Chapter 2.

Python Operators and Control Flow Statements

10 Marks

Basic Operators :

Arithmetic Operators-

- The following operators can be applied to for arithmetic operations:

+	Addition	e.g. x+y
-	Subtraction	e.g. x-y
*	Multiplication	e.g. x*y
/	Division	e.g. x / y
//	Truncating Divison	e.g. x // y
**	Exponentiation / Power	e.g. (x^y)
%	Modulus	e.g. (x % y)
-	Unary Minus	e.g. –x
+	Unary Plus	e.g. x

- The ***truncating division operator*** (//, also known as ***floor division*) *truncates the result to*** an integer and works with ***both integers and floating-point numbers***.
- In **Python 2**, the true division ***operator (/) also truncates the result to an integer*** if the operands are integers.
- Therefore, 7/4 is 1, not 1.75. However, this behavior changes in **Python 3,** where ***division produces a floating-point result***.
- The ***modulus operator(%) returns the remainder*** of the division x // y.

- For example, **7 % 4 is 3**. For floating-point numbers, the modulus operator returns the floating-point remainder of x // y, which is x – (x // y) * y. For complex numbers, the modulo (%) and truncating division operators (//) are invalid.

Comparison/ Relational Operators-

- x < y Less than
- x > y Greater than
- x == y Equal to
- x != y Not equal to
- x >= y Greater than or equal to
- x <= y Less than or equal to
- Comparisons can be chained together, such as in

w < x < y < z

Such expressions are evaluated as

w < x and x < y and y < z

- Expressions such as **x < y > z** are legal but are likely to confuse anyone reading the code (it's important to note that no comparison is made between x and z in such an expression). ***Comparisons involving complex numbers are undefined and result in a TypeError***.
- ***Operations involving numbers are valid only if the operands are of the same type***.
- For built-in numbers, a coercion operation is performed to convert one of the types to the other, as follows:

1. If either operand is a complex number, the other operand is converted to a complex number.

2. If either operand is a floating-point number, the other is converted to a float.

3. Otherwise, both numbers must be integers and no conversion is performed.

Assignment Operators-

- Assignment operators are used in Python to ***assign values to variables***.
- **a = 5** is a simple assignment operator that assigns the value 5 on the right to the variable **a** on the left.
- There are various ***compound operators in Python*** like

a += 5 that adds to the variable and later assigns the same.

It is equivalent to a = a + 5.

Assignment operators in Python

Operator	Example	Equivatent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5

**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

Logical Operators-

- Logical operators are the and, or, not operators.

Logical operators in Python		
Operator	**Meaning**	**Example**
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Logical Operators in Python

```
x = True

y = False

# Output: x and y is False

print('x and y is', x and y)

# Output: x or y is True

print('x or y is', x or y)

# Output: not x is False

print('not x is', not x)
```

Bitwise Operators-

- ***Bitwise operators act on operands as if they were string of binary digits***. It operates bit by bit, hence the name.
- Besides the normal numeric operations (addition, subtraction, and so on), Python supports most of the numeric expressions available in the C language. This includes operators that treat integers as strings of binary bits. For instance, here it is at work performing bitwise shift and Boolean operations:

```
>>> x = 1 # 0001

>>> x << 2 # Shift left 2 bits: 0100

4

>>> x | 2 # Bitwise OR: 0011

3

>>> x & 1 # Bitwise AND: 0001
```

1

- In the first expression, a binary 1 (in base 2, 0001) is shifted left two slots to create a binary 4 (0100). The last two operations perform a binary OR (0001|0010 = 0011) and a binary AND (0001&0001 = 0001). Such bit-masking operations allow us to encode multiple flags and other values within a single integer.

In the table below**:** Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

Bitwise operators in Python

Operator	Meaning	Example
&	Bitwise AND	x& y = 0 (0000 0000)
|	Bitwise OR	x | y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x>> 2 = 2 (0000 0010)
<<	Bitwise left shift	x<< 2 = 40 (0010 1000)

Membership :

- **in** and **not in** are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

- In a dictionary we can only test for presence of key, not the value.

Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Example: Membership operators in Python

```
x = 'Hello world'

y = {1:'a',2:'b'}

# Output: True

print('H' in x)

# Output: True

print('hello' not in x)

# Output: True

print(1 in y)

# Output: False

print('a' in y)
```

- Here, 'H' is in x but 'hello' is not present in x (remember, Python is case sensitive). Similarly, 1 is key and 'a' is the value in dictionary y. Hence, 'a' in y returns False.

- When you create an instance of a class, the type of hat instance is the class itself. To test for membership in a class, use the built-in function **isinstance(obj,cname)**
- This function returns True if an object, obj, belongs to the class cname or any class derived from cname.

example:

```
class A(object): pass

class B(A): pass

class C(object): pass

a = A()          # Instance of 'A'

b = B()          # Instance of 'B'

c = C()          # Instance of 'C'

type(a)          # Returns the class object A

isinstance(a,A)  # Returns True

isinstance(b,A)  # Returns True, B derives from A

isinstance(b,C)  # Returns False, C not derived from A.
```

- Similarly, the built-in **function issubclass(A,B) returns True if the class A is a subclass of class B**.

example:

```
issubclass(B,A) # Returns True

issubclass(C,A) # Returns False
```

Identity Operators:-

- **is** and **is not** are the identity operators in Python. They are used to check ***if two values (or variables) are located on the same part of the memory***. Two variables that are equal does not imply that they are identical.

Identity operators in Python		
Operator	**Meaning**	**Example**
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

Example #4: Identity operators in Python

```
x1 = 5

y1 = 5

x2 = 'Hello'

y2 = 'Hello'

x3 = [1,2,3]

y3 = [1,2,3]

# Output: False

print(x1 is not y1)

# Output: True

print(x2 is y2)
```

 # Output: False

print(x3 is y3)

- Here, we see that x1 and y1 are integers of same values, so they are equal as well as identical. Same is the case with x2 and y2 (strings).
- But x3 and y3 are list. They are equal but not identical. It is because interpreter locates them separately in memory although they are equal.

Operator Precedence (Order of Evaluation (Highest to Lowest)) :

- (...), [...], {...} Tuple, list, and dictionary creation
- s[i], s[i:j] Indexing and slicing
- s.attr Attributes
- f(...) Function calls
- +x, -x, ~x Unary operators
- x ** y Power (right associative)
- x * y, x / y, x // y, x % y Multiplication, division, floor division, modulo
- x + y, x - y Addition, subtraction
- x << y, x >> y Bit-shifting
- x & y Bitwise and
- x ^ y Bitwise exclusive or
- x | y Bitwise or
- x < y, x <= y, x > y, x >= y, x == y, x != y Comparison, identity, and sequence membership tests

x is y, x is not y, x in s, x not in s,

- not x Logical negation
- x and y Logical and
- x or y Logical or
- lambda args: expr Anonymous function

Control Flow:-

- Python programs are structured as a sequence of statements. All language features, including variable assignment, function definitions, classes, and module imports, are statements that have equal status with all other statements.

Conditional Statements

- The **if, else**, and **elif** statements control conditional code execution.

If Statement:

Decision making is required when we want to execute a code only if a certain condition is satisfied.

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

Python if Statement Syntax

```
if test expression:
    statement(s)
```

Here, 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.

In Python, the body of the if statement is indicated by the indentation. Body starts with an indentation and the first unindented line marks the end.

Python interprets non-zero values as True. None and 0 are interpreted as False.

Python if Statement Flowchart

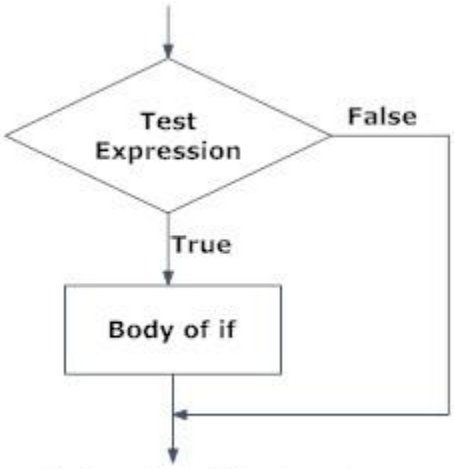


Fig: Operation of if statement

Example: Python if Statement

```
# If the number is positive, we print an appropriate message

num = 3

if num > 0:

    print(num, "is a positive number.")

print("This is always printed.")

num = -1
```

```
if num > 0:
    print(num, "is a positive number.")
print("This is also always printed.")
```

Output:

3 is a positive number.

