GLS UNIVERSITY

210301601 INTRODUCTION TO PYTHON

UNIT– II

Python Programming Constructs and Functions

- Prof. Poonam Dang
- Prof. Neha Samsir

INDEX

Flow of Control

- Decision making is required when we want to execute a code only if a certain condition is satisfied.
- Decision making statements in programming languages decides the direction of flow of program execution.

- **Decision making statements** available in python are:
  - If statement
  - if..else statements
  - Nested if statements
  - If-elif ladder

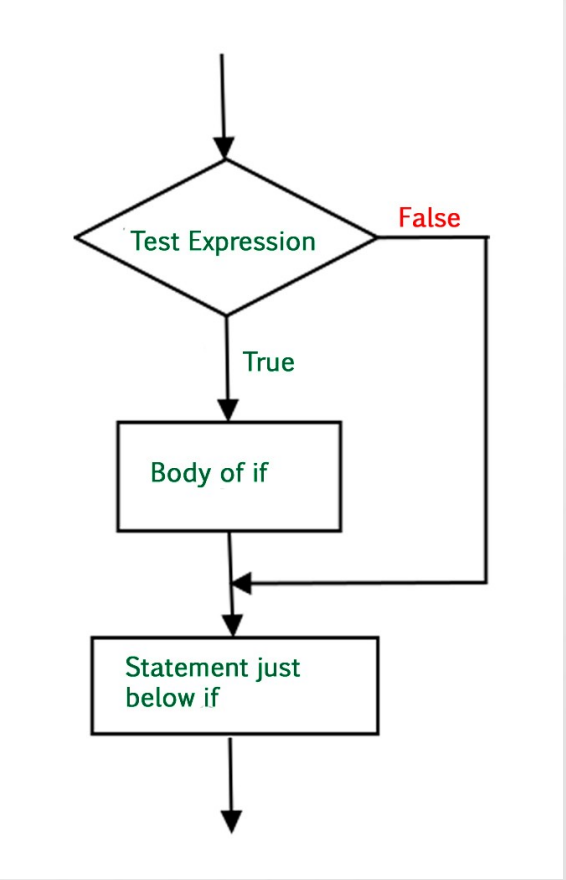
If statement

- if statement is the most simple decision making statement.
- It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

```
if condition:
   # Statements to execute if
   # condition is true
```

```
i = 10
if (i > 15):
   print ("10 is less than 15")
print ("I am Not in if")
```



Indentation

- Python relies on indentation, using whitespace, to define scope in the code.
- Other programming languages often use curly-brackets for this purpose.

Example

```
#If statement, without indentation (will raise an error):
a = 33
b = 200
if b > a:
print("b is greater than a") # you will get an error
```

Indentation

demo_if_error.py:

```
a = 33
b = 200

if b > a:
print("b is greater than a")
```

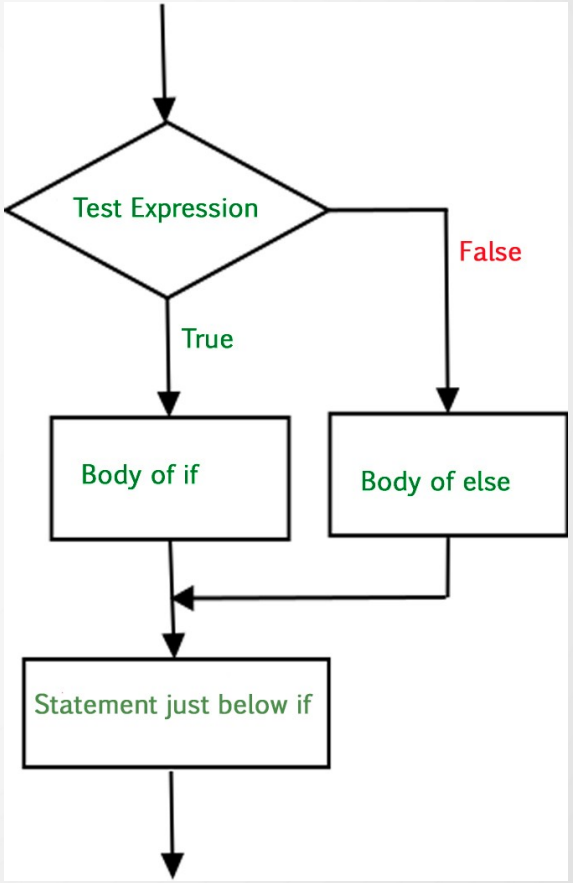
```
C:\Users\My Name>python demo_if_error.py
  File "demo_if_error.py", line 4
    print("b is greater than a")
        ^
IndentationError: expected an indented block
```

