operand is divided by the second	x % y
**	Power: Returns first raised to power second	x ** y

Python Assignment Operators

- Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Python Comparison Operators

- Comparison operators are used to compare two values:

Operator	**Description**	**Syntax**
>	Greater than: True if the left operand is greater than the right	x > y
<	Less than: True if the left operand is less than the right	x < y
==	Equal to: True if both operands are equal	x == y
!=	Not equal to – True if operands are not equal	x != y
>=	Greater than or equal to True if the left operand is greater than or equal to the right	x >= y
<=	Less than or equal to True if the left operand is less than or equal to the right	x <= y

Python Logical Operators

- Logical operators are used to combine conditional statements:

Operator	**Description**	**Example**
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Python Identity Operators

- Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x
print(x is z)                  # returns True because z is the same object as x
print(x is y)                  # returns False because x is not the same object as y, even if
they have the same content
print(x == y)                  # to demonstrate the difference between "is" and "==": this
comparison returns True because x is equal to y
```

Python Membership Operators

- Membership operators are used to test if a sequence is presented in an object:

Operator	**Description**	**Example**
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

```
x = ["apple", "banana"]
print("pineapple" not in x)      # returns True because a sequence with the value
                                 "pineapple" is not in the list
```

```
x = ["apple", "banana"]
print("banana" in x)             # returns True because a sequence with the value
                                 "banana" is in the list
```

Python Bitwise Operators

- Bitwise operators are used to compare (binary) numbers:

Operator	**Name**	**Description**	**Example**
&	AND	Sets each bit to 1 if both bits are 1	x & y
|	OR	Sets each bit to 1 if one of two bits is 1	x | y
^	XOR	Sets each bit to 1 if only one of two bits is 1	x ^ y
~	NOT	Inverts all the bits	~x
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off	x << 2
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	x >> 2

Operator Precedence

- Operator precedence describes the order in which operations are performed.

Example

- Parentheses have the highest precedence, meaning that expressions inside parentheses must be evaluated first:
    - print((6 + 3) - (6 + 3))

Example

- Multiplication * has higher precedence than addition +, and therefore multiplications are evaluated before additions:
    - print(100 + 5 * 3)
- If two operators have the same precedence, the expression is evaluated from left to right.

The precedence order is described in the table below, starting with the highest precedence at the top:

Operator	**Description**
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'td>
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Operator	**Description**
()	Parentheses
**	Exponentiation
+x -x ~x	Unary plus, unary minus, and bitwise NOT
* / // %	Multiplication, division, floor division, and modulus
+ -	Addition and subtraction
<< >>	Bitwise left and right shifts
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
== != > >= < <= is not in not in	Comparisons, identity, and membership operators
not	Logical NOT
and	AND
or	OR

Associativity

- When you have an expression with multiple operators of the same precedence, Python will evaluate them in a left-to-right order.
- However, there are a few operators that are evaluated from right to left in Python:
  - **Exponentiation Operator (**)**
    - result = 2 ** 3 ** 2 # Right-to-left exponentiation
      # result is 512 (2^(3^2))
    - In this example, 3 ** 2 is evaluated first (giving 9), and then 2 ** 9 is evaluated (giving 512).
  - **Assignment Operators with Chaining**
    - a = b = c = 10 # Right-to-left assignment

FUNCTIONS

- A function is a set of code that performs any given task, enabling the programmer to modularize a program. All variables created in function definitions are local variables; they are known only to the function in which they are dec

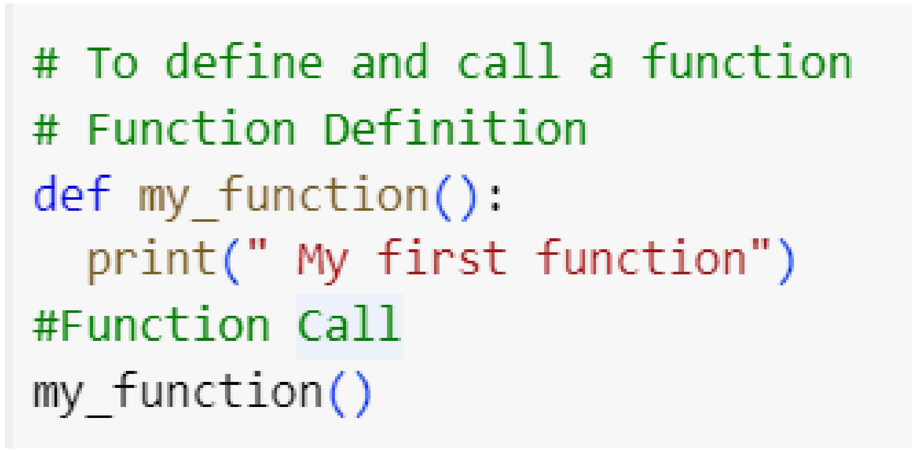
How to Create and Call a Function?

- In Python, you create a function by using the **def** keyword.

