PYTHON PROGRAMMING
LECTURE NOTES

B.TECH III SEM (2023-24)

DEPARTMENT OF

COMPUTER SCIENCE AND ENGINEERING

PRESIDENCY UNIVERSITY

PYTHON PROGRAMMING

OUTCOMES: Upon completion of the course, students will be able to

- Read, write, execute by hand simple Python programs.
- Structure simple Python programs for solving problems.
- Decompose a Python program into functions.
- Represent compound data using Python lists, tuples, dictionaries.
- Read and write data from/to files in Python Programs

TEXT BOOKS

1.Allen B. Downey, ``Think Python: How to Think Like a Computer Scientist‘‘, 2nd edition, Updated for Python 3, Shroff/O‘Reilly Publishers, 2016.
2.R. Nageswara Rao, “Core Python Programming”, dreamtech
3. Python Programming: A Modern Approach, Vamsi Kurama, Pearson

REFERENCE BOOKS:

1. Core Python Programming, W.Chun, Pearson.
2. Introduction to Python, Kenneth A. Lambert, Cengage
3. Learning Python, Mark Lutz, Orielly

INDEX

LISTS, TUPLES, DICTIONARIES
Lists
list operations
list slices
list methods
list loop
mutability
aliasing
cloning lists
list parameters
list comprehension
Tuples
tuple assignment
tuple as return value
tuple comprehension
Dictionaries
operations and methods
comprehension

UNIT – I

INTRODUCTION DATA, EXPRESSIONS, STATEMENTS

Introduction to Python and installation, data types: Int, float, Boolean, string, and list; variables, expressions, statements, precedence of operators, comments; modules, functions---- function and its use, flow of execution, parameters and arguments.

Introduction to Python and installation:

Python is a widely used general-purpose, high level programming language. It was initially designed by **Guido van Rossum in 1991** and developed by Python Software Foundation. It was mainly developed for emphasis on code readability, and its syntax allows programmers to express concepts in fewer lines of code.
Python is a programming language that lets you work quickly and integrate systems more efficiently.

There are two major Python versions- **Python 2 and Python 3**.

- On 16 October 2000, Python 2.0 was released with many new features.
- On 3rd December 2008, Python 3.0 was released with more testing and includes new features.

Beginning with Python programming:

1) Finding an Interpreter:

Before we start Python programming, we need to have an interpreter to interpret and run our programs. There are certain online interpreters like **https://ide.geeksforgeeks.org/**, http://ideone.com/ or http://codepad.org/ that can be used to start Python without installing an interpreter.

***Windows*:** There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) which is installed when you install the python software from **http://python.org/downloads/**

2) Writing first program:

```
# Script Begins
    Statement1
```

Statement2

Statement3

Script Ends

Differences between scripting language and programming language:

SCRIPTING LANGUAGE	PROGRAMMING LANGUAGE
A programming language that supports scripts: programs written for a special run-time environment that automate the execution of tasks	A formal language, which comprises a set of instructions used to produce various kinds of output
Execution speed is slow	Compiler-based languages are executed much faster while interpreter-based languages are executed slower
Can be divided into client-side scripting languages and server-side scripting languages	Can be divided into high-level, low-level languages or compiler-based or interpreter-based languages
Easier to learn	Not as easy to learn
Ex: JavaScript, Perl, PHP, Python and Ruby	Ex: C, C++, and Assembly
Mostly used for web development	Used to develop various applications such as desktop, web, mobile, etc.

Why to use Python:

The following are the primary factors to use python in day-to-day life:

1. **Python is object-oriented**
   Structure supports such concepts as polymorphism, operation overloading and multiple inheritance.
2. **Indentation**
   Indentation is one of the greatest feature in python

3. **It's free (open source)**
   Downloading python and installing python is free and easy
4. **It's Powerful**
   - Dynamic typing
   - Built-in types and tools
   - Library utilities
   - Third party utilities (e.g. Numeric, NumPy, sciPy)
   - Automatic memory management
5. **It's Portable**
   - Python runs virtually every major platform used today
   - As long as you have a compaitable python interpreter installed, python programs will run in exactly the same manner, irrespective of platform.
6. **It's easy to use and learn**
   - No intermediate compile
   - Python Programs are compiled automatically to an intermediate form called byte code, which the interpreter then reads.
   - This gives python the development speed of an interpreter without the performance loss inherent in purely interpreted languages.
   - Structure and syntax are pretty intuitive and easy to grasp.
7. **Interpreted Language**
   Python is processed at runtime by python Interpreter
8. **Interactive Programming Language**
   Users can interact with the python interpreter directly for writing the programs
9. **Straight forward syntax**
   The formation of python syntax is simple and straight forward which also makes it popular.

Installation:

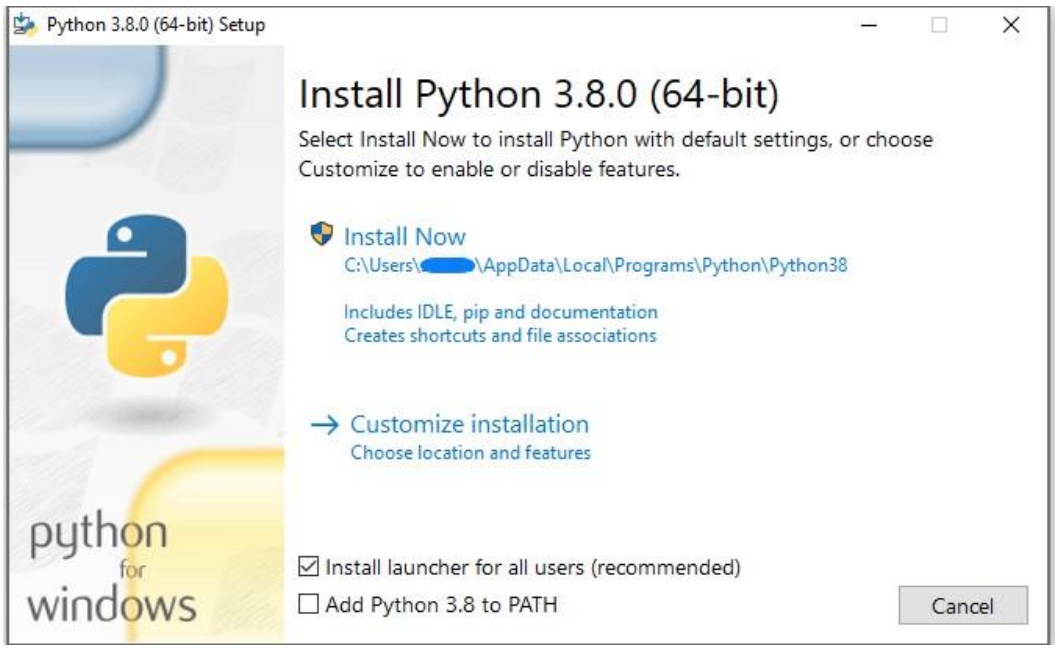
There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) which is installed when you install the python software from http://python.org/downloads/

Steps to be followed and remembered:

Step 1: Select Version of Python to Install.
Step 2: Download Python Executable Installer.
Step 3: Run Executable Installer.
Step 4: Verify Python Was Installed On Windows.

Step 5: Verify Pip Was Installed.

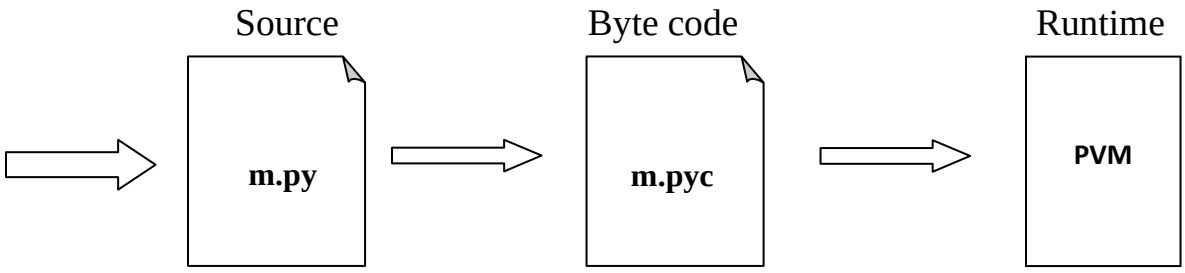
Step 6: Add Python Path to Environment Variables (Optional)



Working with Python

Python Code Execution:

Python's traditional runtime execution model: Source code you type is translated to byte code, which is then run by the Python Virtual Machine (PVM). Your code is automatically compiled, but then it is interpreted.



