A Nutshell of Python

The beginners Syllabus

Anoopkumar Muttathil

KTU-RLMCA 369 2020 MCA

Unit 1

Introduction to Python: Features of Python, How to Run Python, Identifiers, Reserved Keywords, Variables, Input, Output and Import Functions, Operators
Data Types: Numbers, Strings, List, Tuple, Set, Dictionary, Data Type Conversions.
Decision Making, Loops, Nested Loops, Control Statements, Types of Loops

Python Features

Python is one of the most commonly used programming languages today and is an easy language for beginners to learn because of its readability. It is a free, open-source programming language with extensive support modules and community development, easy integration with web services, user-friendly data structures, and GUI-based desktop applications.
Python is a dynamic, high level, free open source and interpreted programming language. It supports object-oriented programming as well as procedural oriented programming.
In Python, we don't need to declare the type of variable because it is a dynamically typed language.
For example, x = 10
Here, x can be anything such as String, int, etc.

Features in Python

There are many features in Python, some of which are discussed below –

1. **Easy to code:**
   Python is a high-level programming language. Python is very easy to learn the language as compared to other languages like C, C#, JavaScript, Java, etc. It is very easy to code in python language and anybody can learn python basics in a few hours or days. It is also a developer-friendly language.
2. **Free and Open Source:**
   Python language is freely available at the official website and you can download it from the given download link below click on the **Download Python** keyword.
   Download Python at *https://www.python.org/downloads/*
   Since it is open-source, this means that source code is also available to the public. So you can download it as, use it as well as share it.
3. **Expressive Language**
   Python can perform complex tasks using a few lines of code. A simple example, the hello world program you simply type print (("Hello World"). It will take only one line to execute, while Java or C takes multiple lines.

4. **Embeddable**
   The code of the other programming language can use in the Python source code. We can use Python source code in another programming language as well. It can embed other language into our code.
5. **Cross-platform**
   Python can run equally on different platforms such as Windows, Linux, UNIX, and Macintosh, etc. So, we can say that Python is a portable language. It enables programmers to develop the software for several competing platforms by writing a program only once.
6. **Interpreted and interactive**
   Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.python has an option namely interactive mode which allows interactive testing and debugging of code.
7. **Object-Oriented Language:**
   One of the key features of python is Object-Oriented programming. Python supports object-oriented language and concepts of classes, objects encapsulation, etc.
8. **GUI Programming Supported:**
   Graphical User interfaces can be made using a module such as PyQt5, PyQt4, wxPython, or Tk in python.
   PyQt5 is the most popular option for creating graphical apps with Python.
9. **High-Level Language:**
   Python is a high-level language. When we write programs in python, we do not need to remember the system architecture, nor do we need to manage the memory.
10. **Portable**
    Due to its open-source nature, Python has been ported (i.e. changed to make it work on) to many many platforms. All your Python programs will work on any of these platforms without requiring any changes at all. You can use Python on Linux, Windows, Macintosh, Solaris, OS/2, Amiga, AROS, AS/400, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE and PocketPC !
11. **Scalable**
    Python provides a better structure and support for large programs than shell scripting. It can be used as a scripting language or can be compiled to bytecode (platform independent code) for building large applications.
12. **Extensible feature:**
    Python is an **Extensible** language. We can write us some Python code into C or C++ language and also we can compile that code in C/C++ language.
13. **Python is Portable language:**
    Python language is also a portable language. For example, if we have python code for windows and if we want to run this code on other platforms such as

Linux, Unix, and Mac then we do not need to change it, we can run this code on any platform.

14. **Python is Integrated language:**
    Python is also an Integrated language because we can easily integrated python with other languages like c, c++, etc.
15. **Interpreted Language:**
    Python is an Interpreted Language because Python code is executed line by line at a time. like other languages C, C++, Java, etc. there is no need to compile python code this makes it easier to debug our code. The source code of python is converted into an immediate form called **bytecode**.
16. **Large Standard Library**
    Python has a large standard library which provides a rich set of module and functions so you do not have to write your own code for every single thing. There are many libraries present in python for such as regular expressions, unit-testing, web browsers, etc.
17. **Dynamically Typed Language:**
    Python is a dynamically-typed language. That means the type (for example- int, double, long, etc.) for a variable is decided at run time not in advance because of this feature we don't need to specify the type of variable.

Python IDEs

1) PyCharm

PayCharm is a cross-platform IDE used for Python programming. This editor can be used on Windows, macOS, and Linux. This software contains API that can be used by the developers to write their own Python plugins so that they can extend the basic functionalities.

Price: Free

2) Spyder

Spyder is a scientific integrated development environment written in Python. This software is designed for and by scientists who can integrate with Matplotlib, SciPy, NumPy, Pandas, Cython, IPython, SymPy, and other open-source software. Spyder

is available through Anaconda (open-source distribution system) distribution on Windows, macOS, and Linux.

Price: Free

3) IDLE

IDLE (Integrated Development and Learning Environment) is a default editor that comes with Python. This software helps a beginner to learn Python easily. IDLE software package is optional for many Linux distributions. The tool can be used on Windows, macOS, and Unix.

Price: free

4) Visual Studio Code

Visual Studio Code (VS Code) is an open-source environment developed by Microsoft. This IDE can be used for Python development. Visual Studio Code is based on Electron which is a framework to deploy Node JS applications for the computer running on the Blink browser engine.

Price: Free

5) Atom

Atom is a useful code editor tool preferred by programmers due to its simple interface compared to the other editors. Atom users can submit packages and them for the software.

Price: Free

6) Jupyter

Jupyter is a tool for people who have just started with data science. It is easy to use, interactive data science IDE across many programming languages that just not work as an editor, but also as an educational tool or presentation.

Price: Free

7) Thonny

Thonny is an IDE for learning and teaching programming, specially designed with the beginner Pythonista scripting environment. It is developed at The University of Tartu, which you can download for free on the Bitbucket repository for Windows, Linux, and Mac.

Price: Free

The Easiest Way to Run Python

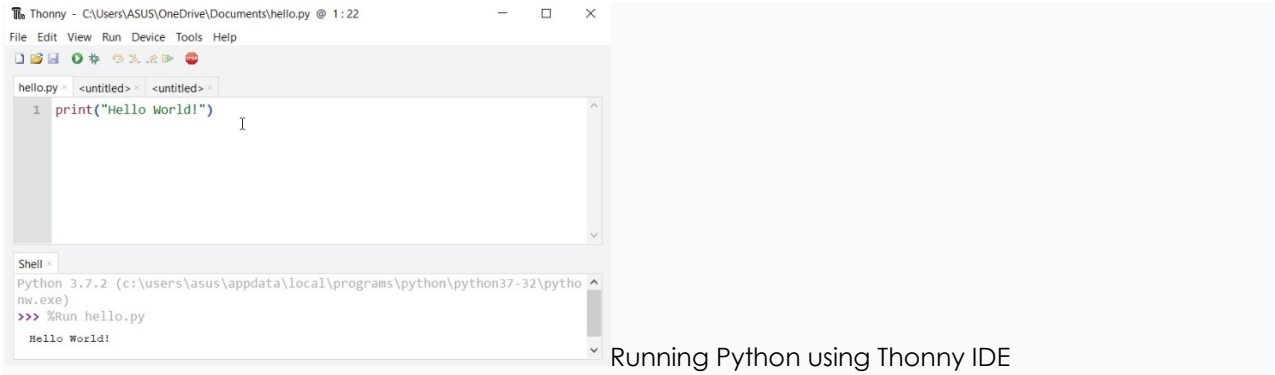
The easiest way to run Python is by using **Thonny IDE**.

The Thonny IDE comes with the latest version of Python bundled in it. So you don't have to install Python separately.

Follow the following steps to run Python on your computer.

1. Download Thonny IDE.
2. Run the installer to install **Thonny** on your computer.
3. Go to: **File** > **New**. Then save the file with `.py` extension. For example, `hello.py`, `example.py`, etc.

   You can give any name to the file. However, the file name should end with **.py**
4. Write Python code in the file and save it.



Running Python using Thonny IDE

5. Then Go to **Run** > **Run current script** or simply click **F5** to run it.

How to Run Python

There are three different ways to start python.

a)Using interactive interpreter

You can start Python from Unix,Dos or any other System that provides u a command line interpreter or shell window.Get into the command line of Python.
For windows/dos it is c:>Python
It brings up the following prompt
>>>
Type the following text at the python prompt and press enter
>>>print (("RLMCA369 programming in python")
The result will be
RLMCA369 programming in python

b)Script from the command line

This method invokes the interpreter with a script parameter which begin the execution of the script and continues until the script is finished. When the script is finished ,the interpreter is no longer active.
Ex:
Write a simple python program in a script named first.py file. Type the following code in the source code in the file

Print (("Programming in Python")

Then execute it using the command

C:> python first.py
The result will be
programming in python

c)Integrated development Environment

U can run python from a graphical user interface (GUI) environment ,if u have a GUI application on ur system that supports Python.**IDLE** is the integerated Development Environment for unix and **PythonWin** is the first windows interface for python.

Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. Python is case sensitive and hence uppercase and lowercase letters are considered distinct.

1.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).

2.Reserved keywords cannot be used as an identifier.

3.Identifiers cannot start with a digit.

4.special symbols like @,!,#,% etc cannot be used in an identifier.

5.Identifiers can be of any length.