```
def name(arguments):
        Statement
        return value
```

- The **def** keyword only creates and defines a function. To call the function, use the function name, followed by parentheses.
- To call the function, use the function name, followed by parentheses.

- Use meaningful names to function which reflects its usage. Give meaningful name to parameters

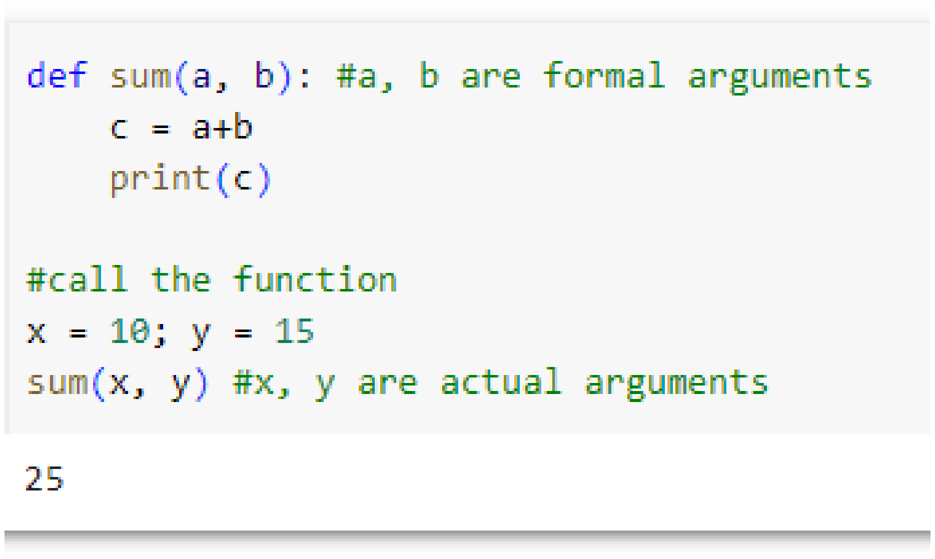


```
# To define and call a function
# Function Definition
def my_function():
  print(" My first function")
#Function Call
my_function()
```

Formal Arguments and Actual Arguments

- when a function is defined, it may have some parameters. These parameters receive values from outside of the function. 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. It is not advisable to use identical names for the arguments

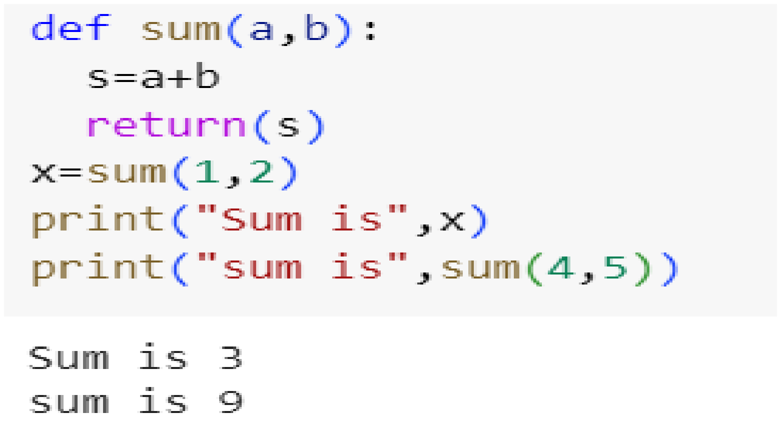


```
def sum(a, b): #a, b are formal arguments
    c = a+b
    print(c)

#call the function
x = 10; y = 15
sum(x, y) #x, y are actual arguments
```

```
25
```

The Return Statement

- To return a value from a function, we use a return statement. It “returns” the result to the caller.



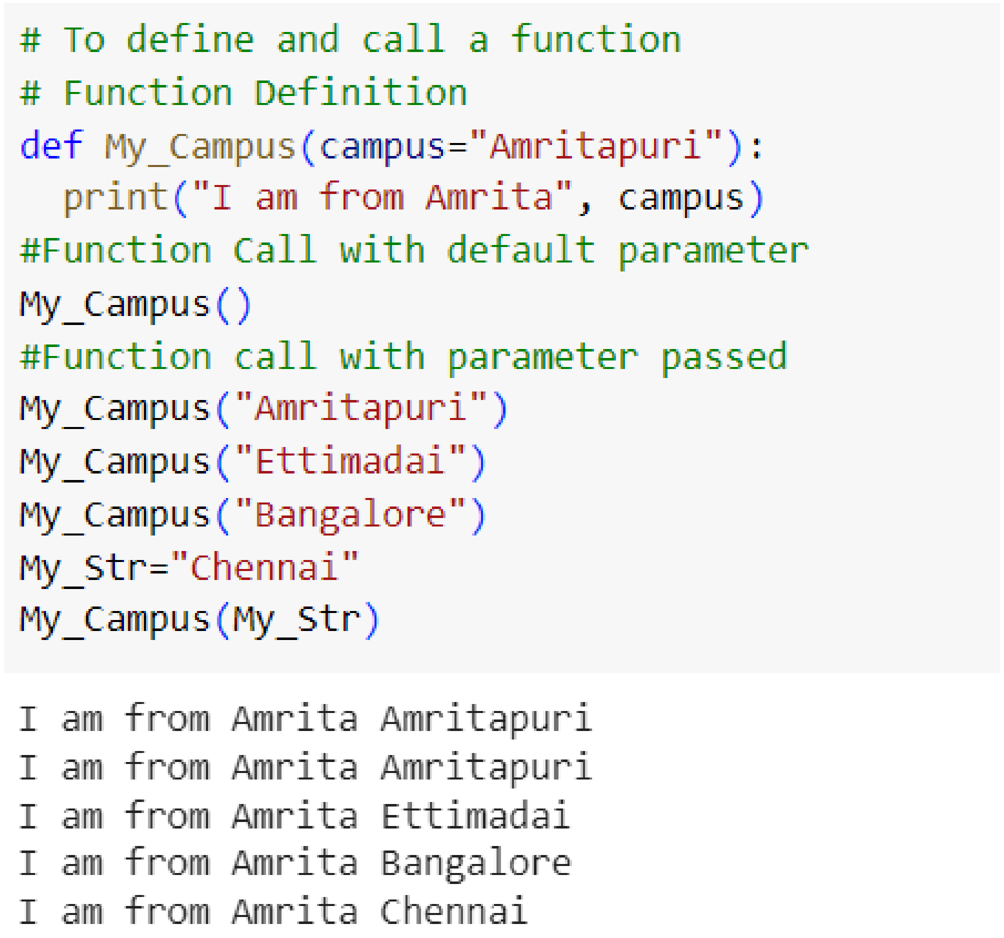
```python
def sum(a,b):
  s=a+b
  return(s)
x=sum(1,2)
print("Sum is",x)
print("sum is",sum(4,5))
```

