UNIT-I

What is Data Science?

Data science is the study of data, combining principles from statistics, mathematics, computer science, and domain expertise to analyze large datasets, uncover hidden patterns, and extract meaningful insights. It's a multidisciplinary field that uses various techniques, algorithms, and tools to extract valuable information from both structured and unstructured data.

Importance of Data Science:

Informed Decision Making: Data science helps businesses make better decisions by understanding trends, identifying opportunities, and mitigating risks.

Improved Efficiency: Organizations can use data science to optimize processes, reduce costs, and improve resource allocation.

Competitive Advantage: By leveraging data insights, businesses can innovate, develop new products, and gain a competitive edge in the market.

Advantages of Data Science:

Better understanding of customers: Data science helps businesses understand customer behavior, preferences, and needs, leading to more personalized experiences and targeted marketing campaigns.

Predictive analytics: Data science enables businesses to predict future trends and outcomes, allowing for proactive planning and decision-making.

Fraud detection: Data science techniques can be used to identify and prevent fraudulent activities in various sectors, such as finance and insurance.

Improved product development: By analyzing data, businesses can identify areas for product improvement, develop new features, and enhance user experience.

Automation of processes: Data science can automate various tasks and processes, freeing up human resources for more strategic initiatives.

The Process of Data Science:

1. **Define the problem:** Clearly identify the business problem or question that needs to be addressed.
2. **Gather data:** Collect relevant data from various sources, ensuring data quality and completeness.
3. **Clean and preprocess data:** Prepare the data for analysis by handling missing values, outliers, and inconsistencies.
4. **Explore and analyze data:** Use statistical methods and data visualization techniques to uncover patterns and insights.
5. **Build and evaluate models:** Develop predictive models using machine learning algorithms and evaluate their performance.
6. **Interpret and communicate results:** Translate the findings into actionable insights and communicate them to stakeholders.

7. Implement and monitor: Put the insights into action and continuously monitor the performance of the models.

UNIT-II

Introduction to python

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum in February 1991.

Advantages of python:

- **Python is Interpreted** − 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 is Interactive** − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language** − Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Characteristics of Python

The following are important characteristics of **Python Programming** −

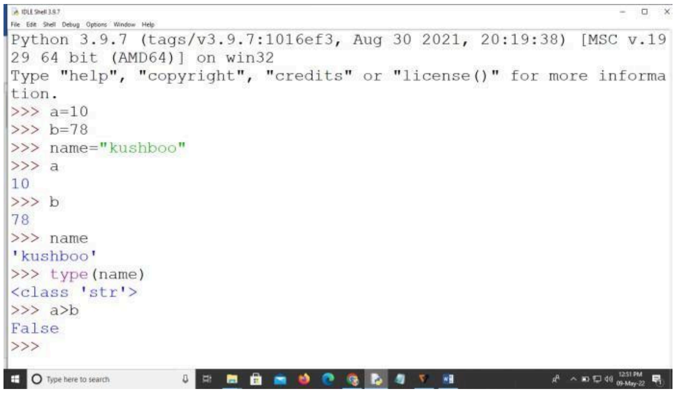
- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Modes of Python Program : To develop **a python program** in 2 different styles.

1. Interactive Mode and
2. Script Mode.

1. Interactive Mode - Interactive mode is a command line shell. Typically the interactive mode is used to test the features of the python, or to run a smaller script that may not be reusable.

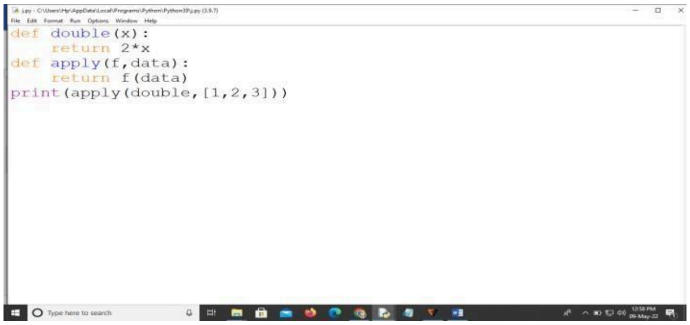
Example for Interactive Mode :



```
Python 3.9.7 (tags/v3.9.7:1016ef3, Aug 30 2021, 20:19:38) [MSC v.19
29 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more informa
tion.
>>> a=10
>>> b=78
>>> name="kushboo"
>>> a
10
>>> b
78
>>> name
'kushboo'
>>> type(name)
<class 'str'>
>>> a>b
False
>>>
```

