LAB # 3 Control Flow Statements and Functions in Python

Controls the order in which the code is executed.

Python if ... else Statement

The **if…elif…else** statement is used in Python for decision making.

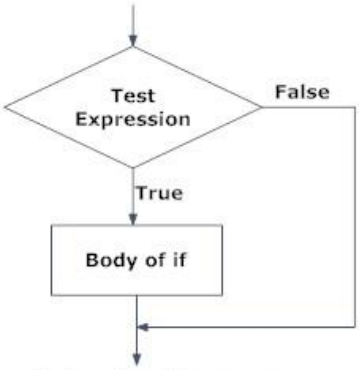


Fig: Operation of if statement

if statement syntax

if test expression:

statement(s)

The program evaluates the test expression and will execute statement(s) only if the text expression is True.

If the text expression is False, the statement(s) is not executed. Python interprets non-zero values as True. None and 0 are interpreted as False.

```
# if Example
num = 1
# try 0, -1 and None
if num>0:
    print("Number is positive")
print("This will print always")      #This print statement always print
```

```
In [1]: runfile('C:/Users/ALI/untitled0.py', wdir='C:/Users/ALI')
Number is positive
This will print always
```

if ... else Statement

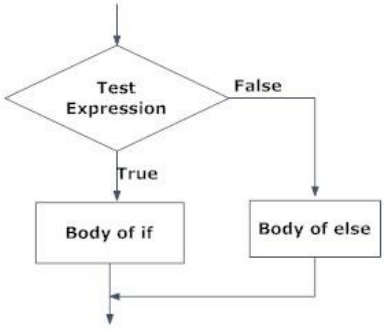


Fig: Operation of if...else statement

Syntax:

```
if test expression:
   Body of if
else:
   Body of else
```

```
# if else Exampple
num = -1
if num > 0:
    print("Positive number")
else:
    print("Negative Number")
print('statements out of the body')
```

```
Negative Number
statemnets out of the body
```

if...elif...else Statement

Syntax:

```
if test expression:
   Body of if
elif test expression:
   Body of elif
else:
   Body of else
```

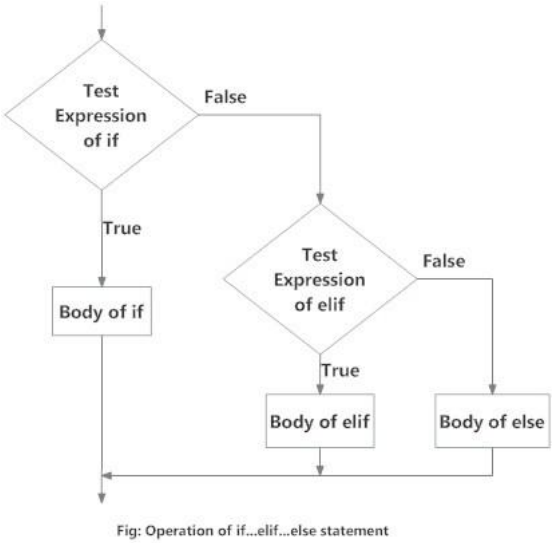


Fig: Operation of if...elif...else statement

```
# if elif statements
num = 0
if num > 0:
    print("Positive number")
elif num == 0:
    print("ZERO")
else:
    print("Negative Number")
```

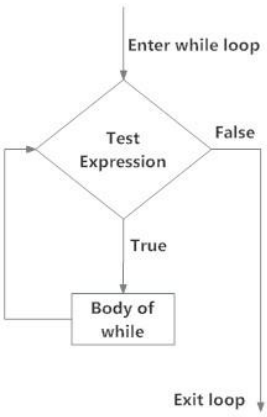
Nested if Statements

We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming

```
# nested example
num = -12
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative Number")
```

Python while Loop

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.