This is always printed.

This is also always printed.

In the above example, `num > 0` is the test expression.
The body of `if` is executed only if this evaluates to `True`.
When variable `num` is equal to 3, test expression is true and body inside body of `if` is executed.
If variable `num` is equal to -1, test expression is false and body inside body of `if` is skipped.
The `print()` statement falls outside of the `if` block (unindented). Hence, it is executed regardless of the test expression.

if...else Statement

Syntax

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

The `if..else` statement evaluates `test expression` and will execute body of `if` only when test condition is `True`.

If the condition is `False`, body of `else` is executed. Indentation is used to separate the blocks.

Python if..else Flowchart

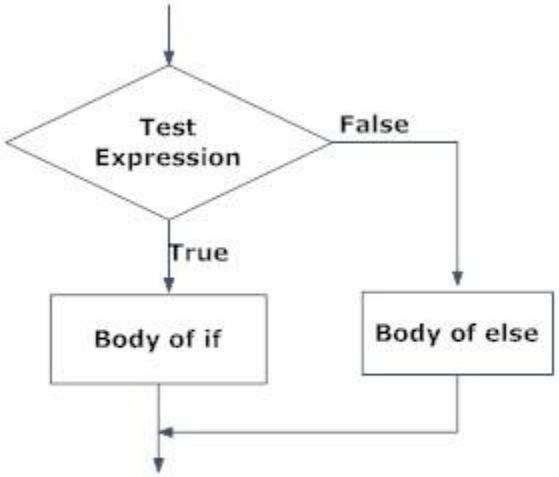


Fig: Operation of if...else statement

Example of if...else

```
# Program checks if the number is positive or negative

# And displays an appropriate message


num = 3


# Try these two variations as well.

# num = -5

# num = 0


if num >= 0:

    print("Positive or Zero")
```

```
else:
    print("Negative number")
```

Output:

Positive or Zero

In the above example, when `num` is equal to 3, the test expression is true and body of `if` is executed and `body` of else is skipped.
If `num` is equal to -5, the test expression is false and body of `else` is executed and body of `if` is skipped.
If `num` is equal to 0, the test expression is true and body of `if` is executed and `body` of else is skipped.

Python if...elif...else Statement

Syntax of if...elif...else

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

The `elif` is short for else if. It allows us to check for multiple expressions.
If the condition for `if` is `False`, it checks the condition of the next `elif` block and so on.
If all the conditions are `False`, body of else is executed.

Only one block among the several `if...elif...else` blocks is executed according to the condition.

The `if` block can have only one `else` block. But it can have multiple `elif` blocks.