2. Script Mode - Script mode is mainly used to develop business applications. In script mode, we can write a group of python statements in any one of the following editors or IDEs

Example for script mode:


j.py - C:\Users\Hp\AppData\Local\Programs\Python\Python39\j.py (3.9.7)
File Edit Format Run Options Window Help
Type here to search

```
def double(x):
    return 2*x
def apply(f,data):
    return f(data)
print(apply(double,[1,2,3]))
```

Python Features

Python's features include −

- **Easy-to-learn** − Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** − Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain** − Python's source code is fairly easy-to-maintain.
- **A broad standard library** − Python's bulk of the library is very portable and cross- platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable** − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable** − You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases** − Python provides interfaces to all major commercial databases.
- **GUI Programming** − Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable** − Python provides a better structure and support for large programs than shell scripting.
- **Free**- Python can be downloaded from internet freely without paying money.
- **Opensource**- Python is open source language and we can see the python code.
- **Advance features**- Python provides advanced features to develop machine learning programs easily.

Python - Variable Types

Variables are nothing but reserved memory locations to store values. Based on the data type of a variable, the interpreter allocates memory and decides the value can be stored in the reserved memory.

Therefore, by assigning different data types to variables, it can store integers, decimals or characters in these variables.

Rules for Python variables:

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

Assigning Values to Variables

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

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

For example :

```
counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string
```

Here, 100, 1000.0 and "John" are the values assigned to *counter*, *miles*, and *name* variables, respectively.

Multiple Assignment

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

For example :

a = b = c = 1

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

For example :

a,b,c = 1,2,"john"

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

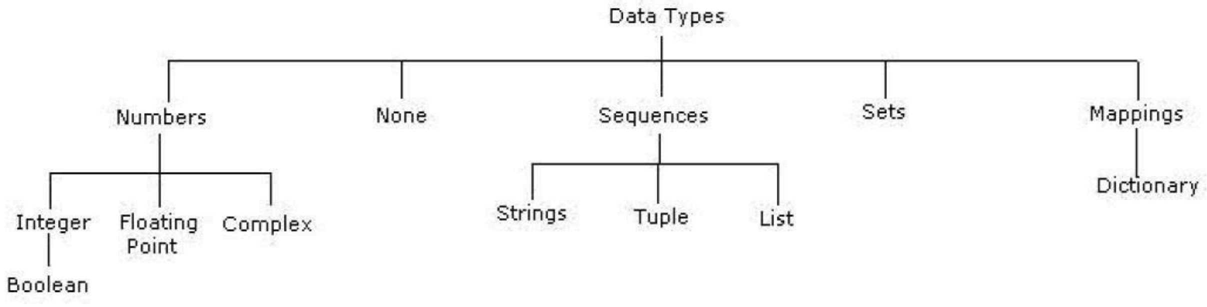
Standard Data Types of python:

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters.

Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types :

- Numbers
- String
- List
- Tuple
- Dictionary
- Sets



Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them.

For example:

var1 = 1

var2 = 10

You 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[,varN]]]]

You can delete a single object or multiple objects by using the del statement. For example :

del var

del var_a, var_b

Python supports four different numerical types −

int (signed integers)

long (long integers, they can also be represented in octal and hexadecimal)

float (floating point real values)

complex (complex numbers)

Examples

Here are some examples of numbers :

Int	Long	float	Complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	4j
-786	0122L	-21.9	9.322-36j
-0490	535633629843L	-90.	-.6545+0j
0x260	-052318172735L	-32.54e100	3+26j

Booleans

These represent the truth values False and True. Boolean values False and True behave like the values 0 and 1, respectively. To get the Boolean Equivalent of 0 or 1, you can type bool(0) or bool(1), python will return False or True respectively.

Python Dictionary:

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be any immutable Python type, usually numbers,strings and tuples.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).

For example :

```
dict = {}
dict['one'] = "This is one"
dict[2]          = "This is two"
tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
```

Sets:

Sets are used to store multiple items in a single variable. A set is a collection which is *unordered* and *unindexed*.

Example:

```
thisset = {"apple", "banana", "cherry"} print(thisset)
```

Python Strings:

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

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

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

For example :

```
str = 'Hello World!' Python
```

Lists:

Lists are the mutable datatype of Python. A list contains items separated by commas and enclosed within square brackets ([]).