```
Sum is 3
sum is 9
```

Types of Arguments

1. Default Arguments

Functions can be defined with default arguments. If the values for the arguments are not supplied when the function is called, the default argument is used.

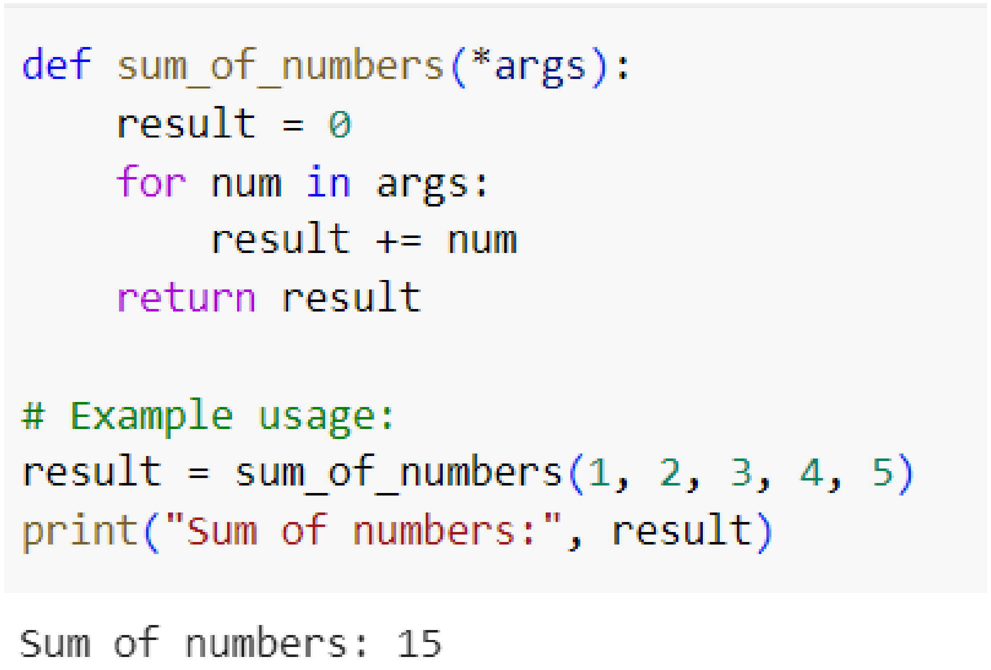


```python
# To define and call a function
# Function Definition
def My_Campus(campus="Amritapuri"):
  print("I am from Amrita", campus)
#Function Call with default parameter
My_Campus()
#Function call with parameter passed
My_Campus("Amritapuri")
My_Campus("Ettimadai")
My_Campus("Bangalore")
My_Str="Chennai"
My_Campus(My_Str)
```