Source code extension is .py
Byte code extension is .pyc (Compiled python code)

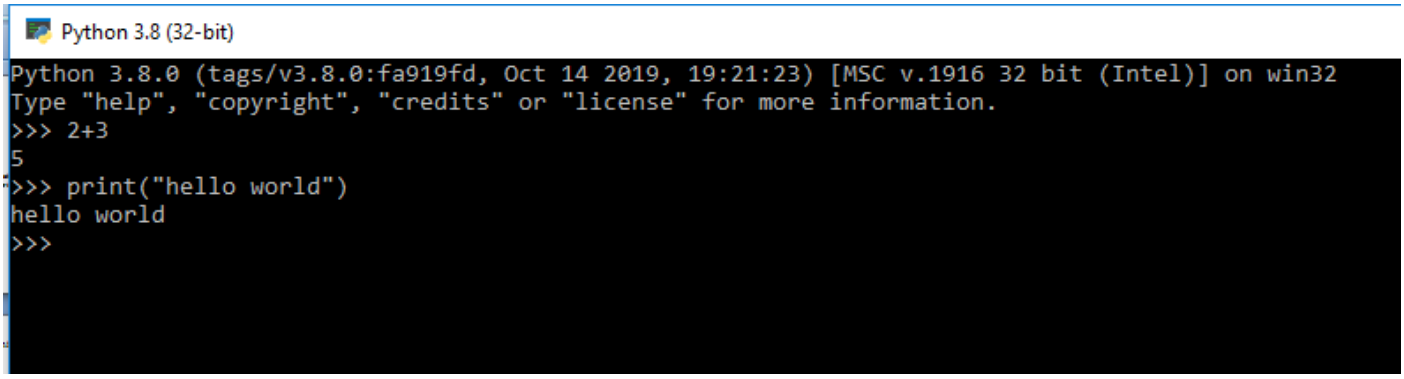
There are two modes for using the Python interpreter:

- Interactive Mode
- Script Mode

Running Python in interactive mode:

Without passing python script file to the interpreter, directly execute code to Python prompt. Once you're inside the python interpreter, then you can start.

```
>>> print("hello world")
hello world
# Relevant output is displayed on subsequent lines without the >>> symbol
>>> x=[0,1,2]
# Quantities stored in memory are not displayed by default.
>>> x
#If a quantity is stored in memory, typing its name will display it.
[0, 1, 2]
>>> 2+3
5
```



The chevron at the beginning of the 1st line, i.e., the symbol >>> is a prompt the python interpreter uses to indicate that it is ready. If the programmer types 2+6, the interpreter replies 8.

Running Python in script mode:

Alternatively, programmers can store Python script source code in a file with the .py extension, and use the interpreter to execute the contents of the file. To execute the script by the interpreter, you have to tell the interpreter the name of the file. For example, if you have a script name MyFile.py and you're working on Unix, to run the script you have to type:

python MyFile.py

Working with the interactive mode is better when Python programmers deal with small pieces of code as you can type and execute them immediately, but when the code is more than 2-4 lines, using the script for coding can help to modify and use the code in future.

Example:

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\pyyy>python e1.py
resource open
the no cant be divisible zero division by zero
resource close
finished
```

Data types:

The data stored in memory can be of many types. For example, a student roll number is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Int:

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

```
>>> print(24656354687654+2)
24656354687656
>>> print(20)
20
>>> print(0b10)
2
```

```
>>> print(0B10)
2
>>> print(0X20)
32
>>> 20
20
>>> 0b10
2
>>> a=10
>>> print(a)
10
```

 # To verify the type of any object in Python, use the type() function:

```
>>> type(10)
<class 'int'>
>>> a=11
>>> print(type(a))
<class 'int'>
```

Float:

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

Float can also be scientific numbers with an "e" to indicate the power of 10.

```
>>> y=2.8
>>> y
2.8
>>> y=2.8
>>> print(type(y))
<class 'float'>
>>> type(.4)
<class 'float'>
>>> 2.
```

2.0

Example:

```
x = 35e3
y = 12E4
z = -87.7e100

print(type(x))
print(type(y))
print(type(z))
```

Output:

```
<class 'float'>
<class 'float'>
<class 'float'>
```

Boolean:

Objects of Boolean type may have one of two values, True or False:

```
>>> type(True)

<class 'bool'>

>>> type(False)

<class 'bool'>
```

String:

1. Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes.

- 'hello' is the same as "hello".
- Strings can be output to screen using the print function. **For example: print("hello").**

```
>>> print("college")

college

>>> type("college")

<class 'str'>
```

```
>>> print('This college')

This college

>>> " "

' '
```

If you want to include either type of quote character within the string, the simplest way is to delimit the string with the other type. If a string is to contain a single quote, delimit it with double quotes and vice versa:

```
>>> print("This is an autonomous (') college")

This is an autonomous (') college

>>> print('This is an autonomous (") college')

This is an autonomous (") college
```

Suppressing Special Character:

Specifying a backslash (\) in front of the quote character in a string "escapes" it and causes Python to suppress its usual special meaning. It is then interpreted simply as a literal single quote character:

```
>>> print("This is an autonomous (\') college")

This is an autonomous (') college

>>> print('This is an autonomous (\") college')

This is an autonomous (") college
```

The following is a table of escape sequences which cause Python to suppress the usual special interpretation of a character in a string:

```
>>> print('a\
....b')
a. ..b
>>> print('a\
b\
c')
```

```
abc
>>> print('a \n b')
a
 b
>>> print("This \n college")
This
 college
```

Escape Sequence	Usual Interpretation of Character(s) After Backslash	"Escaped" Interpretation
\'	Terminates string with single quote opening delimiter	Literal single quote (') character
\"	Terminates string with double quote opening delimiter	Literal double quote (") character
\newline	Terminates input line	Newline is ignored
\\	Introduces escape sequence	Literal backslash (\) character

In Python (and almost all other common computer languages), a tab character can be specified by the escape sequence \t:

```
>>> print("a\tb")
a	b
```

List:

- It is a general purpose most widely used in data structures
- List is a collection which is ordered and changeable and allows duplicate members. (Grow and shrink as needed, sequence type, sortable).
- To use a list, you must declare it first. Do this using square brackets and separate values with commas.
- We can construct / create list in many ways.

Ex:

```
>>> list1=[1,2,3,'A','B',7,8,[10,11]]
>>> print(list1)
[1, 2, 3, 'A', 'B', 7, 8, [10, 11]]
```

```
>>> x=list()
>>> x
[]
----------
>>> tuple1=(1,2,3,4)
>>> x=list(tuple1)
>>> x
[1, 2, 3, 4]
```

Variables:

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Assigning Values to Variables:

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

For example −

```
a= 100          # An integer assignment
b = 1000.0          # A floating point
c = "John"          # A string
print (a)
print (b)
print (c)
```

This produces the following result −

```
100
1000.0
John
```

Multiple Assignment:

Python allows you to assign a single value to several variables simultaneously.

For example :

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables.

For example −

```
a,b,c = 1,2,"This“
```

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

Output Variables:

The Python print statement is often used to output variables.

Variables do not need to be declared with any particular type and can even change type after they have been set.

```
x = 5              # x is of type int
x = "This "        # x is now of type str
print(x)
```

Output: This

To combine both text and a variable, Python uses the **"+"** character:

Example

```
x = "awesome"
print("Python is " + x)
```

Output

Python is awesome

You can also use the + character to add a variable to another variable:

Example

```
x = "Python is "
y = "awesome"
z = x + y
print(z)
```

Output:

Python is awesome

Expressions:

An expression is a combination of values, variables, and operators. An expression is evaluated using assignment operator.

Examples: Y=x + 17

```
>>> x=10
>>> z=x+20
>>> z
30
```

```
>>> x=10
>>> y=20
>>> c=x+y
>>> c
30
```

A value all by itself is a simple expression, and so is a variable.