Valid identifiers

Total max_mark count2 student

Naming conventions

- **Class names** start with an uppercase letter. All other identifiers start with a lowercase letter.
  Ex:Person
- Starting an identifier with a single leading underscore indicates that the identifier is **private.**
  Ex: _sum
- Starting an identifier with two leading underscores indicates a **strongly private identifier.**
  Ex: __sum
- If the identifier also ends with two trailing underscores, the identifier is a language-defined **special name**.
  Ex: __foo__

Reserved Keywords

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords except True,False,None contain lowercase letters only.

and	exec	not
	False	**None**
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
		True
del	import	try

elif	in	while
else	is	with
except	lambda	yield

To retrieve the keywords in Python following code can be given

```
>>> import keyword
>>> print(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', 'retu
rn', 'try', 'while', 'with', 'yield']
>>>
```

Comments in Python

Single-line comments are created simply by beginning a line with the hash (#) character, and they are automatically terminated by the end of line.

#This would be a comment in Python

Comments that span multiple lines – used to explain things in more detail – are created by adding a delimiter (''') on each end of the comment.

```
''' This would be a multiline comment
in Python that spans several lines and
describes your code, your day, or anything you want it to
…
'''
```

Multiline Statements

We can make a statement extend over multiple lines with the line contuation character(\)

For example

```
Gd=first+\
Second+\
Third
```

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

```
Num=[3,4,5,
8,9]
```

Is a valid statement.

Quotes in Python

Python accepts single(‘),double(“) and triple(''') quotes to denote string literals. The same type of quote should be used to start and end the string. The triple quotes are used to span the string across multiple lines.

Word='single word'
Word="this is short"
Par=''' this para
Spanning multiple lines'''

Input output and import functions

Displaying output

The function used to print (output on a screen is the print (statement where u can pass zero or more expressions separated by commas. The print (function converts the expressions u pass into a string and writes the result to a std output .

The actual syntax of the print() function is
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

- Here, objects is the value(s) to be printed.
- The sep separator is used between the values. It defaults into a **space** character.
- After all values are printed, end is printed. It defaults into a **new line**.
- The file is the object where the values are printed and its default value is **sys.stdout (screen)**.
- The flush determines whether the output stream needs to be flushed for any waiting output. A true value forcibly flushes the stream.

```
>>> print(1,2,3,4)
1 2 3 4
>>> print(1,2,3,4,sep="*")
1*2*3*4
>>> print(1,2,3,4,end="$")
1 2 3 4$
>>>
```

```
>>> print("the value is %d and the result id %5.2d" %(23,34.456))
the value is 23 and the result id    34
>>> print("the value is %d and the result id %5.2f" %(23,34.456))
the value is 23 and the result id 34.46
>>> print("the firstname is %s and the second name is %5s"
%("Anoopkumar","Muttathil"))
the firstname is Anoopkumar and the second name is Muttathil
```

Reading the input

Python provides built in functions to read one line of text from std input which by default comes from the keyboard.These functions are is input.

The input Function

The function presents a prompt to the user ,gets input from the user and returns the data input by the user in a string. For example,

```
name = input("What is your name? ")
print ("Hello, %s." % name)
```

import function

when the program grows bigger or when there are segments of code that is frequently used ,it can be stored in different modules.A module is file containing python definitions and statements.Python modules have a filename and end with the extension .py.Definitions inside a module can be imported to another module or the interactive interpreter in Python. We use the import keyword to do this .we can import the math module by typing in import math.

```
>>> import math
>>> math.pi
3.141592653589793
```

Operators

Arithmetic operators: Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication and division.

Operator	*Description*	*Syntax*
+	Addition: adds two operands	x + y
-	Subtraction: subtracts two operands	x - y
*	Multiplication: multiplies two operands	x * y
/	Division (float): divides the first operand by the second	x / y
//	Division (floor): divides the first operand by the second	x // y
%	Modulus: returns the remainder when first operand is divided by the second	x % y
**	Performs Exponential (power) calculation on operands	x**y

```
>>> a=3
>>> b=2
>>> print(a+b)
5
>>> print(a-b)
1
>>> print(a*b)
6
>>> print(a/b)
1.5
>>> print(a//b)
1
>>> print(a**b)
9
>>> print(5//2)
2
>>> print(a%b)
1
```

Relational Operators: Relational operators compares the values. It either returns **True** or **False** according to the condition.

Operator	*Description*	*Syntax*
>	Greater than: True if left operand is greater than the right	x > y
<	Less than: True if left operand is less than the right	x < y
==	Equal to: True if both operands are equal	x == y
!=	Not equal to - True if operands are not equal	x != y
>=	Greater than or equal to: True if left operand is greater than or equal to the right	x >= y
<=	Less than or equal to: True if left operand is less than or equal to the right	x <= y

```
>>> a=10
>>> b=20
>>> print(a>b)
False
>>> print(a<b)
True
>>> print(a!=b)
True
>>> print(a==b)
False
>>> print(a<=b)
True
>>> print(a>=b)
False
>>>
>>>
```

Logical operators: Logical operators perform **Logical AND**, **Logical OR** and **Logical NOT** operations.

Operator	*Description*	*Syntax*
and	Logical AND: True if both the operands are true	x and y
Or	Logical OR: True if either of the operands is true	x or y
not	Logical NOT: True if operand is false	not x

```
>>> a,b=10,20
>>> print(a>b and b>10)
False
>>> print(a>b or b>10)
True
>>> print(not a>b)
True
>>>
```

Bitwise operators: Bitwise operators acts on bits and performs bit by bit operation.

Operator	*Description*	*Syntax*
&	Bitwise AND	x & y
|	Bitwise OR	x | y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x>>
<<	Bitwise left shift	x<<

```
>>> a=3
>>> b=2
>>> print(a&b)
2
>>> print(a|b)
3
>>> print(a^b)
1
>>> print(~a)
-4
>>> print(a<<1)
6
>>> print(a>>1)
1
```

Assignment operators: Assignment operators are used to assign values to the variables.

Operator	*Description*	*Syntax*
=	Assign value of right side of expression to left side operand	x = y + z
+=	Add AND: Add right side operand with left side operand and then assign to left operand	a+=b a=a+b
-=	Subtract AND: Subtract right operand from left operand and then assign to left operand	a-=b a=a-b
*=	Multiply AND: Multiply right operand with left operand and then assign to left operand	a*=b a=a*b
/=	Divide AND: Divide left operand with right operand and then assign to left operand	a/=b a=a/b
%=	Modulus AND: Takes modulus using left and right operands and assign result to left operand	a%=b a=a%b
//=	Divide(floor) AND: Divide left operand with right operand and then assign the value(floor) to left operand	a//=b a=a//b
**=	Exponent AND: Calculate exponent(raise power) value using operands and assign value to left operand	a**=b a=a**b
&=	Performs Bitwise AND on operands and assign value to left operand	a&=b a=a&b
|=	Performs Bitwise OR on operands and assign value to left operand	a|=b a=a|b
^=	Performs Bitwise xOR on operands and assign value to left operand	a^=b a=a^b
>>=	Performs Bitwise right shift on operands and assign value to left operand	a>>=b a=a>>b
<<=	Performs Bitwise left shift on operands and assign value to left operand	a <<= b a= a << b

```
>>> b=20
>>> a=10
>>> a+=b
>>> print(a)
30
>>> a-=b
```

```
>>> print(a)
10
>>>
>>>
```

Identity operators

is and **is not** are the identity operators both are used to check if two values are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

```
>>> a=3
>>> b=3
>>> print(a is b)
True
>>> a="hai"
>>> b="hai"
>>> print(a is b)
True
>>> a=[1,2,3]
>>> b=[1,2,3]
>>> print(a is b)
False
```

Membership operators

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

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

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

```
>>> a={'a','b'}
>>> print('a' in a)
True
>>> print(1 in a)
False
>>> b=[5,6,7,8]
>>> print(5 in b)
True
```

Operator Precedence in python

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

DataTypes

Every value in Python has a datatype. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.
There are various data types in Python. Some of the important types are listed below.

Numbers

Number datatypes store numeric values.Number objects are created when u assign a value to them.
Ex:a=1,b=20
U can also delete the reference to a number object by using the del statement.The syntax of the del statement is
Del var1[,var2,var3……]
Ex: del a,b
Python supports 4 different numerical types.
int can be represented in binary, **octal** and **hexadecimal**(0b1010,0o123,0x12a)
long(can be represented in binary, octal and **float**

complex

Integers can be of any length, it is only limited by the memory available.
A floating point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. 1 is integer, 1.0 is floating point number.

Complex numbers are written in the form, x + yj, where x is the real part and *y* is the imaginary part.

```
>>> a = 1234567890123456789
>>> a
1234567890123456789
>>> b = 0.1234567890123456789
>>> b
0.12345678901234568
>>> c = 1+2j
>>> c
(1+2j)
```

We can use the type() function to know which class a variable or a value belongs to and the isinstance() function to check if an object belongs to a particular class.

```
>>> a=20
>>> print(type(a))
<class 'int'>
>>> a=2.3
>>> print(type(a))
<class 'float'>
>>> a=2+6j
>>> print(type(a))
<class 'complex'>
>>> print(isinstance(2,int))
True
```

Mathematical Functions

Python provides various mathematical functions to perform mathematical calculations.The functions except abs(x), min(x1,x2,x3,…),max(x1,x2,x3,..),round(x[,n]),pow(x,y)need to import the math module.

