

# **PYTHON PROGRAMMING & DATA SCIENCE**

# PYTHON PROGRAMMING & DATA SCIENCE

## UNIT 1

### **Introduction to Python:**

Features of Python, Data types,  
Operators, Input and output, Control Statements.

### **Strings:**

Creating strings and basic operations on  
strings, string testing methods.

Lists, Dictionaries, Tuples.

# PYTHON PROGRAMMING & DATA SCIENCE

## History:

➤ Python was developed by **Guido van Rossum** in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands .

➤ The name Python was selected from the TV Show "**The Complete Monty Python's Circus**", which was broadcasted in BBC from 1969 to 1974.

# PYTHON PROGRAMMING & DATA SCIENCE

## Where we can use Python /Applications:

The most common important application areas are

1. Desktop Applications
2. web Applications
3. database Applications
4. Network Programming
5. Games
6. Data Analysis Applications
7. Machine Learning
8. developing Artificial Intelligence Applications
9. IOT

### **Note:**

- Internally Google and Youtube use Python coding

# PYTHON PROGRAMMING & DATA SCIENCE

## Features in Python

There are many features in Python,  
some of them are:

1. Easy to code
2. Free and Open Source
3. Object-Oriented Language
4. GUI Programming Support
5. High-Level Language

# PYTHON PROGRAMMING & DATA SCIENCE

## Features in Python

6. Extensible feature
7. Python is Portable language
8. Python is Integrated language
9. Interpreted Language
10. Large Standard Library
11. Dynamically Typed Language

# PYTHON PROGRAMMING & DATA SCIENCE

## Features in Python

### 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 also a developer-friendly language.

### 2. Free and Open Source:

Python language is freely available at the official website and can download it from the link [www.python.org](http://www.python.org) .

It is open-source, this means that source code is also available to the public. So we can download it as, use it as well as share it.

### 3. 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.

# PYTHON PROGRAMMING & DATA SCIENCE

## Features in Python

### 4. GUI Programming Support:

Graphical User interfaces can be made using a module such as PyQt5, PyQt4, wxPython, or Tkinter python.

PyQt5 is the most popular option for creating graphical apps with Python.

### 5. 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.

### 6. Extensible feature:

Python is a 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.



# PYTHON PROGRAMMING & DATA SCIENCE

## Features in Python

### **7. 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.

### **8. Python is Integrated language:**

Python is also an Integrated language because we can easily integrated python with other languages like c, c++, etc.

### **9. Interpreted Language:**

Python is an Interpreted Language because Python code is executed line by line at a time.

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.

# PYTHON PROGRAMMING & DATA SCIENCE

## Features in Python

### 10. Large Standard Library:

Python has a large standard library which provides a rich set of module and functions so no need to write own code for every single thing.

There are many libraries present in python for such as regular expressions, unit-testing, web browsers, etc.

### 11. 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 PROGRAMMING & DATA SCIENCE

## Python Data Types

- Data types are the classification or categorization of data items.
- Data type specifies the type of value.
- Python supports the following built-in data types.
  1. Scalar Types
  2. Sequence Type
  3. Mapping Type
  4. Set Types
  5. Mutable and Immutable Types

# PYTHON PROGRAMMING & DATA SCIENCE

## Scalar Types:

Scalar data type contains following data types.

1. int
2. float
3. complex
4. Bool
5. None

# PYTHON PROGRAMMING & DATA SCIENCE

## 1. int:

➤ In Python, integers are zero, positive or negative **whole numbers without a fractional part and having unlimited precision.**

Examples:

-10, 10, 456, 4654654.

➤ Integers can be **binary, octal, and hexadecimal values.**

Examples:

```
>>> 0b11011000 # binary
```

216

```
>>> 0o12 # octal
```

10

```
>>> 0x12 # hexadecimal
```

15

# PYTHON PROGRAMMING & DATA SCIENCE

➤ The `type()` method is used to get the class name.

Example:

```
>>> x=1234567890
```

```
>>> type(x)
```

```
<class 'int'>
```

```
# type of x is int
```

**Note:**

➤ The `int()` function converts a string or float to int.

Example:

```
>>> int('100')
```

```
100
```

# PYTHON PROGRAMMING & DATA SCIENCE

## 2. float:

➤ In Python, floating point numbers (float) are positive and negative **real numbers with a fractional part denoted by the decimal symbol (or) the scientific notation E or e.**

Example:

1234.56, 3.142, -1.55, 0.23,

➤ Floats can **be separated by the underscore \_**.

Example:

123\_42.222\_013 is a valid float

➤ the **float()** function to convert string, int to float.

Example:

```
>>> float('5.5')
```

```
5.5
```

# PYTHON PROGRAMMING & DATA SCIENCE

## 3. complex:

➤ A complex number is a number with real and imaginary components.

Example:

$$5 + 6j$$

## 4. None:

➤ The None represents the null object in Python.  
A None is returned by functions that don't explicitly return a value.



# PYTHON PROGRAMMING & DATA SCIENCE

## 5. bool:

➤ Data with one of two built-in values **True or False**.

Notice that **'T' and 'F' are capital**. **true** and **false** are not valid Booleans.

➤ Examples:

```
b=True
```

```
type(b)
```

```
=>bool
```

# PYTHON PROGRAMMING & DATA SCIENCE

## 2. Sequence Type

➤ A sequence is an **ordered collection of similar or different data types.**

➤ Python has the following built-in sequence data types:

1. [String](#)
2. [List](#)
3. [Tuple](#)

# PYTHON PROGRAMMING & DATA SCIENCE

## 2. Sequence Type

### String:

A string value is a collection of one or more characters put in single, double or triple quotes.

### List:

A list object is an ordered collection of one or more data items, not necessarily of the same type, put in square brackets.

### Tuple:

A Tuple object is an ordered collection of one or more data items, not necessarily of the same type, put in parentheses.

# PYTHON PROGRAMMING & DATA SCIENCE

## 3.Mapping Type

### Dictionary:

- A dictionary Dict() object is an **unordered collection of data in a key:value pair form**.
- A collection of such pairs is enclosed in **curly brackets**.
- Example:  
{ 1:"Raju", 2:"Bill", 3:"Ram", 4: "Farha" }

# PYTHON PROGRAMMING & DATA SCIENCE

## 4.Set Types

set:

- Set is mutable, unordered collection of distinct hashable objects.
- A set object has suitable methods to perform mathematical set operations like union, intersection, difference, etc.

frozenset:

- Frozenset is immutable version of set whose elements are added from other iterables.

# PYTHON PROGRAMMING & DATA SCIENCE

## 5.Mutable and Immutable Types

➤ Numbers, strings, and Tuples are immutable, which means their **contents can't be altered after creation.**

➤ items in a List or Dictionary object can be modified. It is possible to add, delete, insert, and rearrange items in a list or dictionary. Hence, they are mutable objects.

# PYTHON PROGRAMMING & DATA SCIENCE

## Variables:

Variables represents **memory locations to store values.**

(Or)

Variables are **containers for storing data values.**

## Creating Variables

- Python has **no command** for declaring a variable.
- A variable is **created the moment we assign a value**.
- The equal sign (=) is used to assign values to variables.

## Example:

```
x = 5  
y = "cse"  
print(x)  
print(y)
```

# PYTHON PROGRAMMING & DATA SCIENCE

## Rules for creating variables in Python:

- A variable name must start with a letter or the underscore character.
- A variable name cannot start with a number.
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and \_ ).
- Variable names are case-sensitive (name, Name and NAME are three different variables).
- The reserved words(keywords) cannot be used naming the variable.



# PYTHON PROGRAMMING & DATA SCIENCE

## Multiple Assignments to variables:

➤ Python allows to **assign a single value to several variables** simultaneously.

Example :

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

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location.

➤ **multiple objects can be assigned to multiple variables** simultaneously.

Example :

```
a, b, c = 1, 2.5, "cse"
```

➤ integer object is created with the value 1 assigned to the variable a

➤ float object is created with the value 2.5 assigned to the variable b

➤ one string object with the value "cse" is assigned to the variable c.

# PYTHON PROGRAMMING & DATA SCIENCE

Delete a variable:

➤ Python variables can be deleted using the **command del**.

syntax :

del "variable name".

Example:

```
f=11
```

```
del f
```

# KEYWORDS

- The words which are predefined by the **system compiler** are called as **keywords**.
- Keywords are also known as "*Reserved Words*".
- All keywords must in *lower case* characters.

