FUNCTIONS IN PYTHON

This lesson will introduce you to functions in Python. You'll cover the following topics:

- Functions in Python
- Types of functions in python
- Built In Functions
- User-Defined Functions :
    - How To Define a Function
    - How To Call a Function
    - Function Arguments in Python
    - Pass by value - Pass by Reference
    - Variable Scope
    - The **return** statement
    - How To Add Docstrings to a Python Function
    - Python suppporting first class function
- Recurssion Function in Python
- Anonymous Functions in Python - Lamda
- Functional Programming
- Functions vs Methods
- Generators
- Decorators
- Using `main( )` as a Function

What is a Function ? :

A function is a block of reusable code that is used to perform a specific action only when called .

The advantages of using functions are :

- ✓ *Reducing duplication of code*
- ✓ *Decomposing complex problems into simpler pieces*
- ✓ *Improving clarity of the code*
- ✓ *Reuse of code*
- ✓ *Information hiding*

Python Function types :

The following are the different types of Python Functions:

a. Built-in Functions

b. User-defined Functions

c. Recursion Functions

d. Lambda Functions

Built-in Functions :

Built-in functions are the ones that are offered by the language. These functions are already present in the libraries of the language. The developers only need to call the function to get the work done .

Syntax: name_of_the_function(parameters)

To use any built-in function one must follow the format.

Built in functions in Python perform their respective operations when executed.

Python interpreter has various pre-defined functions that are readily available to use. There are 68 built in functions in python 3.4 (changes with respect to updated version)

TABLE OF IN-BUILT FUNCTIONS IN PYTHON				
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

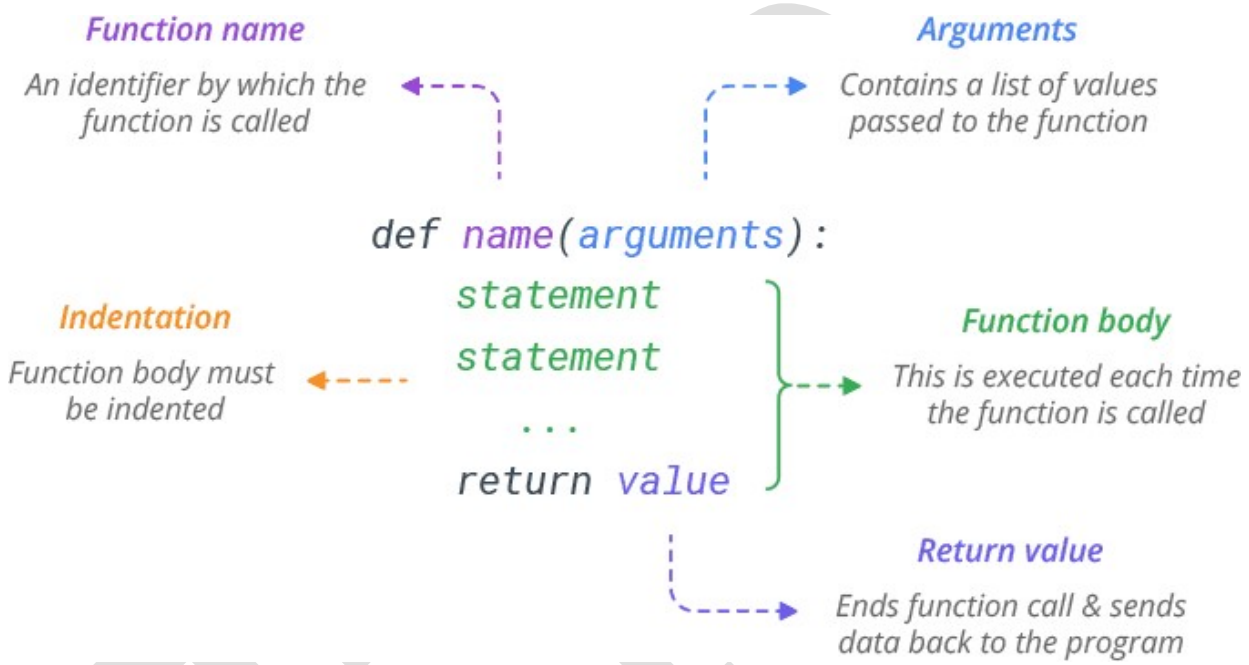
To use these Python built in functions, all we have to do is to call them and pass the relevant argument as mentioned in the description of each built in function.

