PYTHON PROGRAMMING

UNIT -II

Control Statement: Definite iteration for Loop Formatting Text for output, Selection if and if else Statement, Conditional Iteration The While Loop

Strings and Text Files: Accessing Character and Substring in Strings, Data Encryption, Strings and Number Systems, String Methods Text Files.

Control Statements

Introduction: The control statements are the program statements that allow the computer to select a part of the code or repeat an action for specified number of times. These statements are categorized into categories: Selection statements, Iterative/Loop statements and Jump statements.

Definite Iteration: The for Loop

The repetition statements execute an action for specified number of times. These statements are also known as 'Loops'. Each repetition of an action is called 'Pass', or 'Iteration'. If a loop able to repeat an action for a predefined number of times then it is said to be definite Iteration. The Python's for loop supports the definite iteration.

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.

We use for loop when we know the number of times to iterate.

Syntax of for Loop

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

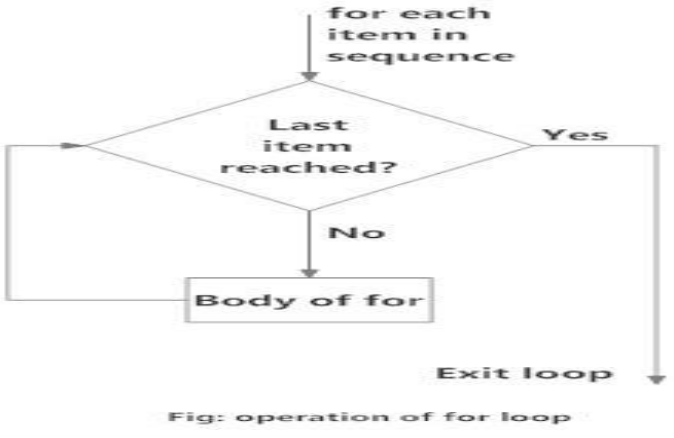


Fig: operation of for loop

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.

Example:

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

```
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
sum = 0
# iterate over the list
for val in numbers:
        sum = sum+val
print("The sum is", sum)
```

output:

The sum is 48

Function range()

In the above example, we have iterated over a list using for loop. However we can also use a range() function in for loop to iterate over numbers defined by range().

range(n): generates a set of whole numbers starting from 0 to (n-1).
For example: range(8) is equivalent to [0, 1, 2, 3, 4, 5, 6, 7]

range(start, stop): generates a set of whole numbers starting from start to stop-1.
For example:range(5, 9) is equivalent to [5, 6, 7, 8]

range(start, stop, step_size): The default step_size is 1 which is why when we didn't specify the step_size, the numbers generated are having difference of 1. However by specifying step_size we can generate numbers having the difference of step_size.
For example: range(1, 10, 2) is equivalent to [1, 3, 5, 7, 9]

ex: we are using range() function to calculate and display the sum of first 5 natural numbers.

Program to print the sum of first 5 natural numbers

```
sum = 0
# iterating over natural numbers using range()
for val in range(1, 6):
   sum = sum + val
# displaying sum of first 5 natural numbers
print(sum)
```

Output: 15

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.
The break keyword can be used to stop a for loop. In such cases, the else part is ignored.
Hence, a for loop's else part runs if no break occurs.
Example:

```
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 No items left.

Count-Controlled Loops

Loops that count through a range of numbers are also called count-controlled loops. The value of the count on each pass is used in computations. When Python executes the type of for loop just discussed above, it actually counts from 0 to the value of the integer expression minus 1 placed inside range () function. On each pass through the loop, the loop variable is bound to the current value of the count.

Fact.py	**Output**

num=int(input('Enter the number:'))
product=1
for i in range(num):
`    """i is loop variable,`
`    its value used inside body for`
`    computing product"""`
    product=product*(i+1)
print(f'{num}! is {product}')
	Enter the number:4 4!
	is 24

Traversing the Contents of a Data Sequence

The data sequence can be list, or tuple, string or set. The loop variable is bound to the next value in the sequence. The sequence of numbers generated using the range function can be converted into list or tuple. This tuple or list can be later used in the place of the sequence.

Add.py	**Output**

l=list(range(1,11)) print(l)
s=0
for x in l: #here x is loop variable and l is list
  s=s+x
print('Sum of Elements of list is',s)
	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
	Sum of Elements of list is 55

The for loop is used to traverse all the keys and values of a dictionary. A variable of the for loop is bound to each key in an unspecified order. It means it retrieves the key and its value in any order.

Example:

```
Grades={"Ramesh":"A","Viren":"B","Kumar":"C"}
for key in Grades:
    print(key,":",str(Grades[key]))
```

Output:

```
Ramesh : A
Viren : B
Kumar : C
```

If you use a dictionary in a for loop, it traverses the keys of the dictionary by default.

```
D = {'name': 'Bob',    'age': 25,    'job': 'Dev'}
for x in D:
   print(x)
```

output:

```
name age job
```

Loops That Count Down

All of our loops until now have counted up from a lower bound to an upper bound. Once in a while, a problem calls for counting in the opposite direction, from the upper bound down to the lower bound. When the step argument is a ***negative number***, the range function generates a sequence of numbers from the first argument down to the second argument plus 1.

Countdown.py	Output

l=list(range(10,0,-1))
print(l)
s=0
for x in l: #here x is loop variable and l is list
s=s+x
print('Sum of Elements of list is',s)
	[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
	Sum of Elements of list is 55

Python Loop Control Statements

Control statements in python are used to control the flow of execution of the program based on the specified conditions. Python supports 3 types of control statements such as,

1) Break

2) Continue

3) Pass

Python break statement

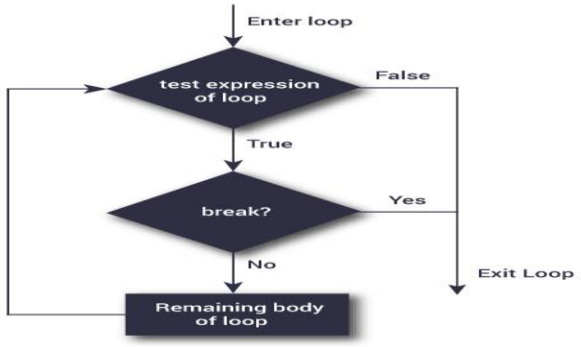
The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

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

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

# codes outside for loop
```

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

# codes outside while loop
```

Example: Python break

```
#number = 0
for number in range(10):
   if number == 5:
      break    # break here
   print('Number is ' + str(number))
print('Out of loop')
```

Output

