

Narayana

PYTHON

Narayana

Python

By

Narayana



Index

1. Introduction to Python	----4
2. Scripting and Programming languages	----6
3. Features of Python	----7
4. Installing Python software	----10
5. What can we do by using Python	----11
6. Advantages and Disadvantages of Python	----11
7. Why should we learn Python	----12
8. How Python's Datatypes Compare to Other Languages	----13
9. Starting IDE	----14
10. The Interactive shell	----14
11. Python application or program development	----15
12. Python Identifier	----17
13. Python Keywords	----19
14. Help() function	----20
15. Indentation	----24
16. Quotations in Python	----26
17. Multiline statement	----26
18. Comments in Python	----27
19. Python Variables	----28
20. Escape Sequence	----33
21. First Python Program	----34
22. Reading data from keyboard	----35
23. Data types in python	----40
24. Type conversion functions	----43
25. Eval function	----47

Assignment - 1

26. String - Data structure	----55
27. List - Data structure	----97
28. Tuple – Data structure	----125
29. Set – Data structure	----148
30. Dictionary – Data structure	----168

Assignment - 2

31. Operators	----201
32. Conditional statements	----219
33. Iterative statements	----242
34. Python Functions	----264
35. Format function	----305
36. Lambda functions	----315
37. OOPS	----325
38. Reading and writing files	----355
39. File handling	----359

Assignment - 3

40. Exception Handling	----382
41. User defined exceptions	----394
42. Modules	----396
43. Unit testing	----411
44. Iterators	----415
45. Generators	----417
46. Decorators	----418
47. Multithreading	----421
48. Database connection	----424
49. Numpy	----431

Assignment – 4

Python FAQs

Introduction to PYTHON



Introduction:

1. Under the leadership of **Andrew Stuart Tanenbaum**, a group of employees developed Distributed Operating System.
2. Group of Employees used ABC scripting language to develop Distributed Operating System.
3. ABC scripting language is very simple and easy to learn and work.
4. **Guido Van Rossum** is a member in that group and he likes ABC scripting language very well as it is very simple and easy.
5. In Christmas holidays, Guido Van Rossum started developing a new language to be simpler and easy compare to ABC scripting language and all other existing languages.
6. Finally he developed a new language.
7. He likes "**Monty Python's Flying Circus**" English daily serial very well.
8. So he has taken the word 'PYTHON' from that serial and kept for his language.
9. So finally Guido Van Rossum developed PYTHON Scripting language at the National Research Institute for Mathematics and Computer Science in Netherlands in 1989 and it available to the public in 1991.
10. Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.
11. Python 1.0 was released in January 1994.
12. Python 2.0 was released In October 2000 and Python 2.7.11 is the latest edition of Python 2.
13. Meanwhile, Python 3.0 was released in December 2008.
14. Python is a general purpose high level programming language.
15. Python is recommended as first programming language for beginners.
16. Guido Van Rossum has developed Python language by taking almost all programming features from different languages.
17. The most of syntaxes in python have taken from C and ABC languages.

Scripting and Programming languages:

Scripting Language:

1. Scripting languages based applications are run by interpreter.
2. Scripting languages based applications are not required explicit compilation.
3. We can run these scripting language based language directly.
4. Examples of scripting languages are Shell Script, Python, Perl, Ruby, Power Shell.. these all languages are run by interpreter, that's why these all languages are called scripting languages.

Programming Languages:

1. Programming language based applications are run by compiler.
2. Programming language based applications are required explicit compilation.
3. Programming language based applications can't run directly without compilation.
4. Examples of Programming languages are C, C++, JAVA, .NET... these all programming languages are run by compiler. That's why these all languages are called Programing languages.

Note: Most of the people call Python is a scripting language because the way of developing Python applications and executing Python applications are similar to scripting languages.

What is Python?

1. Python is a simple, powerful and high level, interactive, object oriented scripting language.
2. Python is a general purpose and portable language.

Different languages used to develop Python

1. Procedural oriented programing language.....C.
2. Object oriented programming language.....C++, JAVA.
3. Scripting language.....Shell Script, Perl
4. Modular Programming language.....Modula-3

Features of Python Language:



1. Python is simple, easy and powerful language.

- a. The syntax of python is very simple and easy to remember, so anybody can easily remember the Python syntaxes without having any programming basics.
- b. Even non-technical persons also can learn or work with python language. So here developers not required focusing on syntaxes,
- c. It allows programmer to concentrate on the solution to the problem rather than the language itself.
- d. Reading a Python program feels almost like reading English, although very strict English. This pseudo-code nature of Python is one of its greatest strengths.
- e. This elegant syntax of Python language makes the developers to express their business logic in very less lines of code. So when developers write less code then automatically application development time, application cost and also application maintenance time also will be decreased.
- f. That's why Python is also called programmer-friendly language.
- g. Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.

2. Python is an Expressive language.

- a. Python language is more expressive. The sense of expressive is the code is easily understandable.

3. Python is Free, Open and Redistribution Language.

- a. Python is an example of FLOSS (Free/Libre and Open Source Software).
- b. **FLOSS** is a community, which itself is based on the concept of sharing knowledge. FLOSS are free for usage, modification and redistribution.

Examples of FLOSS are 1. Linux

- 2. Ubuntu
- 3. LibreOffice
- 4. Mozilla Firefox
- 5. Mono
- 6. Apache Web Server and 7. VLC Player

4. Python is an Interpretable language.

- a. Python is an interpreted language i.e. interpreter executes the python code line by line at a time. So we don't need to compile our program before executing it. This makes debugging easy and thus suitable for beginners.

5. Python is an extensible language.

- a. Python can completely integrate with components of Java and .Net and also can invoke libraries of C and C++. So here Python performs CROSS language operations
- b. If we need a critical piece of code to run very fast, you can code that part of your program in C or C++ and then use them from our Python program.

6. Python is very rich in Libraries.

- a. The Python is very rich in libraries. They help us to do various things involving regular expressions, documentation generation, unit testing, threading, databases, web browsers, CGI(Common Gateway Interface), ftp(File Transfer Protocol), email, XML, WAV files, cryptography, GUI(graphical user interfaces) using Tk(ToolKit), and also other system-dependent stuff. Remember, all this is always available wherever Python is installed.
- b. This is called the "batteries included" philosophy of Python.

7. Python is Oriented Programming language:

- a. Python supports **procedure-oriented programming** as well as **object-oriented programming**. In procedure-oriented languages.
- b. The program is built around procedures or functions which are nothing but reusable pieces of programs. In object-oriented languages.
- c. The program is built around objects which combine data and functionality. Python is very powerful in way of doing object-oriented programming, especially, when compared to languages like C++ or Java.

8. Python is a Portable Language:

- a. Due to its Open source nature, Python has been ported to many flat forms.
- b. All our Python programs will work on any platform without requiring any changes at all. So Python is also called Cross Platform language.
- c. We can use Python on Linux, Windows, Macintosh, Solaris, OS/2, Amiga, AROS, AS/400, BeOS, OS/390, z/OS, Palm OS(also called Garnet OS), QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE and PocketPC !

9. High-level Language:

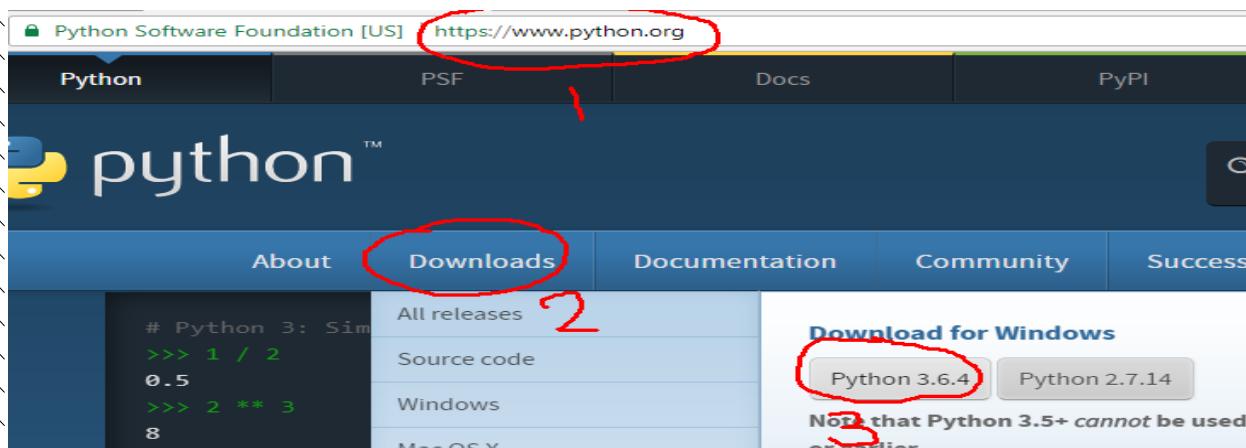
- a. When you write programs in Python, you never need to bother about the low-level details such as managing the memory used by your program, etc.
- b. The memory allocates dynamically when we run the application.

Some other features of Python language

1. Interactive language
2. Beginners language
3. Easy-to-learn
4. Easy-to-work
5. Easy-to-maintain
6. Supports all major commercial databases
7. supports GUI applications
8. Scalable
9. Supports multiple databases
10. Supports indentation

Installing Python3

Step1: Goto www.python.org and download latest version of python, like below



Step2: goto downloads folder and select downloaded python and run and install

Step3: set variable path like bellow

My computer → properties → Advanced system settings → environment variables →

Under user variables, create new variable with name PATH, like below

```
PATH= C:\Users\Narayana\AppData\Local\Programs\Python\Python36-32\Scripts\  
C:\Users\Narayana\AppData\Local\Programs\Python\Python36-32\
```

What can we do by using Python?

By using Python we can develop,

1. GUI Applications.
2. Data Analytics Applications.
3. Gaming Applications.
4. Scientific Applications.
5. Task Automations.
6. Network Applications.
7. Animation Applications.
8. Test Cases Applications, and so on..

Simply Python can be used to make games, do data analysis, control robot, hardware, create GUIs or even to create websites.

Advantages and Disadvantages of Python:

Advantages:

- Open source – Free and can edit source code
- Dynamically typed – No need to specify the type of variable before/after using it.
- Interpreted language – Opposite to compiled language.
- Objected oriented language
- Indentation – Whitespaces, No need of braces.
- Scripting language
- Easy to learn like normal English.
- Developed using ‘C’ language.
- Interface to Other programming languages. – Can include code from C, Java, .Net using Cython, Jython and IronPython interfaces respectively.
- Platform independent – Works on multi platforms.
- Less code compared to other languages.
- No strict typing on variables and containers.
- Used by Lot many MNCs and worlds top most organizations like Google, NASA, MST and Gaming apps etc
- Huge library.
- Multi-Threading and Multi processing.

Disadvantages:

- Slow compared to other programming languages like C,C++ or java
- Threading not fully implemented.
- All strings are not Unicode by default (Fixed in 3.0)
- Indentation – If mix tabs and spaces.
- Not using for mobile applications

Some Other Points about Python:

1. Python is a general-purpose language.
2. It has wide range of applications from Web development (like: Django and Bottle), scientific and mathematical computing (Orange, SymPy, NumPy) to desktop graphical user Interfaces (Pygame, Panda3D).
3. The syntax of the language is clean and length of the code is relatively short. It's fun to work in Python because it allows you to think about the problem rather than focusing on the syntax.

Why should we learn Python?

There are four reasons to learn Python,

1. ***Very Simple Syntax***

Python programming is very fun, it is very easy to understand and write the python code. The syntax feels very natural.

Eg:

```
a=10  
b=20  
sum=a+b  
print(sum)
```

so here if you even don't have any programming knowledge before, we can easily guess that this program is adding two numbers and print result.

2. ***Not overly strict***

You don't need to define the type of a variable in Python. Also, it's not necessary to add semicolon at the end of the statement.

Python enforces you to follow good practices (like proper indentation). These small things can make learning much easier for beginners.

3. ***Expressiveness of the language***

Python allows you to write programs having greater functionality with fewer lines of code. If we write 1000 lines of code by using any programming language, the same functionality can be achieved in Python with just 20% to 30% code of that programming language. This is just an example. You will be amazed how much you can do with Python once you learn the basics.

4. ***Great Community and support***

Python has a large supporting community. These are numerous active forums online which can be handy if you are stuck. Some of them are:

- a. Learn Python Subreddit
- b. Google Forum for Python
- c. Python Questions – stack overflow

How Python's Datatypes Compare to Other Languages

Statically typed language: A language in which types are fixed at compile time. Most statically typed languages enforce this by requiring you to declare all variables with their data types before using them. Java and C are statically typed languages.

Dynamically typed language: A language in which types are discovered at execution time; the opposite of statically typed. VBScript and Python are dynamically typed, because they figure out what type a variable is when you first assign it a value.

Strongly typed language: A language in which types are always enforced. Java and Python are strongly typed. If you have an integer, you can't treat it like a string without explicitly converting it.

Weakly typed language: A language in which types may be ignored; the opposite of strongly typed. VBScript is weakly typed. In VBScript, you can concatenate the string '12' and the integer 3 to get the string '123', then treat that as the integer 123, all without any explicit conversion.

So Python is both *dynamically typed* (because it doesn't use explicit datatype declarations) and *strongly Typed* (because once a variable has a datatype, it actually matters).

Starting IDLE

While the Python interpreter is the software that runs our Python programs, the interactive development environment (IDLE) software is where we will enter your programs, much like a word processor. Let's start IDLE now.

1. On Windows 7 or newer, click the Start icon in the lower-left corner of our screen, enter IDLE in the search box, and select IDLE (Python GUI).
2. On Windows XP, click the Start button and then select Programs → Python 3.6 → IDLE (Python GUI).
3. On Mac OS X, open the Finder window, click Applications, click Python 3.4, and then click the IDLE icon.
4. On Ubuntu, select Applications → Accessories → Terminal and then enter idle3. (You may also be able to click Applications at the top of the screen, select Programming, and then click IDLE 3.)

The Interactive Shell

No matter which operating system you're running, the IDLE window that first appears should be mostly blank except for text that looks something like this:

```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
```

```
Type "copyright", "credits" or "license()" for more information.
```

```
>>>
```

This window is called the interactive shell. A shell is a program that lets us type instructions into the computer, much like the Terminal or Command Prompt on OS X and Windows, respectively. Python's interactive shell lets us enter instructions for the Python interpreter software to run. The computer reads the instructions you enter and runs them immediately.

For example, enter the following into the interactive shell next to the

```
>>> prompt:  
>>> print('Hello, Python Narayana!')  
After we type that line and press enter, the interactive shell should display this in response:  
>>> print('Hello, Python Narayana!')  
Hello, Python Narayana
```

Python Application or program development

Python is considered an interpreted language because Python programs are executed by an interpreter. There are two ways to use the interpreter:

1. **Interactive mode**
2. **Script mode.**

In **interactive mode**, command line shell, we type Python programs and the interpreter prints the result:

```
>>> 10+20  
30
```

The chevron, `>>>`, is the prompt, that is used by interpreter to indicate that it is ready. So that's why when we enter `10+20` and click enter then immediately interpreter return 30.

This interactive mode is not used to develop any business applications. It is just used to test the features of Python

In script mode, we write python program in a file and use the interpreter to execute the content of that file. The extension of this file is .py.

Here we can write our Python program in any of the editors or IDE's,
Different **editors** used for Python program are Notepad, Notepad++, EditPlus, nano, gedit, IDL..
Different **IDE's** used for Python program are Pycharm, Eric, Eclipse, Pyscripter, Netbeans...

This script mode is used to develop business applications.

Eg:

Open Python → File → New File,

Write your Python program herein the new file and save the file with an extension .py.

Like

```
A=10  
B=5  
print(A+B)  
print(A+B)
```

save this file with FirstProgram.py

open the file FirstProgram.py, click on run tab and then run module or press F5.

Now we can see the output in python shell

Identifiers:

A Python identifier is a name used to identify a variable, function, class, module, or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9).

Python does not allow punctuation characters such as @, \$, and % within identifiers. Python is a case sensitive programming language. Thus **Application** and **application** are two different identifiers in Python.

Any identifier starting with two leading underscores indicates a strong private identifier.

1. An identifier should not start with digit but it can contain digits, like

```
>>> 12ab=100      SyntaxError: invalid syntax  
>>> ab12=100
```

2. An identifier can contain underscores(single, double and also triple)

```
>>> _total=10  
>>> to_tal=20  
>>> total_=30  
>>> _to_tal_=40  
>>> __total='Python'  
>>> total__='Narayana'  
>>> to__tal='Nani'  
>>> ___total=40  
>>> ___to__tal_=100
```

3. An identifier must not contain special characters(except underscore)

```
>>> @total=100  SyntaxError: invalid syntax  
>>> to@tal=200  SyntaxError: can't assign to operator  
>>> to$tal=200  SyntaxError: invalid syntax  
>>> total%=200  NameError: name 'total' is not defined  
>>> total!=10      NameError: name 'total' is not defined  
>>> !total=10      SyntaxError: invalid syntax  
>>> total*=10      NameError: name 'total' is not defined
```

```
>>> &to_tal=200 SyntaxError: invalid syntax  
>>> *total=20           SyntaxError: starred assignment target must be in a list or tuple
```

4. Identifiers are case-sensitive, so Python language is a case sensitive language

```
>>> a=100  
>>> A=200  
>>> print(a)      100  
>>> print(A)      200
```

```
>>> var='python'  
>>> print(Var)    NameError: name 'Var' is not defined
```

5. Keywords or reserved words can't be used as identifiers

```
>>> if=100        SyntaxError: invalid syntax  
>>> return='Python' SyntaxError: invalid syntax  
>>> def='dev'     SyntaxError: invalid syntax  
>>> del=200       SyntaxError: invalid syntax
```

Keywords:

The keywords in Python are having some predefined functionality. These keywords may not be used as constant or variable or any other identifier names.

If we use `help('keywords')` then interpreter displays all list of available keywords in Python.

And	Exec	Not
Assert	Finally	Or
Break	For	Pass
Class	From	Print
Continue	Global	Raise
Def	If	Return
Del	Import	Try
Elif	In	While
Else	Is	With
Except	Lambda	Yield

Or

```
>>> import keyword  
>>> keyword.kwlist  
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except',  
'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return',  
'try', 'while', 'with', 'yield']
```

Python contains totally 33 keywords. Out of 33, only True, False and None are capitalized

Help()

This function is used to get python documentation of a specific python keywords or variables

help() takes either one or none arguments.

If we want to give one argument then it should be either python keyword or a variables.

if we specify any python keyword as an argument to the help(), then it will display python documentation of that specified keyword.

eg1:

```
>>> help('if')
The "if" statement
*****
```

The "if" statement is used for conditional execution:

```
if_stmt ::= "if" expression ":" suite
          ("elif" expression ":" suite )*
          ["else" ":" suite]
```

It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true (see section Boolean operations for the definition of true and false); then that suite is executed (and no other part of the "if" statement is executed or evaluated). If all expressions are false, the suite of the "else" clause, if present, is executed.

Related help topics: TRUTHVALUE

eg2:

```
>>> help('return')
The "return" statement
*****
```

```
return_stmt ::= "return" [expression_list]
```

"return" may only occur syntactically nested in a function definition, not within a nested class definition.

If an expression list is present, it is evaluated, else "None" is substituted.

"return" leaves the current function call with the expression list (or "None") as return value.

When "return" passes control out of a "try" statement with a "finally" clause, that "finally" clause is executed before really leaving the function.

In a generator function, the "return" statement indicates that the generator is done and will cause "StopIteration" to be raised. The returned value (if any) is used as an argument to construct "StopIteration" and becomes the "StopIteration.value" attribute.

In an asynchronous generator function, an empty "return" statement indicates that the asynchronous generator is done and will cause "StopAsyncIteration" to be raised. A non-empty "return" statement is a syntax error in an asynchronous generator function.

Related help topics: FUNCTIONS

eg3:

```
>>> help('elif')
The "if" statement
*****

```

The "if" statement is used for conditional execution:

```
if_stmt ::= "if" expression ":" suite
          ("elif" expression ":" suite )*
          ["else" ":" suite]
```

It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true (see section Boolean operations for the definition of true and false); then that suite is executed (and no other part of the "if" statement is executed or evaluated).

If all expressions are false, the suite of the "else" clause, if present, is executed.

Related help topics: TRUTHVALUE

if we specify any variable name as an argument in the help(), then also it displays the entire python documentation of that specified variable

eg1:

```
>>> a=10
>>> help(a)
Help on int object:
```

```
class int(object):
    | int(x=0) -> integer
    | int(x, base=10) -> integer
    |
    | Convert a number or string to an integer, or return 0 if no arguments
    | are given. If x is a number, return x.__int__(). For floating point
    | numbers, this truncates towards zero.
    |
    | If x is not a number or if base is given, then x must be a string,
    | bytes, or bytearray instance representing an integer literal in the
    | given base. The literal can be preceded by '+' or '-' and be surrounded
    | by whitespace. The base defaults to 10. Valid bases are 0 and 2-36.
    | Base 0 means to interpret the base from the string as an integer literal.
    |
    >>> int('0b100', base=0)
    4
    |
    |
#Here it displays all methods which are possible on variable 'a'
```

Now we can directly enter the keywords to get python documentation

Eg1:

```
help> pass
The "pass" statement
*****
pass_stmt ::= "pass"

"pass" is a null operation --- when it is executed, nothing happens.
It is useful as a placeholder when a statement is required
syntactically, but no code needs to be executed, for example:

def f(arg): pass  # a function that does nothing (yet)
class C: pass    # a class with no methods (yet)
```

Eg2:

help> del

The "del" statement

del_stmt ::= "del" target_list

Deletion is recursively defined very similar to the way assignment is defined. Rather than spelling it out in full details, here are some hints.

Deletion of a target list recursively deletes each target, from left to right.

Deletion of a name removes the binding of that name from the local or global namespace, depending on whether the name occurs in a "global" statement in the same code block. If the name is unbound, a "NameError" exception will be raised.

Deletion of attribute references, subscriptions and slicings is passed to the primary object involved; deletion of a slicing is in general equivalent to assignment of an empty slice of the right type (but even this is determined by the sliced object).

Changed in version 3.2: Previously it was illegal to delete a name from the local namespace if it occurs as a free variable in a nested block.

Related help topics: BASICMETHODS

We use quit to come out of help>to >>>

help> quit

You are now leaving help and returning to the Python interpreter.

If you want to ask for help on a particular object directly from the interpreter, you can type "help(object)". Executing "help('string')" has the same effect as typing a particular string at the help> prompt.

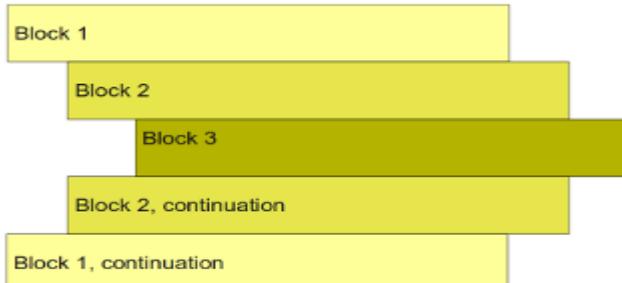
>>>

Indentation:

1. One of the most distinctive features of Python is its use of indentation to mark blocks of code.
2. Generally we use curly braces in C, C++ and JAVA languages but in python curly braces are not allowed to indicate block of code for class, function definition or flow control. Blocks of code are denoted by indentation.
3. Instead of curly braces { }, indentations are used to represents a block.

Syn:

```
Im_a_parents:  
  Im_a_child:  
    Im_a_grand_child  
  Im_another_child:  
Im_another_grand_child:
```



The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. Both blocks in this example are fine:

```
if True:  
    print ("This condition is True")  
else:  
    print ("This condition is False")
```

However, the second block in this example will generate an error because first print in the else part is not following indentation:

```
if True:  
    print ("Hello, Python programmer")  
    print ("This condition is True")  
else:
```

```
print ("Hello, Python programmer")
print ("This condition is False")
```

Eg1: #It is just a dummy example to write indentation

```
a=10
b=20

if a>b:
    print('Hi')
    print(a)
    print('you got a, right?')
if a==b:
    print(a)
    print(b)
    print('both are equal')
    if a<5:
        print(a)
        print('value of a is less then 5')
else:
    print(a,'and',b,'are not equal')
    print('different values')
else:
    print('hello')
    print(b)
    print('byeee...')
print('Thank you...!')
```

Output:

```
hello
20
byeee...
Thank you...!
```

Quotations:

Python accepts single ('), double ("") and triple (''' or '''''') quotes to denote string literals, as long as the same type of quote starts and ends the string.

Generally triple quotes are used to write the string across multiple lines. For example, all the following are legal:

```
word = 'word'  
sentence = "This is a sentence."  
paragraph = """This is a paragraph. It is  
made up of multiple lines and sentences."""
```

Multi-Line Statements:

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example:

```
total = item_one + \  
    item_two + \  
    item_three
```

Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example:

```
days = ['Monday', 'Tuesday', 'Wednesday',  
       'Thursday', 'Friday']
```

Some of the basic functions

Python supports so many functions. Some of those are, print(), type(), id()

Print(): This function displays the output on the screen.

Type(): this function checks the data type of specific variable.

`Id()`: this functions finds address of variable.

Comments:

Comments are used in programming to describe the purpose of the code. This helps you as well as other programmers to understand the intent of the code. Comments are completely ignored by compilers and interpreters.

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the physical line end are part of the comment, and the Python interpreter ignores them.

Python ignores comments, and we can use them to write notes or remind ourselves what the code is trying to do. Any text for the rest of the line following a hash mark (#) is part of a comment.

Sometimes, programmers will put a # in front of a line of code to temporarily remove it while testing a program. This is called commenting out code, and it can be useful when we're trying to figure out why a program doesn't work. We can remove the # later when we are ready to put the line back in.

Python also ignores the blank line after the comment. We can add as many blank lines to our program as we want. This can make our code easier to read, like paragraphs in a book.

```
#!/usr/bin/python

# First comment
print "Hello, Python!"; # second comment
```

This will produce following result:

```
Hello, Python!
```

A comment may be on the same line after a statement or expression:

```
name = "Parlapalli" # declaring variable name
```

You can comment multiple lines as follows:

```
# Python is easy.
# Python is simple, too.
# Python is powerful, too.
# These all are comments.
```

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.

We use these variable names to make program code read more than like English . If we didn't use good names for things in our software, we would get lost when we tried to read our code again.

Assigning Values to Variables:

We can assign single value to single variable or multiple values to multiple variables at a time.
And also we can assign single value to multiple variables.

Assigning single value to single variable:

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:

```
>>> m=10
>>> print(m)           10
>>> type(m)            <class 'int'>
>>> id(m)              1624729872

>>> n=20.4
>>> print(n)           20.4
>>> type(n)            <class 'float'>
>>> id(n)              48638592

>>> r='Narayana'
>>> print(r)           Narayana
```

```
>>> type(r) <class 'str'>
>>> id(r) 54591640

>>> k=2+5j
>>> print(k) (2+5j)
>>> type(k) <class 'complex'>
>>> id(k) 102020112

>>> b=True
>>> print(b) True
>>> type(b) <class 'bool'>
>>> id(b) 1610947664
```

Assigning multiple values to multiple variables

```
>>> a,b,c=10,10.5,'satya'

>>> print(a) 10
>>> type(a) <class 'int'>
>>> id(a) 1624729872

>>> print(b) 10.5
>>> type(b) <class 'float'>
>>> id(b) 49688512

>>> print(c) satya
>>> type(c) <class 'str'>
>>> id(c) 54580896
```

Here, number of variables must be equal to number of values.

If both are having different number of variables or values then interpreter will return error, like below

```
>>> a,b,c,d=1,2,3 ValueError: not enough values to unpack (expected 4, got 3)
>>> a,b,c,d=1,2,3,4,5 ValueError: too many values to unpack (expected 4)
```

Assigning same value to multiple variables

```
>>> x=y=z=10

>>> print(x)      10
>>> type(x)      <class 'int'>
>>> id(x)        1611129760

>>> print(y)      10
>>> type(y)      <class 'int'>
>>> id(y)        1611129760

>>> print(z)      10
>>> type(z)      <class 'int'>
>>> id(z)        1611129760
```

Some Examples on Variables:

Eg1:

```
my_name = 'Ramya'
my_age = 28 # not a lie
my_height = 74 # inches
my_weight = 75 # kgs
my_eyes = 'Blue'
my_teeth = 'White'
my_hair = 'black'

print ("Let's talk about %s." % my_name)
print ("She's %d inches tall." % my_height)
print ("She's %d kgs heavy." % my_weight)
print ("Actually that's not too heavy.")
print ("She's got %s eyes and %s hair." % (my_eyes, my_hair))
print ("Her teeth are usually %s depending on the coffee." % my_teeth)
print("Hi This is %s, If I add %d, %d, and %d I get %d." % (my_name,my_age, my_height, my_weight,
my_age + my_height + my_weight))
```

output:

Let's talk about Ramya.
She's 74 inches tall.
She's 75 kgs heavy.
Actually that's not too heavy.
She's got Blue eyes and black hair.
Her teeth are usually White depending on the coffee.
Hi This is Ramya, If I add 28, 74, and 75 I get 177.

Eg2:

```
cars = 100  
space_in_a_car = 4.0  
drivers = 30  
passengers = 90  
cars_not_driven = cars - drivers  
cars_driven = drivers  
carpool_capacity = cars_driven * space_in_a_car  
average_passengers_per_car = passengers / cars_driven  
  
print ("There are", cars, "cars available.")  
print ("There are only", drivers, "drivers available.")  
print ("There will be", cars_not_driven, "empty cars today.")  
print ("We can transport", carpool_capacity, "people today.")  
print ("We have", passengers, "to carpool today.")  
print ("We need to put about", average_passengers_per_car, "in each car.")
```

output:

There are 100 cars available.
There are only 30 drivers available.
There will be 70 empty cars today.
We can transport 120.0 people today.
We have 90 to carpool today.
We need to put about 3.0 in each car.

Eg3:

```
print( "Mary had a little lamb.")  
print ("Its fleece was white as %s." % 'snow')  
print("And everywhere that Mary went.")  
print( "." * 10)# what'd that do?  
  
end1 = "C"  
end2 = "h"  
end3 = "e"  
end4 = "e"  
end5 = "s"  
end6 = "e"  
end7 = "B"  
end8 = "u"  
end9 = "r"  
end10 = "g"  
end11 = "e"  
end12 = "r"  
  
# watch that comma at the end. try removing it to see what happens  
print( end1 + end2 + end3 + end4 + end5 + end6,)  
print (end7 + end8 + end9 + end10 + end11 + end12)
```

output:

Mary had a little lamb.
Its fleece was white as snow.
And everywhere that Mary went.
.....
Cheese
Burger

Eg4:

```
days = "Mon Tue Wed Thu Fri Sat Sun"  
months = "Jan\nFeb\nMar\nApr\nMay\nJun\nJul\nAug"  
  
print ("Here are the days: ", days)  
print("Here are the months: ", months)
```

Output:

Here are the days: Mon Tue Wed Thu Fri Sat Sun

Here are the months: Jan

Feb

Mar

Apr

May

Jun

Jul

Aug

Escape Sequences:

An escape character allow us use characters that are otherwise impossible to put into a string. An escape character consists of a backslash (\) followed by the character we want to add to the string. (Despite consisting of two characters, it is commonly referred to as a singular escape character.)

For example, the escape character for a single quote is \'. We can use this inside a string that begins and ends with single quotes.

This is the list of all the escape sequences Python supports. You may not use many of these, but memorize their format and what they do anyway. Also try them out in some strings to see if you can make them work.

<u>Escape</u>	<u>What it does..</u>
\\	Backslash (\)
\'	Single- quote ('')
\"	Double- quote (")

\a	ASCII bell (BEL)
\b	ASCII backspace (BS)
\f	ASCII formfeed (FF)
\n	ASCII linefeed (LF)
\N	{name} Character named name in the Unicode database (Unicode only)
\r	ASCII carriage return (CR)
\t	ASCII horizontal tab (TAB)
\uxxxx	Character with 16- bit hex value xxxx (Unicode only)
\Uxxxxxxxxx	Character with 32- bit hex valuexxxxxxxx (Unicode only)
\v	ASCII vertical tab (VT)
\ooo	Character with octal value oo
\xhh	Character with hex value hh

First Python Program

However, Python is one of the easiest languages to learn, and creating "Hello, Python Developer!" program is as simple as writing `print("Hello, Python Developer!")`. So, we are going to write a different program.

Program to Add Two Numbers:

```
>>># Add two numbers
>>>x = 10
>>>y= 20
>>>sum = x+y
>>>print(sum)
```

How this program executes?

Line 1: #Add two numbers

Any line starting with `#` in Python programming is a comment

Line 2: x=10

Here, x is a variable. You can store a value in a variable. Here, 3 is stored in this variable.

Line 3: y=20

Similarly, 20 stored in y variable.

Line 4: sum = x+y

The variables x and y are added using + operator. The result of addition is then stored in another variable sum.

Line 5: print(sum)

The print() function prints the output to the screen. In our case, it prints 30 on the screen.

Reading data from Keyboard:

1. Input to the program can come in various ways, for example from a database, from another computer, from mouse clicks and movements or from the internet.
2. Generally in most cases the input comes from the keyboard. For this purpose, Python provides the function **input()**.
3. The input of the user will be returned as a string without any changes.
4. If this raw input has to be transformed into another data type needed by the algorithm, we can use either **casting (type casting) functions** or the **eval()**.

Direct input from user (default data type is string)

Eg1:

```
>>> n=input('Enter Number: ')
Enter Number: 10
>>> print(n)
10
>>> type(n)
<class 'str'>
>>> id(n)
106802720
```

Eg2:

```
>>> b=input('Enter any float value: ')
Enter any float value: 10.6
>>> print(b)
10.6
>>> type(b)
<class 'str'>
>>> id(b)
106802752
```

Eg3:

```
>>> c=input('Enter any complex number: ')
Enter any complex number: 2+6j
>>> print(c)
2+6j
>>> type(c)
<class 'str'>
>>> id(c)
106802688
```

Eg4:

```
>>> d=input('Enter either True or False: ')
Enter either True or False: True
>>> print(d)
True
>>> type(d)
<class 'str'>
>>> id(d)
106802880
```

Converting type data by using casting function

```
>>> n=int(input('Enter Number: '))
Enter Number: 10
>>> print(n)
10
>>> type(n)
<class 'int'>
```

Converting type of data by using eval functions

```
>>> n1=eval(input('Enter Number: '))
Enter Number: 10
>>> print(n1)
10
>>> type(n1)
<class 'int'>
```

Eg:**In interactive mode:**

```
>>> name=input('What Is Your Name: ')
What Is Your Name: Narayana
>>> loc=input("What is Your Location: ")
What is Your Location: Guntur
>>> print(""" Hello """ + name +", How are You. How is your " +loc+ ".");
Hello Narayana, How are You. How is your Guntur.
```

In script mode:

Open file and write the following code and save the file.

```
name = input("What is your Name: ")
loc = input("What is your Location: ")
print('Hello ' + name + ', How are you. How is Your ' + loc + '.')
```

If we run the file, we can see the following in the python screen.

```
What is your Name: Narayana
What is your Location: Guntur
Hello Narayana, How are you. How is Your Guntur.
```

Difference between python2 and python3

In Python 2 has two versions of input functions, **input()** and **raw_input()**. The **input()** function treats the received data as string if it is included in quotes " or "", otherwise the data is treated as number.

In Python 3, **raw_input()** function is deprecated. Further, the received data is always treated as string.

In Python 2

```
>>> x=input('Enter any value:')
'Enter any value:100'                                #entered data is treated as number
>>> x
100
>>> x=input('Enter any value:')
'Enter any value:'10'                               #entered data is treated as string
>>> x
'10'
>>> x=raw_input("Enter any value:")
'Enter any value:10'                                #entered data is treated as string even without "
>>> x
'10'
>>> x=raw_input("Enter any value:")
```

```
'Enter any value:'10'           #entered data treated as string including "
>>> x                           "'10"

In Python 3

>>> x=input("Enter any value:")
'Enter any value:10

>>> x
'10'

>>> x=input("Enter any value:")
'Enter any value:'10'           #entered data treated as string with or without "
>>> x
"'10"

>>> x=raw_input("Enter any value:") # will result NameError
Traceback (most recent call last):
File "", line 1, in
x=raw_input("Enter any value:")
NameError: name 'raw_input' is not defined
```

Eg:

```
print('What is your age?') # ask for their age
myAge = input()
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

output:

What is your age?

28

You will be 29 in a year.

Execution Process:

```
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```



```
print('You will be ' + str(int('4') + 1) + ' in a year.')
```



```
print('You will be ' + str( 4 + 1 ) + ' in a year.')
```



```
print('You will be ' + 5 + ' in a year.')
```



```
print('You will be ' + '5' + ' in a year.')
```



```
print('You will be 5' + ' in a year.')
```



```
print('You will be 5 in a year.')
```

Data types

In

Python:

1. Data types are some of the keywords in any programming languages.
2. Data types are used to define what type of data that we are passing to the specific variable.
3. Generally variables will not store the values without having data types in other languages.
4. But in Python language, variables allow values without specifying data types also, because Python allocates the data types dynamically at run time when python application is running.
5. In Python language, data types are decided by interpreter based on the value that is given to variable by user.
6. Programmers should not specify the data type of variable when developing the program, if we specify explicitly then interpreter will throw error.

Python supports two types of data types,

1. Basic data types or fundamental data types
2. Composite data types or collections data types

Basic data types or fundamental data types

Python supports different types of basic data types,

1. Int
2. Float
3. Complex
4. Bool
5. str

Int data type:

If we assign integer value to any variable then interpreter decides that variable as int type variable,

Eg:

```
>>> a=10;  
>>> print(a)          10  
>>> type(a)         <class 'int'>  
>>> id(a)           1625909520
```

Float Data type

If we assign float value to any variable then interpreter decides that variable as Float Type variable.

Eg:

```
>>> b=10.5  
>>> print(b)          10.5  
>>> type(b)           <class 'float'>  
>>> id(b)             47460096
```

String Data Type

If we assign string value to any variable then interpreter decides that variable as String type variable.

Eg:

```
>>> c='Narayana'  
>>> print(c)          Narayana  
>>> type(c)           <class 'str'>  
>>> id(c)             52367056
```

Bool Data type

If we assign either True or False to any variable then interpreter decides that variable as Bool type variable.

Eg:

```
>>> d=True  
>>> print(d)          True  
>>> type(d)           <class 'bool'>  
>>> id(d)             1625727600
```

Complex Data type

If we assign complex number to any variable then interpreter makes that variable as Complex type variable.

Eg:

```
>>> e=3+4j  
>>> print(e)          (3+4j)  
>>> type(e)           <class 'complex'>  
>>> id(e)             47539552
```

Type Conversion Functions

We can convert datatype in two different ways:

1. type casting functions
2. eval function

Type casting functions

These type conversion functions are used to convert string type data into required types

1. **int()**: This int() function is used to convert into int data type.

Eg: Get two integers from the user and perform addition on those user values

Way-1

```
>>> a=input("Enter First Number: ")
>>> b=input("Enter Second Number: ")
>>> print (a)
>>> print (b)
>>> type(a)
>>> type(b)
>>> id(a)
>>> id(b)

>>> c=a+b
>>> print(c)
>>> type(c)
>>> id(c)

>>> x=int(a)
>>> y=int(b)
>>> print(x)
>>> print(y)
>>> type(x)
>>> type(y)
>>> id(x)
>>> id(y)
```

Enter First Number: 10
Enter Second Number: 20
10
20
<class 'str'>
<class 'str'>
52351712
6925376

#adding two str variables
1020
<class 'str'>
52351136

#converting str 'a' into int 'x'
#converting str 'b' into int 'y'
10
20
<class 'int'>
<class 'int'>
1625909520
1625909680

```
>>> z=x+y                                #adding two int variables
>>> print(z)                             30
>>> type(z)                            <class 'int'>
>>> id(z)                               1625909840
```

Way-2 (shorter way)

```
a=int(input("Enter First Number: "))
>>> b=int(input("Enter Second Number: "))
>>> print(a)                           Enter First Number: 10
>>> print(b)                           Enter Second Number: 20
>>> type(a)                            10
>>> type(b)                            20
>>> type(a)                            <class 'int'>
>>> type(b)                            <class 'int'>

>>> c=a+b

>>> print(c)                           30
>>> type(c)                            <class 'int'>
```

Way-3 (shortest way)

```
>>> print("The sum is " + str(int(input("Enter First Number: ")) + int(input("Enter Second Number: "))))
>>> Enter First Number: 10
>>> Enter Second Number: 20
>>> The sum is 30
```

2. **Float():** This float() conversion function is used to convert other types into float type.

Way-1

```
>>> a=input("Enter First Number: ")
>>> b=input("Enter Second Number")
>>> print (a)                           Enter First Number: 10.5
>>> print(b)                           Enter Second Number20.5
>>> type(a)                            10.5
>>> type(b)                            20.5
>>> type(a)                            <class 'str'>
>>> type(b)                            <class 'str'>
>>> id(a)                              52354240
>>> id(b)                              52354144
```

```
>>> c=a+b  
>>> print(c)          10.520.5  
>>> type(c)           <class 'str'>  
>>> id(c)              52367096  
  
>>> x=float(a)  
>>> y=float(b)  
>>> print(x)          10.5  
>>> print(y)          20.5  
>>> type(x)            <class 'float'>  
>>> type(y)            <class 'float'>  
>>> id(x)              47459712  
>>> id(y)              47460096  
  
>>> z=x+y  
>>> print(z)          31.0  
>>> type(z)            <class 'float'>  
>>> id(z)              46410368
```

Way-2 (shorter way)

```
>>> a=float(input("Enter First Number: "))      Enter First Number: 10.5  
>>> b=float(input("Enter Second Number"))        Enter Second Number2.3  
>>> print(a)                                     10.5  
>>> print(b)                                     2.3  
>>> type(a)                                      <class 'float'>  
>>> type(b)                                      <class 'float'>  
  
>>> c=a+b  
  
>>> print(c)                                     12.8  
>>> type(c)                                      <class 'float'>
```

Way-3 (shortest way)

```
>>> print(float(input("Enter First Number: "))+float(input("Enter Second Number: ")))  
>>> Enter First Number: 10.5  
>>> Enter Second Number: 20.5
```

```
>>> 31.0
```

3. **Complex()**: This complex() conversion function is used to convert string type data into Complex type.

Way-1

```
>>> a=input("Enter Number: ")          Enter Number: 2+3j
>>> print(a)                         2+3j
>>> type(a)                          <class 'str'>
>>> id(a)                            52351712
>>> x=complex(a)
>>> print(x)                         (2+3j)
>>> type(x)                          <class 'complex'>
>>> id(x)                            47539768
```

Way-2

```
>>> print(complex(input("Enter Number: ")))
>>> Enter Number: 4+6j
>>> (4+6j)
```

4. **Bool()**: This bool() conversion function is used to convert string type data into Boolean type.

Way-1

```
>>> a=input("Enter either True or False: ")
>>> print(a)                         Enter either True or False: True
>>> type(a)                          <class 'str'>
>>> id(a)                            52354272
>>>
>>> x=bool(a)
>>> print(x)                         True
>>> type(x)                          <class 'bool'>
>>> id(x)                            1625727600
```

Way-2

```
>>> print(bool(input("Enter Either True or False: ")))
>>> Enter Either True or False: True
>>> True
```

Eval Function

This function is also used to convert the data into required type. But generally this function converts the type of the variable based on the value.

Converting int value:

```
>>> a=eval(input('enter value for a: '))
enter value for a: 10
>>> print(a)                      10
>>> type(a)                       <class 'int'>
```

Converting float value

```
>>> b=eval(input('enter value for b: '))
enter value for b: 10.6
>>> print(b)                      10.6
>>> type(b)                        <class 'float'>
```

Converting string value

```
>>> c=eval(input("enter value for c: "))
enter value for c: 'narayana'
>>> print(c)                      Narayana
>>> type(c)                        <class 'str'>
```

Converting complex value

```
>>> d=eval(input('enter value for d: '))
enter value for d: 2+5j
>>> print(d)                      (2+5j)
>>> type(d)                        <class 'complex'>
```

Converting bool value

```
>>> e=eval(input('enter value for e: '))
enter value for e:True
```

```
>>> print(e)  
True  
>>> type(e)  
<class 'bool'>
```

Assignment - 1

1. Who is the father of Python?

Ans:

2. What is the language used to develop Distributed Operating System?

Ans:

3. Why did Guido Van Rossum keep the name 'PYTHON' to his language?

Ans:

4. When was the Python developed?

Ans:

5. When did Python available to public?

Ans:

6. What is a Python?

Ans:

7. What are the main features of Python?

Ans:

8. What are the differences between programming and scripting languages?

Ans:

9. Is Python programming language or scripting language? Why?

Ans:

10. What are the different types of languages used to develop the Python language?

Ans:

11. Is Python portable language or not? If yes then why?

Ans:

12. If we use Python language to develop a project, will it take less time or more time to develop the project compare to other compiler languages? If no then why?

Ans:

13. Is Python free source or needs to purchase?

Ans:

14. What are the different top companies using Python?

Ans:

15. Why Python is also called CROSS Plat Form language?

Ans:

16. What are the different types of OSs where we can run python?

Ans:

17. Why Python is also called interpretable language?

Ans:

18. Why debugging of Python program is very easy for developer?

Ans:

19. Give me any four reasons to learn/work on Python?

Ans:

20. Do developers need to focus more on Python Syntax? If no then Why?

Ans:

21. Do we need to think about memory management while application is developing?

Ans:

22. When memory will be allocated to variables in python language?

Ans:

23. What are the different types of modes to develop python application?

Ans:

24. What are the differences between interactive mode and script mode?

Ans:

25. What is the name of >>> symbol in python?

Ans:

26. What is indentation in python?

Ans:

27. What is a variable and what is the purpose of variable in python?

Ans:

28. Who will decide the type of variable in Python language? Developer or interpreter?

Ans:

29. What are the different types of quotes supported in python?

Ans:

30. What is the purpose of comments in any programming language?

Ans:

31. Can we write our programing code after comment in the same physical line? If no, why?

Ans:

32. Can we write comment after our programing code in same physical line?

Ans:

33. How to write comment in multiple lines?

Ans:

34. What is identifier in python? Can a single identifier contain combination of upper case, lower case, underscore and number?

Ans:

35. Is python case-sensitive language or not? If yes, give one example?

Ans:

36. Interpreter decides the type of variable in python then how can a programmer know the type of variable?

Ans:

37. How to see the value in the variable? And also the address of variable in python?

Ans:

38. How to assign multiple heterogeneous values to multiple variables? Give one example?

Ans:

39. How to assign single value to multiple variables?

Ans:

40. How to give input to the program from key board?

Ans:

41. What is the default data type of input()'s value?

Ans:

42. Can we convert string type data to our required data type? Different ways?

Ans:

43. What are the different types of type conversion functions?

Ans:

44. What is eval()?

Ans:

45. What is the difference between type conversion functions and eval()?

Ans:

46. How to read values from user? Give one example?

Ans:

47. What is the purpose of data type in any language?

Ans:

48. Can variable allow the value without specifying its type in python language?

Ans:

49. Can a programmer specify type of variable explicitly while developing application?

Ans:

50. On what basis, interpreter decides the type variable in python language?

Ans:

Composite Datatypes or Collections or Data Structures

Python supports different types of data structures:

1. String
2. List
3. Tuple
4. Set
5. Dictionary

STRING DATA STRUCTURE

1. A sequence of characters is called String.
2. Python supports str data type to represent string type data.
3. String objects are immutable objects that mean we can't modify the existing string object.
4. Insertion order is preserved in string objects.
5. A string allows multiple duplicate characters also.
6. Every character in the string object is represented with unique index.
7. Python supports both forward and backward indexes.
8. Forward index starts with 0 and negative index starts with -1.
9. Python string supports both concatenation and multiplication of string objects.
10. 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.

Eg: creating a string with single quote

```
>>> name='Narayana'  
>>> print(name)           Narayana
```

Eg: creating a string with double quotes

```
>>> name="Narayana"  
>>> print(name)           Narayana
```

Eg: creating a string with triple quotes

```
>>> name="""Narayana"""  
>>> print(name)           Narayana
```

(or)

```
>>> name=""""""Narayana"""""  
>>> print(name)           Narayana
```

Using triple quotes for multiple line comments

```
name="""Hello, Welcome to  
world of PYTHON.  
Working With PYTHON is very fun."""
```

```
print(name)
```

```
Hello, Welcome to  
world of PYTHON.  
Working With PYTHON is very fun.
```

1. We can access individual characters using **indexing** and a range of characters using **slicing**.
2. Index starts from 0. If we try to access a character out of index range then interpreter will raise an `IndexError`.
3. The index must be an integer. If we try to access the data with non-integer indexes values then interpreter raises `TypeError`.

String Indexing : It is nothing but fetching a specific character by using its index number.

Eg:



```
>>> x='narayana'  
>>> print(x)          narayana  
>>> type(x)           <class 'str'>  
>>> id(x)             52418104
```

Forward index

```
>>> print (x[0])      n  
>>> print (x[1])      a  
>>> print (x[2])      r  
>>> print (x[3])      a  
>>> print (x[4])      y  
>>> print (x[5])      a  
>>> print (x[6])      n  
>>> print (x[7])      a
```

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.

Backward index

```
>>> print (x[-1])      a  
>>> print (x[-2])      n  
>>> print (x[-3])      a  
>>> print (x[-4])      y  
>>> print (x[-5])      a  
>>> print (x[-6])      r  
>>> print (x[-7])      a  
>>> print (x[-8])      n
```

If we modify the content of the existing object then the indexes also will be changed

Eg:

```
>>> x=" narayana"  
>>> print(x)          narayana  
>>> type(x)          <class 'str'>  
>>> id(x)            52419584
```

```
>>> print(x[0])        #space is there here, because x contains space as first character.  
>>> print(x[1])        n  
>>> print(x[2])        a  
>>> print(x[3])        r  
>>> print(x[4])        a  
>>> print(x[5])        y  
>>> print(x[6])        a  
>>> print(x[7])        n
```

```
>>> print(x[8])          a  
  
>>> print(x[-1])        a  
>>> print(x[-2])        n  
>>> print(x[-3])        a  
>>> print(x[-4])        y  
>>> print(x[-5])        a  
>>> print(x[-6])        r  
>>> print(x[-7])        a  
>>> print(x[-8])        n  
>>> print(x[-9])        #space is there here, because x contains space as first character.
```

Eg2:

```
>>> st1="Python Narayana"
```

```
>>> st1[0]            'P'  
>>> st1[1]            'y'  
>>> st1[2]            't'  
>>> st1[3]            'h'  
>>> st1[4]            'o'  
>>> st1[5]            'n'  
>>> st1[6]            ''           #space is treated as empty character  
>>> st1[7]            'N'  
>>> st1[8]            'a'  
>>> st1[9]            'r'  
>>> st1[10]           'a'  
>>> st1[11]           'y'  
>>> st1[12]           'a'  
>>> st1[13]           'n'  
>>> st1[14]           'a'  
>>> st1[15]           IndexError: string index out of range  
>>> st1[-1]           'a'  
>>> st1[-2]           'n'  
>>> st1[-3]           'a'  
>>> st1[-16]          IndexError: string index out of range
```

we should use only integers as index numbers, if we try to use then interpreter will return TypeError,

using float numbers:

```
>>> st1[1.5]          TypeError: string indices must be integers
```

using string:

```
>>> st1['a']      TypeError: string indices must be integers
```

using complex number:

```
>>> st1[2+5j]      TypeError: string indices must be integers
```

but we can use bool values(True or False) as index numbers, because python treats True as 1 and False as 0

using bool values

```
>>> st1[True]      'y'  
>>> st1[False]     'P'
```

String slicing:

We can access a range of characters using slicing.

The slicing operator is colon (:)

Eg:

0 1 2 3 4 5 6 7

N A R A Y A N A

-8 -7 -6 -5 -4 -3 | -2 -1

```
>>> a='NARAYANA'
```

```
>>> print(a[0:1])      N  
>>> print(a[0:2])      NA  
>>> print(a[0:3])      NAR  
>>> print(a[0:4])      NARA  
>>> print(a[0:5])      NARAY  
>>> print(a[0:6])      NARAYA  
>>> print(a[0:7])      NARAYAN  
>>> print(a[0:8])      NARAYANA  
>>> print(a[2:8])      RAYANA  
>>> print(a[2:4])      RA  
>>> print(a[7:8])      A
```

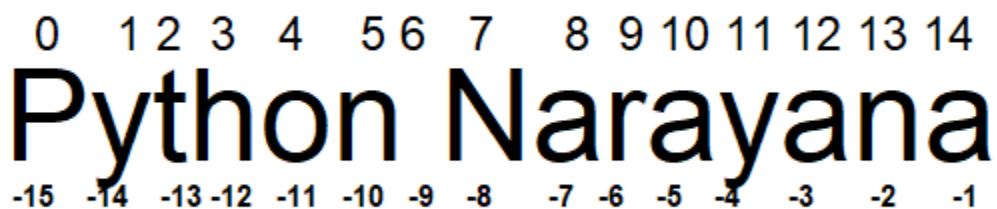
Narayana

PYTHON

Narayana

```
>>> print(a[-8:-6])      NA
>>> print(a[-8:-4])      NARA
>>> print(a[-8:-3])      NARAY
>>> print(a[-5:-4])      A
```

Eg2:



```
>>> st[:]          'Python Narayana'
>>> st[::]         'Python Narayana'
>>> st[4:]         'on Narayana'
>>> st[:4]         'Pyth'
>>> st[:-6]        'Python Na'
>>> st[-10:]       'n Narayana'
>>> st[-10:10]     'n Nar'
>>> st[-20:20]     'Python Narayana'
>>> st[10::2]       'aaa'
```

Here 10 is index starting number and 2 is step size

Syn: st[starting_number::step_size]

#from 10(means 'a' which is between 'r' and 'y'), it takes next 2 multiple characters in forward indexing because 2 is +ve number ,so 'aaa'

>>> st[1::2] 'yhnNryny'

#from 1(means 'y'),it takes all 2 multiple characters, like y,h,n,N,r,y,n... so it is'yhnNryny'

>>> st[-10::3] 'naya'

#from -10(means 'n'), it takes all 3 multiple characters, in forward indexing because 3 is +ve

>>> st[-9::3] ' ra'

#from -9(means ' '), its takes all 3 multiples, in forward indexing because 3 is +ve number

>>> st[-15::2] 'Pto aaaa'

#from -15(means 'P'), it takes all 2 multiple characters inforward indexing bcoz 2 is +ve num

>>> st[10::-2] 'aa otp'

10 means 'a' and negative index -5, it takes all -2 multiples in backward indexing from -5, so -5--> a, -7-->'a', -9-->' ', -11-->'o', -13-->'t',-15-->'P'.. finally 'aa otp'

>>> st[13::-2] 'nyrNnhy'

13 means n and negative index is -2, it takes all -2 multiples in backward indexing from -2, so -2-->n, -4-->y, -6-->r, -8-->N, -10-->n, -12-->h, -14-->y, finally 'nyrNnhy'

>>> st[::-2] 'aaaa otp'

here starting index number is not specified so here interpreter takes from last charater onwards, that is 14, and negative index number is -1, so from -1 it takes all -2 multiples in backward indexing, so it is 'aaaa otp'

>>> st[::-1] 'anayaraN nohtyP'

here also starting index is not specified so it takes from last character onwards, that is 14, and negative index number is -1, so from -1 it takes all -1 multiples in backward indexing, so finally it takes all characters from ending to begining.. that means it displays the gicen string in reverse order

```
>>> st[-10::-2]      'nhy'
```

-10 means 'n', it takes all -2 multiples from -10 in backward indexing, so -10-->n, -12-->h, -14-->y...
finally it is 'nhy'

```
>>> st[-1::-3]      'ayant'
```

-1 means 'a', it takes all -3 multiples from -1 in backward indexing, -1-->a, -4-->y, -7-->a, -10-->n,
-13-->t...finally it is 'ayant'

```
>>> st[-True::-3]      'ayant'
```

it works as above example, -True means -1

```
>>> st[::-True]      'anayaraN nohtyP'
```

it also displays the string in reverse order

String Concatenation:

1. We can concatenate two or more strings into a single string is called concatenation.
2. The + operator is used in Python for concatenation.

Eg:

```
>>> string1='Python'
```

```
>>> string2='Developer'
```

```
>>> print('String1 + string2 :', string1+ ' '+string2)
```

String1 + string2 : Python Developer

String Multiplication:

1. Python supports multiplying or repeating the given string into n number of times.
2. The * operator can be used to repeat the string for a given number of times.

Eg:

```
>>> string1='Python'
```

```
>>> print(string1 * 3)
```

PythonPythonPython

String is immutable object because we cannot replace or alter the existing string object.