# KEYWORDS

➤ *35 keywords* are available in Python.

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
<u>async</u>	<u>elif</u>	if	or	yield

# PYTHON PROGRAMMING & DATA SCIENCE

## Identifiers

- A name in Python program is called identifier.
- It can be class name or function name or module name or variable name

Example :

```
a = 10
```

## Rules to define identifiers in Python:

1. The only allowed characters in Python are
  - i) alphabet symbols(either lower case or upper case)
  - ii) digits(0 to 9)
  - iii) underscore symbol(\_)

# PYTHON PROGRAMMING & DATA SCIENCE

## Rules to define identifiers in Python:

2. Identifier should not starts with digit

Example:

123total (x)

total123 (✓)

3. Identifiers are case sensitive.

Example:

total=10

TOTAL=999

print(total) #10

print(TOTAL) #999

# PYTHON PROGRAMMING & DATA SCIENCE

## Rules to define identifiers in Python:

### Note:

1. If identifier starts with \_ symbol then it indicates that it is private.
2. If identifier starts with \_\_ (two under score symbols) indicating that strongly private identifier.
3. If the identifier starts and ends with two underscore symbols then the identifier is language defined special name, which is also known as magic methods.

Example :

\_\_add\_\_

# PYTHON PROGRAMMING & DATA SCIENCE

## Comments

- Single-line comments begins with a hash(#) symbol and is useful in mentioning that the whole line should be considered as a comment until the end of line.
- A Multi line comment is used to specify comment on many lines.
- In python, triple double quote(“ “ “) and single quote (‘ ‘ ‘)are used for multi-line commenting.



# PYTHON PROGRAMMING & DATA SCIENCE

## Comments

```
# Write a python program to add numbers
a=10  #assigning value to variable a
b=20  #assigning value to variable b
""" print the value of a and b using a new variable """
''' print the value of a and b using a new variable '''
c=a+b
print(c)
|
```

# PYTHON PROGRAMMING & DATA SCIENCE

## Types of Operators:

➤ Python language supports the following types of operators.

### 1. Arithmetic Operators

`+, -, *, /, %, **, //`

### 2. Comparison (Relational) Operators

`=, !=, < >, <, >, <=, >=`

### 3. Assignment Operators

`=, +=, -=, *=, /=, %=, **=, //=`

### 4. Logical Operators

`and, or, not`

# PYTHON PROGRAMMING & DATA SCIENCE

## Types of Operators:

### 5. Bitwise Operators

&, |, ^, ~, <<, >>

### 6. Membership Operators

in, not in

### 7. Identity Operators

is, is not

# PYTHON PROGRAMMING & DATA SCIENCE

## Arithmetic Operators:

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	$a + b = 30$
- Subtraction	Subtracts right hand operand from left hand operand.	$a - b = -10$
* Multiplication	Multiplies values on either side of the operator	$a * b = 200$
/ Division	Divides left hand operand by right hand operand	$b / a = 2$
% Modulus	Divides left hand operand by right hand operand and returns remainder	$b \% a = 0$
** Exponent	Performs exponential (power) calculation on operators	$a ** b = 10$ to the power 20
// Floor Division	The division of operands where the result is the quotient in which the digits after the decimal point are removed.	$9 // 2 = 4$ and $9.0 // 2.0 = 4.0$

# PYTHON PROGRAMMING & DATA SCIENCE

Note:

➤ / operator always performs floating point arithmetic.

Hence it will always returns float value.

➤ Floor division (//) can perform both floating point and integral arithmetic.

➤ If arguments are int type then result is int type.

➤ If atleast one argument is float type then result is float type.

# PYTHON PROGRAMMING & DATA SCIENCE

## Comparison (Relational) Operators

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

# PYTHON PROGRAMMING & DATA SCIENCE

## Assignment Operators

Operator	Description	Example
=	Assigns values from right side operands to left side operand	$c = a + b$ assigns value of $a + b$ into $c$
$+=$ Add AND	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	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	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	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	It takes modulus using two operands and assign the result to left operand	$c \% = a$ is equivalent to $c = c \% a$
$**=$ Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	$c ** = a$ is equivalent to $c = c ** a$
$//=$ Floor Division	It performs floor division on operators and assign value to the left operand	$c //= a$ is equivalent to $c = c // a$

# PYTHON PROGRAMMING & DATA SCIENCE

## Logical Operators

Operator	Description	Example
And Logical AND	If both the operands are true then condition becomes true.	(a and b) is true.
Or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not (a and b) is false.



# PYTHON PROGRAMMING & DATA SCIENCE

## Bitwise Operators



Operator	Description	Example
$\&$ Binary AND	Operator copies a bit to the result if it exists in both operands.	$(a \& b) = 12$ (means 0000 1100)
$ $ Binary OR	It copies a bit if it exists in either operand.	$(a   b) = 61$ (means 0011 1101)
$\wedge$ Binary XOR	It copies the bit if it is set in one operand but not both.	$(a \wedge b) = 49$ (means 0011 0001)
$\sim$ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	$(\sim a) = -61$ (means 1100 0011 in 2's complement form due to a signed binary number.
$\ll$ Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	$a \ll 2 = 240$ (means 1111 0000)
$\gg$ Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	$a \gg 2 = 15$ (means 0000 1111)

# PYTHON PROGRAMMING & DATA SCIENCE

## Membership Operators

Python's membership operators **test for membership in a sequence, such as strings, lists, or tuples.**




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 not in y, here not in results in a 1 if x is not a member of sequence y.




# PYTHON PROGRAMMING & DATA SCIENCE

## Identity Operators

Identity operators compare the memory locations of two objects.



Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	<u>x</u> is y, here is results in 1 if <u>id</u> (x) equals id(y).
is not	Evaluates to <u>false</u> if the variables on either side of the operator point to the same object and true otherwise.	<u>x</u> is not y, here is not results in 1 if id(x) is not equal to id(y).



# PYTHON PROGRAMMING & DATA SCIENCE

## Python Operators Precedence

The following table lists all operators from highest precedence to lowest.

Operator	Description
()	Parenthesis
**	Exponentiation (raise to the power)
~x, +x, -x	Complement, unary plus and minus
* / % //	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

# PYTHON PROGRAMMING & DATA SCIENCE

## Expression:

- An expression is a **combination of variables constants and operators** written according to the syntax of Python language.
- Expressions are evaluated using an assignment statement.

Variable = expression

# PYTHON PROGRAMMING & DATA SCIENCE

## Expression:

Example:

```
a=10 b=22 c=34
x=a*b+c
y=a-b*c
z=a+b+c*c-a
print "x=",x
print "y=",y
print "z=",z
```

Output:

```
x= 254
y= -738
z= 1178
```

# PYTHON PROGRAMMING & DATA SCIENCE

## Input And Output Statements

### Input Statements

➤ The following 2 functions are available to read dynamic input from the keyboard.

1. `raw_input()`
2. `input()`

`raw_input()`

➤ This function always reads the data from the keyboard in the form of String Format.

➤ Type casting is used to convert the string value to specific data type.

# PYTHON PROGRAMMING & DATA SCIENCE

## Input And Output Statements

Example:

```
val2= raw_input("Enter the number: ")
print(type(val2))
val2 = int(val2)
print(type(val2))
print(val2)
```

Input:

enter the number :

243

Output:

<class 'str'>

<class 'int'>



# PYTHON PROGRAMMING & DATA SCIENCE

## Input And Output Statements

### input()

➤ input() function can be used to read data directly in our required format. No need to perform type casting.

Example:

```
x=input("Enter Value")
```

```
type(x)
```

Input	Output
10	int
Durga	str
10.5	float
True	bool

# PYTHON PROGRAMMING & DATA SCIENCE

## Input And Output Statements

### Output Statements

➤ `print( )` is used to output the given data to the standard output device (Screen).

Example:

```
print('cse branch')
```

Output

cse branch

➤ `print( )` is also used to display the combination of message and variable value.

```
num1 = 10      #variable num1 with value 10
```

```
print('The value of variable num1 is ' + num1)
```

Output :

The value of variable num1 is 10

# PYTHON PROGRAMMING & DATA SCIENCE

## format() function in Python

- **str.format()** is one of the *string formatting methods* in Python3, which allows multiple substitutions and value formatting.
- This method concatenate elements within a string through positional formatting.