Function	*Description*
abs(x)	**returns the absolute value of x**
min(x1,x2,…)	**returns the minimum value from the arguments**
max(x1,x2,…)	**returns the maximum value from the arguments**
round(x[,y])	**x is rounded to n digits**
pow(2,3)	**finds x raised to y**
sqrt(x)	**Finds the square root of x**
ceil(x)	**Finds the smallest integer >x**
floor(x)	**Finds the largest integer <=x**
exp(x)	**Returns e^x**
fabs(x)	**returns the absolute value of x** math.fabs(number) will always return a floating point number even if the argument is integer, whereas abs() will return a floating point or an integer depending upon the argument.
log(x)	**Finds the natural logarithm of x x>0**
log10(x)	**Finds the logarithm to the base 10 x>0**
modf(x)	**Returns the integer and decimal part as a tuple,The integer part is returned as a decimal**

```
>>> abs(2)
2
>>> min(3,2,9,10)
2
>>> max(4,3,98,1)
98
>>> round(2.36,1)
2.4
>>> pow(2,3)
8
>>> import math

>>> math.sqrt(25)
5.0
>>> math.ceil(2.7)
3
>>> math.floor(2.4)
2
>>> math.exp(10)
22026.465794806718
>>> math.log(10)
2.302585092994046
>>> math.log10(10)
1.0
>>> math.modf(2.3)
(0.2999999999999998, 2.0)
>>> math.fabs(-2)
2.0
```

Trignometric functions

Function	*Description*
sin(x)	Returns the sin of x in Radians
cos(x)	Returns the cos of x in Radians
tan(x)	Returns the tan of x in Radians
atan(x)	Returns the arc tangent of x in Radians
hypot(x,y)	Returns sqrt(x^2+y^2)
degrees(x)	converts the angle x from Radians to degrees
radians(x)	converts the angle x from degrees to radians

```
>>> math.sin(30)
-0.9880316240928618
>>> math.cos(30)
0.15425144988758405
>>> math.tan(30)
-6.405331196646276
>>> math.degrees(180)
10313.240312354817
>>> math.radians(30)
0.5235987755982988
>>> math.atan(30)
1.5374753309166493
>>> math.atan2(30,2)
1.5042281630190728
>>> math.hypot(2,3)
3.6055512754639896
>>> math.degrees(30)
1718.8733853924696
>>> math.radians(60)
1.0471975511965976
```

Random Number Functions

Choice(seq) returns a random value from a sequence

```
>>> l=[3,2,6,5,9]
>>> random.choice(l)
```

5

Shuffle(list) shuffles the item randomly in a list.returns none

```
>>> random.shuffle(l)
>>> l
[2, 5, 9, 6, 3]
```

Random() returns a random floating point number which lies between 0 and 1

```
>>> import random
>>> random.random()
0.3017362669382584
```

Randrange(start,stop,step)returns a randomly selected number from a range where start shows the starting of the range,stop shows the end of the range and step decides the number to be added to decide a random number.

```
>>> random.randrange(2,10,2)
6
```

Seed(x)gives the starting value for generating a random number.

```
>>> random.seed(10)
>>> random.random()
0.5714025946899135
```

```
>>> random.seed(10)
>>> random.random()
0.5714025946899135
```

Uniform(x,y) generates a random floating point number n such that n>x and n<y

```
>>> random.uniform(2,7)
4.144445273375573
>>>
```

Strings

is sequence of Unicode characters. We can use single quotes or double quotes to represent strings. Multi-line strings can be denoted using triple quotes, ''' or """.

```
>>>print("""Comments multiline 1""")
      """ Comments multiline """
      ''' Comments multiline '''
      print("""Comments multiline 2""")
```

Strings are immutable. Like list and tuple, slicing operator [ ] can be used with string. String index starting at 0 in the beginning of the string and ending at -1.The **+ sign** is the string concatenation operator and the **asterisk(*)** is the repetition operator.

```
>>> a="Welcome"
>>> print(a)
Welcome
>>> print(a[0])
W
>>> print(a[0:4])
Welc
```

```
>>> print(a[2:])
lcome
>>> print(a*2)
WelcomeWelcome
>>> print(a+"Anoopkumar")
WelcomeAnoopkumar
>>>
```

The % operator is used for string formatting.

>>> print("the firstname is %s and the second name is %9.4s ok" %("Anoopkumar","Muttathil"))

the firstname is Anoopkumar and the second name is Mutt ok

Escape Characters

Is a character that gets interpreted when placed in single or quotes.They are represented using backslash notation.These characters are non printable.

Ex

\a	Bell or alert
\b	backspace
\n	new line
\r	carriage return
\t	tab

String formatting operator

This operator is unique to strings and is similar to the formatting operations of the printf() functions of the c programming language.

Format Symbols	***Conversion***
%c	Character
%s	String
%i	Signed decimal integer
%d	Signed decimal integer
%o	Octal integer
%x	Hexadecimal Number(Lowercase letters)
%X	Hexadecimal Numbers(Upper case letters)
%e	Exponential notation with lowercase letter e
%E	Exponential notation with Uppercase letter E
%f	Floating point number

>>> print("The first character of %s is %c"%("Album","A"))
The first character of Album is A
>>> print("The sum of %d and %i is %f"%(3,4,9))
The sum of 3 and 4 is 9.000000
>>>print("The Octal equivalent of %d is %o"%(23,23))
The Octal equivalent of 23 is 27
>>> print("hexa of 283 is %x %X"%(283,283))
hexa of 283 is 11b 11B
>>> print("The exponent notation of 23.456 is %e"%23.456)
The exponent notation of 23.456 is 2.345600e+01

String Formatting functions

max()

The method **max()** returns the max alphabetical character from the string *str*.

```
str = "this is really a string example....wow!!!";
print ("Max character: " + max(str))
str = "this is a string example....wow!!!";
print ("Max character: " + max(str))
```

When we run above program, it produces following result −

```
Max character: y
Max character: x
```

min()

The method **min()** returns the min alphabetical character from the string *str*.

```
str = "this-is-real-string-example....wow!!!"
print ("Min character: " + min(str))
str = "this-is-a-string-example....wow!!!"
print ("Min character: " + min(str))
```

When we run above program, it produces following result −

```
Min character: !
Min character: !
```

len()

The method **len()** returns the length of the string.

```
str = "this is string example....wow!!!"
print ("Length of the string: ", len(str))
```

When we run above program, it produces following result −

```
Length of the string:  32
```

Built-in String Methods

replace(old,new,max)

The method **replace()** returns a copy of the string in which the occurrences of *old* have been replaced with *new*, optionally restricting the number of replacements to *max*.

```
str = "this is string example....wow!!! this is really string"
print (str.replace("is", "was"))
print (str.replace("is", "was", 3))
```

When we run above program, it produces following result −

```
thwas was string example....wow!!! thwas was really string
thwas was string example....wow!!! thwas is really string
```

count(str, beg= 0,end=len(string))

Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.

```
```

```
str = "this is string example....wow!!!";

sub = "i";
print ("str.count(sub, 4, 40) : ", str.count(sub, 4, 40))
sub = "wow";
print ("str.count(sub) : ", str.count(sub))
```

Result

```
str.count(sub, 4, 40) :  2
str.count(sub) :  1
```

expandtabs(tabsize=8)
Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.

```
str = "this is\tstring example....wow!!!";
print ("Original string: " + str)
print ("Defualt exapanded tab: " +  str.expandtabs())
print ("Double exapanded tab: " +  str.expandtabs(16))
```

Result

```
Original string: this is	string example....wow!!!
Defualt exapanded tab: this is string example....wow!!!
Double exapanded tab: this is	string example....wow!!!
```

join(seq)
Merges (concatenates) the string representations of elements in sequence **seq** into **a** string, with separator string.

```

s = "-";
seq = ("a", "b", "c"); # This is sequence of strings.
print (s.join( seq ))
```

When we run above program, it produces following result −

```
a-b-c
```

---

split(str="", num=string.count(str))
Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.

```
str = "Line1-abcdef \nLine2-abc \nLine4-abcd";
print (str.split( ))
print (str.split(' ', 1 ))
```

When we run above program, it produces following result −

```
['Line1-abcdef', 'Line2-abc', 'Line4-abcd']
['Line1-abcdef', '\nLine2-abc \nLine4-abcd']
```

splitlines(num=string.count('\n'))

Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.

```
str = "Line1-a b c d e f\nLine2- a b c\n\nLine4- a b c d";
print (str.splitlines( ))
print (str.splitlines( 0 ))
print (str.splitlines( 3 ))
print (str.splitlines( 4 ))
print (str.splitlines( 5 ))
```

When we run above program, it produces following result −

```
['Line1-a b c d e f', 'Line2- a b c', '', 'Line4- a b c d']
['Line1-a b c d e f', 'Line2- a b c', '', 'Line4- a b c d']
['Line1-a b c d e f\n', 'Line2- a b c\n', '\n', 'Line4- a b c d']
['Line1-a b c d e f\n', 'Line2- a b c\n', '\n', 'Line4- a b c d']
['Line1-a b c d e f\n', 'Line2- a b c\n', '\n', 'Line4- a b c d']
```

find(str, beg=0 end=len(string))

Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.

```
#!/usr/bin/python

str1 = "this is string example....wow!!!";
str2 = "exam";

print (str1.find(str2))
print (str1.find(str2, 10))
print (str1.find(str2, 40))
```

Result

```
15
15
-1
```

rfind(str, beg=0,end=len(string))

Same as find(), but search backwards in string.

```
str1 = "this is really a string example....wow!!!";
```

```
str2 = "is";
print (str1.rfind(str2))
print (str1.rfind(str2, 0, 10))
print (str1.rfind(str2, 10, 0))
print (str1.find(str2))
print (str1.find(str2, 0, 10))
print (str1.find(str2, 10, 0))
```