```
Number is 0
Number is 1
Number is 2
Number is 3
Number is 4
Out of loop
```

The break statement causes a program to break out of a loop.

Python continue statement

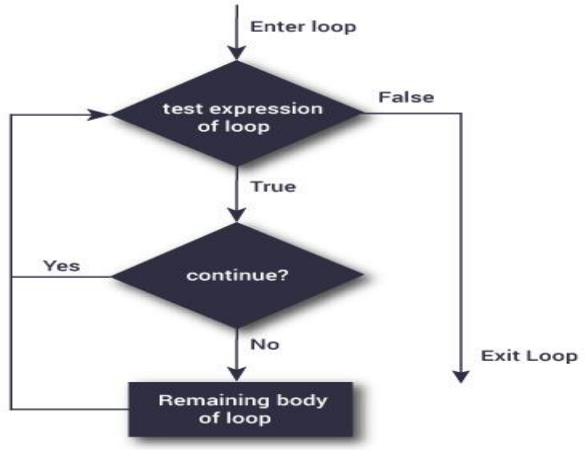
continue Statement in Python is used to skip all the remaining statements in the loop and move controls back to the top of the loop

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.

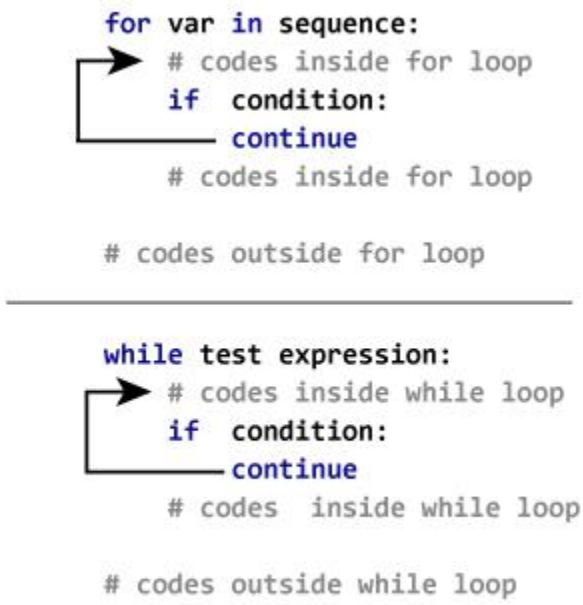
Syntax of Continue

continue

Flowchart of continue



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



Example: Python continue

```
number = 0
for number in range(10):
   if number == 5:
      continue    # continue here
   print('Number is ' + str(number))
print('Out of loop')
```

Output

```
Number is 0
Number is 1
Number is 2
Number is 3
Number is 4
Number is 6
Number is 7
Number is 8
Number is 9
Out of loop
```

Here, Number is 5 never occurs in the output, but the loop continues after that point to print lines for the numbers 6-10 before leaving the loop.

The continue statement causes a program to skip certain factors that come up within a loop, but then continue through the rest of the loop.

Python pass statement

Pass statement is python is a null operation, which is used when the statement is required syntactically

Python pass statement is used as a placeholder inside loops, functions, class, if-statement that is meant to be implemented later.

Syntax

pass

Python pass is a null statement. When the Python interpreter comes across the across pass statement, it does nothing and is ignored.

pass Statement in Python does nothing. A comment can also be added inside the body of the function or class, but the interpreter ignores the comment and will throw an error.

The pass statement can be used inside the body of a function or class body. During execution, the interpreter, when it comes across the pass statement, ignores and continues without giving any error.

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

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

Example:

```
number = 0
for number in range(10):
   if number == 5:
      pass    # pass here
   print('Number is ' + str(number))
print('Out of loop')
```

Output

```
Number is 0
Number is 1
Number is 2
Number is 3
Number is 4
Number is 5
Number is 6
Number is 7
Number is 8
Number is 9
Out of loop
```

By using the pass statement in this program, we notice that the program runs exactly as it would if there were no conditional statement in the program. The pass statement tells the program to disregard that condition and continue to run the program as usual

Formatting Text for output

There are several ways to present the output of a program, data can be printed in a human-readable form, or written to a file for future use or even in some other specified form.

. There are several ways to format output.

- To use formatted string literals, begin a string with f or F before the opening quotation mark or triple quotation mark.
- The str.format() method of strings help a user to get a fancier Output
- Users can do all the string handling by using string slicing and concatenation operations to create any layout that the user wants.

Formatting output using String modulo operator(%) :

The % operator can also be used for string formatting. It interprets the left argument much like a printf()-style format as in C language string to be applied to the right argument.

To this purpose, the modulo operator % is overloaded by the string class to perform string formatting. Therefore, it is often called a string modulo (or sometimes even called modulus) operator.

The string modulo operator (%) is still available in Python(3.x) and the user is using it widely. But nowadays the old style of formatting is removed from the language.

Python includes a general formatting mechanism that allows the programmer to specify ***field widths*** for different types of data. The simplest form of this operation is:

<format_string>%<datum>

This version contains a ***format string***, the ***format operator %,*** and a single ***data value*** to be formatted. The format string is represented as: ***%<field width>s***. The format string begins with % operator, and positive number for right-justification, and 's' is used for string.

Format Information	**character**
Integer	d
String	s
Float	f

%<fieldwidth>.<precision>f

The format information for a data value of type float has the form:

where **.<*precision*>** is optional.

Examples:

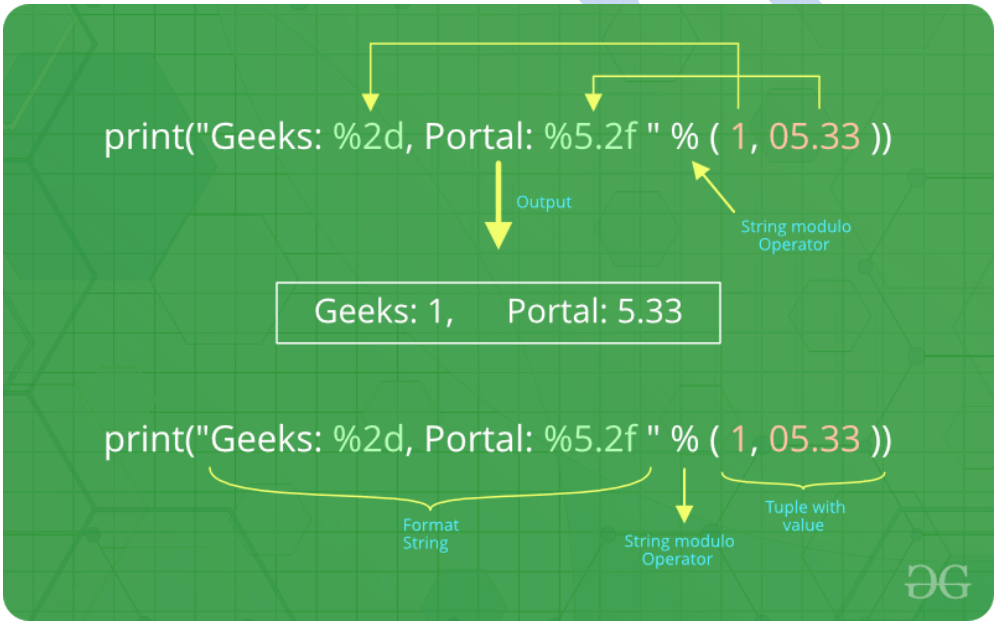
```
>>> "%8s" % "PYT" # field width is 8, right justification, datum is string PYT.
'     PYT'
>>> "%-8s" % "PYT" # field width is -8, left justification, datum is string PYT.
'PYT     '
>>> "%8d" % 1234          # field width is 8, right justification,
        datum is int 1234.  '    1234'
>>> "%-8d" % 1234 # field width is -8, left justification, datum is int
        1234. '1234    '
>>> "%6.2f" % 1234.678 # field width is 6, 2 decimal points, right
        justification, datum is int 1234.678
'1234.68'
```

Ex:# string modulo operator(%) to print

```
print("Geeks : %2d, Portal : %5.2f" % (1, 05.333))

# print integer value

print("Total students : %3d, Boys : %2d" % (240, 120))

# print octal value

print("%7.3o" % (25))

# print exponential value

print("%10.3E" % (356.08977))
```

Output :