Syntax:

```
while test_expression:
   Body of while
```

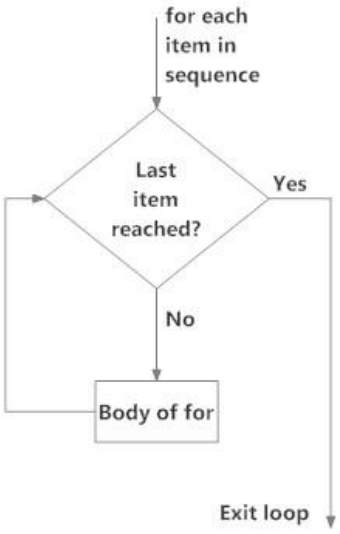
The body of the loop is entered only if the test_expression evaluates to True.

After one iteration, the test expression is checked again.

This process continues until the test_expression evaluates to False.

```
# while loop example (product of numbers in a list)
lst = [10, 20, 30, 40, 50]
product = 1
i=0
while i < len(lst):
    product=lst[i]*product # can use product *= lst[i]
    i=i+1 # can use += 1
print("Product is: {}".format(product))
```

Python for Loop



The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects.

Iterating over a sequence is called traversal.

Syntax:

for element in sequence :

Body of for

Here, element is the variable that takes the value of the item inside the sequence on each iteration.

Loop continues until we reach the last item in the sequence.

```
#for loop example
lst = [10, 20, 30, 40, 50,60]
product = 1
#iterating over the list
for num in lst:
    print(type(num))
    product *= num
print("Product is: {}".format(product))
```

range() function

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers). We can also define the start, stop and step size as range(start,stop,step size). step size defaults to 1 if not provided. This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

```
#for in range example
for i in range(0,10,1):
    print(i)
```

Python break and continue Statements

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 while loop without checking test expression. The break and continue statements are used in these cases.

```
# break example
numbers = [1, 2, 3, 4, 5,6]
for num in numbers:          #iterating over list
    if num == 4:
        break
    print(num)
else:
    print("in the else-block")
print("Outside of for loop")

# Continue example
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num % 2 == 0:
        continue
    print(num)
else:
    print("else-block")
```

Python Functions

Function is a group of related statements that perform a specific task. Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable. It avoids repetition and makes code reusable.

Syntax :

def function_name(parameters):

"""

Doc String

"""

Statement(s)

keyword "def" marks the start of function header

Parameters (arguments) through which we pass values to a function. These are optional

A colon(:) to mark the end of funciton header

Doc string describe what the function does. This is optional

"return" statement to return a value from the function. This is optional

Example:

```
def print_name(name):
    """
    This function prints the name
    """
    print("Hello " + str(name))
```

Function Call

Once we have defined a function, we can call it from anywhere

```
print_name('ALI')
```

Doc String

The first string after the function header is called the docstring and is short for documentation string. Although optional, documentation is a good programming practice, always document your code. Doc string will be written in triple quotes so that docstring can extend up to multiple lines

```
print(print_name.__doc__) # print doc string of the function
```

return Statement

The return statement is used to exit a function and go back to the place from where it was called.

Syntax:

return [expression]

-> return statement can contain an expression which gets evaluated and the value is returned.

-> if there is no expression in the statement or the return statement itself is not present inside a function,

then the function will return None Object

```
def get_sum(lst):
    """
    This function returns the sum of all the elements in a list
    """
    #initialize sum
    _sum = 0

    #iterating over the list
    for num in lst:
        _sum += num
    return _sum

s = get_sum([1, 2, 3, 4])
print(s)

#print doc string
print(get_sum.__doc__)
```

Scope and Life Time of Variables

-> Scope of a variable is the portion of a program where the variable is recognized

-> variables defined inside a function is not visible from outside. Hence, they have a local scope.

-> Lifetime of a variable is the period throughout which the variable exits in the memory.

-> The lifetime of variables inside a function is as long as the function executes.

-> Variables are destroyed once we return from the function.