```
I am from Amrita Amritapuri
I am from Amrita Amritapuri
I am from Amrita Ettimadai
I am from Amrita Bangalore
I am from Amrita Chennai
```

- ## Arbitrary Arguments

Arbitrary arguments are used when you want the function to take an unlimited number of arguments. When you add an asterisk (*), it will receive a set of arguments.

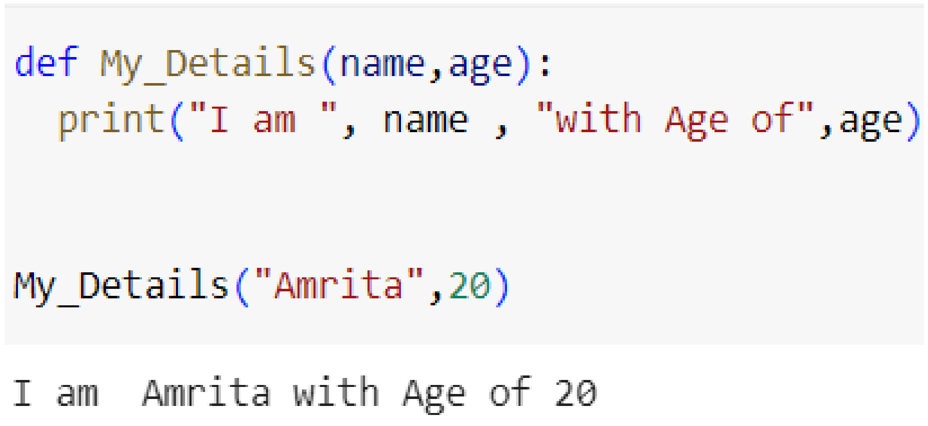


```python
def sum_of_numbers(*args):
    result = 0
    for num in args:
        result += num
    return result

# Example usage:
result = sum_of_numbers(1, 2, 3, 4, 5)
print("Sum of numbers:", result)
```

```
Sum of numbers: 15
```

Keyword Arguments

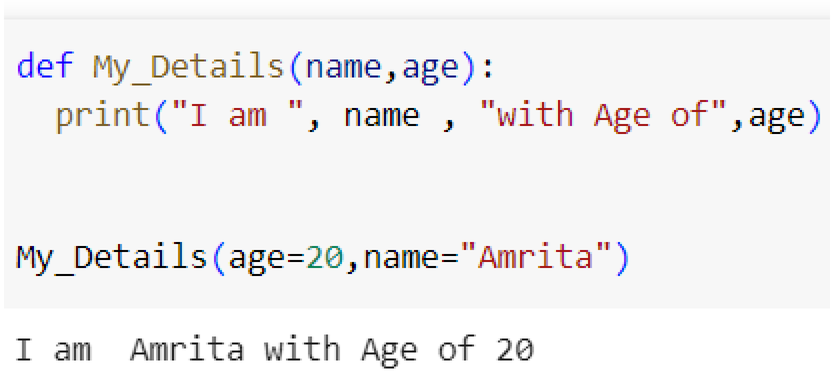
- You can pass the arguments in a non-positional manner using keyword arguments.



```
def My_Details(name,age):
  print("I am ", name , "with Age of",age)


My_Details("Amrita",20)
```

```
I am  Amrita with Age of 20
```

Two parameters function



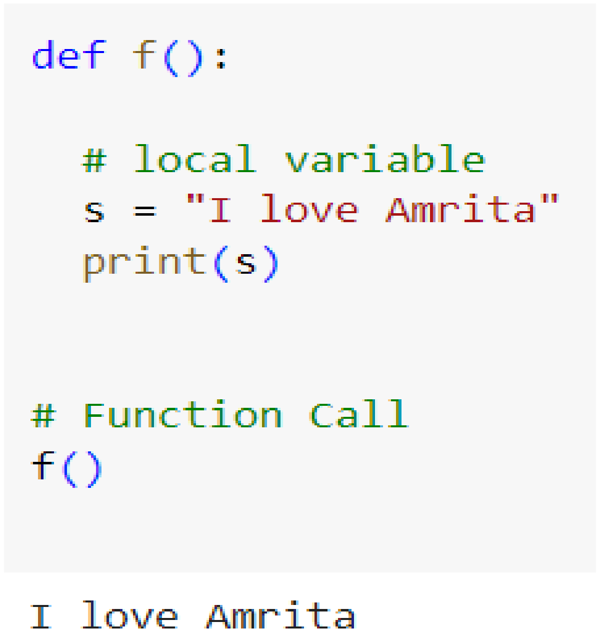
```
def My_Details(name,age):
  print("I am ", name , "with Age of",age)


