**Functions**

**Function:** A block of organized, reusable code that is used to perform a single, related action

# Defining a Function

Rules:

1. Function blocks begin with the keyword def followed by the function name and parentheses ( ( ) ).
2. Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
3. The first statement of a function can be an optional statement - the documentation string of the function or docstring.
4. The code block within every function starts with a colon (:) and is indented.
5. The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

## Syntax

def functionname( parameters ):

"function\_docstring"

function\_suite

return [expression]

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined

## Example

def printme( str ):

"This prints a passed string into this function"

print (str)

return

# Calling a Function

You can execute it by calling it from another function or directly from the Python prompt

printme("This is user defined function!")

This is user defined function!

# Pass by Reference vs Value

All parameters (arguments) in the Python language are passed by reference.

#!/usr/bin/python3

# Function definition is here

def changeme( mylist ):

"This changes a passed list into this function"

print("Inside the function mylist: ", mylist)

mylist[2]=50

print("Inside the function mylist: ", mylist)

return

# Now you can call changeme function

mylist = [10,20,30]

changeme( mylist )

print("Outside the function: ", mylist)

Output:

Inside the function mylist: [10, 20, 30]

Inside the function mylist: [10, 20, 50]

Outside the function: [10, 20, 50]

#!/usr/bin/python3

# Function definition is here

def changeme(mylist):

"This changes a passed list into this function"

mylist = [1,2,3,4] # This would assi new reference in mylist

print ("Inside the function: ", mylist)

return

# Now you can call changeme function

mylist = [10, 20, 30]

changeme(mylist)

print("Outside the function: ", mylist)

Output:

Inside the function: [1, 2, 3, 4]

Outside the function: [10, 20, 30]

# Return Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller.

A return statement with no arguments is the same as return None.

# Function Arguments

1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

## Required Arguments

Arguments passed to a function in correct positional order.

The number of arguments in the function call should match exactly with the function definition.

#!/usr/bin/python3

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print (str)

return

# Now you can call printme function

printme()

Error:

TypeError: printme() missing 1 required positional argument: 'str'

## Keyword Arguments

When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

The order of parameters does not matter.

#!/usr/bin/python3

#Function definition is here

def printinfo( name, age ):

"This prints a passed info into this function"

print("Name: ", name)

print("Age: ", age)

return

# Now you can call printinfo function

printinfo(age = 50, name = "miki")

Output:

Name: miki

Age: 50

## Default Arguments

An argument that assumes a default value if a value is not provided in the function call for that argument.

#!/usr/bin/python3

# Function definition is here

def printinfo( name, age = 35 ):

"This prints a passed info into this function"

print ("Name: ", name)

print ("Age ", age)

return

# Now you can call printinfo function

printinfo(age = 50, name = "miki")

printinfo(name = "miki")

Output:

Name: miki

Age 50

Name: miki

Age 35

## Variable-length Arguments

A function for more arguments than you specified while defining the function.

These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.

An asterisk (\*) is placed before the variable name that holds the values of all non-keyword variable arguments.

This tuple remains empty if no additional arguments are specified during the function call.

**Syntax**

def functionname([formal\_args,] \*var\_args\_tuple ):

"function\_docstring"

function\_suite

return [expression]

#!/usr/bin/python3

# Function definition is here

def printinfo( arg1, \*vartuple ):

"This prints a variable passed arguments"

print ("Output is: ")

print (arg1)

for var in vartuple:

print (var)

return

# Now you can call printinfo function

printinfo( 10 )

printinfo( 70, 60, 50 )

Output:

Output is:

10

Output is:

70

60

50

# Anonymous Functions

Anonymous functions are not declared in the standard manner by using the def keyword.

You can use the lambda keyword to create small anonymous functions.

1. Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
2. An anonymous function cannot be a direct call to print because lambda requires an expression.
3. Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
4. Although it appears that lambdas are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is to stack allocation by passing function, during invocation for performance reasons.

## Syntax

lambda [arg1 [,arg2,.....argn]]:expression

#!/usr/bin/python3

#Function definition is here

sum = lambda arg1, arg2: arg1 + arg2

#Now you can call sum as a function

print ("Value of total : ", sum( 10, 20))

print ("Value of total : ", sum( 20, 45))

Output:

Value of total : 30

Value of total : 65

# Refactoring

Often, you’ll recognize that you could improve the code by breaking it up into a series of functions that have specific jobs. This process is called refactoring. Refactoring makes your code cleaner, easier to understand, and easier to extend.

Example:

import json

def greet\_user():

"""Greet the user by name."""

filename = 'username.json'

try:

with open(filename) as f\_obj:

username = json.load(f\_obj)

except FileNotFoundError:

username = input("What is your name? ")

with open(filename, 'w') as f\_obj:

json.dump(username, f\_obj)

print("We'll remember you when you come back, " + username + "!")

else:

print("Welcome back, " + username + "!")

greet\_user()

Refactoring above code

import json

def get\_stored\_username():

"""Get stored username if available."""

filename = 'username.json'

try:

with open(filename) as f\_obj:

username = json.load(f\_obj)

except FileNotFoundError:

return None

else:

return username

def greet\_user():

"""Greet the user by name."""

username = get\_stored\_username()

if username:

print("Welcome back, " + username + "!")

else:

username = input("What is your name? ")

filename = 'username.json'

with open(filename, 'w') as f\_obj:

json.dump(username, f\_obj)

print("We'll remember you when you come back, " + username + "!")

greet\_user()

# END