Eg:

```
>>> st='Python'
```

```
>>> print(st)      Python
>>> id(st)        42221792

>>> st=st+'s'     #altering the string object

>>> print(st)      Pythons
>>> id(st)        41200640
Id is changed to before modification and after modification of the string st. that's why string is a immutable object.
Here when we try to edit the existing string object(st in 42221792), then the new object (st in 41200640) is created.
```

String packing:

Its nothing but packing all values of defined variables as a single string

```
>>> a='x'
>>> b='y'
>>> c='z'

>>> st=".join([a,b,c])"

>>> print(st)      xyz
>>> type(st)       <class 'str'>
```

String unpacking

1. String unpacking allows extracting all characters of string into different variables automatically.
2. The number of variables must be equal to number of characters in the string.

```
>>> str1="Python"

>>> print(str1)      Python
>>> type(str1)       <class 'str'>
>>> id(str1)         23941472

>>> a,b,c,d,e,f=str1 #string unpacking

>>> print(a)          P
>>> type(a)          <class 'str'>
```

```
>>> print(b)          y  
>>> type(b)         <class 'str'>  
  
>>> print(c)          t  
>>> type(c)         <class 'str'>  
  
>>> print(d)          h  
>>> type(d)         <class 'str'>  
  
>>> print(e)          o  
>>> type(e)         <class 'str'>  
  
>>> print(f)          n  
>>> type(f)         <class 'str'>
```

We can not update the existing characters of the string with new characters,
Eg:

```
>>> st="python"  
>>> print(st)        python  
>>> type(st)       <class 'str'>  
>>> id(st)          90527968  
  
>>> st[0]='X'        TypeError: 'str' object does not support item assignment
```

String Functions:

1. **Capitalize()**: this function converts first letter of first word in the given string into upper case.

Eg1:

```
>>> str1='python developer'  
>>> str1.capitalize()      'Python developer'
```

Eg2:

```
>>> st1='1python 2developer'  
>>> st1.capitalize()      '1python 2developer'
```

Eg3:

```
>>> st2='@gmail'  
>>> st2.capitalize() '@gmail'
```

Eg4:

```
>>> st3='gmail'  
>>> st3.capitalize() 'Gmail'
```

Eg5:

```
>>> st4=' python narayana'  
>>> st4.capitalize() ' python narayana'
```

2. Title(): this function converts first character of each word in the given string into upper case.

Eg1:

```
>>> str1='python developer'  
>>> str1.title() 'Python Developer'
```

Eg2:

```
>>> st1='1python 2developer'  
>>> st1.title() '1Python 2Developer'
```

Eg3:

```
>>> st2='@gmail'  
>>> st2.title() '@Gmail'
```

Eg4:

```
>>> st3='_python developer'  
>>> st3.title() '_ Python Developer'
```

Eg5:

```
>>> st4=' python narayana'  
>>> st4.title() ' Python Narayana'
```

3. **islower()**: this function checks whether the given string contains all lower case letters or not. If all are lower case then it will return True else False.

Eg1:

```
>>> str1='python developer'  
>>> str1.islower() True
```

Eg2:

```
>>> str2="Python"  
>>> str2.islower() False
```

Eg3:

```
>>> st=' '  
>>> st.islower() False
```

Eg4:

```
>>> str3='PYthon DEVEloper'  
>>> str3.islower() False
```

4. **isupper()**:this function checks whether the given string contains all upper case letters or not. If all are upper case then it will return True else False.

Eg1:

```
>>> str1='python developer'  
>>> str1.isupper() False
```

Eg2:

```
>>> str3='PYTHON'  
>>> str3.isupper() True
```

Eg3:

```
>>> st=' '  
>>> st.isupper() False
```

Eg4:

```
>>> str3='PYthon DEVEloper'  
>>> str3.isupper() False
```

5. **lower():** this function converts all letters of given string into lower case.

Eg1:

```
>>> str3='PYTHON'  
>>> str3.lower()  
'python'
```

Eg2:

```
>>> str3='PYthon DEVEloper'  
>>> str3.lower()  
'python developer'
```

6. **upper():** this function converts all letters of given string into upper case.

Eg1:

```
>>> str1='python developer'  
>>> str1.upper()  
'PYTHON DEVELOPER'
```

Eg2:

```
>>> str3='PYthon DEVEloper'  
>>> str3.upper()  
'PYTHON DEVELOPER'
```

7. **len():** this function counts the number of characters in the given string.

```
>>> str1='python developer'  
>>> len(str1)  
16
```

8. **count() :** this function counts no.of occurrences of a specific character in the given string.

Eg1:

```
>>> str1='python developer'  
>>> str1.count('o')  
2
```

Eg2:

```
>>> str3='python developer tcs hyd'  
>>> str3.count(' ')  
3
```

9. **find()**: this function finds the index position of specific character in the given string.

Eg1:

```
>>> str1='python developer'  
>>> str1.find('o')           4      ---for first occurrence of 'o'  
>>> str1.find('o',5)        12     ---for second occurrence of 'o'
```

Eg2:

```
>>> str3='python developer tcs hyd'  
  
>>> str3.index(' ')         6  
>>> str3.index(' ',7)       16  
>>> str3.index(' ',17)      20  
>>> str3.index(' ',21)      ValueError: substring not found
```

10. **split()**: this function splits the given strings into multiple strings

Eg1:

```
>>> str1='python developer'  
>>> str1.split()            ['python', 'developer']
```

Note: the default delimiter is space.

Eg2:

```
>>> str2='python developer in TCS'  
>>> str2.split()            ['python', 'developer', 'in', 'TCS']
```

Eg3:

```
>>> str3='python.developer.in.TCS'  
>>> str3.split('.')          ['python', 'developer', 'in', 'TCS']
```

Eg4:

```
>>> str4='developer'  
>>> str4.split('l')          ['deve', 'oper']
```

Eg5:

```
st="""Hello Renu,
```

Hope you doing fine,,
How is your python language preparation?
You should learn DJANGO Framework as well,
so that you will get more calls.

Thanks,
Python Narayana,
Python Trainer.
'''

>>> Print(St.split('\n'))

```
['Hello Renu,', 'Hope you doing fine,,', 'How is your python language preparation?', 'You should  
learn DJANGO Framework as well,', 'so that you will get more calls.', ", 'Thanks,", 'Python  
Narayana,', 'Python Trainer.', "]
```

>>> st.split(' ')

```
['Hello', 'Renu,\nHope', 'you', 'doing', 'fine,,\nHow', 'is', 'your', 'python', 'language',  
'preparation?\nYou', 'should', 'learn', 'DJANGO', 'Framework', 'as', 'well,\nso', 'that', 'you', 'will',  
'get', 'more', 'calls.\n\nThanks,\nPython', 'Narayana,\nPython', 'Trainer.\n']
```

11. lstrip(): this function removes specific special character to left side of given string.

```
str1='!!!!!!Python Developer!!!!!!'  
>>> str1.lstrip('!')                                 'Python Developer!!!!!!'
```

12. rstrip(): this function removes specific special character to the right side of given string

```
str1='!!!!!!Python Developer!!!!!!'  
>>> str1.rstrip('!')                                 '!!!!!!Python Developer'
```

12.strip(): this function removes specific special character from both sides to the given string.

```
str1='!!!!!!Python Developer!!!!!!'  
>>> str1.strip('!')                                 'Python Developer'
```

13.swapcase(): This function swaps all lower case letters into upper case and vice versa.

Eg1:

```
>>> str1='PyThOn'  
>>> str1.swapcase()  
'pYtHoN'
```

Eg2:

```
>>> str2='PYTHON DEVELOPER'  
>>> str2.swapcase()  
'python developer'
```

Eg3:

```
>>> str3='python developer'  
>>> str3.swapcase()  
'PYTHON DEVELOPER'
```

14. reversed(): this function reverses the string

```
>>> str='Python'  
>>> print(str)  
Python  
  
>>> str=".join(reversed(str))  
>>> print(str)  
nohtyP
```

15. replace(): this function replaces an existing character(s) with new character(s).

Eg1:

```
>>> str1='python learner'  
>>> print(str1)  
python learner  
  
>>> str2=str1.replace('learner','developer')  
>>> print(str2)  
python developer
```

Eg2:

```
>>> st1='SQL Narayana'  
>>> st1=st1.replace('SQL','Python')  
>>> print(st1)  
Python Narayana
```

Note: we can remove any character(s) with non-empty space.

```
>>> str1='Python'  
>>> print(str1)  
Python  
>>> type(str1)  
<class 'str'>
```

```
>>> str2=str1.replace('thon','')
>>> print(str2)
Py
>>> type(str2)
<class 'str'>
```

del command:

we can't delete a specific character or range of characters by using del command.

We can delete the entire string objects permanently by using del command

Eg 1:

```
>>> str1="Python Narayana"      #trying to remove specific character
>>> del str1[0]                TypeError: 'str' object doesn't support item deletion
```

Eg 2:

```
>>> str1="Python Narayana"      #trying to remove specific range of characters
>>> del str1[1:5]                TypeError: 'str' object does not support item deletion
```

Eg 3:

```
>>> str1="Python Narayana"
>>> print(str1)                  Python Narayana
>>> type(str1)                   <class 'str'>
>>> id(str1)                     63806464

>>> del str1                      #deleting entire string str1 object

>>> print(str1)                  #after deleting
NameError: name 'str1' is not defined
```

rjust(), ljust():

The rjust() and ljust() string methods return a padded version of the string they are called on, with spaces inserted to justify the text. The first argument to both methods is an integer length for the justified string.

```
>>> 'python'.rjust(10)
'  python'
```

```
>>> 'python'.rjust(15)
      python

>>> 'python'.ljust(10)
'python  '

>>> 'python'.ljust(15)
'python      '

>>> 'Hello Python'.rjust(20)
      Hello Python'

>>> 'Hello Python'.ljust(20)
'Hello Python      '
```

'python'.rjust(10) says that we want to right-justify 'Hello' in a string of total length 10. 'python' is six characters, so four spaces will be added to its left, giving us a string of 10 characters with 'python' justified right.

An optional second argument to rjust() and ljust() will specify a fill character other than a space character.

```
>>> 'python'.rjust(10,'*')
*****python

>>> 'python'.rjust(15,'@')
'@@@@@@@@@@@python'

>>> 'python'.ljust(10,'*')
'python****'

>>> 'python'.ljust(15,'*')
'python*****'

>>> 'Hello Python'.rjust(20,'!')
'!!!!!!Hello Python'

>>> 'Hello Python'.ljust(20,'!')
```

```
'Hello Python!!!!!!!'
```

Center()

The center() string method works like ljust() and rjust() but centers the text rather than justifying it to the left or right.

```
>>> 'python'.center(10)
' python '
>>> 'python'.center(15)
'   python   '
>>> 'python narayana'.center(20)
' python narayana '
>>> 'python'.center(10,'*')
'*'*python**'
>>> 'python'.center(15,'*')
'*****python*****'
>>> 'python narayana'.center(20,'*')
'***python narayana***'
>>> "Hello".center(20,'=')
'=====Hello====='
```

These methods are mainly useful when we need to print tabular data that has the correct spacing.

How to display the given string in ascending order?

```
>>> st="python"
>>> ".join(sorted(st))"
'hnopty'
>>> st1="PyThON deVELoPer"
#if string contains both upper and lower cases
>>> ".join(sorted(st1))"
'ELNOPPTVdeehory'
```

Here, The join() method is useful when we have a list of strings that need to be joined together into a single string value. The join() method is called on a string, gets passed a list of strings, and returns a string. The returned string is the concatenation of each string in the passed-in list.

How to display the given string in descending order?

```
>>> st="python"  
>>> ".join(reversed(sorted((st))))"           'ytponh'
```

How to display the given string with two dots between each character.

```
>>> s1='python'  
  
>>> s10='..'.join(s1)  
  
>>> s10                         'p..y..t..h..o..n'
```

How to display the given string with space between each character.

```
>>> st='Narayana'  
  
>>> st=' '.join(st)  
  
>>> print(st)                   Narayana
```

By using ::

```
>>> st='python narayana'  
>>> st[::]          'python narayana'  
>>> st[0::]        'python narayana'  
>>> st[1::]        'ython narayana'  
>>> st[2::]        'thon narayana'  
>>> st[3::]        'hon narayana'  
>>> st[4::]        'on narayana'  
>>> st[5::]        'n narayana'  
>>> st[6::]        ' narayana'  
>>> st[7::]        'narayana'  
>>> st[8::]        'arayana'  
>>> st[9::]        'rayana'  
>>> st[10::]       'ayana'  
>>> st[11::]       'yana'  
>>> st[12::]       'ana'  
>>> st[13::]       'na'  
>>> st[14::]       'a'  
>>> st[15::]       ''  
>>> st[16::]       ''  
>>> st[0::1]       'python narayana'  
>>> st[0::2]       'pto aaaa'  
>>> st[0::3]       'ph ra'  
>>> st[0::4]       'poaa'  
>>> st[0::5]       'pna'  
>>> st[0::6]       'p a'  
>>> st[0::7]       'pna'  
>>> st[0::8]       'pa'  
>>> st[0::9]       'pr'  
>>> st[0::10]      'pa'  
>>> st[0::11]      'py'  
>>> st[0::12]      'pa'  
>>> st[0::13]      'pn'  
>>> st[0::14]      'pa'  
>>> st[0::15]      'p'  
>>> st[0::16]      'p'  
>>> st[0::17]      'p'  
>>> st[1::0]       ValueError: slice step cannot be zero  
>>> st[1::1]       'ython narayana'  
>>> st[1::2]       'yhnnryn'  
>>> st[1::3]       'yonan'  
>>> st[1::4]       'ynrn'  
>>> st[1::5]       'y y'
```

Narayana

PYTHON

Narayana

```
>>> st[1::6]      'ynn'  
>>> st[1::7]      'ya'  
>>> st[1::8]      'yr'  
>>> st[1::9]      'ya'  
>>> st[1::10]     'yy'  
>>> st[1::11]     'ya'  
>>> st[1::12]     'yn'  
>>> st[1::13]     'ya'  
>>> st[1::14]     'y'  
>>> st[1::15]     'y'  
>>> st[2::1]      'thon narayana'  
>>> st[2::2]      'to aaaa'  
>>> st[2::3]      'tnaya'  
>>> st[2::4]      't aa'  
>>> st[2::5]      'tna'  
>>> st[2::6]      'taa'  
>>> st[2::7]      'tr'  
>>> st[2::8]      'ta'  
>>> st[2::9]      'ty'  
>>> st[2::10]     'ta'  
>>> st[2::11]     'tn'  
>>> st[2::12]     'ta'  
>>> st[2::13]     't'  
>>> st[3::1]      'hon narayana'  
>>> st[3::2]      'hnnrynn'  
>>> st[3::3]      'h ra'  
>>> st[3::4]      'hny'  
>>> st[3::5]      'han'  
>>> st[3::6]      'hr'  
>>> st[3::7]      'ha'  
>>> st[3::8]      'hy'  
>>> st[3::9]      'ha'  
>>> st[3::10]     'hn'  
>>> st[3::11]     'ha'  
>>> st[3::12]     'h'  
>>> st[3::13]     'h'  
>>> st[4::1]      'on narayana'  
>>> st[4::2]      'o aaaa'  
>>> st[4::3]      'onan'  
>>> st[4::4]      'oaa'  
>>> st[4::5]      'ora'  
>>> st[4::6]      'oa'  
>>> st[4::7]      'oy'  
>>> st[4::8]      'oa'
```

Narayana

PYTHON

Narayana

```
>>> st[4::9]      'on'
>>> st[4::10]     'oa'
>>> st[4::11]     'o'
>>> st[5::1]      'n narayana'
>>> st[5::2]      'nnryn'
>>> st[5::3]      'naya'
>>> st[5::4]      'nrn'
>>> st[5::5]      'na'
>>> st[5::6]      'ny'
>>> st[5::7]      'na'
>>> st[5::8]      'nn'
>>> st[5::9]      'na'
>>> st[5::10]     'n'
>>> st[6::1]      ' narayana'
>>> st[6::2]      ' aaaa'
>>> st[6::2]      ' aaaa'
>>> st[6::4]      ' aa'
>>> st[6::3]      ' ra'
>>> st[6::5]      ' y'
>>> st[6::6]      ' a'
>>> st[6::7]      ' n'
>>> st[6::8]      ' a'
>>> st[6::9]      ''
>>> st[7::1]      'narayana'
>>> st[7::2]      'nryn'
>>> st[7::3]      'nan'
>>> st[7::4]      'ny'
>>> st[7::5]      'na'
>>> st[7::6]      'nn'
>>> st[7::7]      'na'
>>> st[7::8]      'n'
>>> st[7::9]      'n'
>>> st[8::1]      'arayana'
>>> st[8::2]      'aaaa'
>>> st[8::3]      'aya'
>>> st[8::4]      'aa'
>>> st[8::5]      'an'
>>> st[8::6]      'aa'
>>> st[8::7]      'a'
>>> st[8::8]      'a'
>>> st[9::1]      'rayana'
>>> st[9::2]      'ryn'
>>> st[9::3]      'ra'
>>> st[9::4]      'rn'
```

Narayana

PYTHON

Narayana

```
>>> st[9::5]      'ra'  
>>> st[9::6]      'r'  
>>> st[9::7]      'r'  
>>> st[10::1]     'ayana'  
>>> st[10::2]     'aaa'  
>>> st[10::3]     'an'  
>>> st[10::4]     'aa'  
>>> st[10::5]     'a'  
>>> st[10::6]     'a'  
>>> st[11::1]     'yana'  
>>> st[11::2]     'yn'  
>>> st[11::3]     'ya'  
>>> st[11::4]     'y'  
>>> st[11::5]     'y'  
>>> st[12::1]     'ana'  
>>> st[12::2]     'aa'  
>>> st[12::3]     'a'  
>>> st[12::4]     'a'  
>>> st[13::1]     'na'  
>>> st[13::2]     'n'  
>>> st[13::3]     'n'  
>>> st[14::1]     'a'  
>>> st[0::-1]     'p'  
>>> st[1::-1]     'yp'  
>>> st[2::-1]     'typ'  
>>> st[3::-1]     'htyp'  
>>> st[4::-1]     'ohtyp'  
>>> st[5::-1]     'nohtyp'  
>>> st[6::-1]     ' nohtyp'  
>>> st[7::-1]     'n nohtyp'  
>>> st[8::-1]     'an nohtyp'  
>>> st[9::-1]     'ran nohtyp'  
>>> st[10::-1]    'aran nohtyp'  
>>> st[11::-1]    'yaran nohtyp'  
>>> st[12::-1]    'ayaran nohtyp'  
>>> st[13::-1]    'nayaran nohtyp'  
>>> st[14::-1]    'anayaran nohtyp'  
>>> st[15::-1]    'anayaran nohtyp'  
>>> st[0::-2]     'p'  
>>> st[1::-2]     'y'  
>>> st[2::-2]     'tp'  
>>> st[3::-2]     'hy'  
>>> st[4::-2]     'otp'  
>>> st[5::-2]     'nhy'
```

Narayana

PYTHON

Narayana

```
>>> st[6::-2]      ' otp'
>>> st[7::-2]      'nnhy'
>>> st[8::-2]      'a otp'
>>> st[9::-2]      'rnny'
>>> st[10::-2]     'aa otp'
>>> st[11::-2]     'yrnny'
>>> st[12::-2]     'aaa otp'
>>> st[13::-2]     'nyrnny'
>>> st[14::-2]     'aaaa otp'
>>> st[15::-2]     'aaaa otp'
>>> st[0::-3]       'p'
>>> st[1::-3]       'y'
>>> st[2::-3]       't'
>>> st[3::-3]       'hp'
>>> st[4::-3]       'oy'
>>> st[5::-3]       'nt'
>>> st[6::-3]       ' hp'
>>> st[7::-3]       'noy'
>>> st[8::-3]       'ant'
>>> st[9::-3]       'r hp'
>>> st[10::-3]      'anoy'
>>> st[11::-3]      'yant'
>>> st[12::-3]      'ar hp'
>>> st[13::-3]      'nanoy'
>>> st[14::-3]      'ayant'
>>> st[15::-3]      'ayant'
>>> st[0::-4]       'p'
>>> st[1::-4]       'y'
>>> st[2::-4]       't'
>>> st[3::-4]       'h'
>>> st[4::-4]       'op'
>>> st[5::-4]       'ny'
>>> st[6::-4]       ' t'
>>> st[7::-4]       'nh'
>>> st[8::-4]       'aop'
>>> st[9::-4]       'rny'
>>> st[10::-4]      'a t'
>>> st[11::-4]      'ynh'
>>> st[12::-4]      'aaop'
>>> st[13::-4]      'nrny'
>>> st[14::-4]      'aa t'
>>> st[15::-4]      'aa t'
>>> st[16::-4]      'aa t'
>>> st[0::-5]       'p'
```

Narayana

PYTHON

Narayana

```
>>> st[1:-5]      'y'  
>>> st[2:-5]      't'  
>>> st[3:-5]      'h'  
>>> st[4:-5]      'o'  
>>> st[5:-5]      'np'  
>>> st[6:-5]      ' y'  
>>> st[7:-5]      'nt'  
>>> st[8:-5]      'ah'  
>>> st[9:-5]      'ro'  
>>> st[10:-5]     'anp'  
>>> st[11:-5]     'y y'  
>>> st[12:-5]     'ant'  
>>> st[13:-5]     'nah'  
>>> st[14:-5]     'aro'  
>>> st[15:-5]     'aro'  
>>> st[0:-6]       'p'  
>>> st[1:-6]       'y'  
>>> st[2:-6]       't'  
>>> st[3:-6]       'h'  
>>> st[4:-6]       'o'  
>>> st[5:-6]       'n'  
>>> st[6:-6]       ' p'  
>>> st[7:-6]       'ny'  
>>> st[8:-6]       'at'  
>>> st[9:-6]       'rh'  
>>> st[10:-6]      'ao'  
>>> st[11:-6]      'yn'  
>>> st[12:-6]      'a p'  
>>> st[13:-6]      'hny'  
>>> st[14:-6]      'aat'  
>>> st[15:-6]      'aat'  
>>> st[0:-7]       'p'  
>>> st[1:-7]       'y'  
>>> st[2:-7]       't'  
>>> st[3:-7]       'h'  
>>> st[4:-7]       'o'  
>>> st[5:-7]       'n'  
>>> st[6:-7]       ''  
>>> st[7:-7]       'np'  
>>> st[8:-7]       'ay'  
>>> st[9:-7]       'rt'  
>>> st[10:-7]      'ah'  
>>> st[11:-7]      'yo'  
>>> st[12:-7]      'an'
```

Narayana

PYTHON

Narayana

```
>>> st[13::-7]      'n '
>>> st[14::-7]      'anp'
>>> st[15::-7]      'anp'
>>> st[0::-8]        'p'
>>> st[1::-8]        'y'
>>> st[2::-8]        't'
>>> st[3::-8]        'h'
>>> st[4::-8]        'o'
>>> st[5::-8]        'n'
>>> st[6::-8]        ''
>>> st[7::-8]        'n'
>>> st[8::-8]        'ap'
>>> st[9::-8]        'ry'
>>> st[10::-8]       'at'
>>> st[11::-8]       'yh'
>>> st[12::-8]       'ao'
>>> st[13::-8]       'nn'
>>> st[14::-8]       'a '
>>> st[15::-8]       'a '
>>> st[16::-8]       'a '
>>> st[17::-8]       'a '
>>> st[18::-8]       'a '
>>> st[0::-9]         'p'
>>> st[1::-9]         'y'
>>> st[2::-9]         't'
>>> st[3::-9]         'h'
>>> st[4::-9]         'o'
>>> st[5::-9]         'n'
>>> st[6::-9]         ''
>>> st[7::-9]         'n'
>>> st[8::-9]         'a'
>>> st[9::-9]         'rp'
>>> st[10::-9]        'ay'
>>> st[11::-9]        'yt'
>>> st[12::-9]        'ah'
>>> st[13::-9]        'no'
>>> st[14::-9]        'an'
>>> st[15::-9]        'an'
>>> st[0::-10]        'p'
>>> st[1::-10]        'y'
>>> st[2::-10]        't'
>>> st[3::-10]        'h'
>>> st[4::-10]        'o'
>>> st[5::-10]        'n'
```

Narayana

PYTHON

Narayana

```
>>> st[6::-10]      ''
>>> st[7::-10]      'n'
>>> st[8::-10]      'a'
>>> st[9::-10]      'r'
>>> st[10::-10]     'ap'
>>> st[11::-10]     'yy'
>>> st[12::-10]     'at'
>>> st[13::-10]     'nh'
>>> st[14::-10]     'ao'
>>> st[15::-10]     'ao'
>>> st[0::-11]      'p'
>>> st[1::-11]      'y'
>>> st[2::-11]      't'
>>> st[3::-11]      'h'
>>> st[4::-11]      'o'
>>> st[5::-11]      'n'
>>> st[6::-11]      ''
>>> st[7::-11]      'n'
>>> st[8::-11]      'a'
>>> st[9::-11]      'r'
>>> st[10::-11]     'a'
>>> st[11::-11]     'yp'
>>> st[12::-11]     'ay'
>>> st[13::-11]     'nt'
>>> st[14::-11]     'ah'
>>> st[15::-11]     'ah'
>>> st[0::-12]      'p'
>>> st[1::-12]      'y'
>>> st[2::-12]      't'
>>> st[3::-12]      'h'
>>> st[4::-12]      'o'
>>> st[5::-12]      'n'
>>> st[6::-12]      ''
>>> st[7::-12]      'n'
>>> st[8::-12]      'a'
>>> st[9::-12]      'r'
>>> st[10::-12]     'a'
>>> st[11::-12]     'y'
>>> st[12::-12]     'ap'
>>> st[13::-12]     'ny'
>>> st[14::-12]     'at'
>>> st[15::-12]     'at'
>>> st[0::-13]      'p'
>>> st[1::-13]      'y'
```

Narayana

PYTHON

Narayana

```
>>> st[2::-13]      't'  
>>> st[3::-13]      'h'  
>>> st[4::-13]      'o'  
>>> st[5::-13]      'n'  
>>> st[6::-13]      ''  
>>> st[7::-13]      'n'  
>>> st[8::-13]      'a'  
>>> st[9::-13]      'r'  
>>> st[10::-13]     'a'  
>>> st[11::-13]     'y'  
>>> st[12::-13]     'a'  
>>> st[13::-13]     'np'  
>>> st[14::-13]     'ay'  
>>> st[15::-13]     'ay'  
>>> st[16::-13]     'ay'  
>>> st[0::-14]      'p'  
>>> st[1::-14]      'y'  
>>> st[2::-14]      't'  
>>> st[3::-14]      'h'  
>>> st[4::-14]      'o'  
>>> st[5::-14]      'n'  
>>> st[6::-14]      ''  
>>> st[7::-14]      'n'  
>>> st[8::-14]      'a'  
>>> st[9::-14]      'r'  
>>> st[10::-14]     'a'  
>>> st[11::-14]     'y'  
>>> st[12::-14]     'a'  
>>> st[13::-14]     'n'  
>>> st[14::-14]     'ap'  
>>> st[15::-14]     'ap'  
>>> st[0::-15]      'p'  
>>> st[1::-15]      'y'  
>>> st[2::-15]      't'  
>>> st[3::-15]      'h'  
>>> st[4::-15]      'o'  
>>> st[5::-15]      'n'  
>>> st[6::-15]      ''  
>>> st[7::-15]      'n'  
>>> st[8::-15]      'a'  
>>> st[9::-15]      'r'  
>>> st[10::-15]     'a'  
>>> st[11::-15]     'y'  
>>> st[12::-15]     'a'
```

Narayana

PYTHON

Narayana

```
>>> st[13::-15]      'n'  
>>> st[14::-15]      'a'  
>>> st[15::-15]      'a'  
>>> st[0::-16]       'p'  
>>> st[1::-16]        'y'  
>>> st[2::-16]        't'  
>>> st[3::-16]        'h'  
>>> st[4::-16]        'o'  
>>> st[5::-16]        'n'  
>>> st[6::-16]        ''  
>>> st[7::-16]        'n'  
>>> st[8::-16]        'a'  
>>> st[9::-16]        'r'  
>>> st[10::-16]       'a'  
>>> st[11::-16]       'y'  
>>> st[12::-16]       'a'  
>>> st[13::-16]       'n'  
>>> st[14::-16]       'a'  
>>> st[15::-16]       'a'  
>>> st[16::-16]       'a'  
>>> st[17::-16]       'a'  
>>> st[18::-16]       'a'  
>>> st[-1::1]         'a'  
>>> st[-1::2]         'a'  
>>> st[-1::3]         'a'  
>>> st[-2::1]         'na'  
>>> st[-2::2]         'n'  
>>> st[-2::3]         'n'  
>>> st[-2::4]         'n'  
>>> st[-3::1]         'ana'  
>>> st[-3::2]         'aa'  
>>> st[-3::3]         'a'  
>>> st[-3::4]         'a'  
>>> st[-4::1]         'yana'  
>>> st[-4::2]         'yn'  
>>> st[-4::3]         'ya'  
>>> st[-4::4]         'y'  
>>> st[-4::5]         'y'  
>>> st[-5::1]         'ayana'  
>>> st[-5::2]         'aaa'  
>>> st[-5::3]         'an'  
>>> st[-5::4]         'aa'  
>>> st[-5::5]         'a'  
>>> st[-5::6]         'a'
```

Narayana

PYTHON

Narayana

```
>>> st[-6::1]      'rayana'  
>>> st[-6::2]      'ryn'  
>>> st[-6::3]      'ra'  
>>> st[-6::4]      'rn'  
>>> st[-6::5]      'ra'  
>>> st[-6::6]      'r'  
>>> st[-6::7]      'r'  
>>> st[-7::1]      'arayana'  
>>> st[-7::2]      'aaaa'  
>>> st[-7::3]      'aya'  
>>> st[-7::4]      'aa'  
>>> st[-7::5]      'an'  
>>> st[-7::6]      'aa'  
>>> st[-7::7]      'a'  
>>> st[-7::8]      'a'  
>>> st[-8::1]      'narayana'  
>>> st[-8::2]      'nryn'  
>>> st[-8::3]      'nan'  
>>> st[-8::4]      'ny'  
>>> st[-8::5]      'na'  
>>> st[-8::6]      'nn'  
>>> st[-8::7]      'na'  
>>> st[-8::8]      'n'  
>>> st[-8::9]      'n'  
>>> st[-9::1]      ' narayana'  
>>> st[-9::2]      'aaaa'  
>>> st[-9::3]      ' ra'  
>>> st[-9::4]      ' aa'  
>>> st[-9::5]      ' y'  
>>> st[-9::6]      ' a'  
>>> st[-9::7]      ' n'  
>>> st[-9::8]      ' a'  
>>> st[-9::9]      ''  
>>> st[-9::10]     ''  
>>> st[-10::1]     'n narayana'  
>>> st[-10::2]     'nnryn'  
>>> st[-10::3]     'naya'  
>>> st[-10::4]     'nrn'  
>>> st[-10::5]     'na'  
>>> st[-10::6]     'ny'  
>>> st[-10::7]     'na'  
>>> st[-10::8]     'nn'  
>>> st[-10::9]     'na'  
>>> st[-10::10]    'n'
```

Narayana

PYTHON

Narayana

```
>>> st[-10::11]      'n'  
>>> st[-11::1]      'on narayana'  
>>> st[-11::2]      'o aaaa'  
>>> st[-11::3]      'onan'  
>>> st[-11::4]      'oaa'  
>>> st[-11::5]      'ora'  
>>> st[-11::6]      'oa'  
>>> st[-11::7]      'oy'  
>>> st[-11::8]      'oa'  
>>> st[-11::9]      'on'  
>>> st[-11::10]     'oa'  
>>> st[-11::11]     'o'  
>>> st[-11::12]     'o'  
>>> st[-12::1]      'hon narayana'  
>>> st[-12::1]      'hon narayana'  
>>> st[-12::2]      'hnnrynn'  
>>> st[-12::3]      'h ra'  
>>> st[-12::4]      'hny'  
>>> st[-12::5]      'han'  
>>> st[-12::6]      'hr'  
>>> st[-12::7]      'ha'  
>>> st[-12::8]      'hy'  
>>> st[-12::9]      'ha'  
>>> st[-12::10]     'hn'  
>>> st[-12::11]     'ha'  
>>> st[-12::12]     'h'  
>>> st[-12::13]     'h'  
>>> st[-13::1]      'thon narayana'  
>>> st[-13::2]      'to aaaa'  
>>> st[-13::3]      'tnaya'  
>>> st[-13::4]      't aa'  
>>> st[-13::5]      'tna'  
>>> st[-13::6]      'taa'  
>>> st[-13::7]      'tr'  
>>> st[-13::8]      'ta'  
>>> st[-13::9]      'ty'  
>>> st[-13::10]     'ta'  
>>> st[-13::11]     'tn'  
>>> st[-13::12]     'ta'  
>>> st[-13::13]     't'  
>>> st[-13::14]     't'  
>>> st[-14::1]      'ython narayana'  
>>> st[-14::2]      'yhnnrynn'  
>>> st[-14::3]      'yonan'
```

```
>>> st[-14::4]      'ynrn'  
>>> st[-14::5]      'y y'  
>>> st[-14::6]      'ynn'  
>>> st[-14::7]      'ya'  
>>> st[-14::8]      'yr'  
>>> st[-14::9]      'ya'  
>>> st[-14::10]     'yy'  
>>> st[-14::11]     'ya'  
>>> st[-14::12]     'yn'  
>>> st[-14::13]     'ya'  
>>> st[-14::14]     'y'  
>>> st[-14::15]     'y'  
>>> st[-15::1]      'python narayana'  
>>> st[-15::2]      'pto aaaa'  
>>> st[-15::3]      'ph ra'  
>>> st[-15::4]      'poaa'  
>>> st[-15::5]      'pna'  
>>> st[-15::6]      'p a'  
>>> st[-15::7]      'pna'  
>>> st[-15::8]      'pa'  
>>> st[-15::9]      'pr'  
>>> st[-15::10]     'pa'  
>>> st[-15::11]     'py'  
>>> st[-15::12]     'pa'  
>>> st[-15::13]     'pn'  
>>> st[-15::14]     'pa'  
>>> st[-15::15]     'p'  
>>> st[-15::16]     'p'  
>>> st[-1::-1]      'anayaran nohtyp'  
>>> st[-1::-2]      'aaaa otp'  
>>> st[-1::-3]      'ayant'  
>>> st[-1::-4]      'aa t'  
>>> st[-1::-5]      'aro'  
>>> st[-1::-6]      'aat'  
>>> st[-1::-7]      'anp'  
>>> st[-1::-8]      'a '  
>>> st[-1::-9]      'an'  
>>> st[-1::-10]     'ao'  
>>> st[-1::-11]     'ah'  
>>> st[-1::-12]     'at'  
>>> st[-1::-13]     'ay'  
>>> st[-1::-14]     'ap'  
>>> st[-1::-15]     'a'  
>>> st[-1::-16]     'a'
```

```
>>> st[-2::-1]      'nayaran nohtyp'  
>>> st[-2::-2]      'nyrnnhy'  
>>> st[-2::-3]      'nanoy'  
>>> st[-2::-4]      'nrny'  
>>> st[-2::-5]      'nah'  
>>> st[-2::-6]      'nny'  
>>> st[-2::-7]      'n '  
>>> st[-2::-8]      'nn'  
>>> st[-2::-9]      'no'  
>>> st[-2::-10]     'nh'  
>>> st[-2::-11]     'nt'  
>>> st[-2::-12]     'ny'  
>>> st[-2::-13]     'np'  
>>> st[-2::-14]     'n'  
>>> st[-2::-15]     'n'  
>>> st[-3::-1]      'ayaran nohtyp'  
>>> st[-3::-2]      'aaa otp'  
>>> st[-3::-3]      'ar hp'  
>>> st[-3::-4]      'aaop'  
>>> st[-3::-5]      'ant'  
>>> st[-3::-6]      'a p'  
>>> st[-3::-7]      'an'  
>>> st[-3::-8]      'ao'  
>>> st[-3::-9]      'ah'  
>>> st[-3::-10]     'at'  
>>> st[-3::-11]     'ay'  
>>> st[-3::-12]     'ap'  
>>> st[-3::-13]     'a'  
>>> st[-3::-14]     'a'  
>>> st[-4::-1]      'yaran nohtyp'  
>>> st[-4::-2]      'yrnnhy'  
>>> st[-4::-3]      'yant'  
>>> st[-4::-4]      'ynh'  
>>> st[-4::-5]      'y y'  
>>> st[-4::-6]      'yn'  
>>> st[-4::-7]      'yo'  
>>> st[-4::-8]      'yh'  
>>> st[-4::-9]      'yt'  
>>> st[-4::-10]     'yy'  
>>> st[-4::-11]     'yp'  
>>> st[-4::-12]     'y'  
>>> st[-4::-13]     'y'  
>>> st[-5::-1]      'aran nohtyp'  
>>> st[-5::-2]      'aa otp'
```

```
>>> st[-5::-3]      'anoy'  
>>> st[-5::-4]      'a t'  
>>> st[-5::-5]      'anp'  
>>> st[-5::-6]      'ao'  
>>> st[-5::-7]      'ah'  
>>> st[-5::-8]      'at'  
>>> st[-5::-9]      'ay'  
>>> st[-5::-10]     'ap'  
>>> st[-5::-11]     'a'  
>>> st[-5::-12]     'a'  
>>> st[-6::-1]      'ran nohtyp'  
>>> st[-6::-2]      'rnnhy'  
>>> st[-6::-3]      'r hp'  
>>> st[-6::-4]      'rny'  
>>> st[-6::-5]      'ro'  
>>> st[-6::-6]      'rh'  
>>> st[-6::-7]      'rt'  
>>> st[-6::-8]      'ry'  
>>> st[-6::-9]      'rp'  
>>> st[-6::-10]     'r'  
>>> st[-6::-11]     'r'  
>>> st[-7::-1]      'an nohtyp'  
>>> st[-7::-2]      'a otp'  
>>> st[-7::-3]      'ant'  
>>> st[-7::-4]      'aop'  
>>> st[-7::-5]      'ah'  
>>> st[-7::-6]      'at'  
>>> st[-7::-7]      'ay'  
>>> st[-7::-8]      'ap'  
>>> st[-7::-9]      'a'  
>>> st[-8::-1]      'n nohtyp'  
>>> st[-8::-2]      'nnhy'  
>>> st[-8::-3]      'noy'  
>>> st[-8::-4]      'nh'  
>>> st[-8::-5]      'nt'  
>>> st[-8::-6]      'ny'  
>>> st[-8::-7]      'np'  
>>> st[-8::-8]      'n'  
>>> st[-9::-1]      ' nohtyp'  
>>> st[-9::-2]      ' otp'  
>>> st[-9::-3]      ' hp'  
>>> st[-9::-4]      ' t'  
>>> st[-9::-5]      ' y'  
>>> st[-9::-6]      ' p'
```

Narayana

PYTHON

Narayana

```
>>> st[-9::-7]      ''
>>> st[-9::-8]      ''
>>> st[-10::-1]     'nohtyp'
>>> st[-10::-2]     'nhy'
>>> st[-10::-3]     'nt'
>>> st[-10::-4]     'ny'
>>> st[-10::-5]     'np'
>>> st[-10::-6]     'n'
>>> st[-10::-7]     'n'
>>> st[-10::-8]     'n'
>>> st[-11::-1]     'ohtyp'
>>> st[-11::-2]     'otp'
>>> st[-11::-3]     'oy'
>>> st[-11::-4]     'op'
>>> st[-11::-5]     'o'
>>> st[-12::-1]     'htyp'
>>> st[-12::-2]     'hy'
>>> st[-12::-3]     'hp'
>>> st[-12::-4]     'h'
>>> st[-13::-1]     'typ'
>>> st[-13::-2]     'tp'
>>> st[-13::-3]     't'
>>> st[-14::-1]     'yp'
>>> st[-14::-2]     'y'
>>> st[-15::-1]     'p'
>>> st[-15::-2]     'p'
>>> st[-16::-1]     ''
```

String comparison

Eg1:

```
st1='b'  
st2='a'  
  
if st1>st2:  
    print(st1,'is bigger than',st2)  
elif st2>st1:  
    print(st2,'is bigger than',st1)  
else:  
    print('both',st1,'and',st2,'are equal')
```

Output

b is bigger than a

Note: in the above program, we have set st1='b' and st2='a' when we see the alphabet in ascending order(from a to z), then here, 'b' is bigger than 'a'

Eg2:

```
st1='a'  
st2='b'  
  
if st1>st2:  
    print(st1,'is bigger than',st2)  
elif st2>st1:  
    print(st2,'is bigger than',st1)  
else:  
    print('both',st1,'and',st2,'are equal')
```

Output

b is bigger than a

Eg3:

```
st1='a'  
st2='a'  
  
if st1>st2:  
    print(st1,'is bigger than',st2)  
elif st2>st1:  
    print(st2,'is bigger than',st1)  
else:  
    print('both',st1,'and',st2,'are equal')
```

Output

both a and a are equal

Eg4:

```
st1='bc'  
st2='ax'  
  
if st1>st2:  
    print(st1,'is bigger than',st2)  
elif st2>st1:  
    print(st2,'is bigger than',st1)  
else:  
    print('both',st1,'and',st2,'are equal')
```

Output

bc is bigger than ax

Note: interpreter will check only first characters from both strings, so as per first characters, st1 is bigger than st2.

Eg5:

```
st1='ax'  
st2='am'  
  
if st1>st2:  
    print(st1,'is bigger than',st2)  
elif st2>st1:  
    print(st2,'is bigger than',st1)  
else:  
    print('both',st1,'and',st2,'are equal')
```

Output

ax is bigger than am

Note: when the first characters are same in both strings, then interpreter will check the second characters. As per second characters interpreter will decide the bigger string in the given two strings.

Eg6:

```
st1='python'  
st2='narayana'  
  
if st1>st2:  
    print(st1,'is bigger than',st2)  
elif st2>st1:  
    print(st2,'is bigger than',st1)  
else:  
    print('both',st1,'and',st2,'are equal')
```

Output:

python is bigger than narayana

Note: 'p' in the st1 is bigger than 'n' in the st2 string. So st1 variable value is bigger than st2 variable value.

Eg7:

```
st1='1ax'  
st2='am'  
  
if st1>st2:  
    print(st1,'is bigger than',st2)  
elif st2>st1:  
    print(st2,'is bigger than',st1)  
else:  
    print('both',st1,'and',st2,'are equal')
```

Output

am is bigger than 1ax

Note: here interpreter compares '1' from st1 with 'a' from st2 variable.

Generally characters are bigger than numbers, so st2 variable value is bigger than st1 variable value.

Eg8:

```
st1='2ax'  
st2='2am'  
  
if st1>st2:  
    print(st1,'is bigger than',st2)  
elif st2>st1:  
    print(st2,'is bigger than',st1)  
else:  
    print('both',st1,'and',st2,'are equal')
```

Output

2ax is bigger than 2am

Note: in the above example, first two characters are same from both strings, now interpreter decides based on third character. 'x' is bigger than 'm', so st1 variable value bigger than st2 variable value.

Eg9:

```
st1='123'  
st2='a123'  
  
if st1>st2:  
    print(st1,'is bigger than',st2)  
elif st2>st1:  
    print(st2,'is bigger than',st1)  
else:  
    print('both',st1,'and',st2,'are equal')
```

Output

a123 is bigger than 123

Eg10:

```
st1='ax'  
st2='*m'  
  
if st1>st2:  
    print(st1,'is bigger than',st2)  
elif st2>st1:  
    print(st2,'is bigger than',st1)  
else:  
    print('both',st1,'and',st2,'are equal')
```

Output

ax is bigger than *m

Note: in the above example, characters are bigger than special characters. So here 'a' is bigger than 'a'.

Eg11:

```
st1='1x'  
st2='&am'  
  
if st1>st2:  
    print(st1,'is bigger than',st2)  
elif st2>st1:  
    print(st2,'is bigger than',st1)  
else:  
    print('both',st1,'and',st2,'are equal')
```

Output

1x is bigger than &am

Note: in the above program, interpreter compares '1' from st1 with '&' from st2 .

Generally integers are bigger than special characters

LIST

Data structure:

1. A list is a collection of elements. These elements may be homogeneous or heterogeneous.
2. A list is a value that contains multiple values in an ordered sequence. The term list value refers to the list itself (which is a value that can be stored in a variable or passed to a function like any other value)
3. Just as string values are typed with quote characters to mark where the string begins and ends, a list begins with an opening square bracket and ends with a closing square bracket, [].
4. Values inside the list are also called items. Items are separated with commas (that is, they are comma-delimited).
5. A list also allows duplicate elements.
6. Insertion order is preserved in list.
7. List elements are separated by commas and enclosed within square brackets ([]).
8. Every element in the list has its own unique index number.
9. List supports both forward indexing and backward indexing, forward index starts from 0 and backward index starts from -1.
10. We access either specific element by using **indexing** or set of elements by using **slicing** from the List.
11. We can create list in different ways. Like by using list() function, by using square brackets “[]” and also by using range() function.
12. List objects are mutable.

Creating List by using List()

1. This list() allows only one string value with set of characters.
2. If we give int type data in the list() function then interpreter will throw ‘TypeError’ error.

Eg:

```
>>> List1=list()  
>>> print(List1)  
>>> type(List1) #creating empty list  
[]  
<class 'list'>
```

```
>>> List1=list('python')
>>> print(List1)
['p', 'y', 't', 'h', 'o', 'n']
>>> type(List1)
<class 'list'>
```

Creating list by using square brackets “[]”

```
>>> List1=[]
>>> print(List1)
[]
>>> type(List1)
<class 'list'>

>>> List1=[1,2,3,4,5]
>>> print(List1)
[1, 2, 3, 4, 5]
>>> type(List1)
<class 'list'>

>>> List1=[10,11,'Python',5.5,True,2+3j]
>>> print(List1)
[10, 11, 'Python', 5.5, True, (2+3j)]
>>> type(List1)
<class 'list'>
```

Creating list by using range() function:

We can also use range function to create list

Syn: range(StartingIndexValue, LastValue-1, RangeValue)

Here, both StartingIndexValue and RangeValue are optional.

The default StartingIndexValue is 0.

The default RangeValue is 1.

```
Eg: #in python 2 version
>>> List1=list(range(10))
>>> print(List1)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> type(List1)
<class 'list'>

>>> List1=list(range(0,10))
>>> print(List1)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> type(List1)
<class 'list'>

>>> List1=list(range(2,8))
```

```
>>> print(List1) [2, 3, 4, 5, 6, 7]
>>> type(List1) <class 'list'>

>>> List1=list(range(-4,4))
>>> print(List1) [-4, -3, -2, -1, 0, 1, 2, 3]
>>> type(List1) <class 'list'>

>>> List1=list(range(0,10,2))
>>> print(List1) [0, 2, 4, 6, 8]
>>> type(List1) <class 'list'>
```

Creating a list with split() function:

```
>>> str1='Python is very easy and simple language'
>>> lst=str1.split()
>>> print(lst) ['Python', 'is', 'very', 'easy', 'and', 'simple', 'language']
>>> type(lst) <class 'list'>
```

Creating an empty list:

By using []

```
>>> lst=[]
>>> print(lst)
[]
```

By using list()

```
>>> lst1=list()
>>> lst1
[]
>>> type(lst1) <class 'list'>
```

By using range()

```
>>> lst2=range(0)
>>> lst2
[]
>>> type(lst2) <type 'list'>
```

List Indexing:

1. By using list indexing we can fetch specific element from the list.
2. It supports both forward and backward indexing

Eg:

```
>>> List1=[10,20,30,'Python',True,1.5,2+3j]
```

0 1 2 3 4 5 6
[10,20,30,'Python',True,1.5,2+3j]
-7 -6 -5 -4 -3 -2 -1

```
>>> print(List1[0])                10
>>> print(List1[1])                20
>>> print(List1[2])                30
>>> print(List1[3])                Python
>>> print(List1[4])                True
>>> print(List1[5])                1.5
>>> print(List1[6])                (2+3j)
>>> print(List1[-1])              (2+3j)
>>> print(List1[-2])              1.5
>>> print(List1[-3])              True
>>> print(List1[-4])              Python
>>> print(List1[-5])              30
>>> print(List1[-6])              20
>>> print(List1[-7])              10
```

List Slicing:

1. By using slicing we can fetch set of characters from list.
2. It also supports both forward and backward indexing
3. Colon (:) is the slicing operator.

Eg:

```
>>> List1=[10,20,30,'Python',True,1.5,2+3j]
```

0 1 2 3 4 5 6
[10,20,30,'Python',True,1.5,2+3j]
-7 -6 -5 -4 -3 -2 -1

```
>>> print(List1[0:2])                [10, 20]  
>>> print(List1[2:5])                [30, 'Python', True]  
>>> print(List1[2:])                [30, 'Python', True, 1.5, (2+3j)]  
>>> print(List1[2:-1])                [30, 'Python', True, 1.5]  
>>> print(List1[-4:5])                ['Python', True]  
>>> print(List1[-4:])                ['Python', True, 1.5, (2+3j)]  
>>> print(List1[-5:])                [30, 'Python', True, 1.5, (2+3j)]  
>>> print(List1[-5:-2])                [30, 'Python', True]
```

Using ::

Syn: [starting_number :: increment_value]

The default increment value is 1

```
>>> lst=[10,20,30,40,50,60,70,80,90]

>>> lst[1::]          [20, 30, 40, 50, 60, 70, 80, 90]
>>> lst[2::]          [30, 40, 50, 60, 70, 80, 90]
>>> lst[8::]          [90]
>>> lst[0::8]         [10, 90]
>>> lst[0::7]         [10, 80]
>>> lst[0::4]         [10, 50, 90]
>>> lst[0::2]         [10, 30, 50, 70, 90]
>>> lst[0::1]         [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> lst[3::1]         [40, 50, 60, 70, 80, 90]
>>> lst[3::2]         [40, 60, 80]
>>> lst[3::3]         [40, 70]
>>> lst[-1::]         [90]
>>> lst[-2::]         [80, 90]
>>> lst[-3::]         [70, 80, 90]
>>> lst[-4::]         [60, 70, 80, 90]
>>> lst[-5::]         [50, 60, 70, 80, 90]
>>> lst[-6::]         [40, 50, 60, 70, 80, 90]
>>> lst[-7::]         [30, 40, 50, 60, 70, 80, 90]
>>> lst[-8::]         [20, 30, 40, 50, 60, 70, 80, 90]
>>> lst[-9::]         [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> lst[-9::1]        [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> lst[-9::2]        [10, 30, 50, 70, 90]
>>> lst[-9::3]        [10, 40, 70]
>>> lst[-9::4]        [10, 50, 90]
>>> lst[-9::5]        [10, 60]
>>> lst[-9::-1]       [10]
>>> lst[-9::-2]       [10]
>>> lst[1::-6]         [20]
>>> lst[1::-1]         [20, 10]
>>> lst[1::-2]         [20]
>>> lst[1::-3]         [20]
>>> lst[5::-3]         [60, 30]
>>> lst[5::-4]         [60, 20]
>>> lst[5::-1]         [60, 50, 40, 30, 20, 10]
```

```
>>> lst[::-9]      [90]
>>> lst[::-8]      [90, 10]
>>> lst[::-5]      [90, 40]
>>> lst[::-4]      [90, 50, 10]
>>> lst[::-3]      [90, 60, 30]
>>> lst[::-2]      [90, 70, 50, 30, 10]
>>> lst[::-1]      [90, 80, 70, 60, 50, 40, 30, 20, 10]
```

List is a **mutable** object that means we can alter or replace the existing list object.

```
>>> List1=[10,20,30,'Python',True,1.5,2+3j]
>>> print(List1)          [10, 20, 30, 'Python', True, 1.5, (2+3j)]
>>> id(List1)            52330784
```

```
>>> List1[0]=100         #modifying the content of list
```

```
>>> print(List1)          [100, 20, 30, 'Python', True, 1.5, (2+3j)]
>>> id(List1)            52330784
```

Here, the address of list List1 is not changed before and after modification that's why list is a mutable object.

List Concatenation:

Python supports concatenating two or more lists into single list.

Eg:

```
>>> List1=[10,'Python',5.5]
>>> List2=[20,30,'Narayana',3+4j]
>>> print(List1)          [10, 'Python', 5.5]
>>> print(List2)          [20, 30, 'Narayana', (3+4j)]
>>> type(List1)           <class 'list'>
>>> type(List2)           <class 'list'>

>>> List3=List1+List2
>>> print(List3)          [10, 'Python', 5.5, 20, 30, 'Narayana', (3+4j)]
```

```
>>> type(List3) <class 'list'>
```

List Multiplication or List Repetition

Python supports multiplying the given list into N number of times.

Eg:

```
>>> List1=[10,'Python',5.5]
>>> List5=List1*3
>>> print(List5)
[10, 'Python', 5.5, 10, 'Python', 5.5, 10, 'Python', 5.5]
```

List functions:

1. **len():** this function counts no.of elements in the list.

Eg:

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]
>>> len(List1) 8
```

2. **Count():** this function counts the no.of occurrences of specific element in the list.

Eg:

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]
>>> List1.count(10) 2
```

Eg:

```
>>> lst=[1,2,3,True,False,1,0,False]
>>> lst.count(1) 3
>>> lst.count(True) 3
```

Eg:

```
>>> lst=[1,2,3,True,False,1,0,False,1+9j,1.1]
>>> lst.count(0) 3
>>> lst.count(False) 3
```

3. **Index():** this function finds the index value for specific element.

Eg 1:

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]
>>> List1.index(10)          0
```

Eg:

```
>>> lst=[1,2,3,True,False,1,0,False,1+9j,1.1]
>>> lst.index(1,1)          3
```

Eg 2:

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]
>>> List1.index(10,1)       6
```

Index on nested list:

Eg1

```
>>> lst =[100, True, 1, 2, 3, 4, 5, 0, [8, 9], 10, 20, 4]
>>> lst[8].index(9)        1
>>> lst[8].index(8)        0
```

Eg2:

```
>>> lst=[10,20,30,40,[50,60,[100,200,300],70],80,90]
>>> lst.index(10)          0
>>> lst.index(20)          1
>>> lst.index(30)          2
>>> lst.index(40)          3
>>> lst.index(50)          ValueError: 50 is not in list
>>> lst.index([50,60,[100,200,300],70]) 4
>>> lst[4].index(50)       0
>>> lst[4].index(60)       1
>>> lst[4].index([100,200,300]) 2
>>> lst[4][2].index(100)    0
>>> lst[4][2].index(200)    1
>>> lst[4][2].index(300)    2
>>> lst[4].index(70)        3
>>> lst.index(80)          5
>>> lst.index(90)          6
```

List is a dynamic object that means we can increase or decrease the length of the list.

We use append(), extend() and insert() functions to increase the length of the list

We use remove() and pop() to decrease the length of the list.

Increasing the length of the list

4. **Append():** this function adds new element at the end of the existing list.

Eg 1:

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]
>>> List1.append(50)
>>> print(List1) [10, 20, 'Python', 30, True, 'Narayana', 10, (3+4j), 50]
```

Eg 2:

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]
>>> List1.append(60)
>>> print(List1) [10, 20, 'Python', 30, True, 'Narayana', 10, (3+4j), 60]
```

We can also add multiple elements in the list by using append function but those multiple elements work like nested list or sub list in the existing list.

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]
>>> List1.append([0,1,2])
>>> print(List1) [10, 20, 'Python', 30, True, 'Narayana', 10, (3+4j), [0, 1, 2]]
```

5. **Extend():** this function adds one or multiple elements at the end of the existing list

Eg1:

```
>>> lst=[10,20,30,40]
>>> lst.extend([50])
>>> print(lst) [10, 20, 30, 40, 50]
>>> type(lst) <class 'list'>
>>> id(lst) 94793640
```

Eg 2:

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]
>>> List1.extend([70,80,90])
>>> print(List1) [10, 20, 'Python', 30, True, 'Narayana', 10, (3+4j), 70, 80, 90]
```

6. **Insert():** this function adds a element at any required index place in the existing list.

