

Unit-3 : Notes

In Python, a function is a group of related statements that performs a specific task. Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

Furthermore, it avoids repetition and makes the code reusable.

Syntax:

```
def function_name(parameters):
```

```
    """docstring"""
```

```
    statement(s)
```

Above shown is a function definition that consists of the following components:

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

Example:

```
def greet(name):
```

```
    """This function greets to the person passed in as a parameter """
```

```
    print("Hello, " + name + ". Good morning!")
```

How to call a function in python?

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
>>> greet('Katti')
```

Hello, Katti. Good morning!

Docstrings

The first string after the function header is called the docstring and is short for documentation string. It is briefly used to explain what a function does. Although optional, documentation is a good programming practice.

In the above example, we have a docstring immediately below the function header. We generally use triple quotes so that docstring can extend up to multiple lines. This string is available to us as the `__doc__` attribute of the function.

Example:

```
>>> print(greet.__doc__)
```

This function greets to the person passed in as a parameter

The return statement

The return statement is used to exit a function and go back to the place from where it was called.

Syntax of return

```
return [expression_list]
```

This statement can contain an expression that gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the **None** object.

For example:

```
>>> print(greet("Students"))
```

Hello, Students. Good morning!

None

Here, **None** is the returned value since **greet()** directly prints the name and no return statement is used.

Example of return

```
def absolute_value(num):
```

```
    """This function returns the absolute value of the entered number"""
```

```
    if num >= 0:
```

```
        return num
```

```
    else:
```

```
        return -num
```

```
print(absolute_value(2))
```

```
print(absolute_value(-4))
```

Output

2

4

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a local scope.

The lifetime of a variable is the period throughout which the variable exists in the memory. The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Here is an example to illustrate the scope of a variable inside a function.

```
def my_func():  
    x = 10  
    print("Value inside function:",x)
```

```
x = 20  
my_func()  
print("Value outside function:",x)
```

Output

Value inside function: 10

Value outside function: 20

Here, we can see that the value of x is 20 initially. Even though the function my_func() changed the value of x to 10, it did not affect the value outside the function.

This is because the variable x inside the function is different (local to the function) from the one outside. Although they have the same names, they are two different variables with different scopes.

On the other hand, variables outside of the function are visible from inside. They have a global scope.

We can read these values from inside the function but cannot change (write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the keyword **global**.

Arguments

In the user-defined function topic, we learned about defining a function, passing parameters and calling it. Otherwise, the function call will result in an error. Here is an example.

```
def greet(name, msg):
```

```
    """This function greets to the person with the provided message"""
```

```
    print("Hello", name + ', ' + msg)
```

```
greet("Students", "Good morning!")
```

Output

Hello Students, Good morning!

Here, the function greet() has two parameters.

Since we have called this function with two arguments, it runs smoothly and we do not get any error.

If we call it with a different number of arguments, the interpreter will show an error message. Below is a call to this function with one and no arguments along with their respective error messages.

```
>>> greet("Alia") # only one argument
```

```
TypeError: greet() missing 1 required positional argument: 'msg'
```

```
>>> greet() # no arguments
```

```
TypeError: greet() missing 2 required positional arguments: 'name' and 'msg'
```

Variable Function Arguments

Up until now, functions had a fixed number of arguments. In Python, there are other ways to define a function that can take variable number of arguments.

Three different forms of this type are described below.

1. Python Default Arguments

Function arguments can have default values in Python.

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

```
def greet(name, msg="Good morning!"):
    """ If the message is not provided, it defaults to 'Good morning!' """
    print("Hello", name + ', ' + msg)
```

```
greet("Alia")
greet("Shahrukh", "How do you do?")
```

Output

Hello Alia, Good morning!

Hello Shahrukh, How do you do?

In this function, the parameter **name** does not have a default value and is required (mandatory) during a call.

On the other hand, the parameter **msg** has a default value of "Good morning!". So, it is optional during a call. If a value is provided, it will overwrite the default value.

Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

This means to say, non-default arguments cannot follow default arguments. For example, if we had defined the function header above as:

```
def greet(msg = "Good morning!", name):
```

We would get an error as:

SyntaxError: non-default argument follows default argument

2. Python Keyword Arguments

When we call a function with some values, these values get assigned to the arguments according to their position.

For example, in the above function `greet()`, when we called it as `greet("Alia", "How do you do?")`, the value "Alia" gets assigned to the argument **name** and similarly "How do you do?" to **msg**.

Python allows functions to be called using keyword arguments. When we call functions in this way, the order (position) of the arguments can be changed. Following calls to the above function are all valid and produce the same result.

2 keyword arguments

```
greet(name = "Sameer", msg = "Welcome to Python")
```

2 keyword arguments (out of order)

```
greet(msg = "Welcome to Python", name = "Sameer")
```

#1 positional, 1 keyword argument

```
greet("Bruce", msg = "How do you do?")
```

As we can see, we can mix positional arguments with keyword arguments during a function call. But we must keep in mind that keyword arguments must follow positional arguments.

Having a positional argument after keyword arguments will result in errors. For example, the function call as follows:

```
greet(name="Pooja","Welcome to PESU")
```

Will result in an error:

SyntaxError: non-keyword arg after keyword arg

3. Python Arbitrary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function. Python allows us to handle this kind of situation through function calls with an arbitrary number of arguments.

In the function definition, we use an **asterisk (*)** before the parameter name to denote this kind of argument. Here is an example.

```
def greet(*names):  
    """This function greets all the person in the names tuple."""  
    # names is a tuple with arguments  
    for name in names:  
        print("Hello", name)  
  
greet("Ramya", "Mohan", "Siddharth", "Afroz")
```

Output

Hello Ramya

Hello Mohan

Hello Siddharth

Hello Afroz

Here, we have called the function with multiple arguments. These arguments get wrapped up into a tuple before being passed into the function. Inside the function, we use a for loop to retrieve all the arguments back.

Assignment:

Check the correctness of following python code:

```
def funny(a, b="Good morning", *c):  
    statements
```

Recursion

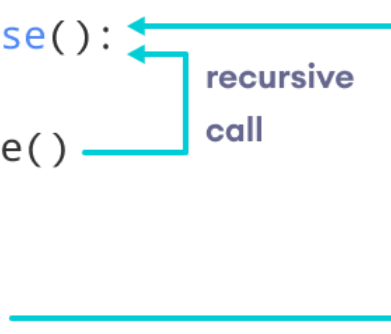
Recursion is the process of defining something in terms of itself.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

Python Recursive Function

In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

```
def recurse():  
    ...  
    recurse()  
    ...  
recurse()
```



Following is an example of a recursive function to find the factorial of an integer.

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is $1*2*3*4*5*6 = 720$.

Example of a recursive function