```
Geeks :  1, Portal : 5.33
Total students : 240, Boys : 120
031
3.561E+02
```



There are two of those in our example: “%2d” and “%5.2f”. The general syntax for a format placeholder is:

%[flags][width][.precision]type

Formatting output using the format method :

The format() method was added in Python(2.6). The format method of strings requires more

manual effort. Users use {} to mark where a variable will be substituted and can provide detailed formatting directives, but the user also needs to provide the information to be formatted. This method lets us concatenate elements within an output through positional formatting. For Example –

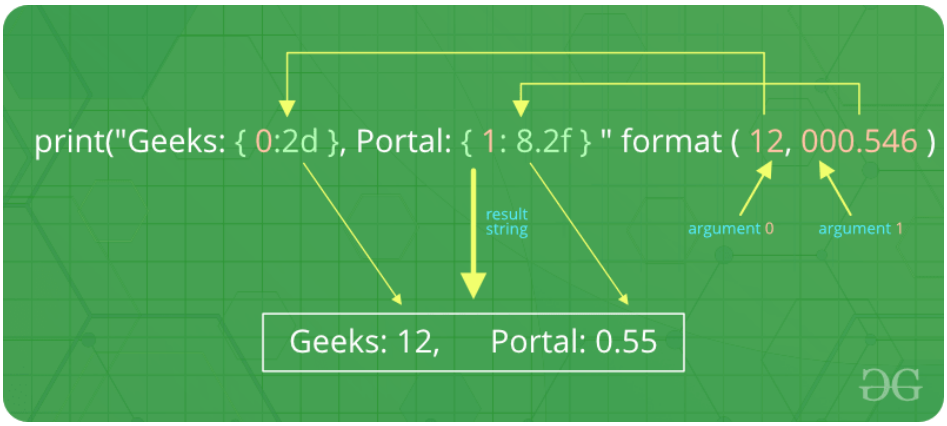
Code 1:

```
print('I love {} for "{}!"'.format('Geeks', 'Geeks'))

# using format() method and referring
# a position of the object
print('{0} and {1}'.format('Geeks', 'Portal'))
 print('{1} and {0}'.format('Geeks', 'Portal'))

# the above formatting can also be done by using f-Strings
# Although, this features work only with python 3.6 or above.
 print(f"I love {'Geeks'} for \"{'Geeks'}!\"")
 # using format() method and referring
# a position of the object
print(f"{'Geeks'} and {'Portal'}")
```

Output :

```
I love Geeks for "Geeks!"
Geeks and Portal
Portal and Geeks
```

The brackets and characters within them (called **format fields**) are replaced with the objects passed into the format() method. A number in the brackets can be used to refer to the position of the object passed into the format() method.



Selection if and if else Statement

Sometimes based on the condition some part of the code should be selected for execution, otherwise other part of the should be selected for execution. This kind of statements are called selection statements. If the condition is true, the computer executes the first ***alternative action*** and skips the ***second alternative***. If the condition is false, the computer skips the first alternative action and executes the second alternative.

In this section, we explore several types of selection statements, or control statements, that allow a computer to make choices.

The ***if-else*** statement is the most common type of selection statement. It is also called a ***two-way selection statement***, because it directs the computer to make a choice between two possible alternatives

In python, decision making is performed by the following statements.

Statement	Description
If Statement	The if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed.
If - else Statement	The if-else statement is similar to if statement except the fact that, it also provides the block of the code for the false case of the condition to be checked. If the condition provided in the if statement is false, then the else statement will be executed.

Indentation in Python

For the ease of programming and to achieve simplicity, python doesn't allow the use of parentheses for the block level code. In Python, indentation is used to declare a block. If two statements are at the same indentation level, then they are the part of the same block.

Generally, four spaces are given to indent the statements which are a typical amount of indentation in python.

Indentation is the most used part of the python language since it declares the block of code. All the statements of one block are intended at the same level indentation. .

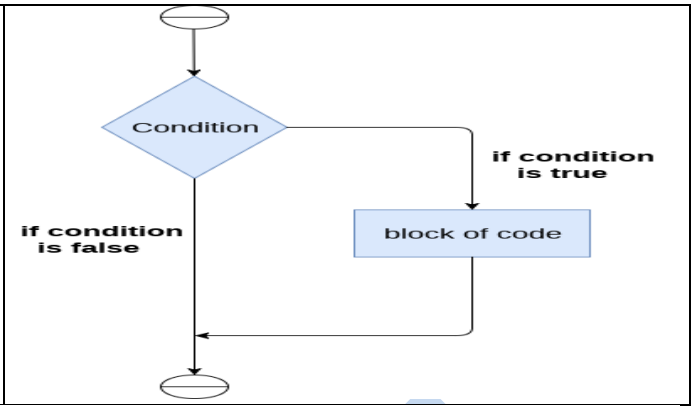
The if statement

The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block.

The condition of if statement can be any valid logical expression which can be either eva luated to true or false.

The syntax of the if-statement is given below.

```
if expression:
statement
```



Example 1

```
num = int(input("enter the number?"))
if num%2 == 0:
   print("Number is even")
```

Output:

```
enter the number?10
Number is even
```

Example 2 : Program to print the largest of the three numbers.

```
a = int(input("Enter a? "));
b = int(input("Enter b? "));
c = int(input("Enter c? "));
if a>b and a>c:
   print("a is largest");
if b>a and b>c:
   print("b is largest");
if c>a and c>b:
   print("c is largest");
```

Output:

```
Enter a? 100
Enter b? 120
Enter c? 130
c is largest
```

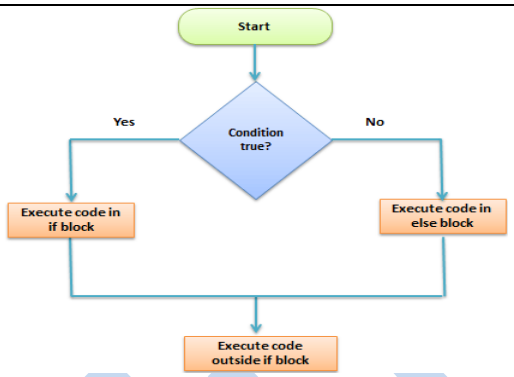
The if-else statement

The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.

If the condition is true, then the if-block is executed. Otherwise, the else-block is executed.

The syntax of the if-else statement is given below.

```
if condition:
    #block of statements
else:
    #another block of statements (else-block)
```



Example 1 : Program to check whether a person is eligible to vote or not.

```
e = int (input("Enter your age? "))
age>=18:
print("You are eligible to vote !!");
e:
print("Sorry! you have to wait !!");
```

Output:

```
Enter your age? 90
You are eligible to vote !!
```

Example 2: Program to check whether a number is even or not.

```
num = int(input("enter the number?"))
if num%2 == 0:
    print("Number is even...")
else:
    print("Number is odd...")
```

Output:

```
enter the number?10
Number is even
```

The elif statement or Multi-Way if Statements

The process of testing several conditions and responding accordingly can be described in code by a ***multi-way selection*** statement. This multi-way decision statement is preferred whenever we need to select one choice among multiple alternatives. The keyword '**elif**' is short for 'else if'.

The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them. We can have any number of elif statements in our program depending upon our need. However, using elif is optional.

The elif statement works like an if-else-if ladder statement in C. It must be succeeded by an if statement.

The syntax and flowchart of the elif statement is given below.

