Python Functions

- Functions - It is a programming constructs that takes input and give output.
- Two types of functions :
  - built in functions
  - user defined functions
- Function remove code reusability issue.
- Function principle :
  - abstraction
  - decomposition

- Function components: def function_name(parameters): ---- # function body """doct string""" logic of function here.... return something function_name(arguments) ----# function call

Creating a function with docstring

In [9]:
```
def is_even(num):
    """ This is a function that take a number as a argument
    and check if the given number is even or odd.
    If the nunber is evem - return "even"
    else return "odd"
    """
    if type(num)==int:
        if num%2==0:
            return "even"
        else:
            return "odd"
    else:
        return "give a integer value"
```

In [11]:
```
is_even(2)
```

Out[11]:
```
'even'
```

```
In [13]: is_even(23)
```

```
Out[13]: 'odd'
```

2 point of views

```
In [17]: is_even("hii")
```

```
Out[17]: 'give a integer value'
```

```
In [21]: # to print doc of the function
         print(is_even.__doc__)
```

```
This is a function that take a number as a argument
   and check if the given number is even or odd.
   If the nunber is evem - return "even"
   else return "odd"
```

`Parameter Vs Arguments`

- Arguments - value pass at time of function call
- Parameter - value comes in function body

Types of Arguments

- Default Argument - here we have default value in parameter
- Positional Argument - default argument and parameter follow a order of 1:1
- Keyword Argument - keyword argument > positional argument

Default Argument

```
In [48]: def power(a=1,b=1):
             return a**b
```

In [38]:
```
print(power())
print(power(10))
print(power(3,3))
```

```
1
10
27
```

Positional Argument

In [46]:
```
print(power(3,2))
print(power(2,3))
```

```
9
8
```

Keyword Argument

In [58]:
```
print(power(b=3,a=2))
print(power(a=3,b=2))
```

```
8
9
```

`*Args and **Kwargs`

args and *kwargs are special Python keywords that are used to pass the variable length of arguments to a function

*Args

- Allow us to pass a variable number of non keyword argument to a function.

In [100…
```
def mul(*args):
    print(args,type(args),*args)
    prod=1
    for i in args:
        prod=prod*i
    return prod
```

```
# When *args is used here, it unpacks each element of args
# (which are tuples) and passes them individually to print.
# So print will treat each tuple within args as a separate argument,
# resulting in each tuple being printed on a separate line
# in environments that auto-format output.
```

In [96]:
```
print(mul(1),mul(1,2),mul(1,2,3,4,5))
```

```
(1,) <class 'tuple'> 1
(1, 2) <class 'tuple'> 1 2
(1, 2, 3, 4, 5) <class 'tuple'> 1 2 3 4 5
1 2 120
```

**Kwargs

- Allows us to pass any number of keyword argument in form of dictionary

In [115…
```
def display(**kwargs):
    print(kwargs,type(kwargs),*kwargs)
    for (key,value) in kwargs.items():
        print(key,'->',value)
```

In [117…
```
display(india='delhi',srilanka='colombo',nepal='kathmandu',pakistan='islamabad')
```

```
{'india': 'delhi', 'srilanka': 'colombo', 'nepal': 'kathmandu', 'pakistan': 'islamabad'} <class 'dict'> india srilanka nepal pa
kistan
india -> delhi
srilanka -> colombo
nepal -> kathmandu
pakistan -> islamabad
```

Points to remember while using `*args and **kwargs`

- order of the arguments matter(normal -> `*args` -> `**kwargs`)
- The words “args” and “kwargs” are only a convention, you can use any name of your choice

How function execute in memory

When a function is called in Python, it executes through several steps in memory:

1. **Function Definition and Storage**: When Python encounters a function definition (using `def`), it creates a function object and stores it in memory. The function name is assigned a reference to this object, allowing it to be called later.

2. **Stack Frame Creation**: When a function is called, a new "stack frame" is created. This stack frame holds:

    - Local variables
    - Arguments passed to the function
    - A reference to the function's code and the current execution point within it
3. **Memory Allocation**: Local variables within the function are stored in memory within the stack frame. Each variable's name is a reference to its value, which may point to data in the heap if it’s a mutable object (like lists or dictionaries).

