What is Function ?

A function is a block of code that only runs when it is called. Python functions return a value using a return statement, if one is specified. A function can be called anywhere after the function has been declared.

By itself, a function does nothing. But, when you need to use a function, you can call it, and the code within the function will be executed.

As our program grows larger and larger, functions make it more organized and manageable.Furthermore, it avoids repetition and makes the code reusable.

Inshort,

Python function is a sequence of statements that execute in a certain order, we associate a name with it. This lets us reuse code.

We define a function using the ‘def’ keyword.

How to Define a Python Function?

Syntax of Function

In [1]:
```python
def function_name(parameters):
    """docstring"""
    statement(s)
```

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

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

7. An optional `return` statement to return a value from the function.

Inshort, A function is denoted by the def keyword, followed by a function name, and a set of parenthesis.

For this example, we're going to create a simple function that prints out the statement It's Monday! to the console. To do so, we can use this code:

In [2]:
```
def print_monday():
    print("It's Monday!")
```

When we run our code, nothing happens. This is because, in order for our function to run, we need to call it. To do so, we can reference our function name like so:

In [3]:
```
def print_monday():
    print("It's Monday!")

print_monday()
```

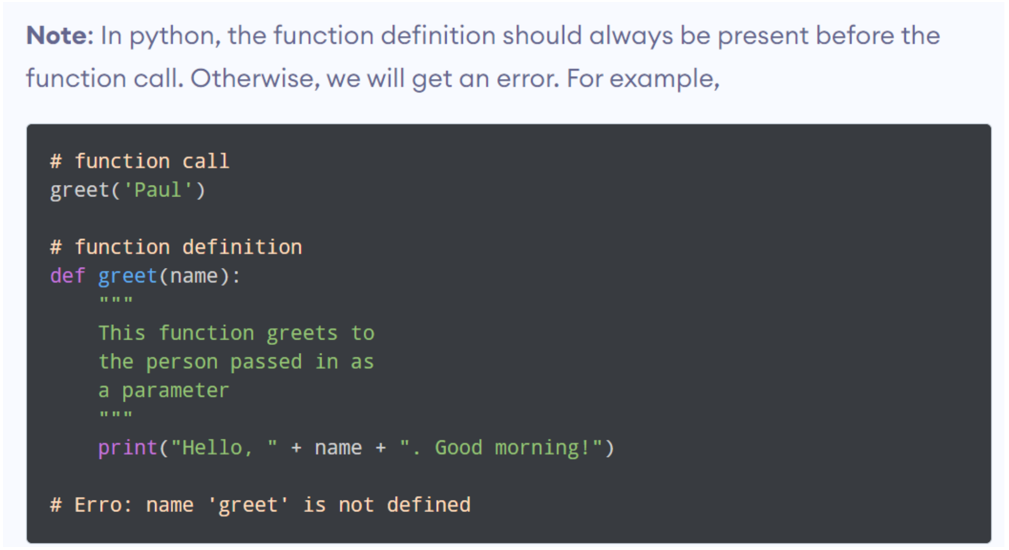
```
It's Monday!
```

Let's break down the main components of our function:

- the def keyword is used to indicate that we want to create a function.
- print_monday is the name of our function. This must be unique.
- () is where our parameters will be stored. We'll talk about this later.
- `:` marks the end of the header of our function.

In [4]:
```
from IPython.display import Image
Image("C:\\Users\\deepali\\OneDrive\\Desktop\\funct.png",width=500)
```

Out[4]:



Note: In python, the function definition should always be present before the function call. Otherwise, we will get an error. For example,

```
# function call
greet('Paul')

# function definition
def greet(name):
    """
    This function greets to
    the person passed in as
    a parameter
    """
    print("Hello, " + name + ". Good morning!")

# Erro: name 'greet' is not defined
```

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. Unless you can remember what you had for dinner last week, always document your code.

In the above image, 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.