If-else statement

- We can use the else statement with if statement to execute a block of code when the condition is false.

Syntax of if else statement:

```
if (condition):
    # Executes this block if
    # condition is true
else:
    # Executes this block if
    # condition is false
```



If-else statement

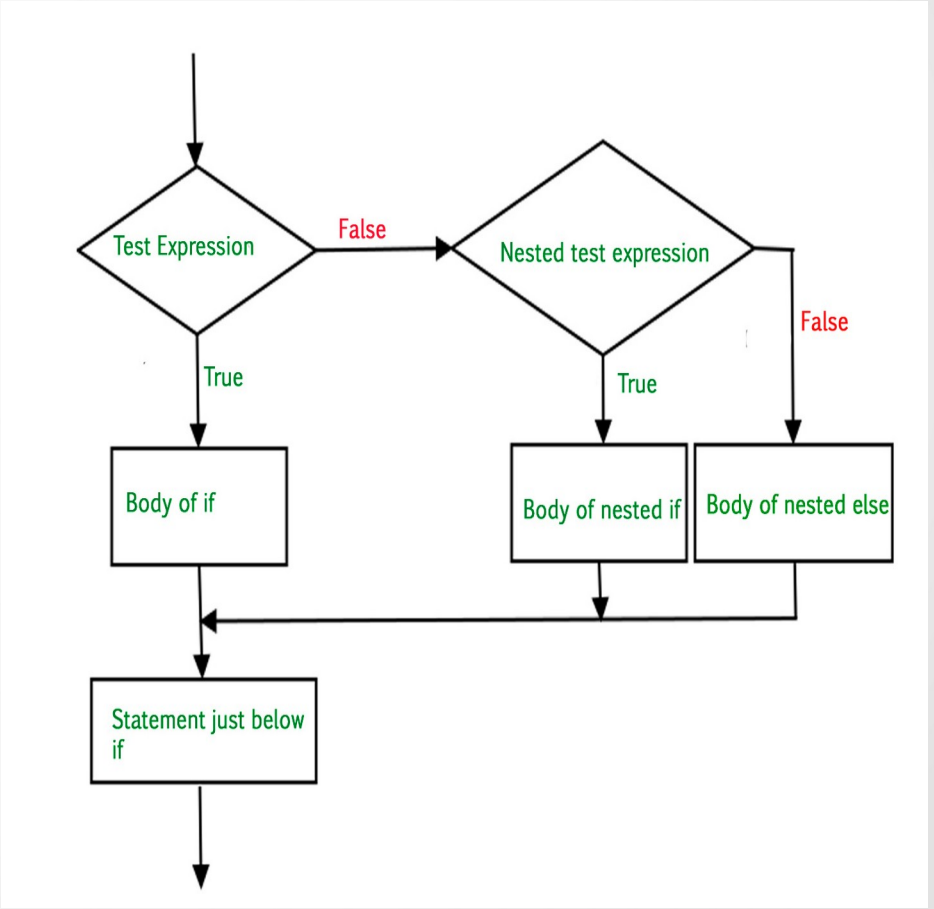
```
i = 20;
if (i < 15):
    print ("i is smaller than 15")
    print ("i'm in if Block")
else:
    print ("i is greater than 15")
    print ("i'm in else Block")
print ("i'm not in if and not in else Block")
```

Nested if statement

- A nested if is an if statement that is the target of another if statement.
- Nested if statements means an if statement inside another if statement.