```
>>> y=20
>>> y
20
```

Python also defines expressions only contain identifiers, literals, and operators. So,

Identifiers: Any name that is used to define a class, function, variable module, or object is an identifier.

Literals: These are language-independent terms in Python and should exist independently in any programming language. In Python, there are the string literals, byte literals, integer literals, floating point literals, and imaginary literals.

Operators: In Python you can implement the following operations using the corresponding tokens.

Operator	Token
add	+
subtract	-
multiply	*
Integer Division	/
remainder	%
Binary left shift	<<
Binary right shift	>>
and	&
or	\
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=
Check equality	==
Check not equal	!=

Some of the python expressions are

Conditional expression:

Syntax: true_value if Condition else false_value

```
>>> x = "1" if True else "2"

>>> x

'1'
```

Statements:

A statement is an instruction that the Python interpreter can execute. We have normally two basic statements, the assignment statement and the print statement. Some other kinds of statements that are if statements, while statements, and for statements generally called as control flows.

Examples:

An assignment statement creates new variables and gives them values:

```
>>> x=10
>>> college="This"
```

An print statement is something which is an input from the user, to be printed / displayed on to the screen (or) monitor.

```
>>> print("This college")

This college
```

Precedence of Operators:

Operator precedence affects how an expression is evaluated.

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first multiplies 3*2 and then adds into 7.

Example 1:

```
>>> 3+4*2

11
```

Multiplication gets evaluated before the addition operation

```
>>> (10+10)*2

40
```

Parentheses () overriding the precedence of the arithmetic operators

Example 2:

```
a = 20
b = 10
c = 15
d = 5
e = 0

e = (a + b) * c / d       #( 30 * 15 ) / 5
print("Value of (a + b) * c / d is ",  e)

e = ((a + b) * c) / d     # (30 * 15 ) / 5
print("Value of ((a + b) * c) / d is ",  e)

e = (a + b) * (c / d);    # (30) * (15/5)
print("Value of (a + b) * (c / d) is ",  e)

e = a + (b * c) / d;      # 20 + (150/5)
print("Value of a + (b * c) / d is ",  e)
```

Output:

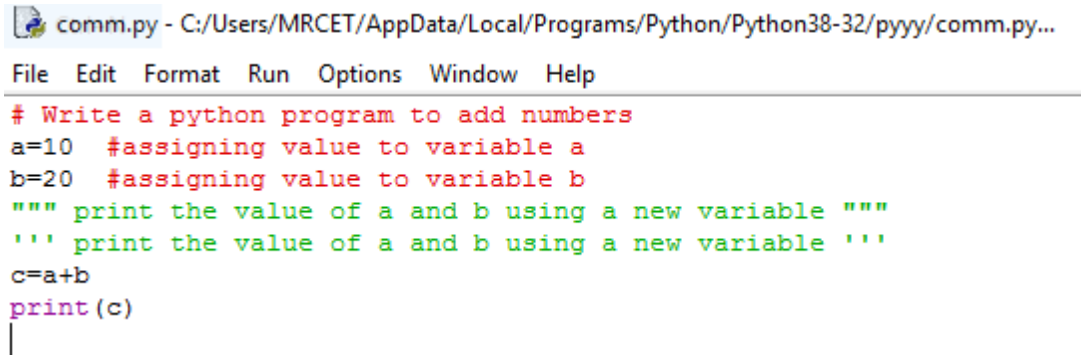
```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/opprec.py
Value of (a + b) * c / d is  90.0
Value of ((a + b) * c) / d is 90.0
Value of (a + b) * (c / d) is 90.0
Value of a + (b * c) / d is  50.0
```

Comments:

Single-line comments begins with a hash(#) symbol and is useful in mentioning that the whole line should be considered as a comment until the end of line.

A Multi line comment is useful when we need to comment on many lines. In python, triple double quote(“ “ “) and single quote(‘ ‘ ‘)are used for multi-line commenting.

Example:



Output:

C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/comm.py

30

Flow of Execution:

1. The order in which statements are executed is called the flow of execution
2. Execution always begins at the first statement of the program.
3. Statements are executed one at a time, in order, from top to bottom.
4. Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.
5. Function calls are like a bypass in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.

Note: When you read a program, don't read from top to bottom. Instead, follow the flow of execution. This means that you will read the def statements as you are scanning from top to bottom, but you should skip the statements of the function definition until you reach a point where that function is called.

Example:

#example for flow of execution

```
print("welcome")
for x in range(3):
    print(x)
print("Good morning college")
```

Output:

C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py

welcome

0

1

2

Good morning college

The flow/order of execution is: 2,3,4,3,4,3,4,5

flowof.py - C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py (3...

File Edit Format Run Options Window Help

```
#example for flow of execution with functions
print("welcome")
for x in range(3):
    print(x)
print("Good morning college")
```

---

flowof.py - C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py (3...

File Edit Format Run Options Window Help

```
#example for flow of execution with functions
def hello():
    print("Good morning")
    print("mrcet")
print("hi")
print("hello")
hello()
print("done!")
```

Output:

C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py

hi

hello

Good morning

This

done!

The flow/order of execution is: 2,5,6,7,2,3,4,7,8

UNIT – II

CONTROL FLOW, LOOPS

Conditionals: Boolean values and operators, conditional (if), alternative (if-else), chained conditional (if-elif-else); Iteration: while, for, break, continue.

Control Flow, Loops:

Boolean Values and Operators:

A boolean expression is an expression that is either true or false. The following examples use the operator ==, which compares two operands and produces True if they are equal and False otherwise:

>>> 5 == 5

True

>>> 5 == 6

False

True and False are special values that belong to the type bool; they are not strings:

>>> type(True)

<class 'bool'>

>>> type(False)

<class 'bool'>

The == operator is one of the relational operators; the others are: x != y # x is not equal to y

x > y # x is greater than y x < y # x is less than y

x >= y # x is greater than or equal to y x <= y # x is less than or equal to y

Note:

All expressions involving relational and logical operators will evaluate to either true or false

Conditional (if):

The if statement contains a logical expression using which data is compared and a decision is made based on the result of the comparison.

Syntax:
if expression:
    statement(s)

If the boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. If boolean expression evaluates to FALSE, then the first set of code after the end of the if statement(s) is executed.

if Statement Flowchart:

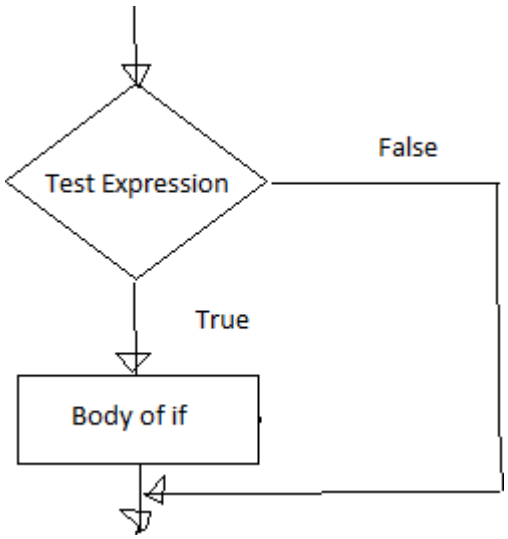


Fig: Operation of if statement

Example: Python if Statement

```
a = 3
if a > 2:
   print(a, "is greater")
print("done")

a = -1
if a < 0:
    print(a, "a is smaller")
print("Finish")
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/if1.py
3 is greater
done
-1 a is smaller
Finish
- - - - - - - - - - - - - - - - - - - - -

a=10

if a>9:

   print("A is Greater than 9")
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/if2.py

A is Greater than 9
```

Alternative if (If-Else):

An else statement can be combined with an if statement. An else statement contains the block of code (false block) that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

The else statement is an optional statement and there could be at most only one else Statement following if.

Syntax of if - else :

```
    if test expression:
```

Body of if stmts

else:

Body of else stmts

If - else Flowchart :

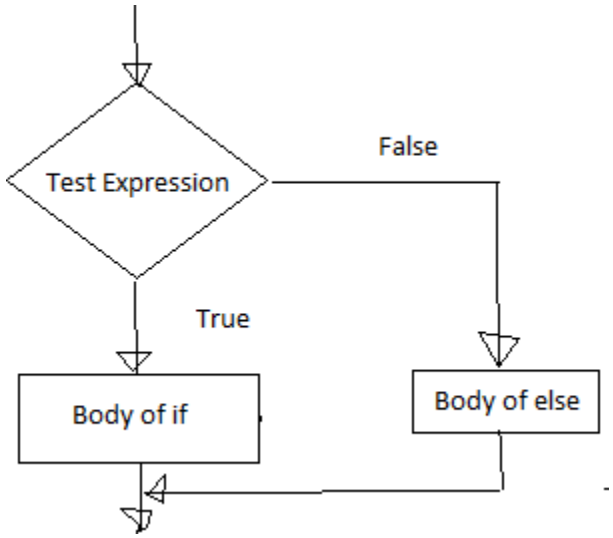


Fig: Operation of if – else statement

Example of if - else:

```
a=int(input('enter the number'))
if a>5:
   print("a is greater")
else:
   print("a is smaller than the input given")
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/ifelse.py
enter the number 2
a is smaller than the input given
----------------------------------------
```

```
a=10
b=20
if a>b:
   print("A is Greater than B")
else:
   print("B is Greater than A")
```

Output:

C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/if2.py
B is Greater than A

Chained Conditional: (If-elif-else):

The elif statement allows us to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE. Similar to the else, the elif statement is optional. However, unlike else, for which there can be at most one statement, there can be an arbitrary number of elif statements following an if.

Syntax of if – elif - else :

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

Flowchart of if – elif - else:

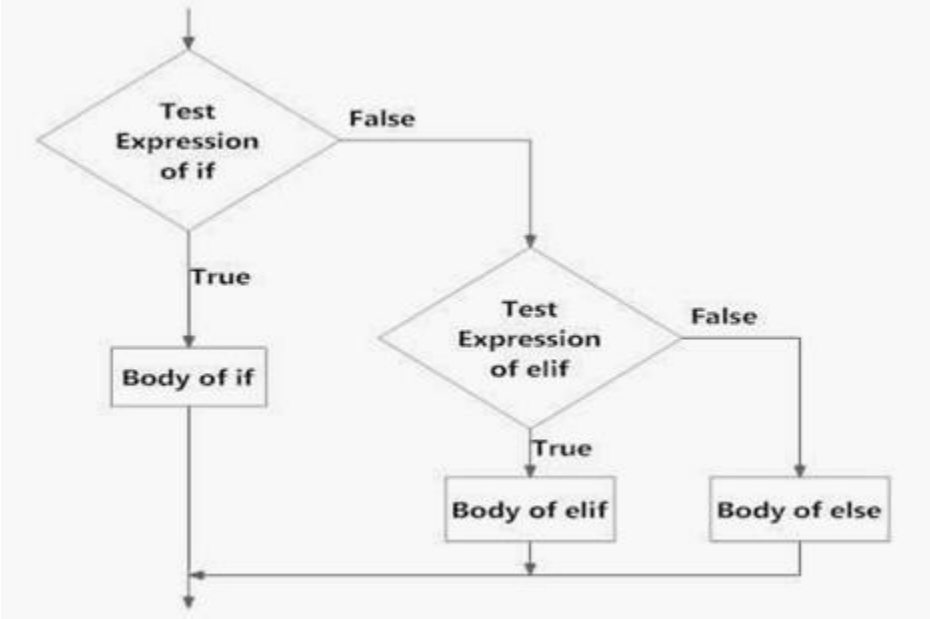


Fig: Operation of if – elif - else statement

Example of if - elif – else:

```
a=int(input('enter the number'))
b=int(input('enter the number'))
c=int(input('enter the number'))
if a>b:
   print("a is greater")
elif b>c:
   print("b is greater")
else:
   print("c is greater")
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/ifelse.py
enter the number5
enter the number2
enter the number9
a is greater
>>>
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/ifelse.py
enter the number2
enter the number5
enter the number9
c is greater
```

```
var = 100
if var == 200:
   print("1 - Got a true expression value")
   print(var)
elif var == 150:
   print("2 - Got a true expression value")
   print(var)
elif var == 100:
   print("3 - Got a true expression value")
   print(var)
else:
   print("4 - Got a false expression value")
   print(var)
print("Good bye!")
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/ifelif.py

3 - Got a true expression value
```

100

Good bye!

Iteration:

A loop statement allows us to execute a statement or group of statements multiple times as long as the condition is true. Repeated execution of a set of statements with the help of loops is called iteration.

Loops statements are used when we need to run same code again and again, each time with a different value.

Statements:

In Python Iteration (Loops) statements are of three types:

1. While Loop
2. For Loop
3. Nested For Loops

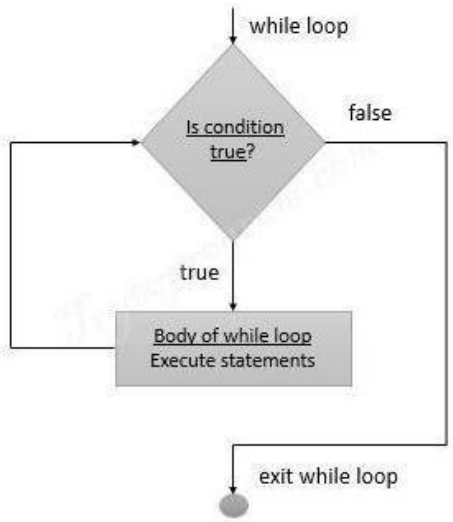
While loop:

- Loops are either infinite or conditional. Python while loop keeps reiterating a block of code defined inside it until the desired condition is met.
- The while loop contains a boolean expression and the code inside the loop is repeatedly executed as long as the boolean expression is true.
- The statements that are executed inside while can be a single line of code or a block of multiple statements.

Syntax:

```
while(expression):
        Statement(s)
```

Flowchart:



Example Programs:

1.

```
i=1
while i<=6:
        print("This college")
        i=i+1
```

output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/wh1.py This
college
This    college
This    college
This    college
This    college
This college
```

2.

```
i=1

while i<=3:
   print("",end=" ")
   j=1
   while j<=1:
      print("CSE DEPT",end="")
      j=j+1
   i=i+1
   print()
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/wh2.py

 CSE DEPT

 CSE DEPT

 CSE DEPT
```

3.

```
    i=1
```

```
j=1
while i<=3:
    print("",end="  ")

    while j<=1:
        print("CSE DEPT",end="")
        j=j+1
    i=i+1
    print()
```

Output:

C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/wh3.py

CSE DEPT

4. _____

```
i = 1
while (i < 10):
      print (i)
      i = i+1
```

Output:

C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/wh4.py

1
2
3
4
5
6
7
8
9

2. _____

```
a = 1
b = 1
while (a<10):
    print ('Iteration',a)
    a = a + 1
    b = b + 1
```

```
            if (b == 4):
                break
        print ('While loop terminated')
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/wh5.py
Iteration 1
Iteration 2
Iteration 3
While loop terminated
- - - - - - - - - - - - - - - -
count = 0
while (count < 9):
   print("The count is:", count)
   count = count + 1
print("Good bye!")
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/wh.py =
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Good bye!
```

For loop:

Python **for loop** is used for repeated execution of a group of statements for the desired number of times. It iterates over the items of lists, tuples, strings, the dictionaries and other iterable objects

Syntax: for var in sequence:

Statement(s)

Holds the value of item in sequence in each iteration

A sequence of values assigned to var in each iteration

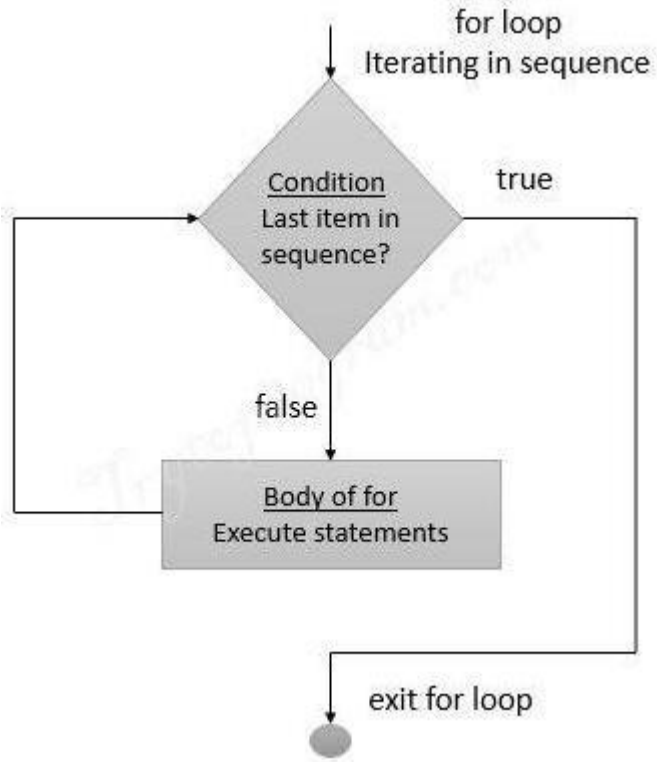
Sample Program:

```
numbers = [1, 2, 4, 6, 11, 20]
seq=0
for val in numbers:
    seq=val*val
    print(seq)
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/fr.py
1
4
16
36
121
400
```

