**Functions**

**Single line function**

def single\_line(): statement

**Function with docstring**

def fn(arg1, arg2,...):

"""docstring"""

statement1

statement2

**Nested Python**

def fn(arg1, arg2,...):

"""docstring"""

statement1

statement2

def fn\_new(arg1, arg2,...):

statement1

statement2

...

...

**Function Call**

def typeOfNum(num): # Function header

# Function body

if num % 2 == 0:

def message():

print("You entered an even number.")

else:

def message():

print("You entered an odd number.")

message()

# End of function

typeOfNum(2) # call the function

typeOfNum(3) # call the function again

**Polymorphism**

* The behaviour of function may vary depending upon the arguments passed to it.
* The same function can accept arguments of different object types.
* If the objects find a matching interface, the function can process them.

Example:

def product(x, y) : return x \* y

print(product(4, 9)) # function returns 36

print(product('Python!', 2)) # function returns

# Python!Python!

print(product('Python 2 or 3?', '3')) # TypeError occurs

* Python is a dynamically typed language which means the types correlate with values, not with variables. Hence, the polymorphism runs unrestricted. That’s one of the primary differences between Python and other statically typed languages such as C++ or Java.

**Parameters and arguments**

**Parameters are variable used in the function**

**Arguments are values we pass to the function parameters.**

**Standard Arguments**

def fn(value):

print(value)

return

**Keyword Based Arguments**

def fn(value):

print(value)

return

fn(value=123) # output => 123

fn(value="Python!") # output => Python!

fn(value1="Python!") # wrong name used in the keyword argument

TypeError: fn() got an unexpected keyword argument 'value1'

**Arguments with Default values**

def daysInYear(is\_leap\_year=False):

if not is\_leap\_year:

print("365 days")

else:

print("366 days")

return

daysInYear()

daysInYear(True)

**Variable arguments**

You may encounter situations when you have to pass additional arguments to a Python function. We refer them as variable-length arguments.

The Python’s print () is itself an example of such a function which supports variable arguments.

To define a function with variable arguments, you need to prefix the parameter with an asterisk (\*)

def inventory(category, \*items):

print("%s [items=%d]:" % (category, len(items)), items)

for item in items:

print("-", item)

return

inventory('Electronics', 'tv', 'lcd', 'ac', 'refrigerator', 'heater')

inventory('Books', 'python', 'java', 'c', 'c++')

Electronics [items=5]: ('tv', 'lcd', 'ac', 'refrigerator', 'heater')

- tv

- lcd

- ac

- refrigerator

- heater

Books [items=4]: ('python', 'java', 'c', 'c++')

- python

- java

- c

- c++

def getMin(\*varArgs) :

min = varArgs[0]

for i in varArgs[1:] :

if i < min :

min = i

return min

min = getMin(21, -11, 17, -23, 6, 5, -89, 4, 9)

print(min)

-89

**Key word arguments**

# Python program to illustrate

# \*kargs for variable number of keyword arguments

def myFun(\*\*kwargs):

for key, value in kwargs.items():

print ("%s == %s" %(key, value))

# Driver code

myFun(first ='Geeks', mid ='for', last='Geeks')

last == Geeks

mid == for

first == Geeks

**Local Variable**

Visibility only inside a code block.

def fn(a, b) :

temp = 1

for iter in range(b) :

temp = temp\*a

return temp

print(fn(2, 4))

print(temp) # error : can not access 'temp' out of scope of function 'fn'

print(iter) # error : can not access 'iter' out of scope of function 'fn'

**Global variable**

x = 5

y = 55

def fn() :

global x

x = [3, 7]

y = [1, 33, 55]

# a local 'y' is assigned and created here

# whereas, 'x' refers to the global name

fn()

print(x, y)

Here, we have got the following three scripts:

* mod\_global.py: It contains the global definition and a function changing and displaying values.
* test1.py: It imports the first file and accesses the global variable.
* test2.py: It is using the “from” clause to import the first file and accessing the global variable.

# mod\_global.py

def fn1() :

global x

x = [1,2] ; y = [20, 200]

# a local 'y' is created – available only within 'f1'

# 'x' can be accessed anywhere after a call to 'f1'

fn1()

try :

print(x, y) # name 'y' is not defined – error

except Exception as ex:

print('y ->', ex)

print('x ->', x)

# test1.py

import mod\_global

print('test1 ->', mod\_global.x)

# test2.py

from mod\_global import \*

print('test2 ->', x)

Recursive Function

def calcFact(num) :

if(num != 1) :

return num \* calcFact(num-1)

else :

return 1

print(calcFact(4))

Output:

24

Function Attributes:

print(dir(testFunc))

**You can assign a function object to any other names.**

def testFunc(a, b) : print('testFunc called')

fn = testFunc

fn(22, 'bb')

Output:

testFunc called