Flowchart of if...elif...else

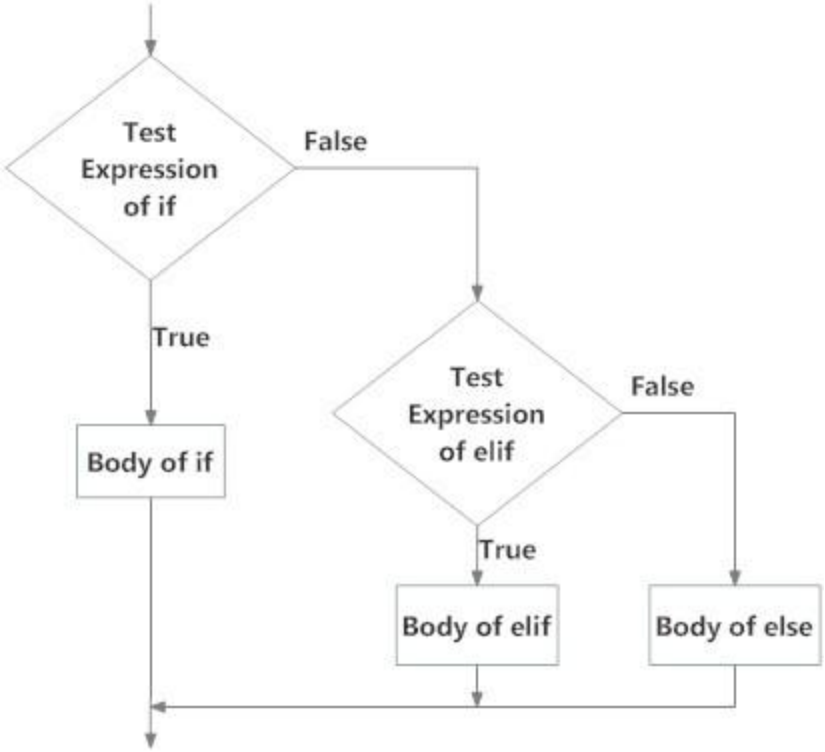


Fig: Operation of if...elif...else statement

Example of if...elif...else

```
# In this program,

# we check if the number is positive or

# negative or zero and

# display an appropriate message


num = 3.4
```

```
# Try these two variations as well:
# num = 0
# num = -4.5

if num > 0:
    print("Positive number")
elif num == 0:
    print("Zero")
else:
    print("Negative number")
```

Output:

```
Positive number
```

When variable `num` is positive, `Positive number` is printed.
If `num` is equal to 0, `Zero` is printed.
If `num` is negative, `Negative number` is printed

Python Nested if statements

We can have a `if...elif...else` statement inside another `if...elif...else` statement. This is called nesting in computer programming.
Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. This can get confusing, so must be avoided if we can.

Python Nested if Example

```
# In this program, we input a number
# check if the number is positive or
# negative or zero and display
# an appropriate message
# This time we use nested if

num = float(input("Enter a number: "))
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative number")
```

Output 1

```
Enter a number: 5
Positive number
```

Output 2

```
Enter a number: -1
Negative number
```

Output 3

```
Enter a number: 0
Zero
```

Looping in 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 of for Loop

```
for val in sequence:
        Body of for
```

Here, val 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. The body of for loop is separated from the rest of the code using indentation.

Flowchart of for Loop

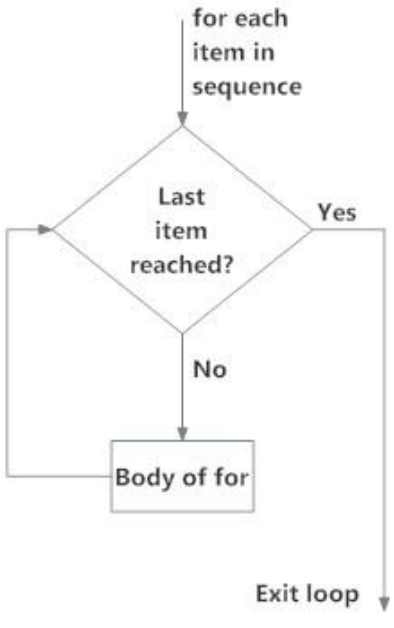


Fig: operation of for loop

Example: Python 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)
```

Output:

```
The sum is 48
```

The 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.

To force this function to output all the items, we can use the function list().
The following example will clarify this.

```
# Output: range(0, 10)
print(range(10))

# Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(list(range(10)))

# Output: [2, 3, 4, 5, 6, 7]
print(list(range(2, 8)))

# Output: [2, 5, 8, 11, 14, 17]
print(list(range(2, 20, 3)))
```

We can use the range() function in for loops to iterate through a sequence of numbers. It can be combined with the len() function to iterate though a sequence using indexing. Here is an example.

```
# Program to iterate through a list using indexing
genre = ['pop', 'rock', 'jazz']

# iterate over the list using index
for i in range(len(genre)):
    print("I like", genre[i])
```

Output:

```
I like pop
I like rock
I like jazz
```

for loop with else

A for loop can have an optional else block as well. The `else` part is executed if the items in the sequence used in for loop exhausts.
break statement can be used to stop a for loop. In such case, the else part is ignored.

Hence, a for loop's else part runs if no break occurs.

Here is an example to illustrate this.