4. **Function Execution and Stack Management**: The Python interpreter executes the function line-by-line within the stack frame. Any new function calls from within this function will create additional stack frames, which are organized in a last-in, first-out order.

5. **Return and Cleanup**: Once a function completes (returns a result or finishes execution), its stack frame is removed from memory, and control returns to the previous frame. The memory for local variables is then freed up.

This stack-based execution and cleanup process helps manage memory efficiently and allows recursive and nested function calls.

Without return statement function

All function return something , if it is returning no valid input then it is returning "None" as a return value.

In [9]:
```python
L = [1,2,3]
print(L.append(4))
print(L)
```

```
None
[1, 2, 3, 4]
```

Variable Scope

Glbal vs Local scope

- Global scope : when variable is not enclosed in any user defined function.
- Local scope : when variable declared inside a user defined function.

In [18]:
```
def g(y):
    print(x)
    print(x+1)
x = 5
g(x)
print(x)
# here x=5 is in global scope but y is in local scope.
# x used inside g() is also global since there is no x in local
```

```
5
6
5
```

In [20]:
```
def f(y):
    x = 1
    x += 1
    print(x)
x = 5
f(x)
print(x)
# here x is in global where as y is local.
# here x inside g() is local var.
```

```
2
5
```

In [ ]:
```
def h(y):
    x += 1
x = 5
h(x)
print(x)
```

```
# here x=5 is global  and y is local in h()  but x inside h() is global
# x inside h() is global so we cannot change in global var in local so it will give error.
```

In [24]:
```
def f(x):
   x = x + 1
   print('in f(x): x =', x)
   return x

x = 3
z = f(x)
print('in main program scope: z =', z)
print('in main program scope: x =', x)
```

```
in f(x): x = 4
in main program scope: z = 4
in main program scope: x = 3
```

Nested Functions

In [28]:
```
def f():
  def g():
    print('inside function g')
    f()
  g()
  print('inside function f')
```

In [ ]:
```
f()
```

In [34]:
```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('in g(x): x =', x)
    h()
    return x

x = 3
z = g(x)
print(z)
```

```
in g(x): x = 4
4
```

In [36]:
```
def g(x):
    def h(x):
        x = x+1
        print("in h(x): x = ", x)
    x = x + 1
    print('in g(x): x = ', x)
    h(x)
    return x

x = 3
z = g(x)
print('in main program scope: x = ', x)
print('in main program scope: z = ', z)
```

```
in g(x): x =  4
in h(x): x =  5
in main program scope: x =  3
in main program scope: z =  4
```

Functions are 1st class citizen

Because function support all operations available to other entities. ex : can be use as an argument,return from a function,assigned as a variable.

- in python data type as well as function are first class citizens

In [54]:
```
# type and id
def square(num):
  return num**2

print(type(square))
print(id(square))
```

```
<class 'function'>
1935512324672
```

```
In [48]: # reassign
         x = square
         print(id(x))
         x(3)

1935501785056

Out[48]: 9

In [50]: # deleting a function
         del square

In [56]: # storing
         L = [1,2,3,4,square]
         L[-1](3)

Out[56]: 9

In [58]: s = {square}
         s

Out[58]: {<function __main__.square(num)>}

In [60]: # returning a function
         def f():
             def x(a, b):
                 return a+b
             return x

         val = f()(3,4)
         print(val)

7

In [62]: # function as argument
         def func_a():
             print('inside func_a')

         def func_b(z):
             print('inside func_c')
```

```
    return z()

print(func_b(func_a))
```

```
inside func_c
inside func_a
None
```