Syn: `insert(index_number,element(s))`

Eg 1:

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]
>>> List1.insert(2,100)
>>> print(List1) [10, 20, 100, 'Python', 30, True, 'Narayana', 10, (3+4j)]
```

Eg 2:

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]
>>> List1.insert(3,'Krishna')
>>> print(List1) [10, 20, 'Python', 'Krishna', 30, True, 'Narayana', 10, (3+4j)]
```

We can also add multiple elements in the list at required place by using insert method but those multiple elements work like nested list or sub list in the existing list

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]
>>> List1.insert(0,[0,1,2,3])
>>> print(List1) [[0, 1, 2, 3], 10, 20, 'Python', 30, True, 'Narayana', 10, (3+4j)]
```

Decreasing the length of list

7. **Remove():**

- This function removes specific element in the existing list.
- This function allows one argument and that should be element name.
- If that specified is not available in the existing list the interpreter will ValueError.
- If we don't specify any element as argument in the remove function then interpreter will return TypeError.

Eg 1:

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]

>>> List1.remove(10)
>>> print(List1) [20, 'Python', 30, True, 'Narayana', 10, (3+4j)]
```

Eg 2:

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]
>>> List1.remove(True)
```

```
>>> print(List1) [10, 20, 'Python', 30, 'Narayana', 10, (3+4j)]
```

Eg3:

```
lst=[10,20,30,40]
```

```
>>> print(lst) [10, 20, 30, 40]  
>>> type(lst) <class 'list'>  
>>> id(lst) 94793520
```

```
>>> lst.remove(50) ValueError: list.remove(x): x not in list
```

Eg4:

```
>>> lst=[10,20,30,40]
```

```
>>> print(lst) [10, 20, 30, 40]  
>>> type(lst) <class 'list'>  
>>> id(lst) 94793520
```

```
>>> lst.remove() TypeError: remove() takes exactly one argument (0 given)
```

8. Pop():

- a. This function removes specific element based on its index position.
- b. This function allows only one argument and that should be index number of an element.
- c. If we don't specify any index number in the pop() as an argument, then interpreter will remove last element from the list
- d. If we specify the out of index range number as an argument in pop() then it will return IndexError.

Eg 1:

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]  
>>> List1.pop(1) 20  
>>> print(List1) [10, 'Python', 30, True, 'Narayana', 10, (3+4j)]
```

Eg 2:

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]  
>>> List1.pop(2) 'Python'  
>>> print(List1) [10, 20, 30, True, 'Narayana', 10, (3+4j)]
```

Eg3:

```
List1=[10,20,'Python',30,True,'Narayana',10,3+4j]
```

```
>>> List1.pop()  
(3+4j)
```

```
>>> print(List1) [10, 20, 'Python', 30, True, 'Narayana', 10]
```

```
>>> List1.pop()  
10
```

```
>>> print(List1) [10, 20, 'Python', 30, True, 'Narayana']
```

Eg4:

```
>>> List1=[10, 20, 'Python', 30, True, 'Narayana']
```

```
>>> List1.pop(7) IndexError: pop index out of range
```

9. Reverse(): this function reverses the existing list.

Eg 1:

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]  
>>> List1.reverse()  
>>> print(List1) [(3+4j), 10, 'Narayana', True, 30, 'Python', 20, 10]
```

10. Copy(): this function copies the existing list into new variable.

Eg:

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]  
>>> x=List1.copy()  
>>> print(List1) [10, 20, 'Python', 30, True, 'Narayana', 10, (3+4j)]  
>>> print(x) [10, 20, 'Python', 30, True, 'Narayana', 10, (3+4j)]
```

11. Clear(): this function clears or removes all elements of the entire list.

Eg:

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]
```

```
>>> List1.clear()  
>>> print(List1)  
[]
```

12. **Max()**: this function finds the maximum value in the given list

Eg:

```
>>> List2=[10,20,30,40]  
>>> max(List2)  
40
```

13. **Min()**: this function finds the minimum value in the given list

Eg:

```
>>> List2=[10,20,30,40]  
>>> min(List2)  
10
```

14. **sort()**: this function sorts the elements.

```
>>> lst=[1,9,5,11,2]  
>>> lst.sort()  
>>> print(lst)  
[1, 2, 5, 9, 11]  
  
>>> l1=[1,2,5,3,7,4,2,True]  
>>> l1.sort()  
>>> print(l1)  
[1, True, 2, 2, 3, 4, 5, 7]
```

Note: by default this function sorts in ascending order, we can also get in descending order by setting **True** for **reverse**.

Eg 1:

```
>>> lst=[1,9,5,11,2]  
>>> lst.sort(reverse=True)  
>>> print(lst)  
[11, 9, 5, 2, 1]
```

Eg 2:

```
>>> l1=[1,2,5,3,7,4,2,True]
```

```
>>> l1.sort(reverse=True)
>>> print(l1) [7, 5, 4, 3, 2, 2, 1, True]
```

• **del command:**

This command is used to remove any specific element in the list or to remove entire list object permanently.

• **Removing specific element by using del command**

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]
>>> del List1[0]
>>> print(List1) [20, 'Python', 30, True, 'Narayana', 10, (3+4j)]

>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]
>>> del List1[4]
>>> print(List1) [10, 20, 'Python', 30, 'Narayana', 10, (3+4j)]
```

• **Removing entire list object by using del command**

Eg:

```
>>> List1=[10,20,'Python',30,True,'Narayana',10,3+4j]
>>> del List1 #deleting lst
>>> print(List1) #after deleting
NameError: name 'lst' is not defined
```

Nested List:

Python supports Nested lists also, it means a list contains another lists.

Eg1:

```
>>> List1=[10,'Python',5.5]
>>> List2=[20,30,'Narayana',3+4j]
>>> List3=[1,True,2,'Durga']

>>> print(List1)           [10, 'Python', 5.5]
>>> print(List2)           [20, 30, 'Narayana', (3+4j)]
>>> print(List3)           [1, True, 2, 'Durga']

>>> NestList=[List1,List2,List3] #creating a list by using existing lists.

>>> print(NestList)         [[10, 'Python', 5.5], [20, 30, 'Narayana', (3+4j)], [1, True, 2, 'Durga']]

>>> type(NestList)          <class 'list'>
>>> print(NestList[0])       [10, 'Python', 5.5]
>>> print(NestList[1])       [20, 30, 'Narayana', (3+4j)]
>>> print(NestList[2])       [1, True, 2, 'Durga']

>>> print(NestList[0][0])    10
>>> print(NestList[0][1])    'Python'
>>> print(NestList[0][2])    5.5
>>> print(NestList[1][0])    20
>>> print(NestList[1][1])    30
>>> print(NestList[1][2])    'Narayana'
>>> print(NestList[1][3])    (3+4j)
```

```
>>> print(NestList[2][0])      1
>>> print(NestList[2][1])      True
>>> print(NestList[2][2])      2
>>> print(NestList[2][3])      'Durga'
```

Eg2: Performing Len(), Count() and Index() on nested list

```
>>> lst=[[100,200],[10,20,30,20,20],[True,1,False,0],[100,200]]
>>> len(lst)                  4
>>> lst.count([100,200])      2
>>> lst.index([True,1,False,0]) 2

>>> lst[1]                     [10, 20, 30, 20, 20]
>>> len(lst[1])                5
>>> lst[1].count(20)           3
>>> lst[1].count(10)           1
>>> lst[1].index(30)           2
>>> lst[1].index(20)           1

>>> lst[2]                     [True, 1, False, 0]
>>> len(lst[2])                4
>>> lst[2].count(True)          2
>>> lst[2].count(0)             2
>>> lst[2].index(True)          0
>>> lst[2].index(False)         2
>>> lst[2].index(False,3)       3

>>> lst[3]                     [100, 200]
>>> len(lst[3])                2
>>> lst[3].count(100)           1
>>> lst[3].count(200)           1
>>> lst[3].index(100)           0
>>> lst[3].index(200)           1
```

Working with index method on nested list:

```
>>> List1=[10,'Python',5.5]  
>>> List2=[20,30,'Narayana',3+4j]  
>>> List3=[1,True,2,'Durga']
```

```
>>> NestList=[List1,List2,List3]
```

```
>>> print(NestList) [[10, 'Python', 5.5], [20, 30, 'Narayana', (3+4j)], [1, True, 2,  
'Durga']]
```

————— 0 ————— 1 ————— 2 ————— .
0 1 2 0 1 2 3 0 1 2 3
[[10, 'Python', 5.5], [20, 30, 'Narayana', (3+4j)], [1, True, 2, 'Durga']]

```
>>> NestList.index([10,'Python',5.5]) 0
```

```
>>> NestList[0].index(10) 0
```

```
>>> NestList[0].index('Python') 1
```

```
>>> NestList[0].index(5.5) 2
```

```
>>> NestList.index([20,30,'Narayana',(3+4j)]) 1
```

```
>>> NestList[1].index(20) 0
```

```
>>> NestList[1].index(30) 1
```

```
>>> NestList[1].index('Narayana') 2
```

```
>>> NestList[1].index((3+4j)) 3
```

```
>>> NestList.index([1, True, 2, 'Durga']) 2
```

```
>>> NestList[2].index(1) 0
```

```
>>> NestList[2].index(True)          0  
  
>>> NestList[2].index(True,1)       1  
  
>>> NestList[2].index(2)           2  
  
>>> NestList[2].index('Durga')     3
```

Adding or removing elements in the nested list

Increasing the length of nested list

```
>>> lst=[5,[2,'a',3],20,30,40]
```

By using append()

```
>>> lst[1].append(4)  
>>> print(lst)      [5, [2, 'a', 3, 4], 20, 30, 40]  
  
>>> lst[1].append(5)  
>>> print(lst)      [5, [2, 'a', 3, 4, 5], 20, 30, 40]
```

By using extend()

```
>>> lst[1].extend(['x','y','z'])  
>>> print(lst)      [5, [2, 'a', 3, 4, 5, 'x', 'y', 'z'], 20, 30, 40]
```

By using insert()

```
>>> lst[1].insert(1,10)  
>>> print(lst)      [5, [2, 10, 'a', 3, 4, 5, 'x', 'y', 'z'], 20, 30, 40]  
  
>>> lst[1].insert(4,'Narayana')  
>>> print(lst)  [5, [2, 10, 'a', 3, 'Narayana', 4, 5, 'x', 'y', 'z'], 20, 30, 40]
```

Decreasing the length of nested list

```
>>> lst      [5, [2, 10, 'a', 3, 'Narayana', 4, 5, 'x', 'y', 'z'], 20, 30, 40]
```

By using remove()

```
>>> lst[1].remove(10)
```

```
>>> print(lst) [5, [2, 'a', 3, 'Narayana', 4, 5, 'x', 'y', 'z'], 20, 30, 40]
```

```
>>> lst[1].remove('Narayana')
>>> print(lst) [5, [2, 'a', 3, 4, 5, 'x', 'y', 'z'], 20, 30, 40]
```

By using pop()

```
>>> lst[1].pop(0) 2
>>> print(lst) [5, ['a', 3, 4, 5, 'x', 'y', 'z'], 20, 30, 40]
```

```
>>> lst[1].pop(1) 3
>>> print(lst) [5, ['a', 4, 5, 'x', 'y', 'z'], 20, 30, 40]
```

By using del command

```
>>> del lst[1][0]
>>> print(lst) [5, [4, 5, 'x', 'y', 'z'], 20, 30, 40]
```

```
>>> del lst[1][1]
>>> print(lst) [5, [4, 'x', 'y', 'z'], 20, 30, 40]
```

By using clear()

```
>>> lst[1].clear()
>>> print(lst) [5, [], 20, 30, 40]
```

Adding elements to empty nested list

```
>>> lst[1].append('python')
>>> print(lst) [5, ['python'], 20, 30, 40]
```

```
>>> lst[1].extend(['Narayana','Django',10,20])
>>> print(lst) [5, ['python', 'Narayana', 'Django', 10, 20], 20, 30, 40]
```

```
>>> lst[1].insert(0,100)
>>> print(lst) [5, [100, 'python', 'Narayana', 'Django', 10, 20], 20, 30, 40]
```

Conversions:

Converting a String into List

```
>>> Str1='Python is very simple and easy language'  
  
>>> print(Str1)          Python is very simple and easy language  
>>> type(Str1)           <class 'str'>  
  
>>> List1=Str1.split()  
  
>>> print(List1)         ['Python', 'is', 'very', 'simple', 'and', 'easy', 'language']  
>>> type(List1)          <class 'list'>
```

Converting a List to String

```
>>> List1=['Python', 'is', 'very', 'simple', 'and', 'easy', 'language']  
  
>>> print(List1)         ['Python', 'is', 'very', 'simple', 'and', 'easy', 'language']  
>>> type(List1)          <class 'list'>  
  
>>> Str2=" ".join(List1)  
  
>>> print(Str2)          Python is very simple and easy language  
>>> type(Str2)           <class 'str'>
```

The main difference between **String** and **List** is mutation.

1. String is immutable whereas List is mutable.
2. Mutable objects can be altered whereas immutable objects can't be altered.

Eg: **String**

```
>>> Str1='Python'  
>>> print(Str1)          Python  
>>> id(Str1)             43143520  
  
>>> Str1[0]='S'          #trying to replace 'P' with 'S' in 'Python' string  
  
Error: TypeError: 'str' object does not support item assignment
```

We can add new letter to the existing String Str1 but now it will create as a new String Str1 variable in the memory.

```
>>> Str1=Str1+'s'  
>>> print(Str1)          Pythons  
>>> id(Str1)            53595200
```

Eg: List

```
>>> Lst1=[10,20,30,40,'Guido',50]  
>>> print(Lst1)          [10, 20, 30, 40, 'Guido', 50]  
>>> id(Lst1)            53663224  
  
>>> Lst1[4]='Rossum'      #trying to replace 'Guido' with 'Rossum' in Lst1  
>>> print(Lst1)          [10, 20, 30, 40, 'Rossum', 50]  
>>> id(Lst1)            53663224
```

List Packing

A list can be created by using a group of variables, it is called list packing

```
>>> a=10  
>>> b=20  
>>> c=True  
>>> d='Py'  
  
>>> list1=[a,b,c,d]  
>>> print(list1)          [10, 20, True, 'Py']  
>>> type(list1)           <class 'list'>
```

List Unpacking

1. List unpacking allows to extract all list elements automatically into different variables.
2. The number of variables must be equal to number of elements in the list.

```
>>> lst=[10,20,'Python',True]  
>>> print(lst)            [10, 20, 'Python', True]
```

```
>>> type(lst)           <class 'list'>  
  
>>> a,b,c,d=lst      #list unpacking  
  
>>> print(a)          10  
>>> type(a)           <class 'int'>  
  
>>> print(b)          20  
>>> type(b)           <class 'int'>  
  
>>> print(c)          Python  
>>> type(c)           <class 'str'>  
  
>>> print(d)          True  
>>> type(d)           <class 'bool'>
```

How to generate a list as per user requirements?

```
start_val=int(input('Enter start value: '))  
end_val=int(input('Enter end value: '))  
step_size=int(input('Enter increment value: '))  
  
UserList=range(start_val,end_val,step_size)  
  
print(UserList)
```

output in Python3

```
Enter start value: 10  
Enter end value: 21  
Enter increment value: 2  
range(10, 21, 2)
```

output in Python2

```
Enter start value: 10  
Enter end value: 21  
Enter increment value: 2  
[10, 12, 14, 16, 18, 20]
```

Adding different lists

We can add elements of different lists by using lambda and map functions
(these functions are clearly explained in the functions concept)

```
s=[1,20,3]
s1=[1,2,3]
print(list(map(lambda x,y:x+y,s,s1)))

s=[1,20,3]
s1=[1,2,3,8]
print([x+y for x,y in zip(s,s1)])

s=[1,20,3,6,8]
s1=[1,2,3,8]
print([x+y for x,y in zip(s,s1)])
output
[2, 22, 6]
[2, 22, 6]
[2, 22, 6, 14]
```

List comprehension:

It provides an easy way to create list objects from any iterable objects based on some conditions.

Syntax: list=[expression for iterative_item in list if condition]

How to display squares for all elements in the given list?

```
>>> lst=[1,2,5,4,3]

>>> lst1=[x*x for x in lst]

>>> print(lst1)          [1, 4, 25, 16, 9]
>>> type(lst1)           <class 'list'>
```

How to display squares for all even elements in the given list?

```
>>> lst=[1,2,5,4,3]
```



```
>>> lst2=[x*x for x in lst if x%2==0]
>>> print(lst2)
[4, 16]
>>> type(lst2)
<class 'list'>
```

How to display all unmatching elements from list1, those elements must not be in list2?

```
>>> lst1=[1,2,3,4,5]
>>> lst2=[1,2,3,6,7]
>>> lst3=[i for i in lst1 if i not in lst2]
>>> lst3
[4, 5]
>>> type(lst3)
<class 'list'>
```

Eg:

```
name=['narayana','sai','krishna','veni']
>>> list1=[[n.upper(),n.capitalize(),n.title(),len(n)] for n in name]
>>> print(list1)
[['NARAYANA', 'Narayana', 'Narayana', 8], ['SAI', 'Sai', 'Sai', 3], ['KRISHNA', 'Krishna', 'Krishna', 7], ['VENI', 'Veni', 'Veni', 4]]
```

List comparison

When comparing lists, the elements will compared from both lists parallelly.

If the first elements from both lists are same then interpreter will compare the second elements from both lists,

If second elements from both lists are also same then interpreter will compare the third elements from both lists.

Syn: cmp(list1,list2)

If list1 is bigger than list2 then interpreter will return 1

If list1 is smaller than list2 then interpreter will return -1

If list1 and list2 are same then interpreter will return 0

Eg1:

```
>>> lst1=[1,2,3]  
>>> lst2=[1,3,5,6]
```

```
>>> print cmp(lst1,lst2)           -1
```

Explanation: here the first element of lst1 is '1' and first element of list2 is '1', so both are same. Now interpreter will compare second elements, like 2 in the lst1 and 3 in the lst2.
2 from lst1 is smaller than 3 from lst2, so interpreter returned -1.

Eg2:

```
>>> lst1=[10,20,30]  
>>> lst2=[10,11,12,13]
```

```
>>> print cmp(lst1,lst2)           1
```

Explanation: here first elements both list are same. 20 from lst1 is bigger than 11, so interpreter returned 1.

Eg3:

```
>>> lst1=[10,20,30]  
>>> lst2=[10,20,30]
```

```
>>> print cmp(lst1,lst2)           0
```

Explanation: here first elements from both lists are same. So interpreter checked second elements from lists, second elements are also same from both lists. Interpreter checked third elements, but third elements are also same. Finally all elements are same from both lists, so interpreter returned 0.

Eg4:

```
>>> lst1=[10,20,30,40]  
>>> lst2=[10,11,12]
```

```
>>> print cmp(lst1,lst2) 1
```

Explanation: first elements are same, 20 from lst1 is bigger than 11 from lst2. So interpreter returned 1.

Eg5:

```
>>> lst1=[1,2,3]  
>>> lst2=[10,20]
```

```
>>> print cmp(lst1,lst2) -1
```

Explanation: 1 from lst1 is smaller than 10 from lst2, so interpreter returned -1.

Eg6:

```
>>> lst1=[10]  
>>> lst2=[1,2,3,4]
```

```
>>> print cmp(lst1,lst2) 1
```

Explanation: 10 from lst1 is bigger than 1 from lst2, so interpreter returned 1

Eg7:

```
>>> lst1=[1,2]  
>>> lst2=[1,2,3]
```

```
>>> print cmp(lst1,lst2) -1
```

Explanation: first two elements(1,2) are same from both lists. Lst1 has no third element but lst2 has 3rd element, that means lst1 is smaller than lst2. So interpreter returned -1.

Eg8:

```
>>> lst1=['a',1,2]  
>>> lst2=[10,20]
```

```
>>> print cmp(lst1,lst2) 1
```

Explanation: lst1 has 'a' as first element and lst2 has 10 as first element. Generally characters are bigger than numbers. So lst1 is bigger than lst2 that's why interpreter returned 1.

Eg9:

```
>>> lst1=[100,1,2]  
>>> lst2=['b',10,20]
```

```
>>> print cmp(lst1,lst2) -1
```

Eg10:

```
>>> lst1=['a','x',1,2]  
>>> lst2=['a','x',10,20]
```



```
>>> print cmp(lst1,lst2) -1
```

Eg11:
>>> lst1=['a','x',1,2]
>>> lst2=['a','x',0,20]

```
>>> print cmp(lst1,lst2) 1
```

Eg12:
>>> lst1=['python',True,'x',1,2]
>>> lst2=['Narayana',False,'x',0,20]

```
>>> print cmp(lst1,lst2) 1
```

Eg13:
>>> lst1=[True,'py','x',1,2]
>>> lst2=[1,'p','Narayana',False,'x',0,20]

```
>>> print cmp(lst1,lst2) 1
```

TUPLE

Data structure

1. Tuple is used to represent a set of homogeneous or heterogeneous elements into a single entity.
2. Tuple objects are immutable that means once if we create a tuple later we cannot modify that tuple object.
3. All elements are separated by commas (,) and enclosed by parentheses. Parentheses are optional.
4. Tuple allows duplicate elements.
5. Every element in the tuple has its own index number
6. Tuple supports both forward indexing and also backward indexing, forward indexing starts from 0 and backward indexing starts from -1.
7. If we take only one element in the tuple then we should use comma (,) after that single element.
8. Tuples can be used as keys to the dictionary.
9. We can create a tuple in different ways, like with tuple(), with () or without () also.
10. The main difference between lists and tuples is- Lists are enclosed in brackets ([]) and their elements and size can be changed, while tuples are enclosed in parentheses (()) and cannot be updated.

Creating a tuple with tuple():

Eg1:

```
>>> tup=tuple([10,20,30,True,'Python'])  
>>> print(tup)                                (10, 20, 30, True, 'Python')  
>>> type(tup)                                <class 'tuple'>  
>>> id(tup)                                    52059760
```

Eg2:

```
>>> t1=tuple([10,20,30,40,50])  
>>> print(t1)                                (10, 20, 30, 40, 50)  
>>> type(t1)                                <class 'tuple'>  
>>> id(t1)                                    96785344
```

Creating an empty tuple:

Eg:

```
>>> tup=()                                     #creating empty tuple
>>> print(tup)                                ()
>>> type(tup)                                 <class 'tuple'>
>>> id(tup)                                    23134256
```

Eg2:

```
>>> t1=tuple()
>>> print(t1)                                ()
>>> type(t1)                                 <class 'tuple'>
>>> id(t1)                                    51183664
```

Creating a tuple with ():

Eg:

```
>>> tup2=(10,20,30,40,50)                      #creating homogeneous tuple
>>> print(tup2)                                (10, 20, 30, 40, 50)
>>> type(tup2)                                 <class 'tuple'>
>>> id(tup2)                                    63484864
```

Creating a tuple without ():

Eg:

```
>>> tup=10,20,True,'Py'                        #creating tuple without parenthesis
>>> print(tup)                                (10, 20, True, 'Py')
>>> type(tup)                                 <class 'tuple'>
>>> id(tup)                                    67086688
```

Creating a tuple with heterogeneous elements:

Eg:

```
>>> tup1=(10,20,30,True,"Python",10.5,3+5j)  #creating heterogeneous tuple
>>> print(tup1)                                (10, 20, 30, True, 'Python', 10.5, (3+5j))
>>> type(tup1)                                 <class 'tuple'>
>>> id(tup1)                                    58963648
```

Creating a tuple with single element:

Creating a tuple with a **single element** is tricky, if we take only one element then the type that tuple will be based on specified element.

```
>>> t2=(1)
>>> t2
1
>>> type(t2)
<type 'int'>
```

```
>>> t2=(True)
>>> print(t2)
True
>>> type(t2)
<type 'bool'>
```

```
>>> t2=('py')
>>> print(t2)
py
>>> type(t2)
<class 'str'>
>>> id(t2)
56695168
```

```
>>> t2=(False)
>>> print(t2)
False
>>> type(t2)
<type 'bool'>
```

```
>>> t3=(10.8)
>>> print(t3)
10.8
>>> type(t3)
<class 'float'>
>>> id(t3)
89797136
```

```
>>> t4=(4+5j)
>>> print(t4)
(4+5j)
>>> type(t4)
<class 'complex'>
>>> id(t4)
94356568
```

So to solve the above problem we should use comma (,) after the element in the tuple, like

```
>>> print(t2)      ('py')
>>> type(t2)       <class 'tuple'>
>>> id(t2)         96772080

>>> t2=(1,)
>>> print(t2)      1
>>> type(t2)       <type 'tuple'>
```

```
>>> t3=(True,)
>>> print(t3)      (True,)
>>> type(t3)       <class 'tuple'>
>>> id(t3)         85294192
```

Tuple Indexing:

Tuple indexing is nothing but fetching a specific element from the existing tuple by using its index value.

Eg:

```
>>> tup=(10,20,30,True,"Python",10.5,3+5j,10)

>>> print(tup)          (10, 20, 30, True, 'Python', 10.5, (3+5j), 10)

>>> type(tup)          <class 'tuple'>

>>> id(tup)            63560624
```

0 1 2 3 4 5 6 7
(10,20,30,True,"Python",10.5,3+5j,10)

-8 -7 -6 -5 -4 -3 -2 -1

```
>>> tup[0]             10
```

```
>>> tup[1]             20
```

```
>>> tup[2]           30
>>> tup[3]           True
>>> tup[4]          'Python'
>>> tup[5]          10.5
>>> tup[6]          (3+5j)
>>> tup[7]           10
>>> tup[-1]          10
>>> tup[-2]          (3+5j)
>>> tup[-3]          10.5
>>> tup[-4]          'Python'
>>> tup[-5]          True
>>> tup[-6]          30
>>> tup[-7]          20
>>> tup[-8]          10
```

Tuple Slicing:

Tuple slicing is nothing but fetching a sequence of elements from the existing tuple by using their index values.

```
>>> tup=(10,20,30,True,"Python",10.5,3+5j,10)
>>> print(tup)          (10, 20, 30, True, 'Python', 10.5, (3+5j), 10)
>>> type(tup)          <class 'tuple'>
>>> id(tup)            63560496
```

0 1 2 3 4 5 6 7
(10,20,30,True,"Python",10.5,3+5j,10)
-8 -7 -6 -5 -4 -3 -2 -1

```
>>> tup[0:4]          (10, 20, 30, True)
>>> tup[0:0]          ()
>>> tup[0:1]          (10,)
>>> tup[0:5]          (10, 20, 30, True, 'Python')
>>> tup[3:5]          (True, 'Python')
>>> tup[2:-2]          (30, True, 'Python', 10.5)
>>> tup[-5:-2]         (True, 'Python', 10.5)
>>> tup[-5:]           (True, 'Python', 10.5, (3+5j), 10)
>>> tup[6:]             ((3+5j), 10)
```

Eg1:

```
tuple1 = ( 'Narayana', 1037 , 1000, 'Python', True )
tinytuple = ('Super', 'Django',True)
print (tuple1)                  # Prints complete tuple
print (tuple1[0])                # Prints first element of the tuple
print (tuple1[1:3])              # Prints elements starting from 2nd till 3rd
print (tuple1[2:])                # Prints elements starting from 3rd element
print (tinytuple * 2)              # Prints tuple two times
print (tuple1 + tinytuple)         # Prints concatenated tuple
```

Output:

```
('Narayana', 1037, 1000, 'Python', True)
Narayana
(1037, 1000)
(1000, 'Python', True)
```

```
('Super', 'Django', True, 'Super', 'Django', True)
('Narayana', 1037, 1000, 'Python', True, 'Super', 'Django', True)
```

Using ::

Eg:

```
>>> t=(10,'py',30,40,10,70,'Narayana','Python')

>>> t[1::]      ('py', 30, 40, 10, 70, 'Narayana', 'Python')
>>> t[2::]      (30, 40, 10, 70, 'Narayana', 'Python')
>>> t[3::]      (40, 10, 70, 'Narayana', 'Python')
>>> t[4::]      (10, 70, 'Narayana', 'Python')
>>> t[5::]      (70, 'Narayana', 'Python')
>>> t[6::]      ('Narayana', 'Python')
>>> t[7::]      ('Python',)
>>> t[8::]      ()

>>> t[1::1]    ('py', 30, 40, 10, 70, 'Narayana', 'Python')
>>> t[1::2]    ('py', 40, 70, 'Python')
>>> t[1::3]    ('py', 10, 'Python')
>>> t[1::4]    ('py', 70)
>>> t[1::5]    ('py', 'Narayana')
>>> t[1::6]    ('py', 'Python')
>>> t[1::7]    ('py',)
>>> t[1::8]    ('py',)

>>> t[2::0]    ValueError: slice step cannot be zero
>>> t[2::1]    (30, 40, 10, 70, 'Narayana', 'Python')
>>> t[2::2]    (30, 10, 'Narayana')
>>> t[2::3]    (30, 70)
>>> t[2::4]    (30, 'Narayana')
>>> t[2::5]    (30, 'Python')
>>> t[2::6]    (30,)
>>> t[2::7]    (30,)

>>> t[3::7]    (40,)
>>> t[3::1]    (40, 10, 70, 'Narayana', 'Python')
>>> t[3::2]    (40, 70, 'Python')
>>> t[3::3]    (40, 'Narayana')
>>> t[3::4]    (40, 'Python')
>>> t[3::5]    (40,)
>>> t[3::6]    (40,)
```

```
>>> t[4::1]      (10, 70, 'Narayana', 'Python')
>>> t[4::2]      (10, 'Narayana')
>>> t[4::3]      (10, 'Python')
>>> t[4::4]      (10,)
>>> t[4::5]      (10,)

>>> t[5::1]      (70, 'Narayana', 'Python')
>>> t[5::2]      (70, 'Python')
>>> t[5::3]      (70,)
>>> t[5::4]      (70,)
>>> t[5::5]      (70,)

>>> t[6::5]      ('Narayana',)
>>> t[6::1]      ('Narayana', 'Python')
>>> t[6::2]      ('Narayana',)
>>> t[6::3]      ('Narayana',)

>>> t[7::1]      ('Python',)
>>> t[7::2]      ('Python',)

>>> t[8::1]      ()

>>> t[1::-1]     ('py', 10)
>>> t[1::-2]     ('py',)
>>> t[1::-3]     ('py',)
>>> t[1::-4]     ('py',)

>>> t[2::-1]     (30, 'py', 10)
>>> t[2::-2]     (30, 10)
>>> t[2::-3]     (30,)

>>> t[3::-1]     (40, 30, 'py', 10)
>>> t[3::-2]     (40, 'py')
>>> t[3::-3]     (40, 10)
>>> t[3::-4]     (40,)
>>> t[3::-5]     (40,)

>>> t[4::-1]     (10, 40, 30, 'py', 10)
>>> t[4::-2]     (10, 30, 10)
>>> t[4::-3]     (10, 'py')
>>> t[4::-4]     (10, 10)
>>> t[4::-5]     (10,)
>>> t[4::-6]     (10,)
```

```
>>> t[5::-1]          (70, 10, 40, 30, 'py', 10)
>>> t[5::-2]          (70, 40, 'py')
>>> t[5::-3]          (70, 30)
>>> t[5::-4]          (70, 'py')
>>> t[5::-5]          (70, 10)
>>> t[5::-6]          (70,)
>>> t[5::-7]          (70,)

>>> t[6::-1]          ('Narayana', 70, 10, 40, 30, 'py', 10)
>>> t[6::-2]          ('Narayana', 10, 30, 10)
>>> t[6::-3]          ('Narayana', 40, 10)
>>> t[6::-4]          ('Narayana', 30)
>>> t[6::-5]          ('Narayana', 'py')
>>> t[6::-6]          ('Narayana', 10)
>>> t[6::-7]          ('Narayana',)
>>> t[6::-8]          ('Narayana',)

>>> t[7::-1]          ('Python', 'Narayana', 70, 10, 40, 30, 'py', 10)
>>> t[7::-2]          ('Python', 70, 40, 'py')
>>> t[7::-3]          ('Python', 10, 'py')
>>> t[7::-4]          ('Python', 40)
>>> t[7::-5]          ('Python', 30)
>>> t[7::-6]          ('Python', 'py')
>>> t[7::-7]          ('Python', 10)
>>> t[7::-8]          ('Python',)
>>> t[7::-9]          ('Python',)

>>> t[8::-1]          ('Python', 'Narayana', 70, 10, 40, 30, 'py', 10)

>>> t[-1::]            ('Python',)
>>> t[-2::]            ('Narayana', 'Python')
>>> t[-3::]            (70, 'Narayana', 'Python')
>>> t[-4::]            (10, 70, 'Narayana', 'Python')
>>> t[-5::]            (40, 10, 70, 'Narayana', 'Python')
>>> t[-6::]            (30, 40, 10, 70, 'Narayana', 'Python')
>>> t[-7::]            ('py', 30, 40, 10, 70, 'Narayana', 'Python')
>>> t[-8::]            (10, 'py', 30, 40, 10, 70, 'Narayana', 'Python')

>>> t[-1::1]           ('Python',)
>>> t[-1::2]           ('Python',)
>>> t[-1::3]           ('Python',)
>>> t[-1::4]           ('Python',)
>>> t[-1::5]           ('Python',)
```

```
>>> t[-1::6]      ('Python',)
>>> t[-1::7]      ('Python',)
>>> t[-1::8]      ('Python',)

>>> t[-2::1]      ('Narayana', 'Python')
>>> t[-2::2]      ('Narayana',)
>>> t[-2::3]      ('Narayana',)
>>> t[-2::3]      ('Narayana',)

>>> t[-3::1]      (70, 'Narayana', 'Python')
>>> t[-3::2]      (70, 'Python')
>>> t[-3::3]      (70,)
>>> t[-3::4]      (70,)

>>> t[-4::1]      (10, 70, 'Narayana', 'Python')
>>> t[-4::2]      (10, 'Narayana')
>>> t[-4::3]      (10, 'Python')
>>> t[-4::4]      (10,)
>>> t[-4::5]      (10,)

>>> t[-5::1]      (40, 10, 70, 'Narayana', 'Python')
>>> t[-5::2]      (40, 70, 'Python')
>>> t[-5::3]      (40, 'Narayana')
>>> t[-5::4]      (40, 'Python')
>>> t[-5::5]      (40,)
>>> t[-5::6]      (40,)

>>> t[-6::1]      (30, 40, 10, 70, 'Narayana', 'Python')
>>> t[-6::2]      (30, 10, 'Narayana')
>>> t[-6::3]      (30, 70)
>>> t[-6::4]      (30, 'Narayana')
>>> t[-6::5]      (30, 'Python')
>>> t[-6::6]      (30,)
>>> t[-6::7]      (30,)

>>> t[-7::1]      ('py', 30, 40, 10, 70, 'Narayana', 'Python')
>>> t[-7::2]      ('py', 40, 70, 'Python')
>>> t[-7::3]      ('py', 10, 'Python')
>>> t[-7::4]      ('py', 70)
>>> t[-7::5]      ('py', 'Narayana')
>>> t[-7::6]      ('py', 'Python')
>>> t[-7::7]      ('py',)
>>> t[-7::8]      ('py',)
```

```
>>> t[-8::1]          (10, 'py', 30, 40, 10, 70, 'Narayana', 'Python')
>>> t[-8::2]          (10, 30, 10, 'Narayana')
>>> t[-8::3]          (10, 40, 'Narayana')
>>> t[-8::4]          (10, 10)
>>> t[-8::5]          (10, 70)
>>> t[-8::6]          (10, 'Narayana')
>>> t[-8::7]          (10, 'Python')
>>> t[-8::8]          (10,)
>>> t[-8::9]          (10,)

>>> t[-9::1]          (10, 'py', 30, 40, 10, 70, 'Narayana', 'Python')

>>> t[-1::-1]          ('Python', 'Narayana', 70, 10, 40, 30, 'py', 10)
>>> t[-1::-2]          ('Python', 70, 40, 'py')
>>> t[-1::-3]          ('Python', 10, 'py')
>>> t[-1::-4]          ('Python', 40)
>>> t[-1::-5]          ('Python', 30)
>>> t[-1::-6]          ('Python', 'py')
>>> t[-1::-7]          ('Python', 10)
>>> t[-1::-8]          ('Python',)

>>> t[-2::-1]          ('Narayana', 70, 10, 40, 30, 'py', 10)
>>> t[-2::-2]          ('Narayana', 10, 30, 10)
>>> t[-2::-3]          ('Narayana', 40, 10)
>>> t[-2::-4]          ('Narayana', 30)

>>> t[-2::-5]          ('Narayana', 'py')
>>> t[-2::-6]          ('Narayana', 10)
>>> t[-2::-7]          ('Narayana',)
>>> t[-2::-8]          ('Narayana',)

>>> t[-3::-1]          (70, 10, 40, 30, 'py', 10)
>>> t[-3::-2]          (70, 40, 'py')
>>> t[-3::-3]          (70, 30)
>>> t[-3::-4]          (70, 'py')
>>> t[-3::-5]          (70, 10)
>>> t[-3::-6]          (70,)
>>> t[-3::-7]          (70,)

>>> t[-4::-1]          (10, 40, 30, 'py', 10)
>>> t[-4::-2]          (10, 30, 10)
>>> t[-4::-3]          (10, 'py')
>>> t[-4::-4]          (10, 10)
>>> t[-4::-5]          (10,)
```

```
>>> t[-5::-1]          (40, 30, 'py', 10)
>>> t[-5::-2]          (40, 'py')
>>> t[-5::-3]          (40, 10)
>>> t[-5::-4]          (40,)
>>> t[-5::-5]          (40,)
>>> t[-6::-1]          (30, 'py', 10)
>>> t[-6::-2]          (30, 10)
>>> t[-6::-3]          (30,)
>>> t[-6::-4]          (30,)
>>> t[-7::-1]          ('py', 10)
>>> t[-7::-2]          ('py',)
>>> t[-7::-3]          ('py',)

>>> t[-8::-1]          (10,)
>>> t[-8::-2]          (10,)
```

Updating list elements

```
>>> lst=['a',10,20,30,'py',True]
>>> print(lst)          ['a', 10, 20, 30, 'py', True]
>>> type(lst)           <class 'list'>
>>> id(lst)             93150224

>>> lst[0]='Django'      #updating 'a' with 'Django'
>>> lst[1]="Python Narayana" #updating 10 with 'Python Narayana'

>>> print(lst)           ['Django', 'Python Narayana', 20, 30, 'py', True]

>>> lst[-1]=False        #updating True with False

>>> print(lst)           ['Django', 'Python Narayana', 20, 30, 'py', False]
```

Tuple concatenation:

We can concatenate two or more tuples in python.

Eg:

```
>>> tup1=(1,2,3,'a',True)      #creating first tuple tup1
>>> print(tup1)                (1, 2, 3, 'a', True)
>>> type(tup)                  <class 'tuple'>
```

```
>>> tup2=(10,20,False,'b')           #creating second tuple tup2
>> print(tup2)                   (10, 20, False, 'b')
>>> type(tup2)                  <class 'tuple'>

>>> tup3=tup1+tup2             #concatenating tup1 and tup2 as tup3
>> print(tup3)                 (1, 2, 3, 'a', True, 10, 20, False, 'b')
>>> type(tup3)                  <class 'tuple'>
```

Tuple multiplication or repetition:

We can multiply or repeat a tuple n number of times.

```
>>> tup1=(1,2,3,'a',True)
>> print(tup1)                  (1, 2, 3, 'a', True)
>>> type(tup1)                  <class 'tuple'>
>>> tup1*3                      (1, 2, 3, 'a', True, 1, 2, 3, 'a', True, 1, 2, 3, 'a', True)
```

We can not update the existing elements of a tuple with new elements,

```
>>> t=(10,20,30,40,50)
>> print(t)                     (10, 20, 30, 40, 50)
>>> type(t)                     <class 'tuple'>
>>> id(t)                       92980352

>>> t[0]=True                  TypeError: 'tuple' object does not support item assignment
>>> t[-1]=100                  TypeError: 'tuple' object does not support item assignment
```

Tuple Functions:

1. **All():** This function returns True if all elements are true, if any element is false (either 0 or False) then it will return False. For empty tuple also it will return True.

Eg:

```
>>> tup=(1,2,3)
>> print(all(tup))            True
```

```
>>> tup=(1,2,3,0)
>>> print(all(tup))          False
```

```
>>> tup=(2,3)
>>> print(all(tup))          True
```

```
>>> tup=()
>>> print(all(tup))          True
```

```
>>> tup=(True,)
>>> print(all(tup))          True
```

2. Any(): this function returns true if any one element is true in the tuple.

Eg:

```
>>> tup=(1,2,3)
>>> print(any(tup))          True
```

```
>>> tup=(1,2,3,0)
>>> print(any(tup))          True
```

```
>>> tup=(False,2,3)
>>> print(any(tup))          True
```

```
>>> tup=(False,0,0,False)
>>> print(any(tup))          False
```

```
>>> tup=()
>>> print(any(tup))          False
```

3. Len(): this function returns no.of elements in the tuple.

```
>>> tup=(1,2,3,4,'a',5.5)
>>> len(tup)                  6
```

4. Count(): This function counts the number of occurrences of a specific elements.

Eg:

```
>>> tup=(1,10,20,True,0)
>>> tup.count(1)           2
>>> tup.count(0)           1
```

5. Index(): This function is used to find the index value of specific element.

Eg:

```
>>> tup=(1,10,20,True,0)
>>> tup.index(0)           4
>>> tup.index(10)          1
>>> tup.index(20)          2
```

6. Max(): this function returns maximum value from the tuple elements.

Eg:

```
>>> tup=(1,3,2,55,3,5,23)
>>> max(tup)               55
```

Eg:

```
>>> t1=(2,8,False,9,True)
>>> max(t1)                 9
```

7. Min(): this function returns minimum value from the tuple elements.

Eg:

```
>>> tup=(1,3,2,55,3,5,23)
>>> min(tup)                1
```

```
>>> t1=(2,8,False,9,True)
>>> min(t1)                  False
```

8. Sorted(): this function sorts the data.

Eg:

```
>>> tup=(1,3,2,55,3,5,23)
>>> sorted(tup)             [1, 2, 3, 3, 5, 23, 55]
```

Note: by default this function sorts the data in ascending order. We can also get in descending order by setting **True** for **Reverse**.

Eg:

```
>>> tup=(1,3,2,55,3,5,23)
>>> sorted(tup,reverse=True)      [55, 23, 5, 3, 3, 2, 1]
```

Or

```
>>> t1=tuple([1,2,3,7,4])
>>> t1                      (1, 2, 3, 7, 4)
>>> t2=reversed(t1)
>>> tuple(t2)                (4, 7, 3, 2, 1)
```

9. Sum(): this function returns sum of all the elements.

Eg1:

```
>>> lst=[1,9,5,11,2]
>>> sum(lst)                  28
```

Eg2:

```
>>> t1=(2,8,False,9,True)
>>> min(t1)                   False
```

DEL Command

We cannot delete the elements of existing tuple but we can delete the entire tuple object by using **del** command.

Eg: >>> tup=(10,20,"Python",1.3)

```
>>> print(tup)              (10, 20, 'Python', 1.3)
>>> type(tup)                <class 'tuple'>
```

```
>>> del tup                 #deleting tuple by using del command.
```

```
>>> print(tup)          #after deleting  
NameError: name 'tup' is not defined
```

We can replace the elements of list but not tuple, like

```
>>> lst=[10,20,30,'Py',True]  
>>> lst[4]=False           #its possible in list  
>>> print(lst)  
[10, 20, 30, 'Py', False]
```

```
>>> tup=(10,20,30,'Py',True)  
>>> tup[4]=False          #it's not possible in tuple  
Traceback (most recent call last):  
  File "<pyshell#15>", line 1, in <module>  
    tup[4]=False  
TypeError: 'tuple' object does not support item assignment
```

How to display the given tuple in ascending order?

```
>>> tup=(10,20,5,3,30)  
  
>>> tup=sorted(tup)  
  
>>> print(tup)           [3, 5, 10, 20, 30]
```

How to display the given tuple in descending order?

```
>>> tup=(10,20,5,3,30)  
  
>>> tup=tuple(sorted(tup,reverse=True))  
  
>>> print(tup)           (30, 20, 10, 5, 3)
```

How to reverse the given tuple?

```
>>> tup=(10,20,5,3,30)  
  
>>> tup=reversed(tup)
```

```
>>> print(tuple(tup))      (30, 3, 5, 20, 10)
```

Nested tuple:

1. Python supports nested tuple, i.e a tuple in another tuple.
2. Tuple allows list as it's element.

Eg1:

```
>>> t1=(1,'a',True)
>>> print(t1)                  (1, 'a', True)
>>> type(t1)                 <class 'tuple'>

>>> t2=(10,'b',False)
>>> print(t2)                  (10, 'b', False)
>>> type(t2)                 <class 'tuple'>

>>> t3=(t1,100,'Python',t2)    #creating a tuple with existing tuples t1 and t2

>>> print(t3)                  ((1, 'a', True), 100, 'Python', (10, 'b', False))
>>> type(t3)                 <class 'tuple'>

>>> print(t3[0])                (1, 'a', True)
>>> print(t3[1])                100
>>> print(t3[2])                Python
>>> print(t3[3])                (10, 'b', False)
>>> print(t3[3][0])              10
>>> print(t3[3][1])              b
>>> print(t3[3][2])              False
>>> print(t3[0][0])              1
>>> print(t3[0][1])              a
>>> print(t3[0][2])              True
>>> t3[0:2]                     ((1, 'a', True), 100)
>>> t3[2:4]                     ('Python', (10, 'b', False))
```

```
>>> t3[-2:4] ('Python', (10, 'b', False))
```

Note: we can't modify any element of the above tuples because tuples are immutable.

If the tuple contains a list as a element then we can modify the elements of the list as it a mutable object.

Eg:2

```
>>> tup=(1,2,[10,12,'a'],(100,200,300),3,'Narayana')
```

```
>>> print(tup) (1, 2, [10, 12, 'a'], (100, 200, 300), 3, 'Narayana')
```

```
>>> type(tup) <class 'tuple'>
```

```
>>> tup[0] 1
```

```
>>> tup[1] 2
```

```
>>> tup[2] [10, 12, 'a']
```

```
>>> tup[3] (100, 200, 300)
```

```
>>> tup[4] 3
```

```
>>> tup[5] 'Narayana'
```

```
>>> tup[0]=50 #trying to replace element 1 with 50, interpreter throws error.
```

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

```
>>> tup[1]=50 #trying to replace element 2 with 50, interpreter throws error.
```

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

```
>>> tup[2]=50 #trying to replace element [10,12,'a'] with 50, interpreter throws error.
```

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

```
>>> tup[2][0]=50 #trying to replace element of list 100 with 50, interpreter accepts.
```

```
>>> print(tup) (1, 2, [50, 12, 'a'], (100, 200, 300), 3, 'Narayana')
```

Conversions:

Converting tuple to list

```
>>> tup=(1,2,4,9,8) #creating a tuple
```

```
>>> print(tup) (1, 2, 4, 9, 8)
```

```
>>> type(tup) <class 'tuple'>
```

```
>>> lst=list(tup)      #converting tuple to list by using list()
```

```
>>> print(lst)        [1, 2, 4, 9, 8]  
>>> type(lst)         <class 'list'>
```

Converting list to tuple

```
>>> lst=[10,20,30,40,'a']    #creating a list  
>>> print(lst)            [10, 20, 30, 40, 'a']  
>>> type(lst)             <class 'list'>
```

```
>>> tup=tuple(lst)        #converting list to tuple by using tuple()  
  
>>> print(tup)           (10, 20, 30, 40, 'a')  
>>> type(tup)            <class 'tuple'>
```

Converting tuple to string

```
>>> tup=('a','b','c')      #creating tuple  
>>> print(tup)           ('a', 'b', 'c')  
>>> type(tup)            <class 'tuple'>
```

```
>>> str1=".join(tup)"     #converting tuple to string by using join method  
  
>>> print(str1)          abc  
>>> type(str1)           <class 'str'>
```

Converting string to tuple

```
>>> str1="Python Narayana" #creating a string  
>>> print(str1)          Python Narayana  
>>> type(str1)           <class 'str'>
```

```
>>> tup=tuple(str1)       #converting a tuple to string by using tuple function.
```

```
>>> print(tup)           ('P', 'y', 't', 'h', 'o', 'n', ' ', 'N', 'a', 'r', 'a', 'y', 'a', 'n', 'a')  
>>> type(tup)            <class 'tuple'>
```

Tuple packing:

We can create a tuple by using existing variables, so its called tuple packing.

```
>>> a=10  
>>> b=20  
>>> c='Python'  
>>> d=2+5j  
  
>>> tup=(a,b,c,d)  
  
>>> print(tup)          (10, 20, 'Python', (2+5j))  
>>> type(tup)          <class 'tuple'>  
>>> id(tup)            62673808
```

Tuple Unpacking:

1. Tuple unpacking allows to extract tuple elements automatically.
2. Tuple unpacking is the list of variables on the left has the same number of elements as the length of the tuple

```
>>> tup=(1,2,3,4)  
  
>>> a,b,c,d=tup      # tuple unpacking  
  
>>> print(a)          1  
>>> print(b)          2  
>>> print(c)          3  
>>> print(d)          4
```

Advantages of Tuple over List

- Generally we use tuple for heterogeneous elements and list for homogeneous elements.
- Iterating through tuple is faster than with list because tuples are immutable, So there might be a slight performance boost.
- Tuples can be used as key for a dictionary. With list, this is not possible because list is a mutable object.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Difference between list and tuple

1.
If we need to add or remove the elements to object in the future then we choose list.
If we don't want add or remove the elements to the object in the future then we choose tuple.
2.
List is represented by []
Tuple is represented by ()
3.
[] are compulsory for list
() are optional for tuple
4.
We can delete specific element by using Del command in the list.
We can't delete specific element by using Del in the tuple.
5.
We can clear all elements of a list by using clear ()
We can't clear all elements of a tuple by using clear ()
6.
When we create a tuple with one element, then we should use comma ',' after the element
Comma is not required in the list to create list with single element.
7.
List is dynamic object
Tuple is static object
8.
We can add or remove elements in the list by using append (), extend (), insert (), remove () and pop ()
We can't add or remove elements in the tuple by using functions.
9.
Range () is used to generate the list.
Range () is not used to generate the tuple.
10.
Split () result stores in list format
Database data stores in tuple format when we fetched data from database to python application.

Tuple comparison:

```
>>> tup1=(10,20,30)
>>> tup2=(11,12)
>>> print cmp(tup1,tup2)
```

-1

```
>>> tup1=(10,20,30)
>>> tup2=(1,2,3,4,5)
>>> print cmp(tup1,tup2)
```

1

```
>>> tup1=(10,20,30)
>>> tup2=(10,20,30,4,5)
>>> print cmp(tup1,tup2)
```

-1

```
>>> tup1=(10,20,30)
>>> tup2=(10,20,30)
>>> print cmp(tup1,tup2)
```

0

```
>>> tup1=(10,30,20)
>>> tup2=(10,20,30)
>>> print cmp(tup1,tup2)
```

1

```
>>> tup1=('a','b','x')
>>> tup2=('a','b','y')
>>> print cmp(tup1,tup2)
```

-1

```
>>> tup1=(True,True)
>>> tup2=(False,True)
>>> print cmp(tup1,tup2)
```

1

SET

Data Structure:

1. A set is unordered collection of unique elements.
2. Set is commonly used in membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.
3. Set will not allow duplicate values.
4. Insertion order is not preserved but elements can be sorted
5. The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set.
6. Sets do not support indexing, slicing,
7. Sets do not support concatenation and multiplication.
8. There are currently two built-in set types,
 - a. set,
 - b. frozenset.

Set:

The set type is mutable - the contents can be changed using methods like add() and remove(). Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set.

Frozenset:

The frozenset type is immutable. Its contents cannot be altered after it is created; it can be used as a dictionary key or as an element of another set.

We can create a set in different ways,

1. Creating an empty set using set() and add elements to that empty set.

Eg:

```
>>> se1=set()                                #creating an empty set with set()

>>> se1.add(10)                             #adding elements to empty set
>>> se1.add(20)                             #adding elements to empty set
>>> se1.add(30)                             #adding elements to empty set
>>> se1.add(10)                             #adding duplicate value to set
>>> print(se1)                            {10, 20, 30}
```

2. Creating a set with elements using set().

Eg:

```
>>> se2=set([1,2,4,'a',2+4j,True])      #creating set with set()
>>> print(se2)                           {1, 2, 4, (2+4j), 'a'}
>>> type(se2)                           <class 'set'>
```

3. Creating a set with curly braces.

Eg:

```
>>> se3={1,2,3,4,"Narayana",True}       #creating a set with curly braces
>>> print(se3)                           {1, 2, 3, 4, 'Narayana'}
>>> type(se3)                           <class 'set'>
```

Performing membership operations:

We use 'in' and 'not in' to perform membership operations.

'in' and 'not' are used to check the specific element is a part of the existing set or not

```
>>> se1={10,20,30,True,100,'Narayana','Python'}
>>> 'Narayana' in se1        True
>>> 'Django' in se1         False
>>> 'Oracle' in se1         False
>>> 1 not in se1            False
>>> 100 not in se1          False
>>> 20 in se1               True
```

Removing duplicate elements from other sequence

Eg1:

```
>>> lst=[10,20,10,40,50,10,20]
>>> lst=list(set(lst))
>>> print(lst)           [40, 10, 20, 50]
>>> type(lst)           <class 'list'>
```

Eg2:

```
>>> st='narayana'
>>> print(st)           narayana
>>> type(st)            <class 'str'>

>>> s=set(st)
>>> print(s)            {'n', 'y', 'a', 'r'}

>>> st=".join(s)
>>> print(st)           nyar
```

Eg3:

```
>>> t=(10,20,30,20,10,20,30,40)
>>> print(t)           (10, 20, 30, 20, 10, 20, 30, 40)
>>> type(t)            <class 'tuple'>

>>> t=tuple(set(t))
>>> print(t)           (40, 10, 20, 30)
>>> type(t)            <class 'tuple'>
```

We can't update the set existing elements with new elements because will not support indexing,

```
>>> se={10,20,40,30,5}

>>> print(se)           {5, 40, 10, 20, 30}
>>> type(se)            <class 'set'>
>>> id(se)              93189584
>>> se[0]=100           TypeError: 'set' object does not support item assignment
```

Set functions

Add(): this function adds new elements to existing set.

Eg:

```
>>> se1={1,2,3,4,5}
```

```
>>> print(se1) {1, 2, 3, 4, 5}
```

```
>>> se1.add(6) #adding elements  
>>> se1.add(7)
```

```
>>> print(se1) {1, 2, 3, 4, 5, 6, 7}
```

Note: we can not add new elements to the frozenset.

```
>>> fs=frozenset([10,20,30,40])  
>>> print(fs) {10,20,30,40}
```

```
>>> fs.add(50) #trying to add new element to frozenset.
```

Error: AttributeError: 'frozenset' object has no attribute 'add'

Remove(): it will remove elements from the set, if that element is not found then it will throw error.

Eg:

```
>>> se1={1,2,3,4,5}  
>>> print(se1) {1, 2, 3, 4, 5}  
>>> type(se1) <class 'set'>
```

```
>>> se1.remove(5) #removing element from set
```

```
>>> se1.remove(4) #removing element from set
```

```
>>> se1.remove(15) #trying to remove element which is not there in set
```

Error: KeyError: 15

Discard(): it will remove elements from the set, if that element is not fund in the set then it will do nothing.

Eg:

```
>>> se1={1,2,3,4,5}  
>>> print(se1) {1, 2, 3, 4, 5}
```

```
>>> se1.discard(7)          #trying to remove element which not there in the set.  
>>> se1.discard(20)        #trying to remove element which not there in the set.  
>>> se1.discard(5)         #removing element which is there in the set  
>>> print(se1)             {1, 2, 3, 4}
```

the difference between remove() and discard() is,

remove(): if we take the element which is not there in the set then it will throw error.

discard(): if we take the element which is not there in the set then it will do nothing, means it will not throw error.

Pop(): This function removes set elements randomly

```
>>> s={10,20,30,40,50}  
>>> s.pop()                40  
>>> s.pop()                10  
>>> s.pop()                50  
>>> s.pop(20)              TypeError: pop() takes no arguments (1 given)  
>>> s.pop()                20  
  
>>> print(s)               {30}
```

Copy(): this function copies the elements of one set to another new set.

Eg:

```
>>> se1={1,2,3,4,5}  
>>> se2=se1.copy()         #copying se1 elements to se2  
>>> se1                  {1, 2, 3, 4, 5}  
>>> se2                  {1, 2, 3, 4, 5}
```

Clear(): this function clears the existing function.

```
>>> se1={1,2,3,4,5}  
>>> print(se1)            {1, 2, 3, 4, 5}  
>>> type(se1)            <class 'set'>  
  
>>> se1.clear()           #clearing the se1, so se1 will become empty set.  
>>> print(se1)            set()
```

Del command

```
>>> se={10,20,30,40,50,60}

>>> print(se)      {40, 10, 50, 20, 60, 30}
>>> type(se)       <class 'set'>
>>> id(se)         88536048

>>> del se[0]      TypeError: 'set' object doesn't support item deletion

>>> del se

>>> print(se)      NameError: name 'se' is not defined
```

Isdisjoint(): this function returns True if both are empty sets or if both sets contains non-matching elements.

Eg:

```
>>> se1=set()
>>> se2=set()
>>> se1.isdisjoint(se2)      True
```

```
>>> se1=set()
>>> se2={1,2,3}
>>> se1.isdisjoint(se2)      True
```

```
>>> se1={1,2,3}
>>> se2={1,2,3,4}
>>> se1.isdisjoint(se2)      False
```

Issubset():

x.issubset(y) returns True, if x is a subset of y. "<=" is an abbreviation for "Subset of".