When we run above program, it produces following result −

```
5
5
-1
2
2
-1
```

index(str, beg=0, end=len(string))

Same as find(), but raises an exception if str not found.

```
str1 = "this is string example....wow!!!";
str2 = "exam";
print (str1.index(str2))
print (str1.index(str2, 10))
print (str1.index(str2, 40))
```

Result

```
15
15
Traceback (most recent call last):
  File "test.py", line 8, in
  print (str1.index(str2, 40))
ValueError: substring not found
shell returned 1
```

rindex(str, beg=0, end=len(string))

Same as index(), but search backwards in string.

```

str1 = "this is string example....wow!!!";
str2 = "is";

print (str1.rindex(str2)
print (str1.index(str2))
```

When we run above program, it produces following result −

```
5
```

```
2
```

Lower()

Returns a copy of the string in which uppercase characters are converted to lower case

```
str = "THIS IS STRING EXAMPLE....WOW!!!";
print (str.lower())
```

When we run above program, it produces following result −

```
this is string example....wow!!!
```

upper()

Converts lowercase letters in string to uppercase.

```
str = "this is string example....wow!!!";
print ("str.capitalize() : ", str.upper())
```

When we run above program, it produces following result −

```
str.capitalize() :  THIS IS STRING EXAMPLE....WOW!!!
```

swapcase()

The method **swapcase()** returns a copy of the string in which all the case-based characters have had their case swapped.

```
str = "this is string example....wow!!!";
print (str.swapcase())
str = "THIS IS STRING EXAMPLE....WOW!!!";
print (str.swapcase())
```

When we run above program, it produces following result −

```
THIS IS STRING EXAMPLE....WOW!!!
this is string example....wow!!!
```

capitalize()

It returns a copy of the string with only its first character capitalized.

```
str = "this is string example....wow!!!";
print ("str.capitalize() : ", str.capitalize())
```

Result

```
str.capitalize() :  This is string example....wow!!!
```

title()

Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.

```

str = "this is string example....wow!!!";
```

```
print (str.title())
```

When we run above program, it produces following result −

```
This Is String Example....Wow!!!
```

lstrip()

The method **lstrip()** returns a copy of the string in which all chars have been stripped from the beginning of the string (default whitespace characters).

```
str = "     this is string example....wow!!!     ";
print (str.lstrip())
str = "88888888this is string example....wow!!!8888888";
print (str.lstrip('8'))
```

When we run above program, it produces following result −

```
this is string example....wow!!!
this is string example....wow!!!8888888
```

rstrip()

The method **rstrip()** returns a copy of the string in which all *chars* have been stripped from the end of the string (default whitespace characters).

```
str = "     this is string example....wow!!!     ";
print (str.rstrip())
str = "88888888this is string example....wow!!!8888888";
print (str.rstrip('8'))
```

When we run above program, it produces following result −

```
this is string example....wow!!!
88888888this is string example....wow!!!
```

strip()

The method **strip()** returns a copy of the string in which all chars have been stripped from the beginning and the end of the string (default whitespace characters).

```
str = "0000000this is string example....wow!!!0000000";
print (str.strip( '0' ))
```

When we run above program, it produces following result −

```
this is string example....wow!!!
```

center(width,fillchar)

The method center() returns centered in a string of length width. Padding is done using the specified fillchar. Default filler is a space.

```
str = "this is string example....wow!!!";
print ("str.center(40, 'a') : ", str.center(40, 'a'))
```

Result

```
str.center(40, 'a') :  aaaathis is string example....wow!!!aaaa
```

ljust(width[, fillchar])

Returns a space-padded string with the original string left-justified to a total of width columns.

```
str = "this is string example....wow!!!";
print (str.ljust(50, '0'))
```

When we run above program, it produces following result −

```
this is string example....wow!!!000000000000000000
```

rjust(width,[, fillchar])

Returns a space-padded string with the original string right-justified to a total of width columns.

```
str = "this is string example....wow!!!";
print (str.rjust(50, '0'))
```

When we run above program, it produces following result −

```
000000000000000000this is string example....wow!!!
```

zfill (width)

Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).

```
str = "this is string example....wow!!!";
print (str.zfill(40))
print (str.zfill(50))
```

When we run above program, it produces following result −

```
00000000this is string example....wow!!!
000000000000000000this is string example....wow!!!
```

startswith(str, beg=0,end=len(string))

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

```
str = "this is string example....wow!!!";
print (str.startswith( 'this' ))
print (str.startswith( 'is', 2, 4 ))
print (str.startswith( 'this', 2, 4 ))
```

When we run above program, it produces following result −

```
True
True
False
```

endswith(suffix, beg=0, end=len(string))

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

```
str = "this is string example....wow!!!";
suffix = "wow!!!";
```

```
print (str.endswith(suffix))
print (str.endswith(suffix,20))
suffix = "is";
print (str.endswith(suffix, 2, 4))
print (str.endswith(suffix, 2, 6))
```

Result

```
True
True
True
False
```

isalpha()

Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.

```
str = "this";  # No space & digit in this string
print (str.isalpha())
str = "this is string example....wow!!!";
print (str.isalpha())
```

When we run above program, it produces following result −

```
True
False
```

isalnum()

Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.

```
str = "this2009";  # No space in this string
print (str.isalnum())
str = "this is string example....wow!!!";
print (str.isalnum())
```

When we run above program, it produces following result −

```
True
False
```

isdigit()

Returns true if string contains only digits and false otherwise.

```
str = "123456";  # Only digit in this string
print (str.isdigit())
str = "this is string example....wow!!!";
print (str.isdigit())
```

When we run above program, it produces following result −

```
True
```

```
False
```

islower()

Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.

```
str = "THIS is string example....wow!!!";
print (str.islower())
str = "this is string example....wow!!!";
print (str.islower())
```

When we run above program, it produces following result −

```
False
True
```

isupper()

Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.

```
str = "THIS IS STRING EXAMPLE....WOW!!!";
print (str.isupper())
str = "THIS is string example....wow!!!";
print (str.isupper())
```

When we run above program, it produces following result −

```
True
False
```

isspace()

Returns true if string contains only whitespace characters and false otherwise.

```
str = "       ";
print (str.isspace())
str = "This is string example....wow!!!";
print (str.isspace())
```

When we run above program, it produces following result −

```
True
False
```

istitle()

Returns true if string is properly "titlecased" and false otherwise.

```

str = "This Is String Example...Wow!!!";
print (str.istitle())

str = "This is string example....wow!!!";
print (str.istitle())
```

When we run above program, it produces following result −

```
True
False
```

List

The list is a most versatile datatype available in Python which can be written as a list of **comma-separated values (items) between square brackets**. Important thing about a list is that items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets. For example −

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"];
```

The values stored in a list can be accessed using the slice operator**[]** and **[:]** with indexes starting at 0 at the beginning of the list and ending with -1.The + sign is the concatenation operation And * is the repetition operation.

```
>>> l=[2,3,41,87]
>>> print(l)
[2, 3, 41, 87]
>>> print(l[0])
2
>>> print(l[1:3])
[3, 41]
>>> print(l[2:])
[41, 87]
>>> print(l*3)
[2, 3, 41, 87, 2, 3, 41, 87, 2, 3, 41, 87]
>>> list=["mdmjd","as",23,6.9]
>>> print(l+list)
[2, 3, 41, 87, 'mdmjd', 'as', 23, 6.9]
```

Updating Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example −

```
list = ['physics', 'chemistry', 1997, 2000];
print ("Value available at index 2 : ")
print (list[2])
list[2] = 2001;
print ("New value available at index 2 :")
print (list[2])
```

When the above code is executed, it produces the following result −

```
Value available at index 2 :
1997
```

```
New value available at index 2 :
2001
```

Delete List Elements

To remove a list element, you can use either the **del** statement if you know exactly which element(s) you are deleting or the remove() method if you do not know. For example −

```
list1 = ['physics', 'chemistry', 1997, 2000];
print (list1)
del list1[2];
print ("After deleting value at index 2 : ")
print (list1)
```

When the above code is executed, it produces following result −

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

Built in list Functions

len(list)

Gives the total **len**gth of the list.

```
list1, list2 = [123, 'xyz', 'zara'], [456, 'abc']
print ("First list length : ", len(list1))
print ("Second list length : ", len(list2))
```

When we run above program, it produces following result −

```
First list length :  3
Second list length :  2
```

max(list)

Returns item from the list with **max** value.

```
>>> l=[2,3,41,87]
>>> print(l)
[2, 3, 41, 87]
>>> max(l)
87
```

min(list)

Returns item from the list with **min** value.

```
>>> min(l)
2
```

list(seq)

Converts a tuple into list.

```
aTuple = (123, 'xyz', 'zara', 'abc');
aList = list(aTuple)
```

```
print ("List elements : ", aList)
```

When we run above program, it produces following result −

```
List elements :  [123, 'xyz', 'zara', 'abc']
```

map(function, iterable, ...)

The map() function applies a given function to each item of an iterable (list, tuple etc.) and returns a list of the results. The returned value from map() (map object) then can be passed to functions like list() (to create a list), set() (to create a set) and so on.