Flowchart:



Iterating over a list:

```
#list of items
list = ['M','R','C','E','T']
i = 1

#Iterating over the list
for item in list:
  print ('college ',i,' is ',item)
  i = i+1
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/lis.py
college 1 is  M
college  2  is  R
college  3  is  C
college  4  is  E
college 5 is  T
```

Iterating over a Tuple:

```
tuple = (2,3,5,7)
print ('These are the first four prime numbers ')
#Iterating over the tuple
for a in tuple:
  print (a)
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/fr3.py
These are the first four prime numbers
2
3
5
7
```

Iterating over a dictionary:

```
#creating a dictionary
college = {"ces":"block1","it":"block2","ece":"block3"}

#Iterating over the dictionary to print keys
print ('Keys are:')
```

```
for keys in college:
  print (keys)

#Iterating over the dictionary to print values
print ('Values are:')
for blocks in college.values():
  print(blocks)
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/dic.py
Keys are:
ces
it
ece
Values are:
block1
block2
block3
```

Iterating over a String:

```
#declare a string to iterate over
college = "

#Iterating over the string
for name in college:
  print (name)
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/strr.py
M
R
C
E
T
```

Nested For loop:

When one Loop defined within another Loop is called Nested Loops.

Syntax:

```
for val in sequence:
        for val in sequence:
```

statements

statements

 # Example 1 of Nested For Loops (Pattern Programs)

```
for i in range(1,6):
  for j in range(0,i):
    print(i, end=" ")
  print('')
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/nesforr.py
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
- - - - - - - - - - - - - - - - -
```

 # Example 2 of Nested For Loops (Pattern Programs)

```
for i in range(1,6):
  for j in range(5,i-1,-1):
    print(i, end=" ")
  print('')
```

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/nesforr.py
```

Output:

```
1 1 1 1 1
2 2 2 2
3 3 3
4 4
```

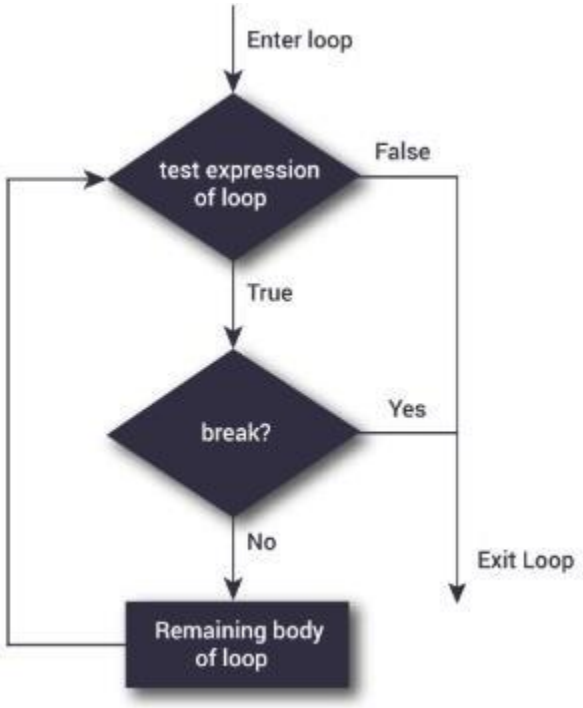
Break and continue:

In Python, **break and continue** statements can alter the flow of a normal loop. Sometimes we wish to terminate the current iteration or even the whole loop without checking test expression. The break and continue statements are used in these cases.

Break:

The break statement terminates the loop containing it and 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.

Flowchart:



The following shows the working of break statement in for and while loop:

```
for var in sequence:
        # code inside for loop
        If condition:
                break (if break condition satisfies it jumps to outside loop)
        # code inside for loop
# code outside for loop
```

```
while test expression

        # code inside while loop
        If condition:
                break (if break condition satisfies it jumps to outside loop)
        # code inside while loop
# code outside while loop
```

Example:

```
for val in " COLLEGE":
    if val == " ":
        break
    print(val)

print("The end")
```

Output:

```
M
R
C
E
T
The end
```

 # Program to display all the elements before number 88

```
for num in [11, 9, 88, 10, 90, 3, 19]:
   print(num)
   if(num==88):
            print("The number 88 is found")
            print("Terminating the loop")
            break
```

Output:

```
11
9
88
The number 88 is found
```

Terminating the loop

```
#_____________________________
for letter in "Python": # First Example
   if letter == "h":
      break
   print("Current Letter :", letter )
```

Output:

C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/br.py =

Current Letter : P

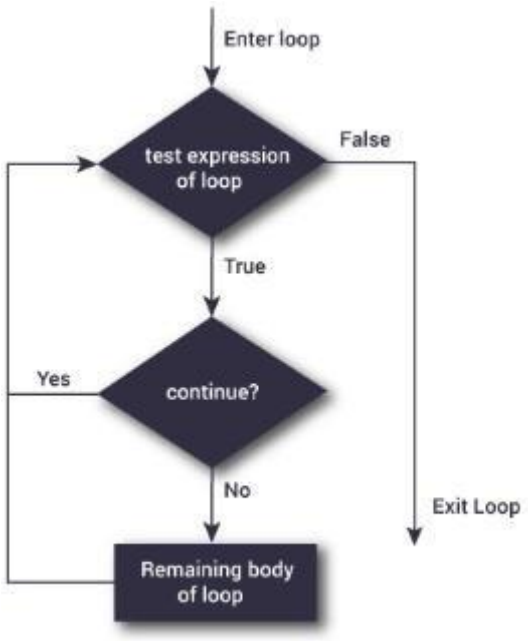
Current Letter : y

Current Letter : t

Continue:

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

Flowchart:



The following shows the working of break statement in for and while loop:

```
for var in sequence:
        # code inside for loop
        If condition:
            continue (if break condition satisfies it jumps to outside loop)
        # code inside for loop
# code outside for loop


while test expression

        # code inside while loop
        If condition:
            continue(if break condition satisfies it jumps to outside loop)
        # code inside while loop
# code 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:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/cont.py
s
t
r
n
g
The end

# program to display only odd numbers

for num in [20, 11, 9, 66, 4, 89, 44]:
```

```
    # Skipping the iteration when number is even
    if num%2 == 0:
        continue
    # This statement will be skipped for all even numbers
    print(num)
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/cont2.py
11
9
89
```

```
#_____________
for letter in "Python": # First Example
    if letter == "h":
        continue
    print("Current Letter :", letter)
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/con1.py
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
```

Pass:

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.
pass is just a placeholder for functionality to be added later.

Example:

```
sequence = {'p', 'a', 's', 's'}
for val in sequence:
    pass
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/f1.y.py
```

```
>>>
```

Similarily we can also write,

```
def f(arg): pass    # a function that does nothing (yet)

class C: pass       # a class with no methods (yet)
```

Strings:

A string is a group/ a sequence of characters. Since Python has no provision for arrays, we simply use strings. This is how we declare a string. We can use a pair of single or double quotes. Every string object is of the type 'str'.

```
>>> type("name")
<class 'str'>
>>> name=str()
>>> name
''
>>> a=str('This')
>>> a
'This'
>>> a=str(This)
>>> a[2]
'c'
>>> 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 we want

String slices:

A segment of a string is called a slice. Selecting a slice is similar to selecting a character:

Subsets of strings can be taken using the slice operator **([ ] and [:])** with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

Slice out substrings, sub lists, sub Tuples using index.

Syntax:[Start: stop: steps]

- Slicing will start from index and will go up to **stop** in **step** of steps.