For example:

Try running the following into the Python shell to see the output:

In [5]:
```python
def greet(name):
    """
    This function greets to
    the person passed in as
    a parameter
    """
    print("Hello, " + name + ". Good morning!")

greet('Paul')
```

```
Hello, Paul. Good morning!
```

In [6]:
```python
print(greet.__doc__)
```

```

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

The return statement

In a function, we use the return statement at the end of the function and it helps us to return the result of the function. This statement terminates the function execution and transfers the result where the function is called.

Note that we cannot use the return statement outside of the function.

Syntax:

In [ ]:
```python
return [expression_list]
```

It can contain the expression which gets evaluated and the value is returned to the caller function. If the return statement has no expression or does not exist itself in the function then it returns the **None** object.

Consider the following example:

Example 1: Creating Function with Return Statement

In [8]:
```python
# defining Function

def sum():
    a = 20
    b = 40
    c = a+b
    return c
# calling sum() function in print statement
print("The sum is given by:",sum())
```

```
The sum is given by: 60
```

In the above code, we have defined the function named sum, and it has a statement c = a+b, which computes the given values, and the result is returned by the return statement to the caller function.

Example 2: Creating function without return statement

In [9]:
```python
def sum():
    a = 20
    b = 40
    c = a+b
# calling sum() function in print statment
print(sum())
```

```
None
```

In the above code, we have defined the same function but this time we use it without the return statement and we have observed that the sum() function returned the None object to the caller function.

Example 3:

In [10]:

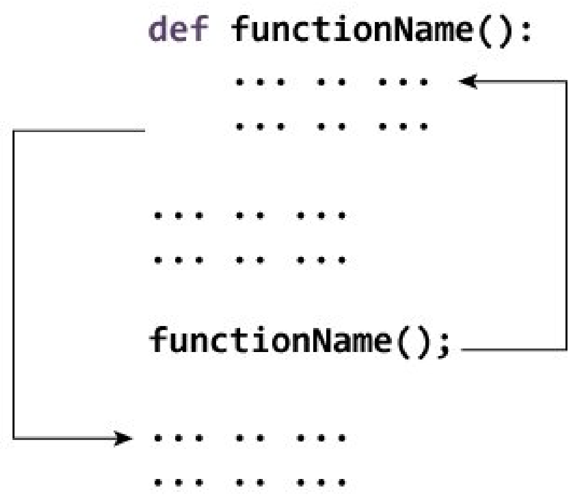
```python
def absolute_value(num):
    """This function returns the absolute
    value of the entered number"""

    if num >= 0:
        return num
    else:
        return -num


print(absolute_value(2))

print(absolute_value(-4))
```

```
2
4
```

How does function works?

In [11]:
```
from IPython.display import Image
Image("C:\\Users\\deepali\\OneDrive\\Desktop\\funct_works.jpg",width=300)
```

Out[11]:


```
def functionName():
    ... .. ...
    ... .. ...

... .. ...
... .. ...

functionName();