My_Details(age=20,name="Amrita")
```

```
I am  Amrita with Age of 20
```

keyword arguments

Local and Global variables in Python

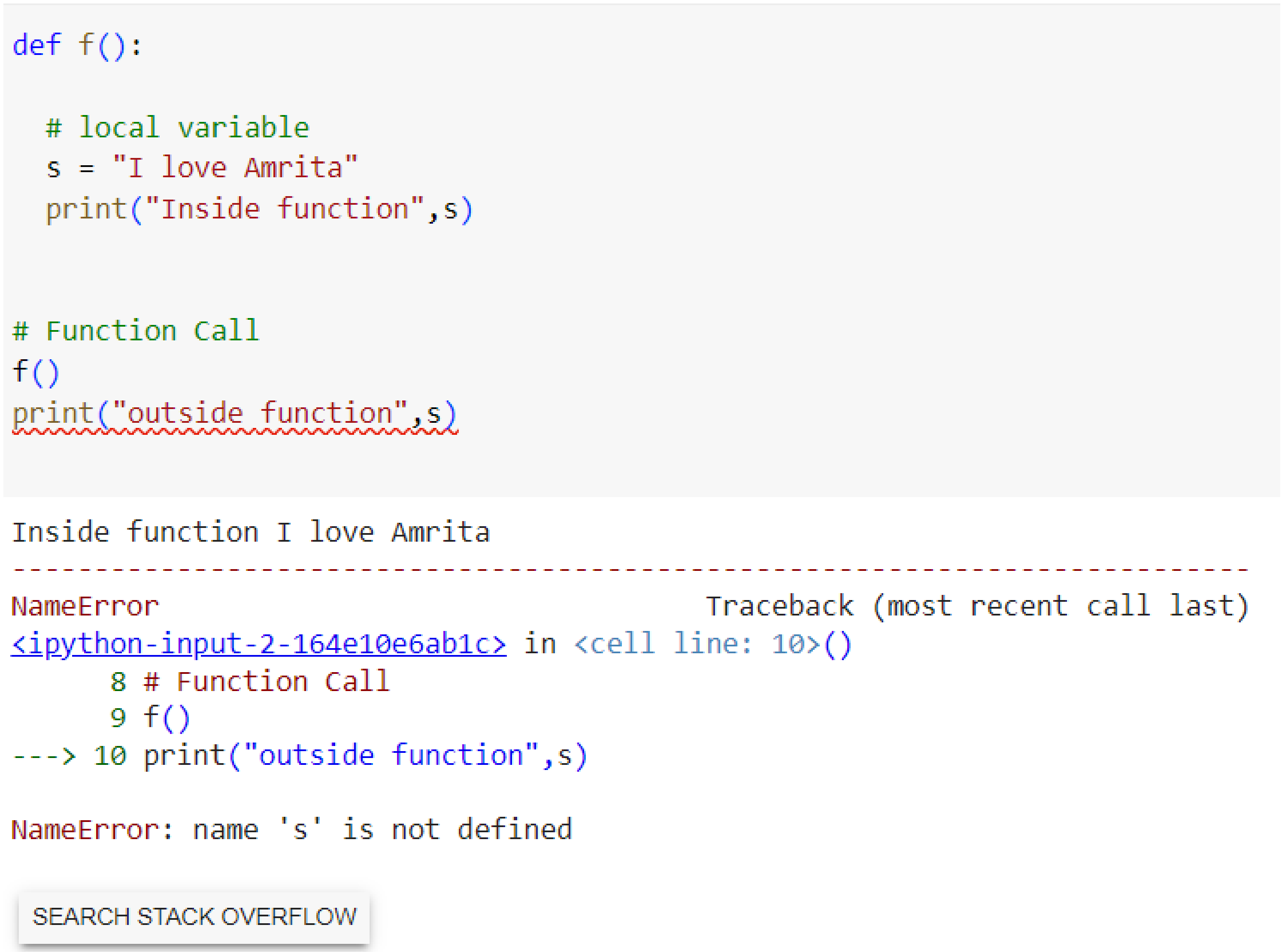
- Local variables
  - Local variables in Python are those which are initialized inside a function and belong only to that particular function. It cannot be accessed anywhere outside the function.



```python
def f():

  # local variable
  s = "I love Amrita"
  print(s)


# Function Call
f()
```

```
I love Amrita
```

- Can we use local variable outside a function?

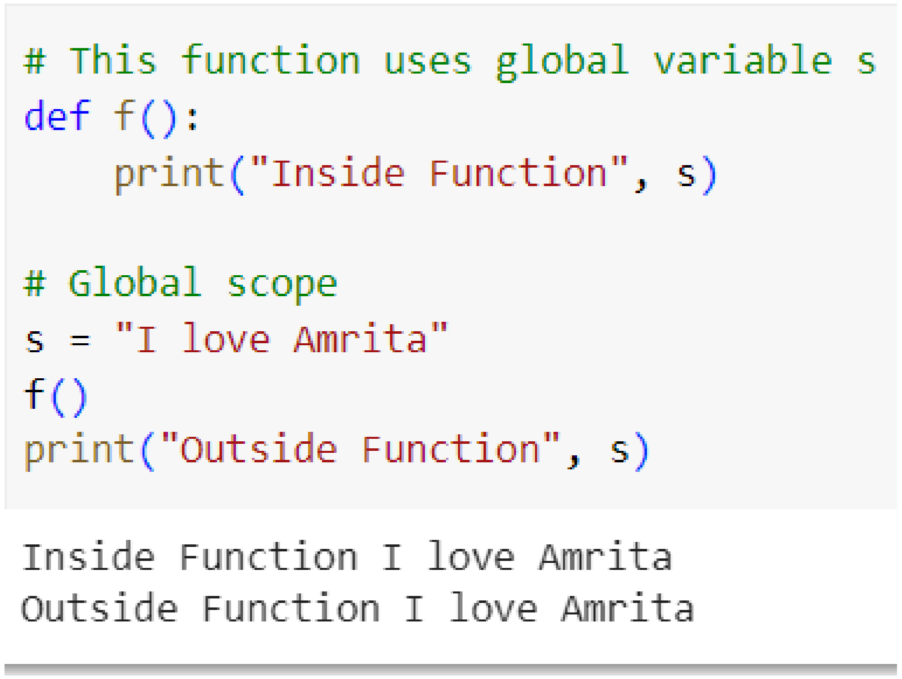