- Default value of start is 0,
- Stop is last index of list
- And for step default is 1

For example 1−

```
str = 'Hello World!'

print str    # Prints complete string

print str[0] # Prints first character of the string

print str[2:5] # Prints characters starting from 3rd to 5th

print str[2:] # Prints string starting from 3rd character print

str * 2 # Prints string two times

print str + "TEST" # Prints concatenated string
```

Output:

```
Hello World!

H

llo

llo World!

Hello World!Hello World!

Hello World!TEST
```

Example 2:

```
>>> x='computer'

>>> x[1:4]

'omp'

>>> x[1:6:2]

'opt'
```

```
>>> x[3:]
'puter'

>>> x[:5]

'compu'

>>> x[-1]

'r'

>>> x[-3:]

'ter'

>>> x[:-2]

'comput'

>>> x[::-2]

'rtpo'

>>> x[::-1]

'retupmoc'
```

Immutability:

It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string.
For example:

```
>>> greeting='This college!'
>>> greeting[0]='n'

TypeError: 'str' object does not support item assignment
```

The reason for the error is that strings are **immutable**, which means we can't change an existing string. The best we can do is creating a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

Note: The plus **(+)** sign is the string concatenation operator and the asterisk **(*)** is the repetition operator

String functions and methods:

There are many methods to operate on String.

S.no	Method name	Description
1.	isalnum()	Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
2.	isalpha()	Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
3.	isdigit()	Returns true if string contains only digits and false otherwise.
4.	islower()	Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
5.	isnumeric()	Returns true if a string contains only numeric characters and false otherwise.
6.	isspace()	Returns true if string contains only whitespace characters and false otherwise.
7.	istitle()	Returns true if string is properly “titlecased” and false otherwise.
8.	isupper()	Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
9.	replace(old, new [, max])	Replaces all occurrences of old in string with new or at most max occurrences if max given.
10.	split()	Splits string according to delimiter str (space if not provided) and returns list of substrings;
11.	count()	Occurrence of a string in another string
12.	find()	Finding the index of the first occurrence of a string in another string
13.	swapcase()	Converts lowercase letters in a string to uppercase and viceversa
14.	startswith(str, beg=0,end=len(string))	Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.

Note:

All the string methods will be returning either true or false as the result

1. isalnum():

Isalnum() method returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.

Syntax:
String.isalnum()

Example:
>>> string="123alpha"
>>> string.isalnum() True

2. isalpha():

isalpha() method returns true if string has at least 1 character and all characters are alphabetic and false otherwise.

Syntax:
String.isalpha()

Example:
>>> string="nikhil"
>>> string.isalpha()
True

3. isdigit():

isdigit() returns true if string contains only digits and false otherwise.

Syntax:
String.isdigit()

Example:
>>> string="123456789"
>>> string.isdigit()
True

4. islower():

Islower() returns true if string has characters that are in lowercase and false otherwise.

Syntax:

String.islower()

Example:
>>> string="nikhil"
>>> string.islower()
True

5. isnumeric():

isnumeric() method returns true if a string contains only numeric characters and false otherwise.

Syntax:
String.isnumeric()

Example:
>>> string="123456789"
>>> string.isnumeric()
True

6. isspace():

isspace() returns true if string contains only whitespace characters and false otherwise.

Syntax:
String.isspace()

Example:
>>> string=" "
>>> string.isspace()
True

7. istitle()

istitle() method returns true if string is properly “titlecased”(starting letter of each word is capital) and false otherwise

Syntax:
String.istitle()

Example:

```
>>> string="Nikhil Is Learning"
>>> string.istitle()
True
```

8. isupper()

isupper() returns true if string has characters that are in uppercase and false otherwise.

Syntax:

String.isupper()

Example:

```
>>> string="HELLO"
>>> string.isupper()
True
```

9. replace()

replace() method replaces all occurrences of old in string with new or at most max occurrences if max given.

Syntax:

String.replace()

Example:

```
>>> string="Nikhil Is Learning"
>>> string.replace('Nikhil','Neha')
 'Neha Is Learning'
```

10.split()

split() method splits the string according to delimiter str (space if not provided)

Syntax:

String.split()

Example:

```
>>> string="Nikhil Is Learning"
>>> string.split()
```

['Nikhil', 'Is', 'Learning']

11.count()

count() method counts the occurrence of a string in another string Syntax:
String.count()

Example:

```
>>> string='Nikhil Is Learning'
>>> string.count('i')
3
```

12.find()

Find() method is used for finding the index of the first occurrence of a string in another string

Syntax:
String.find(„string")

Example:

```
>>> string="Nikhil Is Learning"
>>> string.find('k')
 2
```

13.swapcase()

converts lowercase letters in a string to uppercase and viceversa

Syntax:
String.find(„string")

Example:

```
>>> string="HELLO"
>>> string.swapcase()
 'hello'
```

14.startswith()

Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.

Syntax:

String.startswith(„string“)

Example:

```
>>> string="Nikhil Is Learning"
>>> string.startswith('N')
True
```

15.endswith()

Determines if string or a substring of string (if starting index beg and ending index end are given) ends with substring str; returns true if so and false otherwise.

Syntax:

String.endswith(„string“)

Example:

```
>>> string="Nikhil Is Learning"
>>> string.startswith('g')
True
```

String module:

This module contains a number of functions to process standard Python strings. In recent versions, most functions are available as string methods as well.

It’s a built-in module and we have to **import** it before using any of its constants and classes

Syntax: import string

Note:

help(string) --- gives the information about all the variables ,functions, attributes and classes to be used in string module.

Example:

```
import string
print(string.ascii_letters)
print(string.ascii_lowercase)
print(string.ascii_uppercase)
print(string.digits)
```

```
print(string.hexdigits)
#print(string.whitespace)
print(string.punctuation)
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/strrmodl.py
========================================
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
0123456789abcdefABCDEF
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

Python String Module Classes

Python string module contains two classes – Formatter and Template.

Formatter

It behaves exactly same as str.format() function. This class becomes useful if you want to subclass it and define your own format string syntax.

Syntax: from string import Formatter

Template

This class is used to create a string template for simpler string substitutions

Syntax: from string import Template

LISTS, TUPLES, DICTIONARIES

Lists: list operations, list slices, list methods, list loop, mutability, aliasing, cloning lists, list parameters, list comprehension; Tuples: tuple assignment, tuple as return value, tuple comprehension; Dictionaries: operations and methods, comprehension;

Lists, Tuples, Dictionaries:

List:

- It is a general purpose most widely used in data structures
- List is a collection which is ordered and changeable and allows duplicate members. (Grow and shrink as needed, sequence type, sortable).
- To use a list, you must declare it first. Do this using square brackets and separate values with commas.
- We can construct / create list in many ways.

Ex:

```
>>> list1=[1,2,3,'A','B',7,8,[10,11]]
>>> print(list1)
[1, 2, 3, 'A', 'B', 7, 8, [10, 11]]
----------
>>> x=list()
>>> x
[]
----------
>>> tuple1=(1,2,3,4)
>>> x=list(tuple1)
>>> x
[1, 2, 3, 4]
```

List operations:

These operations include indexing, slicing, adding, multiplying, and checking for membership

Basic List Operations:

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

Python Expression	Results	Description
len([1, 2, 3])	3	Length

[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input −

L = ['This', 'college', '!']

Python Expression	Results	Description
L[2]		Offsets start at zero
L[-2]	college	Negative: count from the right
L[1:]	['college', '!']	ing fetches sections

List slices:

>>> list1=range(1,6)

>>> list1

range(1, 6)

>>> print(list1)

range(1, 6)

```
>>> list1=[1,2,3,4,5,6,7,8,9,10]
>>> list1[1:]
[2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list1[:1]
[1]
>>> list1[2:5]
[3, 4, 5]
>>> list1[:6]
[1, 2, 3, 4, 5, 6]
>>> list1[1:2:4]
[2]
>>> list1[1:8:2]
[2, 4, 6, 8]
```

List methods:

The list data type has some more methods. Here are all of the methods of list objects:

- Del()
- Append()
- Extend()
- Insert()
- Pop()
- Remove()
- Reverse()
- Sort()

Delete: Delete a list or an item from a list

```
>>> x=[5,3,8,6]
>>> del(x[1])          #deletes the index position 1 in a list
>>> x
[5, 8, 6]
--------
```

```
>>> del(x)
>>> x                    # complete list gets deleted
```

Append: Append an item to a list

```
>>> x=[1,5,8,4]
>>> x.append(10)
>>> x
[1, 5, 8, 4, 10]
```

Extend: Append a sequence to a list.