... .. ...
... .. ...
```

Types of Functions

Basically, we can divide functions into the following two types:

1. Built-in functions - Functions that are built into Python.
2. User-defined functions - Functions defined by the users themselves.

1. Built-in functions

Python has several functions that are readily available for use. These functions are called built-in functions.

Function	Description
abs()	Returns the absolute value of a number

Function	Description
all()	Returns True if all items in an iterable object are true
any()	Returns True if any item in an iterable object is true
ascii()	Returns a readable version of an object. Replaces none-ascii characters with escape character
bin()	Returns the binary version of a number
bool()	Returns the boolean value of the specified object
bytearray()	Returns an array of bytes
bytes()	Returns a bytes object
callable()	Returns True if the specified object is callable, otherwise False
chr()	Returns a character from the specified Unicode code.
classmethod()	Converts a method into a class method
compile()	Returns the specified source as an object, ready to be executed
complex()	Returns a complex number
delattr()	Deletes the specified attribute (property or method) from the specified object
dict()	Returns a dictionary (Array)
dir()	Returns a list of the specified object's properties and methods
divmod()	Returns the quotient and the remainder when argument1 is divided by argument2
enumerate()	Takes a collection (e.g. a tuple) and returns it as an enumerate object
exec()	Executes the specified code (or object)
filter()	Use a filter function to exclude items in an iterable object
float()	Returns a floating point number
format()	Formats a specified value
frozenset()	Returns a frozenset object
globals()	Returns the current global symbol table as a dictionary
hasattr()	Returns True if the specified object has the specified attribute (property/method)
hash()	Returns the hash value of a specified object
help()	Executes the built-in help system

Function	Description
hex()	Converts a number into a hexadecimal value
id()	Returns the id of an object
input()	Allowing user input
int()	Returns an integer number
isinstance()	Returns True if a specified object is an instance of a specified object
iter()	Returns an iterator object
len()	Returns the length of an object
list()	Returns a list
locals()	Returns an updated dictionary of the current local symbol table
map()	Returns the specified iterator with the specified function applied to each item
max()	Returns the largest item in an iterable
memoryview()	Returns a memory view object
min()	Returns the smallest item in an iterable
open()	Opens a file and returns a file object
pow()	Returns the value of x to the power of y
print()	Prints to the standard output device
range()	Returns a sequence of numbers, starting from 0 and increments by 1 (by default)
repr()	Returns a readable version of an object
reversed()	Returns a reversed iterator
round()	Rounds a numbers
set()	Returns a new set object
setattr()	Sets an attribute (property/method) of an object
slice()	Returns a slice object
sorted()	Returns a sorted list
@staticmethod()	Converts a method into a static method
str()	Returns a string object

Function	Description
sum()	Sums the items of an iterator
super()	Returns an object that represents the parent class
tuple()	Returns a tuple
type()	Returns the type of an object
vars()	Returns the **dict** property of an object
zip()	Returns an iterator, from two or more iterators

2. User defined Functions

A function that you define yourself in a program is known as user defined function. You can give any name to a user defined function, however you cannot use the Python keywords as function name.

In python, we define the user-defined function using `def` keyword, followed by the function name.

Function name is followed by the parameters in parenthesis, followed by the colon

For example:

In [ ]:
```
def function_name(parameter_1, parameter_2, ...) :
      statements
      ....
```

Calling a user-defined function You can call a user defined function by using function name followed by the arguments in the parenthesis.

For example:

Example of a user-defined function

In the following example we have defined a user-defined function sum. This function can accept two arguments as we have defined this function with two parameters. Inside the print() function we are calling the function sum and passing the numbers x & y as arguments.

In the sum() function we have a return statement that returns the sum of two parameters, that are passed to the function as arguments.

As you can see, we have used a print() function in the following example without even defining that function, this is because print() is a built-in function, which is already available to use and we can just call it.

In [13]:

```
# Program to demonstrate the
# use of user defined functions

def sum(a,b):
    total = a + b
    return total

x = 10
y = 20

print("The sum of",x,"and",y,"is:",sum(x, y))
```

```
The sum of 10 and 20 is: 30
```

Python Function Arguments

In Python, you can define a function that takes variable number of arguments.

Types of Python Function Arguments

1. Default Argument in Python

Python Program arguments can have default values. We assign a default value to an argument using the assignment operator in python(=).

When we call a function without a value for an argument, its default value (as mentioned) is used.

In [14]:
```
def greeting(name='User'):
    print(f"Hello, {name}")
greeting('Deepali')
```

```
Hello, Deepali
```

In [15]:
```
greeting()
```

```
Hello, User
```

Here, when we call greeting() without an argument, the name takes on its default value- ‘User’.

Any number of arguments can have a default value. But you must make sure to not have a non-default argument after a default argument.

In other words, if you provide a default argument, all others succeeding it must have default values as well.

The reason is simple. Imagine you have a function with two parameters.

The first argument has a default value, but the second doesn’t. Now when you call it(if it was allowed), you provide only one argument.

The interpreter takes it to be the first argument. What happens to the second argument, then? It has no clue.

In [16]:
```
def sum(a=1,b):
    return a+b
```