```
>>> se1={1,2,3,4,5}
>>> se2={1,2,3}

>>> se2.issubset(se1)      True
>>> se1.issubset(se2)      False
```

Or

```
>>> se2<=se1          True  
>>> se1<=se2          False
```

Issuperset():

x.issuperset(y) returns True, if x is a superset of y. ">=" is an abbreviation for "issuperset of"

Eg:

```
>>> se1={1,2,3,4,5}  
>>> se2={1,2,3}
```

```
>>> se2.issuperset(se1)    False  
>>> se1.issuperset(se2)    True
```

```
>>> se2>=se1            False  
>>> se1>=se2            True
```

We can also **check the elements** whether they belong to set or not,

Eg:

```
>> se1={1,2,3,"Python",3+5j,8}  
  
>>> 4 in se1            False  
>>> 1 in se1            True  
>>> "Python" in se1     True  
>>> 10 not in se1       True  
>>> "Narayana" not in se1 True
```

Union: it returns the union of two sets, that means it returns all the values from both sets except duplicate values.

The same result we can get by using ' | ' between two sets

Syn: <First_Set>.union(<Second_Set>) or
 <First_Set> | <Second_Set>

Eg 1:

```
>>> se1={1,2,3,4,5}
>>> se2={1,2,3,6,7}

>>> se1.union(se2)           {1, 2, 3, 4, 5, 6, 7}      or
>>> se2.union(se1)           {1, 2, 3, 4, 5, 6, 7}
```

Or

```
>>> se1|se2                {1, 2, 3, 4, 5, 6, 7}
>>> se2|se1                {1, 2, 3, 4, 5, 6, 7}
```

Eg2:

```
>>> se1={1,4,2,'a','Python',False}
>>> se2={False,1,'a',10,20,'Narayana'}

>>> se1.union(se2)    {False, 1, 2, 'Narayana', 4, 'Python', 10, 20, 'a'}
>>> se2.union(se1)    {False, 1, 2, 'Narayana', 4, 'Python', 10, 20, 'a'}

>>> se1|se2            {False, 1, 2, 'Narayana', 4, 'Python', 10, 20, 'a'}
>>> se2|se1            {False, 1, 2, 'Narayana', 4, 'Python', 10, 20, 'a'}
```

Intersection: it returns an intersection elements of two sets, that means it returns only common elements from both sets.

That same operation we can get by sing '&' operator.

Syn: <First_Set>.intersection(<Second_Set>) or
<First_Set> & <Second_Set>

Eg1:

```
>>> se1={1,2,3,4,5}
>>> se2={1,2,3,6,7}

>>> se1.intersection(se2)    {1, 2, 3}      or
>>> se1&se2                {1, 2, 3}

Or

>>> se2.intersection(se1)    {1, 2, 3}
>>> se2&se1                {1, 2, 3}
```

Eg2:

```
>>> se1={False, 1, 2, 4, 'Python', 'a'}  
>>> se2={False, 1, 'Narayana', 10, 20, 'a'}  
  
>>> se1.intersection(se2) {False, 1, 'a'}  
>>> se2.intersection(se1) {False, 1, 'a'}
```

Or

```
>>> se1&se2 {False, 1, 'a'}  
>>> se2&se1 {False, 1, 'a'}
```

Difference: It returns all elements from first set which are not there in the second set.

Syn: <First_Set>.difference(<Secnd_Set>) or
<First_Set> - <Second_Set>

Eg1:

```
>>> se1={1,2,3,4,5}  
>>> se2={1,2,3,6,7}  
  
>>> se1.difference(se2) {4, 5} or  
>>> se1-se2 {4, 5}
```

Or

```
>>> se2.difference(se1) {6, 7} or  
>>> se2-se1 {6, 7}
```

Eg2:

```
>>> se1={False, 1, 2, 4, 'Python', 'a'}  
>>> se2={False, 1, 'Narayana', 10, 20, 'a'}  
  
>>> se1.difference(se2) {2, 4, 'Python'}  
>>> se2.difference(se1) {10, 'Narayana', 20}
```

Or

```
>>> se1-se2 {2, 4, 'Python'}
>>> se2-se1 {10, 'Narayana', 20}
```

Intersection_update: this function will update the First_Set with the result of intersection between First_Set and Second_Set.

Syn: <First_Set>.intersection_update(<Second_Set>)

Eg1:

```
>>> se1={1,2,3,4,5}
>>> se2={1,2,3,6,7}

>>> se1.intersection_update(se2)

>>> print(se1) {1, 2, 3}
>>> print(se2) {1, 2, 3, 6, 7}
```

Or

```
>>> se1={1,2,3,4,5}
>>> se2={1,2,3,6,7}

>>> se2.intersection_update(se1)
>>> print(se1) {1, 2, 3, 4, 5}
>>> print(se2) {1, 2, 3}
```

Eg2:

```
>>> se1={False, 1, 2, 4, 'Python', 'a'}
>>> se2={False, 1, 'Narayana', 10, 20, 'a'}

>>> se1.intersection_update(se2)
```

```
>>> print(se1)      {False, 1, 'a'}
>>> print(se2)      {False, 1, 'Narayana', 10, 20, 'a'}

or

>>> se1={1,4,2,'a','Python',False}
>>> se2={False,1,'a',10,20,'Narayana'}

>>> se2.intersection_update(se1)

>>> print(se2)      {False, 1, 'a'}
>>> print(se1)      {False, 1, 2, 4, 'Python', 'a'}
```

Difference_update: the result of difference between two sets will in First_Set.

Syn: <First_Set>.difference_update(<Second_Set>)

Eg:

```
>>> se1={1,2,3,4,5}
>>> se2={1,2,3,6,7}

>>> se1.difference_update(se2)

>>> print(se1)          {4, 5}
>>> print(se2)          {1, 2, 3, 6, 7}
```

Or

```
>>> se1={1,2,3,4,5}
>>> se2={1,2,3,6,7}

>>> se2.difference_update(se1)

>>> print(se1)          {1, 2, 3, 4, 5}
>>> print(se2)          {6, 7}
```

Eg2:

```
>>> se1={1,4,2,'a','Python',False}  
>>> se2={False,1,'a',10,20,'Narayana'}  
  
>>> se1.difference_update(se2)  
  
>>> print(se1)      {2, 4, 'Python'}  
>>> print(se2)      {False, 1, 'Narayana', 10, 20, 'a'}
```

or

```
>>> se1={1,4,2,'a','Python',False}  
>>> se2={False,1,'a',10,20,'Narayana'}  
  
>>> se2.difference_update(se1)  
  
>>> print(se2)      {'Narayana', 10, 20}  
>>> print(se1)      {False, 1, 2, 4, 'Python', 'a'}
```

Symmetric_difference: It returns unmatching elements from both sets

Syn: <First_Set>.symmetric_difference(<Second_Set>)

Eg:

```
>>> se1={1,2,3,4,5}  
>>> se2={1,2,3,6,7}  
  
>>> se1.symmetric_difference(se2)      {4, 5, 6, 7}  
>>> se2.symmetric_difference(se1)      {4, 5, 6, 7}
```

symmetric_difference_update: it will store the unmatching elements from both sets into First_Set.

Syn: <First_Set>.symmetric_difference_update(<Second_Set>)

Eg1:

```
>>> se1={1,2,3,4,5}  
>>> se2={1,2,3,6,7}  
  
>>> se1.symmetric_difference_update(se2)  
  
>>> print(se1) {4, 5, 6, 7}  
>>> print(se2) {1, 2, 3, 6, 7}
```

Or

```
>>> se1={1,2,3,4,5}  
>>> se2={1,2,3,6,7}  
  
>>> se2.symmetric_difference_update(se1)  
  
>>> print(se1) {1, 2, 3, 4, 5}  
>>> print(se2) {4, 5, 6, 7}
```

Eg2:

```
>>> se1={1,4,2,'a','Python',False}  
>>> se2={False,1,'a',10,20,'Narayana'}  
  
>>> se1.symmetric_difference_update(se2)  
  
>>> print(se1) {2, 'Narayana', 4, 'Python', 10, 20}  
>>> print(se2) {False, 1, 'Narayana', 10, 20, 'a'}
```

```
>>> se1={1,4,2,'a','Python',False}  
>>> se2={False,1,'a',10,20,'Narayana'}  
  
>>> se2.symmetric_difference_update(se1)  
  
>>> print(se1) {False, 1, 2, 4, 'Python', 'a'}  
>>> print(se2) {2, 'Narayana', 4, 'Python', 10, 20}
```

Set Packing:

The process of packing multiple variables a single place surrounded by curly braces is called set packing.

Eg:

```
a=True  
b="Python"  
c="Django"  
d=1  
e=20
```

```
se1={a,b,c,d,e}  
print(se1)          {'Python', True, 20, 'Django'}      #here 1 is repeating, so removed
```

```
se2={d,b,c,a,e}  
print(se2)          {'Python', 1, 20, 'Django'}      #here True is repeating, so removed
```

Set unpacking

The process of retrieving all elements of a set into different variables dynamically is called set unpacking.

The number of variables and the number of set elements must be same

Eg:

```
Se2= {1, 'Python', 20, 'Django'}
```

```
m,n,x,y=se2
```

```
print(m)  
print(type(m))
```

```
print(n)  
print(type(n))
```

```
print(x)  
print(type(x))
```

```
print(y)
```

```
print(type(y))
```

output:

```
1
```

```
<class 'int'>
```

```
Python
```

```
<class 'str'>
```

```
20
```

```
<class 'int'>
```

```
Django
```

```
<class 'str'>
```

Using set in the List:

```
>>> lst=[10,20,30,{True,2,False,3},[1,2,{100,200,300}]]
```

```
>>> print(lst) [10, 20, 30, {False, True, 2, 3}, [1, 2, {200, 100, 300}]]
```

```
>>> len(lst)
```

```
5
```

```
>>> lst[0]
```

```
10
```

```
>>> lst[1]
```

```
20
```

```
>>> lst[2]
```

```
30
```

```
>>> lst[3]
```

```
{False, True, 2, 3}
```

```
>>> len(lst[3]) 4
```

```
>>> lst.index({False,True,2,3}) 3
```

```
>>> lst[3][0] TypeError: 'set' object does not support indexing
```

```
>>> lst[4] [1, 2, {200, 100, 300}]
```

```
>>> len(lst[4]) 3
```

```
>>> lst[4][0] 1
```

```
>>> lst[4][1] 2
```

```
>>> lst[4][2] {200, 100, 300}
```

```
>>> lst[4].index(1) 0
```

```
>>> lst[4].index(2)           1
>>> lst[4].index({200,100,300}) 2
>>> lst[4].index({200,300,100}) 2
>>> lst[4].index({300,100,200}) 2

>>> lst[0:2]                  [10, 20]
>>> lst[0:3]                  [10, 20, 30]
>>> lst[0:4]                  [10, 20, 30, {False, True, 2, 3}]
>>> lst[0:5]                  [10, 20, 30, {False, True, 2, 3}, [1, 2, {200, 100, 300}]]
```

Eg1:

```
>>> lst[0:lst[5][2]]      IndexError: list index out of range
```

Note: the given list has index number upto 4 only, but we are trying to access 5, which is not available.

Eg2:

```
>>> lst[0:lst[4][1]]      [10, 20]
```

Here, the result of lst[4][1] is 2 which is in sublist because lst[4] is sublist in the above example.

So, lst[4][1] is replaced with 2 in the list slicing, like below

So, It is like lst[0:2], the result of lst[0:2] is [10,20]

Eg3:

```
>>> lst[0:lst[4][0]]      [10]
```

It works like above example..

Eg4:

```
>>> lst[lst[0]:lst[4][0]]    []
```

Here, lst[0] means 10.

lst[4][0] means 1.

so, lst[10:1] here the starting value is out of index range so it results in empty list [].

Eg5:

```
>>> lst[lst[0]:lst[4]]  TypeError: slice indices must be integers or None or have an __index__ method
```

Note: Both starting and ending values must be integers, but in this example, lst[4] is a list. so

Interpreter returned TypeError

Eg6:

```
>>> lst[lst[0]:lst[3]]      TypeError: slice indices must be integers or None or have an __index__ method
```

In this example, lst[3] is set, so it returned TypeError.

Eg7:

```
>>> lst[lst[0]:lst[2]]      []
```

Here lst[0] returns 10 and

lst[2] returns 30

so It is like lst[10:30], So here both are out of range values, so interpreter returned [].

Eg8:

```
>>> lst[lst.index(10):lst[2]]      [10, 20, 30, {False, True, 2, 3}, [1, 2, {200, 100, 300}]]
```

Here, lst.index(10) returns 0

lst[2] returns 30

So It is like lst[0:30], So the starting index value is 0 and ending index value is 30, thats why it has returned all from 0 to end of the list. Because ending index value 30 is out of index range.

Eg9:

```
>>> lst[lst.index(10):lst[4][2]]      TypeError: slice indices must be integers or None or have an __index__ method
```

Here, lst.index(10) returns 0, which is int type

lst[4][2] returns {200,100,300} which is a part sublist and its type set...

here the starting index number is int type but ending index number is not a int type, so interpreter returned TypeError.

Eg10:

```
>>> lst[lst.index(10):lst[4].index(2)]      [10]
```

Here, lst.index(10) returns 0

lst[4].index(2) returns 1 which is a part of sublist, the index number of 2 is 1 in the sublist.

so, It is like `lst[0:1]`, the result of this is [10]

Eg11:

```
>>> lst[lst.index(10):lst[4].index({100,200,300})] [10, 20]
```

Here, `lst.index(10)` returns 0

`lst[4].index({100,200,300})` returns 2 which is a part of sublist, the index number of {100,200,300} is

2 in the sublist.

so, It is like `lst[0:2]`, the result of this is [10,20]

Using set in the tuple:

eg1:

```
>>> t=(11,True,{2,3,1},12,13,(10,20,[100,200,300,{25,35,45}]))
```

```
>>> print(t) (11, True, {1, 2, 3}, 12, 13, (10, 20, [100, 200, 300, {25, 35, 45}]))
```

```
>>> len(t) 6
```

```
>>> t[0] 11
```

```
>>> t[1] True
```

```
>>> t[2] {1, 2, 3}
```

```
>>> t[2][0] TypeError: 'set' object does not support indexing
```

```
>>> t[2][1] TypeError: 'set' object does not support indexing
```

```
>>> t[2][2] TypeError: 'set' object does not support indexing
```

```
>>> t[3] 12
```

```
>>> t[4] 13
```

```
>>> t[5] (10, 20, [100, 200, 300, {25, 35, 45}])
```

```
>>> t[5][0] 10
```

```
>>> t[5][1] 20
```

```
>>> t[5][2] [100, 200, 300, {25, 35, 45}]
```

```
>>> t[5][2][0] 100
```

```
>>> t[5][2][1] 200
```

```
>>> t[5][2][2] 300
```

```
>>> t[5][2][3] {25, 35, 45}
```

```
>>> t[5][2][3][0] TypeError: 'set' object does not support indexing
```

```
>>> t[5][2][3][1] TypeError: 'set' object does not support indexing
```

```
>>> t[5][2][3][2]      TypeError: 'set' object does not support indexing

>>> t[0:2]              (11, True)
>>> t[0:3]              (11, True, {1, 2, 3})
>>> t[0:4]              (11, True, {1, 2, 3}, 12)
>>> t[0:5]              (11, True, {1, 2, 3}, 12, 13)
>>> t[0:6]              (11, True, {1, 2, 3}, 12, 13, (10, 20, [100, 200, 300, {25, 35, 45}]))
>>> t[1:5]              (True, {1, 2, 3}, 12, 13)
>>> t[1:6]              (True, {1, 2, 3}, 12, 13, (10, 20, [100, 200, 300, {25, 35, 45}]))
>>> t.index(11)          0
>>> t.index(True)         1
>>> t.index({1,3,2})      2
>>> t.index(12)           3
>>> t.index(13)           4
>>> t.index((10, 20, [100, 200, 300, {25, 35, 45}]))      5
>>> t[5].index(10)         0
>>> t[5].index(20)         1
>>> t[5].index([100, 200, 300, {25, 35, 45}])            2
>>> t[5][2].index(100)      0
>>> t[5][2].index(200)      1
>>> t[5][2].index(300)      2
>>> t[5][2].index({25,35,45})    3
>>> len(t[5])             3
>>> len(t[5][2])           4
>>> len(t[5][2][3])        3
>>> t[1:t[5][0]]          (True, {1, 2, 3}, 12, 13, (10, 20, [100, 200, 300, {25, 35, 45}]))
```

eg2:

```
>>> t1=(11, True, {1, 2, 3}, 12, 13, (1, 2, [100, 200, 300, {25, 35, 45}]))

>>> t1[1:t1[5][0]]          ()
>>> t1[0:t1[5][1]]          (11, True)
>>> t1[1:t1[5]]             TypeError: slice indices must be integers or None or have an __index__ method
>>> t1[1:t1[5][2]]             TypeError: slice indices must be integers or None or have an __index__ method
>>> t1[1:t1[5][1]]             (True,)
```

```
>>> t1[1:t1[5][3]]           IndexError: tuple index out of range  
>>> t1[1:t1[5][2][0]]       (True, {1, 2, 3}, 12, 13, (1, 2, [100, 200, 300, {25, 35, 45}]))  
>>> t1[1:t1[5][2][1]]       (True, {1, 2, 3}, 12, 13, (1, 2, [100, 200, 300, {25, 35, 45}]))  
>>> t1[1:t1[5][2][2]]       (True, {1, 2, 3}, 12, 13, (1, 2, [100, 200, 300, {25, 35, 45}]))  
>>> t1[1:t1[5][2][3]]       TypeError: slice indices must be integers or None or have an __index__  
method
```

Sets and frozen sets support the following operators:

```
key in s      # containment check  
key not in s # non-containment check  
se1 == se2   # s1 is equivalent to s2  
se1 != se2   # s1 is not equivalent to s2  
se1 <= se2   # s1 is subset of s2  
se1 < se2    # s1 is proper subset of s2  
se1 >= se2   # s1 is superset of s2  
se1 > se2    # s1 is proper superset of s2  
se1 | se2    # the union of s1 and s2  
se1 & se2    # the intersection of s1 and s2  
se1 - se2    # the set of elements in s1 but not s2  
se1 ^ se2    # the set of elements in precisely one of s1 or s2
```

DICTIONARY

Data Structure

Introduction:

1. Dictionary is an unordered set of *key: value* pairs, here keys are unique.
2. A pair of braces creates an empty dictionary: {}.
3. The main operations on a dictionary are storing a value with some key and extracting the value given the key.
4. Dictionary keys are not allowed duplicates but dictionary values are allowed duplicates.
5. We can use homogeneous and heterogeneous elements for both keys and values.
6. Insertion order is not preserved.
7. Dictionary keys are immutable and values are mutable.
8. Dictionary will not allow indexing and slicing.
9. Dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like append() and extend().

We can create dictionary in different ways,

1. Creating empty dictionary and adding key:value pairs.

Eg:

```
>>> dic1={}
>>> print(dic1)
                {}
>>> type(dic1)
                <class 'dict'>

>>> dic1['a']=10          #adding Key:Value pair to dictionary
>>> dic1['b']=20          #adding Key:Value pair to dictionary
>>> dic1['c']=30          #adding Key:Value pair to dictionary
```

```
>>> dic1['a']=10          #trying to add same Key:value pair  
>>> dic1['a']=50          #adding new value for same key 'a'  
  
>>> print(dic1)          {'a': 50, 'b': 20, 'c': 30}
```

2. Creating a dictionary with dict() function

Eg:

```
>>> dic1=dict([('a',10),('b',20),('c',30),('d',40)])  
>>> print(dic1)          {'a': 10, 'b': 20, 'c': 30, 'd': 40}  
>>> type(dic1)           <class 'dict'>
```

3. Creating a dictionary with curly braces including key:value pairs

Eg:

```
>>> dic1={'a':10,'b':20,'c':30,'d':40}  
>>> print(dic1)          {'a': 10, 'b': 20, 'c': 30, 'd': 40}  
>>> type(dic1)           <class 'dict'>
```

Accessing data from dictionary

We can assign values to the keys and later we can fetch values by using Keys

Eg1:

```
>>> stuDetails={'Id':100,'Name':'Sai','Age':20,'Marks':90}  
  
>>> print(stuDetails)          {'Id': 100, 'Name': 'Sai', 'Age': 20, 'Marks': 90}  
>>> type(stuDetails)          <class 'dict'>  
  
>>> print(stuDetails['Id'])    100  
>>> print(stuDetails['Age'])   20
```

Eg2:

```
>>> stuDetails={'Id':100,'Name':'Sai', 'subjects':['SQL Server', 'Oracle', 'Python']}
```

```
{"Id":100,'Name':'Sai', 'subjects':['SQL Server', 'Oracle', 'Python']}
```

```
>>> stuDetails {'Id': 100, 'Name': 'Sai', 'subjects': ['SQL Server', 'Oracle', 'Python']}
```

```
>>> stuDetails['Id'] 100
```

```
>>> stuDetails['Name'] 'Sai'
```

```
>>> stuDetails['subjects'] ['SQL Server', 'Oracle', 'Python']
```

```
>>> stuDetails['subjects'][0] 'SQL Server'
```

```
>>> stuDetails['subjects'][1] 'Oracle'
```

```
>>> stuDetails['age'] KeyError: 'age' #this key not available
```

To prevent this type of error we can check whether the specified key existed or not by using `has_key()`.

But this `has_key()` is available in python2 only not in python3 version.

```
>>> stuDetails.has_key('age') False #in python2 version
```

In python3 we have to check by using membership operator '`in`'.

```
>>> 'age' in stuDetails False #in python3 version
```

Adding new key:value pairs

```
{"Id":100,'Name':'Sai', 'subjects':['SQL Server', 'Oracle', 'Python']}
```

If we need to **add** another element then

```
>>> stuDetails['Age']=25 #adding new element to the stuDetails Dictionary
```

```
>>> print(stuDetails) {"Id": 100, 'Name': 'Sai', 'subjects': ['SQL Server', 'Oracle', 'Python'], 'Age': 25}
```

Updating dictionary values

If we need to **change** the age from 25 to 27 then

```
>>> stuDetails['Age']=27          #changing the age from 25 to 27  
>>> print(stuDetails)  
{'Id': 100, 'Name': 'Sai', 'subjects': ['SQL Server', 'Oracle', 'Python'], 'Age': 27}
```

Deleting key:value pairs from dictionary

If we need to **delete** the value 27 from the above dictionary then (by using **pop**)

```
>>> stuDetails.pop('Age')      27  
>>> print(stuDetails)        {'Id': 100, 'Name': 'Sai', 'subjects': ['SQL Server', 'Oracle', 'Python']}
```

We can also **delete** by using **popitem()**

```
#this function removes last key:value pair in the dictionary,  
>>> stuDetails.popitem()     ('subjects', ['SQL Server', 'Oracle', 'Python'])
```

If we need to **delete** all key:value pairs then we can use **clear()**

```
>>> stuDetails.clear()       #deleting all pairs, then we can have an empty dict.  
>>> print(stuDetails)        {}
```

Dictionary functions

keys(): it returns all keys from dict.

Eg:

```
>>> dic1={1: 'Python', 2: (3+5j), 3: (10, 20, 30), 4: [100, 'a', False], 's': 100}  
>>> dic1.keys()            dict_keys([1, 2, 3, 4, 's'])
```

values(): it returns all values from the dict.

Eg:

```
>>> dic1={1: 'Python', 2: (3+5j), 3: (10, 20, 30), 4: [100, 'a', False], 's': 100}  
>>> dic1.values()          dict_values(['Python', (3+5j), (10, 20, 30), [100, 'a', False], 100])
```

copy(): it copies the dict into new dict.

Eg:

```
>>> dic1={1: 'Python', 2: (3+5j), 3: (10, 20, 30), 4: [100, 'a', False], 's': 100}
```

```
>>> dic2=dic1.copy()
```

```
>>> dic1 {1: 'Python', 2: (3+5j), 3: (10, 20, 30), 4: [100, 'a', False], 's': 100}
```

```
>>> dic2 {1: 'Python', 2: (3+5j), 3: (10, 20, 30), 4: [100, 'a', False], 's': 100}
```

pop(): it removes specific key value pair.

Eg:

```
>>> dic1={1: 'Python', 2: (3+5j), 3: (10, 20, 30), 4: [100, 'a', False], 's': 100}
```

```
>>> dic1.pop(1) 'Python'
```

```
>>> print(dic1) {2: (3+5j), 3: (10, 20, 30), 4: [100, 'a', False], 's': 100}
```

Note: we can also remove the key: value pair by using 'del' command.

Eg:

```
>>> del dic1[2]
```

```
>>> print(dic1) {3: (10, 20, 30), 4: [100, 'a', False], 's': 100}
```

clear(): it removes all key:value pairs from dict.

Eg 1:

```
>>> d={10: None, 20: None}
```

```
>>> d.clear()
```

```
>>> print(d) {}
```

Eg 2:

```
>>> dic1= {'id':101,'name':'Sai'}
```

```
>>> dic1.clear()
```

```
>>> dic1 {}
```

Del Command: This command is used to delete either a specific key:value pair or the whole dictionary.

Eg:

```
>>> d={100:'a',101:'b'}  
>>> del d[100]      #deleting specific key:value pair  
>>> d                {101: 'b'}  
>>> del d            #deleting entire dict object  
>>> d                NameError: name 'd' is not defined
```

fromkeys(): we can use tuple elements as keys in the dict and the elements must be unique

Eg:

```
>>> tup=(1,2,3,4,5)          #creating tuple  
>>> dic1.fromkeys(tup)      #taking tuple elements as keys in the dict dic1  
                           {1: None, 2: None, 3: None, 4: None, 5: None}
```

By default values are None, if we need to get 0 as default then,

```
>>> tup=(1,2,3,4,5)  
>>> dic={}.fromkeys(tup,0)  
>>> dic                {1: 0, 2: 0, 3: 0, 4: 0, 5: 0}
```

we can also use list elements as keys in the dict and the elements must be unique

```
>>> lst=[1,2,3,4,5]          #creating list  
  
>>> dic2.fromkeys(lst)      #taking list elements as keys in the dict dic2  
                           {1: None, 2: None, 3: None, 4: None, 5: None}
```

get(): this function is used to get the value of specified key.

Eg:

```
>>> stuDetails=  
{'Id': 100, 'Name': 'Sai', 'subjects': ['SQL Server', 'Oracle', 'Python'], 'id': 1000, 'name': 'nani'}  
  
>>> stuDetails.get('Id')      100  
>>> stuDetails.get('subjects') ['SQL Server', 'Oracle', 'Python']
```

Items(): this function is used to get all items. All key and value pairs will be displayed in tuple format.

Eg:

```
>>> stuDetails={  
    'Id': 100, 'Name': 'Sai', 'subjects': ['SQL Server', 'Oracle', 'Python'], 'id': 1000, 'name': 'nani'}
```

```
>>> stuDetails.items()
```

```
dict_items([('Id', 100), ('Name', 'Sai'), ('subjects', ['SQL Server', 'Oracle', 'Python']), ('id', 1000), ('name', 'nani')])
```

Update(): the current dictionary will be updated with the all key:value pairs from another dictionary.

Eg1:

```
>>> d={'id':1}  
>>> d1={'name':'Sai'}
```

```
>>> d.update(d1)      #updating d with d1
```

```
>>> print(d)          {'id': 1, 'name': 'Sai'}  
>>> print(d1)          {'name': 'Sai'}
```

Eg 2:

```
>>> stuDetails={'Id':100,'Name':'Sai', 'subjects':['SQL Server', 'Oracle', 'Python']}  
>>> stuDetails1={'id':1000,'name':'nani'}
```

```
>>> stuDetails.update(stuDetails1)
```

```
>>> print(stuDetails)  
{'Id': 100, 'Name': 'Sai', 'subjects': ['SQL Server', 'Oracle', 'Python'], 'id': 1000, 'name': 'nani'}
```

Creating a dictionary of dictionaries:

```
emp1={"id":1001,"name":"Python Narayana","loc":"Guntur"}  
emp2={"eid":1002,"ename":"Nagaraju","location":"Hyderabad"}  
  
emps={"emp1":emp1,"emp2":emp2}  
  
print(emps)  
  
print(emps["emp1"]["id"])  
print(emps["emp1"]["name"])  
print(emps["emp1"]["loc"])  
  
print(emps["emp2"]["eid"])  
print(emps["emp2"]["ename"])  
print(emps["emp2"]["location"])
```

Output:

```
{'emp1': {'id': 1001, 'name': 'Python Narayana', 'loc': 'Guntur'}, 'emp2': {'eid': 1002, 'ename': 'Nagaraju', 'location': 'Hyderabad'}}
```

```
1001  
Python Narayana  
Guntur
```

```
1002  
Nagaraju  
Hyderabad
```

List cant be used as keys in the dict

```
>>> d={[1,2,3]:"Python",4:"Django"}          TypeError: unhashable type: 'list'
```

Tuple can be used as keys in dict

```
>>> d={(1,2,3):"Python",4:"Django"}  
  
>>> print(d)          {(1, 2, 3): 'Python', 4: 'Django'}  
>>> d[(1,2,3)]      'Python'  
>>> d[4]             'Django'  
>>> d.keys()         dict_keys([(1, 2, 3), 4])  
>>> d.items()        dict_items([(1, 2, 3), 'Python'), (4, 'Django')])  
>>> d.get((1,2,3))  'Python'  
>>> d.get(4)         'Django'
```

Iterating over dictionary:

```
emp={"id":1001,"name":"Sai","loc":"Guntur","sal":20000,"comm":200,"company":"Infosys"}
```

```
print(emp)  
print(type(emp))  
print(id(emp))
```

```
for i in emp:  
    print(i)
```

output:

```
{'id': 1001, 'name': 'Sai', 'loc': 'Guntur', 'sal': 20000, 'comm': 200, 'company': 'Infosys'}
```

```
<class 'dict'>
```

```
88582544
```

```
id  
name  
loc  
sal  
comm  
company
```

Membership testing on dictionary:

```
emp={"id":1001,"name":"Sai","loc":"Guntur","sal":20000,"comm":200,"company":"Infosys"}  
  
print(emp)      {"id":1001,"name":"Sai","loc":"Guntur","sal":20000,"comm":200,"company":"Infosys"}  
print(type(emp))    <class 'dict'>  
print(id(emp))     94087664  
  
  
print("id" in emp)      True  
print("Name" in emp)    False  
print("name" in emp)    True  
print("fname" not in emp) True  
print("sal" in emp)     True  
print("company" in emp) True
```

How to perform arithmetic operations on the values of a dictionary?

```
>>> d1={'sub1':80,'sub2':90,'sub3':70,'sub4':80}  
>>> print(d1)                  {'sub1': 80, 'sub2': 90, 'sub3': 70, 'sub4': 80}  
  
>>> s=sum(d1.values())  
>>> print(s)                  320  
>>> mx=max(d1.values())  
>>> print(mx)                  90  
  
>>> mn=min(d1.values())  
>>> print(mn)                  70  
  
>>> cnt=len(d1.values())  
>>> print(cnt)                  4
```

Dictionary vs List:

Unlike lists, items in dictionaries are unordered. The first item in a list named lst would be lst[0]. But there is no “first” item in a dictionary.

While the order of items matters for determining whether two lists are the same, it does not matter in what order the key-value pairs are typed in a dictionary.

```
>>> lst1=["Python","Django","Narayana",True,10]  
  
>>> print(lst1) ['Python', 'Django', 'Narayana', True, 10]  
>>> type(lst1) <class 'list'>  
  
>>> lst2=[True,10,"Python","Narayana","Django"]  
>>> print(lst2) [True, 10, 'Python', 'Narayana', 'Django']  
>>> type(lst2) <class 'list'>  
  
>>> lst1==lst2 False
```

Note: lst1 and lst2 are containing same elements but in different places, so both lst1 and lst2 are not equal

```
>>> dict1={"Id":100,"Name":"Sai","Loc":"Hyd","Company":"TCS"}  
  
>>> print(dict1) {'Id': 100, 'Name': 'Sai', 'Loc': 'Hyd', 'Company': 'TCS'}  
>>> type(dict1) <class 'dict'>  
  
>>> dict2={"Name":"Sai","Company":"TCS","Id":100,"Loc":"Hyd"}  
>>> print(dict2) {'Name': 'Sai', 'Company': 'TCS', 'Id': 100, 'Loc': 'Hyd'}  
>>> type(dict2) <class 'dict'>  
  
>>> dict1==dict2 True
```

Note: dict1 and dict2 are containing same key-value pairs and also all are in different places, so both dict1 and dict2 are same. In dictionary the order is not fixed.

Because dictionaries are not ordered, they can't be sliced like lists. Trying to access a key that does not exist in a dictionary will result in a KeyError error message, much like a list's “out-of-range” IndexError error message.

```
>>> dict1={'Id': 100, 'Name': 'Sai', 'Loc': 'Hyd', 'Company': 'TCS'}
```

```
>>> dict1["Fname"]
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    dict1["Fname"]
KeyError: 'Fname'
```

Nested dictionary:

Eg1:

```
>>> emps={1001:{"name":"Nani","loc":"Hyd","sal":10000,"company":"Infosys"},  
        1002:{"name":"Renu","loc":"Bang","sal":20000,"company":"Infosys"},  
        1003:{"name":"Durga","loc":"Chennai","sal":30000,"company":"Infosys"}}  
  
>>> print(emps)  
  
{1001: {'name': 'Nani', 'loc': 'Hyd', 'sal': 10000, 'company': 'Infosys'},  
 1002: {'name': 'Renu', 'loc': 'Bang', 'sal': 20000, 'company': 'Infosys'},  
 1003: {'name': 'Durga', 'loc': 'Chennai', 'sal': 30000, 'company': 'Infosys'}}  
  
>>> type(emps)  
<class 'dict'>  
  
>>> id(emps)  
100638864  
  
>>> emps[1001]          {'name': 'Nani', 'loc': 'Hyd', 'sal': 10000, 'company': 'Infosys'}  
>>> emps[1002]          {'name': 'Renu', 'loc': 'Bang', 'sal': 20000, 'company': 'Infosys'}  
>>> emps[1003]          {'name': 'Durga', 'loc': 'Chennai', 'sal': 30000, 'company': 'Infosys'}  
  
  
>>> emps[1001]["name"]  'Nani'  
>>> emps[1001]["loc"]   'Hyd'  
>>> emps[1001]["sal"]  10000  
>>> emps[1001]["company"] 'Infosys'  
  
>>> emps[1002]["name"]  'Renu'  
>>> emps[1002]["loc"]   'Bang'  
>>> emps[1002]["sal"]  20000
```

```
>>> emps[1002]["company"]      'Infosys'

>>> emps[1003]["name"]        'Durga'
>>> emps[1003]["loc"]         'Chennai'
>>> emps[1003]["sal"]          30000
>>> emps[1003]["company"]     'Infosys'

>>> emps.keys()               dict_keys([1001, 1002, 1003])

>>> emps.values()
dict_values([{'name': 'Nani', 'loc': 'Hyd', 'sal': 10000, 'company': 'Infosys'},
             {'name': 'Renu', 'loc': 'Bang', 'sal': 20000, 'company': 'Infosys'},
             {'name': 'Durga', 'loc': 'Chennai', 'sal': 30000, 'company': 'Infosys'}])

>>> emps.get(1001)            {'name': 'Nani', 'loc': 'Hyd', 'sal': 10000, 'company': 'Infosys'}
>>> emps.get(1002)            {'name': 'Renu', 'loc': 'Bang', 'sal': 20000, 'company': 'Infosys'}
>>> emps.get(1003)            {'name': 'Durga', 'loc': 'Chennai', 'sal': 30000, 'company': 'Infosys'}

>>> emps.items()
dict_items([(1001, {'name': 'Nani', 'loc': 'Hyd', 'sal': 10000, 'company': 'Infosys'}),
            (1002, {'name': 'Renu', 'loc': 'Bang', 'sal': 20000, 'company': 'Infosys'}),
            (1003, {'name': 'Durga', 'loc': 'Chennai', 'sal': 30000, 'company': 'Infosys'})])

>>> del emps[1003]

>>> print(emps)
{1001: {'name': 'Nani', 'loc': 'Hyd', 'sal': 10000, 'company': 'Infosys'},
 1002: {'name': 'Renu', 'loc': 'Bang', 'sal': 20000, 'company': 'Infosys'}}

>>> emps.clear()

>>> print(emps)              {}
```

Eg2:

```
>>> emps={1001: {'name': 'Nani', 'loc': 'Hyd', 'sal': 10000},  
        1002: {'name': 'Renu', 'loc': 'Bang', 'sal': 20000},  
        1003: {'name': 'Durga', 'loc': 'Chennai', 'sal': 30000}}
```

```
>>> print(emps)  
{1001: {'name': 'Nani', 'loc': 'Hyd', 'sal': 10000},  
 1002: {'name': 'Renu', 'loc': 'Bang', 'sal': 20000},  
 1003: {'name': 'Durga', 'loc': 'Chennai', 'sal': 30000}}
```

```
>>> type(emps)  
<class 'dict'>
```

```
>>> id(emps)  
95857136
```

```
>>> emps[1001]["sal"]=25000
```

#retriving specific key value pair

```
>>> print(emps[1001]){'name': 'Nani', 'loc': 'Hyd', 'sal': 25000}
```

#adding new key-value pair to the existing dict

```
>>> emps[1004]={'name':'Python Narayana','loc':"Hyderabad","sal":10000}
```

#retriving all key-value pairs

```
>>> print(emps)  
{1001: {'name': 'Nani', 'loc': 'Hyd', 'sal': 25000},  
 1002: {'name': 'Renu', 'loc': 'Bang', 'sal': 20000},  
 1003: {'name': 'Durga', 'loc': 'Chennai', 'sal': 30000},  
 1004: {'name': 'Python Narayana', 'loc': 'Hyderabad', 'sal': 10000}}
```

#retriving last key-value pair

```
>>> print(emps[1004]){'name': 'Python Narayana', 'loc': 'Hyderabad', 'sal': 10000}
```

#updating sal of key 1004

```
>>> emps[1004]["sal"]=15000
```

#after updating the value of key 1004

```
>>> print(emps[1004]) {'name': 'Python Narayana', 'loc': 'Hyderabad', 'sal': 15000}
```

#removing the specific key-value pair of nested dictionary

```
>>> del emps[1004]["sal"]
```

#after removing the specific key-value from nested dict

```
>>> print(emps[1004]) {'name': 'Python Narayana', 'loc': 'Hyderabad'}
```

#removing the specific key-value pair of nested dictionary

```
>>> del emps[1003]["loc"]
```

#removing the specific key-value pair of nested dictionary

```
>>> del emps[1002]["loc"]
```

#the main dict after removing so many specific key-value pairs from nested dict

```
>>> print(emps)  
{1001: {'name': 'Nani', 'loc': 'Hyd', 'sal': 25000},  
 1002: {'name': 'Renu', 'sal': 20000},  
 1003: {'name': 'Durga', 'sal': 30000},  
 1004: {'name': 'Python Narayana', 'loc': 'Hyderabad'}}
```

##removing the specific key-value pair of main dictionary

```
>>> del emps[1003]
```

#removing the specific key-value pair of main dictionary

```
>>> del emps[1002]
```

#the final data in the amin dict

```
>>> print(emps)
```

```
{1001: {'name': 'Nani', 'loc': 'Hyd', 'sal': 25000}, 1004: {'name': 'Python Narayana', 'loc': 'Hyderabad'}}
```

Eg3:

```
emps = {1: {'name': 'Narayana', 'age': '27', 'sex': 'Male'},  
        2: {'name': 'Veni', 'age': '25', 'sex': 'Female'},  
        3: {'name': 'Sivani', 'age': '24', 'sex': 'Female'},  
        4: {'name': 'Nagaraju', 'age': '29', 'sex': 'Male'}}  
  
del emps[3], emps[4]      #deleting multiple key-value pairs from the dict  
print(emps)  
{1: {'name': 'Narayana', 'age': '27', 'sex': 'Male'}, 2: {'name': 'Veni', 'age': '25', 'sex': 'Female'}}
```

Iterating over dict object:

```
emps = {1001: {'name': 'Narayana', 'age': '27', 'sex': 'Male', 'loc': "Hyderabad"},  
        1002: {'name': 'Veni', 'age': '25', 'sex': 'Female', 'loc': "Hyderabad"},  
        1003: {'name': 'Sivani', 'age': '24', 'sex': 'Female', 'loc': "Bangalore"},  
        1004: {'name': 'Nagaraju', 'age': '29', 'sex': 'Male', 'loc': "Chennai"}}  
  
  
for empid,empinfo in emps.items():  
    print("Emp id :",empid)  
  
    for key in empinfo:  
        print(key + ':',empinfo[key])
```

Output:

```
Emp id : 1001  
name: Narayana  
age: 27  
sex: Male  
loc: Hyderabad
```

```
Emp id : 1002  
name: Veni  
age: 25  
sex: Female  
loc: Hyderabad
```

Emp id : 1003

name: Sivani

age: 24

sex: Female

loc: Bangalore

Emp id : 1004

name: Nagaraju

age: 29

sex: Male

loc: Chennai

All data types- mutable and immutable

Class	Description	Mutable?	Immutable?
Int	Integer value		✓
Str	Character string		✓
Float	Floating-point number		✓
Bool	Boolean value		✓
Complex	Complex value		✓
List	Sequence of objects	✓	
Set	Ordered set of unique objects	✓	
Frozenset	Immutable form set		✓
Tuple	Sequence of objects		✓
Dictionary	Key:value pairs	✓	

Eg:

```
# create a mapping of state to abbreviation
states = {'Oregon': 'OR', 'Florida': 'FL', 'California': 'CA', 'New York': 'NY', 'Michigan': 'MI' }

# create a basic set of states and some cities in them
cities = { 'CA': 'San Francisco', 'MI': 'Detroit', 'FL': 'Jacksonville' }

# add some more cities
cities['NY'] = 'New York'
```

```
cities['OR'] = 'Portland'

# print out some cities
print( '-' * 20)
print ("NY State has: ", cities['NY'])
print ("OR State has: ", cities['OR'])

# print some states
print ('-' * 20)
print ("Michigan's abbreviation is: ", states['Michigan'])
print ("Florida's abbreviation is: ", states['Florida'])

# do it by using the state then cities dict
print ('-' * 20)
print ("Michigan has: ", cities[states['Michigan']])
print ("Florida has: ", cities[states['Florida']])

# print every state abbreviation
print ('-' * 20)
for state, abbrev in states.items():
    print("%s is abbreviated %s" % (state, abbrev))

# print every city in state
print( '-' * 20)
for abbrev, city in cities.items():
    print ("%s has the city %s" % (abbrev, city))

# now do both at the same time
print ('-' * 20)
for state, abbrev in states.items():
    print ("%s state is abbreviated %s and has city %s" % ( state, abbrev, cities[abbrev]))

print ('-' * 20)
# safely get an abbreviation by state that might not be there
state = states.get('Texas', None)

if not state:
    print ("Sorry, no Texas.")
```

```
# get a city with a default value  
city = cities.get('TX', 'Does Not Exist')  
print ("The city for the state 'TX' is: %s" % city)
```

output:

```
-----  
NY State has: New York  
OR State has: Portland  
-----  
Michigan's abbreviation is: MI  
Florida's abbreviation is: FL  
-----  
Michigan has: Detroit  
Florida has: Jacksonville  
-----  
Oregon is abbreviated OR  
Florida is abbreviated FL  
California is abbreviated CA  
New York is abbreviated NY  
Michigan is abbreviated MI  
-----  
CA has the city San Francisco  
MI has the city Detroit  
FL has the city Jacksonville  
NY has the city New York  
OR has the city Portland  
-----  
Oregon state is abbreviated OR and has city Portland  
Florida state is abbreviated FL and has city Jacksonville  
California state is abbreviated CA and has city San Francisco  
New York state is abbreviated NY and has city New York  
Michigan state is abbreviated MI and has city Detroit  
-----  
Sorry, no Texas.  
The city for the state 'TX' is: Does Not Exist
```

Assignment - 2

1. What are the different types of data structures in python?
2. What is a string? What is the data type to represent string value?
3. What is a List? What is the data type of list? Create one list?
4. What is a Tuple? What is the data type? Create one tuple?
5. What is a set? What is the data type? Create one set?
6. What is dictionary? What is the data type of it? Create one dictionary?
7. What is mutable and immutable? Explain each?
8. Is string mutable or immutable? Why?
9. What are the starting index numbers for both forward and backward indexing?

10. What is indexing and slicing? Differences between both?

11. What are the IndexError and TypeError errors?

12. St = 'python developer'

a. Display as 'Python developer'

b. Display as 'Python Developer'

c. Count number of Es in that string.

d. Find index position of first 'o' in the above string.

e. Find index position of second 'o' in the above string.

f. Split the same string into two elements like 'python','developer'

g. Display string in reverse.

h. From the above string, display only 'on dev'.

i. From above string remove 'thon'

j. Copy the same string into other new strings like str1,str2 and str3 at a time

13. Generate a list [10,12,14,16,18,20] by using range function?

14. `lst=[10,20,'Python',10.5,1,10,True,False,0]` what are the results of,

- a. How to display the count of elements in the above list?

- b. How to count no of occurrences of '10' in the above list?

- c. What are the no of occurrences '1' in the above list?

- d. How to add complex number $10+5j$ to the above list?

- e. How to add both complex number $1+2j$ and float value 1.3 to the above list.

- f. How to add bool value 'True' between 10 and 20 in the above list?

- g. How to remove the str value 'Python' from the above list?

- h. How to reverse the above list?

- i. How to copy the above list `lst` to another list `lst1`?

- j. Display index position of float value 10.5 in the above list?

- k. Replace float value 10.5 with 20.5

15. What are the differences between append() and extend() in list?

16. What is the 'Del' command? Explain with one example?

17. How to convert a list to string? Give one example?

18. How to convert a string to list? Give one example?

19. How to convert tuple to list? Give one example?

20. How to convert list to tuple? Give one example?

21. How to convert tuple to string? Give one example?

22. How to convert string to tuple? Give one example?

23. What is tuple? Does tuple allow duplicate elements?

24. Can we add new element to the existing set? How can we add?

25. Can we add new elements to the frozenset? If no, give one example?

26. Let's take two sets

se1={1,2,3,4,5}

se2={1,2,3,6,7}

1. Perform union between se1 and se2, by using 'l' also
2. Perform intersection between se2 and se1, by using '&' also
3. Perform difference between se2 and se1, by using '-' also.
4. What intersection_update?and perform between se1 and se2 and vice versa.
5. What is difference_update?and perform between se1 and se2 and vice versa.
6. What is symmetric_difference? Perform between se1 and se2 and vice versa.
7. What is symmetric_difference_update? perform between se1 and se2 and vice versa.

27. Difference between remove() and discard()?

28. Is frozenset mutable or immutable?

29. What is the result of isdisjoint() if one set contains elements and other set is empty?

30. What is the difference between '=' and '=='? explain with one example?

31. How to check whether the specific element is existed or not in the given list? Example?

32. >>> se1={11,12,13,14,15}
>>> se2={11,12,13,16,17} what is the result of

a. se1==se2?

b. se1!=se2

c. se1<=se2

d. se1<se2

e. se2>se1

f. se1.isdisjoint(se2)

33. In what cases isdisjoint method will be True? Give examples?

34. >>> s1={1,2,3}

>>> s2={1,2,3,4} what is the result of

a. s1<s2

b. s1<=s2

c. s1>s2

- d. `s1>=s2`
- e. `s1.isdisjoint(s2)`
- f. `s1.issubset(s2)`
- g. `s1.issuperset(s2)`
- h. `s2.issubset(s1)`
- i. `s2.issuperset(s1)`
- j. `s1!=s2`
- k. 4 not in s2?
- l. 1 in s2?

35. `>>> s1{},
>>> s2{} what is the result of`

- a. `s1<s2`
- b. `s1<=s2`

- c. `s1>s2`
- d. `s1>=s2`
- e. `s1.isdisjoint(s2)`
- f. `s1.issubset(s2)`
- g. `s1.issuperset(s2)`
- h. `s2.issubset(s1)`
- i. `s2.issuperset(s1)`
- j. `s1!=s2`

36. `>>> s1={True,False,10,20,3.5}`
`>>> s2={1,0} what is the result of`

- a. `s2.issubset(s1)`
- b. `s1.issubset(s2)`

c. `s1.issuperset(s2)`

d. `s2.issuperset(s1)`

e. `s2>s1`

f. `s1<s2`

g. `s1>=s2`

h. `s2<=s1`

i. `s2=s1`

37. Can we use duplicate keys in dictionaries?

38. `>>> set1={10,20,30,40,50}`

`>>> set2={10,20,30,60,70}`

a. Display all distinct elements from both sets

b. Display only nonmatching elements from s1 set.

- c. Display only nonmatching elements from both sets

- d. Store the common elements from both sets into set1

- e. Remove matching elements from set1

- f. Store nonmatching elements from both sets into set2

- g. How to take backup of set1 to new set set3?

39. Create one empty dictionary and add any 5 key:value pairs?

40. Can we list and tuple as values in dictionary? If yes then create one dictionary with tuple and list as values?

41. stuDetails={'Id':100,'Name':'Sai', 'subjects':['SQL Server', 'Oracle', 'Python']}

- a. modify the value of name as 'Durga'

- b. add new pair age:25

- c. display all keys from the stuDetails dictionary?

- d. Display all values from the stuDetails dictionary?

e. Display all subjects which are stored in the stuDetails dictionary?

f. Remove name key from the stuDetails?

g. Create new dictionary from the tuple element? Eg:tup=(1,2,3,4,5)

1. Store value 'Sai' for key 1
2. Store value 'Mahesh' for key 2
3. Store value True for key 3
4. Store value 3+6j for key 4
5. Store value 1000 for key 5

h. remove all pairs from the above dictionary

42. St="python narayana"

how to get the following from the above string

- a. St[3::5]
- b. St[1::1]
- c. St[-1::-3]
- d. St[2::-1]
- e. St[1::15]

FAQs on Data structures

1. How display the given string in lower case?
2. what is the purpose of capitalize()
3. How to display the given string st="PYthon Dev" as "pyTHON dEV"?
4. Is string mutable or immutable object?
5. How to represent an empty string?
6. What is the output format of split function?
7. How to check whether the given string is in lower case or not?
8. How to find the index number of second occurrence 'o' in "oracle developer"?
9. Can we add a new string to the existing string?
10. What is a list and what is the data type of list?
11. How to add a new element to the existing list?
12. What is the insert()?
13. What is the difference between append() and extend()?

14. Can we delete a specific character of a string by using del command?
15. Can we remove all characters from a string by using clear()?
16. Is list mutable or immutable object?
17. How can we decide whether a specific data structure is mutable or immutable?
18. Can we add multiple elements to the existing list at required place?
19. What is the difference between remove() and pop()?
20. How to know the length of a list?
21. Is insertion order preserved or not in list?
22. What is the range() and what is required argument in the range()?
23. What are the different ways to create a list?
24. Can we give duplicate elements in list?
25. How to convert a list into string?
26. What is a tuple?
27. What is the difference between tuple and list?
28. How to create a tuple with one element?
29. Can we create a tuple without () and tuple()? if yes, then how?
30. Can we give duplicate elements in the tuple?
31. How to add a group of elements to the existing tuple?
32. Can we remove elements from tuple?
33. What is the difference between del command and clear()?
34. Can we delete a specific element of a list by using del command?, if yes, then how?
35. Can we delete a specific element from the tuple by using del command? if yes, then how?
36. What is any() and all()?
37. What is a set?
38. Is insertion order preserved or not in tuple?
39. Is insertion order preserved or not in set?
40. Does a set allow duplicate elements?
41. How to add a new element to the existing set?
42. Can we add duplicate element to the existing set?
43. What is issuperset() and issubset()?
44. What is the difference between remove() and discard()
45. What is the main purpose of a set?
46. If a list contains duplicate elements then how to remove duplicate elements from the list?
47. How to convert a list into set?
48. How to convert a set into string?
49. Can we concatenate two sets?
50. Can we multiply a set n number of time?
51. Can we use a list in set?
52. Can we use a set in tuple?
53. Can we concatename a string and a set?
54. How to display [10,30] from [1,3,10,5,7,30,5]?
55. How to reverse a string? example?
56. How to display a string in ascending order(a-z)?
57. How to display a string in descending order(z-a)?
58. How to remove a part of string?
ex: st='oracle dev' o/p:'oracle'
59. How to reverse a list without using reverse()?
60. What is the difference between list and set?

61. What is the main purpose of tuple?
62. What is dictionary?
63. What is the main purpose of dict?
64. Can we add a new key:value pair to the existing dict? how?
65. How to access the value of a specific key from the dict?
66. How to remove a specific key:value pair from the dict?
67. How to update the existing value of a specific key in the dict?
68. How to display all keys from the dict?
69. How to get all values from the dictionary?
70. What are the different ways to create a dict?
71. How to create a dict by using an existing tuple?
72. How to update a dict by using another dict?
73. What is the copy()
74. What is the get () in dict?
75. If the two sets are empty then what is the result of issuperset() and issubset()?
76. If the two sets are having different elements then what is the output of issuperset() and issubset()
77. What is the difference between intersection() and intersection_update()
78. How to create an empty set?
79. What are the different ways to create a string?
80. Is dict mutable or immutable?
81. Can we give a duplicate key in the dict?
82. What the replace()?
83. Can we delete a specific key:value pair by using del command?
84. Can we delete a specific element of a set by using del command?
85. What is the indexing and slicing?
86. Can we generate a heterogeneous list by using range()?
87. How to represent an empty set?
88. What is a list packing and list unpacking?
89. How to perform string packing?
90. Can we concatenate two dicts?
91. Is insertion order preserved or not in dict?
92. What is the difference between title() and capitalize()?
93. What is swapcase()?
94. What is the difference between pop() and popitem()
95. Can we create a new dict by using existing string?
96. When we create a new dict by using an existing obj, then what is the default value for all keys?
97. What is the purpose of join()?
98. Can we count the number of occurrences of a specific element in the set?
99. What is the default delimiter in the split()?
100. Why set is not supporting indexing?

The None Value:

In Python there is a value called `None`, which represents the absence of a value. `None` is the only value of the `NoneType` data type. (Other programming languages might call this value null, nil, or undefined.) Just like the Boolean `True` and `False` values, `None` must be typed with a capital N.

This can be helpful when you need to store something that won't be confused for a real value in a variable. One place where `None` is used is as the return value of `print()`. The `print()` function displays text on the screen, but it doesn't need to return anything in the same way `len()` or `input()` does. But since all function calls need to evaluate to a return value, `print()` returns `None`.

Eg:

```
>>> a=print('Narayana')
Narayana
>>> None==a      True
```

Raw string:

We can place an `r` before the beginning quotation mark of a string to make it a raw string. A raw string completely ignores all escape characters and prints any backslash that appears in the string.

```
>>> print('This \is \python \narayana')
This \is \python
arayana
```

Here, interpreter treated '`\n`' as a new line escaping character, so 'arayana' came in new line.