Syntax:

```
if (condition1):
   # Executes when condition1 is true
   if (condition2):
      # Executes when condition2 is true
   # if Block is end here
# if Block is end here
```



If-else statement

```
i = 10
if (i == 10):
    #  First if statement
    if (i < 15):
        print ("i is smaller than 15")
    # Nested - if statement
    # Will only be executed if statement above
    # it is true
    if (i < 12):
        print ("i is smaller than 12 too")
    else:
        print ("i is greater than 15")
```

Output:

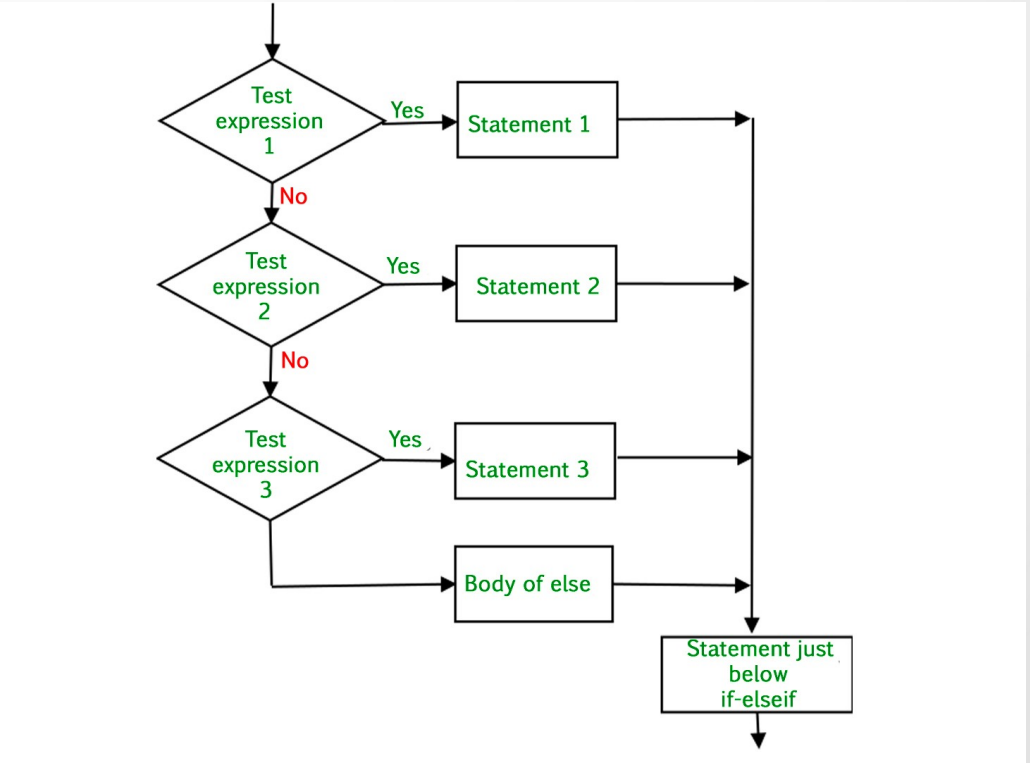
```
i is smaller than 15
i is smaller than 12 too
```

If-elif-else ladder

- The if statements are executed from the top down.
- As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed.
- If none of the conditions is true, then the final else statement will be executed.

Syntax:-

```
if (condition):
    statement
elif (condition):
    statement
.
.
else:
    statement
```



If-elif-else ladder

```
i = 20
if (i == 10):
    print ("i is 10")
elif (i == 15):
    print ("i is 15")
elif (i == 20):
    print ("i is 20")
else:
    print ("i is not present")
```

Output:
i is 20

Loops

- A loop statement allows us to execute a statement or group of statements multiple times.

Python programming language provides following **types of loops**:

- While loop
- For loop
- Nested loop
- For else

While loop

A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.
We generally use this loop when we don't know beforehand, the number of times to iterate.