```
  File "<ipython-input-16-394c235213d9>", line 1
    def sum(a=1,b):
                ^
SyntaxError: non-default argument follows default argument
```

This was all about the default arguments in Python

2. Python Keyword Arguments

With keyword arguments in python, we can change the order of passing the arguments without any consequences.

Let’s take a function to divide two numbers, and return the quotient.

In [17]:
```python
def divide(a,b):
    return a/b
divide(7,2)

```
Out[17]:
```
3.5
```

We can call this function with arguments in any order, as long as we specify which value goes into what.

In [18]:
```python
divide(a=1,b=2)
```
Out[18]:
```
0.5
```

In [19]:
```python
divide(b=2,a=1)
```
Out[19]:
```
0.5
```

As you can see, both give us the same thing. These are keyword python function arguments.

But if you try to put a positional argument after a keyword argument, it will throw Python exception of SyntaxError.

In [20]:
```python
divide(b=2,1)
```
```
  File "<ipython-input-20-c3d95257b15f>", line 1
    divide(b=2,1)
              ^
SyntaxError: positional argument follows keyword argument
```

3. Python Arbitrary Arguments

You may not always know how many arguments you’ll get. In that case, you use an asterisk(*) before an argument name.

In [21]:
```python
def sayhello(*names):
    for name in names:
        print(f"Hello, {name}")
```

And then when you call the function with a number of arguments, they get wrapped into a Python tuple.

We iterate over them using the for loop in python.

In [22]:
```python
sayhello('Deepali','Neelam','Harsh')
```

```
Hello, Deepali
Hello, Neelam
Hello, Harsh
```

This was all about the Python Function Arguments.

Hence, we conclude that Python Function Arguments and its three types of arguments to functions. These are- default, keyword, and arbitrary arguments.

Where default arguments help deal with the absence of values, keyword arguments let us use any order.

Finally, arbitrary arguments in python save us in situations where we’re not sure how many arguments we’ll get.

A Note: Parameters vs. Arguments

The terms parameter and argument refer to the same thing: passing information to a function. But, there is a subtle difference between the two.

A parameter is the variable inside the parenthesis in a function. An argument is the value that is passed to a function when it is called. So, in our last example(cell no.17), “a” and “b” are parameters, and 7 and 2 are arguments.

Let’s see the given example, which contains a function that accepts a string as the argument.

Python Function to find the sum of two variables:

In [23]:
```python
# defining a function with sum of two variables

def sum(num1,num2):
    return num1 + num2

# taking value from a user as an input

num1 = int(input("Enter value of num1 : "))
num2 = int(input("Enter value of num2 : "))

# calculating and printing sum of num1 and num2

print("Sum = ",sum(num1,num2))
```

```
Enter value of num1 : 20
Enter value of num2 : 10
Sum =  30
```

Write a Python program to find the maximum from the given three numbers.

Program to implement the given functionality:

In [24]:
```python
def max_of_two(x,y):
    if x > y:
        return x
    return y
def max_of_three(x,y,z):
    return max_of_two(x,max_of_two(z,y))
```

In [25]:
```python
print(max_of_three(3, 6, -5))

```

```
6
```

Explanation of the Program:

In this example first, we will make a user-defined function named max_of_two to find the maximum of two numbers and then utilizes that function to find the maximum from the three numbers given by using the function max_of_three. For finding the maximum from three numbers, we pick two numbers from those three and apply the max_of_two function to those two, and again apply the max_of_two function to the third number and result of the maximum of the other two.

Write a Python program to calculate the sum of all the numbers present in a list.

Program to implement the given functionality:

In [26]:
```python
def sum(numbers):
    total = 0
    for element in numbers:
        total+=element
    return total
```

In [27]:
```python
print(sum((2,5,1,2)))
```

```
10
```

Explanation of the Program:

In this example, we define a function named sum() that takes a list of numbers as input and we initialized a variable total to zero. Then, with the help of a for loop, we traverse the complete list and then update the value of the total variable to its previous value plus the value traversed at that time. We do the updation of the total variable until we reach the last of the list and then finally we return the value of the total variable.

Write a Python program to calculate the multiplication of all the numbers present in a list.

Program to implement the given functionality:

In [28]:
```
def multipy(numbers):
    total = 1
    for element in numbers:
        total*=element
    return total
```

In [29]:
```
multipy((4,2,9,5))
```

Out[29]:
```
360
```

Explanation of the Program:

In this example, we define a function named multiply() that takes a list of numbers as input and we initialized a variable total to one. Then, with the help of a for loop, we traverse the complete list and then update the value of the total variable to its previous value multiply by the value traversed at that time. We do the updation of the total variable until we reach the last of the list and then finally we return the value of the total variable.

Write a Python program that takes a string as an input and calculates the number of upper case and lower case letters present in the string.

Program to implement the given functionality:

In [30]:
```
def string_test(string):
    d={"UPPER_CASE" : 0,"LOWER_CASE":0}
    for character in string:
        if character.isupper():
            d["UPPER_CASE"]+=1
        elif character.islower():
            d["LOWER_CASE"]+=1
        else:
            pass

    print("Original string : ",string)
    print(" No of Upper_case string : ", d["UPPER_CASE"])
    print(" No of lower_case string : ", d["LOWER_CASE"])

```

In [31]:
```python
string_test("Hi , My name is Deepali. I hope my Python notes are useful to all .")
```