```
>>> lst=[25,16,4,9]
>>> ls=list(map(math.sqrt,lst))
>>> ls
[5.0, 4.0, 2.0, 3.0]
```

Built in list methods

list.append(obj)

Appends object obj to list

```
aList = [123, 'xyz', 'zara', 'abc'];
aList.append( 2009 );
print ("Updated List : ", aList)
```

When we run above program, it produces following result −

```
Updated List :  [123, 'xyz', 'zara', 'abc', 2009]
```

list.count(obj)

Returns count of how many times obj occurs in list

```
aList = [123, 'xyz', 'zara', 'abc', 123];
print ("Count for 123 : ", aList.count(123))
print ("Count for zara : ", aList.count('zara'))
```

When we run above program, it produces following result −

```
Count for 123 :  2
Count for zara :  1
```

list.remove(obj)

Removes object obj from list

```
aList = [123, 'xyz', 'zara', 'abc', 'xyz'];
aList.remove('xyz');
print ("List : ", aList)
aList.remove('abc');
print ("List : ", aList)
```

When we run above program, it produces following result −

```
List :  [123, 'zara', 'abc', 'xyz']
List :  [123, 'zara', 'xyz']
```

list.index(obj)
Returns the lowest index in list that obj appears

```
aList = [123, 'xyz', 'zara', 'abc'];
print ("Index for xyz : ", aList.index( 'xyz' ) )
print ("Index for zara : ", aList.index( 'zara' ) )
```

When we run above program, it produces following result −

```
Index for xyz :  1
Index for zara :  2
```

list.extend(seq)
Appends the contents of seq to list

```
aList = [123, 'xyz', 'zara', 'abc', 123];
bList = [2009, 'manni'];
aList.extend(bList)
print ("Extended List : ", aList )
```

When we run above program, it produces following result −

```
Extended List :  [123, 'xyz', 'zara', 'abc', 123, 2009, 'manni']
```

list.reverse()
Reverses objects of list in place

```
aList = [123, 'xyz', 'zara', 'abc', 'xyz'];
aList.reverse();
print ("List : ", aList)
```

When we run above program, it produces following result −

```
List :  ['xyz', 'abc', 'zara', 'xyz', 123]
```

list.insert(index, obj)
Inserts object obj into list at offset index

```
aList = [123, 'xyz', 'zara', 'abc']
aList.insert( 3, 2009)
print ("Final List : ", aList)
```

When we run above program, it produces following result −

```
Final List : [123, 'xyz', 'zara', 2009, 'abc']
```

list.sort([func])
Sorts objects of list, use compare func if given

```
aList = ['xyz', 'zara', 'abc', 'xyz'];
aList.sort();
print ("List : ", aList)
aList.sort(reverse=True)
print(aList)
```

When we run above program, it produces following result −

```
List :  ['abc', 'xyz', 'xyz', 'zara']
['zara','xyz','xyz','abc']
```

list.pop(index)

Removes and returns object with the index from list

```
>>> ls=[3,1,7,9]
>>> ls.pop(2)
7
>>> ls
[3, 1, 9]
```

List.clear()

Removes all items from a list

```
>>> lst=[4,1,8,6]
>>> lst.clear()
>>> lst
[]
```

List.copy()

Returns a copy of the list.

```
>>> lst=[4,1,8,6]
>>> lst
[4, 1, 8, 6]
>>> l1=lst.copy()
>>> l1
[4, 1, 8, 6]
>>> lst.clear()
>>> lst
[]
>>> l1
[4, 1, 8, 6]
```

Using List as Stacks

The List can be used as a stack. stack is a data structure where the last element added is the first element retrieved. The list methods make it very easy to use a list as a stack. To add an item to the top of the stack, use append().To retrieve an item from the top of the stack , use pop() without an explicit index.

```
>>> s1=[2,45,21,67]
>>> s1.append(43)
>>> s1
[2, 45, 21, 67, 43]
>>> s1.pop()
43
>>> s1
[2, 45, 21, 67]
```

Using List as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved. Queues are Last in first out (LIFO) data Structure. But lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts and pops from the beginning of a list is slow since all of the other elements have to be shifted by one.
To implement a queue, python provides a module called collections in which a method called deque is designed to have fast appends and pops from both ends.

```
>>> import collections
>>> q=collections.deque(["apple","orange","pear"])
>>> q
deque(['apple', 'orange', 'pear'])
>>> q.append("cherry")
>>> q
deque(['apple', 'orange', 'pear', 'cherry'])
>>> q.popleft()
'apple'
>>> q
deque(['orange', 'pear', 'cherry'])
>>>
```

Tuple

A tuple is a sequence of **immutable** Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use **parentheses**, whereas lists use square brackets
Tuples can be considered as read only lists.

```
>>> t1=(3,4,2,1,69)
>>> t2=(999,'Tom')
>>> print(t1)
(3, 4, 2, 1, 69)
>>> print(t1[0])
3
>>> print(t1[0:])
(3, 4, 2, 1, 69)
>>> print(t1[0:-1])
(3, 4, 2, 1)
>>> print(t1[0:5])
(3, 4, 2, 1, 69)
>>> print(t1*2)
(3, 4, 2, 1, 69, 3, 4, 2, 1, 69)
>>> print(t1+t2)
(3, 4, 2, 1, 69, 999, 'Tom')
```

Tuples are immutable which means you cannot update or change the values of tuple elements.

ie: t1[0]=89 is an invalid statement.

Removing individual tuple elements is not possible.
Ie: del t1[0] is an invalid statement
To explicitly remove an entire tuple, just use the **del** statement.
Ie: del t1 is a valid statement

It is possible to pack values to a tuple and unpack values from a tuple.we can create tuple even without paranthesis.The reverse operation is called sequence unpacking and works for any sequence on the right hand side.

```
>>> t1=2,3,4
>>> x,y,z=t1
>>> x
2
>>> y
3
>>> z
4
```

Built in tuple functions

len(tuple)
Gives the total length of the tuple.

```
tuple1, tuple2 = (123, 'xyz', 'zara'), (456, 'abc')
print ("First tuple length : ", len(tuple1))
print ("Second tuple length : ", len(tuple2))
```

When we run above program, it produces following result −

```
First tuple length :  3
Second tuple length :  2
```

max(tuple)
Returns item from the tuple with max value.

```
>>> t1=2,3,4
>>> max(t1)
4
```

min(tuple)
Returns item from the tuple with min value.

```
>>> t1=2,3,4
>>> min(t1)
2
```

tuple(seq)
Converts a list into tuple.

```
aList = (123, 'xyz', 'zara', 'abc');
aTuple = tuple(aList)
print ("Tuple elements : ", aTuple)
```

When we run above program, it produces following result −

```
Tuple elements :  (123, 'xyz', 'zara', 'abc')
```

Set

Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces{ }.It can have any number of items and they may be of different types(integer, float, tuple, string etc).items in a set are not ordered. Since they are unordered we cannot access or change an element of set using indexing or slicing. we can perform set operations like union ,intersection, difference on two sets. Set have unique values. They eliminate duplicates. The slicing operator [] does not work with sets. An empty set is created by the function set().

```
>>> s1={1,2,3}
>>> print(s1)
{1, 2, 3}
>>> s2={1,2,3,2,1,2} #o/p contains only unique values
>>> print(s2)
{1, 2, 3}
>>> s3={[2,3]} # error sets cannot have mutable items
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Accessing Values in a Set

We cannot access individual values in a set. We can only access all the elements together as shown above. But we can also get a list of individual elements by looping through the set.

```
>>> s3={2,3,4,5,61,45}
>>> for i in s3:
...     print(i)
...
2
3
4
5
45
61
>>>
```

Built in Set Functions

len() Returns Length of an Object

```
>>> s3={2,3,4,5,61,45}
>>> len(s3)
6
```

max(set)

returns largest element

```
>>> s3={2,3,4,5,61,45}
>>> max(s3)
```

61
min(set) returns smallest element
>>> s3={2,3,4,5,45}
>>> sum(s3)
59
Sorted(set)
Returns a new sorted set. The set does not sort itself.
>>> s1={9,3,45,21,67}
>>> s1
{67, 3, 9, 45, 21}
>>> sorted(s1)
[3, 9, 21, 45, 67]
any(set)
Returns true if the set contains at least one item.
>>> s1={3,2,16,3}
>>> s1
{16, 2, 3}
>>> any(s1)
True
>>> s1=set()
>>> any(s1)
False
>>>
all(set) Returns true if all the elements are true or the set is empty
>>> s1=set()
>>> all(s1)
True
>>> s={}
>>> all(s)
True
>>> s1={False,0,5}
>>> all(s1)
False

Built in set Methods
Set1.add(obj)
Adds an object obj to set
>>> s1={4,23,7}
>>> print(s1)
{4, 7, 23}
>>> s1.add(67)
>>> print(s1)
{67, 4, 7, 23}
>>>

Set.remove(obj)

Removes an element from the set.Raises keyError if the set is empty.

```
>>> s1={4,23,7}
>>> print(s1)
{4, 7, 23}
>>> s1.remove(7)
>>> print(s1)
{4, 23}
>>> s1.remove(8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 8
>>>
```

Set.discard(obj)

Removes an element obj from the set. Nothing happens if the element to be deleted is not in the list.

```
>>> s1={4,23,7}
>>> s1.discard(7)
>>> print(s1)
{4, 23}
>>> s1.discard(8)
>>> print(s1)
{4, 23}
>>>
```

Set.pop()

Removes and returns an arbitrary set element.Raises KeyError if the set is empty

```
>>> s1={4,23,7}
>>> s1.pop()
4
>>> print(s1)
{7, 23}
>>> s1=set()
>>> s1.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'
>>>
```

Set1.union(set2)