```
digits = [0, 1, 5]

for i in digits:
    print(i)

else:
    print("No items left.")
```

Output:

```
0
1
5
No items left.
```

Here, the for loop prints items of the list until the loop exhausts. When the for loop exhausts, it executes the block of code in the `else` and prints.

Output:

```
No items left.
```

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.

We generally use this loop when we don't know beforehand, the number of times to iterate.

Syntax of while Loop in Python

```
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`. In Python, the body of the while loop is determined through indentation.

Body starts with indentation and the first unindented line marks the end.

Python interprets any non-zero value as `True`. `None` and `0` are interpreted as `False`.

Flowchart of while Loop

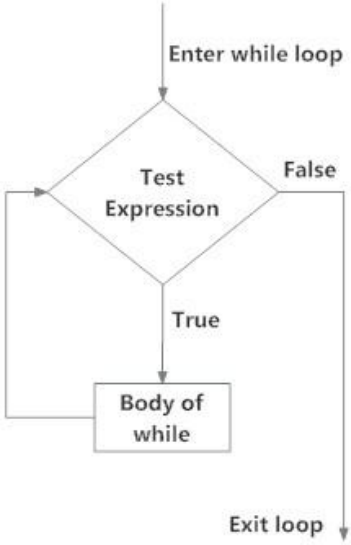


Fig: operation of while loop

Example: Python while Loop

```
# Program to add natural

# numbers upto

# sum = 1+2+3+...+n


# To take input from the user,

# n = int(input("Enter n: "))


n = 10


# initialize sum and counter

sum = 0

i = 1
```

```
while i <= n:
    sum = sum + i
    i = i+1    # update counter

# print the sum
print("The sum is", sum)
```

Output:

```
Enter n: 10
The sum is 55
```

In the above program, the test expression will be `True` as long as our counter variable `i` is less than or equal to `n` (10 in our program).
We need to increase the value of counter variable in the body of the loop. This is very important (and mostly forgotten). Failing to do so will result in an infinite loop (never ending loop).

while loop with else

Same as that of for loop, we can have an optional `else` block with while loop as well.
The `else` part is executed if the condition in the while loop evaluates to `False`.
The while loop can be terminated with a break statement. In such case,
the `else` part is ignored. Hence, a while loop's `else` part runs if no break occurs and the condition is false.

Example:

```
# Example to illustrate
# the use of else statement
# with the while loop
counter = 0
while counter < 3:
    print("Inside loop")
    counter = counter + 1
else:
    print("Inside else")
```

Output:

```
Inside loop
Inside loop
Inside loop
Inside else
```

Here, we use a counter variable to print the string `Inside loop` three times.
On the forth iteration, the condition in while becomes `False`. Hence, the `else` part is executed.

Loop Manipulation using continue, pass, break :

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 checking test expression.

Python break statement

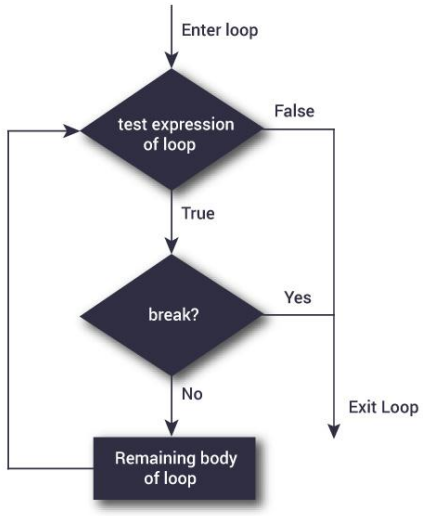
The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

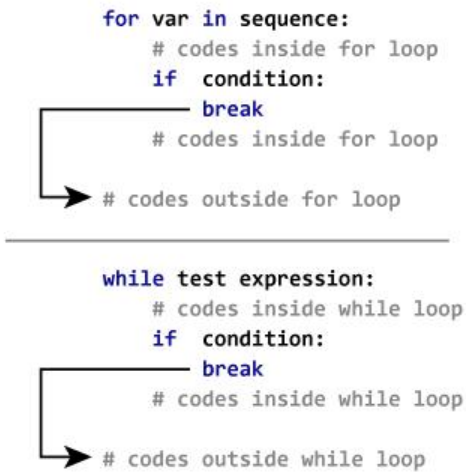
Syntax of break

```
break
```

Flowchart of break



The working of break statement in for loop and while loop is shown below.