User Defined Functions (UDF's) :

The functions that we define, i.e., the functions defined by the userare called as the user defined function

Defining a Function

You can define functions to provide the required functionality. Here is the syntax to define a function in Python.



Above shown is a function syntax that consists of the following components.

- ✓ *Function blocks begin with the keyword def followed by the function name and parentheses ( )*
- ✓ *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.A return statement with no arguments is the same as return None.*

Creating a Function

To define a Python function, use def keyword. Here's the simplest possible function that prints 'Hello, World!' on the screen.

```
def hello():

    print('Hello, World!')
```

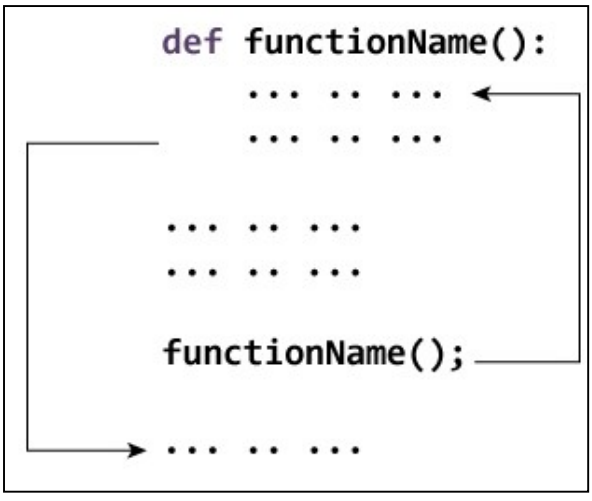
Call a Function

The def statement only creates a function but does not call it. Functions are not executed until they are called.To call a function, we specify the function name with the parentheses ( ).

```
#hello( ) - Throws name error

def hello():

    print('Hello, World!')

hello( ) # Prints Hello, World!
```

Definitions of functions must precede their usage. Otherwise the interpreter will complain with a ***Name Error.****In the above example, we have two call of functions. One line is commented. A function call cannot be ahead of its definition.Uncommenting the line we get a Name Error.*

How Function works in Python?



The whole concept of functions revolves around two things – first, how you define / write your function and second, how and where you call that function by supplying necessary parameters to the function.

Passing Arguments

*You can send information to a function by passing values, known as **arguments**. Arguments are declared after the function name in parentheses.*

Aruguments vs Parameters
***Arguments** are values that are passed into function(or method) when the calling function* ***Parameters** are variables(identifiers) specified in the (header of) function definition*

When you call a function with arguments, the values of those arguments are copied to their corresponding parameters inside the function.

```
# Pass single argument to a function

def hello(name):

    print('Hello,', name)

hello('Bob') # Prints Hello, Bob

hello('Sam') # Prints Hello, Sam
```

You can send as many arguments as you like, separated by commas

```
# Pass two arguments

def func(name, job):

    print(name, 'is a', job)

func('Bob', 'developer') # Prints Bob is a developer
```

Types of Arguments

Python handles function arguments in a very flexible manner, compared to other languages. It supports multiple types of arguments in the function definition. Here's the list:

- ✓ *Positional Arguments*
- ✓ *Keyword Arguments*
- ✓ *Default Arguments*
- ✓ *Variable Length : Positional Arguments (*args) , Keyword Arguments (**kwargs)*

Positional Arguments

The most common are positional arguments, whose values are copied to their corresponding parameters in order (also called required arguments).

In the function definition, you specify a comma-separated list of parameters inside the parentheses , When the function is called, you specify a corresponding list of arguments

```
def func(name, job):

    print(name, 'is a', job)

func('Bob', 'developer') # Prints Bob is a developer
```

The only downside of positional arguments is that you need to pass arguments in the order in which they are defined.

```
def func(name, job):

    print(name, 'is a', job)

func('developer', 'Bob') # Prints developer is a Bob
```

Keyword Arguments

To avoid positional argument confusion, you can pass arguments using the names of their corresponding parameters.In this case, the order of the arguments no longer matters because arguments are matched by name, not by position.

```
# Keyword arguments can be put in any order

def func(name, job):

    print(name, 'is a', job)

func(name='Bob', job='developer') # Prints Bob is a developer

func(job='developer', name='Bob') # Prints Bob is a developer
```

It is possible to combine positional and keyword arguments in a single call. If you do so, specify the positional arguments before keyword arguments.