To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the index ([ ]).Indexes starting at 0 in the beginning of the list and working their way to end -1.

Example :

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ] tinylist =
[123, 'john']
```

Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists.

For example :

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')
```

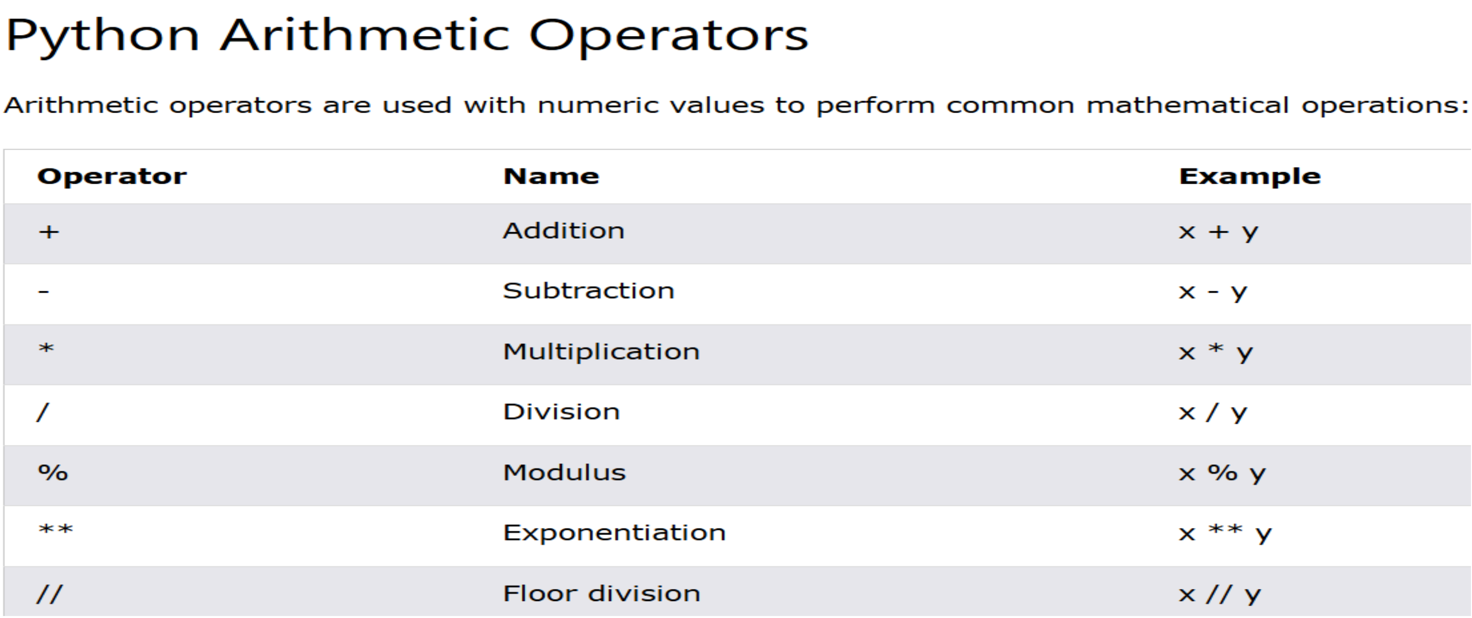
Python - Basic Operators

Operators are the constructs which can manipulate the value of operands.

Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called operator.

Types of Operators: Python language supports the following types of operators.