```
>>> x=[1,2,3,4]
>>> y=[3,6,9,1]
>>> x.extend(y)
>>> x
[1, 2, 3, 4, 3, 6, 9, 1]
```

Insert: To add an item at the specified index, use the insert () method:

```
>>> x=[1,2,4,6,7]
>>> x.insert(2,10)   #insert(index no, item to be inserted)
>>> x
[1, 2, 10, 4, 6, 7]
----------
>>> x.insert(4,['a',11])
>>> x
[1, 2, 10, 4, ['a', 11], 6, 7]
```

Pop: The pop() method removes the specified index, (or the last item if index is not specified) or simply pops the last item of list and returns the item.

```
>>> x=[1, 2, 10, 4, 6, 7]
>>> x.pop()
```

```
7

>>> x

[1, 2, 10, 4, 6]

-----------------------

>>> x=[1, 2, 10, 4, 6]

>>> x.pop(2)

10

>>> x

[1, 2, 4, 6]
```

Remove: The **remove()** method removes the specified item from a given list.

```
>>> x=[1,33,2,10,4,6]

>>> x.remove(33)

>>> x
[1, 2, 10, 4, 6]

>>> x.remove(4)

>>> x

[1, 2, 10, 6]
```

Reverse: Reverse the order of a given list.

```
>>> x=[1,2,3,4,5,6,7]

>>> x.reverse()

>>> x

[7, 6, 5, 4, 3, 2, 1]
```

Sort: Sorts the elements in ascending order

```
>>> x=[7, 6, 5, 4, 3, 2, 1]

>>> x.sort()
```

```
>>> x

[1, 2, 3, 4, 5, 6, 7]

- - - - - - - - - - - - - - -

>>> x=[10,1,5,3,8,7]

>>> x.sort()

>>> x

[1, 3, 5, 7, 8, 10]
```

List loop:

Loops are control structures used to repeat a given section of code a certain number of times or until a particular condition is met.

Method #1: For loop

```
#list of items

list = ['M','R','C','E','T']
i = 1

#Iterating over the list
for item in list:
  print ('college ',i,' is ',item)
  i = i+1
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/lis.py
college 1 is  M
college  2  is  R
college  3  is  C
college  4  is  E
college 5 is  T
```

Method #2: For loop and range()

In case we want to use the traditional for loop which iterates from number x to number y.

```
# Python3 code to iterate over a list
list = [1, 3, 5, 7, 9]
```

```
# getting length of list
length = len(list)

# Iterating the index
# same as 'for i in range(len(list))'
for i in range(length):
    print(list[i])
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/listlooop.py
1
3
5
7
9
```

Method #3: using while loop

```
# Python3 code to iterate over a list
list = [1, 3, 5, 7, 9]

# Getting length of list
length = len(list)
i = 0

# Iterating using while loop
while i < length:
    print(list[i])
    i += 1
```

Mutability:

A mutable object can be changed after it is created, and an immutable object can't.

Append: Append an item to a list

```
>>> x=[1,5,8,4]

>>> x.append(10)

>>> x

[1, 5, 8, 4, 10]
```

Extend: Append a sequence to a list.

```
>>> x=[1,2,3,4]

>>> y=[3,6,9,1]

>>> x.extend(y)

>>> x
```

Delete: Delete a list or an item from a list

```
>>> x=[5,3,8,6]

>>> del(x[1])            #deletes the index position 1 in a list

>>> x

[5, 8, 6]
```

Insert: To add an item at the specified index, use the insert () method:

```
>>> x=[1,2,4,6,7]

>>> x.insert(2,10)   #insert(index no, item to be inserted)
>>> x

[1, 2, 10, 4, 6, 7]

----------

>>> x.insert(4,['a',11])

>>> x

[1, 2, 10, 4, ['a', 11], 6, 7]
```

Pop: The pop() method removes the specified index, (or the last item if index is not specified) or simply pops the last item of list and returns the item.

```
>>> x=[1, 2, 10, 4, 6, 7]

>>> x.pop()

7

>>> x

[1, 2, 10, 4, 6]

----------
```

```
>>> x=[1, 2, 10, 4, 6]

>>> x.pop(2)

10

>>> x

[1, 2, 4, 6]
```

Remove: The **remove()** method removes the specified item from a given list.

```
>>> x=[1,33,2,10,4,6]

>>> x.remove(33)

>>> x

[1, 2, 10, 4, 6]
>>> x.remove(4)

>>> x

[1, 2, 10, 6]
```

Reverse: Reverse the order of a given list.

```
>>> x=[1,2,3,4,5,6,7]

>>> x.reverse()

>>> x

[7, 6, 5, 4, 3, 2, 1]
```

Sort: Sorts the elements in ascending order

```
>>> x=[7, 6, 5, 4, 3, 2, 1]

>>> x.sort()

>>> x

[1, 2, 3, 4, 5, 6, 7]

----------

>>> x=[10,1,5,3,8,7]
```

```
>>> x.sort()

>>> x

[1, 3, 5, 7, 8, 10]
```

Aliasing:

1. An alias is a second name for a piece of data, often easier (and more useful) than making a copy.
2. If the data is immutable, aliases don't matter because the data can't change.
3. But if data can change, aliases can result in lot of hard – to – find bugs.
4. Aliasing happens whenever one variable's value is assigned to another variable.

For ex:

```
a = [81, 82, 83]
b = [81, 82, 83]
print(a == b)
print(a is b)
b = a
print(a == b)
print(a is b)
b[0] = 5
print(a)
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/alia.py
True
False
True
True
[5, 82, 83]
```

Because the same list has two different names, a and b, we say that it is **aliased**. Changes made with one alias affect the other. In the example above, you can see that a and b refer to the same list after executing the assignment statement b = a.

Cloning Lists:

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called cloning, to avoid the ambiguity of the word copy.

The easiest way to clone a list is to use the slice operator. Taking any slice of a creates a new list. In this case the slice happens to consist of the whole list.

Example:

```
a = [81, 82, 83]

b = a[:]       # make a clone using slice

print(a == b)

print(a is b)

b[0] = 5

print(a)

print(b)
```

Output:

C:/Users//AppData/Local/Programs/Python/Python38-32/pyyy/clo.py

```
True
False
[81, 82, 83]
[5, 82, 83]
```

Now we are free to make changes to b without worrying about a

List comprehension:

List:

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

```
>>> list1=[]

>>> for x in range(10):
```

```
        list1.append(x**2)

>>> list1

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

(or)

This is also equivalent to

```
>>> list1=list(map(lambda x:x**2, range(10)))

>>> list1

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

(or)

Which is more concise and redable.

```
>>> list1=[x**2 for x in range(10)]

>>> list1

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Similarily some examples:

```
>>> x=[m for m in range(8)]
>>> print(x)
[0, 1, 2, 3, 4, 5, 6, 7]

>>> x=[z**2 for z in range(10) if z>4]
>>> print(x)
[25, 36, 49, 64, 81]

>>> x=[x ** 2  for x in range (1, 11)   if x % 2 == 1]
>>> print(x)
[1, 9, 25, 49, 81]

>>> a=5
```

```
>>> table = [[a, b, a * b] for b in range(1, 11)]
>>> for i in table:
        print(i)
```

```
[5, 1, 5]
[5, 2, 10]
[5, 3, 15]
[5, 4, 20]
[5, 5, 25]
[5, 6, 30]
[5, 7, 35]
[5, 8, 40]
[5, 9, 45]
[5, 10, 50]
```

Tuples:

A tuple is a collection which is ordered and unchangeable. In Python tuples are written with round brackets.

- Supports all operations for sequences.
- Immutable, but member objects may be mutable.
- If the contents of a list shouldn't change, use a tuple to prevent items from accidently being added, changed, or deleted.
- Tuples are more efficient than list due to python's implementation.