Example:

```
# 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
```

In this program, we iterate through the `"string"` sequence. We check if the letter is `"i"`, upon which we break from the loop. Hence, we see in our output that all the letters up till `"i"` gets printed. After that, the loop terminates.

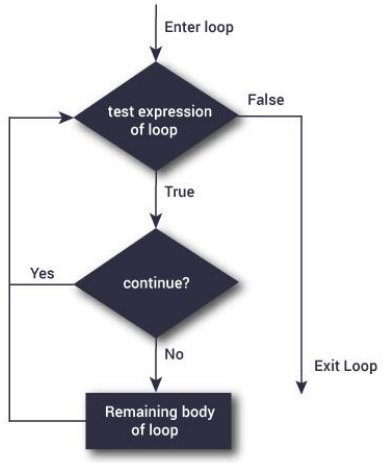
Python continue statement

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.

Syntax of Continue

```
continue
```

Flowchart of continue



The working of continue statement in for and while loop is shown below.

```
for var in sequence:
    # codes inside for loop
    if  condition:
        continue
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        continue
    # codes  inside while loop

# codes outside while loop
```

Example:

```
# 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
```

This program is same as the above example except the break statement has been replaced with continue.

We continue with the loop, if the string is `"i"`, not executing the rest of the block. Hence, we see in our output that all the letters except `"i"` gets printed.

pass statement:

In Python programming, `pass` is a null statement. The difference between a comment and `pass` statement in Python is that, while the interpreter ignores a comment entirely, `pass` is not ignored.

However, nothing happens when pass is executed. It results into no operation (NOP).

Syntax of pass

```
pass
```

Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. They cannot have an empty body. The interpreter would complain. So, we use the pass statement to construct a body that does nothing.

Example:

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

We can do the same thing in an empty function or class as well.

```
def function(args):
    pass

class example:
    pass
```

Programs based on Concept in chapter 2:

program to add two numbers

```
num1 = 1.5
num2 = 6.3

# Add two numbers
sum = float(num1) + float(num2)

# Display the sum
print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
```

Output:-

```
The sum of 1.5 and 6.3 is 7.8
```

Program to Add Two Numbers Provided by The User

```
# Store input numbers
num1 = input('Enter first number: ')
num2 = input('Enter second number: ')

# Add two numbers
sum = float(num1) + float(num2)

# Display the sum
print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
```

Output:-

```
Enter first number: 1.5
Enter second number: 6.3
The sum of 1.5 and 6.3 is 7.8
```

Python Program to calculate the square root

```
# To take the input from the user
num = float(input('Enter a number: '))

num_sqrt = num ** 0.5
print('The square root of %0.3f is %0.3f'%(num ,num_sqrt))
```

Input:

8

Output:

```
The square root of 8.000 is 2.828
```

Python Program to find the area of triangle

```
a = 5
b = 6
c = 7

# Uncomment below to take inputs from the user
# a = float(input('Enter first side: '))
# b = float(input('Enter second side: '))
# c = float(input('Enter third side: '))

# calculate the semi-perimeter
s = (a + b + c) / 2

# calculate the area
area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
print('The area of the triangle is %0.2f' %area)
```

Output:-

```
The area of the triangle is 14.70
```

Python program to swap two variables

```
x = 5
y = 10

# To take inputs from the user
#x = input('Enter value of x: ')
#y = input('Enter value of y: ')

# create a temporary variable and swap the values
temp = x
x = y
y = temp

print('The value of x after swapping: {}'.format(x))
print('The value of y after swapping: {}'.format(y))
```

Output:-

```
The value of x after swapping: 10
The value of y after swapping: 5
```