```
def f():

  # local variable
  s = "I love Amrita"
  print("Inside function",s)


# Function Call
f()
print("outside function",s)
```

```
Inside function I love Amrita
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-164e10e6ab1c> in <cell line: 10>()
      8 # Function Call
      9 f()
---> 10 print("outside function",s)

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

Global Variables

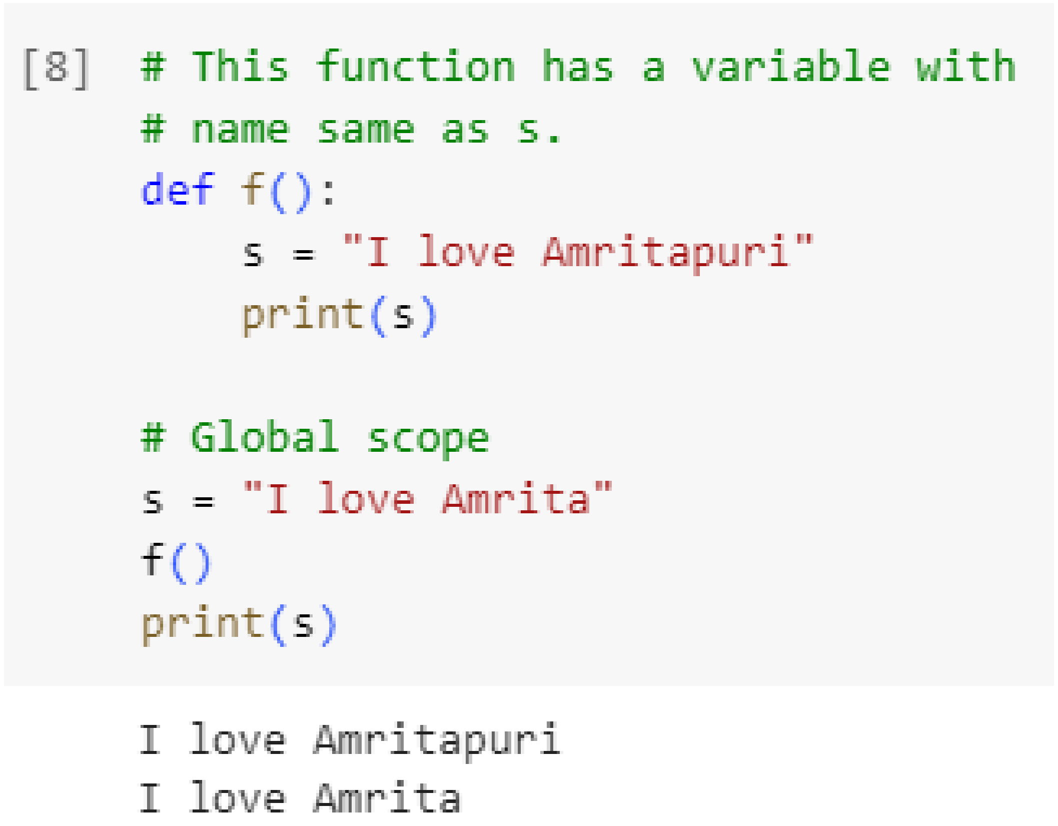
- These are those which are defined outside any function and which are accessible throughout the program, i.e., inside and outside of every function.



```
# This function uses global variable s
def f():
    print("Inside Function", s)

# Global scope
s = "I love Amrita"
f()
print("Outside Function", s)
```

```
Inside Function I love Amrita
Outside Function I love Amrita
```