```
>>> print(r'This \is \python \narayana')
This \is \python \narayana
```

here, interpreter ignores all escaping characters because of '`r`'.

Because this is a raw string, Python considers the backslash as part of the string and not as the start of an escape character. Raw strings are helpful if we are typing string values that contain many backslashes, such as the strings used for regular expressions

Operators

An **operator** is a symbol that tells the compiler to perform certain mathematical or logical manipulations. **Operators** are used in program to manipulate data and variables.

Python language supports the following types of operators.

1. Arithmetic Operators
2. Assignment Operators
3. Comparison (i.e., Relational) Operators
4. Logical Operators
5. Bitwise Operators
6. Membership Operators
7. Identity Operators

Arithmetic Operators:

Assume variable 'a' holds 4 and variable 'b' holds 2, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	$a + b = 6$
-	Subtraction - Subtracts right hand operand from left hand operand	$a - b = 2$
*	Multiplication - Multiplies values on either side of the operator	$a * b = 8$
/	Division - Divides left hand operand by right hand operand	$a/b = 2$
%	Modulus - Divides left hand operand by right hand operand and returns remainder	$a \% b = 0$
**	Exponent - Performs exponential (power) calculation on operators	$a^{**}b = (b \text{ to the power of } a) 16$
//	Floor Division - The division of operands where the result is the	$9//2$ is equal to

quotient in which the digits after the decimal point are removed.

4 and 9.0//2.0 is equal to 4.0

Comparison Operators:

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(a == b) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.

Eg:

```

>>> a=10
>>> b=5
>>> a==b      False
>>> a==15     False
>>> a=='c'    False
>>> a<10     False
>>> a<=10    True
>>> a>5      True
>>> b==5      True
>>> b>=5     True
>>> a!=10    False

```

```
>>> a!=1      True
>>> b!=1      True
>>> b!=5      False
```

Assignment Operators:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	c = a + b will assign value of a + b into c
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**=	Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//=	Floor Division and assigns a value, Performs floor division on operators and assign value to the left operand	c // a is equivalent to c = c // a

Eg1:

```
a = 20
b = 10
```

```
c = 0

print('a value is',a)
print('b value is',b)

c += a
print ("Add AND - Value of c is ", c )

c *= a
print ("Multiply AND - Value of c is ", c )

c /= a
print ("Divide AND - Value of c is ", c )

c=2
c %= a
print ("Modulus AND - Value of c is ", c)

c **= a
print ("Exponent AND - Value of c is ", c)

c // a
print ("Floor Division AND - Value of c is ", c)

c -= a
print ("Subtract AND - Value of c is ", c )
```

output:

```
a value is 20
b value is 10
Add AND - Value of c is 20
Multiply AND - Value of c is 400
Divide AND - Value of c is 20.0
Modulus AND - Value of c is 2
Exponent AND - Value of c is 1048576
Floor Division AND - Value of c is 52428
Subtract AND - Value of c is 52408
```

Bitwise Operators:

Bitwise operator works on bits and performs bit by bit operation. Assume if $a = 60$; and $b = 13$; Now in binary format they will be as follows:

```
a = 0011 1100
b = 0000 1101
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a = 1100 0011
```

There are following Bitwise operators supported by Python language

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	($a \& b$) will give 12 which is 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	($a b$) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	($a ^ b$) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	($\sim a$) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operand's value is moved left by the number of bits specified by the right operand.	$a << 2$ will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand.	$a >> 2$ will give 15 which is 0000 1111

Eg:

```
>>> 10<<2          40
>>> 12<<2          48
```

```
>>> 2<<2      8  
>>> 1<<2      4  
>>> 10>>2     2  
>>> 11>>2     2  
>>> 13>>2     3  
>>> 20>>2     5  
>>> 30>>3     3  
>>> 30>>4     1
```

Logical Operators:

There are following logical operators supported by Python language.

Those are NOT, AND and OR

NOT

X	Not x
True	False
False	True

AND

X	Y	X AND Y
True	False	False
False	True	False
True	True	True
False	False	False

OR

X	Y	X OR Y
True	False	True
False	True	True
True	True	True
False	False	False

Eg1:

```
>>> a=10
>>> b=20
>>> a==10 and b==20      True
>>> a==20 and b==10      False
>>> a==10 and b==10      False
>>> not a==20 and b==20  True
>>> not a==11 and not b==21 True
>>> a==20 or b==20       True
>> not a==12 and not b==10 True
```

Eg2:

```
>>> a=10
>>> b=20
>>> a>=15 and b==20      False
>>> not a>=15 and b==20  True
>>> not a>=15 and not b==20 False
>>> not(not a>=15 and not b==20) True
```

Membership Operators:

In addition to the operators discussed previously, Python has membership operators, which test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators explained below:

Operator	Description	Example
In	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.

not in

Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

$x \text{ not in } y$, here **not in** results in a 1 if x is not a member of sequence y .

Eg1:

```
>>> lst=[1,2,3,4,'Python',True]
>>> 4 in lst      True
>>> 10 in lst     False
>>> 3 not in lst  False
>>> 20 not in lst True
```

Eg2:

```
>>> st="Python Narayana"
>>> 't' in st      True
>>> 'p' in st      False
>>> 'p' not in st   True
>>> 'P' in st      True
>>> 'Na' in st     True
>>> 'Nr' in st    False
>>> ' ' in st      True
>>> 'aa' in st     False
>>> 'aa' not in st True
```

Eg3:

```
>>> d={'Id':100,'Name':'Narayana','Loc':'Hyd'}
>>> 'Id' in d      True
>>> 100 in d       False
>>> 'Sal' not in d  True
>>> 'Sal' in d     False
>>> 'Loc' in d     True
>>> 'Hyd' in d     False
```

Identity Operators:

Identity operators compare the memory locations or references of two objects. There are two Identity operators explained below:

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is not results in 1 if id(x) is not equal to id(y).

Eg 1:

```
>>> st="Sai"  
>>> id(st) 57264032
```

```
>>> st1="Sai"  
>>> id(st1) 57264032
```

```
>>> st is st1  True  
>>> st is not st1 False
```

Eg 2:

```
>>> lst=[1,2,3,4]  
>>> id(lst) 57297136
```

```
>>> lst1=[1,2,3,4]  
>>> id(lst1) 57240512
```

```
>>> lst is lst1 False  
>>> lst is not lst1 True
```

Operators Precedence

The following table lists all operators from highest precedence to lowest.

Operator	Description
**	Exponentiation (raise to the power)
~ + -	complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^	Bitwise exclusive 'OR' and regular 'OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //=-+= *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Exercise on operators:

1. True and True
2. False and True
3. 1 == 1 and 2 == 1
4. "test" == "test"
5. == 1 or 2 != 1
6. True and 1 == 1
7. False and 0 != 0
8. True or 1 == 1
9. "test" == "testing"

Narayana

PYTHON

Narayana

10. `1 != 0 and 2 == 1`
11. `test" != "testing"`
12. `"test" == 1`
13. `not (True and False)`
14. `not (1 == 1 and 0 != 1)`
15. `not (10 == 1 or 1000 == 1000)`
16. `not (1 != 10 or 3 == 4)`
17. `not ("testing" == "testing" and "Zed" == "Cool Guy")`
18. `1 == 1 and not ("testing" == 1 or 1 == 0)`
19. `"chunky" == "bacon" and not (3 == 4 or 3 == 3)`
20. `3 == 3 and not ("testing" == "testing" or "Python" == "Fun")`

Knowledge test

1. What is Python?
2. Name some of the features of Python.
3. Is python a case sensitive language?
4. What are the supported data types in Python?
5. What is the output of print str if str = 'Hello World!'?
6. What is the output of print str[0] if str = 'Hello World!'?
7. What is the output of print str[2:5] if str = 'Hello World!'?
8. What is the output of print str[2:] if str = 'Hello World!'?
9. What is the output of print str * 2 if str = 'Hello World!'?
10. What is the output of print str + "TEST" if str = 'Hello World!'?
11. What is the output of print list if list = ['abcd', 786 , 2.23, 'john', 70.2]?

12. What is the output of print list[0] if list = ['abcd', 786 , 2.23, 'john', 70.2]?

13. What is the output of print list[1:3] if list = ['abcd', 786 , 2.23, 'john', 70.2]?

14. What is the output of print list[2:] if list = ['abcd', 786 , 2.23, 'john', 70.2]?

15. What is the output of print tinylist * 2 if tinylist = [123, 'john']?

16. What is the output of print list + tinylist * 2 if list = ['abcd', 786 , 2.23, 'john', 70.2] and tinylist = [123, 'john']?

17. What are tuples in Python?

18. What is the difference between tuples and lists in Python?

19. What is the output of print tuple if tuple = ('abcd', 786 , 2.23, 'john', 70.2)?

20. What is the output of print tuple[0] if tuple = ('abcd', 786 , 2.23, 'john', 70.2)?

21. What is the output of print tuple[1:3] if tuple = ('abcd', 786 , 2.23, 'john', 70.2)?

22. What is the output of print tuple[2:] if tuple = ('abcd', 786 , 2.23, 'john', 70.2)?

23. What is the output of print tinytuple * 2 if tinytuple = (123, 'john')?

24. What is the output of print tuple + tinytuple if tuple = ('abcd', 786 , 2.23, 'john', 70.2) and tinytuple = (123, 'john')?

25. What are Python's dictionaries?
26. How will you create a dictionary in python?
27. How will you get all the keys from the dictionary?
28. How will you get all the values from the dictionary?
29. How will you convert a string to an int in python?
30. How will you convert a string to a long in python?
31. How will you convert a string to a float in python?
32. How will you convert a object to a string in python?
33. How will you convert a string to a tuple in python?
34. How will you convert a string to a list in python?
35. How will you convert a string to a set in python?
36. How will you create a dictionary using tuples in python?
37. How will you convert a string to a frozen set in python?
38. How will you convert an integer to a character in python?

39. How will you convert a single character to its integer value in python?

40. What is the purpose of ** operator?

41. What is the purpose of // operator?

42. What is the purpose of is operator?

43. What is the purpose of not in operator?

44. What is the purpose break statement in python?

45. What is the purpose continue statement in python?

46. What is the purpose pass statement in python?

47. How can you pick a random item from a list or tuple?

48. How can you pick a random item from a range?

49. How can you get a random number in python?

50. How will you set the starting value in generating random numbers?

51. How will you capitalizes first letter of string?

52. How will you check in a string that all characters are alphanumeric?

53. How will you check in a string that all characters are digits?

54. How will you check in a string that all characters are in lowercase?

55. How will you check in a string that all characters are numerics?

56. How will you check in a string that all characters are whitespaces?

57. How will you check in a string that it is properly titlecased?

58. How will you check in a string that all characters are in uppercase?

59. How will you merge elements in a sequence?

60. How will you get the length of the string?

61. How will you get a space-padded string with the original string left-justified to a total of width columns?

62. How will you convert a string to all lowercase?

63. How will you remove all leading whitespace in string?

64. How will you get the max alphabetical character from the string?

65. How will you get the min alphabetical character from the string?

66. How will you replaces all occurrences of old substring in string with new string?

67. How will you remove all leading and trailing whitespace in string?

68. How will you change case for all letters in string?

69. How will you get titlecased version of string?

70. How will you convert a string to all uppercase?

71. How will you check in a string that all characters are decimal?

72. What is the difference between del() and remove() methods of list?

73. What is the output of len([1, 2, 3])?

74. What is the output of [1, 2, 3] + [4, 5, 6]?

75. What is the output of ['Hi!'] * 4?

76. What is the output of 3 in [1, 2, 3]?

77. What is the output of for x in [1, 2, 3]: print x?

78. What is the output of L[2] if L = [1,2,3]?

79. What is the output of L[-2] if L = [1,2,3]?

80. What is the output of L[1:] if L = [1,2,3]?
81. How will you compare two lists?
82. How will you get the length of a list?
83. How will you get the max valued item of a list?
84. How will you get the min valued item of a list?
85. How will you get the index of an object in a list?
86. How will you insert an object at given index in a list?
87. How will you remove last object from a list?
88. How will you remove an object from a list?
89. How will you reverse a list?
90. How will you sort a list?

CONDITIONAL STATEMENTS

Conditional statements will decide the execution of a block of code based on the expression. The conditional statements return either True or False.

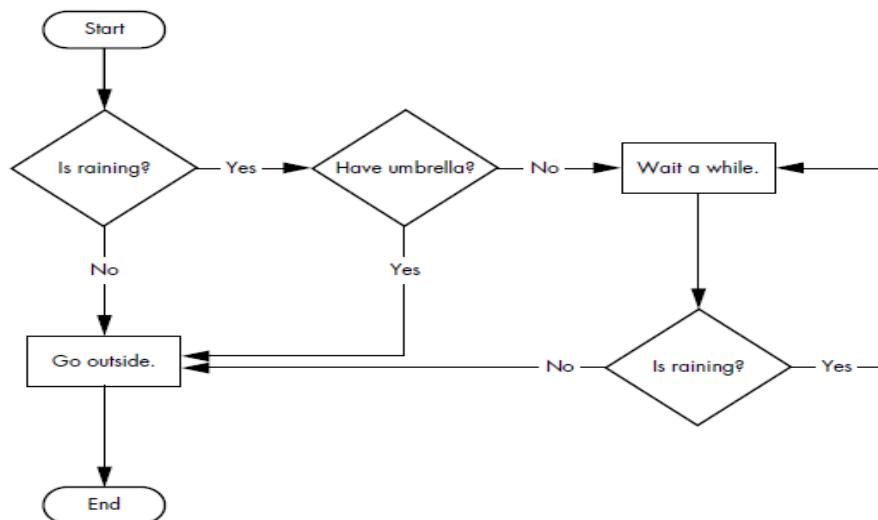
A Program is just a series of instructions to the computer, But the real strength of programming isn't just executing one instruction after another.

Based on how the expressions evaluate, the program can decide to skip instructions, repeat them, or choose one of several instructions to run. In fact, you almost never want your programs to start from the first line of code and simply execute every line, straight to the end. Flow control statements can decide which Python instructions to execute under which conditions.

Python supports four types of conditional statements,

1. Simple IF or IF statement
2. IF – ELSE Statement
3. IF ELSE IF (elif) Statement
4. Nested IF statement

These flow control statements directly correspond to the symbols in a flowchart, the below figure shows a flowchart for what to do if it's raining. just Follow the path made by the arrows from Start to End.



In a flowchart, there is usually more than one way to go from the start to the end. The same is true for lines of code in a computer program. Flowcharts represent these branching points with diamonds, while the other steps are represented with rectangles. The starting and ending steps are represented with rounded rectangles.

But before we learn about flow control statements, we first need to learn how to represent those yes and no options, and we need to understand how to write those branching points as Python code.

IF Statement

The Python if statement is same as it is with other programming languages. It executes a set of statements conditionally, based on the value of a logical expression.

Here is the general form of a one way if statement.

Syntax:

```
if expression :  
    statement_1  
    statement_2  
    ....
```

In the above syntax, expression specifies the conditions it produces either true or false. If the expression evaluates true then the same amount of indented statement(s) following it will be executed. This group of the statement(s) is called a block.

Eg 1:

```
marks=int(input("Enter your percentage of marks to know pass or failed: "))  
  
if marks>=35:  
    print("You are passed ...")
```

output:

```
Enter your percentage of marks to know pass or failed: 45  
You are passed ...
```

Eg 2: compare values and display message

```
a=int(input("Enter First Value: "))
b=int(input("Enter Second Value: "))
c=int(input("Enter third Value: "))
if a>b<c:
    print("a is greater then b and also c")
```

or

To text max value in two values

```
max = a if (a > b) else b;
```

or

```
if a>b:
    x=a
else:
    x=b
```

Eg 3: if name is more then or one character then do all string methods

```
name=input("Enter name: ")

if len(name)>0:
    print "The length of name is: ", len(name)
    print "The given name is: ",name
    print "The capitalization of name is: ",name.capitalize()
    print "The title of the name is: ",name.title()
    print "The lower case form of given string: ",name.lower()
    print "The upper case form of given string: ",name.upper()
    print "The reverse of given string: ", ".join(reversed(name))"
    print "The asc order of given string is: ", ".join(sorted(name))"
    print "The desc order of given string is: ", ".join(reversed(sorted(name)))"
```

Eg4:

```
people = 20
cats = 30
dogs = 15

if people < cats:
    print ("Too many cats! The world is doomed!")

if people > cats:
    print ("Not many cats! The world is saved!")

if people < dogs:
    print ("The world is drooled on!")

if people > dogs:
    print ("The world is dry!")

dogs += 5

if people >= dogs:
    print ("People are greater than or equal to dogs.")

if people <= dogs:
    print ("People are less than or equal to dogs.")

if people == dogs:
    print ("People are dogs.")
```

output:

Too many cats! The world is doomed!
The world is dry!
People are greater than or equal to dogs.
People are less than or equal to dogs.
People are dogs.

IF - ELSE Statement:

In Python if-else statement has two blocks, first block follows the expression and the other block follows the else clause. Here is the syntax.

Syntax:

```
if expression :  
    statement_1  
    statement_2  
    ....  
    else :  
        statement_3  
        statement_4  
    ....
```

In the above case, if the expression evaluates to true then the same amount of indented statements(s) follow the expression and if the expression evaluates to false the same amount of indented statements(s) follow else block.

Eg 1: Write python script to check whether the given number is even or odd number ?

```
a=int(input('Enter Your number: '))  
if a%2==0:  
    print(a," is a even number")  
else:  
    print(a," is a odd number")
```

output:

Enter Your number: 20
20 is a even number

Or

Enter Your number: 21
21 is a odd number

Eg 2: Write a python script to know passed or failed

```
marks=int(input("Enter your percentage of marks to know pass or failed: "))
if marks>=35:
    print("Your are passed")
else:
    print("Your are failed")
```

output:

Enter your percentage of marks to know pass or failed: 20

Your are failed`

Or

Enter your percentage of marks to know pass or failed: 40

Your are passed

Eg 3: check whether he entered proper name or not, if it is more than or equal to one character then do all string operations else display “please enter valid name”.

```
name=input("Enter name: ")

if len(name)<=0:
    print("Please enter a valid name")
else:
    print ("The length of name is: ", len(name))
    print ("The given name is: ",name)
    print ("The capitalization of name is: ",name.capitalize())
    print ("The title of the name is: ",name.title())
    print ("The lower case form of given string: ",name.lower())
    print ("The upper case form of given string: ",name.upper())
    print ("The reverse of given string: ", ".join(reversed(name)))
    print ("The asc order of given string is: ", ".join(sorted(name)))
    print ("The desc order of given string is: ", ".join(reversed(sorted(name))))
```

Output:

Enter name: python dev

The length of name is: 10

The given name is: python dev

The capitalization of name is: Python dev
The title of the name is: Python Dev
The lower case form of given string: python dev
The upper case form of given string: PYTHON DEV
The reverse of given string: ved nohtyp
The asc order of given string is: dehnoptvy
The desc order of given string is: yvtponhed

Output:

Enter name:

Please enter a valid name

Eg 4: enter two values and find out bigger value

```
num1=int(input("Enter first Number: "))  
num2=int(input("Enter second Number: "))  
  
if num1>num2:  
    print (num1," is greater then ",num2)  
else:  
    print (num1, "is smaller then ",num2)
```

output:

```
Enter first Number: 10  
Enter second Number: 5  
10 is greater then 5
```

```
Enter first Number: 5  
Enter second Number: 7  
5 is smaller then 7
```

elif statement:

It will check the condition1 first, if the condition1 is true then it will execute the block of statements which are following the condition, if the condition1 is false then it will check the condition2. If the condition2 is true then it will execute the block of statements which follow the condition2, if the condition2 is false then it will check the condition3, like this it will check all conditions. If all conditions are false then it will execute the else block.

Syn:

```
if condition1 :  
    statement_1  
    statement_2  
elif condition2:  
    statement_3  
    statement_4  
elif condition3:  
    statement_5  
    statement_6  
elif condition4:  
    statement_7  
    statement_8  
else :  
    statement_9  
    statement_10  
....
```

Eg 1: marks example

```
marks=int(input("Enter your percentage of marks : "))  
if marks>=0 and marks<35:  
    print("You are failed")  
elif marks>=35 and marks<50:  
    print("You got 3rd class")  
elif marks>=50 and marks<60:  
    print("You got 2nd class")  
elif marks>=60 and marks<75:  
    print("you got 1st class")
```

```
elif marks>=75 and marks<=100:  
    print("you got distinction")  
else:  
    print('Invalid marks')
```

output:

```
Enter your percentage of marks : 50  
You got 2nd class
```

```
Enter your percentage of marks : 90  
you got distinction
```

```
Enter your percentage of marks : 20  
You are failed
```

Eg 2: food timings example

```
time=int(input("Enter your time: "))  
if time>7 and time <10:  
    print("Its time to have Breakfast..")  
elif time>=10 and time<12:  
    print("Its time to have Brunch..")  
elif time>=12 and time<15:  
    print("Its time to have Lunch..")  
elif time>=15 and time<18:  
    print("Its time to have Snacks")  
elif time>=18 and time<20:  
    print("Its time to have Dinner")  
elif time>=20 and time<=24:  
    print("Its sleeping time")  
elif time>=1 and time <=7:  
    print("Its sleeping time")  
else:  
    print('you entered invalid time')
```

output:

Enter your time: 8
Its time to have Breakfast..

Enter your time: 13
Its time to have Lunch..

Enter your time: -2
you entered invalid time

Enter your time: 3
Its sleeping time

Eg3: Find bigger value of two given values

```
a=int(input("Enter First Value: "))  
b=int(input("Enter Second Value: "))  
if a>b:  
    print(a,'is greater than ',b)  
elif b>a:  
    print(b,'is greater than ',a)  
else:  
    print(a,'and',b,'are same values')
```

Output1:

Enter First Value: 10
Enter Second Value: 15
15 is greater than 10

Output2:

Enter First Value: 10
Enter Second Value: 10
10 and 10 are same values

Output3:

Enter First Value: 15
Enter Second Value: 10

15 is greater then 10

Eg4: Find biggest value of three given values

```
a=int(input("Enter First Value: "))
b=int(input("Enter Second Value: "))
c=int(input("Enter third Value: "))

if a>b and a>c:
    print(a,'is greater then ',b,'and',c)
elif b>c:
    print(b,'is greater then ',a,'and',c)
elif a==b==c:
    print('three are same values')
else:
    print(c,'is greater then ',a,'and',b)
```

Output 1:

```
Enter First Value: 10
Enter Second Value: 20
Enter third Value: 15
20 is greater then 10 and 15
```

Output 2:

```
Enter First Value: 10
Enter Second Value: 5
Enter third Value: 30
30 is greater then 10 and 5
```

Output 3:

```
Enter First Value: 10
Enter Second Value: 3
Enter third Value: 5
10 is greater then 3 and 5
```

Output 4:

```
Enter First Value: 10
Enter Second Value: 10
Enter third Value: 10
```

Three are same values

Eg5: how to find the type of user data

```
var1 = eval(input('Enter any type of data: '))
if (type(var1) == int):
    print("Type of the user data is Integer")
elif (type(var1) == float):
    print("Type of the user data is Float")
elif (type(var1) == complex):
    print("Type of the user data is Complex")
elif (type(var1) == bool):
    print("Type of the user data is Bool")
elif (type(var1) == str):
    print("Type of the user data is String")
elif (type(var1) == tuple):
    print("Type of the user data is Tuple")
elif (type(var1) == set):
    print("Type of the user data is Set")
elif (type(var1) == dict):
    print("Type of the user data is Dictionaries")
elif (type(var1) == list):
    print("Type of the user data is List")
else:
    print("Type of the user data is Unknown")
```

output1:

```
Enter any type of data: 10
Type of the user data is Integer
```

output2:

```
Enter any type of data: 20.5
Type of the user data is Float
```

output3:

```
Enter any type of data: 'Python Narayana'
Type of the user data is String
```

output4:

Enter any type of data: [10,20,30]

Type of the user data is List

output5:

Enter any type of data: {2,3,4,6}

Type of the user data is Set

output6:

Enter any type of data: {1:'a',2:'b'}

Type of the user data is Dictionaries

Eg6:

```
age=int(input("Enter your age: "))

if age>0 and age<4:
    print('You are cute and little small baby')
elif age>=4 and age<9:
    print('You are a primary school student')
elif age>=9 and age<16:
    print('You are a high school student')
elif age>16 and age<=18:
    print('Congratulations, Now you are a intermediate college student')
elif age>18 and age<21:
    print('You are a undergraduate student')
elif age>21 and age<=26:
    print('I hope you are searching for job, all the best')
elif age>60:
    print('I think you are doing job,,, congratulations.....')
elif age==60:
    print('It is better to retire from job')
else:
    print('You can play with your grand childrens if you are alive')
```

Output1

Enter your age: 7
You are a primary school student

Output 2:

Enter your age: 12
You are a high school student

Output 3:

Enter your age: 20
You are a undergraduate student

Output 4:

Enter your age: 12
You are a high school student

Output 5:

Enter your age: 33
I think you are doing job,,, congratulations.....

Output 6:

Enter your age: 55
I think you are doing job,,, congratulations.....

Output 7:

Enter your age: 59
I think you are doing job,,, congratulations.....

Output 8:

Enter your age: 60
It is better to retire from job

Output 9:

Enter your age: 88
You can play with your grand childrens if you are alive

Eg7:

```
people = 30
```

```
cars = 40
```

```
buses = 15
```

```
if cars > people:  
    print ("We should take the cars.")  
elif cars < people:  
    print ("We should not take the cars.")  
else:  
    print ("We can't decide.")  
  
if buses > cars:  
    print ("That's too many buses.")  
elif buses < cars:  
    print ("Maybe we could take the buses.")  
else:  
    print ("We still can't decide.")  
if people > buses:  
    print ("Alright, let's just take the buses.")  
else:  
    print ("Fine, let's stay home then.")
```

output:

We should take the cars.

Maybe we could take the buses.

Alright, let's just take the buses.

Eg8:

```
print ("You enter a dark room with two doors. Do you go through door #1 or door #2?")
door = input("> ")
if door == "1":
    print ("There's a giant bear here eating a cheese cake. What do you do?")
    print ("1. Take the cake.")
    print ("2. Scream at the bear.")

    bear = input("> ")
    if bear == "1":
        print ("The bear eats your face off. Good job!")
    elif bear == "2":
        print ("The bear eats your legs off. Good job!")
    else:
        print ("Well, doing %s is probably better. Bear runs away." % bear)

elif door == "2":
    print ("You stare into the endless abyss at Cthulhu's retina.")
    print ("1. Blueberries.")
    print ("2. Yellow jacket clothespins.")
    print ("3. Understanding revolvers yelling melodies.")

    insanity = input("> ")
    if insanity == "1" or insanity == "2":
        print ("Your body survives powered by a mind of jello. Good job!")
    else:
        print ("The insanity rots your eyes into a pool of muck. Good job!")
else:
    print ("You stumble around and fall on a knife and die. Good job!")
```

output:

```
You enter a dark room with two doors. Do you go through door #1 or door #2?
> 1
There's a giant bear here eating a cheese cake. What do you do?
1. Take the cake.
2. Scream at the bear.
> 1
The bear eats your face off. Good job!
```

Nested if statement:

Python supports using if statements in another if statement.

If the upper if condition is true then the inner if will be evaluated, if the upper if is false then it will not check the inner if statement

If condition1:

 If condition2:

 If condition31:

 Statement1

 Statement2

 Elif condition32:

 Statement1

 Statement2

 Elif condition33:

 Statement1

 Statement2

 Else:

 Statement1

 Statement2

 Elif condition21:

 Statement1

 Statement2

 Elif condition22:

 Statement1

 Statement2

 Else:

 Statement1

 Statement2

Elif condition11:

 Statement1

 Statement2

Else:

 Statement1

Statement2

Eg 1:

Database contains only male records, if user enters about female then display 'female records are not available' and if user enters about male then check employees name, if male employee name existed then display his details, if that name is not available then display 'nobody is there with that name'. If he enters any wrong gender then display 'you entered wrong gender'. Finally display 'thank you' at the end of result.

male records displaying

```
gender=input("Enter gender: ")
name= input("Enter name ")

if gender=="Female" or gender=="Male":
    if gender=='Female':
        print("Female records are not available")
    else:
        if name=='Satya' and gender=="Male":
            print("Satya is from hyd and working as SE")
        elif name=='Narayana' and gender=="Male":
            print("Narayana is from nagpura and working as ASE")
        elif name=='Nani' and gender=="Male":
            print("Nani is from Hyd and working as TL")
        else:
            print("No body is there with that name")
    else:
        print("You entered wrng gender: ")
print("Thank")
output:
```

```
Enter gender: Male
Enter name Satya
Satya is from hyd and working as SE
Thank
```

Or

Male records display example

```
gender=input("Enter your gender: ")  
name=input("Enter your name: ")  
  
if gender=="Male" or gender=="Female":  
    if gender!="Female":  
        if name=="Satya":  
            print("Satya is from Hyd and having 10 years exp")  
        elif name=="Sai":  
            print("Sai is from mumbai and having 20 years exp")  
        elif name == "narayana":  
            print("Narayana is having 6 years exp and from hyd")  
        else:  
            print("sorry, your name is not available in the database")  
    else:  
        print("Sorry Female records are not available in the database..")  
else:  
    print("Sorry, You entered invalid gender")  
  
print("Thank You .....
```

output:

```
Enter your gender: Male1  
Enter your name: Satya  
Sorry, You entered Male1 , its invalid gender  
Please check once  
Thank You .....
```

```
Enter your gender: Male  
Enter your name: Sai  
Sai is from mumbai and having 20 years exp  
Thank You .....
```

```
Enter your gender: Female  
Enter your name: Renu  
Sorry Female records are not available in the database..  
Thank You .....
```

Eg2: checking the eligibility of interview

```
name=input('Enter you name: ')  
  
qual= input('Enter your qualification: ')  
  
year=int(input('Enter passed out year: '))  
  
per=eval(input('Enter your percentage: '))  
  
qual=qual.lower()  
  
name=name.capitalize()  
  
if qual=='btech' or qual=='be':  
  
    if year==2016 or year==2017:  
  
        if per>=0 and per<35:  
  
            print('Hello',name,',dont come to interview because you got ',per,'percentage only')  
  
        elif per>=35 and per<50:  
  
            print('Hello',name,', please come to interview with work exp')  
  
        elif per >=50 and per<60:  
  
            print('Hello',name,',please come to interview after two months')  
  
        elif per>=60 and per<75:  
  
            print('Hello',name,', please come to interview tomorrow')  
  
        elif per>=75 and per <=100:  
  
            print('Hello',name,',please come to interview today because you got ',per,'percentage')  
  
    else:
```

```
print('Hello',name,'you entered',per,'. Its invalid percentage')

elif year<2016:

    print('Hello',name,'You entered ',year,'. So you are not fresher')

else:

    print('Hello',name,'You entered ',year,'. Its invalid passedout year')

else:

    print('Hello',name,', you entered ',qual,'. So you are not eligible')

    print('thank you for your interest in our company')
```

output 1:

```
Enter you name: Nani
Enter your qualification: Btech
Enter passed out year: 2017
Enter your percentage: 88
Hello Nani ,please come to interview today because you got 88 percentage
thank you for your interest in our company
```

output 2:

```
Enter you name: Nani
Enter your qualification: Degree
Enter passed out year: 2016
Enter your percentage: 75
Hello Nani , you entered degree . So you are not eligible
thank you for your interest in our company
```

output 3:

```
Enter you name: Krishna
Enter your qualification: be
Enter passed out year: 2018
Enter your percentage: 90
Hello Krishna You entered 2018 . Its invalid passedout year
thank you for your interest in our company
```

output 4:

```
Enter you name: Madhu
```

```
● Enter your qualification: btech
● Enter passed out year: 2014
● Enter your percentage: 77
● Hello Madhu You entered 2014 . So you are not fresher
● thank you for your interest in our company
```

● output 5:

```
● Enter you name: Venu
● Enter your qualification: btech
● Enter passed out year: 2017
● Enter your percentage: 10
● Hello Venu ,dont come to interview because you got 10 percentage only
● thank you for your interest in our company
```

● Eg3

```
country=input("Please Enter your country name: ")
state=input('Please enter your state name: ')
district=input('Please enter your district name: ')
mandal=input('Please enter your mandal name: ')
city=input('Please enter your village name: ')
friend_name=input('Please enter your father name: ')

country=country.lower()
state=state.lower()
district=district.lower()
mandal=mandal.lower()
city=city.lower()
friend_name=friend_name.lower()

if country=='india':
    if state=='ap':
        if district=='guntur':
            if mandal=='veldhurthi':
                if city=='macherla':
                    if friend_name=='venkat':
                        print('He finished B.Tech and searching for job')
                    elif friend_name=='krishna':
                        print('He is working in Wipro in Hyderabad')
```

```
elif friend_name=='veni':  
    print('She is studying degree in Guntur')  
elif friend_name=='sreenu':  
    print('He is working as software Enginner in Bangalore')  
else:  
    print('Nobody is there named ',friend_name)  
else:  
    print('You eneterd wrong city name. It should be macharla.... please try again,,')  
else:  
    print('You eneterd wrong mandal name. It shold be veldhurthi... plase try again,,')  
else:  
    print('You eneterd wrong district name. It shold be guntur... plase try again,,')  
else:  
    print('You eneterd wrong state name. It shold be ap... plase try again,,')  
else:  
    print('You eneterd wrong country name. It shold be india... plase try again,,')
```

Output1:

```
Please Enter your country name: india  
Please enter your state name: ap  
Please enter your district name: guntur  
Please enter your mandal name: veldhurthi  
Please enter your village name: macherla  
Please enter your father name: venkat  
He finished B.Tech and searching for job
```

Output2:

```
Please Enter your country name: US  
Please enter your state name: AP  
Please enter your district name: GUNTUR  
Please enter your mandal name: Veldhurthi  
Please enter your village name: Macherla  
Please enter your father name: Sreenu  
You eneterd wrong country name. It shold be india... plase try again,,
```

Output3:

```
Please Enter your country name: INDIA  
Please enter your state name: Telangana  
Please enter your district name: Hyderabad  
Please enter your mandal name: Hyderabad  
Please enter your village name: KPHB  
Please enter your father name: Satya  
You eneterd wrong state name. It shold be ap... plase try again,,
```

Iterative statements

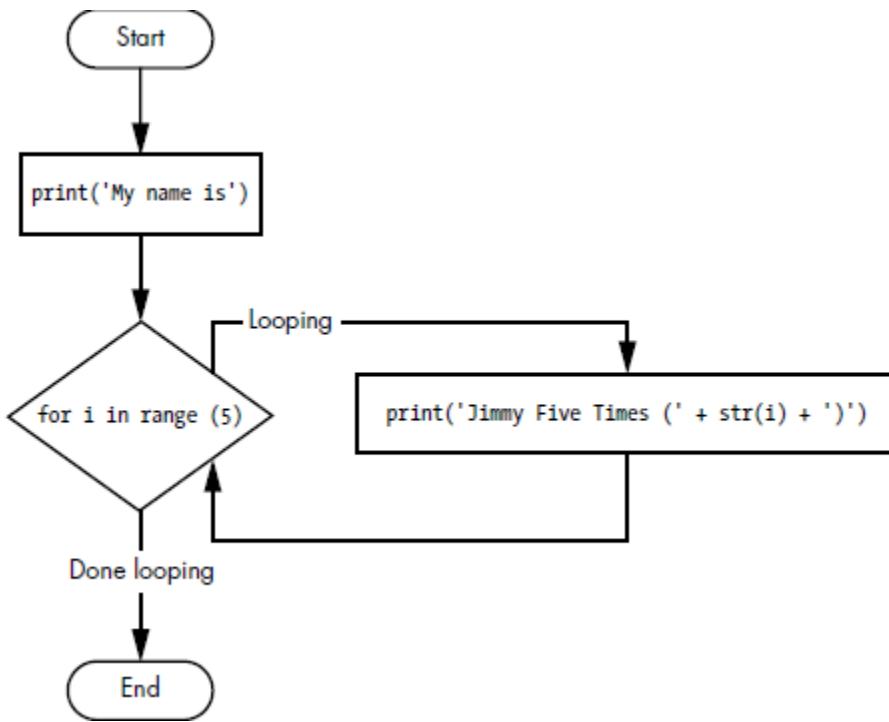
For loop:

For loop allows a code block to be repeated a certain number of times.

Eg1:

```
print('My name is')
for i in range(5):
    print('Jimmy Five Times (' + str(i) + ')')
```

The code in the for loop's clause is run five times. The first time it is run, the variable *i* is set to 0. The `print()` call in the clause will print `Jimmy Five Times (0)`. After Python finishes an iteration through all the code inside the for loop's clause, the execution goes back to the top of the loop, and the for statement increments *i* by one. This is why `range(5)` results in five iterations through the clause, with *i* being set to 0, then 1, then 2, then 3, and then 4. The variable *i* will go up to, but will not include, the integer passed to `range()`.



Output:

```
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

Note:

We can use break and continue statements inside for loops as well. The continue statement will continue to the next value of the for loop's counter, as if the program execution had reached the end of the loop and returned to the start. In fact, we can use continue and break statements only inside while and for loops. If we try to use these statements elsewhere, Python will give you an error.

Eg2: for loop on string

```
st="Python Developer"
for i in st:
    print(i)
```

output:

```
P
y
t
h
o
n

D
e
v
e
l
o
p
e
r
```

Eg3: for loop on list object

```
lst=[1,2,3.5,"Python",4+5j,True]
for i in lst:
    print(i)
```

output:

```
1
2
3.5
Python
(4+5j)
True
```

Eg4: for loop on tuple object

```
tup=(1,2,3,True,False,"narayana")
for i in tup:
    print(i)
```

output:

```
1
2
3
True
False
narayana
```

Eg5: for loop on set object

```
se={2,3,'Python',0,0,True,2+6j,'Narayana'}
for i in se:
    print(i)
```

output:

```
0
Python
2
3
True
```

Narayana
(2+6j)

Eg6: for loop on dict object

```
dic={1:'a',2:'b','c':45}
```

```
for i in dic:  
    print(i)
```

output:

```
1  
2  
c
```

Eg7: displaying 10th table by using range function

```
for i in range(1,11):  
    r=10*i  
    print('10', '*', i, '=', r)
```

output:

```
10 * 1 = 10  
10 * 2 = 20  
10 * 3 = 30  
10 * 4 = 40  
10 * 5 = 50  
10 * 6 = 60  
10 * 7 = 70  
10 * 8 = 80  
10 * 9 = 90  
10 * 10 = 100
```

For loop with else

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts.

break statement can be used to stop a for loop. In such case, the else part is ignored.

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

Here is an example to illustrate this.

Eg:

```
MyList = ["Narayana", 100, "Python", True]

for i in MyList:
    print(i)
else:           #here else is optional, without else also possible
    print("Specified ", len(MyList), " items over.")
```

Output:

```
Narayana
100
Python
True
Specified 4 items over.
```

While Loop:

We can make a block of code execute over and over again with a while statement. The code in a while clause will be executed as long as the while statement's condition is True.

In Python programming language, A **while** loop statement repeatedly executes a target statement as long as a given condition is true.

We can see that a while statement looks similar to an if statement. The difference is in how they behave. At the end of an if clause, the program execution continues after the if statement. But at the end of a while clause, the program execution jumps back to the start of the while statement. The while clause is often called the while loop or just the loop.

An if statement and a while loop that use the same condition and take the same actions based on that condition.

While expression:

```
Statement1
Statement2
Statement3
.....
```

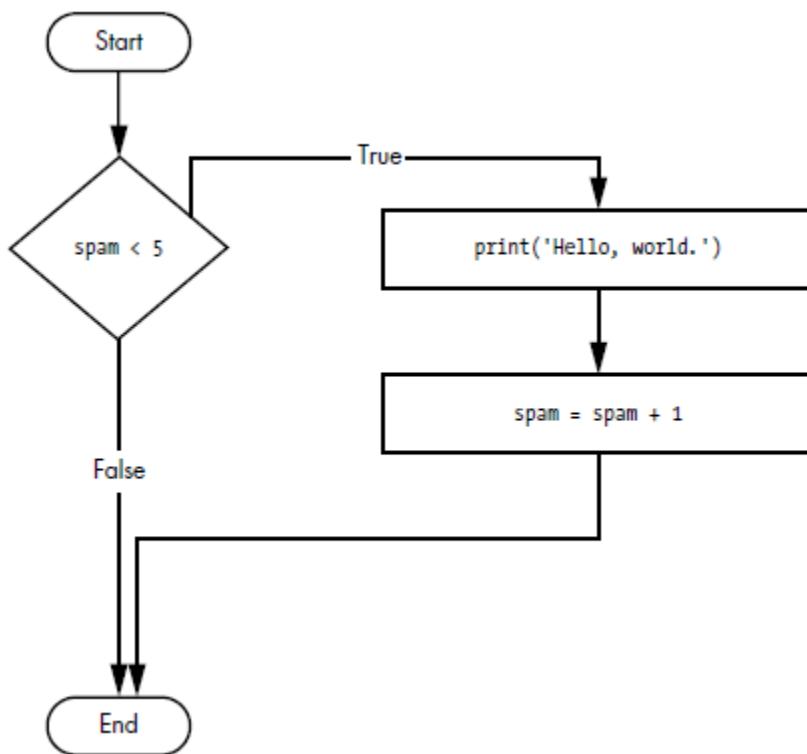
Here lets take an example in both if statement and while loop

#by using if statement

```
spam = 0
if spam<5:
    print("Hello, world.")
    spam = spam + 1
```

output:

Hello, world.

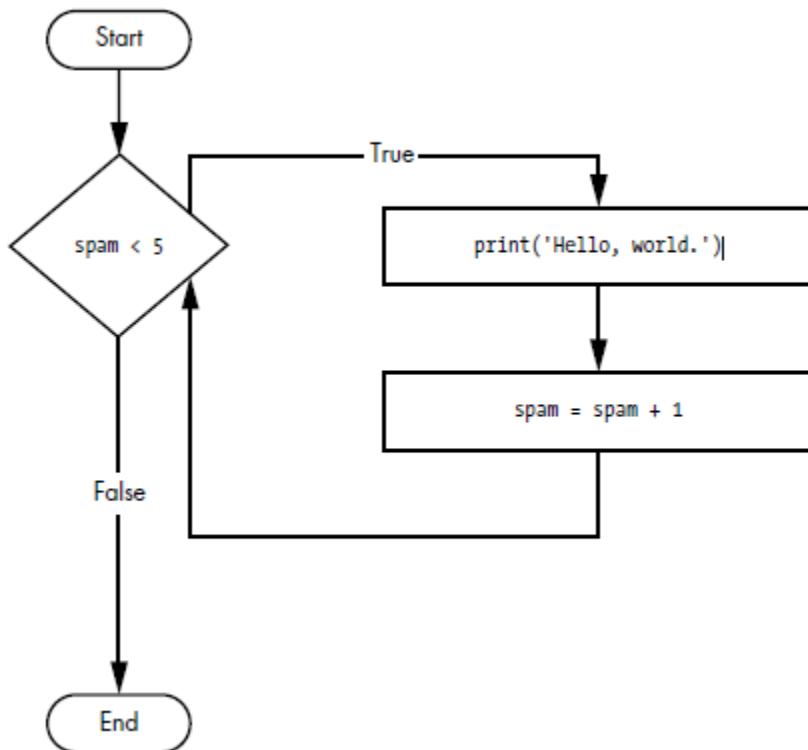


#by using while loop

```
spam=0
while spam<5:
    print("Hello, world.")
    spam=spam+1
```

output:

```
Hello, world.  
Hello, world.  
Hello, world.  
Hello, world.  
Hello, world.
```



These statements are similar—both if and while check the value of spam, and if it is less than five, they print a message. But when you run these two code blocks, something very different happens for each one. For the if statement, the output is simply "Hello, world.". But for the while statement, it's "Hello, world." repeated five times!

Eg1: Display first 5 numbers

```
num=1  
while(num<=5):  
    print("the count is: ",num)  
    num+=1  
print("Thank You")
```

output:

```
the count is: 1  
the count is: 2  
the count is: 3  
the count is: 4  
the count is: 5
```

Thank You

Eg2: Display squares of first 10 numbers

```
myNum = 11  
myVar = 0  
while myVar < myNum :  
    print 'Square of ' + str(myVar) + ' is ' + str(myVar ** 2)  
    myVar += 1
```

output:

```
Square of 0 is 0  
Square of 1 is 1  
Square of 2 is 4  
Square of 3 is 9  
Square of 4 is 16  
Square of 5 is 25  
Square of 6 is 36  
Square of 7 is 49  
Square of 8 is 64  
Square of 9 is 81  
Square of 10 is 100
```

Eg3: Display sum of all natural number of given number

```
num = 5  
sum = 0  
i = 1  
while i <= num:  
    sum = sum + i
```

```
i = i+1  
print("The sum is", sum)
```

outout: The sum is 15

Eg4: Display all even number to before given number

```
Target_Num = 10  
Var = 1  
while Var < Target_Num :  
    if Var % 2 != 0 :  
        Var += 1  
        continue  
    print('This number = ' + str(Var))  
    Var += 1  
print("Displayed all even number before ",Var)
```

output:

```
This number = 2  
This number = 4  
This number = 6  
This number = 8  
Displayed all even number before 10
```

Eg5: checking username and password

```
while True:  
    name=input("Enter Name: ")  
    if name != 'Narayana':  
        continue  
    print("Hello ", name , ", Please enter your password..")  
    pwd=input("Enter Password: ")  
    if pwd=='DurgaSoft':  
        print("You entered correct details")  
        print("Congratulations ", name)  
        break  
    print("Thank you ", name)
```

output:

```
Enter Name: Narayana
Hello Narayana , Please enter your password..
Enter Password: DurgaSoft
You entered correct details
Congratulations Narayana
Thank you Narayana
```

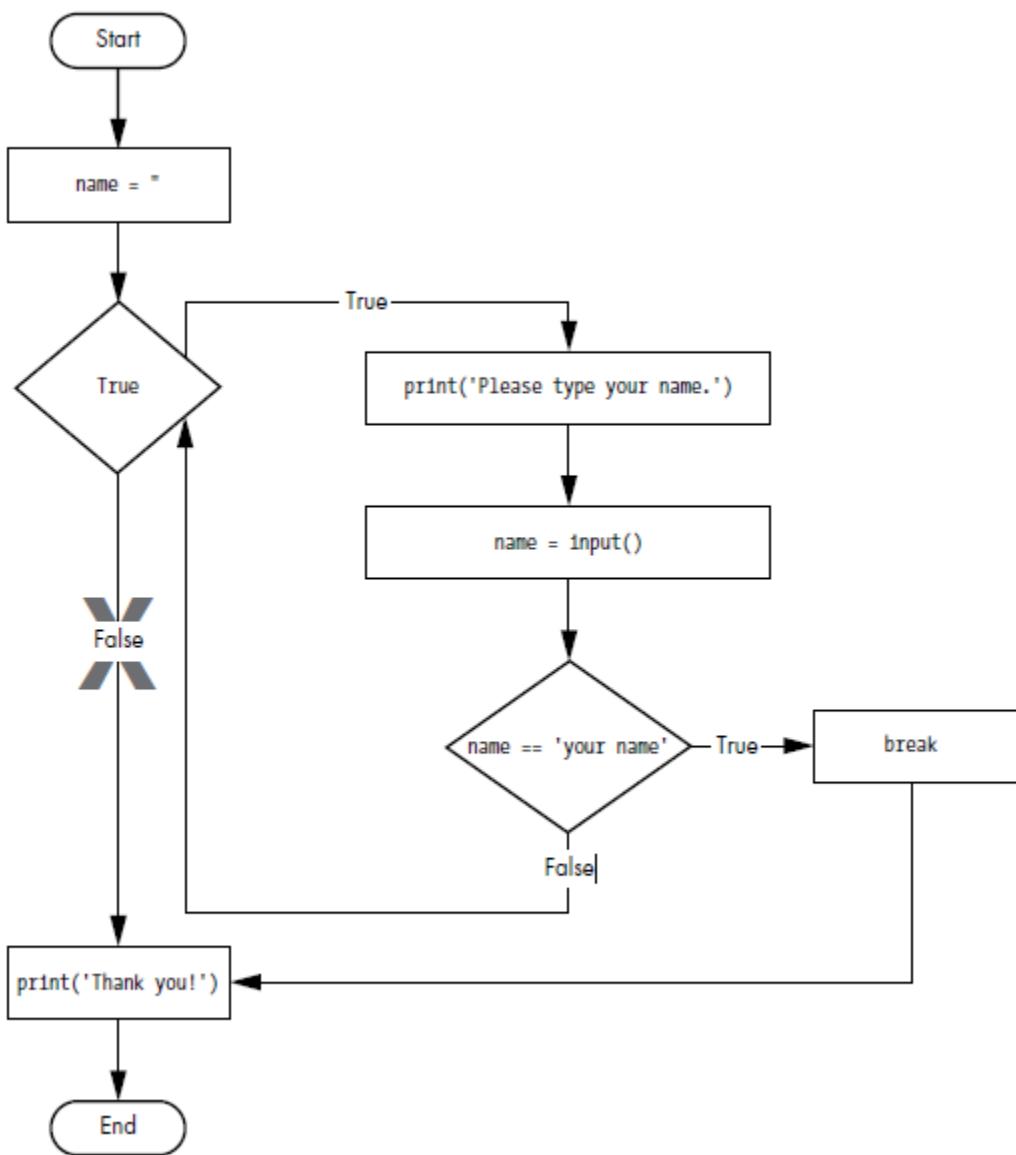
Break statement:

There is a shortcut to getting the program execution to break out of a while loop's clause early. If the execution reaches a break statement, it immediately exits the while loop's clause. In code, a break statement simply contains the break keyword.

Eg1:

```
while True:
    print('Please type your name.')
    name = input()
    if name == 'python narayana':
        break
    print('Thank you!')
```

Flow

**output 1:**

Please type your name.
python narayana

output 2:

Please type your name.
python
Thank you!
Please type your name.
narayana

```
Thank you!  
Please type your name.  
Python narayana  
Thank you!  
Please type your name.  
Python Narayana  
Thank you!  
Please type your name.  
python narayana
```

Eg 2:

```
birthdays = {'Nagaraju': 'Apr 1', 'Nani': 'Dec 12', 'Satya': 'Mar 4'}  
while True:  
    print('Enter a name: (blank to quit)')  
    name = input()  
    if name == "":  
        break  
    if name in birthdays:  
        print(birthdays[name] + ' is the birthday of ' + name)  
    else:  
        print('I do not have birthday information for ' + name)  
        print('What is their birthday?')  
        bday = input()  
        birthdays[name] = bday  
        print('Birthday database updated.')
```

output:

```
Enter a name: (blank to quit)
Narayana
I do not have birthday information for Narayana
What is their birthday?
mar 10
Birthday database updated.
Enter a name: (blank to quit)
Koti
I do not have birthday information for Koti
What is their birthday?
may 20
Birthday database updated.
Enter a name: (blank to quit)
Nagaraju
Apr 1 is the birthday of Nagaraju
Enter a name: (blank to quit)
Satya
Mar 4 is the birthday of Satya
Enter a name: (blank to quit)
Koti
may 20 is the birthday of Koti
Enter a name: (blank to quit)
Sai
I do not have birthday information for Sai
What is their birthday?
dec 12
Birthday database updated.
Enter a name: (blank to quit)
```

Continue statement:

Like break statements, continue statements are used inside loops. When the program execution reaches a continue statement, the program execution immediately jumps back to the start of the loop and reevaluates the loop's condition.

Eg1:

```
while True:  
    print('Who are you?')  
    name = input()  
    if name != 'narayana':  
        continue  
    print('Hello, Narayana. What is the password? (It is my mailid.)')  
    password = input()  
    if password == 'pythonnarayana':  
        break  
    print('Access granted.')
```

output 1:

```
Who are you?  
narayana  
Hello, Narayana. What is the password? (It is my mailid.)  
pythonnarayana  
Access granted.
```

Outpu 2:

```
Who are you?  
Narayana  
Who are you?  
NARAYANA  
Who are you?  
Nagaraju
```

```
Who are you?  
Komali  
Who are you?  
narayana  
Hello, Narayana. What is the password? (It is my mailid.)  
python  
Who are you?  
narayana  
Hello, Narayana. What is the password? (It is my mailid.)  
pythonnarayana  
Access granted.
```

While loop with string isX methods

Along with `islower()` and `isupper()`, there are several string methods that have names beginning with the word `is`. These methods return a Boolean value that describes the nature of the string. Here are some common `isX`

`isalpha()` returns True if the string consists only of letters and is not blank.

`isalnum()` returns True if the string consists only of letters and numbers and is not blank.

`isdecimal()` returns True if the string consists only of numeric characters and is not blank.

`isspace()` returns True if the string consists only of spaces, tabs, and newlines and is not blank.

`istitle()` returns True if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.

```
>>> 'python'.isalpha()          True  
>>> 'python2'.isalpha()        False  
>>> 'python2'.isalnum()        True
```

```
>>> 'python'.isalnum()           True
>>> '1345'.isalnum()           True
>>> '1345'.isdecimal()         True
>>> '134.5'.isdecimal()        False
>>> ''.isspace()               True
>>> 'py 3'.isspace()           False
>>> 'This Is Python Narayana'.istitle()  True
>>> 'This Is python Narayana'.istitle()   False
>>> 'This Is PYTHON Narayana'.istitle()   False
>>> 'This Is Python 1037'.istitle()       True
```

The `isX` string methods are helpful when we need to validate user input. For example, the following

program repeatedly asks users for their age and a password until they provide valid input.

```
while True:
    print('Enter your age:')
    age = input()
    if age.isdecimal():
        break
    print('Please enter a number for your age.')
while True:
    print('Select a new password (letters and numbers only):')
    password = input()
    if password.isalnum():
        break
    print('Passwords can only have letters and numbers.')  
output1:
```

Enter your age:

28

Select a new password (letters and numbers only):

1234abcd

output2:

Enter your age:

12a

Please enter a number for your age.

Enter your age:

ab28

Please enter a number for your age.

Enter your age:

28

Select a new password (letters and numbers only):

1234@abcd

Passwords can only have letters and numbers.

Select a new password (letters and numbers only):

12345

While loop with else

We can have an optional `else` block with while loop as well.

The `else` part is executed if the condition in the while loop evaluates to `False`. The while loop can be terminated with a [break statement](#).

In such case, the `else` part is ignored. Hence, a while loop's `else` part runs if no break occurs and the condition is false.

```
cnt = 0
while cnt < 3:
    print("Now we are in Inside loop")
    cnt = cnt + 1
```

```
else:  
    print("Now we are in else block")  
  
output:  
Now we are in Inside loop  
Now we are in Inside loop  
Now we are in Inside loop  
Now we are in else block
```

Eg1:

```
i = 0  
numbers = []  
  
while i < 6:  
    print ("At the top i is %d" % i)  
    numbers.append(i)  
  
    i = i + 1  
    print ("Numbers now: ", numbers)  
    print ("At the bottom i is %d" % i)  
  
print ("The numbers: ")  
  
for num in numbers:  
    print (num)
```

output:

```
At the top i is 0  
Numbers now: [0]  
At the bottom i is 1  
  
At the top i is 1  
Numbers now: [0, 1]  
At the bottom i is 2  
  
At the top i is 2  
Numbers now: [0, 1, 2]  
At the bottom i is 3
```

Narayana

PYTHON

Narayana

At the top i is 3

Numbers now: [0, 1, 2, 3]

At the bottom i is 4

At the top i is 4

Numbers now: [0, 1, 2, 3, 4]

At the bottom i is 5

At the top i is 5

Numbers now: [0, 1, 2, 3, 4, 5]

At the bottom i is 6

The numbers:

0

1

2

3

4

5

While loop working list data structure

Generally, When we first begin writing programs, it's tempting to create many individual variables to store a group of similar values. For example, if I wanted to store subjects names that i know, I might be tempted to write code like this:

```
>>> sub1="SQL Server"  
>>> sub2="MSBI"  
>>> sub3="Python"  
>>> sub4="Django"  
>>> sub5="Oracle"  
>>> sub6="Power BI"  
>>> sub7="MYSQL"
```

This is a bad way to write code like above. For one thing, if the number of subjects changes, our program will never be able to store more subjects than we have variables. These types of programs also have a lot of duplicate or nearly identical code in them.

Consider how much duplicate code is in the following program,

```
print('Enter the name of sub 1:')  
SubName1 = input()  
print('Enter the name of sub 2:')  
SubName2 = input()  
print('Enter the name of sub 3:')  
SubName3 = input()  
print('Enter the name of sub 4:')  
SubName4 = input()  
print('Enter the name of sub 5:')  
SubName5 = input()  
print('Enter the name of sub 6:')
```

```
SubName6 = input()  
print('The subjects are ', SubName1 ,',', SubName2 ,',',SubName3 ,',',SubName4 ,',', SubName5 ,',',SubName6)
```

output:

```
Enter the name of sub 1:
```

```
"SQL Server"
```

```
Enter the name of sub 2:
```

```
"MSBI"
```

```
Enter the name of sub 3:
```

```
"Python"
```

```
Enter the name of sub 4:
```

```
"Django"
```

```
Enter the name of sub 5:
```

```
"Oracle"
```

```
Enter the name of sub 6:
```

```
"MYSQL"
```

```
The subjects are SQL Server,MSBI,Python,Django,Oracle,MYSQL,SQLDBA
```

```
Instead of using multiple, repetitive variables, you can use a single variable that contains a list value.
```

```
SubNames = []  
while True:  
    print('Enter the name of sub ' + str(len(SubNames) + 1) + ' (Or enter nothing to stop.):')  
    name = input()  
    if name == "":  
        break  
    SubNames = SubNames + [name] # list concatenation  
print('The Subject names are:')  
for name in SubNames:  
    print(' ' + name)
```

output:

Enter the name of sub 1 (Or enter nothing to stop.):

"SQL Server"

Enter the name of sub 2 (Or enter nothing to stop.):

"Oracle"

Enter the name of sub 3 (Or enter nothing to stop.):

"Python"

Enter the name of sub 4 (Or enter nothing to stop.):

"Django"

Enter the name of sub 5 (Or enter nothing to stop.):

"MSBI"

Enter the name of sub 6 (Or enter nothing to stop.):

"Power BI"

Enter the name of sub 7 (Or enter nothing to stop.):

The Subject names are:

"SQL Server"

"Oracle"

"Python"

"Django"

"MSBI"

"Power BI"

The benefit of using a list is that our data is now in a structure, so our program is much more flexible in processing the data than it would be with several repetitive variables.

PYTHON FUNCTIONS

A function is like a mini program within the program

Functions are first class objects in python what it means that they can be treated as just like any other variables and you can pass them as arguments to another function or return them as return statement.

They are known in most programming languages, sometimes also called subroutines or procedures. Functions are used to utilize code in more than one place in a program. The only way without functions to reuse code consists in copying the code.

A function in Python is defined by a def statement. The general syntax looks like this:

```
def function-name(Parameter list):           #function definition
    statements, i.e. the function body

function-name(actual parameters list)        #function call
```

A major purpose of functions is to group code that gets executed multiple times. Without a function defined, you would have to copy and paste this code each time, and the program would look like this:

Eg:

```
def display():
    print("Hello")
    print("Python Narayana!!!!")
display()
display()
display()
```

output:

```
Hello
Python Narayana!!!!
Hello
Python Narayana!!!!
Hello
Python Narayana!!!!
```

We can also get the result without using def function also, but we need to write the statement manually in the print, like below

```
print("Hello")
print("Python Narayana!!!!")
print("Hello")
print("Python Narayana!!!!")
print("Hello")
print("Python Narayana!!!!")
```

output

```
Hello
Python Narayana!!!!
Hello
Python Narayana!!!!
Hello
Python Narayana!!!!
```

In general we always want to avoid duplicating code, because if we decide to update the code or we need to find a bug then we have to remember to change the code everywhere we copied it.