Default Arguments

You can specify default values for arguments when defining a function. The default value is used if the function is called without a corresponding argument.

In short, defaults allow you to make selected arguments optional.

```
# Set default value 'developer' to a 'job' parameter

def func(name, job='developer'):

    print(name, 'is a', job)

func('Bob', 'manager')# Prints Bob is a manager

func('Bob')# Prints Bob is a developer
```

Variable Length Arguments (*args and **kwargs)

Variable length arguments are useful when you want to create functions that take unlimited number of arguments. Unlimited in the sense that you do not know beforehand how many arguments can be passed to your function by the user.

This feature is often referred to as Arbitrary Arguments or var-args

***args**

*When you prefix a parameter with an asterisk * , it collects all the unmatched positional arguments into a tuple. Because it is a normal tuple object, you can perform any operation that a tuple supports, like indexing, iteration etc.*

Following function prints all the arguments passed to the function as a tuple.

```
def print_arguments(*args):

    print(args)

print_arguments(1, 54, 60, 8, 98, 12) # Prints (1, 54, 60, 8, 98, 12)
```

****kwargs**

*The ** syntax is similar, but it only works for keyword arguments. It collects them into a new dictionary, where the argument names are the keys, and their values are the corresponding dictionary values.*

```
def print_arguments(**kwargs):

        print(kwargs)

print_arguments(name='Bob', age=25, job='dev')

# Prints {'name': 'Bob', 'age': 25, 'job': 'dev'}
```

Number of Arguments :

Any function in python should be called with the same number of arguments as in its definition excluding the default arguments. This means that python will give an error message every time a function is called with the wrong number of arguments. The default arguments if not provided will be assumed with the default value.

```
def func(name, job):

    print(name, 'is a', job)

func('Bob') # TypeError: func() takes exactly 2 arguments(1 given)

#When you run the above code in the terminal, you get a Type Error
```

We need to pass another argument (a string) to the function as job as shown below:

```
func('Bob', 'developer') # Prints Bob is a developer
```

**The argument of a function can be any type of expression, including arithmetic operators:*

```
import math

x = math.sin(360*2*math.pi)

print(x) # Prints -3.133115067780141e-14
```

And even function calls:

```
import math

x = math.exp(math.log(3.14))

print(x)# Prints 3.1399999999999997
```

Is Python Call by Value or Call by Reference ? :

In C, Java and some other language we pass value to a function either by value or by reference widely knows as "Pass/Call By Value" and "Pass/Call By Reference"

Call/Pass By Value:.

In pass-by-value, the function receives a copy of the argument objects passed to it by the caller, stored in a new location in memory.

You pass values of parameter to the function,if any kind of change done to those parameters inside the function ,those changes not reflected back in your actual parameters.

Call By Reference :

In pass-by-reference, the function receives reference to the argument objects passed to it by the caller, both pointing to the same memory location.

you pass reference of parameters to your function.if any changes made to those parameters inside function those changes are get reflected back to your actual parameters.

In Python Neither of these two concepts are applicable, rather the values are sent to functions by means of object reference.

Pass-by-object-reference :

In Python, (almost) everything is an object. What we commonly refer to as "variables" in Python are more properly called names. Likewise, "assignment" is really the binding of a name to an object. Each binding has a scope that defines its visibility, usually the block in which the name originates.