### Using a Single Formatter :

#### **Syntax :**

**{ }.format(value)**

#### **Parameters :**

**(value)** : Can be an integer, floating point numeric constant, string, characters or even variables.

**Returntype** : Returns a formatted string with the value passed as parameter in the placeholder position.

# PYTHON PROGRAMMING & DATA SCIENCE

## Example 1:

```
print("Sammy has {} balloons.".format(5))
```

## Output:

Sammy has 5 balloons.

## Example 2:

```
x = 5;
```

```
y = 10
```

```
print('The value of x is {} and y is {}'.format(x,y))
```

## Output:

The value of x is 5 and y is 10

# INPUT AND OUTPUT



## Using Multiple Formatters :

### Syntax :

`{{}}.format(value1, value2)`

### Parameters :

**(value1, value2)** : Can be integers, floating point numeric constants, strings, characters and even variables.

➤ the **number of values passed** as parameters in `format()` method **must be equal to** the **number of placeholders** created in the string.

# INPUT AND OUTPUT



## Example 1:

```
# Multiple placeholders in format() function  
my_string = "{} and {} are science portals."  
print (my_string.format("GitHub", "codeacademy"))  
# different datatypes can be used in formatting  
print ("Hi ! My name is {} and I am {} years old".format("User", 19))  
# The values passed as parameters are replaced in order of their entry  
print ("This is {} {} {} {}".format("one", "two", "three", "four"))
```

## Output:

GitHub and codeacademy are science portals.

Hi ! My name is User and I am 19 years old

This is one two three four



# INPUT AND OUTPUT

## Formatters with Positional and Keyword Arguments :

### **Syntax :**

`{0} {1}.format(positional_argument, keyword_argument)`

### **Parameters :**

- **Positional\_argument** can be integers, floating point numeric constants, strings, characters and even variables.
- **Keyword\_argument** is essentially a variable storing some value, which is passed as parameter.



# INPUT AND OUTPUT

## Example:

```
print("Every {} should know the use of {} {} programming and {}"  
      .format("programmer", "Open", "Source", "Operating Systems"))
```

*Every programmer should know the use of Open Source programming and Operating Systems*

*# Use the index numbers of the values to change the order that  
# they appear in the string*

```
print("Every {3} should know the use of {2} {1} programming and {0}"  
      .format("programmer", "Open", "Source", "Operating  
Systems"))
```

*Every Operating Systems should know the use of Source Open  
programming and programmer*



# Control Flow Tools

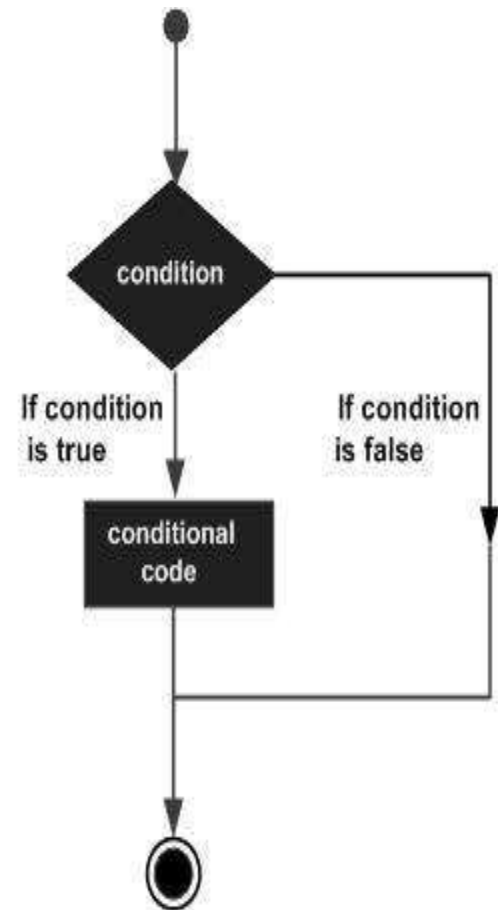
In python there are **2** types of Control Flow Statements  
. Those are

- 1. Conditional Statements or Decision Making Statements**
- 2. Looping Statements**

# Control Flow Tools

## Decision Making:

- Conditional statements or Decision statements are used, when we need to perform different actions depending on whether the specified condition evaluates to true or false.
- the general form of a typical decision making Structure:
- Python programming language assumes any **non-zero** and **non-null** values as **TRUE**, and if it is either **zero** or **null**, then it is assumed as **FALSE** Value.



# Control Flow Tools

Different types of Decision Making Statements are:

1. If statement
2. If...else statement
3. Nested if statements

Note:

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

# Control Flow Tools

## IF Statement:

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

Syntax

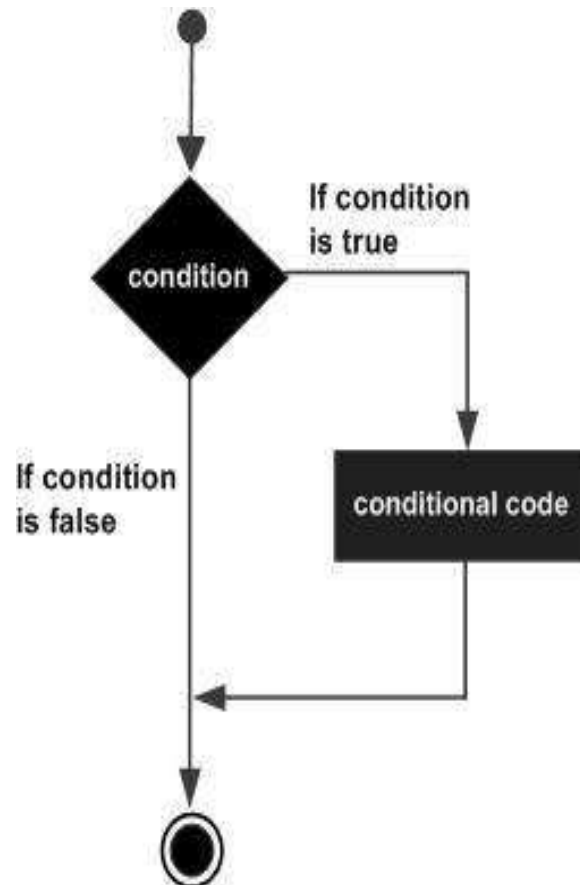
if expression:

statement(s)

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

# Control Flow Tools

The Flowchart is as follows:



# Control Flow Tools

## Example 1:

```
#!/usr/bin/python
var1 = 100
if var1:
    print "1 - Got a true expression
value"
    print var1
var2 = 0
if var2:
    print "2 - Got a true expression
value"
    print var2
print "Good bye!"
```

## Output:

```
1 - Got a true
expression value
100
Good bye!
```

# Control Flow Tools

## Example 2:

```
#!/usr/bin/python
x = int(raw_input("Please enter an
integer: "))
if x < 0:
    print("Enter only +ve number")
print("ok")
```

## Result:

Input:

Please enter an integer: 33

Output:

Ok

Input:

Please enter an  
integer: -200

Output:

Enter only +ve number  
Ok

# Control Flow Tools

## IF...ELSE Statements

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

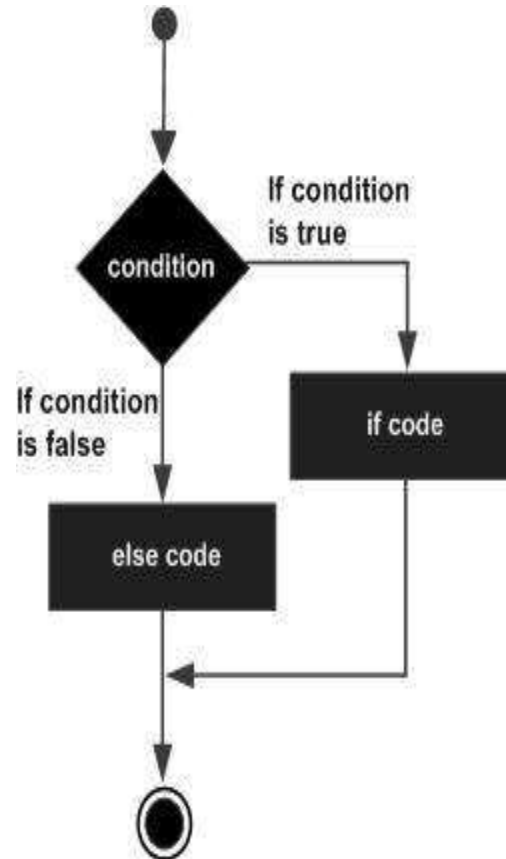
## Syntax

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



# Control Flow Tools

The Flowchart is as follows:



# Control Flow Tools

## Example 1:

```
#!/usr/bin/python
var1 = 100
if var1:
    print "1 - Got a true expression
value"
    print var1
else:
    print "1 - Got a false expression
value"
    print var1
```

## Output:

```
1 - Got a true
expression value
100
```

# Control Flow Tools

## IF...ELIF...ELSE Statements

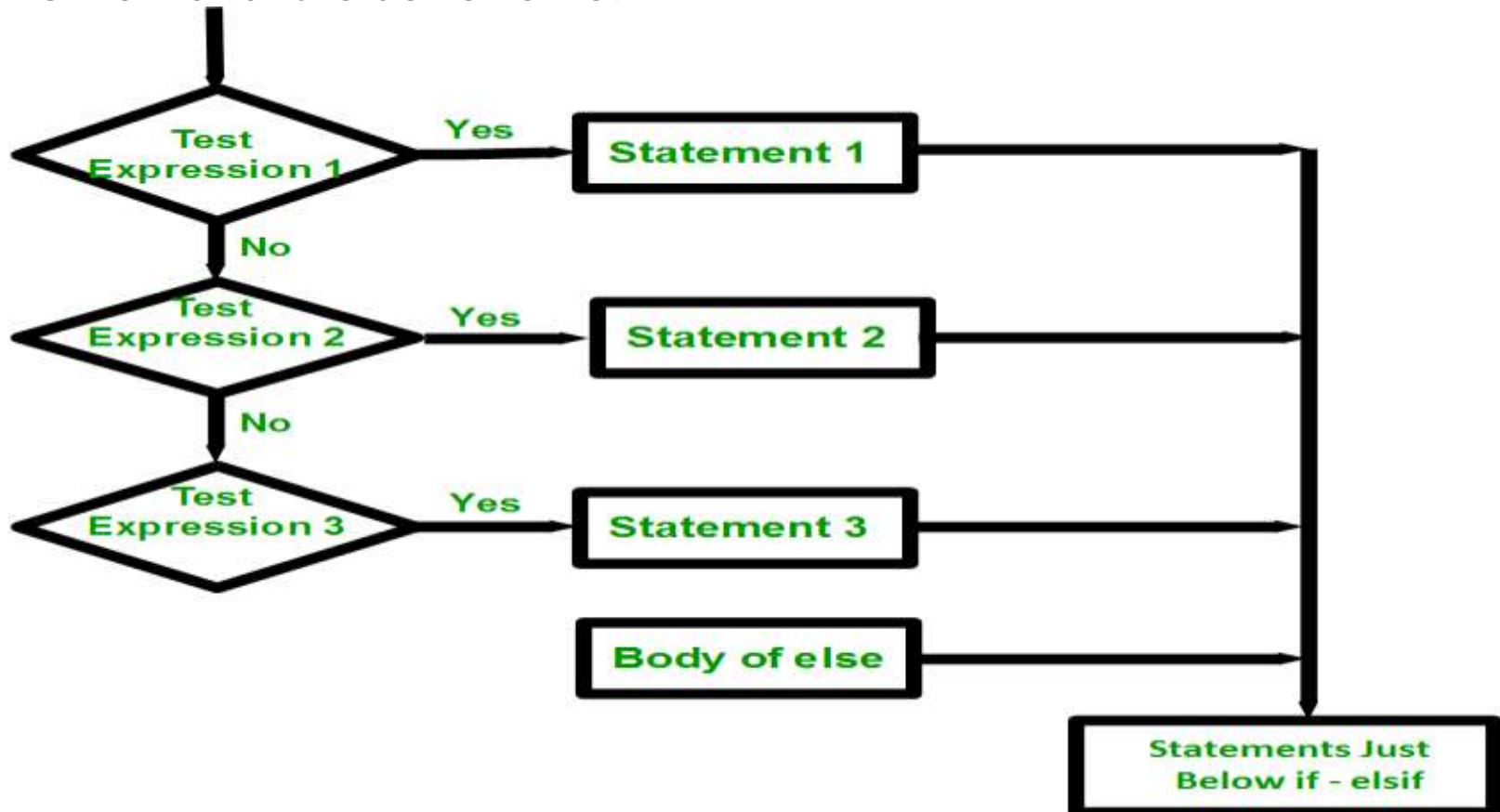
- The **elif** statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE

### Syntax

```
if expression1:  
    statement(s)  
elif expression2:  
    statement(s)  
elif expression3:  
    statement(s)  
else:  
    statement(s)
```

# Control Flow Tools

The flowchart is as follows:



# Control Flow Tools

## Example 1:

```
#!/usr/bin/python
var = 100
if var == 200:
    print "1 - Got a true expression value"
    print var
elif var == 150:
    print "2 - Got a true expression value"
    print var
elif var == 100:
    print "3 - Got a true expression value"
    print var
else:
    print "4 - Got a false expression value"
    print var
print "Good bye!"
```

## Output:

```
3 - Got a true
expression value
100
Good bye!
```

# Control Flow Tools

## nested IF statements:

- There may be a situation when we want to **check for another condition after a condition resolves to true**. In such a situation, we can use the nested **if** statements.
- In a nested **if** statement, we can have an **if...elif...else** construct inside another **if...elif...else** construct.

# Control Flow Tools

Syntax is as follows

```
if expression1:  
    statement(s)  
    if expression2:  
        statement(s)  
    elif expression3:  
        statement(s)  
    elif expression4:  
        statement(s)  
    else:  
        statement(s)  
else:  
    statement(s)
```

# Control Flow Tools

## Example 1:

```
#!/usr/bin/python
var = 100
if var < 200:
    print "Expression value is less than 200"
    if var == 150:
        print "Which is 150"
    elif var == 100:
        print "Which is 100"
    elif var == 50:
        print "Which is 50"
    elif var < 50:
        print "Expression value is less than 50"
else:
    print "Could not find true expression"

print "Good bye!"
```

## Output:

```
Expression value is less than
200
Which is 100
Good bye!
```



# Condition

## Conditional execution

- Conditions

change the flow of program execution

- **if** statement is used in

Syntax:

```
if condition:  
    statement 1  
    ...
```

statement 1

...

## Alternative execution

- Two possibilities (**else** is used)

Syntax:

```
if condition:  
    statement 1  
    ...  
else:  
    statement 2  
    ...
```

## Chained conditionals

- More than two possibilities.
- **elif** is used.

Syntax:

```
if condition:  
    statement 1  
elif condition:  
    statement 2  
elif condition:  
    statement 3  
else:  
    statement 3  
    ...
```

## Nested conditional

- Another Condition in one condition

Syntax:

```
if condition1:  
    if condition2:  
        statement 1  
    else:  
        statement 2  
else:  
    statement 3  
else:  
    statement 3
```

statement 3

# Python - Loops

- Loop statement allows executing a statement or group of statement multiple times.
- Python programming language provides following 3 types of loops.
  1. While loop
  2. For loop
  3. Nested loops

# Python - Loops

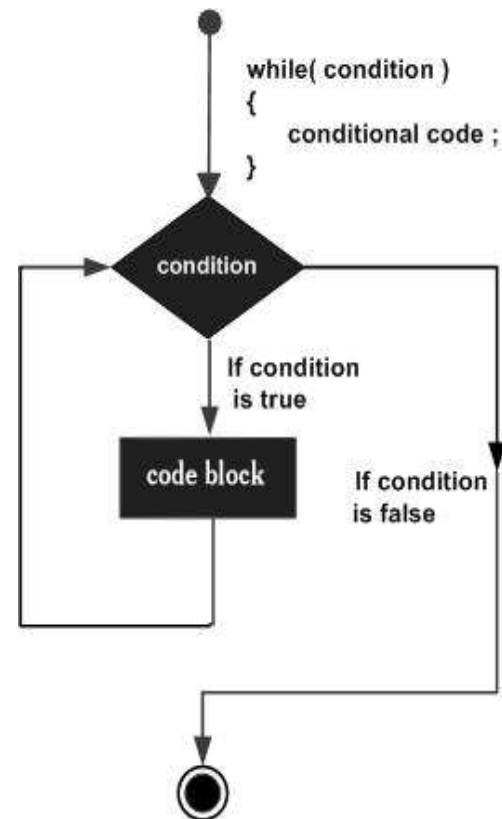
## While Loop

### Syntax:

while expression:  
    statement(s)

- When a given condition **is true** then a **statement or group of statement repeats** which are in the loop body.