Program to check if a number is prime or not

```
num = 407

# To take input from the user
#num = int(input("Enter a number: "))

# prime numbers are greater than 1
if num > 1:
   # check for factors
   for i in range(2,num):
      if (num % i) == 0:
         print(num,"is not a prime number")
         print(i,"times",num//i,"is",num)
         break
   else:
      print(num,"is a prime number")

# if input number is less than
```

```
# or equal to 1, it is not prime
else:
   print(num,"is not a prime number")
```

Output:-

```
407 is not a prime number
11 times 37 is 407
```

Python program to check if year is a leap year or not

```
year = 2000

# To get year (integer input) from the user
# year = int(input("Enter a year: "))

if (year % 4) == 0:
   if (year % 100) == 0:
      if (year % 400) == 0:
         print("{0} is a leap year".format(year))
      else:
         print("{0} is not a leap year".format(year))
   else:
      print("{0} is a leap year".format(year))
else:
   print("{0} is not a leap year".format(year))
```

Output:-

```
2000 is a leap year
```

Python program to find the largest number among the three input numbers

```
num1 = 10
num2 = 14
num3 = 12

# uncomment following lines to take three numbers from user
#num1 = float(input("Enter first number: "))
#num2 = float(input("Enter second number: "))
#num3 = float(input("Enter third number: "))

if (num1 >= num2) and (num1 >= num3):
   largest = num1
elif (num2 >= num1) and (num2 >= num3):
   largest = num2
else:
   largest = num3

print("The largest number is", largest)
```

Output:-

```
The largest number is 14.0
```

Python program to find the factorial of a number provided by the user.

```
num = 7

# To take input from the user
#num = int(input("Enter a number: "))

factorial = 1

# check if the number is negative, positive or zero
if num < 0:
   print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
```

```
  print("The factorial of 0 is 1")
else:
  for i in range(1,num + 1):
     factorial = factorial*i
  print("The factorial of",num,"is",factorial)
```

Output:-

```
The factorial of 7 is 5040
```

Program for Multiplication table (from 1 to 10) in Python

```
num = 12

# To take input from the user
# num = int(input("Display multiplication table of? "))

# Iterate 10 times from i = 1 to 10
for i in range(1, 11):
  print(num, 'x', i, '=', num*i)
```

Output:-

```
12 x 1 = 12
12 x 2 = 24
12 x 3 = 36
12 x 4 = 48
12 x 5 = 60
12 x 6 = 72
12 x 7 = 84
12 x 8 = 96
12 x 9 = 108
12 x 10 = 120
```

Python Program to convert temperature in celsius to fahrenheit

```
celsius = 37.5

# calculate fahrenheit
fahrenheit = (celsius * 1.8) + 32
print('%0.1f degree Celsius is equal to %0.1f degree Fahrenheit' %(celsius,fahrenheit))
```

Output:-

```
37.5 degree Celsius is equal to 99.5 degree Fahrenheit
```

Python program to check if the input number is odd or even.

```
# A number is even if division by 2 gives a remainder of 0.
# If the remainder is 1, it is an odd number.

num = int(input("Enter a number: "))
if (num % 2) == 0:
   print("{0} is Even".format(num))
else:
   print("{0} is Odd".format(num))
```

Output:-

```
Enter a number: 43
43 is Odd
```

Python program to check if the number is an Armstrong number or not

```
# take input from the user
num = int(input("Enter a number: "))

# initialize sum
sum = 0
```

```
# find the sum of the cube of each digit
temp = num
while temp > 0:
   digit = temp % 10
   sum += digit ** 3
   temp //= 10

# display the result
if num == sum:
   print(num,"is an Armstrong number")
else:
   print(num,"is not an Armstrong number")
```

Output:-

```
Enter a number: 663
663 is not an Armstrong number
```

Output:-

```
Enter a number: 407
407 is an Armstrong number
```

Program for Factorial of a number using recursion

```
def recur_factorial(n):
   if n == 1:
      return n
   else:
      return n*recur_factorial(n-1)

num = 7

# check if the number is negative
if num < 0:
   print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
   print("The factorial of 0 is 1")
```

```
else:
  print("The factorial of", num, "is", recur_factorial(num))
```

Output:-

```
The factorial of 7 is 5040
```