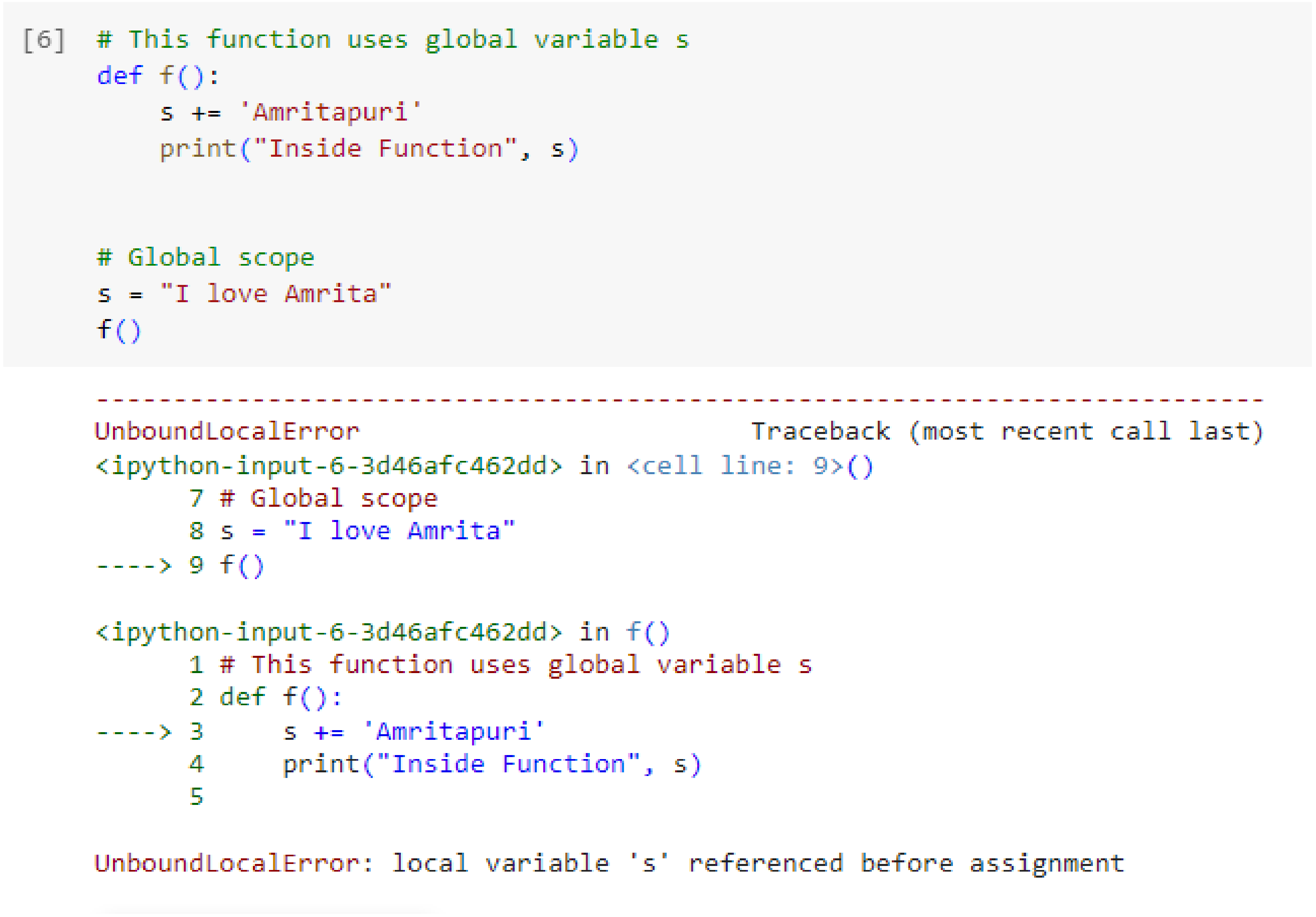
- If a variable with the same name is defined inside the scope of the function as well then it will print the value given inside the function only and not the global value.



```
[8]  # This function has a variable with
     # name same as s.
     def f():
         s = "I love Amritapuri"
         print(s)

     # Global scope
     s = "I love Amrita"
     f()
     print(s)
```

```
I love Amritapuri
I love Amrita
```

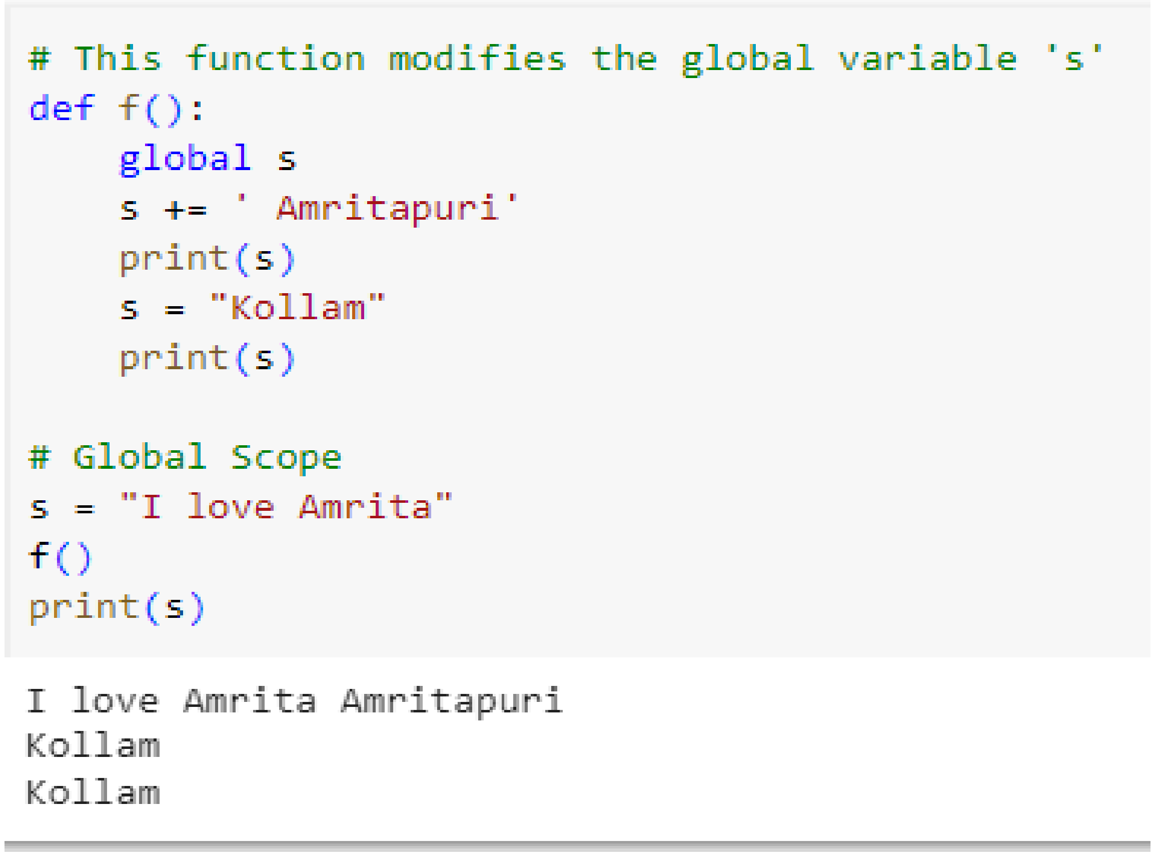
- if we try to change the value of a global variable inside the function, the below error occurs due to ambiguity.



```
[6] # This function uses global variable s
    def f():
        s += 'Amritapuri'
        print("Inside Function", s)



    # Global scope
    s = "I love Amrita"
    f()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-6-3d46afc462dd> in <cell line: 9>()
      7 # Global scope
      8 s = "I love Amrita"
----> 9 f()

<ipython-input-6-3d46afc462dd> in f()
      1 # This function uses global variable s
      2 def f():
----> 3     s += 'Amritapuri'
      4     print("Inside Function", s)
      5

UnboundLocalError: local variable 's' referenced before assignment
```