```
if[boolean expression]:
   [statements]
elif [boolean expresion]:
   [statements]
elif [boolean expresion]:
   [statements]
else:
   [statements]
```

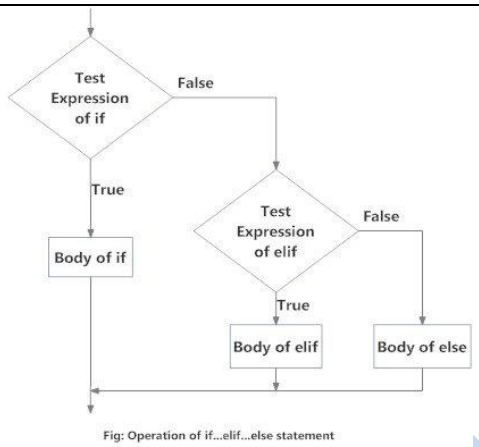


Fig: Operation of if...elif...else statement

Example 1

```
number = int(input("Enter the number?"))
if number==10:
   print("number is equals to 10")
elif number==50:
   print("number is equal to 50");
elif number==100:
   print("number is equal to 100");
else:
   print("number is not equal to 10, 50 or 100");
```

Output:

```
Enter the number?15
number is not equal to 10, 50 or 100
```

Example 2

```
marks = int(input("Enter the marks? "))
if marks > 85 and marks <= 100:
  print("Congrats ! you scored grade A ...")
elif marks > 60 and marks <= 85:
  print("You scored grade B + ...")
elif marks > 40 and marks <= 60:
  print("You scored grade B ...")
elif (marks > 30 and marks <= 40):
  print("You scored grade C ...")
else:
  print("Sorry you are fail ?")
```

Logical Operators and Compound Boolean Expressions

Often a course of action must be taken if either of two conditions is true. For example, valid inputs to a program often lie within a given range of values. Any input above this range should be rejected with an error message, and any input below this range should be dealt with in a similar fashion. The two conditions can be combined in a Boolean expression that uses the ***logical operator or***. The ***logical operator and*** also can be used to construct a different compound Boolean expression to express this logic.

Example Program using the logical or

```
marks=int(input('Enter your marks')) if
marks>100 or marks < 0:
    print('Marks must be within the range 0 to 100') else:
    print('Here you can write the code to process marks')
```

Example Program using the logical and

```
marks=int(input('Enter your marks')) if marks>0
and marks < 100:
    print('Here you can write the code to process marks')

else:
    print('Marks must be within the range 0 to 100')
```

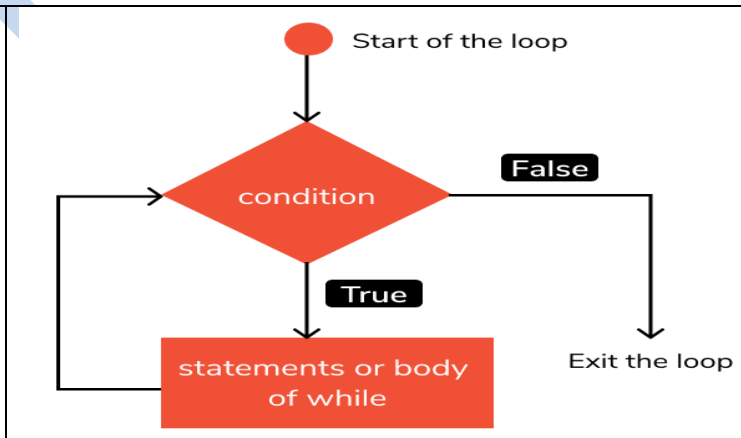
Conditional Iteration The While Loop

In python, while loop is used to execute a block of statements repeatedly until a given a condition is satisfied. And when the condition becomes false, the line immediately after the loop in program is executed.

We use while loop when we don't know the number of times to iterate.

Syntax of while Loop in Python

```
while expression:
    Body of while
```



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

Example1:

```
# Python program to illustrate
# while loop
count = 0
while (count < 3):
    count = count + 1
    print("Hello")
output:
Hello
Hello
Hello
```

Example2:

```
# Program to add 10 natural
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
```

Example	WHILE

i=1
while i < 4:
  print(i)
  i+=1
print('END')
1
2
3
END
	i=1
	while i < 4:
	  print(i)
	  i+=1
	  print('END')
	1
	END
	2
	END
	3
	END

Infinite while loop:

If the condition is given in the while loop never becomes false, then the while loop will never terminate, and it turns into the infinite while loop.

Any non-zero value in the while loop indicates an always-true condition, whereas zero indicates the always-false condition. This type of approach is useful if we want our program to run continuously in the loop without any disturbance.

Example1:

This will print the word ‘hello’ indefinitely because the condition will always be true.

```
while True:
  print("hello")
```

Example 2:

```
num = 1
while num<5:
  print(num)
```

This will print '1' indefinitely because inside loop we are not updating the value of num, so the value of num will always remain 1 and the condition num < 5 will always return true.

Using else Statement with While Loop

Python supports to have an else statement associated with a loop statement.

If the else statement is used with a while loop, the else statement is executed when the condition becomes false.

The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 5, otherwise else statement gets executed.

Example:

```
count = 0
while count < 5:
  print count, " is  less than 5"
  count = count + 1
else:
  print count, " is not less than 5"
```

result:

```
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5
```

Single statement while block: Just like the if block, if the while block consists of a single statement the we can declare the entire loop in a single line as shown below:

```
# Python program to illustrate

# Single statement while block

count = 0

while (count == 0): print("Hello Geek")
```

Note: It is suggested not to use this type of loops as it is a never ending infinite loop where the condition is always true and you have to forcefully terminate the compiler.

Write a python program to find sum of all the digits of a given number?

Total.py	Output
num=int(input('Enter number :')) rem=0 tot=0 while num>0: rem=num%10 tot=tot+rem num=num//10 print('The sum of all digits is:',tot)	Enter number :234 The sum of all digits is: 9

Write a python program to determine whether a given number is palindrome or not.

Palindrome.py	Output
num=int(input('Enter number :')) rem=0 x=num rev=0 while num>0: rem=num%10 rev=rev*10+rem num=num//10 print('The reversed number is:',rev) if x==rev: print(x,'is palindrome') else: print(x,'is not palindrome')	Enter number :121 The reversed number is: 121 121 is palindrome Enter number :123 The reversed number is: 321 123 is not palindrome

Jump statements with while loop

We can use the jump statements inside the while loop

we have three jump statements: break, continue and pass.

- ✓ When **break** is used inside the while loop it terminates the current loop and resumes execution at the next statement, just like the traditional break statement in C.
- ✓ The **continue** statement skips all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.
- ✓ The **pass** statement does nothing, but it simply transfers control to the next statement.

Else with while loopIt is executed when the condition becomes false (with while loop), but not when the loop is terminated by a break statement.

Syntax:

```
i = 1
while i < 6:
  print(i) i
  += 1
else:
  print ("i is no longer less than 6")
```

Strings and Text Files

Strings:

A string is a sequence of characters. You can access the characters one at a time with the **bracket operator:**

```
>>> Fruit = 'banana'
>>> letter = fruit[1]
```

The second statement selects character number 1 from fruit and assigns it to letter.

The expression in brackets is called an **index.** The index indicates which character in the sequence you want (hence the name). Index starts **with zero (0).**

You can use any expression, including variables and operators, as an index, but the value of the index has to be an **integer.** Otherwise you get:

```
>>> letter = fruit[1.5]
```

TypeError: string indices must be integers, not float

Text Files:

A file is a collection of records. A record is a group of related data items. These data items may contain information related to students, employees, customers, etc. In other words, a file is a collection of numbers, symbols and text and can be considered a stream of characters.

To permanently store the data created in a program, the user needs to save it in a file on the disk or some other device. The data stored in a file is used to retrieve the user's information either in part or whole.

Various operations carried out on a file are:

a) Creating a file
b) Opening a file

c) Reading from a file

d) Writing to file

e) Closing a file.

String:

String is a collection of alphabets, words or other characters. It is one of the primitive data structures and are the building blocks for data manipulation.

Python has a built-in string class named str. Python strings are "**immutable**" which means they cannot be changed after they are created. For string manipulation, we create new strings as we go to represent computed values because of their immutable property.

Strings can be created by enclosing characters inside a single quote or double-quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

Ex:

```
# all of the following are equivalent
my_string = 'Hello'
print(my_string)
my_string = "Hello"
print(my_string)
my_string = '''Hello'''
print(my_string)
# triple quotes string can extend multiple lines
my_string = """Hello, welcome to
        the world of Python"""
print(my_string)
```

output :

```
Hello
Hello
Hello
Hello, welcome to
```

the world of Python

Concatenation of Strings: Joining of two or more strings into a single one is called concatenation.

The **+ operator** does this in Python. Simply writing two string literals together also concatenates them.

The *** operator** can be used to repeat the string for a given number of times.

Ex:

```
# Python String Operations

str1 = 'Hello'

str2 ='World!'

print('str1 + str2 = ', str1 + str2) # using +

print('str1 * 3 =', str1 * 3) # using *
```

Output: str1 + str2 = HelloWorld!

str1 * 3 = HelloHelloHello

membership property in a string using in and not in:

```
sub_string1 = 'ice'

sub_string2 = 'glue'

string1 = 'ice cream'

if sub_string in string1:

    print("There is " + sub_string + " in " + string1)

if sub_string2 not in string1:

    print("Phew! No " + sub_string2 + " in " + string1)
```

There is ice in ice cream

Phew! No glue in ice cream

Iterating Through a string

We can iterate through a string using a for loop. Here is an example to count the number of 'l's in a string.

```
# Iterating through a string

count = 0

for letter in 'Hello World':

    if(letter == 'l'):

        count += 1

print(count,'letters found')
```

output:

3 letters found

subscript operator

The subscript operator is defined as square brackets []. It is used to access the elements of string, list tuple, and so on.

All the characters of a string can be accessed using the index operator ([ ]), also called subscript operator.

Indexing in Python means referring to an element of an iterable by its position within the iterable.

As a string is a sequence of characters, the characters in a string can be accessed one at a time through the index operator. The characters in a string are zero-based, i.e., the first character of the string is stored at the 0th position.

- Each character can be accessed using their index number.
- To access characters in a string we have two ways:
  - **Positive index number**
  - **Negative index number**

Positive indexing example in Python

In **Python Positive indexing**, we pass a positive index that we want to access in square brackets. The index number starts from **0** which denotes the **first character** of a string.

Negative indexing example in Python

In **negative indexing in Python**, we pass the negative index which we want to access in square brackets. Here, the index number starts from index number **-1** which denotes the **last character** of a string.

Example:

```
my_str = "Python Guides"
print(my_str[-1])
```

Indexing in python string

String indexing in python allows you to access individual characters from the string directly by using the index. Here, **"(str[3])"** is used to get **"c"**.

Example:

```
str = 'Welcome'
print(str[3])
```

Python String Slicing or Substring of a String

We can access individual characters using **indexing** and a range of characters using **slicing.** Index starts from 0. Trying to access a character out of index range will raise an **IndexError.** The index must be an integer. We can't use floats or other types, this will result into **TypeError.**

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on. We can access a range of items in a string by using the slicing operator :(colon).

If S is a string, the expression S [start : stop : step] returns the portion of the string from index start to index stop, at a step size step.

Syntax

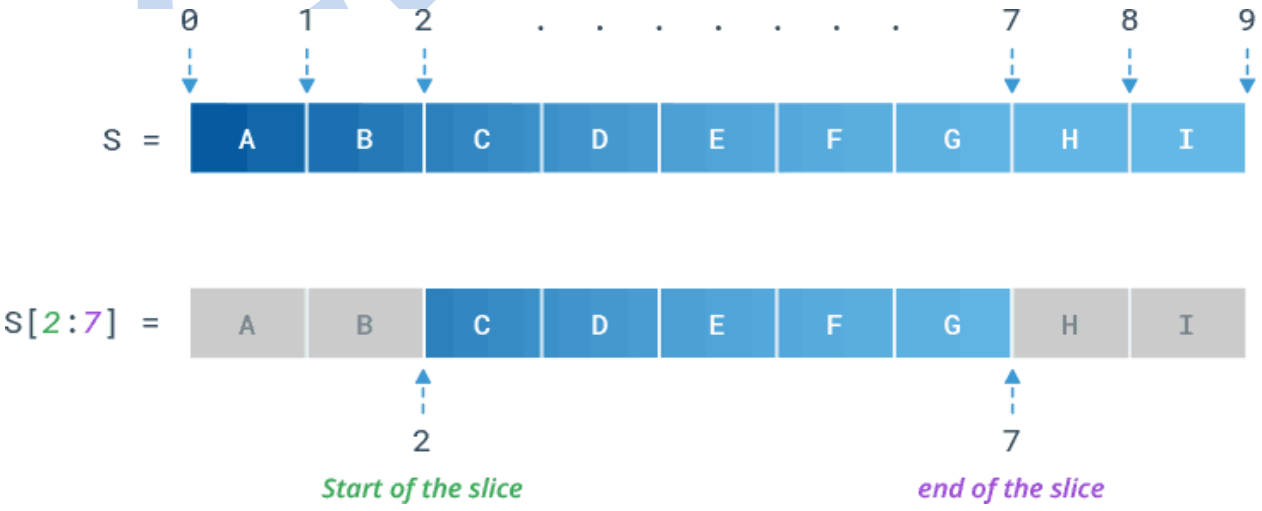
```
S[start:stop:step]
```

Start position End position The increment

Ex:

Here is a basic example of string slicing.

```
S = ' A B C D E F G H I '
print(S[2:7])   #  C D E F G
```

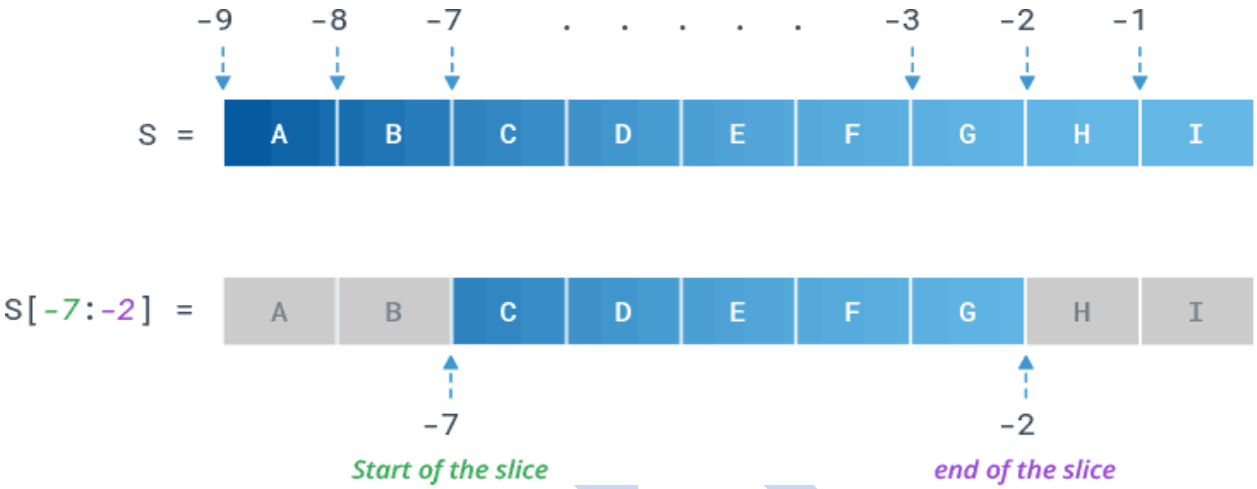


Note that the item at index 7 'H' is not included.

Slice with Negative Indices

You can also specify negative indices while slicing a string.