Returs the union of twosets as a new set

```
>>> s1={4,23,7}
>>> s2={8,9,6,2,4}
>>> s3=s1.union(s2)
>>> print(s3)
{2, 4, 6, 7, 8, 9, 23}
```

>>>
Set1.update(s2)
Update a set with the union of itself and others.The result will be stored in set1.
>>> s1={4,23,7}
>>> s2={8,9,6,2,4}
>>> s1.update(s2)
>>> print(s1)
{2, 4, 6, 7, 8, 9, 23}
>>>
Enumerate(set)
Returns an enumerate object.It contains the index and value of all the items of set as pair.
>>> s1={3,4,52}
>>> print(set(enumerate(s1)))
{(0, 3), (2, 52), (1, 4)}
>>>
Any(set)
returns true if the set contains at least one item,false otherwise
>>> s1={3,4,52}
>>> any(s1)
True
>>> s1=set()
>>> any(s1)
False
>>>
All(set)
Returns true if all elements are true or the set is empty
>>> s={2,3,4}
>>> all(s)
True
>>> s=set()
>>> all(s)
True
>>> s={0,23,6}
>>> all(s)
False
>>> s={False,23,6}
>>> all(s)
False
Built in Set methods
1.set1.add(obj) Adds an element obj to set.
>>> s={2,3,1}
>>> s.add(9)
>>> print(s)

```
{1, 2, 3, 9}
>>>
```

2.set.remove(obj) Removes an element obj from the set.Raises key error if the element is not present

```
>>> s={2,3,1}
>>> s.remove(3)
>>> print(s)
{1, 2}
>>> s.remove(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 3
```

3.set.discard(obj) Removes an element obj from the set.nothing happens if the element is not present

```
>>> s={2,3,1}
>>> s.discard(1)
>>> s
{2, 3}
>>> s.discard(10)
```

4.set.pop().Removes and returns an arbitrary set element.Raises key error if the set is empty

```
>>> s={2,3,1}
>>> s.pop()
1
>>> s
{2, 3}
```

5.set1.union(set2). Returns the union of two sets as a new set.

```
>>> s1={2,3}
>>> s2={4,6,3}
>>> s3=s1.union(s2)
>>> s3
{2, 3, 4, 6}
```

6.set1.update(set2).update a set with the union of itself and others.The result will be stored in set1.

```
>>> s1={2,3}
>>> s2={4,6,3}
>>> s1.update(s2)
>>> s1
{2, 3, 4, 6}
```

7.set1.intersection(set2).Returns the ingersection of two sets as a new set.

```
>>> s1={2,3}
>>> s2={4,6,3}
>>> s1.intersection(s2)
{3}
```

8.set1.intersection_update(set2).update the set with the intersection of itself and another. The result will be stored in set1.

```
>>> s1={2,3}
>>> s2={4,6,3}
>>> s1.intersection_update(s2)
>>> s1
{3}
>>>
```

9.set1.difference(set2).Returns the difference of two or more sets into a new set.

```
>>> s1={2,3}
>>> s2={4,6,3}
>>> s1.difference(s2)
{2}
```

10.set1.difference_update(set2).Remove all elements of another set set2 from set1 and the result is stored in set1.

```
>>> s1={2,3}
>>> s2={4,6,3}
>>> s1.difference_update(s2)
>>> s1
{2}
```

11.set1.symmetric_difference(set2).Returs the symmetric difference of two sets as a new set.

```
>>> s1={2,3}
>>> s2={4,6,3}
>>> s1.symmetric_difference(s2)
{2, 4, 6}
```

12.set1.symmetric_diffrence_update(set2).updates a set with the symmetric difference of itself and another.

```
>>> s1={2,3}
>>> s2={4,6,3}
>>> s1.symmetric_difference_update(s2)
>>> s1
{2, 4, 6}
```

13.set1.isdisjoint(set2).Returs true if two sets have a null intersection

```
>>> s1={2,3}
>>> s2={4,6,3}
>>> s1.isdisjoint(s2)
False
```

14.set1.issubset(set2).Returns true if set1 is a subset of s2.

```
>>> s1={2,3}
>>> s2={4,6,3}
>>> s1.issubset(s2)
False
```

15.set1.issubset(set2).Returs true if set1 is a superset of set2

```
>>> s1={2,3}
>>> s2={4,6,3}
>>> s1.issuperset(s2)
False
```

frozenset

frozenset is a new class that has the characteristics of a set ,but its elements cannet be changed once assigned.frozensts are immutable sets.frozensets can be used as keys to a dictionary.

Frozensets can be created using the function frozenset().This datatype suppors methods like difference(),intersection(),isdisjoint(),issubset(),issuperset(),symmetric_difference and union.being immutable it does not have methods like add(),remove(),update(),difference_update(),intersection_update(),symmetric_difference_update() etc.

```
>>> set1=frozenset({2,3,4,5})
>>> print(set1)
frozenset({2, 3, 4, 5})
>>> set2=frozenset({7,6,5})
>>> print(set2)
frozenset({5, 6, 7})
>>> print(set1.intersection(set2))
frozenset({5})
>>>
```

Dictionary

Is an unordered collection of key-value pairs. Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Ex:

```
>>> d={}
>>> d['one']="pen"
>>> d[2]="pencil"
>>> d['age']=23
>>> d['name']='basil'
>>> d
{'one': 'pen', 2: 'pencil', 'age': 23, 'name': 'basil'}
>>> print(d['name'])
```

```
basil
>>> print(d['age'])
23
```

Properties of keys

1.More than one entry per key is not allowed ie : no duplicate key is allowed.when duplicate keys are encountered during assignment the last assignment is taken.
2.keys are immutable.This means keys can be numbers,strings or tuple.but it does not permit mutable objects like lists.

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example −

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry
print (("dict['Age']: ", dict['Age'])
print( "dict['School']: ", dict['School'])
```

When the above code is executed, it produces the following result −

```
dict['Age']:  8
dict['School']:  DPS School
```

Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.To explicitly remove an entire dictionary, just use the **del** statement. Following is a simple example −

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
del dict['Name']; # remove entry with key 'Name'
dict.clear();     # remove all entries in dict
del dict ;        # delete entire dictionary
print ("dict['Age']: ", dict['Age']
print ("dict['School']: ", dict['School']
```

This produces the following result. Note that an exception is raised because after **del dict** dictionary does not exist any more −

```
dict['Age']:
Traceback (most recent call last):
  File "test.py", line 8, in <module>
    print ("dict['Age']: ", dict['Age'];
TypeError: 'type' object is unsubscriptable
```

Built in dictionary methods

1.len(dict).Gives the length of the dictionary

```
dict = {'Name': 'Zara', 'Age': 7};
print( "Length : %d" % len (dict))
```

When we run above program, it produces following result −

```
Length : 2
```

2.str(dict) Produces a printable string representation of a dictionary

```
>>> dict = {'Name': 'Zara', 'Age': 7}
>>> str(dict)
"{'Name': 'Zara', 'Age': 7}"
>>>
```

3.type(variable)

Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

```
>>> dict = {'Name': 'Zara', 'Age': 7}
>>> type(dict)
<class 'dict'>
```

Methods with Description

dict.clear()

Removes all elements of dictionary *dict*

```
dict = {'Name': 'Zara', 'Age': 7};
print( "Start Len : %d" %  len(dict))
dict.clear()
print (("End Len : %d" %  len(dict))
```

When we run above program, it produces following result −

```
Start Len : 2
End Len : 0
```

dict.copy()

Returns a shallow copy of dictionary *dict*

```
dict1 = {'Name': 'Zara', 'Age': 7};
dict2 = dict1.copy()
print (("New Dictionary : %s" %  str(dict2))
```

When we run above program, it produces following result −

```
New Dictionary : {'Age': 7, 'Name': 'Zara'}
```

dict.keys()

Returns list of dictionary dict's keys

```
dict = {'Name': 'Zara', 'Age': 7}
print (("Value : %s" %  dict.keys())
```

When we run above program, it produces following result −

```
Value : ['Age', 'Name']
```

dict.values()
Returns list of dictionary *dict*'s values

```
dict = {'Name': 'Zara', 'Age': 7}
print (("Value : %s" %  dict.values())
```

When we run above program, it produces following result −

```
Value : [7, 'Zara']
```

dict.items()
Returns a list of *dict*'s (key, value) tuple pairs

```
dict = {'Name': 'Zara', 'Age': 7}
print ("Value : %s" %  dict.items()
```

When we run above program, it produces following result −

```
Value : [('Age', 7), ('Name', 'Zara')]
```

dict.update(dict2)
Adds dictionary *dict2*'s key-values pairs to *dict*

```
dict = {'Name': 'Zara', 'Age': 7}
dict2 = {'Sex': 'female' }
dict.update(dict2)
print (("Value : %s" %  dict)
```

When we run above program, it produces following result −

```
Value : {'Age': 7, 'Name': 'Zara', 'Sex': 'female'}
```

dict.has_key(key)
Returns *true* if key in dictionary *dict*, *false* otherwise

```
dict = {'Name': 'Zara', 'Age': 7}
print (("Value : %s" %  dict.has_key('Age'))
print (("Value : %s" %  dict.has_key('Sex'))
```

When we run above program, it produces following result −

```
Value : True
Value : False
```

dict.get(key, default=None)
For *key* key, returns value or default if key not in dictionary

```
dict = {'Name': 'Zabra', 'Age': 7}
print (("Value : %s" %  dict.get('Age'))
print (("Value : %s" %  dict.get('Education', "Never"))
```

When we run above program, it produces following result −

```
Value : 7
Value : Never
```

dict.setdefault(key, default=None)