### Flow Chart



# Python - Loops

## ➤ Example

```
#!/usr/bin/python
count = 0
while (count < 5):
    print 'The count is:', count
    count = count + 1
print "Good bye!"
```

## Output

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
Good bye!
```

# Python - Loops

## Using else Statement with While Loop

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

### Example

```
#!/usr/bin/python
count = 10
while count < 5:
    print count, " is less than 5"
    count = count + 1
else:
    print count, " is not less than 5"
```

### Output:

10 is not less than 5

# Python - Loops

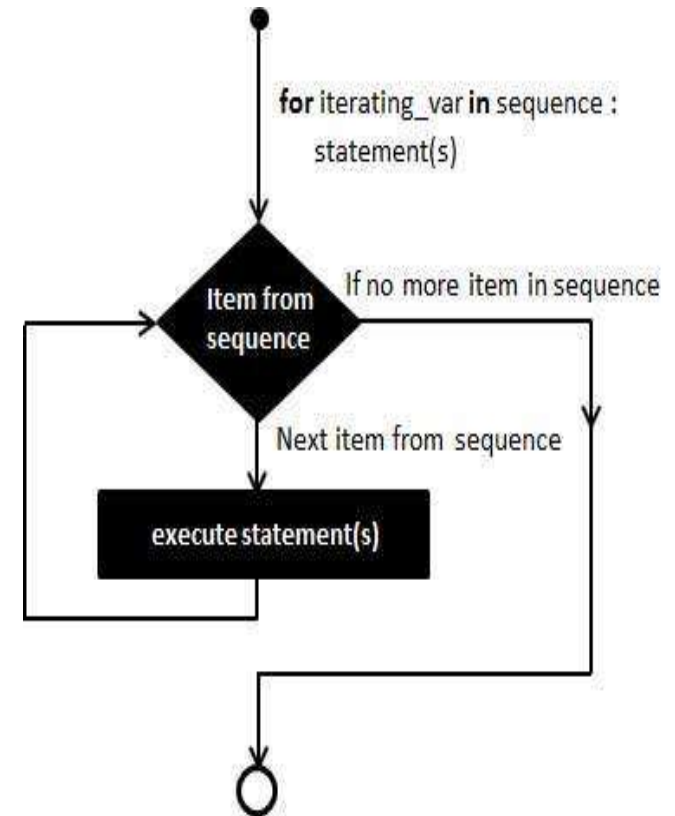
## For Loop

### 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

### Flow Chart



# Python - Loops

## Example 1:

```
s1="ece"  
for letter in s1:  
    print 'Current Letter :',  
    letter
```

## Output:

```
Current Letter : e  
Current Letter : c  
Current Letter : e
```

## Example 2:

```
fruits = ['banana', 'apple',  
          'mango']  
for fruit in fruits:  
    print 'Current fruit :',  
    fruit
```

## Output:

```
Current fruit : banana  
Current fruit : apple  
Current fruit : mango
```

# Python Nested Loops

- Writing one loop inside another loop is called nested loops.
- The syntax for a **nested for loop** statement is as follows –

for iterating\_var in sequence:

    for iterating\_var in sequence:

        statements(s)

    statements(s)

- The syntax for a **nested while loop** statement is as follows

while expression:

    while expression:

        statement(s)

    statement(s)



# Python Nested Loops

## Example:

program to find the prime numbers from 2 to 50

```
#!/usr/bin/python
```

```
i = 2
```

```
while(i < 25):
```

```
    j = 2
```

```
    while(j <= (i/j)):
```

```
        if not(i%j): break
```

```
        j = j + 1
```

```
    if (j > i/j) : 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

# Control Flow Tools

## break and continue Statements, and else Clauses on Loops:

- The break statement is used in loop control statements such as while, and for statements to terminate the execution of the loop.
- When the keyword break is encountered inside any loop, control automatically skip entire loop and passes to the statements available after the loop.

**Syntax:        break;**

- The continue statement is used in while and for statements to terminate the current iteration of the loop.

# Control Flow Tools

## break and continue Statements, and else Clauses on Loops:

- When the keyword `continue` is encountered inside any loop, compiler **skips the remaining statements available after the `continue` statement and control reaches to next iteration** of the loop.
- **Syntax: `continue;`**
- Loop statements may have an `else` clause; it is executed when the loop terminates through exhaustion of the list (with `for`) or when the condition becomes false (with `while`), but not when the loop is terminated by a `break` statement.

# Python - Loops

## Example 1:

```
#!/usr/bin/python
for letter in 'Python':
    if letter == 'h':
        break
    print 'Current Letter
:', letter
```

## Output:

```
Current Letter : P
Current Letter : y
Current Letter : t
```

## Example 2:

```
#!/usr/bin/python
for letter in 'Python':
    if letter == 'h':
        continue
    print 'Current Letter
:', letter
```

## Output:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
```

# Control Flow Tools

## **range() function :**

- range() function is used to iterate over a sequence of numbers.
- It generates lists containing arithmetic progressions.

Example 1:

```
>>> range(10)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Example 2:

```
>>>range(5, 10)
```

```
[5, 6, 7, 8, 9]
```

# Control Flow Tools

➤ To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```
>>>a = ['Mary', 'had', 'a', 'little', 'lamb']  
>>>for i in range(len(a)):  
...     print i, a[i]
```

Output:

```
0 Mary  
1 had  
2 a  
3 little  
4 lamb
```

# Control Flow Tools

## Example:

```
for n in range(2, 10):  
    for x in range(2, n):  
        if n % x == 0:  
            print n, 'equals',  
x, '*', n/x  
            break  
        else:  
            print n, 'is a prime  
number'
```

## Output:

```
2 is a prime number  
3 is a prime number  
4 equals 2 * 2  
5 is a prime number  
6 equals 2 * 3  
7 is a prime number  
8 equals 2 * 4  
9 equals 3 * 3
```

# Control Flow Tools

- `pass` Statement is used when a statement is required syntactically but we do not want any command or code to execute.
- The **`pass`** statement is a *null operation*; nothing happens when it executes.



# Python - Loops

## Example :

```
#!/usr/bin/python
for letter in 'Python':
    if letter == 'h':
        pass
    print 'This is pass
block'
    print 'Current Letter :',
letter
```

## Output:

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
```

# Control Flow Tools

- `pass` Statement is used when a statement is required syntactically but we do not want any command or code to execute.
- The **`pass`** statement is a *null operation*; nothing happens when it executes.

# Built-in Data Types



## Built-in Data Types:

➤ Python has the following 4 built-in data types:

1. Text Type : str
2. Numeric Types : int, float
3. complexSequence Types : list, tuple, range
4. Mapping Type : dict

➤ complexSequence Types and Mapping Type are called as Data Structures in Python.



# DATA STRUCTURES

## List Data Type:

- Lists are used to store multiple items in a single variable.
- Lists are created using square brackets:

Example:

## Create a List:

```
thislist = ["apple", "banana", "cherry"]  
print(thislist)
```

Output:

```
['apple', 'banana', 'cherry']
```

- List items are ordered, changeable, and allow duplicate values.



# DATA STRUCTURES

## List Data Type:

- List items are indexed, the first item has index [0], the second item has index [1]... etc.

## Access List Items:

- List items are **accessed** by referring to the index number:

### Example 1:

```
#!/usr/bin/python  
thislist = ["apple", "banana", "cherry"]  
print(thislist[1])
```

### Output:

banana



# DATA STRUCTURES

## List Data Type:

### Example 2:

```
#!/usr/bin/python  
list1 = ['physics', 'chemistry', 1997, 2000]  
list2 = [10, 20, 30, 40, 50, 60, 70 ]  
print ("list1[0]: ", list1[0])  
Print( "list2[1]: ", list2[1])  
Print(list2[0:4])
```

### Output:

```
list1[0]: physics  
list2[1]: 20
```



# DATA STRUCTURES

## List Data Type:

- Negative indexing means **start from the end**
- -1 refers to the last item, -2 refers to the second last item etc.
- [start:end:step]

## **Example :**

```
#!/usr/bin/python  
thislist = ["apple", "banana", "cherry"]  
print(thislist[-1])
```

## **Output:**

cherry



# DATA STRUCTURES

## List Data Type:

- We can specify a range of indexes by specifying where to start and where to end the range.
- When specifying a range, the return value will be a new list with the specified items.
- The search begins from starting index (included) to ending index (not included).