Example:

```
global_var = "global variable"

def test_life_time():
    """
    This function test the life time of a variables
    """
    local_var = "local variable"
    print(local_var)       #print local variable local_var

    print(global_var)      #print global variable global_var


#calling function
test_life_time()

#print global variable global_var
print(global_var)

#print local variable local_var
print(local_var)
```

Python program to print Highest Common Factor (HCF) of two numbers

```
def computeHCF(a, b):
    """
    Computing HCF of two numbers
    """
    smaller = b if a > b else a  #consice way of writing if else
statement

    hcf = 1
    for i in range(1, smaller+1):
        if (a % i == 0) and (b % i == 0):
            hcf = i
    return hcf

num1 = 6
num2 = 36

print("H.C.F of {0} and {1} is: {2}".format(num1, num2,
computeHCF(num1, num2)))
```

Types Of Functions

Built-in Functions

User-defined Functions

Built-in Functions

1. abs()
2. all()
3. dir()
4. divmod exp: print(divmod(9, 2)) #print quotient and remainder as a tuple
5. enumerate()

User-defined Functions

Functions that we define ourselves to do certain specific task are referred as user-defined functions. If we use functions written by others in the form of library, it can be termed as library functions.

Advantages

User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.

If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.

Programmars working on large project can divide the workload by making different functions.

Example:

Python program to make a simple calculator that can add, subtract, multiply and division

```
def add(a, b):
    """
    This function adds two numbers
    """
    return a + b

def multiply(a, b):
    """
    This function multiply two numbers
    """
    return a * b

def subtract(a, b):
    """
    This function subtract two numbers
    """
    return a - b

def division(a, b):
```

```
    """
    This function divides two numbers
    """
    return a / b

print("Select Option")
print("1. Addition")
print ("2. Subtraction")
print ("3. Multiplication")
print ("4. Division")

#take input from user
choice = int(input("Enter choice 1/2/3/4"))

num1 = float(input("Enter first number:"))
num2 = float(input("Enter second number:"))
if choice == 1:
    print("Addition of {0} and {1} is {2}".format(num1, num2, add(num1,
num2)))
elif choice == 2:
    print("Subtraction of {0} and {1} is {2}".format(num1, num2,
subtract(num1, num2)))
elif choice == 3:
    print("Multiplication of {0} and {1} is {2}".format(num1, num2,
multiply(num1, num2)))
elif choice == 4:
    print("Division of {0} and {1} is {2}".format(num1, num2,
division(num1, num2)))
else:
    print("Invalid Choice")
```

Function Arguments

```
def greet(name, msg):
    """
    This function greets to person with the provided message
    """
    print("Hello {0} , {1}".format(name, msg))
```

#call the function with arguments

```
greet("ALI", "Good Morning")
```

#suppose if we pass one argument

```
greet("ALI") #will get an error
```

Different Forms of Arguments

1. Default Arguments

We can provide a default value to an argument by using the assignment operator (=).

```
def greet(name, msg="Good Morning"):
    """
    This function greets to person with the provided message
    if message is not provided, it defaults to "Good Morning"
    """
    print("Hello {0} , {1}".format(name, msg))

greet("ALI", "Good Night")
```

#with out msg argument

```
greet("Ali")
```

Once we have a default argument, all the arguments to its right must also have default values.

```
def greet(msg="Good Morning", name)
```

will get a SyntaxError : non-default argument follows default argument

2. Keyword Arguments

kwargs allows you to pass keyworded variable length of arguments to a function. You should use **kwargs if you want to handle named arguments in a function

Example:¶

```
def greet(**kwargs):
    """
    This function greets to person with the provided message
    """
    if kwargs:
        print("Hello {0} , {1}".format(kwargs['name'], kwargs['msg']))

greet(name="Ali", msg="Good Morning")
```

3. Arbitary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function.Python allows us to handle this kind of situation through function calls with arbitrary number of arguments.

Example:

```
def greet(*names):
    """
    This function greets all persons in the names tuple
    """
```

```
    print(names)

    for name in names:
        print("Hello,  {0} ".format(name))

greet("Ali", "Saad", "Adnan", "Zubair")
```

Recurison

We know that in Python, a function can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions.

Example:

```
#python program to print factorial of a number using recurion
def factorial(num):
    """
    This is a recursive function to find the factorial of a given
number
    """
    return 1 if num == 1 else (num * factorial(num-1))

num = 5
print ("Factorial of {0} is {1}".format(num, factorial(num)))
```

Factorial of 5 is 120

Advantages

1.Recursive functions make the code look clean and elegant.