```
def factorial(x):  
    """This is a recursive function to find the factorial of an integer"""  
    if x == 1:  
        return 1  
    else:  
        return (x * factorial(x-1))  
num = 3  
print("The factorial of", num, "is", factorial(num))
```

Output

The factorial of 3 is 6

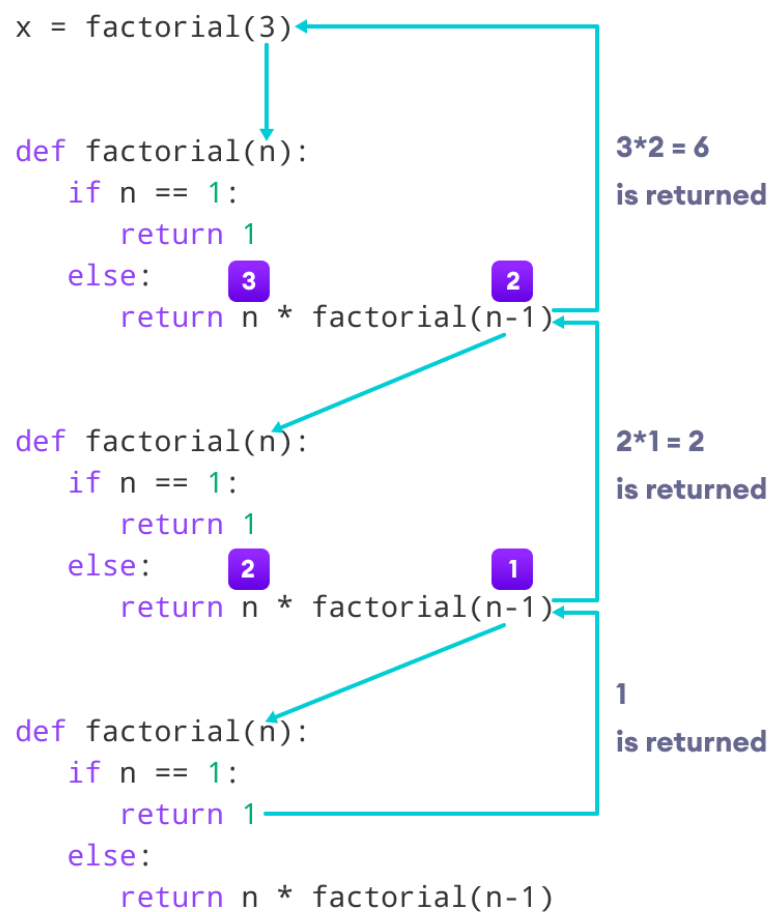
In the above example, factorial() is a recursive function as it calls itself.

When we call this function with a positive integer, it will recursively call itself by decreasing the number.

Each function multiplies the number with the factorial of the number below it until it is equal to one. This recursive call can be explained in the following steps.

```
factorial(3)      # 1st call with 3  
3 * factorial(2)  # 2nd call with 2  
3 * 2 * factorial(1) # 3rd call with 1  
3 * 2 * 1        # return from 3rd call as number=1  
3 * 2            # return from 2nd call  
6                # return from 1st call
```

Let's look at an image that shows a step-by-step process of what is going on:



Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.

By default, the maximum depth of recursion is 1000. If the limit is crossed, it results in RecursionError. Let's look at one such condition.

Try this:

```
def recursor():  
    recursor()  
recursor()
```

Advantages of Recursion

- Recursive functions make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.
- Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

- Sometimes the logic behind recursion is hard to follow through.
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
- Recursive functions are hard to debug.

Global Variables

In Python, a variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.

Example 1: Create a Global Variable

```
x = "global"

def fun():
    print("x inside:", x)

fun()

print("x outside:", x)
```

Output

```
x inside: global
x outside: global
```

What if you want to change the value of x inside a function?

```
x = "global"
```

```
def fun():
```

```
    x = x * 2
```

```
    print(x)
```

```
fun()
```

Output

UnboundLocalError: local variable 'x' referenced before assignment

The output shows an error because Python treats **x** as a local variable and **x** is also not defined inside **fun()**.

To make this work, we use the **global** keyword.

Local Variables

A variable declared inside the function's body or in the local scope is known as a local variable.

Example 2: Accessing local variable outside the scope

```
def fun():
```

```
    y = "local"
```

```
fun()
```

```
print(y)
```

Output

NameError: name 'y' is not defined

The output shows an error because we are trying to access a local variable **y** in a global scope whereas the local variable only works inside **fun()** or local scope.

Example 3: Create a Local Variable

Normally, we declare a variable inside the function to create a local variable.