### Example :

```
#!/usr/bin/python
```

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]  
print(thislist[2:5])
```

### Output:

```
['cherry', 'orange', 'kiwi']
```





# DATA STRUCTURES

## List operations

The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

the \* operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```



# DATA STRUCTURES

## Updating Lists

- We can **update single or multiple elements** of lists by giving the slice on the left-hand side of the assignment operator, and can add to elements in to list.

### 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]
```

### Output:

Value available at index 2 : 1997

New value available at index 2 : 2001

# DATA STRUCTURES



## Delete List Elements

- To remove a list element, we can use either the **del statement** or the **remove()** method.
- **del statement** is used if we know **exactly which element(s)** to delete. **otherwise** use **remove()** method.

**Example :**

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

**Output:**

```
['physics', 'chemistry', 1997, 2000]
```

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

# DATA STRUCTURES



## The del statement:

For example:

```
a = [-1, 1, 66.25, 333, 333, 1234.5]
```

```
del a[0]
```

```
a
```

```
[1, 66.25, 333, 333, 1234.5]
```

```
del a[2:4]
```

```
a
```

```
[1, 66.25, 1234.5]
```

del can also be used to delete entire variables:

```
del a
```



# DATA STRUCTURES

## List Methods

➤ The list data type has some more methods. Some of them are:

### 1. **append(x)**

Add an item to the **end of the list**;

### 2. **extend(L)**

Extend the list by **appending all the items in the given list**;

### 3. **insert(i, x)**

**Insert an item at a given position.** The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

# DATA STRUCTURES



## List Methods

### 4. `remove(x)`

Remove the first item from the list whose value is `x`. It is an error if there is no such item.

### 5. `pop([i])`

Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list.

### 6. `index(x)`

Return the index in the list of the first item whose value is `x`. It is an error if there is no such item.

# DATA STRUCTURES



## List Methods

### 7. `count(x)`

Return the number of times `x` appears in the list.

### 8. `sort()`

Sort the items of the list, in place.

### 9. `reverse()`

Reverse the elements of the list, in place.



# DATA STRUCTURES

An **example** that uses most of the list methods:

```
a = [66.25, 333, 333, 1, 1234.5]
```

```
print a.count(333)
```

```
print a.count(66.25)
```

```
print a.count('x')
```

**Output:**

```
2 1 0
```

```
a.insert(2, -1)
```

```
a.append(333)
```

```
a
```

**Output:**

```
[66.25, 333, -1, 333, 1, 1234.5, 333]
```



# DATA STRUCTURES



```
[66.25, 333, -1, 333, 1, 1234.5, 333]
```

```
a.index(333)
```

```
1
```

```
a.remove(333)
```

```
a
```

```
[66.25, -1, 333, 1, 1234.5, 333]
```

```
a.reverse()
```

```
a
```

```
[333, 1234.5, 1, 333, -1, 66.25]
```

```
a.sort()
```

```
a
```

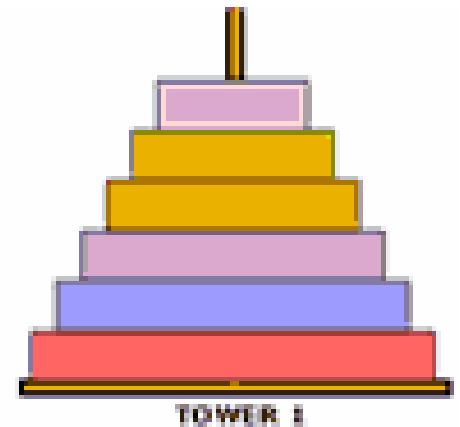
```
[-1, 1, 66.25, 333, 333, 1234.5]
```

# DATA STRUCTURES



## Using Lists as Stacks:

- The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”).
- 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.



# DATA STRUCTURES



Using Lists as Stacks:

For example:

```
stack = [3, 4, 5]
```

```
stack.append(6)
```

```
stack.append(7)
```

```
Print(stack)
```

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

```
stack.pop()
```

```
7
```

# DATA STRUCTURES



Using Lists as Stacks:

stack

[3, 4, 5, 6]

stack.pop()

6

stack.pop()

5

stack

[3, 4]

# DATA STRUCTURES



## Using Lists as Queue:

- It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”).
- While appends and pops from the end of list is fast, doing inserts or pops from the beginning of a list is slow.
- To implement a queue, use collections. deque which was designed to have fast appends and pops from both ends.



# DATA STRUCTURES



## Using Lists as Queue:

### Example:

```
from collections import deque
queue = deque(["Linux", "Programming", "Script"])
print("Original Queue values are:")
print(queue)
queue.append("Ece")
queue.append("Section")
print("After adding Queue values are:")
print(queue)
queue.popleft()
queue.popleft()
print("After Removing Queue values are:")
print(queue)
```

## Output:

Original Queue values are:

```
deque(['Linux', 'Programming', 'Script'])
```

After adding Queue values are:

```
deque(['Linux', 'Programming', 'Script', 'Ece', 'Section'])
```

After Removing Queue values are:

```
deque(['Script', 'Ece', 'Section'])
```



# DATA STRUCTURES

## Functional (List) Programming Tools

➤ There are **three** built-in functions that are very useful .

1. filter() 2. map() 3. reduce().

### 1. Filter(function, sequence)

➤ Returns a sequence consisting of the items from the sequence for which function(item) is true.

#### **Example:**

```
def f(x): return x % 2 != 0 and x % 3 != 0  
print(filter(f, range(2, 25)))
```

#### **Output:**

[5, 7, 11, 13, 17, 19, 23]

Computes primes up to 25



# DATA STRUCTURES

## Functional (List) Programming Tools

### 2. Map(function, sequence)

- Calls function(item) for each of the sequence's items

#### Example:

```
def cube(x): return x*x*x  
print(map(cube, range(1, 11)))
```

#### Output:

[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]

Computes the cube for the range of 1 to 11





# DATA STRUCTURES

## Functional (List) Programming Tools

### Reduce(function, sequence)

- Returns a single value constructed by calling the binary function (function)

#### Example:

```
def add(x,y): return x+y  
print(reduce(add, range(1, 11)))
```

#### Output:

55

Computes the sum of the numbers 1 to 10

# DATA STRUCTURES



## Loop Lists

➤ we can loop through the list items by using a **for loop**:

**For example:**

```
thislist = ["apple", "banana", "cherry"]
```

```
for x in thislist:
```

```
    print(x)
```

**Output:**

apple

banana

Cherry

# DATA STRUCTURES



- we can also loop through the list items by referring to their index number.
- Use the `range()` and `len()` functions to create a suitable iterable.

For example:

```
thislist = ["apple", "banana", "cherry"]  
for i in range(len(thislist)):  
    print(thislist[i])
```

Output:

```
apple  
banana  
Cherry
```



# DATA STRUCTURES

➤ we can loop through the **list items by using a while loop.**

**For example:**

```
thislist = ["apple", "banana", "cherry"]
```

```
i = 0
```

```
while i < len(thislist):
```

```
    print(thislist[i])
```

```
    i = i + 1
```

**Output:**

apple

banana

Cherry



# DATA STRUCTURES

## List Comprehensions:

- List comprehensions provide a concise way to create lists without resorting to use of map(), filter() and/or lambda.
- List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

The Syntax

`newlist = [expression for item in iterable if condition == True]`

- The return value is a new list, leaving the old list unchanged.



# DATA STRUCTURES

Without list comprehension:

**Example:**

```
fruits = ["apple", "banana",  
"cherry", "kiwi", "mango"]  
newlist = []
```

```
for x in fruits:
```

```
    if "a" in x:
```

```
        newlist.append(x)
```

```
print(newlist)
```

**Output:**

```
['apple', 'banana', 'mango']
```

With list comprehension:

**Example:**

```
fruits = ["apple", "banana", "cherry",  
"kiwi", "mango"]
```

```
newlist = [x for x in fruits if "a" in x]
```

```
print(newlist)
```

**Output:**

```
['apple', 'banana', 'mango']
```



# DATA STRUCTURES

## Tuple:

- Tuples are used to store **multiple items in a single variable**.
- A tuple is a collection which **is ordered and unchangeable**.
- Tuples are written with **round brackets**.