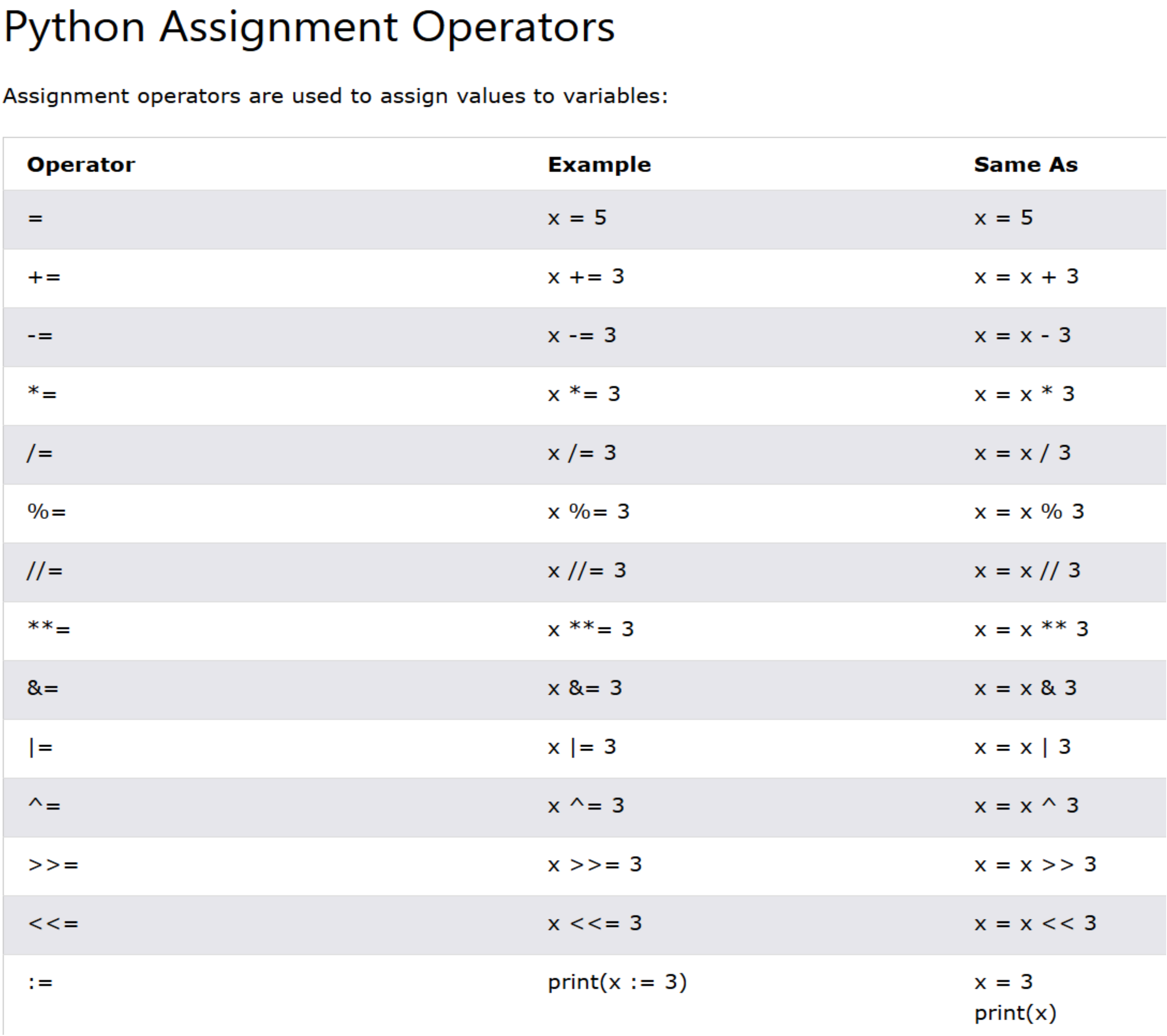
- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators



Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y



Python Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

:=	print(x := 3)	x = 3
		print(x)