```
S = ' A B C D E F G H I '

print(S[-7:-2]) #  C D E F G
```



Slice with Positive & Negative Indices

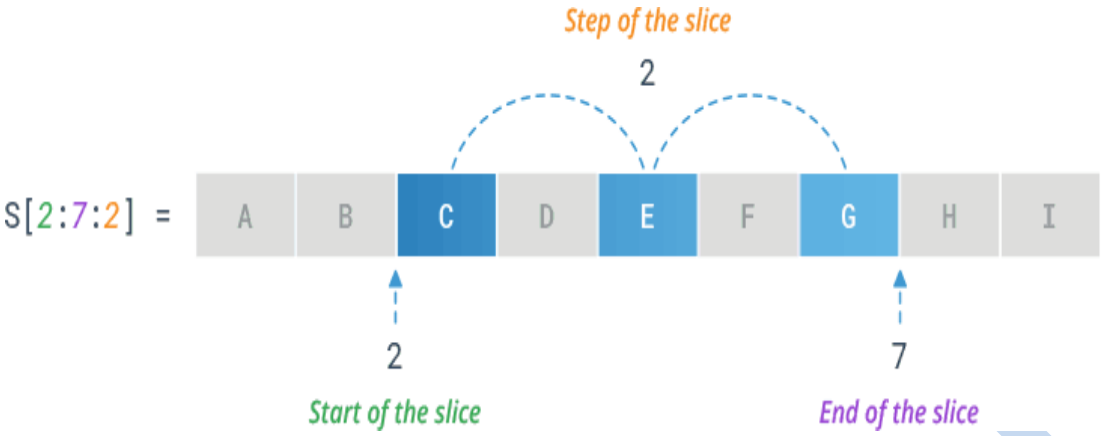
You can specify both positive and negative indices at the same time.

```
S = ' A B C D E F G H I '

print(S[2:-5])  #  C D
```

Specify Step of the Slicing

You can specify the step of the slicing using step parameter. The step parameter is optional and by default 1.

```
# Return every 2nd item between position 2 to 7

S = ' A B C D E F G H I '

print(S[2:7:2]) #  C E G
```



Negative Step Size

You can even specify a negative step size.

```
# Returns every 2nd item between position 6 to 1 in reverse order

S = ' A B C D E F G H I '

print(S[6:1:-2])   #   G E C
```

Slice at Beginning & End

Omitting the start index starts the slice from the index 0. Meaning, S[:stop] is equivalent to S[0:stop]

```
# Slice first three characters from the string

S = ' A B C D E F G H I '

print(S[:3])   #   A B C
```

Whereas, omitting the stop index extends the slice to the end of the string. Meaning, S[start:] is equivalent to S[start:len(S)]

```
# Slice last three characters from the string

S = ' A B C D E F G H I '

print(S[6:])   #   G H I
```

Data Encryption

Plain Text

The plain text message is the text which is readable and can be understood by all users. The plain text is the message which undergoes cryptography.

Cipher Text

Cipher text is the message obtained after applying cryptography on plain text.

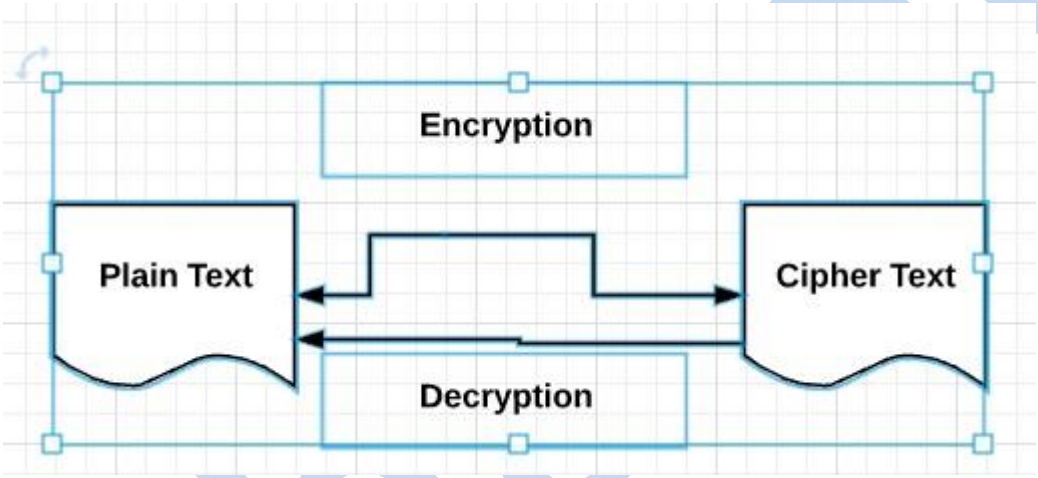
Encryption

The process of converting plain text to cipher text is called encryption. It is also called as encoding.

Decryption

The process of converting cipher text to plain text is called decryption. It is also termed as decoding.

The diagram given below shows an illustration of the complete process of cryptography −



Encryption:
Encryption is the process of encoding the data. i.e converting plain text into ciphertext. This conversion is done with a key called an encryption key.

Decryption:
Decryption is a process of decoding the encoded data. Converting the ciphertext into plain text. This process requires a key that we used for encryption.

We require a key for encryption. There two main types of keys used for encryption and decryption. They are Symmetric-key and Asymmetric-key.

Symmetric-key Encryption:
In symmetric-key encryption, the data is encoded and decoded with the same key. This is the easiest way of encryption, but also less secure. The receiver needs the key for decryption, so a safe way need for transferring keys. Anyone with the key can read the data in the middle.

Example:

Install the python cryptography library with the following command.

pip install cryptography

Steps:

- Import Fernet
- Then generate an encryption key, that can be used for encryption and decryption.
- Convert the string to byte string, so that it can be encrypted.
- Instance the Fernet class with the encryption key.
- Then encrypt the string with Fernet instance.
- Then it can be decrypted with Fernet class instance and it should be instanced with the same key used for encryption.

Ex:

```
from cryptography.fernet import Fernet

# we will be encryting the below string.
message = "hello geeks"

# here I'm using fernet to generate key

key = Fernet.generate_key()

# Instance the Fernet class with the key

fernet = Fernet(key)

# be encoded to byte string before encryption
encMessage = fernet.encrypt(message.encode())

print("original string: ", message)
print("encrypted string: ", encMessage)
# encoded byte string is returned by decrypt method,
# so decode it to string with decode methos
decMessage = fernet.decrypt(encMessage).decode()

print("decrypted string: ", decMessage)
```

Output:

```
original string:  hello geeks
encrypted string:  b'gAAAAABgE4gyG_0ceYqYzE8_qRFbiQ6EO_6ms-uSXiCK9af2PTp4a8e_ONxc2XyO71rzaKxPHvG-jh0iq0CDEky3F_Qmjv8Cdw=='
decrypted string:  hello geeks
```

Asymmetric-key Encryption:

In Asymmetric-key Encryption, we use two keys a public key and private key. The public key is used to encrypt the data and the private key is used to decrypt the data. By the name, the public key can be public (can be sent to anyone who needs to send data). No one has your private key, so no one the middle can read your data.

Example:

Install the python rsa library with the following command.

pip install rsa

Steps:

- Import rsa library
- Generate public and private keys with rsa.newkeys() method.
- Encode the string to byte string.
- Then encrypt the byte string with the public key.
- Then the encrypted string can be decrypted with the private key.
- The public key can only be used for encryption and the private can only be used for decryption.