```
def fun():  
    y = "local"  
    print(y)  
  
fun()
```

Output

local

Let's take a look at the earlier problem where **x** was a global variable and we wanted to modify **x** inside `fun()`.

Global and local variables

Here, we will show how to use global variables and local variables in the same code.

Example 4: Using Global and Local variables in the same code

```
x = "global "  
  
def fun():  
    global x  
    y = "local"  
    x = x * 2  
    print(x)  
    print(y)  
  
fun()
```

Output

global global
local

In the above code, we declare **x** as a global and **y** as a local variable in the **fun()**. Then, we use multiplication operator ***** to modify the global variable **x** and we print both **x** and **y**.

After calling the **fun()**, the value of **x** becomes global, global because we used the **x * 2** to print two times global. After that, we print the value of local variable **y** i.e local.

Example 5: Global variable and Local variable with same name

```
x = 5
def fun():
    x = 10
    print("local x:", x)
fun()
print("global x:", x)
```

Output

local x: 10

global x: 5

In the above code, we used the same name **x** for both global variable and local variable. We get a different result when we print the same variable because the variable is declared in both scopes, i.e. the local scope inside **fun()** and global scope outside **fun()**.

When we print the variable inside **fun()** it outputs local **x: 10**. This is called the local scope of the variable.

Similarly, when we print the variable outside the **fun()**, it outputs global **x: 5**. This is called the global scope of the variable.

Nonlocal Variables

Nonlocal variables are used in nested functions whose local scope is not defined. This means that the variable can be neither in the local nor the global scope.

We use **nonlocal** keywords to create nonlocal variables.

Example : Create a nonlocal variable

```
def outer():  
    x = "local"  
  
    def inner():  
        nonlocal x  
        x = "nonlocal"  
        print("inner:", x)  
    inner()  
    print("outer:", x)  
outer()
```

Output

inner: nonlocal

outer: nonlocal

In the above code, there is a nested **inner()** function. We use **nonlocal** keywords to create a nonlocal variable. The **inner()** function is defined in the scope of another function **outer()**.

Note : If we change the value of a nonlocal variable, the changes appear in the local variable.

Before getting into what a closure is, we have to first understand what a nested function and nonlocal variable is.

A function defined inside another function is called a nested function. Nested functions can access variables of the enclosing scope.

In Python, these non-local variables are read-only by default and we must declare them explicitly as non-local (using **nonlocal** keyword) in order to modify them.

Following is an example of a nested function accessing a non-local variable.

```
def print_msg(msg):  
    # This is the outer enclosing function  
  
    def printer():  
        # This is the nested function  
        print(msg)  
    printer()  
  
# We execute the function  
# Output: Hello  
print_msg("Hello")
```

Output

Hello

We can see that the nested `printer()` function was able to access the non-local **msg** variable of the enclosing function.

Defining a Closure Function

In the example above, what would happen if the last line of the function **print_msg()** returned the **printer()** function instead of calling it? This means the function was defined as follows:

```
def print_msg(msg):  
    # This is the outer enclosing function  
  
    def printer():  
        # This is the nested function  
  
        print(msg)  
  
    return printer # returns the nested function
```

Now let's try calling this function.

Output: Hello

```
another = print_msg("Hello")  
another()
```

Output

Hello

That's unusual.

The **print_msg()** function was called with the string "Hello" and the returned function was bound to the name **another**. On calling **another()**, the message was still remembered although we had already finished executing the **print_msg()** function.

This technique by which some data ("Hello" in this case) gets attached to the code is called closure in Python.

This value in the enclosing scope is remembered even when the variable goes out of scope or the function itself is removed from the current namespace.

Try running the following in the Python shell to see the output.

```
>>> del print_msg
```

```
>>> another()
```

Hello

```
>>> print_msg("Hello") # Error
```

Here, the returned function still works even when the original function was deleted.

When do we have closures?

As seen from the above example, we have a closure in Python when a nested function references a value in its enclosing scope.

The criteria that must be met to create closure in Python are summarized in the following points.

- We must have a nested function (function inside a function).
- The nested function must refer to a value defined in the enclosing function.
- The enclosing function must return the nested function.

When to use closures? So what are closures good for?

- Closures can avoid the use of global values and provides some form of data hiding. It can also provide an object oriented solution to the problem.
- When there are few methods (one method in most cases) to be implemented in a class, closures can provide an alternate and more elegant solution. But when the number of attributes and methods get larger, it's better to implement a class.

Here is a simple example where a closure might be more preferable than defining a class and making objects. But the preference is all yours.