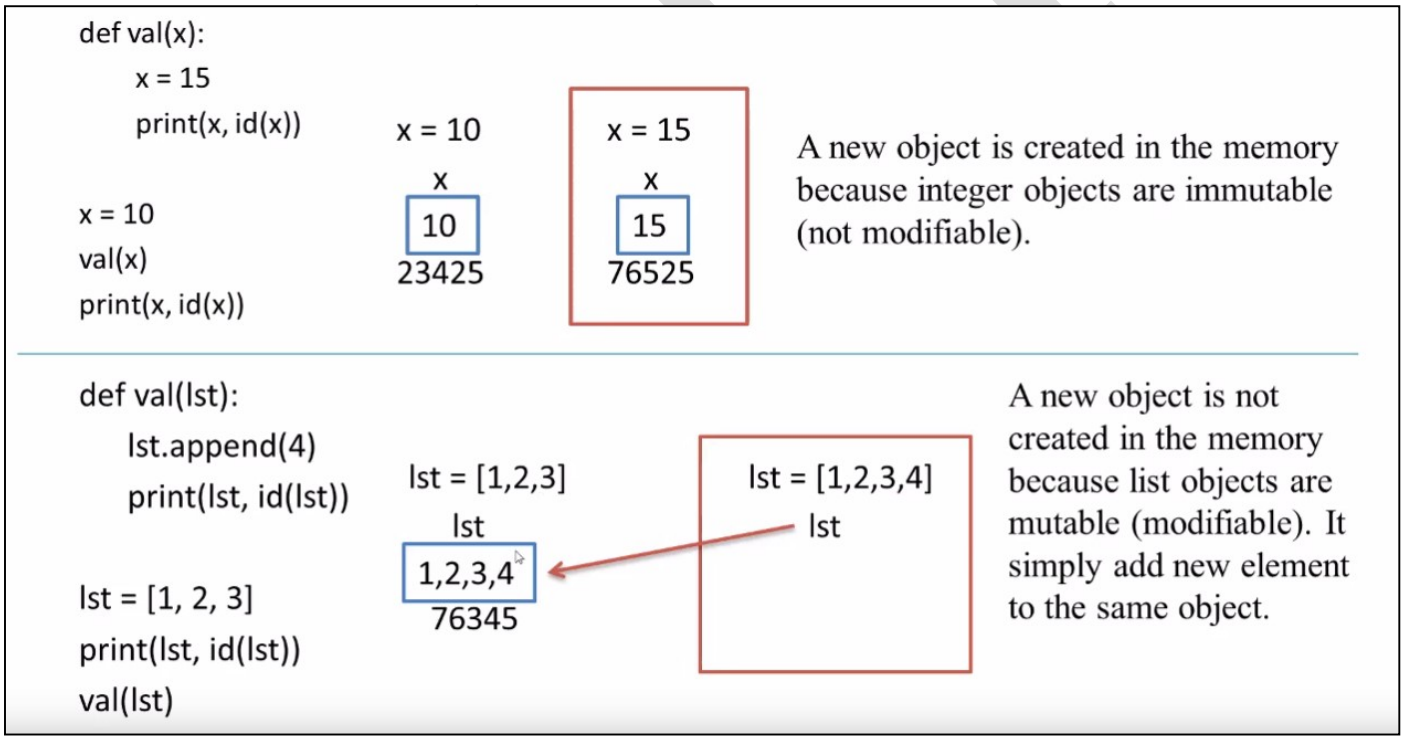
In Python,Values are passed to function by object reference.

If object is immutable(not modifiable) than the modified value is not available outside the function.

Immutable objects:
int, float, complex, string, tuple, frozen set [note: immutable version of set]

If object is mutable (modifiable) than modified value is available outside the function.

Mutable objects:
list, dict, set,



*Function is said to have a **side effect** if it changes anything outside of its function definition like changing arguments passed to the function or changing a global variable.*

PYTHON VARIABLES SCOPE

Not all variables are accessible from all parts of our program. The part of the program where the variable is accessible is called its "scope" and is determined by where the variable is declared.

Python has three different variable scopes :

Local scope - Local Variable
Global scope - Global Variable
Enclosing scope - Non Local Variable

Local Scope

A variable declared within a function has a LOCAL SCOPE and variable is called as LOCAL VARIABLE It is accessible from the point at which it is declared until the end of the function, and exists for as long as the function is executing.

```
def myfunc():
    x = 42          # local scope x
    print(x)
myfunc()            # prints 42
```

Local variables are removed from memory when the function call exits. Therefore, trying to get the value of the local variable outside the function causes an error.

```
def myfunc():
    x = 42          # local scope x
myfunc()print(x)            # Triggers NameError: x does not exist
```

Global Scope

A variable declared outside all functions has a GLOBAL SCOPE and variable is called as GLOBAL VARIABLE .It is accessible throughout the file, and also inside any file which imports that file.

```
x = 42                  # global scope x

def myfunc():

    print(x)       # x is 42 inside def

myfunc( )

print(x)               # x is 42 outside def
```

*Global variables are often used for **flags** (boolean variables that indicate whether a condition is true). For example, some programs use a flag named verbose to report more information about an operation.*

```
verbose = True

def op1():

    if verbose:

        print('Running operation 1')
```

Modifying Globals Inside a Function - Global Keyword

Although you can access global variables inside or outside of a function, you cannot modify it inside a function.