```
import rsa

# generate public and private keys with
# key length should be atleast 16
publicKey, privateKey = rsa.newkeys(512)

# this is the string that we will be encrypting
message = "hello geeks"

# encode to byte string before encryption
# with encode method
encMessage = rsa.encrypt(message.encode(), publicKey)

print("original string: ", message)
print("encrypted string: ", encMessage)

# public key cannot be used for decryption
decMessage = rsa.decrypt(encMessage, privateKey).decode()

print("decrypted string: ", decMessage)
```

Output:

```
original string:  hello geeks
encrypted string:  b'\x8f\xd0>\xce\xdf>\xec\x14\xb4R\x93\xab+\xcd\x18\xaci\x949\xfcCr\x8e\xe9\xfb
decrypted string:  hello geeks
```

Strings and number system:

- A string is a collection of one or more characters (letters, numbers, symbols). You may need to convert strings to numbers or numbers to strings fairly often.
- The number systems refer to the number of symbols or characters used to represent any numerical value. The number system that you typically use everyday is called decimal. In the decimal system, you use ten different symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. With these ten symbols, you can represent any quantity. Binary, hexadecimal, and octal refer to different number systems

Data conversion in Python can happen in two ways: either you tell the compiler to convert a data type to some other type explicitly, or the compiler understands this by itself and does it for you. In the former case, you're performing an explicit data type conversion, whereas in the latter, you're doing an implicit data type conversion.
There are two-types of type-conversions in Python:

Explicit Conversion: In explicit conversion, users convert the data type in to their required type using int(), float(), str(), etc.

The general form of an explicit data type conversion is as follows:
(required_data_type)(expression)

Implicit Conversion: In implicit conversion, the python interpreter itself converts the lower data type to greater data type

example:

```
a_int = 1
b_float = 1.0
c_sum = a_int + b_float
print(c_sum)
print(type(c_sum))
```

Python String to Int Conversion

It is crucial for a developer to know the different **Python Data types** and conversions between them. Let's take Python String to Int conversion itself, whenever we get the input from the user we typically use the **input() function**

The **input() function** reads a data entered by the user and converts it into a **string** and returns it. Even though the user has entered a **numeric value,** it will be automatically converted to **String** by Python, which cannot be used directly for any manipulation.

For example, if we want to **divide** the number entered by the user by **2.**

```
>>> num = input("Enter a number : ")
Enter a number : 10
```

```
>>> num
'10'
```

The value stored inside **num** is not an **integer 10** but the **string '10'**. Let's check the type of **num** for a confirmation.

```
>>> type(num)
<class 'str'>
```

If we try to divide the **num** by **2**, we will get an **Unsupported operand type error**.

```
>>> print( num /2)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in
    print( num /2)
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

We cannot perform **division operation** on a **String**. So we need to perform the type conversion into the corresponding data type. Let's limit it to **int** for now.

Using int() function to convert Python String to Int

For **integer** related operation, Python provides us with **int class**. We can use the **int() function** present in the **int class** to convert Python String to Int.

The **int()** function comes in two flavors

- **int(x)** – Returns the integer objects constructed from the argument passed, it returns **0** if no argument is passed. It will create an integer object with the default **base 10 (decimal).**
- **int (x, base)** – This method also returns the **integer object** with the corresponding **base** passed as the argument.

Let's try to fix the issue which has happened in the above code. All we need to do is to just pass the **num** variable to the **int()** function to get it converted to an integer.

```
>>> num = int(num)
>>> type(num)
<class 'int'>
>>> print(num /2)
5.0
```

Python Int to String Conversion

Converting a **Python Int to String** is pretty straight forward, we just need to pass the **integer** to the **str() function**. We don't have to worry about the **type of number** or its **bases**.

```
>>> num = 13
>>> strNum = str(num)
```

```
>>> strNum
'13'
>>> binaryNum = '0b1101'
>>> strBinaryNum = str(binaryNum)
>>> strBinaryNum
'0b1101'
>>> hexNum = '0xA12'
>>> strHexNum = str(hexNum)
>>> strHexNum
'0xA12'
```

Python Number Systems :

The python number system is representing the way of using the below numbers in Language.

Binary Number System

Octal Number System

Decimal Number System

Hexadecimal Number System

Binary Number System :

In general, a binary number represents a 0 or 1 in the system.

The base or radix of the binary number system is 2.

The possible digits that are used in a binary number system are 0 and 1.

If we wanted to store a binary number in python variable, that number should sharts with ***0b***

Example: Python binary Number System

```
>>> x=0b1010
>>> print('value is:',x)
value is: 10
>>>
```

Note: we can not give the **x=ob1020** since binary numbers contain only 0 and 1.

If so we will get an error message like SyntaxError: invalid syntax.

Octal Number System :

The base or radix of the octal number system is 8.

The possible digits that are used in the octal number system are 0 to 7.

To represent an octal number in Python, the number should start with ***0 (python2) or 0o (python3)***

Example: Python octal Number System

```
x=0o123
print('Value is : '+x)
```

Output :

```
(Value is : 83)
>>> x=0o123
>>> print(x)
83
```

Note: we can not give the **x=o180** since octal numbers contain from 0 to 7. If so we will get an error message like SyntaxError: invalid token.

Decimal Number System :

The base or radix of the decimal number system is 10.

The possible digits that are used in the decimal number system are 0 to 9.

The default number system followed by python is the decimal number system.

```
x=1234
x=1234
 print('Value is : '+x)
```

Output :

(Value is : 1234)

Note: we can not give the **x=1234p** since the decimal numbers contain from 0 to 9. If so we will get an error message like SyntaxError: invalid syntax.

Hexadecimal Number System :

The base or radix of the hexadecimal number system is 16.

The possible digits that are used in hexadecimal number systems are 0 to 9 and a to f.

To represent a hexadecimal number in Python, the number should start with ***0x***.

```
x=0x25
print('Value is :'+x)
```

Output :

(Value is : 37)

```
>>> x=0x25
>>> print(x)
37
```

Converting to different Bases

We are able to successfully convert a string to an integer using the **int()** function, which is of **base 10.**

Let's try to convert the String to **different bases** other than decimal such as **binary (base 2), octal (base 8)** and **hexadecimal (base 16)**

```
>>> val = '1101'
>>> base2int = int(val, 2)
>>> base2int
13
>>> val = '13'
>>> base8int = int(val, 8)
>>> base8int
11
>>> val = '1A'
```

```
>>> base16int = int(val, 16)
>>> base16int
26
```

Optionally we can also pass the String with the **prefix** for each **base.**

- **'0b'**or **'0B' for binary**
- **'0o'** or **'0O' for octal**
- **'0x'** or **'0X' for hexadecimal**

```
>>> prefixedVal = '0b110'
>>> base2int = int(prefixedVal, 2)
>>> base2int
6
>>> prefixedVal = '0o11'
>>> base8int = int(prefixedVal, 8)
>>> base8int
9
>>> prefixedVal = '0xB'
>>> base16int = int(prefixedVal, 16)
>>> base16int
11
```

Conversion program by using predefined functions

Python built-in functions bin(), oct(), or hex() can be used to convert an integer to a binary, octal, or hexadecimal string respectively.

```
dec = int(input("Enter a decimal number: "))

 print(bin(dec),"in binary.")

print(oct(dec),"in octal.")
print(hex(dec),"in hexadecimal."
```

Output:

```
Enter a decimal number: 10
0b1010 in binary.
0o12 in octal.
0xa in hexadecimal.
```

String Methods

Python has a set of built-in methods that you can use on strings.