```
Original string :  Hi , My name is Deepali. I hope my Python notes are useful to all .
 No of Upper_case string :  5
 No of lower_case string :  44
```

Explanation of the Program:

In this example, we initialized a dictionary having two keys named UPPER_CASE and LOWER_CASE with values 0. Then, with the help of a for loop, we traverse the string and check whether each character is either lower case or upper case and whatever that character is we increment the value of that variable by one, and we do the same process until we reached upto the end of the string.

Recursive function [Very important concept for interview]

A function is said to be a recursive if it calls itself. For example, lets say we have a function abc() and in the body of abc() there is a call to the abc().

Python example of Recursion In this example we are defining a user-defined function factorial(). This function finds the factorial of a number by calling itself repeatedly until the base case(We will discuss more about base case later, after this example) is reached.

Write a Python program that takes a number(non-negative integer) as an argument and calculates its factorial.

Program to implement the given functionality:

In [34]:
```python
def factorial(number):
    """This function calls itself to find
    the factorial of a number"""
    if number ==1:
        return 1
    else:
        return number*factorial(number-1)

number = int(input("Enter a number to compute factorial :"))
print(factorial(number))
```

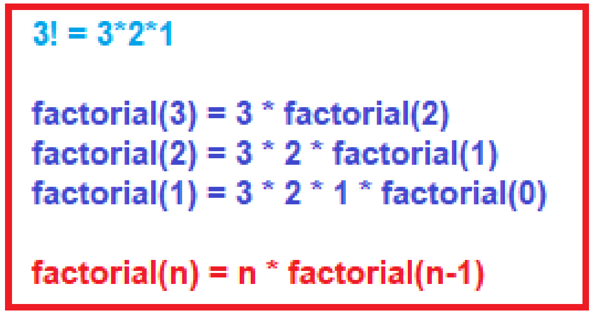
```
Enter a number to compute factorial :5
120
```

Explanation of the Program:

To implement this functionality, we use the concept of recursion with the base condition. Here we make a function named factorial that takes a number as an input and then recursively calls the same function up to we reach the base condition included in that function.

To understand the recursive definition of the program, let’s understand the below image:

In [35]:
```python
from IPython.display import Image
Image("C:\\Users\\deepali\\OneDrive\\Desktop\\factorial.png",width=300)
```

Out[35]:

3! = 3*2*1

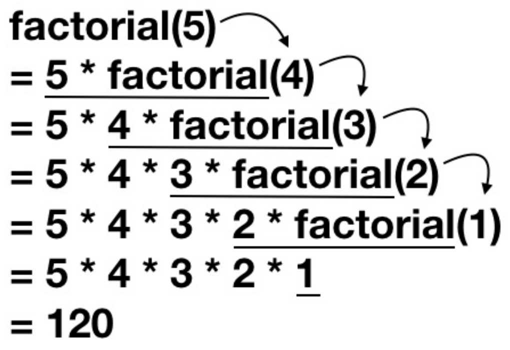
factorial(3) = 3 * factorial(2)
factorial(2) = 3 * 2 * factorial(1)
factorial(1) = 3 * 2 * 1 * factorial(0)

factorial(n) = n * factorial(n-1)

In [36]:

```python
from IPython.display import Image
Image("C:\\Users\\deepali\\OneDrive\\Desktop\\Recursion.jpg",width=300)
```

Out[36]:



factorial(5)
= 5 * factorial(4)
= 5 * 4 * factorial(3)
= 5 * 4 * 3 * factorial(2)
= 5 * 4 * 3 * 2 * factorial(1)
= 5 * 4 * 3 * 2 * 1
= 120

Note: factorial(1) is a base case for which we already know the value of factorial. The base case is defined in the body of function with this code:

In [ ]:

```python
if num == 1:
    return 1
```

What is a base case in recursion ?

When working with recursion, we should define a base case for which we already know the answer. In the above example we are finding factorial of an integer number and we already know that the factorial of 1 is 1 so this is our base case.

Each successive recursive call to the function should bring it closer to the base case, which is exactly what we are doing in above example.

We use base case in recursive function so that the function stops calling itself when the base case is reached. Without the base case, the function would keep calling itself indefinitely.

Why use recursion in programming?

We use recursion to break a big problem in small problems and those small problems into further smaller problems and so on. At the end the solutions of all the smaller subproblems are collectively helps in finding the solution of the big main problem.

Advantages of recursion Recursion makes our program:

1. Easier to write.
2. Readable – Code is easier to read and understand.
3. Reduce the lines of code – It takes less lines of code to solve a problem using recursion.

Disadvantages of recursion

1. Not all problems can be solved using recursion.
2. If you don’t define the base case then the code would run indefinitely.
3. Debugging is difficult in recursive functions as the function is calling itself in a loop and it is hard to understand which call is causing the issue.
4. Memory overhead – Call to the recursive function is not memory efficient.

Python Anonymous/Lambda Function

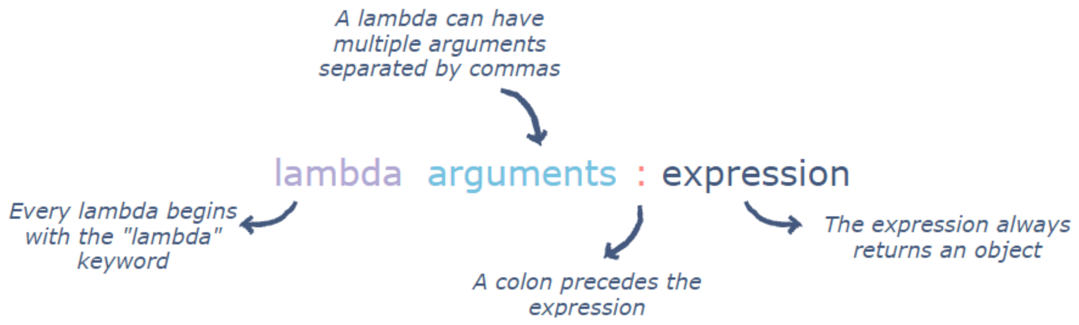
In Python, an anonymous function is a function that is defined without a name.

While normal functions are defined using the def keyword in Python, anonymous functions are defined using the lambda keyword.

Hence, anonymous functions are also called lambda functions.

In [37]:
```python
from IPython.display import Image
Image("C:\\Users\\deepali\\OneDrive\\Desktop\\lambda.png",width=600)
```

Out[37]:


A lambda can have multiple arguments separated by commas
lambda arguments : expression
Every lambda begins with the "lambda" keyword
A colon precedes the expression
The expression always returns an object

How to use lambda Functions in Python?

A lambda function in python has the following syntax.

Syntax of Lambda Function in python
lambda arguments: expression

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

Example of Lambda Function in python

Here is an example of lambda function that doubles the input value.

In [38]:
```python
# program to show use of lambda function

double = lambda x:x * 2
print(double((5)))
```

```
10
```

In the above program, `lambda x: x * 2` is the lambda function. Here `x` is the argument and `x * 2` is the expression that gets evaluated and returned.

This function has no name. It returns a function object which is assigned to the identifier double. We can now call it as a normal function. The statement

In [39]:
```python
double = lambda x: x * 2

```

is nearly the same as:

In [40]:
```python
def double(x):
    return x*2
```

The purpose of lambdas

A lambda is much more readable than a full function since it can be written in-line. Hence, it is a good practice to use lambdas when the function expression is small.

The beauty of lambda functions lies in the fact that they return function objects. This makes them helpful when used with functions like `map` or `filter` which require function objects as arguments.

Map with lambda

The `map()` function in Python takes in a function and a list.

The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

Here is an example use of map() function to double all the items in a list.

In [41]:
```
# Program to double each item in a list using map()

my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))

print(new_list)
```

```
[2, 10, 8, 12, 16, 22, 6, 24]
```

Filter with lambda

The `filter()` function in Python takes in a function and a list as arguments.

The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

Here is an example use of filter() function to filter out only even numbers from a list.

In [42]:
```
# Program to filter out only the even items from a list
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(filter(lambda x: (x%2 == 0) , my_list))

print(new_list)
```

```
[4, 6, 8, 12]
```

Global , Local and Nonlocal 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.

Let's see an example of how a global variable is created in Python.

Example 1: Create a Global Variable

In [43]:
```python
x = "global"

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


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

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

In the above code, we created x as a global variable and defined a foo() to print the global variable x. Finally, we call the foo() which will print the value of x.