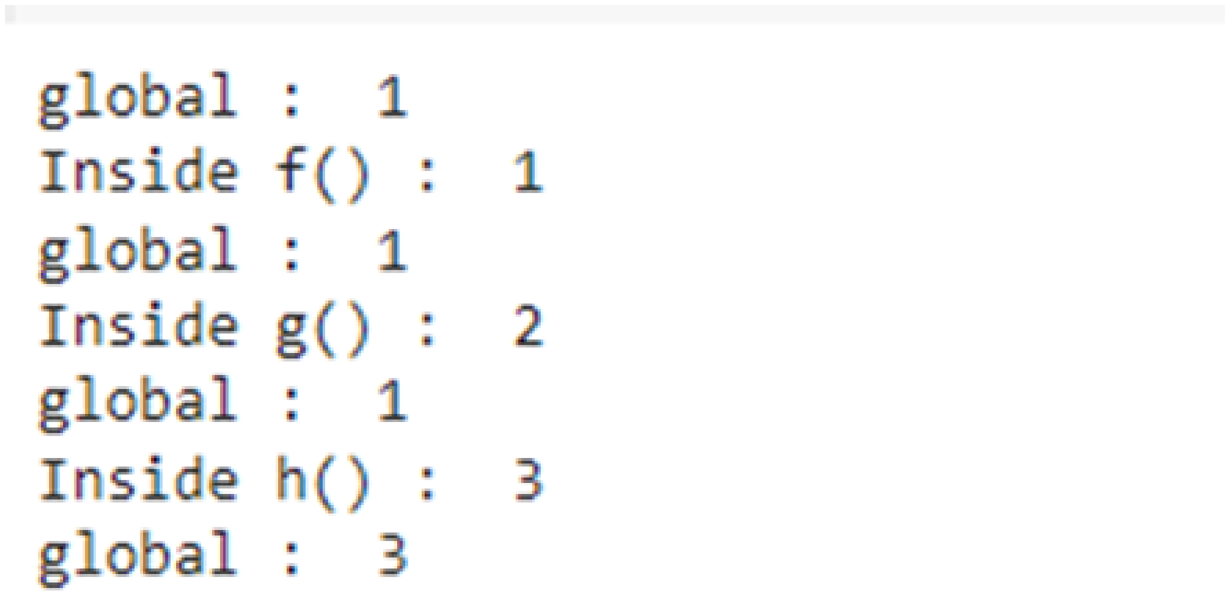
- The above code will result in an error. Specifically, it will raise a **UnboundLocalError** with the message "local variable 's' referenced before assignment."
- The reason for this error is that within the function **f()**, when you try to modify the **s** variable using **+=**, Python treats it as a local variable, and you are attempting to modify it before assigning a value to it within the local scope.
- To modify a global variable within a function, you should use the **global** keyword to indicate that you want to work with the global variable



```python
# This function modifies the global variable 's'
def f():
    global s
    s += ' Amritapuri'
    print(s)
    s = "Kollam"
    print(s)

# Global Scope
s = "I love Amrita"
f()
print(s)
```

```
I love Amrita Amritapuri
Kollam
Kollam
```

```python
a = 1

# Uses global because there is no local 'a'
def f():
    print('Inside f() : ', a)

# Variable 'a' is redefined as a local
def g():
    a = 2
    print('Inside g() : ', a)

# Uses global keyword to modify global 'a'
def h():
    global a
    a = 3
    print('Inside h() : ', a)


# Global scope
print('global : ', a)
f()
print('global : ', a)
g()
print('global : ', a)
h()
print('global : ', a)
```



```
global :  1
Inside f() :  1
global :  1
Inside g() :  2
global :  1
Inside h() :  3
global :  3
```