The len() function returns the length of a string:

```
a = "Hello, World!"
print(len(a))
```

The strip() method removes any whitespace from the beginning or the end:

```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
```

The lower() method returns the string in lower case:

```
a = "Hello, World!"
print(a.lower())
```

The upper() method returns the string in upper case:

```
a = "Hello, World!"
print(a.upper())
```

The replace() method replaces a string with another string:

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

The split() method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string

encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title

isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value

strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

Text files: reading/writing text and numbers from/to a file :

Reading and Writing to text files in Python

Python provides inbuilt functions for creating, writing and reading files. There are two types of files that can be handled in python, normal text files and binary files (written in binary language,0s and 1s).

- **Text files:** In this type of file, Each line of text is terminated with a special character called EOL (End of Line), which is the new line character ('\n') in python by default.
- **Binary files:** In this type of file, there is no terminator for a line and the data is stored after converting it into machine understandable binary language.

File Access Modes

Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once its opened. These modes also define the location of the File Handle in the file. File handle is like a cursor, which defines from where the data has to be read or written in the file. There are 6 access modes in python.

1. Read Only ('r') : Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exists, raises I/O error. This is also the default mode in which file is opened.
2. Read and Write ('r+') : Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exists.
3. Write Only ('w') : Open the file for writing. For existing file, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exists.
4. Write and Read ('w+') : Open the file for reading and writing. For existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.

5. Append Only ('a') : Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
6. Append and Read ('a+') : Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

Python File Modes

Mode	Description
'r'	Open a file for reading. (default)
'w'	Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
'x'	Open a file for exclusive creation. If the file already exists, the operation fails.
'a'	Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.
't'	Open in text mode. (default)
'b'	Open in binary mode.
'+'	Open a file for updating (reading and writing)

Opening a File

It is done using the open() function. No module is required to be imported for this function.

```
File_object = open(r"File_Name","Access_Mode")
```

The file should exist in the same directory as the python program file else, full address of the file should be written on place of filename.

Note: The r is placed before filename to prevent the characters in filename string to be treated as special character.

The r can be ignored if the file is in same directory and address is not being placed.

While specifying the exact path, characters prefaced by \ (like \n \r \t etc.) are interpreted as special characters. You can escape them using:

raw strings like r'C:\new\text.txt'

double backslashes like 'C:\\new\\text.txt'

```
# Open function to open the file "MyFile1.txt"
# (same directory) in append mode and
file1 = open("MyFile1.txt","a")
  file2 = open(r"D:\Text\MyFile2.txt","w+")
```

Here, file1 is created as object for MyFile1 and file2 as object for MyFile2

Closing a file

close() function closes the file and frees the memory space acquired by that file. It is used at the time when the file is no longer needed or if it is to be opened in a different file mode.

File_object.close()

```
# Opening and Closing a file "MyFile.txt"
# for object name file1.
file1 = open("MyFile.txt","a")
file1.close()
```

Writing to a file

There are two ways to write in a file.

1. write() : Inserts the string str1 in a single line in the text file.
   File_object.write(str1)
2. writelines() : For a list of string elements, each string is inserted in the text file.Used to insert multiple strings at a single time.
   File_object.writelines(L) for L = [str1, str2, str3]

Reading from a file

There are three ways to read data from a text file.

1. read() : Returns the read bytes in form of a string. Reads n bytes, if no n specified, reads the entire file.
   File_object.read([n])
2. readline() : Reads a line of the file and returns in form of a string.For specified n, reads at most n bytes. However, does not reads more than one line, even if n exceeds the length of the line.
   File_object.readline([n])
3. readlines() : Reads all the lines and return them as each line a string element in a list.
   File_object.readlines()

Note: '\n' is treated as a special character of two bytes

With Statement

You can also work with file objects using the with statement. It is designed to provide much cleaner syntax and exceptions handling when you are working with code. That explains why it's good practice to use the with statement where applicable.

```
with open("testfile.txt") as file:
    data = file.read()
    print(data)
```

EXAMPLE:

 # Program to show various ways to read and

```
# write data in a file.
file1 = open("myfile.txt","w")
L = ["This is Delhi \n","This is Paris \n","This is London \n"]
file1.write("Hello \n")
file1.writelines(L)
file1.close() #to change file access modes
file1 = open("myfile.txt","r+")
print( "Output of Read function is ")
print (file1.read())
# seek(n) takes the file handle to the nth
# bite from the beginning.
file1.seek(0)

print ("Output of Readline function is ")
print (file1.readline())

file1.seek(0)
# To show difference between read and readline
print ("Output of Read(9) function is ")
print (file1.read(9))

file1.seek(0)
print ("Output of Readline(9) function is ")
print (file1.readline(9))

file1.seek(0)
# readlines function
print ("Output of Readlines function is ")
print (file1.readlines())
file1.close()
```

Output:

Output of Read function is

Hello

This is Delhi

This is Paris

This is London

Output of Readline function is

```
Hello
Output of Read(9) function is
Hello
Th

Output of Readline(9) function is
Hello
Output of Readlines function is
['Hello \n', 'This is Delhi \n', 'This is Paris \n', 'This is London \n']
```

Appending to a file

```
# Python program to illustrate
# Append vs write mode
file1 = open("myfile.txt","w")
L = ["This is Delhi \n","This is Paris \n","This is London \n"]
file1.close()
# Append-adds at last
file1 = open("myfile.txt","a")#append mode
file1.write("Today \n")
file1.close()
file1 = open("myfile.txt","r")
print ("Output of Readlines after appending")
print (file1.readlines())
file1.close()
# Write-Overwrites
file1 = open("myfile.txt","w")#write mode
```

```
file1.write("Tomorrow \n")

file1.close()

file1 = open("myfile.txt","r")

print ("Output of Readlines after writing")

print (file1.readlines())

print

file1.close()
```

Output:

Output of Readlines after appending

['This is Delhi \n', 'This is Paris \n', 'This is London \n', 'Today \n']

Output of Readlines after writing

['Tomorrow \n']

Writing Numbers to a file

- The write method expects a string as an argument.
- Therefore, if we want to write other data types, such as integers or floating point numbers then the numbers must be first converted into strings before writing them to an output file.
- In python ,the value of most datatypes can be converted to strings by using str function.
- In order to read the numbers correctly, we need to separate them using special characters, such as " " (space) or '\n' (new line).

Program:

```
obj1 = open("Numbers.txt","w")

for x in range(1,21):

    x = str(x)

    obj1.write(x)

    obj1.write(" ")

obj1.close()
```

Reading Numbers from a file :

- All of the file input operations return the data to the program as string.
- If these strings represents other types of data ,such as integers or floating point numbers , the programmers must convert them to the appropriate types.
- In python , the string represents of numbers nd float point numvers can be converted to the numbers by using in tans float functions.
- In the text file contains intergers separated by new lines
- The **strip** method is used to remove the newline and runs the **int function** to obtain the integer value

#Code Example using read()

```
fp1 = open("numbers.txt","r")
num = fp1.read( )
print(num)
print(type(num))
```

#Code Example for sum of numbers in a file

```
f=open("numbers.txt","r")
s=0
for line in f:
   line=line.strip()
   n=int(line)
   s=s+n
print("The sum is ",s)
```

output:

```
1
2
3
4

<class 'str'>
The sum is  10
```

program 2:

```
#Code Example using readline( )
obj1 = open("numbers.txt","w")
for x in range(1,5):
    x = str(x)
    obj1.write(x)
    obj1.write(" \n")
obj1.close()
```

```
fp1 = open("numbers.txt","r")
num = fp1.read( )
print(num)
print(type(num))

fp1 = open("numbers.txt","r")
sum = 0
for i in fp1:
    num1 = int(fp1.readline( ))
    sum = sum + num1
print(sum)
```

output:

```
1
2
3
4

<class 'str'>
6
```