1. Write a python function to add two numbers

```
>>> def add(x,y):  
    print(x+y)  
  
>>> add(10,20)      30
```

2. Write a python function to add three values given by user

```
n1=eval(input('Enter first value: '))  
n2=eval(input('Enter second value: '))  
n3=eval(input('Enter third value: '))
```

```
def addthreenums(a,b,c):  
    print('The sum of ',a,',',b,'and',c,'is',a+b+c)  
  
addthreenums(n1,n2,n3)
```

output:

```
Enter first value: 10  
Enter second value: 20  
Enter third value: 30  
The sum of  10 , 20 and 30 is 60
```

3. Write a python function to find maximum value of two given values

```
>>> def max(x,y):  
    if x>y:  
        print(x)  
    else:  
        print(y)  
  
>>> max(1,2)      2  
>>> max(10,2)     10
```

4. Write a python function to find max value of three values given by user

```
n1=eval(input('Enter first value: '))
n2=eval(input('Enter second value: '))

def maxoftwo(a,b):
    if a>b:
        print(a,'is more then',b)
    elif a<b:
        print(b,'is more then',a)
    else:
        print('both are equal values')

maxoftwo(n1,n2)
```

Output1:

```
Enter first value: 10
Enter second value: 13
13 is more then 10
```

Output2:

```
Enter first value: 15
Enter second value: 15
both are equal values
```

Output3:

```
Enter first value: 4
Enter second value: 7
7 is more then 4
```

5. Write a python function to find max value of three values given user

```
n1=eval(input('Enter first value: '))
n2=eval(input('Enter second value: '))
n3=eval(input('Enter third value: '))

def maxofthreevalue(a,b,c):
    if a>b and a>c:
        print(a,'is more then',b, 'and',c)
    elif b>c:
```

```
print(b,'is more then',a,'and',c)
elif a==b==c:
    print('three are same values')
else:
    print(c,'is more then',a,'and',b)

maxofthreevalue(n1,n2,n3)
```

Output1:

```
Enter first value: 15
Enter second value: 20
Enter third value: 10
20 is more then 15 and 10
```

Output2:

```
Enter first value: 1
Enter second value: 3
Enter third value: 6
6 is more then 1 and 3
```

Output3:

```
Enter first value: 12
Enter second value: 12
Enter third value: 12
three are same values
```

6. Write a python function to display n integer numbers as per user requirements?

```
def displayvalues(a):
    for i in range(1,a):
        print(i)

x=int(input('Enter your value: '))
x=x+1
displayvalues(x)
```

output:

```
● Enter your value: 11
● 1
● 2
● 3
● 4
● 5
● 6
● 7
● 8
● 9
● 10
● 11
●
```

7. write a python function to display nth table as per user requirement

```
● def table(x):
●     for i in range(1,11):
●         print(x,'*',i,'=',i*x)
●
● n=int(input("Enter any value: "))
● table(n)
●
```

output:

```
● Enter any value: 10
● 10 * 1 = 10
● 10 * 2 = 20
● 10 * 3 = 30
● 10 * 4 = 40
● 10 * 5 = 50
● 10 * 6 = 60
● 10 * 7 = 70
● 10 * 8 = 80
● 10 * 9 = 90
● 10 * 10 = 100
●
```

8. write a python function to display first all even numbers before as per user number

```
def evennums(x):
    for i in range(0,x,2):
        print(i)

n=int(input('Enter any value: '))
m=n+1
evennums(m)

Output 1:
Enter any value: 10
0
2
4
6
8
10

Output 2:
Enter any value: 11
0
2
4
6
8
10
```

9. write a python function to swap two given numbers

```
n1=int(input('Enter first value: '))
n2=int(input('Enter second value: '))
def swaping(a,b):
    temp=a
    a=b
    b=temp
```

```
print('the first value after swapping is',a)
print('the second value after swapping is',b)
```

```
swaping(n1,n2)
```

output1:

```
Enter first value: 10
Enter second value: 15
the first value after swapping is 15
the second value after swapping is 10
```

output2:

```
Enter first value: Python
Enter second value: Django
the first value after swapping is Django
the second value after swapping is Python
```

10. Write a python function to check the given number is even or odd?

```
def evenorodd(n):
    if n%2==0:
        print('The given number is',n,'. Its even number')
    else:
        print('The given number is',n,'. Its odd number')
```

```
num=int(input("Enter any number to check even or odd: "))
evenorodd(num)
```

output 1:

```
Enter any number to check even or odd: 12
The given number is 12 . Its even number
```

Output 2:

Enter any number to check even or odd: 11

The given number is 11 . Its odd number

11. Write Python function to check the given number is positive or negative number?

```
def posorneg(n):
    if n>=0:
        print('The given number is',n,'. It is positive number')
    else:
        print('The given number is',n,'. It is negative number')

num=int(input('Enter any number to check positive or negative: '))
posorneg(num)
```

Output 1:

Enter any number to check positive or negative: 10

The given number is 10 . It is positive number

Output 2:

Enter any number to check positive or negative: 0

The given number is 0 . It is positive number

Output 3:

Enter any number to check positive or negative: -12

The given number is -12 . It is negative number

12. Write a python function to display first n number of even numbers as per user requirements?

#it display all even numbers in a list format

```
x=int(input('Enter value: '))
```

```
l=[]
```

```
for i in range(2,10000,2):
```

```
    l.append(i)
```

```
    if len(l)==x:
```

```
        break
```

```
print(l)
```

output:

```
Enter value: 10
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

#it displays all even numbers one by one

```
x=int(input('Enter value: '))
```

```
l=[]
```

```
for i in range(2,10000,2):
```

```
    l.append(i)
```

```
    if len(l)==x:
```

```
        break
```

```
for j in l:
```

```
    print(j)
```

output

```
Enter value: 10
```

```
2
```

```
4
```

```
6
```

```
8
```

```
10
```

```
12
```

```
14
```

```
16
```

```
18
```

```
20
```

13. Write a python function to calculate total and avg marks of student

```
Name=input('Enter Your Name ')  
Tel=eval(input('Enter your Telugu marks: '))  
Hin=eval(input('Enter your Hindi marks: '))  
Eng=eval(input('Enter your English marks: '))  
Maths=eval(input('Enter your Maths marks: '))  
Sci=eval(input('Enter your Science marks: '))  
Soc=eval(input('Enter your Social marks: '))  
  
def marks(Tel,Hin,Eng,Maths,Sci,Soc):  
    total= Tel+Hin+Eng+Maths+Sci+Soc  
    avg=total/6  
  
    print('Your total marks are: ',total)  
    print('Your avg marks are: ',avg)  
    if Tel>=35 and Hin >=28 and Eng>35 and Maths>35 and Sci>=35 and Soc>=35:
```

```
print('Congratulation',Name,' You are passed in all exams and your total marks are',total )  
else:  
    print('Sorry',Name,'You are failed in this academic year')  
print('Thank you for using www.manabadi.com')  
  
marks(Tel,Hin,Eng,Maths,Sci,Soc)
```

output 1:

```
Enter Your Name Nani  
Enter your Telugu marks: 60  
Enter your Hindi marks: 66  
Enter your English marks: 77  
Enter your Maths marks: 67  
Enter your Science marks: 78  
Enter your Social marks: 89  
Your total marks are: 437  
Your avg marks are: 72.83333333333333  
Congratulation Nani , You are passed in all exams and your total marks are 437  
Thank you for using www.manabadi.com
```

Output 2:

```
Enter Your Name Satya  
Enter your Telugu marks: 45  
Enter your Hindi marks: 34  
Enter your English marks: 56  
Enter your Maths marks: 67  
Enter your Science marks: 66  
Enter your Social marks: 77  
Your total marks are: 345  
Your avg marks are: 57.5
```

Congratulation Satya , You are passed in all exams and your total marks are 345

Thank you for using www.manabadi.com

Output 3:

Enter Your Name Satya

Enter your Telugu marks: 67

Enter your Hindi marks: 67

Enter your English marks: 56

Enter your Maths marks: 45

Enter your Science marks: 34

Enter your Social marks: 89

Your total marks are: 358

Your avg marks are: 59.666666666666664

Sorry Satya You are failed in this academic year

Thank you for using www.manabadi.com

14.write a python program to check whether the given number is morethen or lessthen

10?

```
def checknum(x):
    if x==10:
        print('the given value is 10')
    elif x<=10:
        print('then given number is less then 10, because you entered ',x)
    else:
        print('then given number is more then 10, because you entered ',x)

a=int(input('Enter Any Value: '))
checknum(a)
```

output 1:

```
Enter Any Value: 17  
then given number is more then 10, because you entered 17
```

output 2:

```
Enter Any Value: 2  
then given number is less then 10, because you entered 2
```

output 3:

```
Enter Any Value: 10  
the given value is 10
```

15. Write a python function to check whether the given number is divisible by 5

```
def divisible(x):  
    if x%5==0:  
        print('the given number ',x,' is divisible by 5')  
    else:  
        print('the given number ',x,' is not divisible by 5')  
  
a=int(input('Enter Any Value: '))  
divisible(a)
```

output 1:

```
Enter Any Value: 10  
the given number 10 is divisible by 5
```

Output 2:

```
Enter Any Value: 23  
the given number 23 is not divisible by 5
```

16. Write a python program to accept three numbers from user and print them in ascending and decending order

```
a=int(input('Enter First value: '))
b=int(input('Enter Second value: '))
c=int(input('Enter Third value: '))
def ascordes(x,y,z):
    if x>=y and x>=z:
        if y>=z:
            print('The descending order is ',x,y,z)
            print('The ascending order is ',z,y,x)
        else:
            print('The descending order is ',x,z,y)
            print('The ascending order is ',y,z,x)
    elif y>=x and y>=z:
        if x>=z:
            print('Then descending order is ',y,x,z)
            print('Then ascending order is ',z,x,y)
        else:
            print('Then descending order is ',y,z,x)
            print('Then ascending order is ',x,z,y)
    elif z>=x and z>=y:
        if x>=y:
            print('Then descending order is ',z,x,y)
            print('Then ascending order is ',y,x,z)
        else:
            print('Then descending order is ',z,y,x)
            print('Then ascending order is ',x,y,z)
ascordes(a,b,c)
```

Output 1:

```
Enter First value: 100
Enter Secondvalue: 200
Enter Third value: 150
Then descending order is 200 150 100
Then ascending order is 100 150 200
```

Output 2:

```
Enter First value: 200
Enter Secondvalue: 150
Enter Third value: 100
The descending order is 200 150 100
The ascending order is 100 150 200
```

Output 3:

```
Enter First value: 150
Enter Secondvalue: 100
Enter Third value: 200
Then descending order is 200 150 100
Then ascending order is 100 150 200
```

17.write a python function to display same message n number of times as per the user**requirements**

```
def display(m,n):
    for i in range(n):
        print(m)
msg=input('Enter your messge: ')
num=int(input('Enter your number: '))
display(msg,num)
```

output:

```
● Enter your messge: Hello, Python Narayana
```

```
● Enter your number: 4
```

```
● Hello, Python Narayana
```

18. Write a python program to accept user's marital status, gender and age to check if he/she is eligible for marriage or not.

```
● def marriageelible(name,ms,gen,age):  
●     ms=ms.replace(ms[1:],").lower()  
●     gen=gen.replace(gen[1:],").lower()  
●  
●     if ms=='m':  
●         print('Hello',name,', you are not allowed to marry again')  
●     elif ms=='u':  
●         if gen=='m':  
●             if age>=21:  
●                 print('Hello',name,', Congratulations.., You can marry now..')  
●             else:  
●                 print('Hello',name,', You need to wait ',21-age,'years to get marry')  
●         elif gen=='f':  
●             if age>=18:  
●                 print('Hello',name,', Congratulations.., You can marry now..')  
●             else:  
●                 print('Hello',name,', You need to wait ',18-age,'years to get marry')  
●         else:  
●             print('Hello',name,', You entered', name,', This is invalid gender')  
●     else:  
●         print('Hello',name,'You entered',ms,',This is invalid marital status')  
●  
● name=input('Enter your name: ')  
● ms=input('Enter your marital status (married or unmarried): ')  
● gen=input('Enter you gender (male or female): ')
```

```
age=int(input('Enter your age: '))  
marriageelible(name,ms,gen,age)
```

Output 1:

```
Enter your name: Siva  
Enter your marital status (married or unmarried): married  
Enter you gender (male or female): male  
Enter your age: 30  
Hello Siva , you are not allowed to marry again
```

Output 2:

```
Enter your name: Siva  
Enter your marital status (married or unmarried): unmarried  
Enter you gender (male or female): male  
Enter your age: 15  
Hello Siva , You need to wait 6 years to get marry
```

Output 3:

```
Enter your name: Siva  
Enter your marital status (married or unmarried): unmarried  
Enter you gender (male or female): Male  
Enter your age: 23  
Hello Siva , Congratulations.., You can marry now..
```

Output 4:

```
Enter your name: Komali  
Enter your marital status (married or unmarried): Unmarried  
Enter you gender (male or female): female  
Enter your age: 17  
Hello Komali , You need to wait 1 years to get marry
```

Output 5:

```
Enter your name: Komali  
Enter your marital status (married or unmarried): married  
Enter you gender (male or female): female  
Enter your age: 23  
Hello Komali , you are not allowed to marry again
```

Output 6:

```
Enter your name: Komali  
Enter your marital status (married or unmarried): unmarried  
Enter you gender (male or female): female  
Enter your age: 22  
Hello Komali , Congratulations.., You can marry now..
```

19. Write a python program to reverse the given number

```
def reverseorder(x):  
    x=str(x)  
    x=".join(reversed(x))  
    x=int(x)  
    print('the given number is',n,', after reversing the number is ',x)  
n=int(input('Enter any number to get reverse order: '))  
reverseorder(n)
```

output:

```
Enter any number to get reverse order: 3764  
the given number is 3764 , after reversing the number is 4673
```

20. Write a python function to find sum of its digits

```
def sumofnums(x):  
    y=0  
    for i in range(x):  
        y=y+i  
    print('The given number is',n,', the sum of the digits is',y)  
  
n=int(input("Enter any number: "))  
m=n+1  
sumofnums(m)
```

output:

```
Enter any number: 10  
The given number is 10 , the sum of the digits is 55
```

21. Write a python program to display the output like below,

```
* * *  
* * *  
* * *  
def stars(x):  
    for i in range(x):  
        for j in range(x):  
            print('*', end = ' ')  
        print('\n')
```

```
n=int(input('Enter any number: '))
```

```
stars(n)
```

Output 1:

```
Enter any number: 3
```

```
* * *
```

```
* * *
```

```
* * *
```

Output 2:

```
Enter any number: 5
```

```
* * * * *
```

```
* * * * *
```

```
* * * * *
```

```
* * * * *
```

```
* * * * *
```

22. Write a python program to display output like below,

```
*
```



```
* *
```



```
* * *
```



```
* * * *
```



```
* * * * *
```

```
def stars1(n):  
    for i in range(n):  
        for j in range(i):  
            print('*',end=' ')  
        print('\n')
```

```
n=int(input("Enter any number: "))
```

```
n=n+1
```

```
stars1(n)
```

Output:

Enter any number: 5

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

23. Write python function to display the following output:

```
* * * * *  
* * * * *  
* * * * *  
* * * *  
* * *  
* *  
*  
  
def stars2(n):  
    for i in range(n,0,-1):  
        for j in range(i):  
            print('*',end=' ')  
        print('\n')  
  
n=int(input('Enter any number: '))  
n=n+1  
stars2(n)
```

output:

Enter any number: 5

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

24. Write python function to display the following output:

```
* * * * *
* * * *
* * *
* *
*
def stars3(n):
    for i in range(n):
        for j in range(i):
            print(' ',end=' ')
        for k in range(i,n):
            print('*' ,end=' ')
        print('\n')

n=int(input('Enter any number: '))
n=n+1
stars3(n)
```

output:

```
Enter any number: 5
```

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

Other Functions

Eg 1:

```
def printing( str ):  
    print(str)  
    return;  
  
printing("Hello Python Narayana, I'm the first call to the function definition!")  
printing("Hey Python Narayana, I'm the second call to the same function definition")
```

output

```
Hello Python Narayana, I'm the first call to the function definition!  
Hey Python Narayana, I'm the second call to the same function definition
```

Eg2:

```
def naming( name, sep, loc ):  
    print(name,sep,loc)  
    return;  
  
naming( name = "Python Narayana",sep=',', loc="Guntur")  
naming( name = "Nagaraju",sep=',',loc="Hyderabad")
```

Output:

```
Python Narayana , Guntur  
Nagaraju , Hyderabad
```

Eg3:

```
def empdetails(empname, emprole):  
    print ("Emp Name: ", empname)  
    print ("Emp Role: ", emprole)  
    return;  
  
print("Calling in proper sequence")  
empdetails(empname = "Python Narayana",emprole = "Trainer" )
```

```
print('\n')
print("Calling in opposite sequence")
empdetails(emprole = "Manager",empname = "Nagaraju")
```

output:

```
Calling in proper sequence
Emp Name: Python Narayana
Emp Role: Trainer

Calling in opposite sequence
Emp Name: Nagaraju
Emp Role: Manager
```

Eg 4:

```
def python():
    print("Hello Python!")
    print("You are the easiest language amoung all languages")
    print("You are too powerful language.")
    print("Everyone can understand you.")
    print("Thank you very much..")

def django():
    print("Hello Django!")
    print("It is very happy to work with you!")
    print("You are developed by using 100% pure python language")
    print("You are versatile and scalable framework")
    print("All Python developers must need you")
```

```
def sql():
    print("Hello Sql")
    print("These is no project with you")
    print("You are very simple and easy to learn and work")
    print("You play very keyrole in the interviews")
    print("Thank you very much")
```

```
def main():
    python()
    django()
    sql()
```

```
main()
```

output:

```
Hello Python!
You are the easiest language amoung all languages
You are too powerful language.
Everyone can understand you.
Thank you very much..
```

```
Hello Django!
It is very happy to work with you!
You are developed by using 100% pure python language
You are versatile and scalable framework
All Python developers must need you
```

```
Hello Sql
These is no project with you
You are very simple and easy to learn and work
```

You play very keyrole in the interviews

Thank you very much

Eg 5:

```
def sumProblem(x, y):
    sum = x + y
    sentence = 'The sum of {} and {} is {}'.format(x, y, sum)
    print(sentence)

def main():
    sumProblem(2, 3)
    sumProblem(1234567890123, 535790269358)
    a = int(input("Enter an integer: "))
    b = int(input("Enter another integer: "))
    sumProblem(a, b)

main()
```

output

The sum of 2 and 3 is 5.

The sum of 1234567890123 and 535790269358 is 1770358159481.

Enter an integer: 10

Enter another integer: 20

The sum of 10 and 20 is 30.

Eg 6:

```
def sqrs(x):
    return x*x

print(sqrs(2))
print(sqrs(3) + sqrs(4))
print(sqrs(5) + sqrs(6) +sqrs(7))
print(sqrs(5) + sqrs(3) - sqrs(5))
```

output

```
4
25
110
9
```

Eg 7:

```
def main():
    x = 'Python'
    y = 'Developer'
    display(x,y)

def display(x,y):
    print(x,y)

main()
```

output:

```
Python Developer
```

Default arguments

Default parameters assume a default value if a value is not provided by the actual parameters in the function call.

```
def display_message(times,message):  
    for i in range(times):  
        print(message)  
  
display_message(4,'Python Narayana')
```

output

```
Python Narayana  
Python Narayana  
Python Narayana  
Python Narayana
```

So we can set some default values to the formal parameters in the function definition. Those are called default arguments. So that if we don't specify actual parameters in the function call then interpreter takes formal parameters values and continue the operation.

```
def display_message(times=5,message="This is Python time"):  
    for i in range(times):  
        print(message)  
  
display_message()  
  
output:  
  
This is Python time  
This is Python time  
This is Python time  
This is Python time  
This is Python time
```

In the above function we didn't pass the actual parameters in the function call so interpreter has taken the default values and continued the operation.

If we pass the actual values when we have default values already in the function definition, then interpreter takes actual values and continue the operation.

```
def display_message(times=5,message="This is Python time"):
    for i in range(times):
        print(message)

display_message(2,'Python Narayana')

Output

Python Narayana
Python Narayana
```

Generally the first actual parameter will map to the first formal parameter and second actual parameters will map to the second formal parameters and so on...

If we give those mappings in the reverse way then it will throw error like,

```
def display_message(times=5,message="This is Python time"):
    for i in range(times):
        print(message)

>>> display_message('Narayana',3)

Output:      TypeError: 'str' object cannot be interpreted as an integer
```

In the above case, we can specify the parameters names while passing the value in the function call. If we specify those names in the function call then those are called **keyword arguments**

Keyword argument:

A keyword argument in a function call identifies the argument by a formal parameter name.

The python interpreter is then able to use these keywords to connect the values with formal parameters.

```
def display_message(times=5,message="This is Python time"):
    for i in range(times):
        print(message)

display_message(message='Python Narayana',times=2)
```

Output

```
Python Narayana  
Python Narayana
```

Eg:

```
def details( name, cell ):  
    "This prints a passed info into this function"  
    print("Name: ", name)  
    print("Cell: ", cell)  
    return  
  
details( cell=9999999999, name="Narayana" )
```

output:

```
Name: Narayana  
Cell: 9999999999
```

Scope of the variables:

All variables in a program may not be accessible at all locations in that program, they are accessible depends on where you have declared the variables.

Parameters and variables that are assigned in a called function are said to exist in that function's local scope. Variables that are assigned outside all functions are said to exist in the global scope. A variable that exists in a local scope is called a local variable, while a variable that exists in the global scope is called a global variable. A variable must be one or the other; it cannot be both local and global.

Think of a scope as a container for variables. When a scope is destroyed, all the values stored in the scope's variables are forgotten. There is only one global scope, and it is created when our program begins. When our program terminates, the global scope is destroyed, and all its variables are forgotten. Otherwise, the next time we ran our program, the variables would remember their values from the last time we ran it.

A local scope is created whenever a function is called. Any variables assigned in this function exist within the local scope. When the function returns, the local scope is destroyed, and these variables are forgotten. The next time we call this function, the local variables will not remember the values stored in them from the last time the function was called.

The reason Python has different scopes instead of just making everything a global variable is so that when variables are modified by the code in a particular call to a function, the function interacts with the rest of the program only through its parameters and the return value. This narrows down the list code lines that may be causing a bug. If our program contained nothing but global variables and had a bug because of a variable being set to a bad value, then it would be hard to track down where this bad value was set. It could have been set from anywhere in the program—and our program could be hundreds or thousands of lines long! But if the bug

is because of a local variable with a bad value, we know that only the code in that one function could have set it incorrectly.

While using global variables in small programs is fine, it is a bad habit to rely on global variables as our programs get larger and larger

Local Variables Cannot Be Used in the Global Scope

```
def display():
    empno=1001      #local variable
    ename='Nani'     #local variable
    print("Emp number is",empno," and name is ",ename)

display()
print("Emp number is",empno)      #using local variable in global scope
```

output:

```
Emp number is 1001 and name is Nani
Traceback (most recent call last):
  File "C:/Users/Narayana/AppData/Local/Programs/Python/Python36-32/ddddddd.py", line 7, in <module>
    print("Emp number is",empno)
NameError: name 'empno' is not defined
```

Global Variables Can Be Read from a Local Scope

```
empno=1001          #global variable  
def display():  
    ename='Nani'      #local variable  
    print("Emp number is",empno," and name is ",ename)      #using global variable in the local scope  
  
display()  
print("Emp number is",empno)
```

output:

```
Emp number is 1001 and name is Nani  
Emp number is 1001
```

Python supports two types of variables,

1. Local variables
2. Global variables

Local variables:

Local variables are variables which are declared outside of a function.

Let's an example with local and global variables scopes,

Here loc_var is a local variable and glo_var is a global variable.

```
glo_var=0  
  
def cal_product_of_two_values(num1,num2):  
    loc_var = num1*num2  
    print("The result of LOCAL Variable is: ",loc_var)  
    return loc_var
```

```
print("The result of GLOBAL Variable before calling the function is: ",glo_var)
glo_var=cal_product_of_two_values(3,5)
print("The result of GLOBAL Variable after calling the function is: ",glo_var)
```

Output:

```
The result of GLOBAL Variable before calling the function is: 0
The result of LOCAL Variable is: 15
The result of GLOBAL Variable after calling the function is: 15
```

We can also use global variable in the functions like,

```
glo_var=0
def cal_product_of_two_values(num1,num2):
    loc_var = num1*num2
    print("The result f LOCAL Variable is: ",loc_var)
    print("The result of GLOBAL Variable in the function is: ",glo_var)
    return loc_var

print("The result of GLOBAL Variable before calling the function is: ",glo_var)
glo_var=cal_product_of_two_values(3,5)
print("The result of GLOBAL Variable after calling the function is: ",glo_var)
```

output

```
The result of GLOBAL Variable before calling the function is: 0
The result f LOCAL Variable is: 15
The result of GLOBAL Variable in the function is: 0
The result of GLOBAL Variable after calling the function is: 15
```

We can't use local variable outside the function where it is declared, if we use the interpreter will throw error,

```
glo_var=0
def cal_product_of_two_values(num1,num2):
    loc_var = num1*num2
    print("The result f LOCAL Variable is: ",loc_var)
    return loc_var

print("The result of GLOBAL Variable before calling the function is: ",glo_var)
glo_var=cal_product_of_two_values(3,5)
```

```
print("The result of GLOBAL Variable after calling the function is: ",glo_var)
print("The result of LOCAL Variable after function call is: ",loc_var)
```

Output:

```
The result of GLOBAL Variable before calling the function is: 0
The result f LOCAL Variable is: 15
The result of GLOBAL Variable after calling the function is: 15
NameError: name 'loc_var' is not defined
```

Variable length arguments:

*args and **kwargs are used in function definitions to pass a variable number of arguments to a function. The single asterisk form (*args) is used to pass a *non-keyworded*, variable-length argument list, and the double asterisk form is used to pass a *keyworded*, variable-length argument list. Here is an example of how to use the non-keyworded form. This example passes one formal (positional) argument, and two more variable length arguments.

The general function contains a formal (positional) argument, non-keyworded argument and keyworded argument.

The syntax of a function is

```
some_func(fargs,*args,**kwargs)
```

Variable length non-keyworded arguments,

let's an example of using one formal and multiple variable length non-keyworded arguments,

Eg1:

```
def multi_args(a,*x):
    print("Formal arg is:",a)
    for i in x:
        print("The non_keyworded arg is:",i)
    return

multi_args(10,20,'Narayana','Python')
```

output:

```
Formal arg is: 10
The non_kwarged arg is: 20
The non_kwarged arg is: Narayana
The non_kwarged arg is: Python
```

Eg2:

```
def varLenArgFunc(*varvallist ):
    print ("The Output is: ")
    for varval in varvallist:
        print (varval)
    return;
print("Calling with single value")
varLenArgFunc(55)
print("Calling with multiple values")
varLenArgFunc(50,'Django',60,'Narayana',70,'Python',80)
```

Output:

Calling with single value

The Output is:

55

Calling with multiple values

The Output is:

50

Django

60

Narayana

70

Python

80

Using *args in calling function

Eg1:

```
def multi_args(a,*x):
    print("Formal arg is:",a)
    for i in x:
        print("The non_keyworded arg is:",i)
    return

tup1=(100,'Py','Sai')      #creating a tuple with multiple args

multi_args(10,*tup1)       #using tuple in the function call as nonkeyworded arg.
```

Output:

```
Formal arg is: 10
The non_keyworded arg is: 100
The non_keyworded arg is: Py
The non_keyworded arg is: Sai
```

Eg2:

```
def multi(a,x,y,z):
    print("Formal arg is:",a)
    print("nonkeyworded arg is:",x)
    print("nonkeyworded arg is:",y)
    print("nonkeyworded arg is:",z)

tup1=(100,'Py','Sai')

multi(10,*tup1)
```

Variable length keyworded arguments:

Let's an example of using one formal and multiple variable length keyworded arguments,

```
def mul_kwargs(a,**x):
    print("The formal arg is: ",a)
    for i in x:
        print("Another keyworded arg is: %s: %s" % (i,x[i]))
mul_kwargs(a=10,b=20,c=30)
```

Output:

```
The formal arg is: 10
Another keyworded arg is: b: 20
Another keyworded arg is: c: 30
```

Using **kwarg in the function call

```
def mul_kwargs(a,**x):
    print("The formal arg is: ",a)
    for i in x:
        print("Another keyworded arg is: %s: %s" % (i,x[i]))

dict={"arg1":1,"arg2":2,"arg3":"Sai"}

mul_kwargs(a=10,**dict)

output:

The formal arg is: 10
Another keyworded arg is: arg1: 1
Another keyworded arg is: arg2: 2
Another keyworded arg is: arg3: Sai
```

Some examples

Eg1:

```
def addingval(a,*b):
    print(a,b)

addingval(10,20,30,'d',40)
output: 10 (20, 30, 'd', 40)
```

Eg2:

```
def addingval(a,**b):
    print(a,b)

addingval(a=10,b=20,c=30,d=40)
output: 10 {'b': 20, 'c': 30, 'd': 40}
```

Eg3:

```
def av(a,b=10,*c,**d):
    print(a,b,c,d)

av(1,2,'a1',3,'f',4,5)
output: 1 2 ('a1', 3, 'f', 4, 5) {}
```

Eg4:

```
def av(a,b=10,*c,**d):
    print(a,b,c,d)

av(1,2,a1=3,c='f',d=4,r=5)
output: 1 2 () {'a1': 3, 'c': 'f', 'd': 4, 'r': 5}
```

Eg5:

```
def av(a,b=10,*c,**d):
    print(a,b,c,d)

av(1,2,3,'a','f',True,x=10,y=20)
output: 1 2 (3, 'a', 'f', True) {'x': 10, 'y': 20}
```

Example for using all kinds of arguments in single function like formal argument, *args and **kwargs.

```
def mul_kwargs(a,x,y,z,arg1,arg2,arg3):
    print("a :",a)
    print("x :",x)
    print("y :",y)
    print("z :",z)
    print("arg1:",arg1)
    print("arg2:",arg2)
    print("arg3:",arg3)

dict={"arg1":"sai", "arg2":100,"arg3":"Narayana"}
tup=(10,20,30)
mul_kwargs(10,*tup,**dict)
```

Output:

```
a : 10
x : 10
y : 20
z : 30
arg1: sai
arg2: 100
arg3: Narayana
```

Arguments packing and unpacking

When we need to call a function definition with function call which is having a **list** with size 3. So if we pass simply then it will not work. Let's check once,

```
def pack_var(arg1,arg2,arg3):
    print(arg1,arg2,arg3)

my_list=[100,200,300]          #creating list with size 3 elements
pack_var(my_list)              #calling a func with my_list

output: TypeError: pack_var() missing 2 required positional arguments: 'arg2' and 'arg3'.
```

In this case we should use unpack the my_list. This is called **argument unpacking**

```
def pack_var(arg1,arg2,arg3):
    print(arg1,arg2,arg3)

my_list=[100,200,300]

pack_var(*my_list)

Output: 100 200 300
```

We can also pack the arguments, like,

When we don't know how many arguments need to be passed to a python function, we can pack all arguments in a tuple. This is called **argument packing**

```
def sum_of_args(*args):
    sum=0
    for i in range(0,len(args)):
        sum=sum+args[i]
    return sum

print(sum_of_args(2,20,30))

Output: 52

or

print(sum_of_args(20,20,30))

Output: 70
```

Eg:

```
def cheese_and_crackers(cheese_count, boxes_of_crackers):
    print ("You have %d cheeses!" % cheese_count)
    print ("You have %d boxes of crackers!" % boxes_of_crackers)
    print ("Man that's enough for a party!")
    print ("Get a blanket.\n")
```

```
print ("We can just give the function numbers directly:")
cheese_and_crackers(20, 30)

output:
We can just give the function numbers directly:
You have 20 cheeses!
You have 30 boxes of crackers!
Man that's enough for a party!
Get a blanket.
```

Python Format function:

Some basics about formatting

```
print("This is {} and working as {}".format('Narayana','Python Developer'))
output: This is Narayana and working as Python Developer

print("This is {0} and working as {1}".format('Narayana','Python Developer'))
output: This is Narayana and working as Python Developer

print("This is {1} and working as {0}".format('Narayana','Python Developer'))
output: This is Python Developer and working as Narayana

print("This is {1} and working as {1}".format('Narayana','Python Developer'))
output: This is Python Developer and working as Python Developer

print("We eat {} time per {}".format(3,"day"))
output: We eat 3 time per day
```

Display squares and cubes for given numbers:**Traditional way**

```
for i in range(1,11):  
    print(i, " ",i*i," ",i*i*i)
```

Output

```
1  1  1  
2  4  8  
3  9  27  
4  16 64  
5  25 125  
6  36 216  
7  49 343  
8  64 512  
9  81 729  
10 100 1000
```

By using format function

```
>>> for i in range(1,11):  
    print("{:1d} {:6d} {:6d}".format(i,i*i,i*i*i))
```

Output:

```
1  1  1  
2  4  8  
3  9  27  
4  16 64  
5  25 125  
6  36 216  
7  49 343  
8  64 512  
9  81 729  
10 100 1000
```

Format function with dictionary

```
persons={'name':'Sai', "Age":28}

#Traditional way of using keys in the sentence
print( "My name is " +persons['name']+ " and i am "+str(persons['Age'])+" years old")
print( "My name is " ,persons['name'], " and i am ",str(persons['Age']), " years old")

#by using format function also we can use value with corresponding keys in the sentence

print("My name is {} and i am {} years old".format(persons['name'],persons['Age']))
print("My name is {0} and i am {1} years old".format(persons['name'],persons['Age']))
print("My name is {0[name]} and i am {1[Age]} years old".format(persons,persons))
print("My name is {0[name]} and i am {0[Age]} years old".format(persons))
```

Output:

```
My name is Sai and i am 28 years old
My name is Sai and i am 28 years old
My name is Sai and i am 28 years old
My name is Sai and i am 28 years old
My name is Sai and i am 28 years old
My name is Sai and i am 28 years old
```

Format function with list

```
lst=["Narayana","Python",3]
print("My Name is {0[0]} and i have been giving training on {0[1]} for {0[2]} years ".format(lst))
```

Output: My Name is Narayana and i have been giving training on Python for 3 years

Adding formatting to the place holder

```
for i in range(1,5):
    msg="The current value is {:02} ".format(i)
    print(msg)
```

Output:

```
The current value is 01
The current value is 02
```

The current value is 03
The current value is 04

Format function with tuple

```
lst=("Sai","Python",2)  
print("My Name is {0[0]} and i have been giving training on {0[1]} for {0[2]} years ".format(lst))
```

Output: My Name is Sai and i have been giving training on Python for 2 years

Format dates #about dates we will discuss in modules concept

Eg1:

```
import datetime  
  
my_date = datetime.datetime(2017,10,16,23,20,44)  
  
sentence="{:%B %d, %Y}".format(my_date)  
  
print(sentence)
```

Output: October 16, 2017

Eg2:

```
import datetime  
  
my_date = datetime.datetime(2017,10,16,23,20,44)  
  
print("{0:%B %d, %Y} fell on {0:%A} and {0:%w} day of this week and {0:%d} day of this month and {0:%j} day of the year".format(my_date))
```

Output

Output: October 16, 2017 fell on Monday and 1 day of this week and 16 day of this month and 289 day of the year

Some other examples

Eg1:

```
age = input("How old are you? ")
height = input("How tall are you? ")
weight = input("How much do you weigh? ")
print ("So, you're {} years old, {} feet tall and {} kgs weight.".format(age, height, weight))
```

output:

```
How old are you? 27
How tall are you? 5.9
How much do you weigh? 75
So, you're 27 years old, 5.9 feet tall and 75 kgs weight.
```

Eg2:

```
def add(a, b):
    print ("ADDING {} + {}".format(a, b))
    return a + b

def subtract(a, b):
    print ("SUBTRACTING {} - {}".format (a, b))
    return a - b

def multiply(a, b):
    print ("MULTIPLYING {} * {}".format (a, b))
    return a * b

def divide(a, b):
    print ("DIVIDING {} / {}".format(a, b))
    return a / b

print ("Let's do some math with just functions!")

age = add(30, 5)
height = subtract(78, 4)
weight = multiply(90, 2)
iq = divide(100, 2)

print ("Age: {}, Height: {}, Weight: {}, IQ: {}".format (age, height, weight, iq))
```

Output:

Let's do some math with just functions!
ADDING 30 + 5
SUBTRACTING 78 - 4
MULTIPLYING 90 * 2
DIVIDING 100 / 2
Age: 35, Height: 74, Weight: 180, IQ: 50.0

Write the outputs for following functions:**Eg1:**

```
def method2(c=10,*y):  
    print(y)  
method2(1)
```

Output:**Eg2:**

```
def method3(y,x=10):  
    print(y)  
    print(x)  
  
method3(1)
```

Output:

Eg3:

```
def method3(y):
    print(y)
    print(x)

method3(1)
```

Output:**Eg4:**

```
def method1(a,b=10,*c):
    print(a)
    print()
    print(b)
    print(c)

method1(5,4,'Py')
```

Output:**Eg5:**

```
def method1(a,b=10,*c,**d):
    print(a)
    print()
    print(b)
    print(c)

method1(1,2 )
```

Output:

Eg6:

```
def method1(a,**d):
    print(a)
    print()
method1('Py')
```

Output:**Eg7:**

```
def method1(**d):
    print(d)
c={'a':10,'b':20}
method1(c)
```

Output:**Eg8:**

```
def method1(**d):
    print(d)
c={'a':10,'b':20}
method1(**c)
```

Output:

Eg9:

```
def method1(x,**d):
    print(x)
    print(d)

c={'a':10,'b':20}
method1('Py',**c)
```

Output:**Eg10:**

```
def method1(*x,**d):
    print(x)
    print(d)

b=[1,2,3]
c={'a':10,'b':20}
method1(b,**c)
```

Output:**Eg11:**

```
def method1(*x,**d):
    for i in x:
        print(i)
    print(d)

b=[1,2,3]
c={'a':10,'b':20}
method1(*b,**c)
```

Output:

Eg12:

```
def method1(*x,**d):
    for i in x:
        print(i)
    print(d)

c={'a':10,'b':20}
method1(**c)
```

Output:

Eg13:

```
def method1(a,b=10,*c,**d):
    print(a)
    print(b)
    print(c)
    print(d)

lst=[1,2,3]
method1(lst)
```

Output:

Eg14:

```
def method1(a,b=10,*c,**d):
    print(a)
    print(b)
    print(c)
    print(d)

lst=[1,2,3]
method1(*lst)
```

Output:

LAMBDA FUNCTIONS

1. Python supports the creation of anonymous functions (i.e. functions that are not bound to a name) at runtime, using a construct called "lambda".
2. This is not exactly the same as lambda in functional programming languages, but it is a very powerful concept that's well integrated into Python
3. These functions are throw-away functions, i.e. they are just needed where they have been created.
4. We use a lambda function when we require a nameless function for short time.
5. In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments).
6. Actually, we don't absolutely need lambda; we could get along without it. But there are certain situations where it makes writing code a bit easier, and the written code a bit cleaner. What kind of situations? ... Situations in which (a) the function is fairly simple, and (b) it is going to be used only once.
7. Suppose we need to create a function that is going to be used only once — called from *only one* place in our application. Then we don't need to give the function a name. It can be “anonymous”. And we can just define it right in the place where we want to use it. That's where lambda is useful.
8. Lambda functions are used along with built-in functions like filter(), map() and reduce().

“Lambda is a tool for building anonymous functions.”

The general syntax of a lambda function is quite simple:

`lambda argument_list: expression`

The argument list consists of a comma separated list of arguments and the expression is an arithmetic expression using these arguments.

Eg:

`a= lambda x,y: x if x>y else y`

`print(a(20,5))`

`20`

In the above script `lambda x,y: x if x>y else y` is lambda function whereas x,y are the arguments and `x if x>y else y` is expression.

The above function has no name and it returns a function object which is assigned to a identifier ‘a’.

Generally we use `def` keyword to create a function definition but in this case we don’t use `def` keyword instead we use `lambda` keyword.

We can use either `def` or `lambda` methods to create a function.

Write a function to add two numbers?

By using def

```
def add_nums(x,y):
    return x+y
print(add_nums(1,2))      3
```

by using Lambda

```
a=lambda x,y:x+y
print(a(1,2))      3
```

Write a function to find maximum value of given two numbers

By using def

```
def max_nums(x,y):
    if x>y:
        return x
    else:
        return y
```

Output: print(max_nums(20,5)) 20

By using lambda

```
a= lambda x,y: x if x>y else y
output: print(a(20,5)) 20
```

Write a function to calculate square for given number

By using def

```
def square_val(x):
    return x*x
output: print(square_val(10)) 100
```

By using lambda

```
a = lambda x: x*x
output: print(a(10)) 100
```

Lambda functions are mainly used in combination with the functions **filter()**, **map()** and **reduce()**.

Filter():

Filter function mainly takes two arguments, first one is a function and second one is list of arguments.

```
r=filter(func,seq)
```

This function calls all the items from the existing sequence and a new sequence is returned which contains the elements that are evaluated to True.

1. Write a function to return only odd numbers from the existing list which contains values from 1 to 20?

```
lst=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

```
odd_nums = list(filter(lambda x: (x%2==1),lst))
```

```
print(odd_nums)
```

Output: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

2. Write a python code to separate positive values from the given list which contains both positive and negative numbers

```
lst=range(-10,10)
```

```
print(lst) [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
a= list((filter(lambda x:x>0,lst)))
```

```
print(a)
```

Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]

3. Write a python script to get intersection elements from both lists

```
a=[1,2,3,4,5,6]
```

```
b=[4,10,11,1]
```

```
lst=list(filter(lambda x:x in a,b))
```

```
print(lst) [4, 1]
```

4. Write a python script to get all numbers which are divisible by 3 from the given list

```
lst = [12, 8, 14, 22, 11, 34, 19, 18, 33]
x=list(filter(lambda x: x % 3 == 0, lst))
print (x)
or
print (list(filter(lambda x: x % 3 == 0, [12, 8, 14, 22, 11, 34, 19, 18, 33])))
output:
[12, 18, 33]
```

Map():

The map function contains two arguments like,

```
r=map (func,seq)
```

The first argument func is the name of a function and the second a sequence (e.g. a list) seq. map() applies the function func to all the elements of the sequence seq. It returns a new list with the elements changed by func

1. Write a function to add elements from different sets

```
a = [1,2,3,4],
b = [17,12,11,10]
c = [-1,-4,5,9]

list(map(lambda x,y:x+y, a,b))          [18, 14, 14, 14]
list(map(lambda x,y,z:x+y+z, a,b,c))    [17, 10, 19, 23]
```

2. Write a python script to find squares for all elements in the given list

```
a=[2,4,5,6]
a=map(lambda x:x*x,a)
print(a)      [4, 16, 25, 36]
```

3. Write a python script to add 10 to each element in the given list

```
lst= [21, 8, 12, 2, 7, 4, 19, 22, 11]
a=list(map(lambda x: x + 10, lst))
print(a)

or

lst= [21, 8, 12, 2, 7, 4, 19, 22, 11]
print(list(map(lambda x: x + 10, lst)))

output

[31, 18, 22, 12, 17, 14, 29, 32, 21]
```

4. Write a python script to display length of each word in the given example?

```
sen='This is Python Narayana and I am giving training on both Python and Django'
sen=sen.split(' ')
a=list(map(lambda x:(x,len(x)),sen))
for i in a:
    print(i)

output:

('This', 4)
('is', 2)
```

Narayana

PYTHON

Narayana

```
('Python', 6)
('Narayana', 8)
('and', 3)
('I', 1)
('am', 2)
('giving', 6)
('training', 8)
('on', 2)
('both', 4)
('Python', 6)
('and', 3)
('Django', 6)
```

Eg2:

```
for i in list(map(lambda x:(x,len(x)), 'This is Python Narayana and I am giving training on Python'.split(' '))):
    print(i)

output:
('This', 4)
('is', 2)
('Python', 6)
('Narayana', 8)
('and', 3)
('I', 1)
('am', 2)
('giving', 6)
('training', 8)
('on', 2)
```

```
('Python', 6)
```

Since it's a built-in, **map** is always available and always works the same way.

```
from operator import add
```

```
a=[1,2,3,4]
```

```
b=(1,2,3,4)
```

```
x=map(add,a,b)
```

```
print(x)[2, 4, 6, 8]
```

If function is **None**, the **identity** function is assumed; if there are multiple arguments, **map()** returns a list consisting of **tuples** containing the corresponding items from all iterables (a kind of transpose operation).

```
m = [1,2,3]
```

```
n = [1,4,9]
```

```
new_tuple = map(None, m, n)
```

```
print(new_tuple)
```

```
output: [(1, 1), (2, 4), (3, 9)]
```

We can still use lambda as a function and list of functions as sequence, like

```
def square_values(x):  
    return (x**2)
```

```
def cube_values(x):  
    return (x**3)
```

```
funcs = [square_values, cube_values]
```

```
for r in range(5):
```

```
    value = map(lambda x: x(r), funcs)
```

```
    print value
```

```
output:
```

```
[0, 0]
```

```
[1, 1]
```

```
[4, 8]
```

```
[9, 27]
```

```
[16, 64]
```

Reduce()

Reduce function contains two arguments like,

```
x=reduce(func,seq)
```

The function continually applies the function func() to the sequence seq. It returns a single value.

Syn:

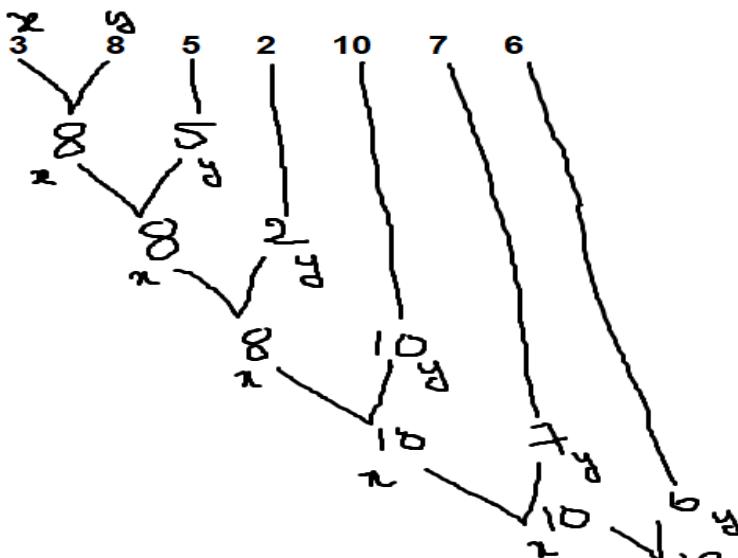
If seq = [s₁, s₂, s₃, ... , s_n], calling reduce(func, seq) works like this:

- Initially the first two elements of seq will be applied to func, i.e. func(s₁,s₂) The list on which reduce() works looks now like this: [func(s₁, s₂), s₃, ... , s_n]
- In the next step func will be applied on the previous result and the third element of the list, i.e. func(func(s₁, s₂),s₃) The list looks like this now: [func(func(s₁, s₂),s₃), ... , s_n]
- Continue like this until just one element is left and return this element as the result of reduce()

1. Write a function to find maximum number from given list by using lambda

```
>>> lst=[3,8,5,2,10,7,6]
>>> maxval=reduce(lambda x,y:x if x>y else y,lst)
>>> print(maxval)      10
```

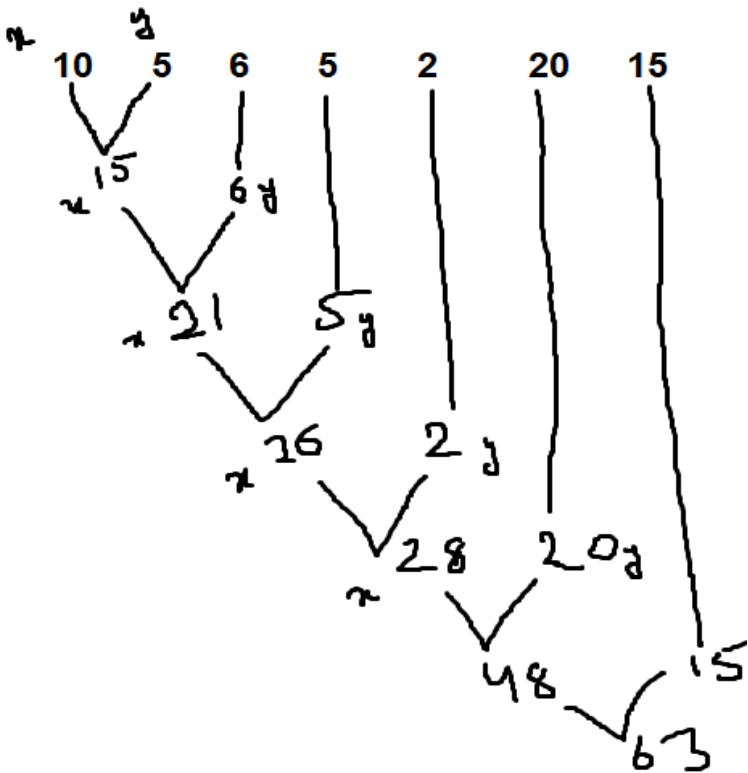
Working Process:



2. Write a python script to add all the elements in the given list

```
>>> lst=[10,5,6,5,2,20,15]  
>>> print(reduce(lambda x,y:x+y,lst))  
63
```

Working Process



3. Write a python script to add all even numbers before 100

```
a= reduce(lambda x,y: x+y, range(0,101,2))  
print(a)      2550
```

Object Oriented Programming

1. Introduction to OOP
2. Class and Object
3. Accessing Variables and Methods from class
4. Self-keyword
5. Docstring
6. Class or static variables
7. Constructor
8. Instance or non-static variables
9. Data hiding
10. Data abstraction
11. Encapsulation
12. Tightly encapsulation
13. Inheritance
 - a. Single level inheritance
 - b. Multi-level inheritance
 - c. Hierarchical inheritance
 - d. Multiple inheritance
14. Polymorphism
 - a. Method overloading
 - b. Method overriding

Introduction:

In all the programs, we have designed our program around functions i.e. blocks of statements which manipulate the data. This is called the *procedure-oriented* way of programming. There is another way of organizing your program which is to combine data and functionality and wrap it inside something called an object. This is called the *object oriented* programming paradigm.

Classes and objects are the two main aspects of object oriented programming. A class creates a new type where objects are instances of the class.

Classes and Objects

1. Object is simply a collection of data (variables) and methods (functions).
2. Objects are an encapsulation of variables and functions into a single entity.
3. Objects get their variables and functions from classes.
4. Classes are essentially a template to create your objects.
5. Class is a blueprint for the object
6. A class is the blueprint from which the individual objects are created. Class is composed of three things: a name, attributes, and operations

Class → data members and member functions or

data and functions or

data and member or

variables and functions or

variable and method or

states and behaviors

Eg1:

If we think of **class as a sketch(Phototype)** of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. **House is the object.**

We can make many houses from the description. So we can create many objects from a class. An object is also called an **instance of a class** and the process of creating this object is called **instantiation**

Eg2:

In an apartment, we may have multiple houses are there and almost all are having same structure.

we define a class by using class keyword

Accessing Variable

```
class ClsName:  
    VarName = "Python"  
  
    def function(self):  
        print("Python is very simple and easy language")
```

```
ObjName=ClName()
```

```
print(ObjName.VarName)
```

output: Python

We can create multiple different objects with the same class(which is having variables and functions).

Each object contains its own copy of the variables

```
class ClsName:  
    VarName = "Python"  
  
    def function(self):  
        print("Python is very simple and easy language")
```

```
ObjNameA=ClName()  
ObjNameB=ClName()
```

```
ObjNameB.VarName="Developer"  
print(ObjNameA.VarName)  
print(ObjNameB.VarName)  
  
output: Python  
Developer
```

Accessing Function:

We can also access the function to the object separately like a variable,

```
class ClsName:  
    VarName = "Python"  
  
    def function(self):  
        print("Python is very simple and easy language")  
  
ObjName=ClisName()  
  
ObjName.function()  
print(ClsName.__doc__)  
  
Output:  
Python is very simple and easy language
```

self

Class methods have only one specific difference from ordinary functions - they must have an extra first name that has to be added to the beginning of the parameter list, but you **do not** give a value for this parameter when you call the method, Python will provide it. This particular variable refers to the object *itself*, and by convention, it is given the name `self`.

Although, you can give any name for this parameter, it is *strongly recommended* that you use the name `self` - any other name is definitely frowned upon. There are many advantages to using a standard name - any reader of your program will immediately recognize it and even specialized IDEs (Integrated Development Environments) can help you if you use `self`.

Note for C++/Java/C# Programmers

The `self` in Python is equivalent to the `this` pointer in C++ and the `this` reference in Java and C#. You must be wondering how Python gives the value for `self` and why you don't need to give a value for it. An example will make this clear. Say you have a class called `MyClass` and an instance of this class called `myobject`. When you call a method of this object as `myobject.method(arg1, arg2)`, this is automatically converted by Python into `MyClass.method(myobject, arg1, arg2)` - this is all the special `self` is about. This also means that if you have a method which takes no arguments, then you still have to have one argument - the `self`.

Docstring

The first string is called docstring and it has brief description about the class.

Docstring is not mandatory but it is recommended.

When we define a class then class creates a local namespace where all its attributes are defined, attributes are may be data or functions.

There are special attributes also in it that begins with double underscore (`__`), for example `__doc__` which displays the docstring in that class.

Lets see some examples on class and objects,

```
class ClsName:  
    "This ClsName class contains VarName and fucntion attributes"  
    VarName = "Python"  
  
    def function(self):  
        print("Python is very simple and easy language")
```

In the above example, just one class is created with one variable and one function.

```
class ClsName:  
    "This ClsName class contains VarName and fucntion attributes"  
    VarName = "Python"
```

```
def function(self):
    print("Python is very simple and easy language")
ObjName=ClsName()
```

Now the object ObjName holds the object of class ClsName which contains the variable and function that are defined in ClsName class.

Now we can access variables and functions separately.

Static variables or class variables

1. The variables which are declared inside the class and outside all the methods are known as class variables or static variables.
2. Class or static variables are shared by all objects.
3. The data which common for all the objects is recommended to represent as static variable or class variable.
4. Memory will be allocated only once for all class or static variables.
5. We can also modify the values of static or class variables
6. We can access the static or class variables within the class or outside the class also by using class name.

Eg:

```
class Myclass:
    i="Python"          #static or class variable
    j="Dev"            ##static or class variable

    def DisMethod(self):
        print(Myclass.i)  #accessing static variables with class name
        print(Myclass.j)

x1=Myclass()
print(x1.i)          #accessing static variables outside the class
print(x1.j)
x1.DisMethod()       #it works like DisMethod(x1)
```

output:

```
Python
Dev
Python
Dev
```

In C++ and Java, we can use static keyword to make a variable as class variable. The variables which don't have preceding static keyword are instance variables

Python doesn't require a static keyword. *All variables which are declared in class declaration are class variables or static variables. And variables which are declared inside class methods are instance variables or non-static variables.*

Constructor:

1. Constructor is a special function which we call automatically when we create object for respective class.
2. It can be defined by using `__init__()` in python
3. It will not return anything
4. It is mainly used for initializing the variable

`__init__` is a kind of constructor, when a instance of a class is created, python calls `__init__()` during the instantiation to define additional behavior that should occur when a class is instantiated, basically setting up some beginning values for that object or running a routine required on instantiation.

`init` is an abbreviation for initialization.

Non-static variables or instance variables

1. The variables which are declared inside the method and with the 'self' keyword is known as instance variable or non-static variable.
2. Instance variables are owned by the specific instances of a class. This means the instance or non-static variables are different for different objects (every object has a copy of it).
3. Instance variables are always introduced with the word self. They are typically introduced and initialized in a constructor method named `__init__`.
4. We can define instance variables in a constructor or in a method.
5. The data which is separate for every object is recommended to represent as a instance variable or non-static variable.
6. Memory will be allocated for all instance variables whenever we create a object.

7. Instance variables of a class can be accessed within the same class by using 'self' keyword. And same variables also can be accessed outside the class by using reference variable name.