Benefits of using a Function

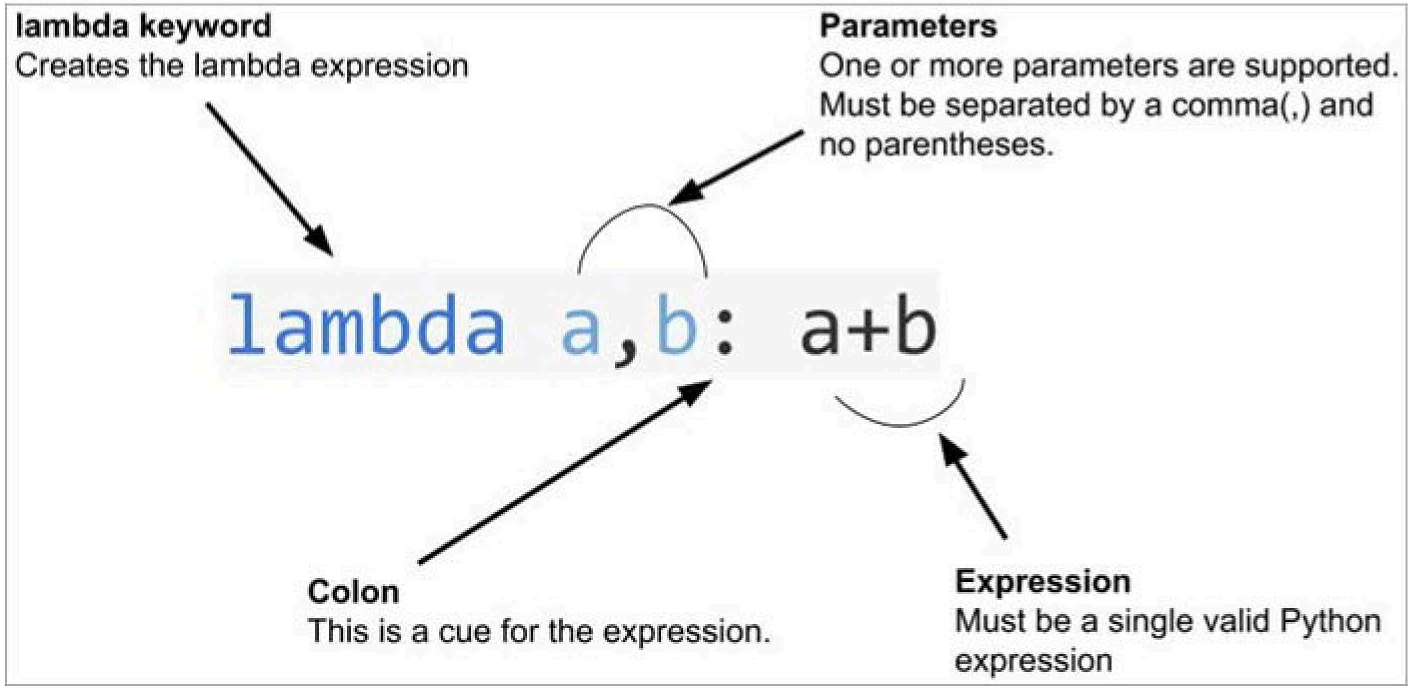
- Code Modularity
- Code Readibility
- Code Reusability

`Lambda Function`

Lambda Function

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.


lambda keyword
Creates the lambda expression
Parameters
One or more parameters are supported. Must be separated by a comma(,) and no parentheses.
lambda a,b: a+b
Colon
This is a cue for the expression.
Expression
Must be a single valid Python expression

```
In [75]: # x -> x^2
         (lambda x:x**2)(4)

Out[75]: 16
```

```
In [70]: # x,y -> x+y
         a = lambda x,y:x+y
         a(5,2)

Out[70]: 7
```

Diff between lambda vs Normal Function

- No name
- lambda has no return value(infact,returns a function)
- lambda is written in 1 line
- not reusable

Then why use lambda functions?