Python Comparison Operators

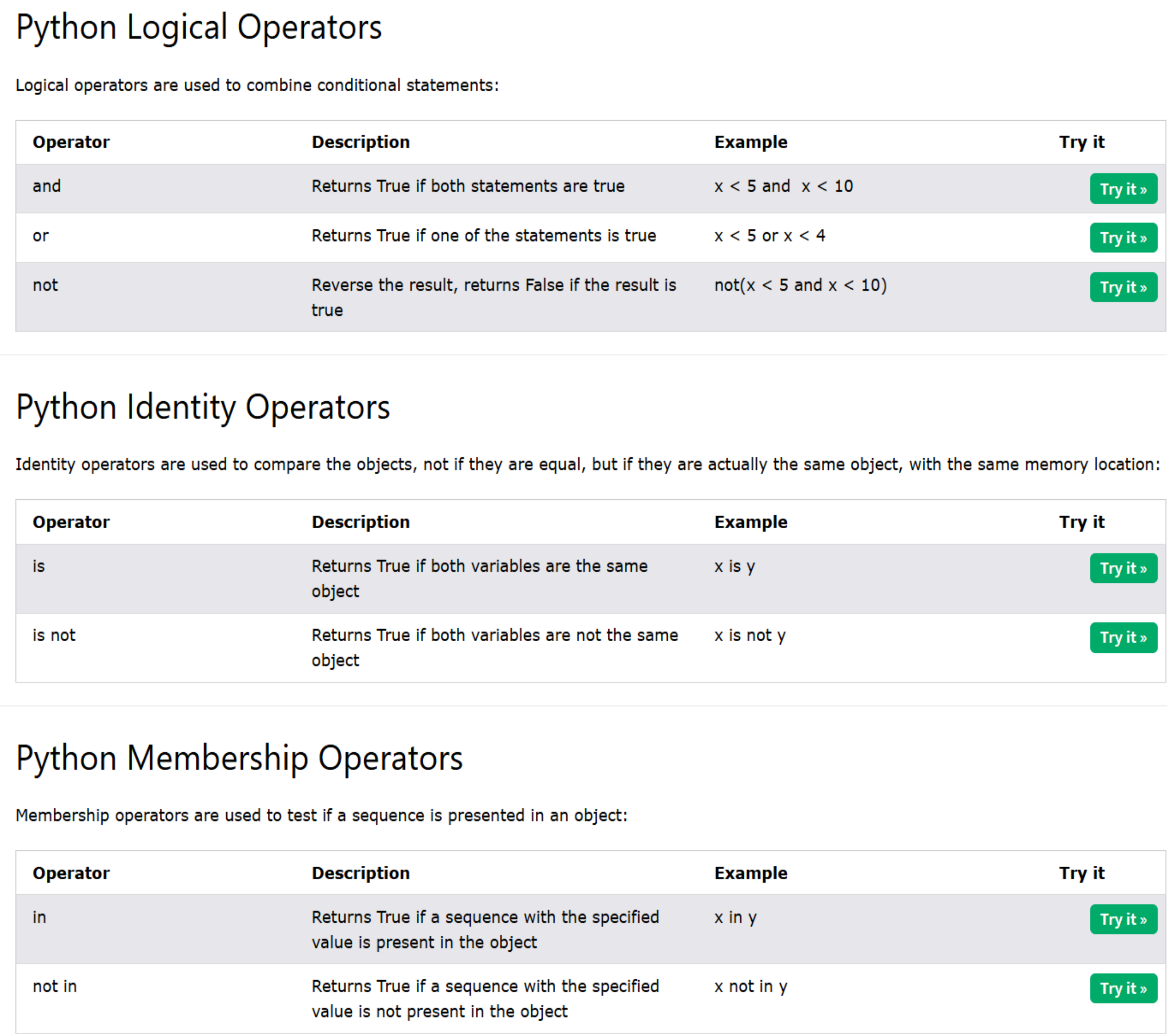
Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Python Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)



Python Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example	Try it
and	Returns True if both statements are true	x < 5 and x < 10	Try it »
or	Returns True if one of the statements is true	x < 5 or x < 4	Try it »
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)	Try it »

Python Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example	Try it
is	Returns True if both variables are the same object	x is y	Try it »
is not	Returns True if both variables are not the same object	x is not y	Try it »

Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example	Try it
in	Returns True if a sequence with the specified value is present in the object	x in y	Try it »
not in	Returns True if a sequence with the specified value is not present in the object	x not in y	Try it »

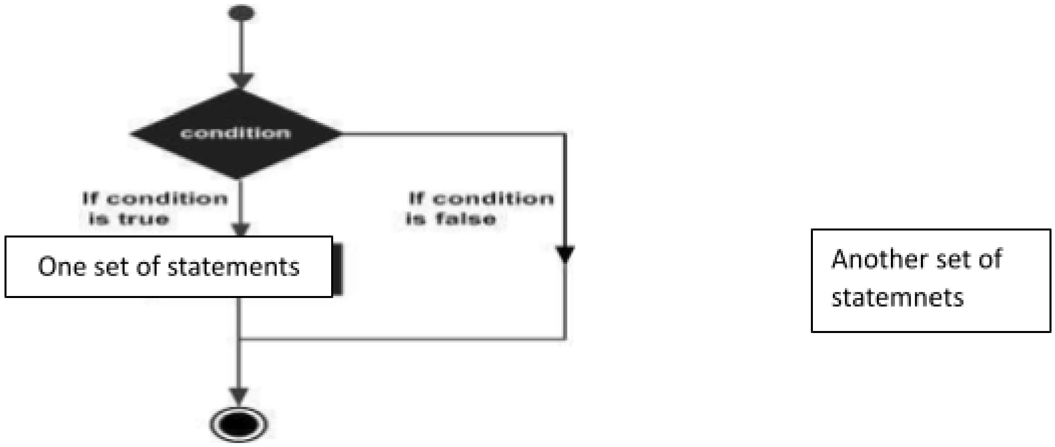
Control Statements

Control statements are classified into three type in python. They are:

1. Decision making statements
2. Looping statements
3. Jumping statements/loop control statements

1. Decision Making Statement:

- Conditional Statement in Python performs different computations or actions depending on whether a specific Boolean
- constraint evaluates to true or false. Conditional statements are handled by IF statements in Python.