Syntax:

```
while <test_expression>:
    Body of while
```

- In while loop, test expression is checked first.
- The body of the loop is entered only if the test_expression evaluates to True.
- After one iteration, the test expression is checked again.
- This process continues until the test_expression evaluates to False.

While loop

```
# while loop
count = 0
while (count < 3):
    count = count + 1
    print("Hello TY")
```

Output:
Hello TY
Hello TY
Hello TY

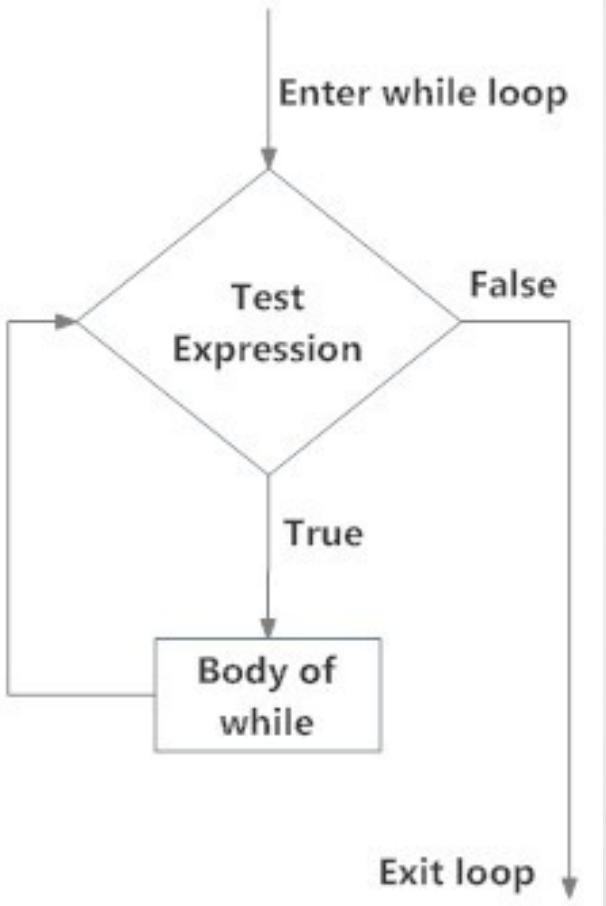


Fig: operation of while loop

While and else statement

- There is a structural similarity between while and else statement.
- Both have a block of statement(s) which is only executed when the condition is true.
- The difference is that block belongs to if statement executes once whereas block belongs to while statement executes repeatedly.

An optional else clause with while statement.

The syntax is:

```
while (expression) :
    statement_1
    statement_2
    ......
else :
    statement_3
    statement_4
```

While and else statement

The while loop repeatedly tests the expression (condition) and, if it is true, executes the first block of program statements.
The else clause is only executed when the condition is false it may be the first time it is tested and will not execute if the loop breaks, or if an exception is raised.
If a break statement executes in first program block and terminates the loop then the else clause does not execute.

The else block just after for/while is executed only when the loop is NOT terminated by a break statement.

While and else statement

```
counter = 0

while counter < 3:
    print("Inside loop")
    counter = counter + 1
else:
    print("Inside else")
```

Output:
Inside loop
Inside loop
Inside loop
Inside else

For Loop

In Python for loop is used to iterate over the items of any sequence including the Python list, string, tuple etc.
The for loop is also used to access elements from a container (for example list, string, tuple) using built-in function range().
One of the most common looping statements is the for loop.

The syntax of this loop is:

```
for <variable> in <sequence>:
        statements
```

- **for** – the keyword that indicates that the for statement begins.
- **variable** – specifies the name of the variable that will hold a single element of a sequence.
- **in** – indicates that the sequence comes next.
- **sequence** – the sequence that will be stepped through.
- **statements** – the statements that will be executed in each iteration.