Here's an example that tries to reassign a global variable inside a function.

```
x = 42                  # global scope x

def myfunc():

    x = 0

    print(x)        # local x is 0

myfunc( )

print(x)              # global x is still 42
```

Here, the value of global variable x didn't change. Because Python created a new local variable named x; which disappears when the function ends, and has no effect on the global variable.

To access the global variable rather than the local one, you need to explicitly declare x global, using the global keyword.

```
x = 42                  # global scope x

def myfunc():

    global x        # declare x global

    x = 0

    print(x)        # global x is now 0

myfunc( )

print(x)              # x is 0
```

The x inside the function now refers to the x outside the function, so changing x inside the function changes the x outside it.

Here's another example that tries to update a global variable inside a function.

```
x = 42                  # global scope x

def myfunc():

    x = x + 1       # raises UnboundLocalError

    print(x)

myfunc()
```

Here, Python assumes that x is a local variable, which means that you are reading it before defining it.

The solution, again, is to declare x global.

```
x = 42                  # global scope x

def myfunc():

    global x

    x = x + 1       # global x is now 43

    print(x)

myfunc()

print(x)            # x is 43
```

There's another way to update a global variable from a no-global scope – use globals() function.

Python globals()

The globals() method returns the dictionary of the current global symbol table.

A symbol table is a data structure maintained by a compiler which contains all necessary information about the program.

These include variable names, methods, classes, etc.

There are mainly two kinds of symbol table.

1. *Local symbol table*
2. *Global symbol table*

***Local** symbol table stores all information related to the local scope of the program, and is accessed in Python using locals() method.*

The local scope could be within a function, within a class, etc.

*Likewise, a **Global** symbol table stores all information related to the global scope of the program, and is accessed in Python using globals() method.*

The global scope contains all functions, variables which are not associated to any class or function.

Syntax of globals()

The globals table dictionary is the dictionary of the current module

The syntax of globals() method is: globals()

globals() Parameters

The globals() method doesn't take any parameters.

Return value from globals()

The globals() method returns the dictionary of the current global symbol table.

Example 1: How globals() method works in Python ?

```
a = 100
b = 4
def foo():
    x = 100 # x is a local variable
print(globals())
```

```
Expected Output :

{'__builtins__': <module 'builtins' (built-in)>,
'__cached__': None,'
__doc__': None,
:.....
'a': 100,
'b': 4,
'foo': <function foo at 0x7f699ca1e2f0>,
'print': <function print at 0x7f699ca1e6a8>}
```

The variable we have defined in the module comes at last.Notice that the local variable x defined inside the foo() function is not included in the result. To access the local namespace use the locals() function.

Example 2: Modify global variable using global( )

```
a = 5
def  func():
        c = 10
        d = c + a
        globals()['a'] = d   # Calling globals()
        print (a)

func()
```

Enclosing Scope :

If a variable is declared in an enclosing function, it is non local to nested functions. It allows you to assign to variables in an outer, but no-global, scope.

Here's an example that tries to reassign enclosing (outer) function's local variable inside a nested (inner) function.

```
def f1():

    x = 42    #Non-local variable

    def f2():    # nested function

        x = 0

        print(x)       # x is 0

    f2()

    print(x)           # x is still 42

f1()
```

Here, the value of existing variable x didn't change. Because Python created a new local variable named x that shadows the variable in the outer scope.

Preventing that behavior is where the nonlocal keyword comes in.

```
def f1():

    x = 42

    def f2():     # nested function

        nonlocal x

        x = 0

        print(x)       # x is now 0

    f2()

    print(x)           # x remains 0

f1()
```

The x inside the nested function now refers to the x outside the function, so changing x inside the function changes the x outside it.

The usage of nonlocal is very similar to that of global, except that the former is primarily used in nested methods.