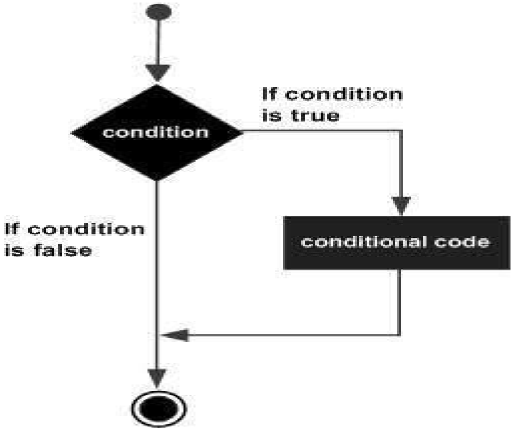


(a).if statement: In Python if statement is a statement which is used to test specified condition. The if statement executes only when specified condition is true. General Python **syntax** for a simple if statement is:

if condition :

indented Statement Block

Flow chart



Example program:

```
Age=int(input("enter your age")) if
Age>60:
   print("senior citizen")
print("entered age is",Age)
```

If the condition is true, then do the indented statements. If the condition is not true, then skip the indented statements.

(b) if else statement: The if statement accepts an expression and then executes the specified statements If the condition is true. If the condition is not true, then executes the specified statements indented below else.

The **syntax** of the if...else statement is −

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

```
 else:
     statement(s)
```

Example program:

```
num=int(input("enter a number:")) if
num%2==0:
      print("entered number is even number") else:
      print("entered number is odd number")
```

(c) if-elif statement:

The elif statement allows to check multiple expressions for TRUE then 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:
      statements(s)
```

Example:

```
marks=int(input("Enter the marks")) if
marks>=75:
     print("Grade=A") elif
marks>=60:
     print("Grade=B") elif
marks>=50:
     print("Garde=C") else:
     print("fail")
```

2. **Loops:** Loops are used to iterate over elements of a sequence, it is often used when a piece of code which you want to repeat "n" number of time. There are two types of loops in python. They are:

1. For loop
2. While loop

(a) For Loop - A for loop is used for iterating over a sequence (i.e., is either a list, a tuple or a string).With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

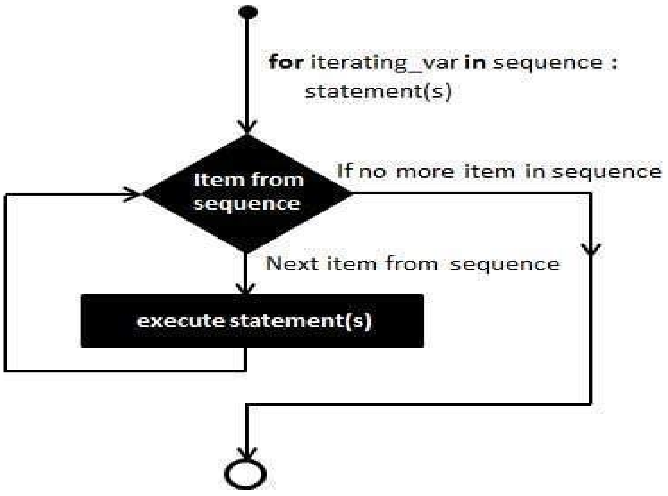
Syntax:

```
for var in sequence: statements(s)
else:
      statements(s)
```

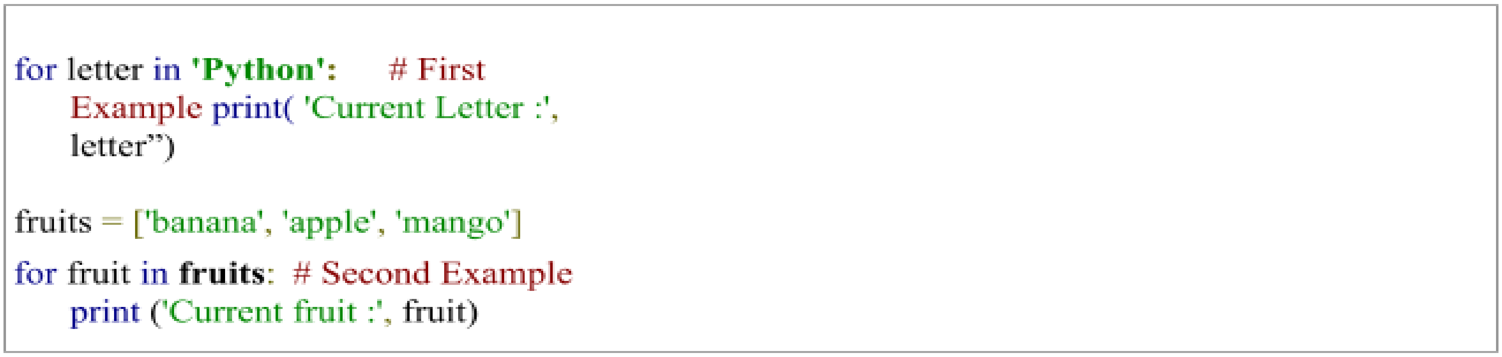
Here, var is the variable that takes the value of the item inside the sequence on each iteration. Else statement(s) will be executed when loop terminated normally(without break statement)

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.

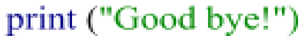
Flow Diagram



Example



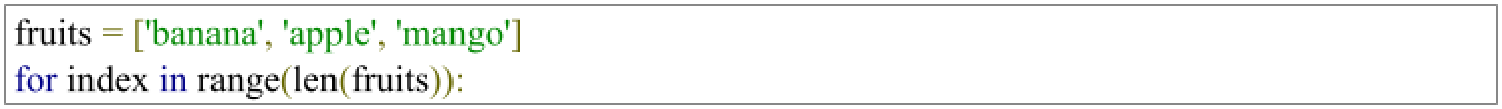
```
for letter in 'Python':     # First
     Example print( 'Current Letter :',
     letter”)

fruits = ['banana', 'apple', 'mango']
for fruit in fruits:  # Second Example
     print ('Current fruit :', fruit)
```



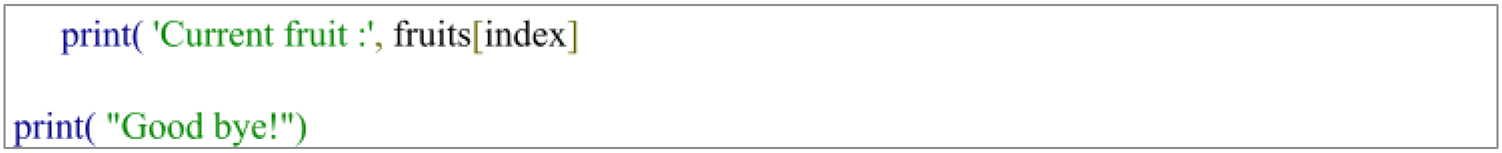
```
print ("Good bye!")
```

Iterating by Sequence Index

An alternative way of iterating through each item is by index offset into the sequence itself. Following is a simple example −



```
fruits = ['banana', 'apple', 'mango']
for index in range(len(fruits)):
```



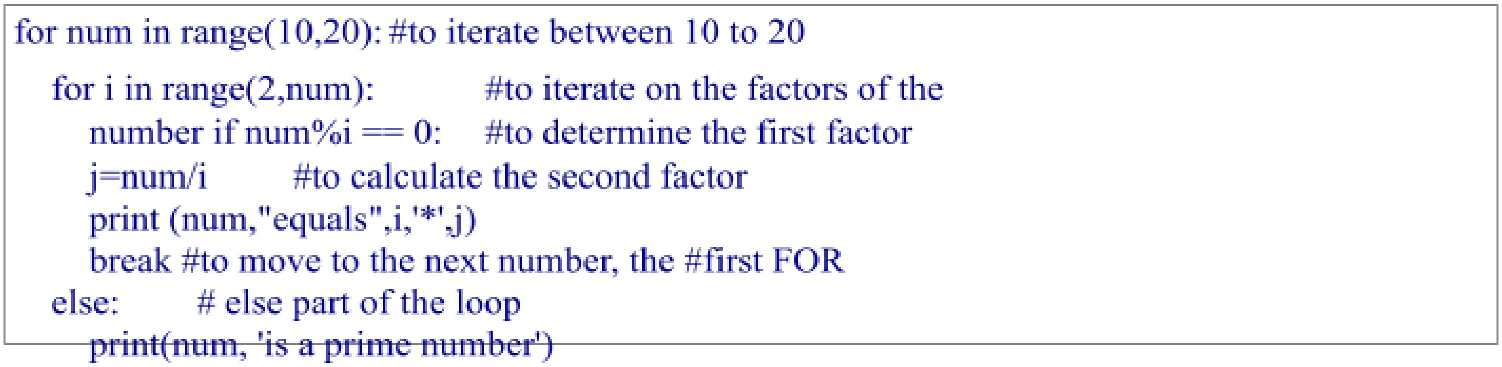
```
    print( 'Current fruit :', fruits[index]