For Loop

```
# Program to find the sum of all
numbers stored in a list

# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum
sum = 0

# iterate over the list
for val in numbers:
        sum = sum+val

# Output: The sum is 48
print("The sum is", sum)
```

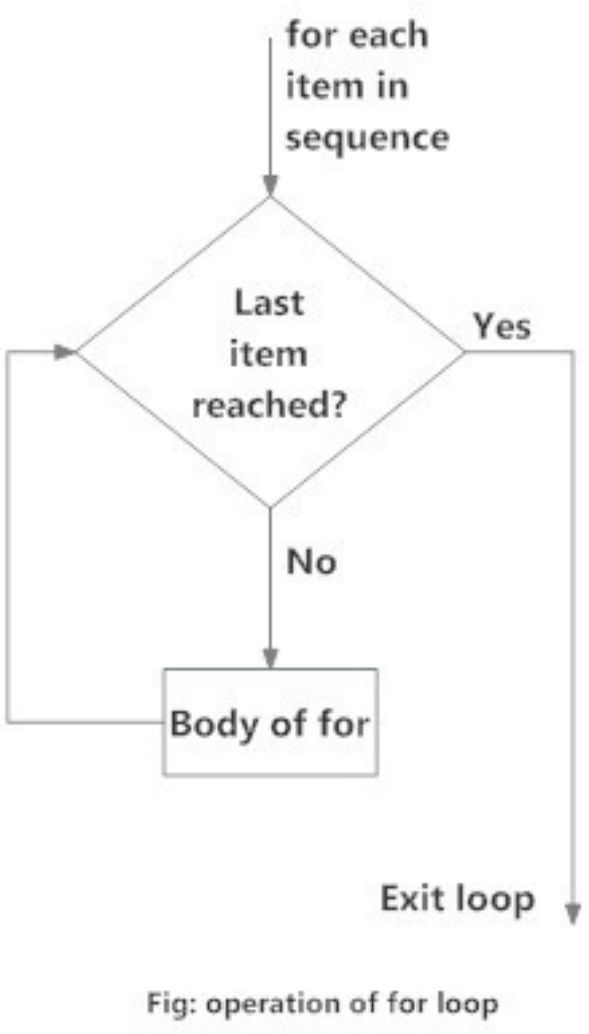


Fig: operation of for loop

For Else Loop

- In most of the programming languages (C/C++, Java, etc), the use of else statement has been restricted with the if conditional statements.
- But Python also allows us to use the else condition with for loops.
- The else block just after for/while is executed only when the loop is NOT terminated by a break statement.

```
for i in range(1, 4):
    print(i)
else:  # Executed because no break in for
    print("No Break")
```

Output:

1

2

3

No Break

For Else Loop

- for i in range(1, 4):
  print(i)
  break

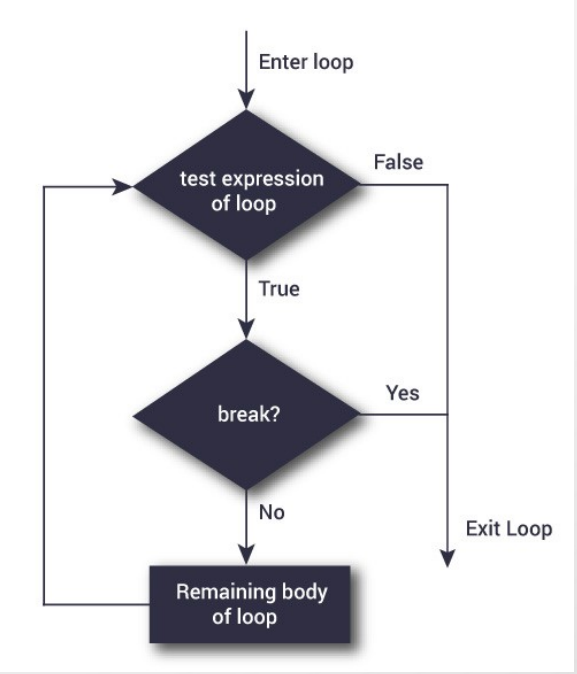
else: # Not executed as there is a break
  print("No Break")

Output:
1

Break, Continue and Pass Statement

In Python, break and continue statements can alter the flow of a normal loop.
Loops iterate over a block of code until test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without cheking test expression.

Break: The break statement in Python terminates the current loop and resumes execution at the next statement.