Eg 1:

```
class employee:  
    loc="Hyderabad"                      #class variable  
  
    def __init__(self,Empno,Ename,Sal):  
        self.Empno=Empno                  #instance variable  
        self.Ename=Ename                  #instance variable  
        self.Sal=Sal                      #instance variable  
  
    def display(self):  
        print("Emp number is ",self.Empno)  
        print("Emp name is ",self.Ename)  
        print("Emp Salary is ",self.Sal)  
        print ("Emp Location is ",employee.loc)  
  
emp1=employee(101,"Sai",10000)  
emp1.display()  
  
emp2=employee(102,"Nani",20000)  
emp2.display()  
  
emp3=employee(103,"Renu",30000)  
emp3.display()  
  
output:  
  
Emp number is 101  
Emp name is Sai  
('Emp Salary is 10000  
Emp Location is Hyderabad
```

```
Emp number is 102  
Emp name is Nani  
Emp Salary is 20000  
Emp Location is Hyderabad  
Emp number is 103  
Emp name is Renu  
Emp Salary is 30000  
Emp Location is Hyderabad
```

Eg2:

```
class Myclass:  
    def method1(self):  
        self.i="Python"  
  
    def display(self):  
        print(self.i)  
  
Obj1= Myclass ()  
Obj1.method1()  
Obj1.display()          #returns Python  
  
Obj1.i="It is very easy"  
Obj1.display()          #returns It is very easy  
  
Obj2= Myclass ()  
Obj2.method1()  
Obj2.display()          #returns Python  
  
Obj2.i="it is very powerful"  
Obj2.display()          #returns it is very powerful
```

```
Obj1.display()           #returns It is very easy
Obj2.display()           #returns it is very powerful
```

Output:

```
Python
It is very easy
Python
it is very powerful
It is very easy
it is very powerful
```

Data hiding: outside person cant access our internal data directly or our internal data should not go out directly this OOP feature is nothing but data hiding.

After validation or authentication only, outside person can access our internal data.

Eg 1: after providing proper username and password we can able to access our gmail inbox information

Eg 2: even though we are valid customer of the bank, we can able to access our account information and we can not access others account information

By declaring data member (variable) as private (__) only we can achieve data hiding.

Eg:

```
class account:
    __a=100
    def getBal():
        #validation
        return __balance
```

The biggest advantage of data hiding is security.

It is highly recommended to declare a data member as private (with underscore)

Data abstraction:

Hiding internal information and just highlight the set of services what we are offering is the concept of abstraction.

Through bank atm GUI screen, bank people are highlighting the set of services what they are offering without highlighting internal implementation, like the server they used to work, the database they used to the customer details, the language they used to implement communication between application and database.

1. Outside person doesn't know how it is implemented internally, so **security** is provided.
2. For example a bank ATM GUI is developed by python now, and python program execution speed is not upto mark, so the new language is there 'sython', now we can change internal implementation from python to sython without effecting the GUI display and end-user. We can do any modifications to internal implementation without effecting to end-user. That's why **enhancement** became very easy.
3. For example we need to know how the ATM card is implemented and everything about ATM card before use, then nobody can use ATM cards. Because it's not possible to know everything. That's why without knowing anything about internal implementation we are using ATM cards, and also WhatsApp. So here its improving **easiness** to end-user.
4. So we are able to do any internal implementation changes without effecting end user so here **Maintainability** of application becomes very easy.

Encapsulation

1. The process of binding the data members and corresponding methods into a single unit is called encapsulation.
2. Every python class is example of encapsulation
3. Class student has data members plus data methods is encapsulation
4. Encapsulation is nothing but the combination of data hiding and data abstraction.
5. If any component follows data hiding and data abstraction then that component is called encapsulated component.

6. Encapsulation is about ensuring the safe storage of data as attributes in an instance.
7. Encapsulation tells us that :
 - a. Data should only be accessed through instance methods.
 - b. Data should always be correct based on the validation requirement set in the class methods.
 - c. And Data should be safe from changes by external processes.

Encapsulation = data hiding + abstraction

Eg: capsule with medicine inside

```
class account:  
    __balance=100000  
  
    def getBal():  
        # validation  
        print(__balance)  
  
    def setBal(self,balance):  
        #validation  
        self.balance=balance
```

let's take bank ATM GUI screen and imagine there are only two button are there, like **BalanaceEnquiry** and **UpdateBalance**.

The above class contains data members which are declared as private for hiding from the others and also class contains methods **like** getBal() and setBal().

When the end-user clicks on BalanaceEnquiry button then automatically it will call getBal() method, and when user clicks on **UpdateBalance** button then automatically it will call setBal() method.

The complete and confidential data is hidden inside the class and just abstraction is given in the GUI.

So finally the above class contains data members and methods.

The main advantages of encapsulation are

1. We can achieve security

2. Enhancement will become very easy
3. It improves maintainability of application

The main advantage of encapsulation is security but the main disadvantage of encapsulation is it increases length of the code and slows down execution.

A class is said to be tightly encapsulated if and only if each and every variable declared as private

Whether class contains corresponding getter or setter methods or not and whether these methods are declared as public or not these things we are not required to check.

```
class classA:  
    __a=10  
    __x=100  
  
class classB(classA):  
    b=20  
  
class classC(classA):  
    __c=30
```

here classA and classC are tightly encapsulated classes because each and every variable under these two classes are private so nobody can access from outside but everyone can access variable under classB.

If parent class is not tightly encapsulated then automatically all corresponding its child classes are not tightly encapsulated

```
class classA:  
    a=10  
    x=100  
  
class classB(classA):  
    __b=20  
    __y=40  
  
class classC(classA):
```

```
__c=30
```

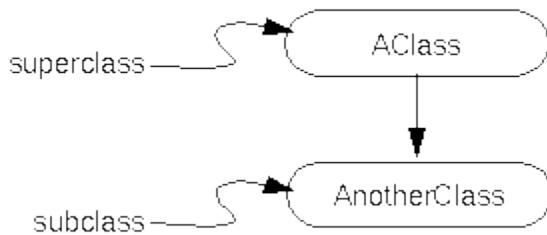
```
__z=50
```

here, even though classB and classC are having all private variables but these two classes are the child classes of classA which is not tightly encapsulated class.

Here, all non-private variables from classA can be accessed from classB and classC that's why classB and classC are not tightly encapsulated classes.

Inheritance

Classes can be derived from other classes. The derived class (the class that is derived from another class) is called a subclass. The class from which it's derived is called the superclass. The following figure illustrates these two types of classes:



The subclass inherits state and behavior in the form of variables and methods from its superclass. The subclass can use just the items inherited from its superclass as is, or the subclass can modify or override it. So, as you drop down in the hierarchy, the classes become more and more specialized:

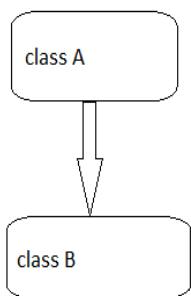
We use inheritance for code reusability.

Python supports different levels of inheritance

1. Single level inheritance
2. Multi-level inheritance
3. Multiple inheritance
4. Hierarchical inheritance

Single level inheritance

The process of inheriting all attributes from one class to the other class is known as single level inheritance.



Eg:

```
class A:    a1=100
    def methodA1(self):
        print('this is methodA1 belongs to class A')

    def methodA2(self):
        print('this is methodA2 belongs to class A')

class B(A):
    b1=200
    def methodB1(self):
        print('this is methodB1 belongs to class B')

    def methodB2(self):
        print('this is methodB2 belongs to class B')

ObjB=B()
print(ObjB.b1)
ObjB.methodB1()
ObjB.methodB2()
```

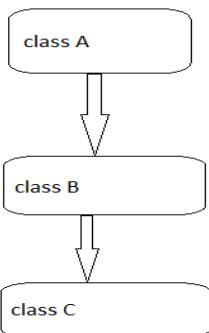
```
print(ObjB.a1)  
ObjB.methodA1()  
ObjB.methodA2()
```

Output:

```
200  
this is methodB1 belongs to class B  
this is methodB2 belongs to class B  
100  
this is methodA1 belongs to class A  
this is methodA2 belongs to class A
```

Multi-level inheritance

The process of inheriting all attributes from one base class to derived class, again from this derived class to another derived class and so on...



```
class A:  
    a1=100  
    def methodA1(self):
```

```
    print('this is methodA1 belongs to class A')
```

```
def methodA2(self):  
    print('this is methodA2 belongs to class A')
```

```
class B(A):  
    b1=200  
    def methodB1(self):  
        print('this is methodB1 belongs to class B')
```

```
def methodB2(self):  
    print('this is methodB2 belongs to class B')
```

```
class C(B):  
    c1=300  
    def methodC1(self):  
        print('this is methodC1 belongs to class C')  
    def methodC2(self):  
        print('this is methodC2 belongs to class C')
```

```
class D(C):  
    d1=400  
    def methodD1(self):  
        print('this is methodD1 belongs to class D')  
    def methodD2(self):  
        print('this is methodD2 belongs to class D')
```

```
ObjD=D()
```

```
print(ObjD.d1)
ObjD.methodD1()
ObjD.methodD2()

print(ObjD.c1)
ObjD.methodC1()
ObjD.methodC2()

print(ObjD.b1)
ObjD.methodB1()
ObjD.methodB2()

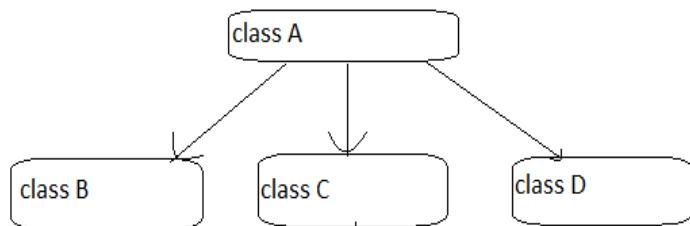
print(ObjD.a1)
ObjD.methodA1()
ObjD.methodA2()

Output:

400
this is methodD1 belongs to class D
this is methodD2 belongs to class D
300
this is methodC1 belongs to class C
this is methodC2 belongs to class C
200
this is methodB1 belongs to class B
this is methodB2 belongs to class B
100
this is methodA1 belongs to class A
this is methodA2 belongs to class A
```

Hierarchical inheritance

The process of inheriting all attributes from one base class into multiple derived classes is called hierarchical inheritance.



```
class A:  
    a1=100  
    def methodA1(self):  
        print('this is methodA1 belongs to class A')  
  
    def methodA2(self):  
        print('this is methodA2 belongs to class A')  
  
class B(A):  
    b1=200  
    def methodB1(self):  
        print('this is methodB1 belongs to class B')  
    def methodB2(self):  
        print('this is methodB2 belongs to class B')  
  
class C(A):  
    c1=300  
    def methodC1(self):  
        print('this is methodC1 belongs to class C')
```

```
def methodC2(self):
    print('this is methodC2 belongs to class C')

class D(A):
    d1=400
    def methodD1(self):
        print('this is methodD1 belongs to class D')

    def methodD2(self):
        print('this is methodD2 belongs to class D')

ObjD=D()
print(ObjD.d1)
ObjD.methodD1()
ObjD.methodD2()
print(ObjD.a1)
ObjD.methodA1()
ObjD.methodA2()

ObjC=C()
print(ObjC.c1)
ObjC.methodC1()
ObjC.methodC2()
print(ObjC.a1)
ObjC.methodA1()
ObjC.methodA2()

ObjB=B()
print(ObjB.b1)
ObjB.methodB1()
ObjB.methodB2()
print(ObjB.a1)
```

Narayana

PYTHON

Narayana

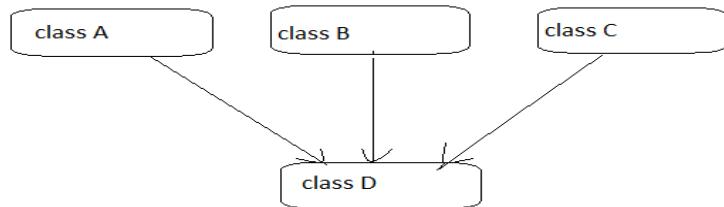
```
ObjB.methodA1()
ObjB.methodA2()

output:

400
this is methodD1 belongs to class D
this is methodD2 belongs to class D
100
this is methodA1 belongs to class A
this is methodA2 belongs to class A
300
this is methodC1 belongs to class Cthis is methodC2 belongs to class C
100
this is methodA1 belongs to class A
this is methodA2 belongs to class A
200
this is methodB1 belongs to class B
this is methodB2 belongs to class B
100
this is methodA1 belongs to class A
this is methodA2 belongs to class A
```

Multiple inheritance

The process of inheriting all attributes from multiple base classes into single derived class this process is known as multiple inheritance



```
class A:  
    a1=100  
    def methodA1(self):  
        print('this is methodA1 belongs to class A')  
  
    def methodA2(self):  
        print('this is methodA2 belongs to class A')  
  
class B:  
    b1=200  
    def methodB1(self):  
        print('this is methodB1 belongs to class B')  
  
    def methodB2(self):  
        print('this is methodB2 belongs to class B')  
  
class C:  
    c1=300  
    def methodC1(self):  
        print('this is methodC1 belongs to class C')  
  
    def methodC2(self):  
        print('this is methodC2 belongs to class C')
```

```
class D(C,B,A):
    d1=400
    def methodD1(self):
        print('this is methodD1 belongs to class D')
    def methodD2(self):
        print('this is methodD2 belongs to class D')
```

```
ObjD=D()
print(ObjD.d1)
ObjD.methodD1()
ObjD.methodD2()
print(ObjD.c1)
ObjD.methodC1()
ObjD.methodC2()
print(ObjD.b1)
ObjD.methodB1()
ObjD.methodB2()
print(ObjD.a1)
ObjD.methodA1()
ObjD.methodA2()
```

Output:

```
400
this is methodD1 belongs to class D
this is methodD2 belongs to class D
300
```

```
this is methodC1 belongs to class C  
this is methodC2 belongs to class C  
200  
this is methodB1 belongs to class B  
this is methodB2 belongs to class B  
100  
this is methodA1 belongs to class A  
this is methodA2 belongs to class A
```

Polymorphism

- This is also a “built in” python example of polymorphism.
- The same operation results in different behaviors depending on the type of data is given.
- The same idea(same operation- different behavior) can be applied to our own class and objects.
- Polymorphism is the ability to leverage the same interface for different underlying forms such as data types or classes. This permits functions to use entities of different types at different times.
- For object-oriented programming in Python, this means that a particular object belonging to a particular class can be used in the same way as if it were a different object belonging to a different class.
- Polymorphism allows for flexibility and loose coupling so that code can be extended and easily maintained over time.
- Generally polymorphism supports method overloading and method overriding but python will not support method overloading.

Method overloading

The concept of defining multiple methods with same name but different number of parameters is called method overloading.

```
class classA:  
    def method1(self):  
        print('this method belongs to class classA')  
  
class classB:  
    def method1(self,a):  
        print('this method belongs to class classB')  
  
ObjB=classB()  
ObjB.method1(10)  
ObjB.method1()      #it will return error so overloading will not support  
  
Output:  
this method belongs to class classB
```

Method overriding

The concept of defining multiple methods with the same name with the same number of parameters is called method overriding.

```
class classA:  
    def method1(self):  
        print('this method belongs to class classA')  
  
class classB:  
    def method1(self):  
        print('this method belongs to class classB')  
  
ObjB=classB()  
ObjB.method1()  
  
Output:  
this method belongs to class classB
```

Real time example for Method overloading

```
class BankAccount:  
    def __init__(self,balance=10000):  
        self.balance=balance  
  
    def deposite(self,value):  
        self.balance=self.balance+value  
        print('The current balance is: ',self.balance)  
  
    def withdraw(self,value):  
        self.balance=self.balance-value  
        print('The current balance is: ',self.balance)  
  
  
class SavingsAccount(BankAccount):  
    def __init__(self,balance=10000):  
        self.balance=balance  
  
    def deposite(self, value):  
        self.balance=self.balance+(value*1.03)  
  
        print('The current balance in savings account is: ',self.balance)  
  
  
class CurrentAccount(BankAccount):  
    def __init__(self,balance=10000):  
        self.balance=balance  
  
    def withdraw(self,value):  
        if value>1000:  
            print('You can withdraw less than 1000 only')  
        else:  
            self.balance=self.balance-value  
            print('Your current amount in current account is', self.balance)
```

```
SA=SavingsAccount()
CA=CurrentAccount()

while True:
    print('1.Savings Account')
    print("2.Current Account")
    MOption=int(input('Please select the account type: '))
    if MOption==1:
        print('1.Withdraw')
        print('2.Deposite')
        SOption=int(input('Please select any operation type: '))
        if SOption==1:
            value=int(input('Please enter amout to withdraw from savings account: '))
            SA.withdraw(value)
        elif SOption==2:
            value=int(input('Please enter amount to deposite in savings account: '))
            SA.deposite(value)
        else:
            print('You entered ',SOption,'Its invalid operatoion')
    elif MOption==2:
        print('1.Withdraw')
        print('2.Deposite')
        SOption=int(input('Please select any operation type: '))
        if SOption==1:
            value=int(input('Please enter amount to withdraw from current account: '))
            CA.withdraw(value)
        elif SOption==2:
            value=int(input('Please enter amount to deposite in current account: '))
            CA.deposite(value)
        else:
```

```
print('You entered ',SOption,'Its invalid operatoion')
else:
    print('You entered ',MOption,'Its invalid Account Type')
break
```

#this example is done without break statement

Output1:

```
1.Savings Account
2.Current Account
Please select the account type: 1
1.Withdraw
2.Deposite
Please select any operation type: 1
Please enter amout to withdraw from savings account: 2000
The current balance is: 8000
```

Output2:

```
1.Savings Account
2.Current Account
Please select the account type: 1
1.Withdraw
2.Deposite
Please select any operation type: 2
Please enter amount to deposite in savings account: 3000
The current balance in savings account is: 11090.0
```

Output3:

```
1.Savings Account
2.Current Account
Please select the account type: 2
```

```
1.Withdraw  
2.Deposite  
Please select any operation type: 1  
Please enter amount to withdraw from current account: 1500  
You can withdraw less than 1000 only
```

Output4:

```
1.Savings Account  
2.Current Account  
Please select the account type: 2  
1.Withdraw  
2.Deposite  
Please select any operation type: 1  
Please enter amount to withdraw from current account: 500  
Your current amount in current account is 9500
```

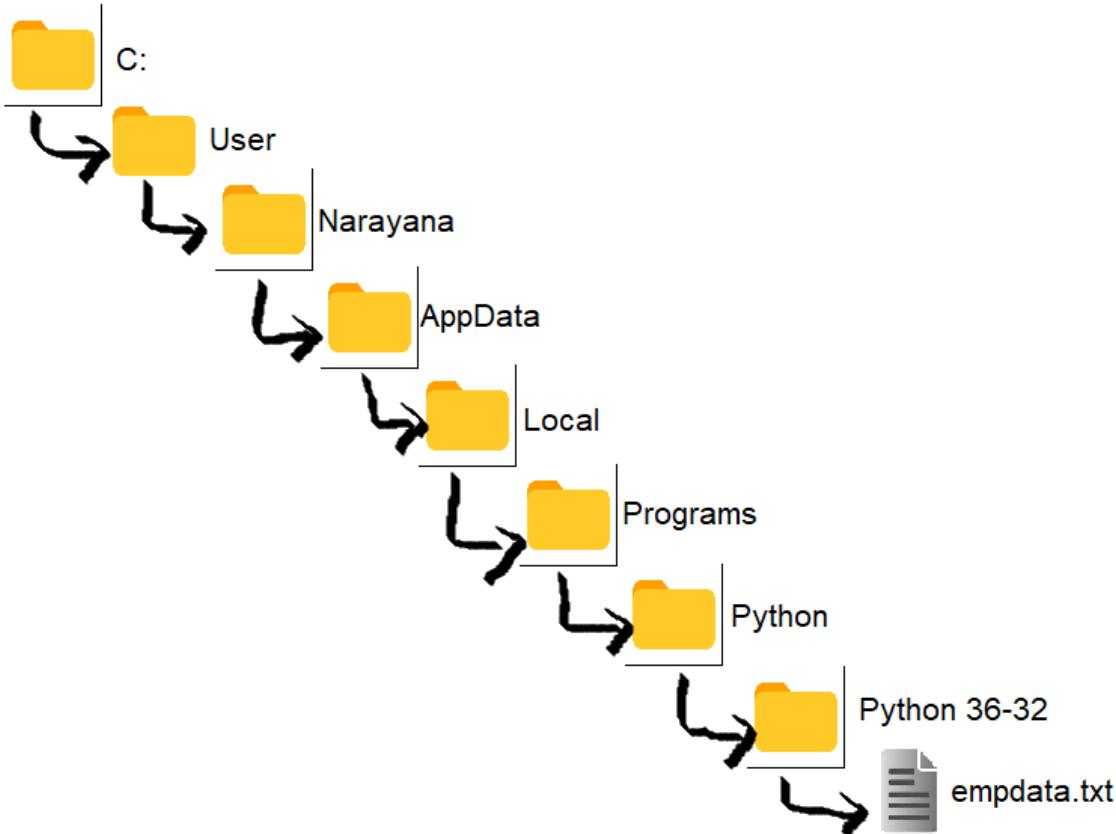
Output5:

```
1.Savings Account  
2.Current Account  
Please select the account type: 2  
1.Withdraw  
2.Deposite  
Please select any operation type: 2  
Please enter amount to deposit in current account: 2000  
The current balance is: 11500
```

Reading and writing files:

Variables are a fine way to store data while your program is running, but if you want your data to persist even after your program has finished, you need to save it to a file.

A file has two key properties: a filename (usually written as one word) and a path. The path specifies the location of a file on the computer. For example,



There is a file on my windows 10 laptop with the filename empdata.txt in the path

'C:\Users\Narayana\AppData\Local\Programs\Python\Python36-32'. these all refer to folders or directories. Folders can contain files and other folders.

For example, empdata.docx is in the python 36-32 folder, which is inside the python folder, which is inside the Program folder, which is inside the Local folder, which is inside the AppData folder. which is inside the Narayana folder, Which is inside the Users,Which is inside the C drive.

C:\ is called root folder which contains all other folders

working with slashes (\ or /)

On Windows, paths are written using backslashes (\) as the separator between folder names. OS X and Linux, however, use the forward slash (/) as their path separator.

If we want our programs to work on all operating systems, we will have to write our Python scripts to handle both cases.

Fortunately, this is simple to do with the os.path.join() function.

If we pass it the string values of individual file and folder names in our path, os.path.join() will return a string with a file path using the correct path separators.

```
>>> import os  
>>> os.path.join('a','b','c')  
'a\\b\\c'  
  
>>> os.path.join('Users','Narayana','AppData','Local','Programs','Python','Python36-32')  
'Users\\Narayana\\AppData\\Local\\Programs\\Python\\Python36-32'
```

The os.path.join() function is helpful if we need to create strings for filenames.

For example, the following example joins names from a list of filenames to the end of a folder's name:

```
>>> files=["empdetails.txt",'custdata.docx','productsdata.csv','salesdetails.txt']  
  
>>> for name in files:  
  
    print(os.path.join("C:\\\\Users\\\\Narayana\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32",name))
```

```
C:\\Users\\Narayana\\AppData\\Local\\Programs\\Python\\Python36-32\\empdetails.txt
```

```
C:\\Users\\Narayana\\AppData\\Local\\Programs\\Python\\Python36-32\\custdata.docx
```

```
C:\\Users\\Narayana\\AppData\\Local\\Programs\\Python\\Python36-32\\productsdata.csv
```

```
C:\\Users\\Narayana\\AppData\\Local\\Programs\\Python\\Python36-32\\salesdetails.txt
```

The Current Working Directory

Every program that runs on our computer has a current working directory or cwd. Any filenames or paths that do not begin with the root folder are assumed to be under the current working directory. We can get the current working directory as a string value with the `os.getcwd()` function and change it with `os.chdir()`.

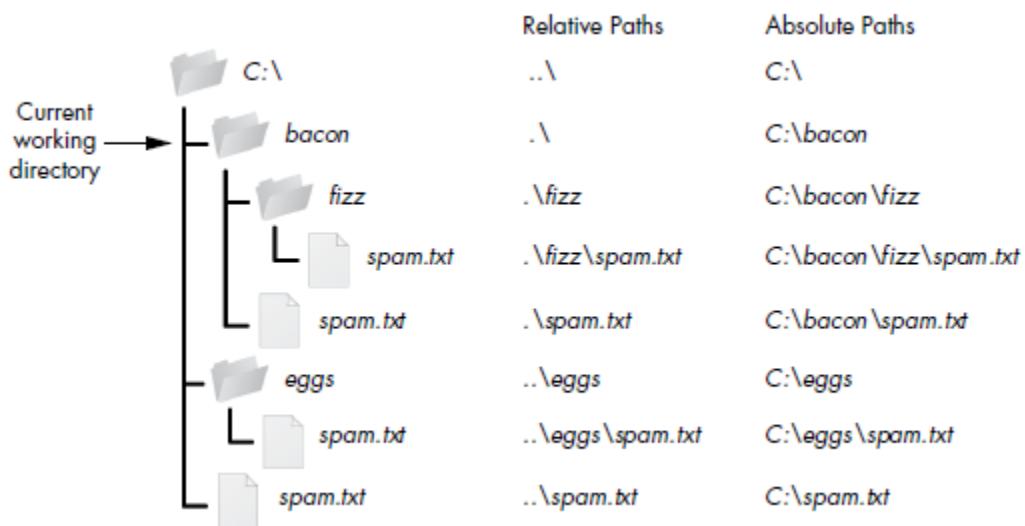
```
>>> os.getcwd()  
'C:\\Users\\Narayana\\AppData\\Local\\Programs\\Python\\Python36-32'  
>>> os.chdir("E:\\FilesFolder")  
>>> os.getcwd()  
'E:\\FilesFolder'
```

Absolute vs. Relative Paths

There are two ways to specify a file path.

- An absolute path, which always begins with the root folder
- A relative path, which is relative to the program's current working directory

There are also the dot (.) and dot-dot (..) folders. These are not real folders but special names that can be used in a path. A single period ("dot") for a folder name is shorthand for "this directory." Two periods ("dot-dot") means "the parent folder."



File Sizes and Folder Contents:

Once we have ways of handling file paths, we can then start gathering information about specific files and folders. The `os.path` module provides functions for finding the size of a file in bytes and the files and folders inside a given folder.

- Calling `os.path.getsize(path)` will return the size in bytes of the file in the path argument.

- Calling `os.listdir(path)` will return a list of filename strings for each file in the path argument. (Note that this function is in the `os` module, not `os.path`.)

```
>>> os.path.getsize('C:\\\\Users\\\\Narayana\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32\\\\empdata.txt')
```

```
94
```

Here, `empdata.txt` file size is 94bytes.

```
>>> os.listdir("C:\\\\Users\\\\Narayana\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32")
```

```
['all functions.py', 'bankexample.py', 'browser.py', 'comp.py', 'compre.py', 'data.pdf', 'Decorator Example.py', 'details.docx', 'details1.docx', 'empdata.txt', 'multithreading.py', 'mydata.pdf', 'myfile.py', 'ourdata.csv', 'python3.dll', 'snakegame.py', 'tcl', 'test.py', 'testing.py', 'testing12.py', 'Tools', 'unittst1.py', 'urdata.docx', 'userexp.py', 'ut.py', 'ut1.py', 'vcruntime140.dll', 'webpage.py', 'while.py', 'zipp.py', '__pycache__']
```

Here, it is displaying all files which are in python 36-32 folder.

If we want find the total size of all the files in this directory, then we can use `os.path.getsize()` and `os.listdir()` together.

```
>>> totalsize=0
>>> for name in os.listdir("C:\\\\Users\\\\Narayana\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32"):
    totalsize=totalsize+os.path.getsize(os.path.join("C:\\\\Users\\\\Narayana\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32",name))
>>> print(totalsize)
9552273
```

So the total size the folder `python36-32` is 9552273 bytes

File Handling

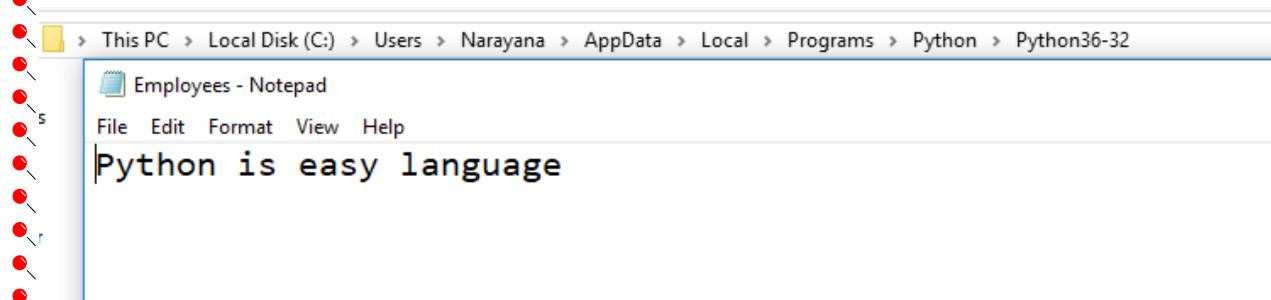
1. Generally the data can be placed in different places like in database, in file etc..
2. Python supports different kinds of files like .txt file, .pdf file, excel file, csv file, etc
3. The file can be placed in python installed folder or in shared folder or in any other directory in the machine.
4. In python we can open any file with open().
5. Open() function has two arguments like open('filename','mode,)
6. The argument 'filename' is a file name or path of the file.
7. If the file is available in python installed folder then we don't need to specify the path of the file, because an interpreter goes to the python installed folder by default.
8. If the file is in shared folder or in any other folder then we need to specify the path of the file, so that the interpreter will goto the specified path and search for the specific file.
9. The second argument 'mode' is the type of operation that will be performed on the file.
10. The different types of modes are w(write), r(read), a(append),r+(both read and write).
11. The default argument is r(read)

Create a empty text file in the folder where python is installed

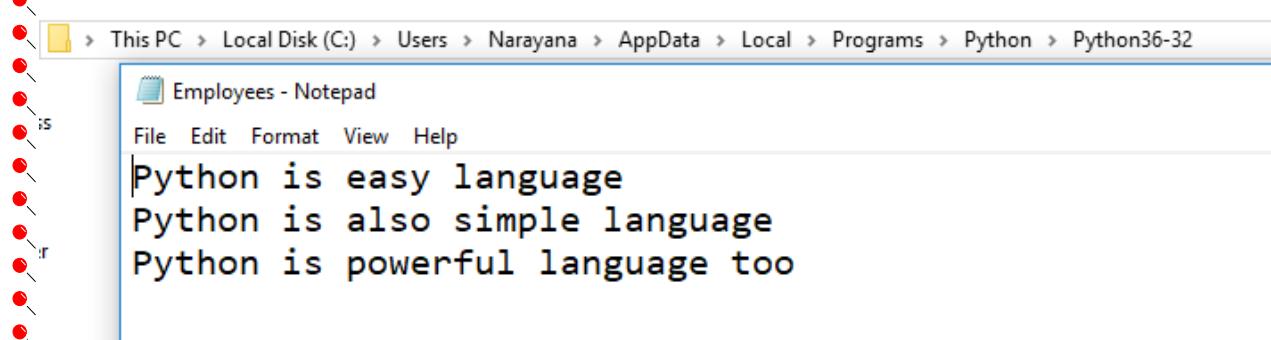
C:\Users\Narayana\AppData\Local\Programs\Python\Python36-32\Employees.txt

1. Writing the data to file:

```
a=open('Employees.txt','w')
a.write('Python is easy language')
a.close()
```

**2. Appending new data to the existing file**

```
a=open('Employees.txt','a')
a.write('\nPython is also simple language\nPython is powerful language too')
a.close()
```



Note: Now let's take the following data in the file to perform read operations

```
tcs,narayana,venu,balu
wipro,suresh,naresh,ramesh
info,renu,amala,kavya,ramu
cts,veni,satya,durga,ramya
```

3. Displaying all the data from the file

```
a=open('Employees.txt','r')
x=a.read()
print(x)

or

print(open('Employees.txt','r').read())

output:

tcs,narayana,venu,balu
wipro,suresh,naresh,Ramesh
info,renu,amala,kavya,ramu
cts,veni,satya,durga,ramya
```

4. Displaying first line from the file

```
a=open('Employees.txt','r')
v=a.readline()
print(v)

or

print(open('Employees.txt','r').readline())

or

a=open('Employees.txt','r')
v=a.readlines()
print(v[0])

or

print(open('Employees.txt','r').readlines()[0])

output:

tcs,narayana,venu,balu
```

5. Displaying second line from the file

```
a=open('Employees.txt','r')
v=a.readlines()
print(v[1])

or

print(open('Employees.txt','r').readlines()[1])

output:
wipro,suresh,naresh,ramesh
```

6. Displaying 3rd line from the file

```
a=open('Employees.txt')
v=a.readlines()
print(v[2])

or

print(open('Employees.txt').readlines()[2])

output
info,renu,amala,kavya,ramu
```

7. Displaying last line in the file

```
a=open('Employees.txt','r')
v=a.readlines()
print(v[-1])

or

print(open('Employees.txt','r').readlines()[-1])

output:
cts,veni,satya,durga,ramya
```

8. Counting number of lines in the file

```
a=open('Employees.txt','r')
v=a.readlines()
print(len(v))

output: 4
```

9. Displaying first word from each line

```
a=open('Employees.txt','r')
v=a.readlines()
for i in v:
    k=i.split(',')
    print(k[0])

or

for i in open('Employees.txt','r').readlines():
    print(i.split(',')[0])

output:
tcs
wipro
info
cts
```

10. Displaying second word from each line

```
a=open('Employees.txt','r')
v=a.readlines()
for i in v:
    k=i.split(',')
    print(k[1])
```

```
or  
for i in open('Employees.txt','r').readlines():  
    print(i.split(',')[1])  
  
output:  
  
Narayana  
suresh  
renu  
veni
```

11. Displaying last word from the each line

```
a=open('Employees.txt','r')  
v=a.readlines()  
for i in v:  
    k=i.split(',')  
    print(k[-1])  
  
or  
for i in open('Employees.txt','r').readlines():  
    print(i.split(',')[-1])  
  
output  
  
balu  
Ramesh  
ramu  
ramya
```

12. Displaying first word in the first line

```
a=open('Employees.txt','r')  
v=a.readlines()  
x=v[0]  
y=x.split(',')  
print(y[0])
```

```
or  
print(open('Employees.txt','r').readlines()[0].split(',')[0])  
  
output:  
tcs
```

13. Displaying second word in the first line

```
a=open('Employees.txt','r')  
v=a.readlines()  
x=v[0]  
y=x.split(',')  
print(y[1])  
  
or  
  
print(open('Employees.txt','r').readlines()[0].split(',')[1])  
  
output:  
narayana
```

14. Displaying third word in the second line

```
a=open('Employees.txt','r')  
v=a.readlines()  
x=v[1]  
y=x.split(',')  
print(y[2])  
  
or  
  
print(open('Employees.txt','r').readlines()[1].split(',')[2])  
  
output:  
naresh
```

15. Displaying last word in the last line

```
a=open('Employees.txt','r')
v=a.readlines()
x=v[-1]
y=x.split(',')
print(y[-1])
or
print(open('Employees.txt','r').readlines()[-1].split(',')[-1])
output:
ramya
```

16. Displaying first character from each line

```
a=open('Employees.txt','r')
v=a.readlines()
for i in v:
    print(i[0])
or
v=open('Employees.txt','r').readlines()
for i in v:
    print(i[0])
output:
t
w
i
c
```

17. Displaying second character from each line

```
a=open('Employees.txt','r')
v=a.readlines()
for i in v:
    print(i[1])
or
v=open('Employees.txt','r').readlines()
for i in v:
    print(i[1])
output:
c
i
n
t
```

18. Displaying last character from each line

```
a=open('Employees.txt','r')
v=a.readlines()
for i in v:
    print(i[-2])
or
v=open('Employees.txt','r').readlines()
for i in v:
    print(i[-2])
output:
u
h
u
y #last but one character from last line
```

19. Display first character from the file

```
a=open('Employees.txt')
v=list(a.read())
print(v[0])
or
print(list(open('Employees.txt').read())[0])
output: t
```

20. Display last character from the file

```
a=open('Employees.txt')
v=list(a.read())
print(v[-1])
or
print(list(open('Employees.txt').read())[-1])
output: a
```

21. Displaying first character of second word in the first line

```
a=open('Employees.txt')
v=a.readlines()
x=v[0]
y=x.split(',')
z=y[1]
print(z[0])
or
print(open('Employees.txt').readlines()[0].split(',')[1][0])
output: n
```

22. Displaying third character of third word in the third line

```
a=open('Employees.txt')
v=a.readlines()
x=v[2]
y=x.split(',')
z=y[2]
print(z[2])

or

print(open('Employees.txt').readlines()[2].split(',')[2][2])

output: a
```

23. Displaying second character of third word in the fourth line

```
a=open('Employees.txt')
v=a.readlines()
x=v[3]
y=x.split(',')
z=y[2]
print(z[1])

or

print(open('Employees.txt').readlines()[3].split(',')[2][1])

output: a
```

24. Displaying fourth character in the first line

```
a=open('Employees.txt')
v=a.readlines()
x=v[0]
print(x[4])

or

print(open('Employees.txt').readlines()[0][4])
```

```
output: n
```

25. Displaying 10th character in the third line

```
a=open('Employees.txt')  
v=a.readlines()  
x=v[2]  
print(x[10])  
  
or  
  
print(open('Employees.txt').readlines()[2][10])  
  
output: a
```

26. Displaying number of characters in the file(including commas)

```
a=open('Employees.txt','r')  
v=a.read()  
print(len(v))  
  
output: 103
```

27. Displaying number of characters in the file(excluding commas)

```
a=open('Employees.txt','r')  
v=a.readlines()  
c=0  
for i in v:  
    x=i.split(',')  
    for j in x:  
        c=c+len(j)  
print(c)  
  
output: 89
```

28. Displaying number of characters in the first line (including commas)

```
a=open('Employees.txt','r')
v=a.readline()
print(len(v))

or

a=open('Employees.txt','r')
v=a.readlines()
print(len(v[0]))

output: 23
```

29. Displaying number of characters in the first line(excluding commas)

```
a=open('Employees.txt')
v=a.readlines()
x=v[0].split(',')
k=0
for i in x:
    k=k+len(i)
print(k)

output: 20
```

30. Displaying number of characters in the second line (including commas)

```
a=open('Employees.txt','r')
v=a.readlines()
print(len(v[1]))

output: 27
```

31. Displaying number of characters in the last line (excluding commas)

```
a=open('Employees.txt')
v=a.readlines()
x=v[-1].split(',')
k=0
for i in x:
    k=k+len(i)
print(k)

output: 22
```

32. Displaying number of characters in the last line(including commas)

```
a=open('Employees.txt','r')
v=a.readlines()
print(len(v[-1]))

output: 26
```

33. Displaying number of characters in the last line(excluding commas)

```
a=open('Employees.txt')
v=a.readlines()
x=v[-1].split(',')
k=0
for i in x:
    k=k+len(i)
print(k)

output: 22
```

34.Counting number of characters in first word

```
a=open('Employees.txt')
v=a.readlines()
a=v[0].split(',')
print(len(a[0]))

output: 3
```

35.Counting number of characters in the last word of the first line

```
a=open('Employees.txt')
v=a.readlines()
x=v[0].split(',')
print(len(x[-1]))

output: 5
```

36.Counting number of characters in the last word of the last line

```
a=open('Employees.txt')
v=a.readlines()
x=v[-1].split(',')
print(len(x[-1]))

output: 5
```

37.Counting number of characters in the file

```
a=open('Employees.txt')
v=len(a.read())
print(v)

output:103
```

38.Counting number of words in the first line

```
a=open('Employees.txt')  
v=a.readlines()  
print(len(v[0].split(',')))  
  
output: 4
```

39.Counting number of words in the last line

```
a=open('Employees.txt')  
v=a.readlines()  
x=v[-1].split(',')  
print(len(x))  
  
output: 5
```

40.Counting number of words in the second line

```
a=open('Employees.txt')  
v=a.readlines()  
x=v[1].split(',')  
print(len(x))
```

41.Counting number of words in the file

```
a=open('Employees.txt')  
v=a.readlines()  
x=0  
for i in v:  
    k=i.split(',')  
    x=x+len(k)  
print(x)  
  
output: 18
```

42.Counting number of lines in the file

```
a=open('Employees.txt')  
v=a.readlines()  
print(len(v))  
  
output: 4
```

43.Counting number of vowels in the file

```
a=open('Employees.txt')  
b=a.read()  
v="aeiouAEIOU"  
d={}.fromkeys(v,0)  
  
for i in b:  
    if i in d:  
        d[i]=d[i]+1  
  
print(d)  
  
output: {'a': 18, 'A': 0, 'e': 6, 'i': 3, 'o': 2, 'l': 0, 'u': 6, 'O': 0, 'E': 0, 'U': 0}
```

44.Make the first word as header and remaining words as members

```
a=open('Employees.txt','r')  
v=a.readlines()  
for i in v:  
    x=i.split(',')  
    print x[0],'\n-----'  
    for j in range(1,len(x)):  
        print(x[j])  
  
output:  
-----  
Narayana
```

Narayana

PYTHON

Narayana

venu
balu
wipro

suresh
naresh
ramesh
info

renu
amala
kavya
ramu
cts

veni
satya
durga
ramya

Assignment - 3

1. What are the identity operators? And explain each one?
2. What are the membership operators? And explain each one?
3. What is the difference between division and floor division? Explain with examples?
4. What are the bitwise operators?
5. If $a=5$ and $b=7$ then what are the results of
 - a. $a \& b =$
 - b. $a | b =$
 - c. $a ^ b =$
 - d. $a >> 2 =$
 - e. $a << 2 =$
 - f. $b >> 3 =$
 - g. $b << 3 =$
6. How to check whether specific element is there or not in the given list?
7. How to compare the addresses of two variables?
8. What is for loop? Give one example?
9. What is while loop? Give one example?
10. Display 10th table by using for loop?

11. How to add first 10 numbers?
12. What is def keyword? And give one example by using def keyword?
13. What is a function definition and function call?
14. What are the actual parameters and formal parameters?
15. What are *args and **kwargs? Purpose of each one?
16. What is lambda function? When we use lambda functions?
17. What are the other functions that we use along with lambda functions?
18. What is filter function? Given one example?
19. What is map function? Give one example?
20. What is reduce function? Give one example?
21. A list has both even and odd number, then how to access only even numbers? Explain with example?

22. A list has both negative and positive numbers, then how to access only positive numbers? Explain with example?
23. We have two lists, each one has some elements but some elements are matching in two lists, then how to check what are the matching elements? Explain with one example?
24. A list has so many int type values, how to access greatest element among all? Explain with one example?
25. How to find squares of all the elements in the list?
26. How add all elements from both sets as per their index positions?
27. Write a python program for the following scenario, one MNC is conducting interviews, they asked candidates to fill the online form,
- a. Enter your name, (this student name should appear in every display statement)
 - b. Qualification must be B.Tech/BE
 - i. If he enters other qualification then display “You are not eligible”
 - ii. If he enters B.Tech or B.E then check passed out year
 - c. Passedout year must be either 2016 or 2017
 - i. If he enters 2015 or below then display “You are not Fresher”
 - ii. If he enters 2018 or above then display “You entered invalid year”
 - iii. If he enters either 2016 or 2017 then check percentage
 - d. If percentage
 - i. between 70 and 100, then display “You can attend the interview on coming Saturday”
 - ii. between 50 and 70, then “You can attend interview after two months”
 - iii. between 35 and 50, then “First you can work in some other companies first two years and then come to interview in our company with experience”
 - iv. If between 0 and 35, then “please don’t attend the interview in our company at any time”
 - v. If he enters less than 0 and more than 100 then “Invalid percentage”
 - e. In any case,display “**Thank You for showing interest in our company**” at the end

28. You need to ask user to enter any three values and display him the maximum value out of those three values?

- If the user values like a, b and c then the result must like,
a is greater than b and c or
b is greater than a and c or
c is greater than a and b

29. You need to ask user to enter his required product name, if the product name length is more than 1 character then perform the following operations on the entered product name else display “You entered invalid product name”, The operations are,

- Find the length of given string
- Display first character in upper case
- Check whether that product name contains “x” or not
- Display given product name in reverse order
- Display given string in ascending order like from a to z.
- Display given string in descending order like from z to a.

30. The given string is “NaRaYaNa” then how to display as “nArAyAnA”

31. The given string is “NARAYANA” then how to separate as ‘N’, ‘R’, ‘Y’, ‘N’

32. What is the output for the following:

```
def a1(a,*b):
    print(a,b)
a1(1,2,3)
```

Output:

33. What is the output for the following:

```
def av(a,b=10,*c,**d):
    print(a,b,c,d)
av(1,2,a1=3,c='f',d=4,r=5)
```

Output:

34. What is the output for the following:

```
def av(a,b=10,*c,**d):
    print(a,b,c,d)
av(1,2,'a1',3,'f',4,5)
output:
```

35. What is the difference between readline() and readlines() methods?

File Name is "CustomerData.txt"

```
1,John,Software, 30000, Hyd, Python
2,Viyaan,Data Analys',40000,Bang,Oracle
3,Renu ,Manager,50000,Mumbai,Python
```

36. Fetch last word in the file?

37. Display the salary of Viyaan?

38. Retrieve the second line data from the file?

39. Count number of a's and e's from the file?

40. How to count number rows in the file?

41. How to count number of words in the file?

42. How to retrieve all employee names from the file?

Exception Handling:

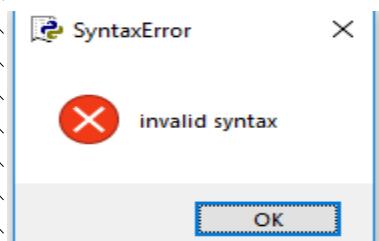
- An Exception is a run time error that happens during the execution of program.
- An exception is an error that happens during the execution of a program.
- Python raises an exception whenever it tries to execute invalid code.
- Error handling is generally resolved by saving the state of execution at the moment the error occurred and interrupting the normal flow of the program to execute a special function or piece of code, which is known as the exception handler. Depending on the kind of error ("division by zero", "file open error" and so on) which had occurred, the error handler can "fix" the problem and the program can be continued afterwards with the previously saved data.
- Exceptions handling in Python is very similar to Java. The code, which harbours the risk of an exception, is embedded in a try block. But whereas in Java exceptions are caught by catch clauses, we have statements introduced by an "except" keyword in Python. It's possible to "create custom-made" exceptions: With the raise statement it's possible to force a specified exception to occur.
- Generally any programming language supports two types of errors,
 1. Syntax errors
 2. Runtime errors

Syntax Errors: these errors will raise at development time and generally these errors will raise because of invalid syntax, so these are called syntax errors.

As these errors will occur at development time, the developer is responsible for the errors, developer needs to handle them.

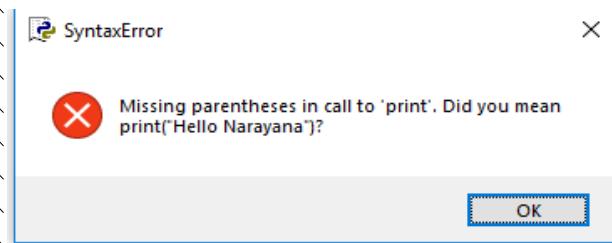
Eg1:

```
a=10  
b=5  
if a>b:  
    print(a)  
else:  
    print(b)
```



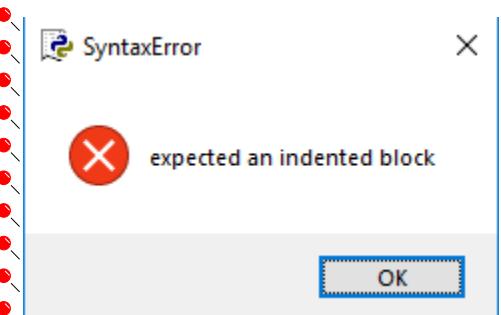
Eg2:

```
print "Hello Narayana"
```



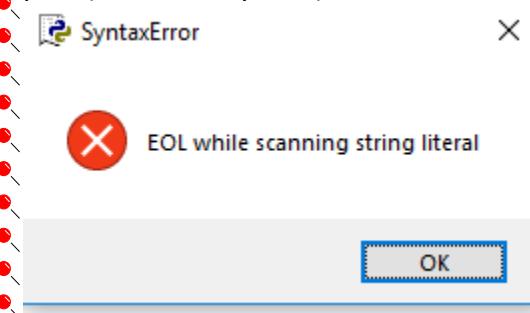
Eg3:

```
a=10  
b=5  
if a>b:  
    print(a)  
else:  
    print(b)
```



Eg4:

```
print('Hello Narayana')
```



Runtime errors: these are also called exceptions.

When the program is executing, if something goes wrong because of end user input or, programming logic or memory problems etc then we will call them runtime errors.

An exception is nothing but an unwanted or unexpected block which disturb the normal execution flow of program.

These exceptions are two types,

1. System defined exceptions
2. User defined exceptions

System defined exceptions:

These exceptions are defined by system so these are called system defined or pre-defined exceptions.

Some of system defined exceptions are,

ZeroDivisionError

```
NameError  
IndexError  
TypeError  
ValueError  
KeyError  
KeyBoardInturptError  
SleepingError  
FileNotFoundException  
ModuleNotFoundError  
LookupError  
IndentationError  
NotImplementedError
```

Note: Exception handling does not mean repairing exception, we have to define an alternative way to continue rest of the program normally.

Python supports handling the exceptions by using Try and except block.

Eg1:

```
a=int(input('Enter First Value: '))  
b=int(input('Enter Second Value: '))  
c=a/b  
print('The result is ', c)
```

output1:

```
Enter First Value: 10  
Enter Second Value: 4  
The result is 2.5
```

Output2:

Enter First Value: 10.5
Enter Second Value: 2.5
The result is 4.2

In this example, we don't have any exception, so its not required to use try and except block, if user gives denominator value 0 then it will throw an exception ZeroDivisionError.

Eg:

```
a=eval(input('Enter First Value: '))  
b=eval(input('Enter Second Value: '))  
c=a/b  
print('The result is ', c)
```

output:

Enter First Value: 10

Enter Second Value: 0

Traceback (most recent call last):

File "C:/Users/Narayana/AppData/Local/Programs/Python/Python36-32/ex

ceptionsfile.py", line 3, in <module>

c=a/b

ZeroDivisionError: division by zero

In order to handle errors, you can set up exception handling blocks in your code. The keywords `try` and `except` are used to catch exceptions. When an error occurs within the `try` block, Python looks for a matching `except` block to handle it. If there is one, execution jumps there.

```
a=eval(input('Enter First Value: '))  
b=eval(input('Enter Second Value: '))  
try:  
    c=a/b  
    print('The result is ', c)  
except:  
    print('Error Occured')
```

output:

Enter First Value: 10
Enter Second Value: 0
Error Occurred

- If user gives the value of b is 0, then automatically try block will raises ZeroDivisionError exception.
- Immediately the exception will be thrown to he except block and it will display 'Error Occurred' message.
- If user give the value of b more then 0, then try block will execute and return output results. Here except block will not execute.
- This except block is called **default except block**, because this except block will return same message for all types of exceptions which are thrown by try block.

Eg2:

```
a=eval(input('Enter First Value: '))
b=eval(input('Enter Second Value: '))
try:
    c=a/b
    print('The result is ', c)
except:
    print('Error Occured')

output:
```

Enter First Value: 10

Enter Second Value: 5

Error Occurred

In the example the try block has thrown TypeError to the except block, but except block has return same 'Error Occurred'

Eg3:

```
a=eval(input('Enter First Value: '))
b=eval(input('Enter Second Value: '))
try:
    c=a/d
    print('The result is ', c)
except:
    print('Error Occured')

output:
```

Enter First Value: 10

Enter Second Value: 5

Error Occurred

In the example the try block has thrown NameError to the except block, but except block has return same 'Error Occurred'.

Multiple Except blocks

We can use different except blocks to handle different exceptions separately,
A try statement may have more than one except clause for different exceptions. But at most one except clause will be executed.

Eg: using separate except block for handling ZeroDivisionError

```
a=eval(input('Enter First Value: '))
b=eval(input('Enter Second Value: '))
try:
    c=a/b
    print('The result is ', c)
except ZeroDivisionError:
    print('Please enter non-zero for denominator')
except:
    print('Error Occurred')
```

Output1:

```
Enter First Value: 10
Enter Second Value: 0
Please enter non-zero for denominator
```

Output2:

```
Enter First Value: 10
Enter Second Value: 3
The result is  3.333333333333335
```

Output3:

```
Enter First Value: 10
Enter Second Value: 'g'
Error Occurred
```

In the above example, we have used two except blocks the earlier one is for handling ZeroDivisionError and later one to handle remaining all other exceptions.

The default except block must be last except block when we have multiple blocks in the program.

Eg: using multiple except blocks for handling multiple exceptions separately

```
a=eval(input('Enter First Value: '))
b=eval(input('Enter Second Value: '))
try:
    c=a/b
    print('The result is ', c)
except ZeroDivisionError:
    print('Please enter non-zero for denominator')
except NameError:
    print('Please use defined names only')
except TypeError:
    print('Please use proper types only')
except:
    print('Error Occurred')
```

output:
Enter First Value: 10
Enter Second Value: 0
Please enter non-zero for denominator

```
Eg1
try:
    fh = open("testfile", "w")
    fh.write("This is my test file for exception handling!!")
except IOError:
    print "Error: can't find file or read data"
else:
    print "Written content in the file successfully"
    fh.close()
```

output:
Written content in the file successfully

```
a=eval(input('Enter First Value: '))
b=eval(input('Enter Second Value: '))
try:
    c=a/d
    print('The result is ', c)
except ZeroDivisionError:
    print('Please enter non-zero for denominator')
except NameError:
    print('Please use defined names only')
except TypeError:
    print('Please use proper types only')
```

```
except:  
    print('Error Occurred')
```

output:
Enter First Value: 10
Enter Second Value: 2
Please use defined names only

```
a=eval(input('Enter First Value: '))  
b=eval(input('Enter Second Value: '))  
try:  
    c=a/b  
    print('The result is ', c)  
except ZeroDivisionError:  
    print('Please enter non-zero for denominator')  
except NameError:  
    print('Please use defined names only')  
except TypeError:  
    print('Please use proper types only')  
except:  
    print('Error Occurred')
```

output:
Enter First Value: 10
Enter Second Value: 's'
Please use proper types only

Else block:

The try ... except statement has an optional else clause. An else block has to be positioned after all the except clauses. An else clause will be executed if the try clause doesn't raise an exception.

```
a=eval(input('Enter First Value: '))  
b=eval(input('Enter Second Value: '))  
try:  
    c=a/b  
    print('The result is ', c)  
except ZeroDivisionError:  
    print('Please enter non-zero for denominator')  
except NameError:  
    print('Please use defined names only')  
except TypeError:  
    print('Please use proper types only')
```

```
except:  
    print('Error Occurred')  
else:  
    print('Program executed successfully')  
  
Output1:  
Enter First Value: 10  
Enter Second Value: 2  
The result is 5.0  
Program executed successfully
```

```
Output2  
Enter First Value: 10  
Enter Second Value: 'a'  
Please use proper types only
```

```
Output3:  
Enter First Value: 10  
Enter Second Value: 0  
Please enter non-zero for denominator
```

Finally:

Until Python 2.5, the `try` statement came in two flavours. You could use a `finally` block to ensure that code is always executed, or one or more `except` blocks to catch specific exceptions. You couldn't combine both `except` blocks and a `finally` block, because generating the right bytecode for the combined version was complicated and it wasn't clear what the semantics of the combined statement should be.

No matter what happened previously, the *final-block* is executed once the code block is complete and any raised exceptions handled. Even if there's an error in an exception handler or the *else-block* and a new exception is raised, the code in the *final-block* is still run.

Eg:

```
a=eval(input('Enter First Value: '))  
b=eval(input('Enter Second Value: '))  
try:  
    c=a/b  
    print('The result is ', c)  
except ZeroDivisionError:  
    print('Please enter non-zero for denominator')  
except NameError:  
    print('Please use defined names only')  
except TypeError:  
    print('Please use proper types only')  
except:
```

```
print('Error Occurred')
else:
    print('Program executed successfully')
finally:
    print('thank you')

output1:
Enter First Value: 10
Enter Second Value: 4
The result is 2.5
Program executed successfully
thank you
```

```
Output2:
Enter First Value: 10
Enter Second Value: 's'
Please use proper types only
thank you
```

```
Output3:
Enter First Value: 10
Enter Second Value: 0
Please enter non-zero for denominator
thank you
```

Assert: it is used to check the value, if it is satisfied the condition then it will start the executing the flow.

If it is not satisfied the condition, then will return AssertionError exception

The assert statement is intended for debugging statements. It can be seen as an abbreviated notation for a conditional raise statement, i.e. an exception is only raised, if a certain condition is not True.

Without using the assert statement, we can formulate it like this in Python:

```
if not <some_test>:
    raise AssertionError(<message>)
```

Eg:

```
a=eval(input('Enter First Value: '))
b=eval(input('Enter Second Value: '))
```

```
assert(b>0)
try:
    c=a/b
    print('The result is ', c)
except ZeroDivisionError:
    print('Please enter non-zero for denominator')
except NameError:
    print('Please use defined names only')
except TypeError:
    print('Please use proper types only')
except:
    print('Error Occurred')
else:
    print('Program executed successfully')
finally:
    print('thank you')

output:
Enter First Value: 10
Enter Second Value: 0
Traceback (most recent call last):
  File "C:/Users/Narayana/AppData/Local/Programs/Python/Python36-32/ex
ceptionsfile.py", line 3, in <module>
    assert(b>0)
AssertionError
Output2:
Enter First Value: 10
Enter Second Value: 5
The result is 2.0
Program executed successfully
thank you
```

User defined exception:

Some times user can also define exceptions to indicate something is going wrong, those are called customized exceptions or programmatic exceptions.