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

In [44]:
```
x = "global"

def foo():
    x = x * 2
    print(x)

foo()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-44-5bb9d7e58dc3> in <module>
      5     print(x)
      6 
----> 7 foo()

<ipython-input-44-5bb9d7e58dc3> in foo()
      2 
      3 def foo():
----> 4     x = x * 2
      5     print(x)
      6 

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 foo().

To make this work, we use the global keyword

What is the global keyword

In Python, global keyword allows you to modify the variable outside of the current scope. It is used to create a global variable and make changes to the variable in a local context.

Rules of global Keyword
The basic rules for global keyword in Python are:

1. When we create a variable inside a function, it is local by default.

2. When we define a variable outside of a function, it is `global` by default. You don't have to use `global keyword`.
3. We use `global keyword` to read and write a `global variable` inside a function.
4. Use of `global keyword` outside a function has no effect.

Use of global Keyword

Let's take an example.

Example 1: Accessing global Variable From Inside a Function

In [45]:
```python
a = 1 # global variable

def add():
    print(a)

add()
```

```
1
```

However, we may have some scenarios where we need to modify the global variable from inside a function.

Example 2: Modifying Global Variable From Inside the Function

In [46]:
```python
a = 1

def add():
    a = a+2 # increment a by 2
    print(a)

add()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-46-cafc5b44c9e4> in <module>
      5     print(a)
      6 
----> 7 add()

<ipython-input-46-cafc5b44c9e4> in add()
      2 
      3 def add():
----> 4     a = a+2 # increment a by 2
      5     print(a)
      6 

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

This is because we can only access the global variable but cannot modify it from inside the function.

The solution for this is to use the global keyword.

Example 3: Changing Global Variable From Inside a Function using global

In [47]:
```python
c = 0 # global variable

def add():
    global c
    c = c + 2 # increment by 2
    print("Inside add():", c)

add()
print("In main:", c)
```

```
Inside add(): 2
In main: 2
```

In the above program, we define c as a global keyword inside the add() function.

Then, we increment the variable c by 1, i.e c = c + 2. After that, we call the add() function. Finally, we print the global variable c.

As we can see, change also occurred on the global variable outside the function, c = 2.

Global Nested Function

Example 4: Using a Global Variable in Nested Function

In [48]:
```python
def foo():
    x = 20

    def bar():
        global x
        x = 25

    print("Before calling bar: ", x)
    print("Calling bar now")
    bar()
    print("After calling bar: ", x)

foo()
print("x in main: ", x)
```

```
Before calling bar:  20
Calling bar now
After calling bar:  20
x in main:  25
```

In the above program, we declared a global variable inside the nested function bar(). Inside foo() function, x has no effect of the global keyword.

Before and after calling bar(), the variable x takes the value of local variable i.e x = 20. Outside of the foo() function, the variable x will take value defined in the bar() function i.e x = 25. This is because we have used global keyword in x to create global variable inside the bar() function (local scope).

If we make any changes inside the bar() function, the changes appear outside the local scope, i.e. foo().

Local Variables

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

Example : Accessing local variable outside the scope

In [49]:
```python
def foo():
    s = "Local"
foo()
print(s)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-49-06e1daa78987> in <module>
      2     s = "Local"
      3 foo()
----> 4 print(s)

NameError: name 's' 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 foo() or local scope.

Example : Create a Local Variable

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

In [50]:
```python
def foo():
    s = "Local"
    print(s)
foo()
```

```
Local
```

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.

Let's see an example of how a nonlocal variable is used in Python.

We use nonlocal keywords to create nonlocal variables.

Example : Create a nonlocal variable

In [51]:
```python
def outer():
    x = "local"

    def inner():
        nonlocal x
        x = "nonlocal"
        print("inner:", x)

    inner()
    print("outer:", x)


outer()
```

```
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.