They are used with HOF

In [81]:
```
# check if a string has 'a'
(lambda s: 'a' in s)("apple")
```

Out[81]:
```
True
```

In [83]:
```
# odd or even
a = lambda x:'even' if x%2 == 0 else 'odd'
a(6)
```

Out[83]:
```
'even'
```

Higher Order Functions

Function that return a function or a function thatt receive another function

In [88]:
```
# a HOF function
def HOF_function(f,L):
    output=[]
    for i in L:
        output.append(f(i))
    print(output)
L=[1,2,3,4,5,6]
HOF_function(lambda x:x**3,L)
```

```
[1, 8, 27, 64, 125, 216]
```

In [92]:
```
def square(x):
  return x**2

def HOF_function(f,L):
    output=[]
    for i in L:
        output.append(f(i))
    print(output)
```

```
L=[1,2,3,4,5,6]
HOF_function(square,L)
```

```
[1, 4, 9, 16, 25, 36]
```

Map

- Map is a function that takes a lambda function and then storage over which function will work.
- It will move on each element and run the function expresion and finally create a map object.
- That later typecast into required data type

In [96]:
```
L=[1,2,1,43,5,32,2,1,34,2,43,5,6,7,34,1,34,7,8,6]
list(map(lambda x:L.count(x),L))
```

Out[96]:
```
[4, 3, 4, 2, 2, 1, 3, 4, 3, 3, 2, 2, 2, 2, 3, 4, 3, 2, 1, 2]
```

In [98]:
```
# square the items of a list
list(map(lambda x:x**2,[1,2,3,4,5]))
```

Out[98]:
```
[1, 4, 9, 16, 25]
```

In [100…
```
# odd/even labelling of list items
L = [1,2,3,4,5]
list(map(lambda x:'even' if x%2 == 0 else 'odd',L))
```

Out[100…
```
['odd', 'even', 'odd', 'even', 'odd']
```

In [106…
```
# fetch names from a list of dict

users = [
    {
        'name':'Rahul',
        'age':45,
        'gender':'male'
    },
    {
        'name':'Nitish',
        'age':33,
        'gender':'male'
    },
    {
        'name':'Ankita',
        'age':50,
        'gender':'female'
    }
]

list(map(lambda user:user['gender'],users))
```

Out[106…

```
['male', 'male', 'female']
```

Filter

- Filter is a function similar to maps but it also filter the value according to condition
- If we use maps instead of filter then it will return a booleans set of value

In [119…

```
L=[1,2,4,5,67,2,1,13,4,5,67,89,9]
list(filter(lambda x:x>5,L))
```

Out[119…

```
[67, 13, 67, 89, 9]
```

In [126…

```
# fetch fruits starting with 'a'
fruits = ['apple','guava','cherry']

list(filter(lambda x:x.startswith('a'),fruits))
```

Out[126…

```
['apple']
```

Reduce

In Python, `reduce` is a function from the `functools` module that applies a specified function cumulatively to the items in a sequence, reducing the sequence to a single value.

Syntax:

```python
from functools import reduce

result = reduce(function, sequence, initializer)
```

- **function**: A function that takes two arguments and returns a single result.
- **sequence**: The iterable (like a list) whose elements will be reduced.
- **initializer** (optional): An initial value for the reduction. If provided, `initializer` is placed before the items of the sequence.

How It Works

`reduce` starts with the first two items in the sequence, applies the function, and then takes the result and applies the function to the next item in the sequence, and so on, until only a single result is left.

Example

Suppose you want to find the product of a list of numbers:

```python
from functools import reduce

numbers = [1, 2, 3, 4]
result = reduce(lambda x, y: x * y, numbers)
print(result)  # Output: 24
```

Here's how it proceeds:

1. (1 * 2 = 2)
2. (2 * 3 = 6)
3. (6 * 4 = 24)

The final result is `24`.

When to Use `reduce`

`reduce` is useful for operations that need to reduce a list to a single cumulative value, such as summing, multiplying, or applying custom accumulation operations. However, in many cases, using `sum`, `min`, or `max` might be simpler and more readable if they fit the use case.

In [132…
```
# sum of all item
import functools

functools.reduce(lambda x,y:x+y,[1,2,3,4,5])
```

Out[132…
```
15
```

In [136…
```
# find min
functools.reduce(lambda x,y:x if x>y else y,[23,11,45,10,1])
```

Out[136…
```
45
```

END