Similar to get(), but will set dict[key]=default if *key* is not already in dict

```
dict = {'Name': 'Zara', 'Age': 7}
print("Value : %s" %  dict.setdefault('Age', None))
print("Value : %s" %  dict.setdefault('Sex', None))
```

When we run above program, it produces following result −

```
Value : 7
Value : None
```

dict.fromkeys(seq[, value])

Create a new dictionary with keys from seq and values *set* to *value*.

```
seq = ('name', 'age', 'sex')
dict = dict.fromkeys(seq)
print (("New Dictionary : “,dict)
dict = dict.fromkeys(seq, 10)
print (("New Dictionary : ", dict)
```

When we run above program, it produces following result −

```
New Dictionary : {'age': None, 'name': None, 'sex': None}
New Dictionary : {'age': 10, 'name': 10, 'sex': 10}
```

DataType conversions

We may need to perform conversions between the built-in types.To convert different datatypes ,use the typename as function.There are several built in functions to perform conversion from one datatype to other.These functions return a new object representing the converted value.

function	Description	Example

int(x,[base]	Converts x to integer,base specifies the base if x is a string	>>> x=int("123")
		>>> print(x)
		123
		>>> x=int("123",8)
		>>> print(x)
		83
		>>> x=int("123",16)
		>>> print(x)
		291
		>>> x=int("1010",2)
		>>> x
		10

float(x)	Convert x to floating point	>>> x=float("23.5")
		>>> x
		23.5

complex(2,3)	Creates a complex number	>>> c1=complex(2,3) >>> c1

function	Description	Example
		(2+3j)

str(x)	Converts x to a string representation	>>> x=str(23)
		>>> x
		'23'

eval(str)	Evaluates a string and returns an object	>>> s=eval("28*3+5")
		>>> s
		89

tuple(s)	Converts s to a tuple	>>> s=[2,3,4,5]
		>>> t=tuple(s)
		>>> t
		(2, 3, 4, 5)

list(s)	Converts s to a list	>>> t=(3,4,5)
		>>> l=list(t)
		>>> l
		[3, 4, 5]

set(s)	Converts s to a set	>>> s1=set(s)
		>>> s1
		{3, 4, 5}

dict(d)	Creates a dictionary.d must be a sequence of (key,value) pairs	l=[(1,21),('roll',23),('age',34)]
		>>> d=dict(l)
		>>> d
		{1: 21, 'roll': 23, 'age': 34}

frozenset(s)	Converts s to a frozen set	>>> s={2,3,4,5}
		>>> fs=frozenset(s)
		>>> fs
		frozenset({2, 3, 4, 5})

chr(x)	Converts x to a character	>>> c=chr(65)
		>>> c
		'A'

ord(c)	Converts character to an integer value	>>> x=ord('A')
		>>> x
		65

hex(x)	Converts an integer to hexadecimal string	>>> s=hex(10)
		>>> s
		'0xa'

oct(x)	Converts an integer to n octal string	>>> s=oct(123)
		>>> s
		'0o173'

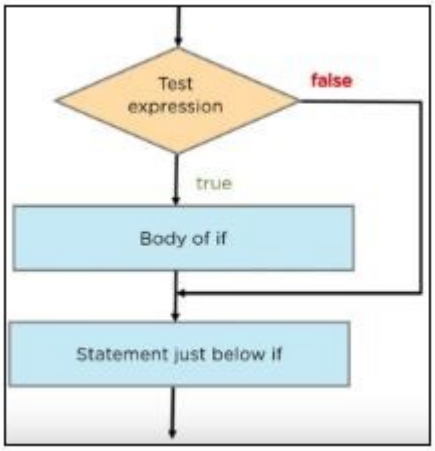
Decision Making

Sometimes, in a program, we may want to make a decision based on a condition. We know that an expression's value can be True or False. We may want to do

something only when a certain condition is true. For example, assume variables a and b. If a is greater, then we want to print ("a is greater". Otherwise, we want to print ("b is greater". For this, we use an if-statement.

if Statements

An if statement in python takes an expression with it. If the expression amounts to True, then the block of statements under it is executed. If it amounts to False, then the block is skipped and control transfers to the statements after the block. But remember to indent the statements in a block equally. This is because we don't use curly braces to delimit blocks. Also, use a colon(:) after the condition.



Syntax

```
If testexp:
        Stments
```

Ex:

```
a=int(input("Enter a No"))
if (a>10):
        print("no >10")
print("This is always printed")
```

output1

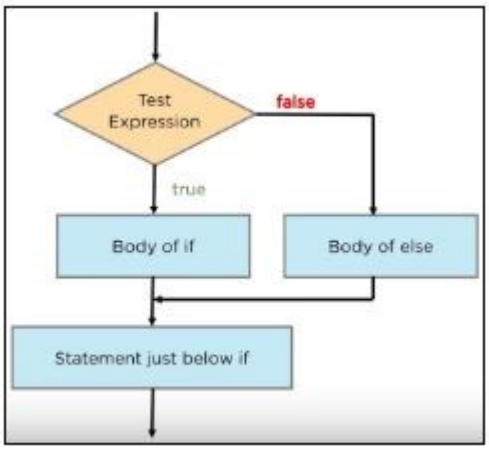
```
Enter a No12
no >10
This is always printed
```

Output2

```
Enter a No2
This is always printed
```

if-else Statements

What happens when the if condition is untrue? We can mention that it in the block after the else statement. An else statement comes right after the block after 'if'.



Syntax

```
if testexp:
        Statements
else:
        statements
```

Ex:

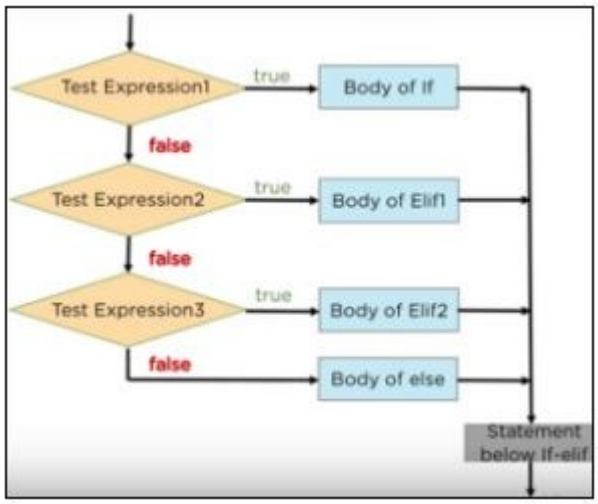
```
a=int(input("Enter a No"))
if  a%2==0:
        print("no is even")
else:
        print("no is odd")
print("pgm ends")
```

Output1
Enter a No2
no is even
pgm ends

output2
Enter a No3
no is odd
pgm ends

if –elif..else statement

Python allows the elif keyword as a replacement to the else-if statements in Java or C++. When we have more than one condition to check, we can use it. If condition 1 isn't True, condition 2 is checked. If it isn't true, condition 3 is checked.



if testexp:
body of if
elif testexp
body of elif
..
..
else:
body of else

Ex:

```
m=int(input("enter mark"))
if  m>=80:
        print("distiction")
elif m>=60:
        print("firstclass")
elif m>=50:
        print("second class")
else:
        print("failed")
```

Output1
enter mark90
distiction

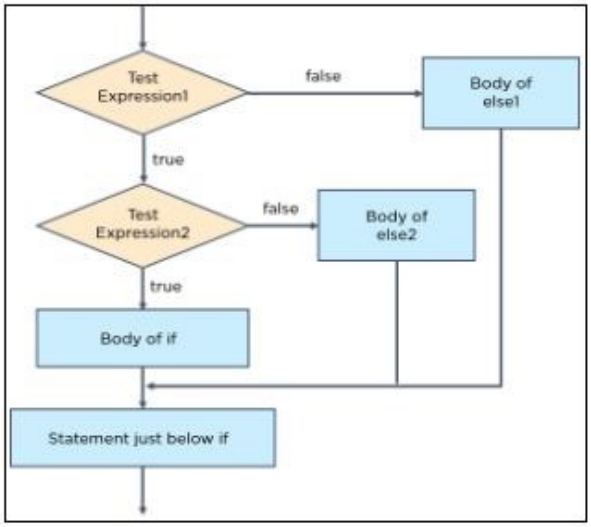
Output2
enter mark40
failed

Output3
enter mark55
second class

Output4
enter mark60
firstclass

Nested if statements

You can put an if statement in the block under another if statement. This is to implement further checks.



Example

```
x=int(input("enter no1"))
y=int(input("enter no2"))
z=int(input("enter no3"))
if  x>y:
        if x>z:
                print("largest is",x)
        else:
                print("largest is",z)
else:
        if y>z:
                print("largest is",y)
        else:
                print("largest is",z)
```

output
enter no12
enter no23
enter no34
largest is 34

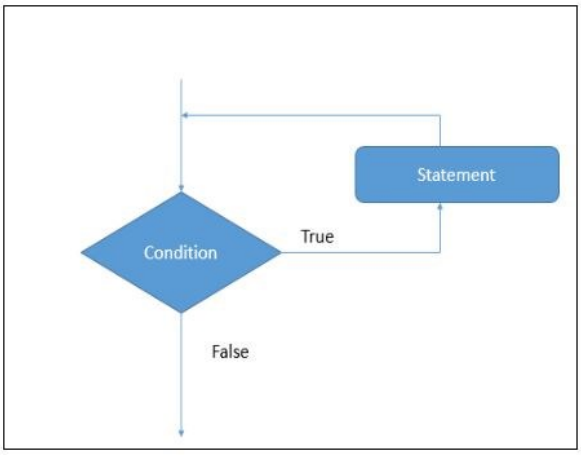
output
enter no14
enter no22
enter no33
largest is 33

output
enter no14
enter no23
enter no32
largest is 32

output
enter no12
enter no25
enter no33
largest is 33

Loops

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.
A loop statement allows us to execute a statement or group of statements multiple times.