Scoping Rule – LEGB Rule

When a variable is referenced, Python follows LEGB rule and searches up to four scopes in this order:

first in the local (L) scope,

then in the local scopes of any enclosing (E) functions and lambdas,

then in the global (G) scope,

and finally in then the built-in (B) scope

and stops at the first occurrence.

If no match is found, Python raises a ***NameError*** *exception.*

Return Value

To return a value from a function, simply use a return statement. Once a return statement is executed, nothing else in the function body is executed.

```
# Return sum of two values

def sum(a, b):

    return a + b

x = sum(3, 4)

print(x) # Prints 7
```

Remember! a python function always returns a value. So, if you do not include any return statement, it automatically returns ***None****.*

Return Multiple Values

Python has the ability to return multiple values, something missing from many other languages. You can do this by separating return values with a comma.

```
# Return addition and subtraction in a tuple

def func(a, b):

    return a+b, a-b

result = func(3, 2)

print(result)

# Prints (5, 1)
```

When you return multiple values, Python actually packs them in a single tuple and returns it. You can then use multiple assignment to unpack the parts of the returned tuple.

```
# Unpack returned tuple

def func(a, b):

    return a+b, a-b

add, sub = func(3, 2)

print(add) # Prints 5

print(sub) # Prints 1
```

Return vs print

Note that return and print are not interchangeable. print is a Python function that prints data to the screen. It enables us, users, see the data. return statement, on the other hand, makes data visible to the program.

Docstring

You can attach documentation to a function definition by including a string literal just after the function header. Docstrings are usually triple quoted to allow for multi-line descriptions.

```
def hello():

    """This function prints

        message on the screen"""

    print('Hello, World!')
```

To print a function's docstring, use the Python help() function and pass the function's name.

```
# Print docstring in rich format

help(hello)
```

```
# Help on function hello in module __main__:

# hello()

#       This function prints

#       message on the screen
```

You can also access the docstring through __doc__ attribute of the function.

```
# Print docstring in a raw format

print(hello.__doc__)

# Prints This function prints message on the screen
```

Python's Functions Are First-Class

Functions in Python are objects,When function is defined, python interpreter internally creates an Object They can be manipulated like other objects in Python. Therefore functions are called first-class citizens. This is not true in other OOP languages like Java or C#.

Characteristics of first-class functions :

- ✓ *It can be passed as an argument like int string. Means we can pass the whole function as an argument to second functions as an argument.*
- ✓ *It can be written as return value like int, string mean s we can write function a function from that called function.*
- ✓ *It is assignable to variable or it can assign to a variable*
- ✓ *We can store in a data structure like tuple, sets, dictionary.*

So the function is an object and it's first-class objects.

We willl be using this yell function for demonstration purposes. It's a simple toy example with easily recognizable output:

```
def yell(text):
    return text.upper( ) + '!'
>>> yell('hello')
Output :'HELLO!'
```

Assigning Functions to Variables

Because the yell function is an object in Python you can assign it to another variable, just like any other object:

```
>>> bark = yell
```

This line doesn't call the function. It takes the function object referenced by yell and creates a second name pointing to it, bark. You could now also execute the same underlying function object by calling bark:

```
>>> bark('woof')
Output :'WOOF!'
```

Function objects and their names are two separate concerns. Here's more proof: You can delete the function's original name (yell). Because another name (bark) still points to the underlying function you can still call the function through it:

```
>>> del yell
>>> yell('hello?') # NameError: "name 'yell' is not defined"
>>> bark('hey')
Output :'HEY!'
```

Functions Can Be Stored In Data Structures

As functions are first-class citizens you can store them in data structures, just like you can with other objects. For example, you can add functions to a list:

```
>>> funcs = [bark, str.lower, str.capitalize]

>>> funcs

[<function yell at 0x10ff96510>,

 <method 'lower' of 'str' objects>,

 <method 'capitalize' of 'str' objects>]
```

Accessing the function objects stored inside the list works like it would with any other type of object:

```
>>> for f in funcs:

        print(f, f('hey there'))

<function yell at 0x10ff96510> 'HEY THERE!'

<method 'lower' of 'str' objects> 'hey there'

<method 'capitalize' of 'str' objects> 'Hey there'
```

You can even call a function object stored in the list without assigning it to a variable first. You can do the lookup and then immediately call the resulting "disembodied" function object within a single expression:

```
>>> funcs[0]('heyho')

Output :'HEYHO!'
```