```
def make_multiplier_of(n):  
    def multiplier(x):  
        return x * n  
    return multiplier  
  
# Multiplier of 3  
times3 = make_multiplier_of(3)  
  
# Multiplier of 5  
times5 = make_multiplier_of(5)  
  
print(times3(9))
```

```
print(times5(3))  
print(times5(times3(2)))
```

Output

27

15

30

Decorators in Python

Python has an interesting feature called decorators to add functionality to an existing code.

This is also called metaprogramming because a part of the program tries to modify another part of the program at compile time.

Note:

We must be comfortable with the fact that everything in Python (Yes! Even classes), are objects. Names that we define are simply identifiers bound to these objects. Functions are no exceptions, they are objects too (with attributes). Various different names can be bound to the same function object.

Example

```
def first(msg):  
    print(msg)  
first("Hello")  
second = first  
second("Hello")
```

Output

Hello

Hello

When you run the code, both functions **first** and **second** give the same output. Here, the names **first** and **second** refer to the same function object.

Functions can be passed as arguments to another function.

If you have used functions like map, filter and reduce in Python, then you already know about this.

Such functions that take other functions as arguments are also called higher order functions. Here is an example of such a function.

```
def inc(x):
```

```
    return x + 1
```

```
def dec(x):
```

```
    return x - 1
```

```
def operate(func, x):
```

```
    result = func(x)
```

```
    return result
```

We invoke the function as follows.

```
>>> operate(inc,3)
```

```
4
```

```
>>> operate(dec,3)
```

```
2
```

Basically, a decorator takes in a function, adds some functionality and returns it.

```
def make_pretty(func):  
    def inner():  
        print("I got decorated")  
        func()  
    return inner
```

```
def ordinary():  
    print("I am ordinary")
```

When you run the following codes in shell,

```
>>> ordinary()  
I am ordinary  
>>> # let's decorate this ordinary function  
>>> pretty = make_pretty(ordinary)  
>>> pretty()  
I got decorated  
I am ordinary
```

In the example shown above, **make_pretty()** is a decorator. In the assignment step:

```
pretty = make_pretty(ordinary)
```

The function **ordinary()** got decorated and the returned function was given the name **pretty**.

We can see that the decorator function added some new functionality to the original function. This is similar to packing a gift. The decorator acts as a wrapper. The nature of the object that got decorated (actual gift inside) does not alter. But now, it looks pretty (since it got decorated).

Generally, we decorate a function and **reassign** it as,

ordinary = make_pretty(ordinary).

This is a common construct and for this reason, Python has a syntax to simplify this.

We can use the **@ symbol** along with the name of the decorator function and place it above the definition of the function to be decorated. For example,

```
@make_pretty
def ordinary():
    print("I am ordinary")
```

is equivalent to

```
def ordinary():
    print("I am ordinary")
ordinary = make_pretty(ordinary)
```

This is just a syntactic sugar to implement decorators.

Decorating Functions with Parameters

The above decorator was simple and it only worked with functions that did not have any parameters. What if we had functions that took in parameters like:

```
def divide(a, b):  
    return a/b
```

This function has two parameters, a and b. We know it will give an error if we pass in b as 0.

```
>>> divide(2,5)
```

```
0.4
```

```
>>> divide(2,0)
```

```
Traceback (most recent call last):
```

```
...
```

```
ZeroDivisionError: division by zero
```

Now let's make a decorator to check for this case that will cause the error.

```
def smart_divide(func):  
    def inner(a, b):  
        print("I am going to divide", a, "and", b)  
        if b == 0:  
            print("Whoops! cannot divide")  
            return  
        return func(a, b)  
    return inner  
  
@smart_divide  
def divide(a, b):  
    print(a/b)
```


This new implementation will return None if the error condition arises.

```
>>> divide(2,5)
```

I am going to divide 2 and 5

0.4

```
>>> divide(2,0)
```

I am going to divide 2 and 0

Whoops! cannot divide

In this manner, we can decorate functions that take parameters.