Developers can write their exceptions as per their requirements, so they are called User defined exceptions.

Exceptions need to be derived from the Exception class, either directly or indirectly.

We use raise keyword to raise user defined exceptions.

Eg1:

```
class TooYoungException(Exception):
    def __init__(self,arg):
        self.arg=arg

class TooOldException(Exception):
    def __init__(self,arg):
        self.arg=arg

age=int(input('Please enter your age: '))

if age<=20:
    raise TooYoungException('You are too young, so wait for some time')
elif age>=40:
    raise TooOldException('You are too old, so dont get marry')
else:
    print('Now you are',age,'. So you can happily marry')
```

Output 1:

```
Please enter your age: 25
Now you are 25 . So you can happily marry
```

Output 2:

```
Please enter your age: 10
TooYoungException: You are too young, so wait for some time
```

Output 3:

```
Please enter your age: 55
TooOldException: You are too old, so dont get marry
```

Eg2:

```
class LessThanZeroException(Exception):
    def __init__(self,arg):
        self.arg=arg

class MoreThanHundredException(Exception):
    def __init__(self,arg):
        self.arg=arg

per=int(input('Please Enter your percentage: '))

if per<0:
    raise LessThanZeroException('You entered less than 0 marks')
elif per >100:
    raise MoreThanHundredException('You eneted more than 100 marks')
else:
    if per<35:
        print('You are failed')
    elif per>=35 and per<50:
        print('You got 3rd class')
    elif per>=50 and per<60:
        print('You got 2nd class')
    elif per>60 and per <75:
        print('You got 1st class')
    else:
        print('You got distinction')
```

output 1:

```
Please Enter your percentage: 10
You are failed
```

output 2:

```
Please Enter your percentage: 50
You got 2nd class
```

output 3:

```
Please Enter your percentage: 88
You got distinction
```

output 4:

```
Please Enter your percentage: -20
LessThanZeroException: You entered less than 0 marks
```

Modules:

If we need to reuse the code then we can define the functions, but if we need to reuse number of functions then we have to go for **modules**

A module is nothing but a file with extension .py which may contains methods, variables and also classes.

Modules are Python .py files that consist of Python code. Any Python file can be referenced as a module. A Python file called **Mymodule.py** has the module name of **Mymodule** that can be imported into other Python files or used on the Python command line interpreter.

Modules are three types:

1. Standard modules
2. User defined modules
3. 3rd party modules

Standard modules: these modules are already defined and kept in python software. So when we install python then automatically these standard modules will install in our machine.

Like math, calendar, os, sys, logging, threading modules..

User defined modules: these modules are defined by users as per their requirement. So here user defined module is nothing but the .py file which contains methods, variables an also classes.

3rd party modules: these modules are already defined by some other people and kept in internet. So we can download and install in our machines by using pip.

pip is a package management system used to install and manage software packages written in Python

Like pymysql, cx_Oracle..

In Python, modules are accessed by using the `import` statement.

When our current file is needed to use the code which is already existed in other files then we can import that file (module).

When Python imports a module called **Mymodule** for example, the interpreter will first search for a built-in module called **Mymodule**. If a built-in module is not found, the Python interpreter will then search for a file named **Mymodule.py** in a list of directories that it receives from the `sys.path` variable.

We can import module in three different ways:

1. Import <module_name>
2. From <module_name> import <method_name>
3. From <module_name> import *

Import <module_name>: this way of importing module will import all methods which are in that specified module.

Eg: `import math`

Here this import statement will import all methods which are available in math module. We may use all methods or may use required methods as per business requirement.

From <module_name> import <method_name>:

Here, this import statement will import a particular method from that module which is specified in the import statement.

We can't use other methods which are available in that module as we specified particular method name in the import statement.

Note: we need to avoid this type of modules as they may cause name clashes in the current python file.

From <module_name>import <*>:

Here, this import statement will import all methods from the specified module and also it will import all other modules which are imported into that specified module_name.

Standard modules

1. Sys module
2. Calendar module
3. Datetime module
4. Math module
5. Os module and so many

Math module

```
>>> import math  
  
>>> math.ceil(10.9)      11  
>>> math.ceil(10.1)      11  
  
>>> math.floor(10.9)     10  
>>> math.floor(10.1)     10  
  
>>> math.fabs(10)        10.0  
>>> math.fabs(-10)       10.0  
>>> math.fabs(-9)        9.0  
  
>>> math.factorial(5)    120  
>>> math.factorial(10)   3628800  
>>> math.factorial(3)    6  
  
>>> math.fmod(10,2)      0.0  
>>> math.fmod(10,3)      1.0  
>>> math.fmod(50,3)      2.0  
>>> math.fmod(19,3)      1.0  
>>> math.fmod(17,3)      2.0  
  
>>> math.fsum([1,2,4])   7.0  
>>> math.fsum([1,2.5,4.5]) 8.0  
  
>>> math.modf(10)        (0.0, 10.0)  
>>> math.modf(7)          (0.0, 7.0)  
>>> math.modf(3)          (0.0, 3.0)  
  
>>> math.trunc(10.89)    10  
>>> math.trunc(7.87234)  7  
>>> math.trunc(0.34)     0  
>>> math.trunc(-3.34)   -3
```

Narayana

PYTHON

Narayana

```
>>> math.exp(2)          7.38905609893065
>>> math.exp(3)          20.085536923187668

>>> math.pow(2,3)         8.0
>>> math.pow(2,-3)        0.125
>>> math.pow(-2,-3)      -0.125
>>> math.pow(-2,3)        -8.0
>>> math.pow(3,3)          27.0

>>> math.sqrt(100)        10.0
>>> math.sqrt(36)          6.0
>>> math.sqrt(49)          7.0

>>> math.sin(1)            0.8414709848078965
>>> math.sin(100)          -0.5063656411097588
>>> math.sin(0.5)          0.479425538604203

>>> math.cos(1)            0.5403023058681398
>>> math.cos(100)          0.8623188722876839
>>> math.cos(0.5)          0.8775825618903728

>>> math.tan(1)             1.5574077246549023
>>> math.tan(100)           -0.5872139151569291
>>> math.tan(0.5)           0.5463024898437905

>>> math.pi                  3.141592653589793
>>> math.e                     2.718281828459045
```

Calendar Module:

calendar.day_name: An array that represents the days of the week in the current locale.

1. Displaying all week names one by one

```
import calendar as c
for i in c.day_name:
    print(i)
```

output:

```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

2. Display week name character by character as per the week index number given by user:

```
import calendar as c
a=input('enter number from 0 to 6:')
for i in c.day_name[a]:
    print(i)
```

output:

```
enter number from 0 to 6:1
T
u
e
s
d
a
y
```

3. Display first character of each day

```
import calendar as c  
  
for i in c.day_name:  
    print(i[0])
```

output:

```
M  
T  
W  
T  
F  
S  
S
```

4. Display all days except 'day' part in each name?

```
import calendar as c  
  
for i in c.day_name:  
    print(i[:-3])  
  
#or  
  
for i in c.day_name:  
    print(i.replace('day', ''))
```

output:
Mon
Tues
Wednes
Thurs
Fri
Satur
Sun

5. Display only 'day' part from each day

```
import calendar as c  
  
for i in c.day_name:  
    print(i[-3:])
```

Output

```
day  
day  
day  
day  
day  
day  
day  
day
```

**6. Display any day as per user requirement, if he enters invalid index number
then display 'invalid index number'?**

```
import calendar as c  
  
num=int(input('Enter index number: '))  
  
a=list(c.day_name)  
  
if num >=0 and num<=6:  
    print(a[num])  
else:  
    print('invalid index number')
```

Output 1:

```
Enter index number: 2  
Wednesday
```

Output 2:

```
Enter index number: -1  
invalid index number
```

Output 3:

```
Enter index number: 10  
invalid index number
```

Output 4:

Enter index number: 6
Sunday

calendar.day_abbr: An array that represents the abbreviated days of the week in the current locale.

Display all week abbreviations

```
import calendar as c
```

```
for i in c.day_abbr:  
    print(i)
```

output

```
Mon  
Tue  
Wed  
Thu  
Fri  
Sat  
Sun
```

calendar.month_name: An array that represents the months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and month_name[0] is the empty string.

```
import calendar as c  
for i in c.month_name:  
    print(i)
```

output

```
January  
February  
March  
April  
May
```

```
June
July
August
September
October
November
December
```

calendar.month_abbr: An array that represents the abbreviated months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and month_abbr[0] is the empty string.

```
import calendar as c
for i in c.month_abbr:
    print(i)

output:
=====
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
```

calendar.monthrange(year, month): Returns weekday of first day of the month and number of days in month, for the specified year and month.

```
import calendar as c  
print(c.monthrange(2018,4))  
print(c.monthrange(2017,2))  
print(c.monthrange(2011,9))
```

output:
(6, 30)
(2, 28)
(3, 30)

calendar.isleap(year): Returns True if year is a leap year, otherwise False.

```
import calendar as c  
print(c.isleap(2019))  
print(c.isleap(2010))  
print(c.isleap(2020))
```

output:
False
False
True

calendar.leapdays(y1, y2): Returns the number of leap years in the range from y1 to y2 (exclusive), where y1 and y2 are years.

```
import calendar as c  
print(c.leapdays(2010,2020))  
print(c.leapdays(2000,2017))  
print(c.leapdays(1988,2018))
```

output:
2
5
8

calendar.weekday(year, month, day): Returns the day of the week (0 is Monday) for year (1970–...), month (1–12), day (1–31).

```
import calendar as c
print(c.weekday(2018,4,18))
print(c.weekday(1988,3,10))
print(c.weekday(1989,8,15))
```

output:

```
2
3
1
```

calendar.weekheader(n): Return a header containing abbreviated weekday names. n specifies the width in characters for one weekday

```
import calendar as c
print(c.weekheader(1))
print(c.weekheader(2))
print(c.weekheader(3))
```

output:

```
M T W T F S S
Mo Tu We Th Fr Sa Su
Mon Tue Wed Thu Fri Sat Sun
```

calendar.monthcalendar(year, month): Returns a matrix representing a month's calendar. Each row represents a week; days outside of the month are represented by zeros. Each week begins with Monday unless set by `setfirstweekday()`.

```
import calendar as c
print(c.monthcalendar(2018,4))
print(c.monthcalendar(2015,6))
```

output:

```
[[0, 0, 0, 0, 0, 0, 1], [2, 3, 4, 5, 6, 7, 8], [9, 10, 11, 12, 13, 14, 15], [16, 17, 18, 19, 20, 21, 22], [23, 24, 25, 26, 27, 28, 29], [30, 0, 0, 0, 0, 0, 0]]
```

```
[[1, 2, 3, 4, 5, 6, 7], [8, 9, 10, 11, 12, 13, 14], [15, 16, 17, 18, 19, 20, 21], [22, 23, 24, 25, 26, 27, 28], [29, 30, 0, 0, 0, 0, 0]]
```

calendar.calendar(year[, w[, l[c]]]): Returns a 3-column calendar for an entire year as a multi-line string using the formatyear() of the TextCalendar class.

```
import calendar as c
print(c.calendar(2018))
```

output:

2018

January	February	March
Mo Tu We Th Fr Sa Su	Mo Tu We Th Fr Sa Su	Mo Tu We Th Fr Sa Su
1 2 3 4 5 6 7	1 2 3 4	1 2 3 4
8 9 10 11 12 13 14	5 6 7 8 9 10 11	5 6 7 8 9 10 11
15 16 17 18 19 20 21	12 13 14 15 16 17 18	12 13 14 15 16 17 18
22 23 24 25 26 27 28	19 20 21 22 23 24 25	19 20 21 22 23 24 25
29 30 31	26 27 28	26 27 28 29 30 31

April	May	June
Mo Tu We Th Fr Sa Su	Mo Tu We Th Fr Sa Su	Mo Tu We Th Fr Sa Su
1	1 2 3 4 5 6	1 2 3
2 3 4 5 6 7 8	7 8 9 10 11 12 13	4 5 6 7 8 9 10
9 10 11 12 13 14 15	14 15 16 17 18 19 20	11 12 13 14 15 16 17
16 17 18 19 20 21 22	21 22 23 24 25 26 27	18 19 20 21 22 23 24
23 24 25 26 27 28 29	28 29 30 31	25 26 27 28 29 30
30		

July	August	September
Mo Tu We Th Fr Sa Su	Mo Tu We Th Fr Sa Su	Mo Tu We Th Fr Sa Su
1	1 2 3 4 5	1 2
2 3 4 5 6 7 8	6 7 8 9 10 11 12	3 4 5 6 7 8 9
9 10 11 12 13 14 15	13 14 15 16 17 18 19	10 11 12 13 14 15 16
16 17 18 19 20 21 22	20 21 22 23 24 25 26	17 18 19 20 21 22 23
23 24 25 26 27 28 29	27 28 29 30 31	24 25 26 27 28 29 30
30 31		

October	November	December
Mo Tu We Th Fr Sa Su	Mo Tu We Th Fr Sa Su	Mo Tu We Th Fr Sa Su
1 2 3 4 5 6 7	1 2 3 4	1 2
8 9 10 11 12 13 14	5 6 7 8 9 10 11	3 4 5 6 7 8 9
15 16 17 18 19 20 21	12 13 14 15 16 17 18	10 11 12 13 14 15 16
22 23 24 25 26 27 28	19 20 21 22 23 24 25	17 18 19 20 21 22 23
29 30 31	26 27 28 29 30	24 25 26 27 28 29 30
		31

Sys module:

Eg1:

```
import sys  
print('The command line arguments are:')  
for i in sys.argv:  
    print(i)  
  
output  
The command line arguments are:  
C:/Users/Narayana/AppData/Local/Programs/Python/Python36-32/sysmod.py
```

Eg2:

```
import sys  
print('\n\nThe PYTHONPATH is', sys.path, '.\n')  
  
output  
The PYTHONPATH is ['C:/Users/Narayana/AppData/Local/Programs/Python/Python36-32',  
'C:\\\\Users\\\\Narayana\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32\\\\Lib\\\\idlelib',  
'C:\\\\Users\\\\Narayana\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32\\\\python36.zip',  
'C:\\\\Users\\\\Narayana\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32\\\\DLLs',  
'C:\\\\Users\\\\Narayana\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32\\\\lib',  
'C:\\\\Users\\\\Narayana\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32',  
'C:\\\\Users\\\\Narayana\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python36-32\\\\lib\\\\site-packages'] .
```

Eg3:

```
import sys  
user_input = sys.stdin.readline()  
print("Input : " + user_input)
```

Eg4:

```
import sys  
print(sys.copyright)  
  
output  
Copyright (c) 2001-2018 Python Software Foundation.  
All Rights Reserved.  
  
Copyright (c) 2000 BeOpen.com.  
All Rights Reserved.  
  
Copyright (c) 1995-2001 Corporation for National Research Initiatives.  
All Rights Reserved.
```

Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.

Eg5:

```
import sys  
print("Python Narayana")  
sys.exit(1)  
print("Hello")  
  
output  
Python Narayana
```

Eg6:

```
import sys  
variable = "Python Narayana"  
print(sys.getrefcount(0))  
print(sys.getrefcount(variable))  
print(sys.getrefcount(None))  
  
output  
955  
3  
6874
```

Datetime module:**Eg1:**

```
import datetime  
dt=datetime.date(2018,8,26)  
print(dt)  
print(dt.year)  
print(dt.month)  
print(dt.day)  
  
output  
2018-08-26  
2018  
8  
26
```

Eg2:

```
import datetime  
dt1=datetime.time(10,11,12,236587)  
print(dt1)  
print(dt1.hour)  
print(dt1.minute)  
print(dt1.second)  
print(dt1.microsecond)  
  
output  
10:11:12.236587  
10  
11  
12  
236587
```

Eg3:

```
import datetime  
dt2=datetime.datetime(2018,5,28,12,23,34,567893)  
print(dt2)  
print(dt2.year)  
print(dt2.month)  
print(dt2.day)  
print(dt2.hour)  
print(dt2.minute)  
print(dt2.second)  
print(dt2.microsecond)  
  
output  
2018-05-28 12:23:34.567893  
2018  
5  
28  
12  
23  
34  
567893
```

Unit testing

1. unittest is the batteries-included test module in the Python standard library.
2. A testing unit should focus on one tiny bit of functionality and prove it correct.
3. Each test unit must be fully independent. Each test must be able to run alone, Try hard to make tests that run fast. If one single test needs more than a few milliseconds to run, development will be slowed down or the tests will not be run as often as is desirable.
4. Learn your tools and learn how to run a single test or a test case. Then, when developing a function inside a module, run this function's tests frequently.
5. Always run the full test suite before a coding session, and run it again after. This will give you more confidence that you did not break anything in the rest of the code.
6. If you are in the middle of a development session and have to interrupt your work, it is a good idea to write a broken unit test about what you want to develop next.
7. When coming back to work, you will have a pointer to where you were and get back on track faster.
8. The first step when you are debugging your code is to write a new test pinpointing the bug.
9. Use long and descriptive names for testing functions

Python supports different inbuilt methods,

1. `assertEqual()`
2. `assertNotEqual()`
3. `assertTrue()`
4. `assertFalse()`
5. `assertIs()`
6. `assertIsNot()`
7. `assertIsNone()`
8. `assertIsNotNone()`
9. `assertIn()`
10. `assertNotIn()`
11. `assertIsInstance()`
12. `assertNotIsInstance()`

Eg1:

```
import unittest  
def fun(x):  
    return x + 1  
  
class MyTest(unittest.TestCase):  
    def test(self):  
        self.assertEqual(fun(3), 4)  
  
unittest.main()
```

output:

```
Ran 1 test in 0.016s  
OK
```

Eg2:

```
import unittest  
def sum1(m,n):  
    return m+n  
def diff1(m,n):  
    return m-n  
def mul1(m,n):  
    return m*n  
  
class Myunit(unittest.TestCase):  
    def test_sum1(self):  
        self.assertEqual(sum1(2,3),5)  
    def test_diff1(self):  
        self.assertEqual(diff1(4,2),2)  
    def test_mul1(self):  
        self.assertEqual(mul1(2,3),6)  
    def test_casetesting(self):  
        st='python'  
        self.assertTrue(st.islower(),True)  
    def test_casetesting(self):  
        st1='PYtTHON'  
        self.assertFalse(st1.isupper(),False)  
  
#for testing all test cases of the class  
unittest.main()
```

Output1:

```
Ran 4 tests in 0.016s
```

```
OK
```

Eg3:

```
import unittest
def sum1(m,n):
    return m+n
def diff1(m,n):
    return m-n
def mul1(m,n):
    return m*n

class Myunit(unittest.TestCase):
    def test_sum1(self):
        self.assertEqual(sum1(2,3),6)
    def test_diff1(self):
        self.assertEqual(diff1(4,2),2)
    def test_mul1(self):
        self.assertEqual(mul1(2,3),8)
    def test_casetesting(self):
        st='python'
        self.assertTrue(st.islower(),True)
    def test_casetesting(self):
        st1='PYtHON'
        self.assertFalse(st1.isupper(),False)
unittest.main()
```

Output:

```
FAIL: test_mul1 (__main__.Myunit)
```

```
AssertionError: 6 != 8
=====

```

```
FAIL: test_sum1 (__main__.Myunit)
```

```
AssertionError: 5 != 6
Ran 4 tests in 0.016s
FAILED (failures=2)
```

```
We can also test a specific test rather than all tests.
```

Eg4:

```
import unittest
def sum1(m,n):
    return m+n

class Myunit(unittest.TestCase):
    def test_sum1(self):
        self.assertEqual(sum1(2,3),6)
    def test_casetesting(self):
        st='python'
        self.assertTrue(st.islower(),True)
    def test_casetesting(self):
        st1='PYtHON'
        self.assertFalse(st1.isupper(),False)

obj=Myunit()
obj.test_sum1
```

Eg5:

```
import unittest
def sum1(m,n):
    return m+n

class Myunit(unittest.TestCase):
    def test_sum1(self):
        self.assertEqual(sum1(2,3),8)
    def test_casetesting(self):
        st='python'
        self.assertTrue(st.islower(),True)
    def test_casetesting(self):
        st1='PYtHON'
        self.assertFalse(st1.isupper(),False)

obj=Myunit()
obj.test_sum1()
```

output:

```
AssertionError: 5 != 8
```

Iterators:

Iterator in python is used with a 'for in loop'. Python lists, tuples, dicts and sets are all examples of inbuilt iterators. These types are iterators because they implement following methods. In fact, any object that wants to be an iterator must implement following methods.

1. `__iter__` method is used to start new iteration, this method is called on initialization of an iterator. This should return an object that has a `next` or `__next__` (in Python 3) method.
2. `__next__` method should return the next value for the iterable. When an iterator is used with a 'for in' loop, the for loop implicitly calls `next()` on the iterator object. This method should raise a `StopIteration` to signal the end of the iteration.

Eg1: Iterating over string object:

```
st='Python Narayana'  
for i in st:  
    print(i)
```

Output:

```
P  
y  
t  
h  
o  
n  
  
N  
a  
r  
a  
y  
a  
n  
a
```

Eg2: Iterating over list object

```
lst=[10,True,'Venkat',10.5,20+5j]  
for i in lst:  
    print(i)
```

output:

```
10
True
Venkat
10.5
(20+5j)
```

Eg3: Iterating over tuple object

```
tup=(1,False,'Siva',4+6j,1.8)
for i in tup:
    print(i)
```

output:

```
1
False
Siva
(4+6j)
1.8
```

Eg4: Iterating over set object

```
set1={10,'Narayana','Venkat','Siva',False,20.8}
for i in set1:
    print(i)
```

output:

```
10
Venkat
Siva
Narayana
20.8
```

Eg5: iterating over dict object

```
dict1={'id':101,'Name':'Venkat','Company':'TCS','Sal':20000}
for i in dict1:
    print(i)
```

output:

```
id  
Name  
Company  
Sal
```

Generators:

Generators are very easy to implement, but a bit difficult to understand.

Generators are used to create iterators, but with a different approach. Generators are simple functions which return an iterable set of items, one at a time, in a special way.

When an iteration over a set of item starts using the for statement, the generator is run. Once the generator's function code reaches a "yield" statement, the generator yields its execution back to the for loop, returning a new value from the set. The generator function can generate as many values (possibly infinite) as it wants, yielding each one in its turn.

Here is a simple example of a generator function which returns 7 random integers:

```
Eg:  
import random  
  
def lottery():  
    # returns 6 numbers between 1 and 40  
    for i in range(6):  
        yield random.randint(1, 40)  
  
    # returns a 7th number between 1 and 15  
    yield random.randint(1,15)  
  
for random_number in lottery():  
    print("And the next number is... %d!" %(random_number))
```

Decorators:

A decorator is a function that takes another function and returns a newer and prettier version of that function.

The decorator will not modify the existing functionality of other functions, it will just add new functionality to the existing functionality of the functions.

Decorators concept is the most beautiful and most powerful design possibilities in Python, but at the same time the concept is considered by many as complicated to get into. To be precise, the usage of decorators is very easy, but writing decorators can be complicated, especially if you are not experienced with decorators and some functional programming concepts.

Eg1:

```
def dec_fun(func):          #decorator function
    def inner():
        print("Hello, I am decorator function and i will data to the display()")
        func()
    return inner

@dec_fun
def display():              #ordinary function
    print("I am ordinary function and i am going to dec_fun() to take new functionality from dec_fun() and i will come back with decorator data")

display()                  #ordinary function call
```

Output:

```
Hello, I am decorator function and i will data to the display()
I am ordinary function and i am going to dec_fun() to take new functionality from dec_fun() and i will come back with
decorator data.
```

Eg2:

```
def smart_divide(func):
    def inner(a,b):
        print("I am going to divide",a,"and",b)
        if b == 0:
            print("Whoops! cannot divide")
        return
```

```
return func(a,b)
return inner

@smart_divide
def divide(a,b):
    return a/b

Output1:
>>> divide(3,6)
I am going to divide 3 and 6
0.5

Output2:
>>> divide(15,0)
I am going to divide 15 and 0
Whoops! cannot divide

Output3:
>>> divide(4,10)
I am going to divide 4 and 10
0.4

Output4:
>>> divide(12,4)
I am going to divide 12 and 4
3.0

Eg3:

import time

def decor(func):
    def wrap(*args):
        start=time.time()
        result=func(*args)
        end=time.time()
        print(func.__name__,"has taken",end-start)
        return result
    return wrap

@decor
def sqrs(m):
    x=[]
```

Narayana

PYTHON

Narayana

```
# start=time.time()
for i in m:
    x.append(i*i)
# end=time.time()
# print('sqrs has taken',end-start)
return x

@decor
def cubs(m):
    y=[]
    # start=time.time()
    for i in m:
        y.append(i*i*i)
    # end=time.time()
    # print('cube has taken',end-start)
    return y

@decor
def fours(m):
    z=[]
    # start=time.time()
    for i in m:
        z.append(i*i*i*i)
    # end=time.time()
    # print('fours has taken',end-start)
    return z

lst=range(1000000)
a=sqrs(lst)
#print(a)

b=cubs(lst)
#print(b)

c=fours(lst)
#print(c)

Output
sqrs has taken 0.3657684326171875
cubs has taken 0.5173401832580566
fours has taken 0.5259397029876709
```

Multi Threading:

Multithreading is nothing but running multiple operations parallel.

Running several threads is similar to running several different programs concurrently, but with the following benefits –

- Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.
- Threads sometimes called light-weight processes and they do not require much memory overhead; they are cheaper than processes.

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running.

Eg1:

```
import threading
class Thread1(threading.Thread):
    def run(self):
        i=1
        while i<=10:
            print(i)
            i=i+1

class Thread2(threading.Thread):
    def run(self):
        j=11
        while j<=20:
            print(j)
            j=j+1

class Thread3(threading.Thread):
    def run(self):
        k=21
        while k<=30:
            print(k)
```

```
k=k+1
```

```
obj1=Thread1()  
obj2=Thread2()  
obj3=Thread3()  
  
obj1.start()  
obj2.start()  
obj3.start()
```

output

```
11121  
21222  
31323  
41424  
51525  
61626  
71727  
81828  
91929  
102030
```

Eg2:

```
import threading  
  
def print_cube(num):  
    print("Cube: {}".format(num * num * num))  
  
def print_square(num):  
    print("Square: {}".format(num * num))  
  
if __name__ == "__main__":  
  
    t1 = threading.Thread(target=print_square, args=(10,))  
    t2 = threading.Thread(target=print_cube, args=(10,))  
  
    t1.start()  
    t2.start()
```

Narayana

PYTHON

Narayana

```
t1.join()  
t2.join()  
  
print("Done!")  
  
output:  
Square: 100Cube: 1000  
  
Done!
```

Database connection

• Step 1: Open mysql

• Step 2: Enter password 'root' to login to the mysql database

• Enter password: ****

• Welcome to the MySQL monitor. Commands end with ; or \g.

• Your MySQL connection id is 1

• Server version: 5.0.41-community-nt MySQL Community Edition (GPL)

• Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

• Step 3: To see list of databases

```
mysql> show databases;
```

```
+-----+
```

```
| Database      |
```

```
+-----+
```

```
| information_schema |
```

```
| mysql        |
```

```
| test         |
```

```
+-----+
```

```
3 rows in set (0.01 sec)
```

• Step 4: Creating user defined database

```
mysql> create database pythondb;
```

```
Query OK, 1 row affected (0.00 sec)
```

• Step 5: To see list of databases

```
mysql> show databases;
```

```
+-----+
```

```
| Database      |
```

```
+-----+
```

```
| information_schema |
```

```
| mysql        |
```

```
| pythondb     |
```

```
| test         |
```

```
+-----+
```

```
4 rows in set (0.00 sec)
```

• Using a specific database:

```
mysql> use pythondb;  
Database changed
```

Creating a table in pythondb:

```
mysql> create table pytab(id int,  
                           name varchar(10),  
                           loc varchar(10),  
                           sal int);  
Query OK, 0 rows affected (0.11 sec)
```

To see table description:

```
mysql> desc pytab;  
+-----+-----+-----+-----+  
| Field | Type   | Null | Key | Default | Extra |  
+-----+-----+-----+-----+  
| id   | int(11) | YES  |   | NULL    |       |  
| name | varchar(10)| YES |   | NULL    |       |  
| loc  | varchar(10)| YES |   | NULL    |       |  
| sal  | int(11)  | YES |   | NULL    |       |  
+-----+-----+-----+-----+  
4 rows in set (0.02 sec)
```

Inserting data into the table

```
mysql> insert into pytab values(1,'Narayana','Hyd',1000);  
Query OK, 1 row affected (0.08 sec)
```

```
mysql> insert into pytab values(2,'Satya','Hyd',1500);  
Query OK, 1 row affected (0.06 sec)
```

```
mysql> insert into pytab values(3,'Anu','Bang',1100);  
Query OK, 1 row affected (0.06 sec)
```

```
mysql> insert into pytab values(4,'Krishna','Chennai',1200);  
Query OK, 1 row affected (0.06 sec)
```

```
mysql> insert into pytab values(5,'Chandu','Chennai',1100);  
Query OK, 1 row affected (0.06 sec)
```

To display table data:

```
mysql> select * from pytab;
+----+-----+-----+-----+
| id | name  | loc   | sal  |
+----+-----+-----+-----+
| 1  | Narayana | Hyd   | 1000 |
| 2  | Satya   | Hyd   | 1500 |
| 3  | Anu     | Bang  | 1100 |
| 4  | Krishna  | Chennai | 1200 |
| 5  | Chandu  | Chennai | 1100 |
+----+-----+-----+
5 rows in set (0.00 sec)
```

Installing Pymysql:

Goto command prompt,

1. C:\Users\Narayana>cd..
2. C:\Users>cd..
3. C:\>cd C:\Python27\Scripts
4. C:\Python27\Scripts>pip install matplotlib

Collecting matplotlib

5. C:\Python27\Scripts>pip install pymysql
6. Collecting pymysql
Using cached PyMySQL-0.7.11-py2.py3-none-any.whl
Installing collected packages: pymysql
Successfully installed pymysql-0.7.11

1. Open python
2. Open new file and write the below code

```
import pymysql
connect=pymysql.Connection(host='localhost',user='root',password='root',db='pythondb')
conn=connect.cursor()
sql="select * from pytab;"
conn.execute(sql)
d=conn.fetchall()
print(d)
```

output:

```
((1, 'Narayana', 'Hyd', 1000), (2, 'Satya', 'Hyd', 1500), (3, 'Anu', 'Bang', 1100), (4, 'Krishna', 'Chennai', 1200), (5, 'Chandu', 'Chennai', 1100))
```

To get all the rows one by one:

```
import pymysql
connect=pymysql.Connection(host='localhost',user='root',password='root',db='pythondb')
conn=connect.cursor()
sql="select * from pytab;"
conn.execute(sql)
d=conn.fetchall()
for i in d:
    print(i)
```

output

```
(1, 'Narayana', 'Hyd', 1000)
(2, 'Satya', 'Hyd', 1500)
(3, 'Anu', 'Bang', 1100)
(4, 'Krishna', 'Chennai', 1200)
(5, 'Chandu', 'Chennai', 1100)
```

To get all employee names:

```
import pymysql
connect=pymysql.Connection(host='localhost',user='root',password='root',db='pythondb')
conn=connect.cursor()
sql="select * from pytab;"
conn.execute(sql)
d=conn.fetchall()
for i in d:
    print(i[1])
```

output:

```
Narayana
Satya
Anu
Krishna
Chandu
```

To get all the details of employee 'Narayana'

```
import pymysql
connect=pymysql.Connection(host='localhost',user='root',password='root',db='pythondb')
conn=connect.cursor()
sql="select * from pytab where name='Narayana';"
conn.execute(sql)
d=conn.fetchall()
for i in d:
    print(i)

output:
(1, 'Narayana', 'Hyd', 1000)
```

To get all employee names who are working in 'Chennai'

```
import pymysql
connect=pymysql.Connection(host='localhost',user='root',password='root',db='pythondb')
conn=connect.cursor()
sql="select name from pytab where loc='Chennai';"
conn.execute(sql)
d=conn.fetchall()
for i in d:
    print(i)

output:
('Krishna',)
('Chandu',)

Or
```

```
import pymysql
connect=pymysql.Connection(host='localhost',user='root',password='root',db='pythondb')
conn=connect.cursor()
sql="select * from pytab where loc='Chennai';"
conn.execute(sql)
d=conn.fetchall()
for i in d:
    print(i[1])

output:
Krishna
Chandu
```

Fetchone(): this function is used to fetch the first record from the table

Eg1:

```
import pymysql
connect=pymysql.Connection(host='localhost',user='root',password='root',db='pythondb')
conn=connect.cursor()
sql="select * from pytab;"
conn.execute(sql)
d=conn.fetchone()
print(d)

output:
(1, 'Narayana', 'Hyd', 1000)
```

Eg2:

```
import pymysql
connect=pymysql.Connection(host='localhost',user='root',password='root',db='pythondb')
conn=connect.cursor()
sql="select * from pytab where loc='Chennai';"
conn.execute(sql)
d=conn.fetchone()
print(d)

output:
(4, 'Krishna', 'Chennai', 1200)
```

Fetchmany(n): this function is used to fetch n number of records

Eg1:

```
import pymysql
connect=pymysql.Connection(host='localhost',user='root',password='root',db='pythondb')
conn=connect.cursor()
sql="select * from pytab;"
conn.execute(sql)
d=conn.fetchmany(3)
for i in d:
    print(i)

output:
(1, 'Narayana', 'Hyd', 1000)
(2, 'Satya', 'Hyd', 1500)
(3, 'Anu', 'Bang', 1100)
```

Eg2:

```
import pymysql
connect=pymysql.Connection(host='localhost',user='root',password='root',db='pythondb')
conn=connect.cursor()
sql="select * from pytab where loc='Chennai';"
conn.execute(sql)
d=conn.fetchmany(2)
for i in d:
    print(i)
```

output:

```
(4, 'Krishna', 'Chennai', 1200)
(5, 'Chandu', 'Chennai', 1100)
```

Numpy:

1. The Python programming language was not initially designed for numerical computing, but attracted the attention of the scientific/engineering community early on,
2. A special interest group called matrix-sig was founded in 1995 with the aim of defining an array computing package. Among its members was Python designer/maintainer [Guido van Rossum](#), who implemented extensions to make array computing easier.
3. NumPy is an open source extension module for Python.
4. The module NumPy provides fast precompiled functions for numerical routines.
5. It adds support to Python for large, multi-dimensional arrays and matrices.
6. It also supplies a large library of high-level mathematical functions to operate on these arrays.
7. SciPy (Scientific Python) extends the functionalites of NumPy with further useful functions.
8. Both NumPy and SciPy are usually not installed by default. NumPy has to be installed before installing SciPy.
9. NumPy is based on two earlier Python array packages. Those are Numeric and Numarray.
10. NumPy is a merger of those two, i.e. it is build on the code of Numeric and the features of Numarray.
11. One of the main advantages of NumPy is its advantage in time compared to standard Python.

Installation steps:

1. Install numpy and opencv 2-4.13
2. Goto opencv folder C:\Users\Narayana\Downloads\opencv\build\python\2.7\x86 and copy cv2 file.
3. Goto python folder C:\Python27\Lib\site-packages and paste that cv2 file here.
4. Goto command prompt C:\Users\Narayana>c:\Python27\python.exe and run it.
5. Import numpy

Benefits of numpy over List:

1. Requires less memory
2. Runs very fast
3. Convenient to developer

Requires less memory:

```
import numpy as np
import sys
import time
list1=range(1000)
print("the memory for list1 is ", sys.getsizeof(1)*len(list1))

array1=np.arange(1000)
print("the memory for array is ", array1.size*array1.itemsize)

output:

the memory for list1 is  12000
the memory for array is  4000
```

runs very fast

```
import numpy as np
import sys
import time

list1=list(range(100000))
list2=list(range(100000))

arr1= np.array(list1)
arr2=np.array(list2)

start=time.time()
result=[x+y for x,y in zip(list1,list2)]
print("Python list took: ",(time.time()-start)*1000)

start=time.time()
result=arr1+arr2
print("numpy took: ",(time.time()-start)*1000)

output

Python list took: 62.99996376037598
numpy took: 0.9999275207519531
```

Very convenient for developer

We can't add elements of two lists directly, but two arrays can be added by using numpy

Eg:

```
import numpy as np  
  
arr1=np.array([1,2,3,4,5])  
arr2=np.array([10,20,30,40,50])  
  
add_result=arr1+arr2  
print(add_result)  
  
sub_result=arr2-arr1  
print(sub_result)  
  
mul_result=arr1*arr2  
print(mul_result)
```

```
import numpy as np
```

#creating one dimensional array

```
a=np.array([1,2,3])  
print(a)  
print(a.ndim)  
print(a.itemsize)  
print(a.shape)  
print(a.dtype)
```

output:

```
[1 2 3]  
1  
4  
(3,)  
int32
```

```
#creating two dimensional array (2*3 matrix)
```

```
a=np.array([[1,2,3],[2,3,4]])  
print(a)  
print(a.ndim)  
print(a.itemsize)  
print(a.shape)  
print(a.dtype)
```

```
output:
```

```
[[1 2 3]  
 [2 3 4]]  
2  
4  
(2, 3)  
int32
```

```
#creating 4*2 matrix
```

```
a=np.array([[1,2],[2,3],[3,4],[4,5]])
```

```
print(a)  
print(a.ndim)  
print(a.itemsize)  
print(a.dtype)  
print(a.shape)
```

```
output:
```

```
[[1 2]  
 [2 3]  
 [3 4]  
 [4 5]]  
2  
4  
int32  
(4, 2)
```

```
#creating array with float type data
a=np.array([[1,2],[2,3],[3,4],[4,5]],dtype=np.float64)
print(a)
print(a.ndim)
print(a.itemsize)
print(a.dtype)
print(a.shape)

output:
[[ 1.  2.]
 [ 2.  3.]
 [ 3.  4.]
 [ 4.  5.]]
2
8
float64
(4, 2)
```

```
#creating matrix with complex numbers
a=np.array([[1,2],[2,3],[3,4],[4,5]],dtype=np.complex)
print(a)
print(a.ndim)
print(a.itemsize)
print(a.dtype)
print(a.shape)

output
[[ 1.+0.j  2.+0.j]
 [ 2.+0.j  3.+0.j]
 [ 3.+0.j  4.+0.j]
 [ 4.+0.j  5.+0.j]]
2
16
complex128
(4, 2)
```

```
#Creating matrix with placeholder number:
```

```
Eg 1:
```

```
a=np.zeros((2,3))  
print(a)
```

```
output:
```

```
[[ 0.  0.  0.]  
 [ 0.  0.  0.]]
```

```
Eg 2:
```

```
a=np.ones((2,3))  
print(a)
```

```
output:
```

```
[[ 1.  1.  1.]  
 [ 1.  1.  1.]]
```

```
#Creating a list with range function
```

```
l=range(5)  
print(l)  
print(l[0])  
print(l[1])  
print(l[2])  
print(l[3])  
print(l[4])
```

```
output:
```

```
[0, 1, 2, 3, 4]  
0  
1  
2  
3  
4
```

```
#Creating an array with arrange method
```

```
a=np.arange(1,5)
print(a)
print(a[0])
print(a[1])
print(a[2])
print(a[3])
```

```
Output:
```

```
[1 2 3 4]
1
2
3
4
```

```
a=np.arange(1,10,2)
print(a)
print(a[0])
print(a[1])
print(a[2])
print(a[3])
print(a[4])
```

```
Output:
```

```
[1 3 5 7 9]
1
3
5
7
9
```

```
#creating an array with linspace  
  
Eg1:  
  
a=np.linspace(1,5,10)  
print(a)  
  
output:  
  
[ 1. 1.44444444 1.88888889 2.33333333 2.77777778 3.22222222  
 3.66666667 4.11111111 4.55555556 5. ]  
  
Eg 2:  
  
a=np.linspace(1,5,5)  
print(a)  
  
output  
  
[ 1. 2. 3. 4. 5.]  
  
  
#Reshaping array  
  
a=np.array([[1,2,3],[2,3,4],[4,5,6],[5,6,7]])  
print(a)  
print(a.shape)  
  
output:  
  
[[1 2 3]  
 [2 3 4]  
 [4 5 6]  
 [5 6 7]]  
  
(4, 3)  
  
print(a.reshape(3,4))  
  
[[1 2 3 2]  
 [3 4 4 5]  
 [6 5 6 7]]
```

Narayana

PYTHON

Narayana

```
print(a.reshape(2,6))
[[1 2 3 2 3 4]
 [4 5 6 5 6 7]]
print(a.reshape(6,2))
[[1 2]
 [3 2]
 [3 4]
 [4 5]
 [6 5]
 [6 7]]
#platten array
a=np.array([[1,2,3],[2,3,4],[4,5,6],[5,6,7]])
print(a)
print(a.ravel())
print(a)
output
[[1 2 3]
 [2 3 4]
 [4 5 6]
 [5 6 7]]
[1 2 3 2 3 4 4 5 6 5 6 7]
[[1 2 3]
 [2 3 4]
 [4 5 6]
 [5 6 7]]
#Aggregate operations
a=np.array([[1,2,3],[2,3,4],[4,5,6],[5,6,7]])
print(a.max())
print(a.min())
print(a.sum())
print(a)
```

Narayana

PYTHON

Narayana

```
print(a.sum(axis=0))  
print(a.sum(axis=1))
```

output:

```
7  
1  
48  
[[1 2 3]  
 [2 3 4]  
 [4 5 6]  
 [5 6 7]]  
  
[12 16 20]  
  
[ 6 9 15 18]
```

#arithmetic operations

```
a1=np.array([[1,2,3],[2,3,4]])  
  
a2=np.array([[4,5,6],[3,4,5]])  
  
print(a1)  
  
[[1 2 3]  
 [2 3 4]]  
  
print(a2)  
  
[[4 5 6]  
 [3 4 5]]  
  
print(a1+a2)  
  
[[5 7 9]  
 [5 7 9]]
```

Narayana

PYTHON

Narayana

```
print(a1-a2)
[[[-3 -3 -3]
 [-1 -1 -1]]
print(a1*a2)
[[ 4 10 18]
 [ 6 12 20]]
print(a1/a2)
[[[0 0 0]
 [0 0 0]]
print(a1%a2)
[[1 2 3]
 [2 3 4]]
```

Assignment - 4

1. What is the meaning of data hiding?
2. What is the main purpose of inheritance? Types of inheritance?
3. What is the difference between data abstraction and encapsulation?
4. What is method overriding and method overloading?
5. What is the difference between static and class variables?
6. What is docstring in python?
7. How to print specified docstring in the output?
8. What are the different modes in file handling?
9. What is the default mode in the file handling?

10. What is seek and tell methods?

11. How add new data to the existing file which contains data already?

12. Write the data "Python developer" to the empty file?

13. How to add "Oracle Developer" to the same above file (this should be second line)?

14. How to find vowels in the string in file?

15. How to count number of words in the file?

16. What is purpose of logging?

17. What are the different logging levels?

18. What is the default logging level?

19. How to print the data in logging when it is DEBUG level?

20. Can we use logging.debug when the logging level set to WARNING?

21. How to create a log file?

22. What is exception?

23. How to handle exceptions in python?

24. What is the 'else' in exception hadling? Write a small program by using else?

25. What is the 'finally'? write a small program by using finally?

26. How to use multiple exception classes in single except?

27. What is assert?

28. What is multithreading?
29. What is numpy?
30. How it is better than python list? Give any three reasons?
31. How to create an array by using list?
32. Create a two dimensional array?
33. We use range() to create a list in python, then what do we use in numpy?
34. Create a 3*4 array?
35. Create a 2*4 array float values?
36. How to see shape of an array?
37. What is item size for numeric element in array?

38. How to create an array for place holder 1?

39. How to reshape an array? Give an example?

40. How to a string "Python" as character by character from file?

Python FAQS

Part-1

1. What is the diff between title() and capitalize()?
2. How to check whether the given string in lower case or not?
3. How to know how many characters are existed in the given string?
4. What is a split()? and what is the default delimiter in the split()
5. What is the output format of split()?
6. Is insertion order preserved or not in string?
7. How to reverse the given string?
8. How to get ascending order of the given string?
9. How to get descending order of the given string?
10. How to display the given string 'python' as 'p..y..t..h..o..n'
11. Is string mutable or immutable?
12. Can we delete a specific character by using del command?
13. Can we clear a string with clear()?
14. How to perform a string packing?
15. What is the slicing operator?
16. How to replace a part of string with new string?
17. How to display the given string 'NaRaYaNa' as 'nArAyAnA'?
18. What is a list?
19. Is a list mutable or immutable object?
20. How can we decide a list is a mutable object?
21. List is a dynamic object, why?
22. How to know how many 10s are available in the list [10,20,10,10,30]
23. How to know the index number of a specific element in the list?
24. What is the indexing and slicing?
25. What is list packing and list unpacking?
26. How to add a new element to the existing list?
27. What is the difference between append() and extend()?
28. How to add a new element at required place in the list?
29. Can we add multiple elements by using append()?
30. How to add a sub list to the existing list object?
31. What is the difference between remove() and pop()?
32. How to know the highest value in the list?
33. How to convert a list into string?
34. How to convert a string into list?
35. How to remove all elements from the list?
36. Can we delete a specific element by using del command?
37. What are the different ways to create a list?

38. What is the range() and what is the syntax of range()?
39. What is the mandatory arg in the range()?
40. What is the difference between python2 and python3 in case of range()?
41. What is the datatype of range() in python3?
42. What is the a tuple?
43. What is the difference between tuple and list?
44. Is tuple mutable or immutable object?
45. Is insertion order of tuple elements preserved or not?
46. Can we give duplicate elements in the tuple?
47. What is all() and any()?
48. How to create a tuple with single element?
49. Can we create a tuple with using () and also tuple()?
50. Can we delete a specific element by using clear() in the tuple?
51. How to convert a tuple into list? and a list into tuple?
52. When do we choose a tuple and when do we choose a list?
53. Can we use a list in the tuple or not?
54. Can we use a tuple in the list or not?
55. How to find sum of al elements in the tuple?
56. Can we use heterogeneous elements in the tuple?
57. What is the set?
58. Is a set allow duplicate elements or not?
59. Is insertion order of set elements is preserved or not?
60. Is a set dynamic object or not?
61. Is a set mutable or immutable object?
62. How to create a empty set?
63. How to represent an empty set?
64. How to add an element to the existing set?
65. Can we add a duplicate element to the existing set?
66. What is the difference between remove() and discard()?
67. What is the main purpose of set?
68. What is the main purpose of tuple?
69. A list has so many duplicate elements, how to remove all duplicate elements from the list?
70. What are the operators that we use to perform membership opeartions?
71. What is the difference between issuperset() and issubset()?
72. In which case both issuperset() and issubset() will return True?
73. In which case both issuperset() and issubset() will return False?
74. If both sets are having completely uncommon elements, then what is the result of isdisjoint()?
75. What are the different set operations that we perform in set?
76. What is the difference between list and set?
77. How to convert a set into list?
78. How to convert a set into string?
79. What is a dict?

80. What is the main purpose of dict?
81. How to update the values of a dict?
82. How to retrieve the values by using keys?
83. How to remove a specific key:value pair?
84. What is the difference between pop() and popitem()?
85. How to get all keys for dict?
86. How to update the dict by using another dict?
87. How to get all key:value pairs as items?
88. How to get all values from dict?
89. How to create a dict by using tuple?
90. How to remove all key:value pairs from the dict?
91. Can we use a set in the list and tuple?
92. Can we use a list or tuple in the set?
93. Can we convert a heteroginous list in to string?
94. Can we split a list by using split()?
95. How to reverse a tuple()?
96. How to display the descending order of tuple elements?
97. Is a dict mutable or immutable object?
98. Is a dict allow duplicate keys or not?
99. Can we update keys in the dict?

Part - 2

1. What are the identity operators and membership operators
2. What is the difference between bitwise leftshift and bitwise rightshift operators
3. What is diff b/w actual arguments and formal arguments?
4. What are the positional arguments?
5. What is keyword arguments and nonkeyword args?
6. What is the difference between *args and **kwargs?
7. What is enumerate()?
8. What is difference between def function and lambda function?
9. What are the different types of functions which are used with lambda functions?
10. What is a filter() and purpose?
11. What is a map() and purpose?
12. What is a reduce function() and purpose?
13. Why do we use conditional statements in our application?
14. What is the differnce between elif and nested if statement?
15. How to display 10th table by using for loop?
16. What is the difference between for loop and while loop?
17. How to iterate a list by using for loop?
18. What is continue and break statements?

19. What is the meaning of comprehension?
20. What are the different types of comprehensions available in python?
21. How to read data from text file?
22. How to get last line from the file?
23. How to count number lines in the file?
24. How to count number of words in the file?
25. How to get first word from all the lines in the file?
26. How to take a specific page data from the pdf file?
27. What is the difference between seek() and tell()?
28. What is the difference between readline() and readlines()?
29. What are the different types of modes available in the file operations?
30. What is the default mode in the file operations?
31. What is the difference between append(a) and write(w)modes?
32. What are the different oops concepts?
33. What is the data abstraction?
34. What is the inheritance? and types?
35. Which type of inheritance you used in your last project?
36. What is polymorphism?
37. What is the difference between method overloading and overriding?
38. Can we implement method overloading completely in python?
39. What is the class and object?
40. What are the different tyoes of variables in python?
41. What is a module? types of modules?
42. List out some 3rd party modules?
43. How to install 3rd party modules in the python?
44. What is pip? and purpose of pip?
45. What are the different ways to import modules in python?
46. Can we use members of other module, without importing the module?
47. What is exception? how to handle exception?
48. Can we implement user defined exceptions in python?
49. What is the difference between else and finally blocks?
50. What is the logging? purpose of logging?
51. What are the different levels of logging?
52. What is the default log level in the python?
53. What is the extension of log file?
54. What is the iterator? and one example?
55. What is the genetor? and one example?
56. What is the decorator? and one example?
57. How to perform unit testing in python?
58. Which modules we use to perform unit testing in python?
59. Which database you used in your last project?
60. What is the module that we use to get data from Oracle database?

61. How did you connect to your database in your project?
 62. What is the difference between numpy arrays and python list?
 63. Write a python function to **swap** two numbers?
 64. Write a python function to check whether a number is **prime** or not?
 65. Write a python function to find **factorial** of a number?
 66. Write a python function to check whether a number is **palindrome** or not?
 67. Write a python function to check whether a number if **Armstrong** or not.
 68. Write a python function to check whether a number if **Even or Odd**.
 69. Write a python function to get **Fibonacci** series?
 70. Write a python function to check whether a number is even or odd?
 71. Write a python function to check whether a number is positive or negative?
 72. Write a python function to find max value of three given values?
 73. Write a python function to check whether the given number is divisible by 5
 74. Write a python function to reverse the given number?
 75. What is comprehension?

Part-3:

1. Which of the following is valid identifier?
 - a. 1var
 - b. py
 - c. def
 - d. tot\$!
 2. tuple is the _____ version of list
 - a. read only
 - b. read and write
 - c. write only
 - d. none of the above
 3. What is the output of “nani”+3 ?
 - a. nani10
 - b. TypeError
 - c. naninaninanani
 - d. NameError
 4. What is the output of “python”*2.5?
 - a. pythonpythonpyt
 - b. TypeError
 - c. pythonpython
 - d. Name Error

5. What is the output of $3/2*4+3+(10/5)**3-2$

- a. 15.0
- c. 20
- b. 19.0
- d. 10.0

6. From math import pi

```
print(math.pi) ?
```

- a. 3.1415
- c. TypeError
- b. NameError
- d. no output

7. If a=13.485

- a. ceil(a) =
- b. floor(a)=
- c. trunc(a)=
- d. round(a)=

8. Which function is used to read data from user?

- a. print()
- c. type()
- b. input()
- d. readinput()

9. What is the output of **print()**?

- a. TypeError
- c. No Data
- b. one new line
- d. NameError

10. What is the output of **print('Hello\tpython')**?

- a. Hello\tpython
- c. Hello python
- b. PrintError
- d. '\` is not allowed in a string

11. a,b,c=10,20,30 then how to display 10 20 30 (separate by space)?

```
12. Print('Hello')
    print('Python')
    print(Developer)
```

Output:

Hello
Python
Developer

But how to get **Hello Python Developer**

13. the result of split method is:

- a. list
- b. set
- c. tuple
- d. dict

14. which function will manipulate the list and return some value?

- a. remove()
- b. pop()
- c. clear()
- d. extend()

15. pop function will remove the last element in the list, for example it's a empty list then pop returns?

- a. None
- b. NameError
- c. IndexError
- d. []

16. lst=[2,13,1,'a',True], then output of **lst.sort()**?

- a. 'a',1,True,2,13
- b. TypeError
- c. 1,True, 2, 13,'a'
- d. IndexError

17. lst=[10,20,30] then what is the output of a in a=lst+[100]?

- a. Error
- b. 10,20,30
- c. 100
- d. 10,20,30,100

18. What is the value of c in

```
a='p'  
b='d'  
c=a,b  
print(c)  
  
a. 'pd'  
b. ('p','d')  
c. ['p','d']  
d. {'p'.'d'}
```

19. If a=10.6 and b=int(a) then what is the value of b?

- a. 10.6
- b. 10
- c. 0
- d. Error

20. Which of the following function checks in a string that all characters are whitespaces?

- a. islower()
- b. isspace()
- c. isnumeric()
- d. istitle()

21. What is the output of L[1:] if L = [1,2,3]?

- a. 2,3
- b. 3
- c. 2
- d. None of the above.

22. Which of the following function convert a sequence of tuples to dictionary in python?

- a. set(x)
- b. frozenset(s)
- c. dict(d)
- d. chr(x)

23. Which of the following function checks in a string that all characters are digits?

- a. shuffle(lst)
- b. isalnum()
- c. capitalize()
- d. isdigit()

24. What is the following function reverses objects of list in place?

- a. list.reverse()
- b. list.pop(obj=list[-1])
- c. list.sort([func])
- d. list.remove(obj)

25. Which of the following function of dictionary gets all the values from the dictionary?

- a. getvalues()
- b. values()
- c. value()
- d. None of the above.

26. Which of the following operator in python evaluates to true if the variables on either side of the operator point to the same object and false otherwise?

- a. **
- b. Is
- c. //
- d. not in

27. What is the output of print tuple[0] if tuple = ('abcd', 786 , 2.23, 'john', 70.2)?

- a. ('abcd', 786 , 2.23, 'john', 70.2)
- b. Error
- c. abcd
- d. a

28. Which of the following function checks in a string that all characters are alphanumeric?

- a. shuffle(lst)
- b. isalnum()
- c. capitalize()
- d. isdigit()

29. What is the output of L[-2] if L = [1,2,3]?

- a. 1
- b. 3
- c. 2
- d. 0

30. Which of the following statement causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating?

- a. break
- b. pass
- c. continue
- d. None of the above

31. Which of the following function checks in a string that all characters are numeric?

- a. `islower()`
- b. `isspace()`
- c. `isnumeric()`
- d. `istitle()`

32. Which of the following function convert a String to an object in python?

- a. `repr(x)`
- b. `tuple(s)`
- c. `eval(str)`
- d. `list(s)`

33. what is the result of the following function

```
def fast (items= []):  
    items.append (1)  
    return items  
  
print fast ()  
print fast ()
```

34. what is the result of the following

```
a = 'abcdefghijkl'  
print (a[:3] + a[3:])
```

35. How would you produce a list with unique elements from a list with duplicate elements?

```
dups = ['a','b','c','d','d','d','e','a','b','f','g','g','h']
```

36. What is the datatype of 'words' variable in the following

```
wordList='1,3,2,4,5,3,2,1,4,3,2'.split(',')
```

- a. Int
- b. List
- c. string
- d. tuple

37. What is the output of the following,

```
print int("1") + 1
```

38. What is the output of the following,

```
"abcd"[2:]
```

39. What is the output of the following,

```
str1 = 'hello'  
str1[-1:]
```

40. What arithmetic operators cannot be used with strings ?

- a. *
- b. -
- c. +
- d. none of the above

41. What is the output of the following,

```
print(r"\nhello")
```

What is the output of the following,

```
print('new' 'line')
```

42. What is the output of the following,

```
str1="helloworld"
```

- a. str1[:]
- b. str1[::]
- c. str1[-1:]
- d. str1[-1::]
- e. str1[::-1]

43. What will be the output of the following code Lst=['a','b','c','d','e']?

```
Print lst[10:] ?
```

44. Sum of all elements in a list ([1,2,3,...,100] with one line of code.

45. How to sum all odd numbers upto 100?

46. What will be the output of the following code

```
a = 1  
a, b = a+1, a+1  
print a  
print b
```

47. Create a new list that converts the following list of number strings to a list of numbers.

```
strs = ['1','21','53','84','50','66','7','38','9']
```

Thank You