## Example

### Create a Tuple:

```
thistuple = ("apple", "banana", "cherry")  
print(thistuple)
```

### Output:

```
('apple', 'banana', 'cherry')
```



# DATA STRUCTURES

➤ Tuple items are ordered, unchangeable, and allow duplicate values.

➤ Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

## Example

```
my_tuple = ('p','e','r','m','i','t')
print(my_tuple[0])
print(my_tuple[5])
n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(n_tuple[0][3])
print(n_tuple[1][1])
print(my_tuple[-1])
print(my_tuple[-6])
```

## Output:

p  
t  
s  
4  
t  
p



- Accessing Values in Tuples
- To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –
- [Live Demo](#)`#!/usr/bin/python`
- `tup1 = ('physics', 'chemistry', 1997, 2000)`
- `tup2 = (1, 2, 3, 4, 5, 6, 7 )`
- `print ("tup1[0]: ", tup1[0])`
- `print ("tup2[1:5]: ", tup2[1:5])`

# Built-in functions in tuples

Sr.No.	Function with Description
1	<a href="#"><u>cmp(tuple1, tuple2)</u></a> Compares elements of both tuples.
2	<a href="#"><u>len(tuple)</u></a> Gives the total length of the tuple.
3	<a href="#"><u>max(tuple)</u></a> Returns item from the tuple with max value.
4	<a href="#"><u>min(tuple)</u></a> Returns item from the tuple with min value.
5	<a href="#"><u>tuple(seq)</u></a> Converts a list into tuple.

```
t1 = tuple()
print('t1=', t1)
# creating a tuple from a list
t2 = tuple([1, 6, 9])
print('t2=', t2)
# creating a tuple from a string
t1 = tuple('Java')
print('t1=',t1)
# creating a tuple from a dictionary
t1 = tuple({4: 'four', 5: 'five'})
print('t1=',t1)
```

# Tuple methods

- **Count() Method**
- The count() method of Tuple returns the number of times the given element appears in the tuple.
- **Syntax:** tuple.count(element)
- # Creating tuples
- Tuple1 = (0, 1, 2, 3, 2, 3, 1, 3, 2)
- Tuple2 = ('python', 'geek', 'python', 'for', 'java', 'python')
- # count the appearance of 3
- res = Tuple1.count(3)
- print('Count of 3 in Tuple1 is:', res)
- # count the appearance of python
- res = Tuple2.count('python')
- print('Count of Python in Tuple2 is:', res)

## **Example 2: Counting tuples and lists as elements in Tuples**

- `# Creating tuples`
- `Tuple = (0, 1, (2, 3), (2, 3), 1, [3, 2], 'geeks', (0,))`
- `# count the appearance of (2, 3)`
- `res = Tuple.count((2, 3))`
- `print('Count of (2, 3) in Tuple is:', res)`
- `# count the appearance of [3, 2]`
- `res = Tuple.count([3, 2])`
- `print('Count of [3, 2] in Tuple is:', res)`

- **Index() Method**
- The Index() method returns the first occurrence of the given element from the tuple.
- **Syntax:**

`tuple.index(element, start, end)`

`# Creating tuples`

`Tuple = (0, 1, 2, 3, 2, 3, 1, 3, 2)`

`# getting the index of 3`

`res = Tuple.index(3)`

`print('First occurrence of 3 is', res)`

`# getting the index of 3 after 4th`

`# index`

`res = Tuple.index(3, 4)`

`print('First occurrence of 3 after 4th index is:', res)`

# DATA STRUCTURES



## Dictionary:

- Dictionaries are used to store data values in key:value pairs.
- A dictionary is a collection which is unordered, changeable and does not allow duplicates.
- Dictionaries are written with curly brackets, and have keys and values:

## Example

### Create a Dictionary:

```
thisdict = {  
    "Rno": 501,  
    "Name": "KUMAR ",  
    "Group": "CSE"  
}  
print(thisdict)
```

### Output:

```
{'Roll no': 501, 'Name':  
'KUMAR', 'Group': 'CSE'}
```



## DATA STRUCTURES

- Dictionary items are **unordered**, **changeable**, and **does not allow duplicates**.
- Dictionary items are presented **in key:value pairs**, and can be **referred to by using the key name**.

### Example

```
my_dict = {'name': 'Jack', 'age': 26}  
print(my_dict['name'])  
print(my_dict.get('age'))
```

### Output:

```
Jack  
26
```



```
Employee = {"Name": "John", "Age": 29, "
salary":25000,"Company":"GOOGLE"}
print(type(Employee))
print("printing Employee data .... ")
print(Employee)
```

```
# Creating an empty Dictionary
```

```
Dict = {}
```

```
print("Empty Dictionary: ")
```

```
print(Dict)
```

```
# Creating a Dictionary
```

```
# with dict() method
```

```
Dict = dict({1: 'keyboard', 2: 'cpu', 3:'monitor'})
```

```
print("\\nCreate Dictionary by using dict(): ")
```

```
print(Dict)
```

```
# Creating a Dictionary
```

```
# with each item as a Pair
```

```
Dict = dict([(1, 'Devansh'), (2, 'Sharma')])
```

```
print("\\nDictionary with each item as a pair: ")
```

```
print(Dict)
```

```
Employee = {"Name": "John", "Age": 29, "salary":25000  
, "Company": "GOOGLE"}  
print(type(Employee))  
print("printing Employee data .... ")  
print("Name : %s" %Employee["Name"])  
print("Age : %d" %Employee["Age"])  
print("Salary : %d" %Employee["salary"])  
print("Company : %s" %Employee["Company"])
```

Python allows dictionary comprehensions. We can create dictionaries using simple expressions. A dictionary comprehension takes the form ***{key: value for (key, value) in iterable}***

### **Python Dictionary Comprehension Example**

Here we have two [lists](#) named keys and value and we are iterating over them with the help of [zip\(\)](#) function

```
# Lists to represent keys and values
```

```
keys = ['a', 'b', 'c', 'd', 'e']
```

```
values = [1,2,3,4,5]
```

```
# but this line shows dict comprehension here
```

```
myDict = { k:v for (k,v) in zip(keys, values)}
```

```
# We can use below too
```

```
# myDict = dict(zip(keys, values))
```

```
print (myDict)
```

```
name = [ "Manjeet", "Nikhil", "Shambhavi", "Astha" ]  
roll_no = [ 4, 1, 3, 2 ]
```

```
# using zip() to map values  
mapped = dict(zip(name, roll_no))
```

```
print(mapped)
```

```
name = ["Manjeet", "Nikhil", "Shambhavi", "Astha"]
roll_no = [4, 1, 3, 2]
marks = [40, 50, 60, 70]

# using zip() to map values
mapped = zip(name, roll_no, marks)

# converting values to print as list
mapped = list(mapped)

# printing resultant values
print("The zipped result is : ", end="")
print(mapped)
```

Name Age salary Company

## Iterating Dictionary:

A dictionary can be iterated using for loop as given below.