2.A complex task can be broken down into simpler sub-problems using recursion.

3.Sequence generation is easier with recursion than using some nested iteration.

Disadvantages

1.Sometimes the logic behind recursion is hard to follow through.

2.Recursive calls are expensive (inefficient) as they take up a lot of memory and time.

3.Recursive functions are hard to debug.

Modules

Modules refer to a file containing Python statements and definitions.

A file containing Python code, for e.g.: abc.py, is called a module and its module name would be "abc".

We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.

We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

How to import a module?

We use the import keyword to do this.

```
import example #imported example module
```

Using the module name we can access the function using dot (.) operation.

```
example.add(10, 20)
```

Python has a lot of standard modules available.

https://docs.python.org/3/py-modindex.html

Examples:

```
import math
print(math.pi)

import datetime
datetime.datetime.now()
```

import with renaming

```
import math as m
print(m.pi)
```

from...import statement

We can import specific names form a module without importing the module as a whole.

```
from datetime import datetime
datetime.now()
```

import all names

```
from math import *
print("Value of PI is " + str(pi))
```

dir() built in function

We can use the dir() function to find out names that are defined inside a module.

```
Import math
dir(math)
```

Lab Tasks:

1. Write a Python program to find those numbers which are divisible by 7 and multiple of 5, between 1500 and 2700 (both included)
2. Write a Python program to convert temperatures to and from Celsius, Fahrenheit.
   [Formula: c/5 = f-32/9 [where c = temperature in Celsius and f = temperature in Fahrenheit]

Expected Output :

60°C is 140 in Fahrenheit

45°F is 7 in Celsius

3. Write a Python program to construct the following pattern, using a nested for loop.

```
*
* *
* * *
* * * *
* * * * *
* * * *
* * *
* *
*
```

4. Write a Python program to count the number of even and odd numbers from a series of numbers.

5. Write a Python program that prints each item and its corresponding type from the following list.

Sample List : datalist = [1452, 11.23, 1+2j, True, 'w3resource', (0, -1), [5, 12], {"class":'V', "section":'A'}]

6. Write a Python program to get the Fibonacci series between 0 to 50.

7. Write a Python program to check the validity of password input by users.
Validation :

a. At least 1 letter between [a-z] and 1 letter between [A-Z].

b. At least 1 number between [0-9].

c. At least 1 character from [$#@].

d. Minimum length 6 characters.

e. Maximum length 16 characters.

8. Write a Python program to check a string represent an integer or not.
Expected Output:

```
Input a string: Python
The string is not an integer.
```

9. Write a Python function to find the Max of three numbers.

10. Write a Python program to reverse a string.

Sample String : "1234abcd"

Expected Output : "dcba4321"

11. Write a Python function to check whether a number is in a given range.

12. Write a Python function that accepts a string and calculate the number of upper case letters and lower case letters.

Sample String : 'The quick Brow Fox'

Expected Output :

No. of Upper case characters : 3

No. of Lower case Characters : 12

13. Write a Python function to check whether a number is perfect or not.

According to Wikipedia : In number theory, a perfect number is a positive integer that is equal to the sum of its proper positive divisors, that is, the sum of its positive divisors excluding the number itself (also known as its aliquot sum). Equivalently, a perfect number is a number that is half the sum of all of its positive divisors (including itself).

Example : The first perfect number is 6, because 1, 2, and 3 are its proper positive divisors, and 1 + 2 + 3 = 6. Equivalently, the number 6 is equal to half the sum of all its positive divisors: (1 + 2 + 3 + 6) / 2 = 6. The next perfect number is 28 = 1 + 2 + 4 + 7 + 14. This is followed by the perfect numbers 496 and 8128.

14. Write a Python function that checks whether a passed string is palindrome or not.

Note: A palindrome is a word, phrase, or sequence that reads the same backward as forward, e.g., madam or nurses run.

15. Write a Python program to access a function inside a function.

16. Write a recursive function to calculate the sum of numbers from 0 to 10.