Functions Can Be Passed To Other Functions

Because functions are objects you can pass them as arguments to other functions. Here's a greet function that formats a greeting string using the function object passed to it and then prints it:

```
def greet(func):
    greeting = func('Hi, I am a Python program')
    print(greeting)
```

You can influence the resulting greeting by passing in different functions. Here's what happens if you pass the yell function to greet:

```
>>> greet(yell)    -- 'HI, I AM A PYTHON PROGRAM!'
```

Of course you could also define a new function to generate a different flavor of greeting.

```
def whisper(text):
    return text.lower() + '...'
>>> greet(whisper)'hi, i am a python program...'
```

The ability to pass function objects as arguments to other functions is powerful. It allows you to abstract away and pass around behavior in your programs. In this example, the greet function stays the same but you can influence its output by passing in different greeting behaviors.

Functions that can accept other functions as arguments are also called ***higher-order functions****. They are a necessity for the functional programming style.*

Functions Can Be Nested

Python allows functions to be defined inside other functions. These are often called nested functions or inner functions.

Here's an example:

```
def speak(text):
    def whisper(t):
        return t.lower() + '...'
    return whisper(text)
>>> speak('Hello, World')
Output : 'hello, world...'
```

Now, what's going on here? Every time you call speak it defines a new inner function whisper and then calls it.And whisper does not exist outside speak:

```
>>> whisper('Yo')    -- NameError: "name 'whisper' is not defined"
>>> speak.whisper    -- AttributeError: "'function' object has no attribute 'whisper'"
```

If you wanted to access that nested whisper function from outside speak you can return the inner function to the caller of the parent function.

For example, here's a function defining two inner functions. Depending on the argument passed to top-level function it selects and returns one of the inner functions to the caller:

```
def get_speak_func(volume):
    def whisper(text):
        return text.lower() + '...'
    def yell(text):
        return text.upper() + '!'
    if volume > 0.5:
        return yell
    else:
        return whisper
```

Notice how get_speak_func doesn't actually call one of its inner functions—it simply selects the appropriate function based on the volume argument and then returns the function object:

```
>>> get_speak_func(0.3)

<function get_speak_func.<locals>.whisper at 0x10ae18>

>>> get_speak_func(0.7)

<function get_speak_func.<locals>.yell at 0x1008c8>
```

Of course you could then go on and call the returned function, either directly or by assigning it to a variable name first:

```
>>> speak_func = get_speak_func(0.7)
>>> speak_func('Hello')
Output : 'HELLO!'
```

This means not only can functions accept behaviors through arguments but they can also return behaviors.

Functions Can Capture Local State

You just saw how functions can contain inner functions and that it's even possible to return these (otherwise hidden) inner functions from the parent function.

Not only can functions return other functions, these inner functions can also capture and carry some of the parent function's state with them.

Let's slightly rewrite the previous get_speak_func example to illustrate this. The new version takes a "volume" and a "text" argument right away to make the returned function immediately callable:

```
def get_speak_func(text, volume):
    def whisper():
        return text.lower() + '...'
    def yell():
        return text.upper() + '!'
    if volume > 0.5:
        return yell
    else:
        return whisper
>>> get_speak_func('Hello, World', 0.7)()
```

Output : 'HELLO, WORLD!'

Take a good look at the inner functions whisper and yell now. Notice how they no longer have a text parameter? But somehow they can still access the text parameter defined in the parent function. In fact, they seem to capture and "remember" the value of that argument.

Functions that do this are called ***lexical closures*** *(or just closures, for short). A closure remembers the values from its enclosing lexical scope even when the program flow is no longer in that scope.*

Python function redefinition

Python is dynamic in nature. It is possible to redefine an already defined function

```
#function redefination

from time import gmtime, strftime

def show_message(msg):
    print(msg)

show_message("Ready.")

def show_message(msg):
    print(strftime("%H:%M:%S", gmtime()))
    print(msg)

show_message("Processing.")
```

From the time module we import two functions which are used to compute the current time. We define a show_message() function. Later we provide a new definition of the same function.

The first definition of a function only prints a message to the console.

Later in the source code, we set up a new definition of the showMessage() function. The message is preceded with a timestamp.

```
#Output
Ready.
23:49:33 Processing.
```