Example 1

**# for loop to print all the keys of a dictionary**

```
Employee = {"Name": "John", "Age": 29, "salary": 25000, "Company": "GOOGLE"}
```

```
for x in Employee:
```

```
    print(x)
```

Output: name, age, salary, company

Name Age salary Company



Example 2

**#for loop to print all the values of the dictionary**

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}  
for x in Employee:  
    print(Employee[x])
```

Output: john, 29, 25000, Google

### Example - 3

#for loop to print the values of the dictionary by using values() method.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}  
for x in Employee.values():  
    print(x)
```

**Output:**john, 29, 25000, Google

John 29 25000 GOOGLE

Example 4

**#for loop to print the items of the dictionary by using items() method.**

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
```

```
for x in Employee.items():  
    print(x)
```

# Python Set

- A Python set is the collection of the unordered items. Each element in the set must be unique, immutable, and the sets remove the duplicate elements. Sets are mutable which means we can modify it after its creation.

## Creating a set

The set can be created by enclosing the comma-separated immutable items with the curly braces {}.

Python also provides the set() method, which can be used to create the set by the passed sequence.

# Example 1: Using curly braces

```
Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}  
print(Days)  
print(type(Days))  
print("looping through the set elements ... ")  
for i in Days:  
    print(i)
```

## Example 2: Using set() method

```
Days = set(["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"])
print(Days)
print(type(Days))
print("looping through the set elements ... ")
for i in Days:
    print(i)
```

# Empty curly braces will create dictionary

```
set3 = {}
```

```
print(type(set3))
```

# Empty set using set() function

```
set4 = set()
```

```
print(type(set4))
```

# Adding items to the set

- Python provides the **add()** method and **update()** method which can be used to add some particular item to the set. The add() method is used to add a single element whereas the update() method is used to add multiple elements to the set. Consider the following example.

- Example: 1 - Using add() method

```
Months = set(["January", "February", "March", "April", "May", "June"])
```

```
print("\nprinting the original set ... ")
```

```
print(months)
```

```
print("\nAdding other months to the set...");
```

```
Months.add("July");
```

```
Months.add ("August");
```

```
print("\nPrinting the modified set...");
```

```
print(Months)
```

```
print("\nlooping through the set elements ... ")
```

```
for i in Months:
```

```
    print(i)
```



## Example - 2 Using update() function

```
Months = set(["January","February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(Months)
print("\nupdating the original set ... ")
months.update(["July","August","September","October"]);
print("\nprinting the modified set ... ")
print(Months);
```

# Removing items from the set

- Python provides the **discard()** method and **remove()** method which can be used to remove the items from the set.
- The difference between these function, using **discard()** function if the item does not exist in the set then the set remain unchanged whereas **remove()** method will through an error.

# Example-1 Using discard() method

```
months = set(["January", "February", "March", "April", "May", "June"])

print("\nprinting the original set ... ")
print(months)
print("\nRemoving some months from the set...");
months.discard("January");
months.discard("May");
print("\nPrinting the modified set...");
print(months)
print("\nlooping through the set elements ... ")
for i in months:
    print(i)
```

# Output:

- printing the original set ...
- {'February', 'January', 'March', 'April', 'June', 'May'}
- Removing some months from the set...  
Printing the modified set... {'February', 'March', 'April', 'June'}
- looping through the set elements ...
- February March April June

## Example-2 Using remove() function

```
months = set(["January", "February", "March", "April", "May", "June"]) print("\\nprinting the original set ... ")  
print(months)  
print("\\nRemoving some months from the set...");  
  
months.remove("January");  
months.remove("May");  
print("\\nPrinting the modified set...");  
print(months)
```

**Pop() method, Python provides the clear() method to remove all the items from the set.**

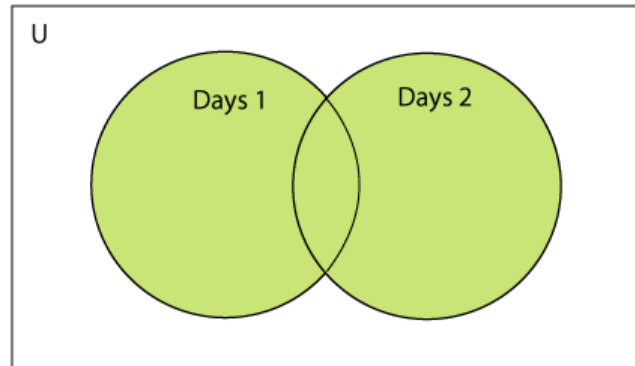
- We can also use the pop() method to remove the item. Generally, the pop() method will always remove the last item but the set is unordered, we can't determine which element will be popped from set.
- Consider the following example to remove the item from the set using pop() method.

```
Months = set(["January","February", "March", "April", "May", "June"])
print("\nprinting the original set ... ")
print(Months)
print("\nRemoving some months from the set...");
Months.pop();
Months.pop();
print("\nPrinting the modified set...");
print(Months)
```

- Difference between `discard()` and `remove()`
- Despite the fact that **`discard()`** and **`remove()`** method both perform the same task, There is one main difference between `discard()` and `remove()`.
- If the key to be deleted from the set using `discard()` doesn't exist in the set, the Python will not give the error. The program maintains its control flow.
- On the other hand, if the item to be deleted from the set using `remove()` doesn't exist in the set, the Python will raise an error.

# Python Set Operations

- set can be performed mathematical operation such as union, intersection, difference, and symmetric difference. Python provides the facility to carry out these operations with operators or methods. We describe these operations as follows.
- Union of two Sets
- The union ( $|$ ) operator combines the items of the sets and by using the pipe ( $|$ ) operator, the resulting set contains all the items of both sets.





- **Example 1: using union | operator**

```
Days1 = {"Monday", "Tuesday", "Wednesday", "Thursday", "Sunday"}
```

```
Days2 = {"Friday", "Saturday", "Sunday"}
```

```
print(Days1 | Days2)                #printing the union of the sets
```

# Union() method

- Python also provides the **union()** method which can also be used to calculate the union of two sets. Consider the following example.
- **Example 2: using union() method**

```
Days1 = {"Monday","Tuesday","Wednesday","Thursday"}
```

```
Days2 = {"Friday","Saturday","Sunday"}
```

```
print(Days1.union(Days2))           #printing the union of the sets
```

# Intersection of two sets

- Intersection of two sets
- The intersection of two sets can be performed by the **and &** operator or the **intersection() function**. The intersection of the two sets is given as the set of the elements that common in both sets.

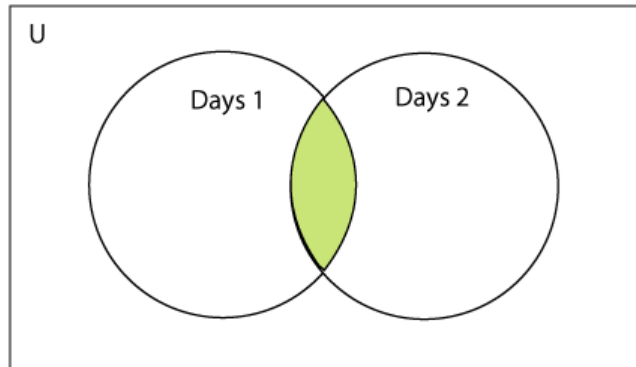
```
Days1 = {"Monday","Tuesday", "Wednesday", "Thursday"}
```

```
Days2 = {"Monday","Tuesday","Sunday", "Friday"}
```

```
print(Days1&Days2)      #prints the intersection of the t  
wo sets
```

# Intersection() method

- `set1 = {"Devansh", "John", "David", "Martin"}`
- `set2 = {"Steve", "Milan", "David", "Martin"}`
- `print(set1.intersection(set2))` #prints the intersection of



# The `intersection_update()` method

- The **`intersection_update()`** method removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).

```
a = {"Devansh", "bob", "castle"}  
b = {"castle", "dude", "emyway"}  
c = {"fuson", "gaurav", "castle"}  
a.intersection_update(b, c)  
print(a)
```

Output:

```
{'castle'}
```

# Python Built-in set methods

SN	Method	Description
1	<a href="#">add(item)</a>	It adds an item to the set. It has no effect if the item is already present in the set.
2	<code>clear()</code>	It deletes all the items from the set.
3	<code>copy()</code>	It returns a shallow copy of the set.
4	<code>difference_update(...)</code>	It modifies this set by removing all the items that are also present in the specified sets.
5	<a href="#">discard(item)</a>	It removes the specified item from the set.
6	<code>intersection()</code>	It returns a new set that contains only the common elements of both the sets. (all the sets if more than two are specified).
7	<code>intersection_update(...)</code>	It removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).
8	<code>Isdisjoint(...)</code>	Return True if two sets have a null intersection.
9	<code>Issubset(...)</code>	Report whether another set contains this set.
10	<code>Issuperset(...)</code>	Report whether this set contains another set.
11	<a href="#">pop()</a>	Remove and return an arbitrary set element that is the last element of the set. Raises <code>KeyError</code> if the set is empty.
12	<a href="#">remove(item)</a>	Remove an element from a set; it must be a member. If the element is not a member, raise a <code>KeyError</code> .
13	<code>symmetric_difference(...)</code>	Remove an element from a set; it must be a member. If the element is not a member, raise a <code>KeyError</code> .
14	<code>symmetric_difference_update(...)</code>	Update a set with the symmetric difference of itself and another.
15	<code>union(...)</code>	Return the union of sets as a new set. (i.e. all elements that are in either set.)
16	<code>update()</code>	Update a set with the union of itself and others.