Break, Continue and Pass Statement

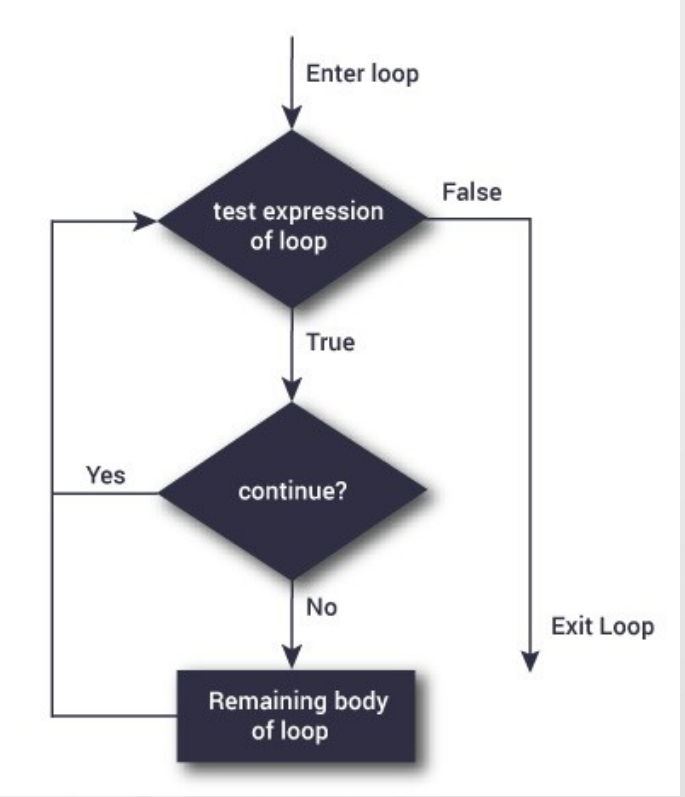
```
# Use of break statement inside loop

for val in "string":
    if val == "i":
        break
    print(val)

print("The end")
```

Output:

```
s
t
r
The end
```

Break, Continue and Pass Statement

Continue: The continue statement is used to skip the rest of the code inside a loop for the current iteration only.
Loop does not terminate but continues on with the next iteration.



Break, Continue and Pass Statement

```
# Program to show the use of continue statement inside loops
for val in "string":
    if val == "i":
        continue
    print(val)

print("The end")
```

Output:

```
s
t
r
n
g
The end
```

Break, Continue and Pass Statement

- **Pass:**
- In Python, pass is a null statement.
  The difference between a comment and pass statement is that, while the interpreter ignores a comment entirely, pass is not ignored.
  Nothing happens when pass is executed.
  It results into no operation (NOP).

```
# pass is just a placeholder for functionality to be added later.
sequence = {'p', 'a', 's', 's'}
for val in sequence:
    Pass
```

Output:

Break, Continue and Pass Statement

```
for letter in 'Python':
   if letter == 'h':
      pass
      print 'This is pass block'
   print 'Current Letter :', letter

print "Good bye!"
```

Output:

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye!
```

Functions

A function is a block of program statements which can be used repetitively in a program.
Functions help break our program into smaller and modular chunks.
As our program grows larger and larger, functions make it more organized and manageable.

Syntax:

```
def function_name(parameters):
    """docstring"""
    statement(s)
```

Example:

```
def my_function():
    print("Hello From My Function!")
```

Functions

- Keyword def marks the start of function header.
- A function name to uniquely identify it.
- 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 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 same indentation level (usually 4 spaces).
- An optional return statement to return a value from the function.

Functions

- **Function Call**

- To define a function, we can call it from another function.
- To call a function, simply type the function name with appropriate parameters.

- **Syntax:**
  - function_name(arg1, arg2)

- **Docstring:**
- The first string after the function header is called the docstring and is short for documentation string.
- It is used to explain in brief, what a function does.

Functions

Example:

```
 def printme( str ):
   "This prints a passed string into this function"
   print (str)
   return

# Now you can call printme function
printme("This is first call to the user defined function!")
printme("Again second call to the same function")
```

Output:

```
This is first call to the user defined function!
Again second call to the same function
```

Python Programming Constructs and Functions

- **Pass by reference vs value**

- All parameters (arguments) in the Python language are passed by reference.
- It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

Python Programming Constructs and Functions

Example:

```
# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   print ("Values inside the function before change: ", mylist)

   mylist[2]=50
   print ("Values inside the function after change: ", mylist)
   return

# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

Ouput:

```
Values inside the function before change:  [10, 20, 30]
Values inside the function after change:  [10, 20, 50]
Values outside the function:  [10, 20, 50]
```

Python Programming Constructs and Functions

Example:

```
# 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 ("Values inside the function: ", mylist)
   return

# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

Output:

```
Values inside the function:  [1, 2, 3, 4]
Values outside the function:  [10, 20, 30]
```

Function Arguments

We can call a function by using the following types of formal arguments −

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Function Arguments

Required Arguments:

- Required arguments are the arguments passed to a function in correct positional order.
- Here, the number of arguments in the function call should match exactly with the function definition.

Function Arguments

Example:

```
# 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()
```

When the above code is executed, it produces the following result −

```
Traceback (most recent call last):
   File "test.py", line 11, in <module>
      printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

Function Arguments

- **Keyword Arguments**

- Functions can also be called using keyword arguments.
- Arguments which are preceded with a variable name followed by a '=' sign (e.g. var_name='') are called keyword arguments.
- All the keyword arguments passed must match with one of the arguments accepted by the function.
- User can also change the order of appearance of the keyword.

Function Arguments

Example:

```
# 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( str = "My string")
```

Output:

```
My string
.
```

Function Arguments

Example:

```
# 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
```

Function Arguments

- **Default Argument Values**

- In function's parameters list we can specify a default value(s) for one or more arguments.
- A default value can be written in the format "argument1 = value", therefore we will have the option to declare or not declare a value for those arguments.

Function Arguments

Example:

```
# 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
```

Function Arguments

- **Variable-length arguments**

- Sometimes user need to process 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.
- An asterisk (*) is placed before the variable name that holds the values of all nonkeyword 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]

Function Arguments

```
# 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

printinfo( 10 )
printinfo( 70, 60, 50 )
```

Output:

```
Output is:
10
Output is:
70
60
50
```

Function Arguments

The Anonymous Functions

-
- These functions are called anonymous because they are not declared in the standard manner by using the def keyword.
- In Python, small anonymous (unnamed) functions can be created with lambda keyword.

Syntax:

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

- Lambda functions **can have any number of arguments but only one expression**. The expression is evaluated and returned.
- An anonymous function cannot be called directly to print because lambda requires an expression.
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

Function Arguments

Example:

```
# 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, 20 ))
```

Output:

```
Value of total :  30
Value of total :  40
```

Functions

- **return Statement:**

- The return statement is used to exit a function and go back to the place from where it was called.
- Return statement without an expression argument returns none.

- **Syntax:**
  - return [expression_list]

Functions

- # Function definition is here

```
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2
   print ("Inside the function : ", total)
   return total

# Now you can call sum function
total = sum( 10, 20 )
print ("Outside the function : ", total )
```

Output:

```
Inside the function :  30
Outside the function :  30
```

Scope of Variables

- Scope of a variable can be determined by the part in which variable is defined.
- Each variable cannot be accessed in each part of a program.
- There are two types of variables based on Scope:
  - **Local Variable.**
  - **Global Variable.**

- Variables declared inside a function body is known as **Local Variable**.
- These have a local access thus these variables cannot be accessed outside the function body in which they are declared.

Variable defined outside the function is called **Global Variable**.
Global variable is accessed all over program thus global variable have widest accessibility.

Scope of Variables

Example:

```
total = 0   # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2; # Here total is local variable.
   print ("Inside the function local total : ", total)
   return total

# Now you can call sum function
sum( 10, 20 )
print ("Outside the function global total : ", total )
```

Output:

```
Inside the function local total :  30
Outside the function global total :  0
```