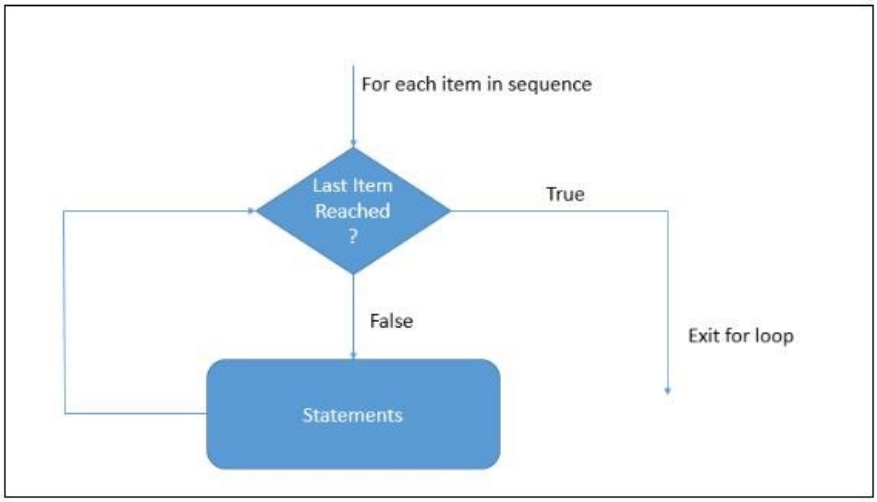
For loop

It has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax

```
for iterating_var in sequence:
   statements(s)
```

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating_var*. Next, the statements block is executed. Each item in the list is assigned to *iterating_var*, and the statement(s) block is executed until the entire sequence is exhausted.



Ex:

```
l=[2,3,4,56]
sum=0
for i in l:
    sum=sum+i
print("Sum is",%sum)
```

output
sum is 65

range function

The range function is used to create a sequence of numbers.
Syntax

```
range(start,end,step)
```

the default value for start is zero and step is 1
range(10) will generate the sequence from 0,1,..9
range(1,10,2) generate the sequence 1,3,5..9
ex:

```
for i in range(5):
    print(i)
print("================")
for i in range(1,10,2):
    print(i)
```

output
0
1
2
3
4

==========
1
3
5
7
9

For loop with else statement.

If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has finished iterating the list.a break statement can be used to stop a for loop.in this case the else part is ignored.

Ex:

```
l=[3,4,512,89]
x=int(input("Enter an element to search"))
for i in l:
    if i==x:
        print("element found in the list")
        break
else:
```

```
        print("Element not found")
```

output
Enter an element to search12
Element not found

output
Enter an element to search512
element found in the list

while loop

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

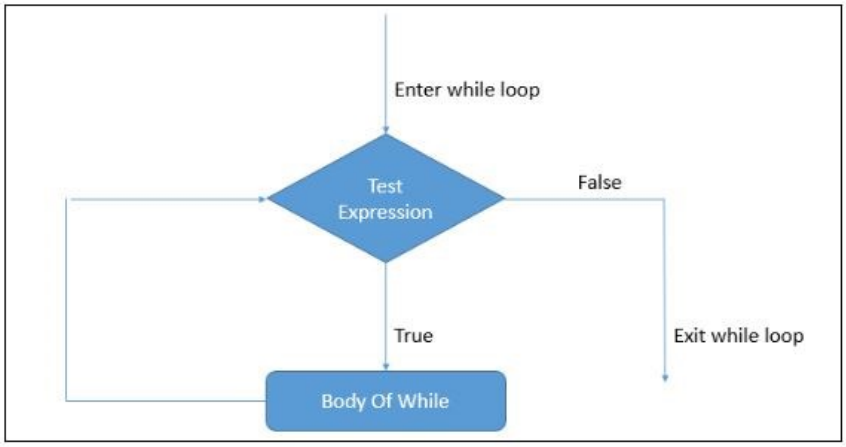
Syntax

The syntax of a **while** loop in Python programming language is −

```
while expression:
   statement(s)
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.



Ex:

```
i=1
while i<=10:
        if i%2==0:
                print("even ",i)
        i=i+1
```

output
even 2
even 4
even 6
even 8
even 10

while loop with else

If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.if the while loop is terminated using a break statement the else part is ignored.
Ex:

```
l=[3,4,512,89]
x=int(input("Enter an element to search"))
i=0
while i<len(l):
    if l[i]==x:
        print("element found in the list")
        break
    i=i+1
else:
    print("Element not found")
```

output
Enter an element to search89
element found in the list

output
Enter an element to search34
Element not found

Nested Loops

Python programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.
Syntax

```
for iterating_var in sequence:
  for iterating_var in sequence:
    statements(s)
  statements(s)
```

The syntax for a **nested while loop** statement in Python programming language is as follows −

```
while expression:
  while expression:
    statement(s)
  statement(s)
```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

```
for i in range(2,100,1):
    count=0
    for j in range(1,i+1,1):
        if i%j==0:
```

```
                    count=count+1
            if count==2:
                print(i ,"is prime")
```

output

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
```

Same program using nested while loop

```
i=1
while i<=100:
    count=0
    j=1
    while (j<=i):
        if i%j==0:
            count=count+1
        j=j+1
    if count==2:
        print(i ,"is prime")
    i=i+1
```

output

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
```

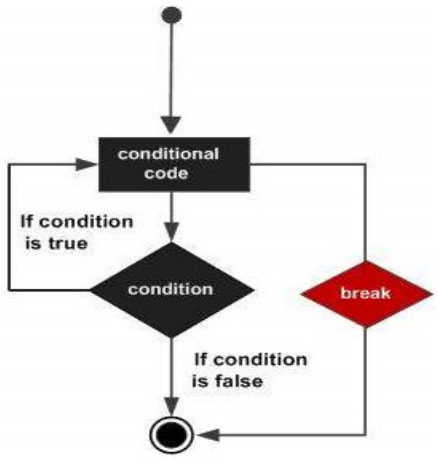
83 is prime
89 is prime
97 is prime

Control statements

Control statements change the execution from normal sequence.

break statement

Terminates the loop statement and transfers execution to the statement immediately following the loop.



Ex:

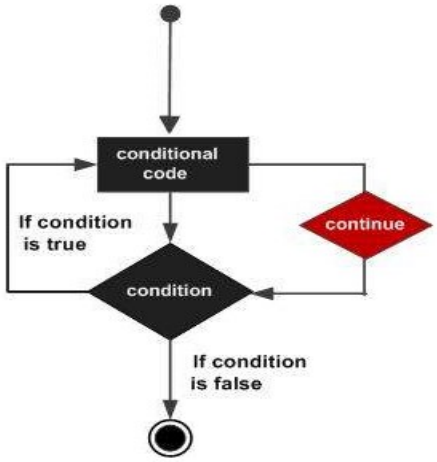
```
i=1
while i<=10:
        if i%5==0:
                break
        print(i)
        i=i+1
```

output

```
1
2
3
4
```

continue statement

Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.




```
i=1
while i<=10:
        if i%5==0:
                i=i+1
                continue
        print(i)
        i=i+1
```

output

1
2
3
4
6
7
8
9

pass statement

The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute

The **pass** statement is a *null* operation; nothing happens when it executes. The **pass** is also useful in places where your code will eventually go, but has not been written yet

Ex:

```
for i in range(10):
        pass
print(i)
```

output

9

Types of Loops

Inifinite loop

A loop becomes infinite loop if the condition never becomes false.This results in a loop that never ends.

Ex:

```
i=1
while i==1:
        print("msg")
```

This loop never ends press cntrl+c to stop.

Loops with condition at the top

This is a normal while loop with condition at the top without any braek statements.The loop terminates when the condition is false.

```
i=1
while i<=5:
        print("msg")
        i=i+1
```

output

```
msg
msg
msg
msg
msg
```

Loop with condition at the middle

This kind of loop can be implemented using an infinite loop along with a conditional break in between the body of the loop.

Ex:

```
vowels="aeiou"
while True:
        v=input("Enter a character")
        if v in vowels:
                print(v," is vowel")
                break
        print("not vowel enter another")
```

output

```
Enter a characterg
not vowel enter another

Enter a characterk
not vowel enter another

Enter a characterа
a  is vowel
```

Loop with condition at the bottom

This kind of loop ensures that the body of the loop is executed at least once.it can be implemented using an infinite loop along with a conditional break at the end.

Ex:

```
while True:
        x=int(input("no1"))
        y=int(input("no2"))
        ch=int(input("Enter choice 1.addition 2.subtraction 3.Exit"))
        if ch==1:
                print(x+y)
        if ch==2:
                print(x-y)
        if ch==3:
                break
```

output

```
no12
no23
Enter choice 1.addition 2.subtraction
3.Exit 1
5
no13
no24
Enter choice 1.addition 2.subtraction
3.Exit 2
-1
no13
no25
Enter choice 1.addition 2.subtraction
3.Exit3

print("sum is ",sum)Nested Loops,
Control Statements, Types of
Loops
```

The End – Module I