We can construct tuple in many ways:

```
X=()     #no item tuple
X=(1,2,3)
X=tuple(list1)
X=1,2,3,4
```

Example:

```
>>> x=(1,2,3)
>>> print(x)
(1, 2, 3)
>>> x
(1, 2, 3)
----------
```

```
>>> x=()
>>> x
()
------------------
>>> x=[4,5,66,9]
>>> y=tuple(x)
>>> y
(4, 5, 66, 9)
--------------------
>>> x=1,2,3,4
>>> x
(1, 2, 3, 4)
```

Some of the operations of tuple are:

- Access tuple items
- Change tuple items
- Loop through a tuple
- Count()
- Index()
- Length()

Access tuple items: Access tuple items by referring to the index number, inside square brackets

```
>>> x=('a','b','c','g')
>>> print(x[2])
c
```

Change tuple items: Once a tuple is created, you cannot change its values. Tuples are unchangeable.

```
>>> x=(2,5,7,'4',8)
>>> x[1]=10
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    x[1]=10
TypeError: 'tuple' object does not support item assignment
>>> x
(2, 5, 7, '4', 8)     # the value is still the same
```

Loop through a tuple: We can loop the values of tuple using for loop

```
>>> x=4,5,6,7,2,'aa'
>>> for i in x:
        print(i)


4
5
6
7
2
aa
```

Count (): Returns the number of times a specified value occurs in a tuple

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> x.count(2)
4
```

Index (): Searches the tuple for a specified value and returns the position of where it was found

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> x.index(2)
1
```

(Or)

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> y=x.index(2)
>>> print(y)
1
```

Length (): To know the number of items or values present in a tuple, we use len().

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> y=len(x)
>>> print(y)
12
```

Tuple Assignment

Python has tuple assignment feature which enables you to assign more than one variable at a time. In here, we have assigned tuple 1 with the college information like college name, year, etc. and another tuple 2 with the values in it like number (1, 2, 3… 7).

For Example,

Here is the code,

- >>> tup1 = ('This', 'eng college','2004','cse', 'it','csit');
- >>> tup2 = (1,2,3,4,5,6,7);
- >>> print(tup1[0])
- This
- >>> print(tup2[1:4])
- (2, 3, 4)

Tuple 1 includes list of information of This

Tuple 2 includes list of numbers in it

We call the value for [0] in tuple and for tuple 2 we call the value between 1 and 4

Run the above code- It gives name This for first tuple while for second tuple it gives number (2, 3, 4)

Tuple as return values:

A Tuple is a comma separated sequence of items. It is created with or without (). Tuples are immutable.

A Python program to return multiple values from a method using tuple

```
# This function returns a tuple
def fun():
    str = "This college"
    x   = 20
    return str, x; # Return tuple, we could also
                   # write (str, x)
 # Driver code to test above method
str, x = fun() # Assign returned tuple
print(str)
```

```
print(x)
```

Output:

```
C:/Users//AppData/Local/Programs/Python/Python38-32/tupretval.py
This college
20
```

Tuple comprehension:

Tuple Comprehensions are special: The result of a tuple comprehension is special. You might expect it to produce a tuple, but what it does is produce a special "generator" object that we can iterate over.

For example:

```
>>> x = (i for i in 'abc') #tuple comprehension
>>> x
<generator object <genexpr> at 0x033EEC30>

>>> print(x)
<generator object <genexpr> at 0x033EEC30>
```

You might expect this to print as ('a', 'b', 'c') but it prints as <generator object <genexpr> at 0x02AAD710> The result of a tuple comprehension is not a tuple: it is actually a generator. The only thing that you need to know now about a generator now is that you can iterate over it, but ONLY ONCE.

So, given the code

```
>>> x = (i for i in 'abc')
>>> for i in x:
        print(i)

a
b
c
```

Create a list of 2-tuples like (number, square):

```
>>> z=[(x, x**2) for x in range(6)]
>>> z
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

Set:

Similarly to list comprehensions, set comprehensions are also supported:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}

>>> x={3*x for x in range(10) if x>5}
>>> x
{24, 18, 27, 21}
```

Dictionaries:

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

- Key-value pairs
- Unordered

We can construct or create dictionary like:

X={1:'A',2:'B',3:'c'}
X=dict([('a',3) ('b',4)]
X=dict('A'=1,'B' =2)

Example:

```
>>> dict1 = {"brand":"This","model":"college","year":2004}
>>> dict1
{'brand': 'This', 'model': 'college', 'year': 2004}
```

Operations and methods:

Methods that are available with dictionary are tabulated below. Some of them have already been used in the above examples.

Method	Description
clear()	Remove all items form the dictionary.

copy()	Return a shallow copy of the dictionary.
fromkeys(seq[, v])	Return a new dictionary with keys from seq and value equal to v (defaults to None).
get(key[,d])	Return the value of key. If key doesnot exit, return d (defaults to None).
items()	Return a new view of the dictionary's items (key, value).
keys()	Return a new view of the dictionary's keys.
pop(key[,d])	Remove the item with key and return its value or d if key is not found. If d is not provided and key is not found, raises KeyError.
popitem()	Remove and return an arbitary item (key, value). Raises KeyError if the dictionary is empty.
setdefault(key[,d])	If key is in the dictionary, return its value. If not, insert key with a value of d and return d (defaults to None).
update([other])	Update the dictionary with the key/value pairs from other, overwriting existing keys.
values()	Return a new view of the dictionary's values

Below are some dictionary operations:

To access specific value of a dictionary, we must pass its key,

```
>>> dict1 = {"brand":"This","model":"college","year":2004}
>>> x=dict1["brand"]
>>> x
'This'
```

To access keys and values and items of dictionary:

```
>>> dict1 = {"brand":"This","model":"college","year":2004}
>>> dict1.keys()
dict_keys(['brand', 'model', 'year'])
>>> dict1.values()
dict_values(['This', 'college', 2004])
>>> dict1.items()
dict_items([('brand', 'This'), ('model', 'college'), ('year', 2004)])
```

```
>>> for items in dict1.values():
        print(items)


This
college
2004

>>> for items in dict1.keys():
        print(items)


brand
model
year

>>> for i in dict1.items():
        print(i)


('brand', 'This')
('model', 'college')
('year', 2004)
```

Some more operations like:

- Add/change

- Remove
- Length
- Delete

Add/change values: You can change the value of a specific item by referring to its key name

```
>>> dict1 = {"brand":"This","model":"college","year":2004}
>>> dict1["year"]=2005
>>> dict1
{'brand': 'This', 'model': 'college', 'year': 2005}
```

Remove(): It removes or pop the specific item of dictionary.

```
>>> dict1 = {"brand":"This","model":"college","year":2004}
>>> print(dict1.pop("model"))
college
>>> dict1
{'brand': 'This', 'year': 2005}
```

Delete: Deletes a particular item.

```
>>> x = {1:1, 2:4, 3:9, 4:16, 5:25}
>>> del x[5]
>>> x
```

Length: we use len() method to get the length of dictionary.

```
>>>{1: 1, 2: 4, 3: 9, 4: 16}
{1: 1, 2: 4, 3: 9, 4: 16}
>>> y=len(x)
>>> y
4
```

Iterating over (key, value) pairs:

```
>>> x = {1:1, 2:4, 3:9, 4:16, 5:25}
>>> for key in x:
        print(key, x[key])


1 1
2 4
3 9
```

```
4 16
5 25
>>> for k,v in x.items():
        print(k,v)


1 1
2 4
3 9
4 16
5 25
```

List of Dictionaries:

```
>>> customers = [{"uid":1,"name":"John"},
   {"uid":2,"name":"Smith"},
        {"uid":3,"name":"Andersson"},
         ]
>>> >>> print(customers)
[{'uid': 1, 'name': 'John'}, {'uid': 2, 'name': 'Smith'}, {'uid': 3, 'name': 'Andersson'}]

## Print the uid and name of each customer
>>> for x in customers:
        print(x["uid"], x["name"])


1 John
2 Smith
3 Andersson

## Modify an entry, This will change the name of customer 2 from Smith to Charlie
>>> customers[2]["name"]="charlie"
>>> print(customers)
[{'uid': 1, 'name': 'John'}, {'uid': 2, 'name': 'Smith'}, {'uid': 3, 'name': 'charlie'}]

## Add a new field to each entry

>>> for x in customers:
        x["password"]="123456" # any initial value


>>> print(customers)
```

```
[{'uid': 1, 'name': 'John', 'password': '123456'}, {'uid': 2, 'name': 'Smith', 'password':
'123456'}, {'uid': 3, 'name': 'charlie', 'password': '123456'}]

## Delete a field
>>> del customers[1]
>>> print(customers)
[{'uid': 1, 'name': 'John', 'password': '123456'}, {'uid': 3, 'name': 'charlie', 'password':
'123456'}]

>>> del customers[1]
>>> print(customers)
[{'uid': 1, 'name': 'John', 'password': '123456'}]

## Delete all fields

>>> for x in customers:
        del x["uid"]


>>> x
{'name': 'John', 'password': '123456'}
```

Comprehension:

Dictionary comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
>>> z={x: x**2 for x in (2,4,6)}
>>> z
{2: 4, 4: 16, 6: 36}

>>> dict11 = {x: x*x for x in range(6)}
>>> dict11
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```