print( "Good bye!")
```

Using else Statement with For Loop

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

- If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.

The following example illustrates the combination of an else statement with a for statement that searches for prime numbers from 10 through 20.



```
for num in range(10,20): #to iterate between 10 to 20
   for i in range(2,num):         #to iterate on the factors of the
      number if num%i == 0:    #to determine the first factor
      j=num/i         #to calculate the second factor
      print (num,"equals",i,'*',j)
      break #to move to the next number, the #first FOR
   else:       # else part of the loop
      print(num, 'is a prime number')
```

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 −

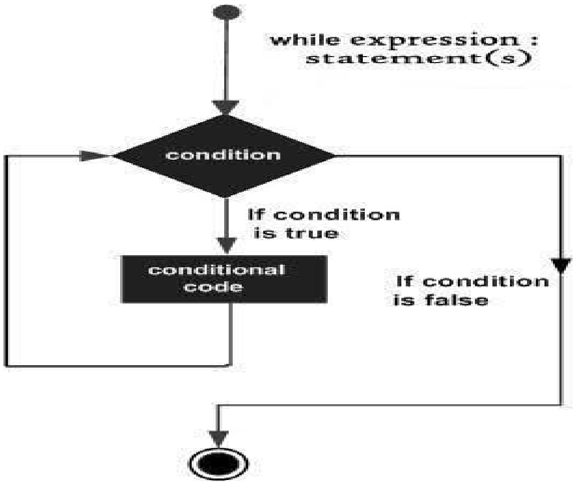
```
while expression:

    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.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

Flow Diagram



Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
count = 0

while (count < 5):

   print('The count is:', count)(
   count = count + 1

print ("Good bye!")
```

1. The Infinite Loop

A loop becomes infinite loop if a condition never becomes FALSE. You must use caution when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop.

An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.

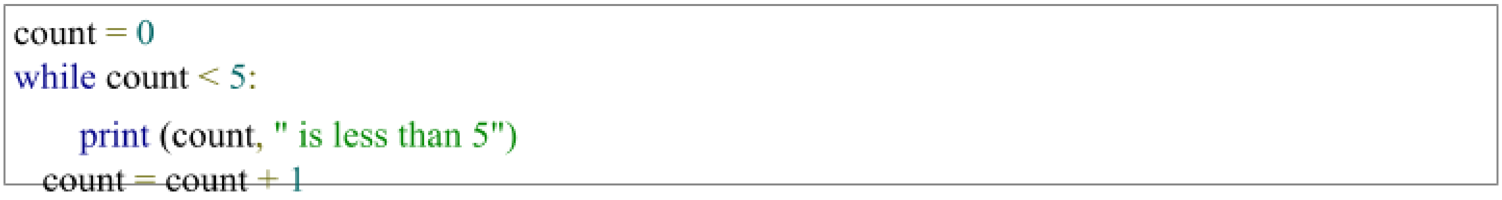
```
var = 1

while var == 1 : # This constructs an infinite loop
      num = input("Enter a number :")
      print( "You entered: ", num)

print ("Good bye!")
```

Using else Statement with While Loop:

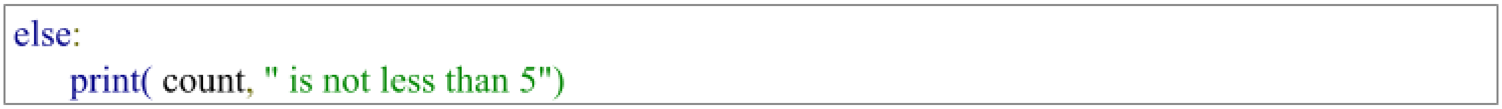
Python supports to have an **else** statement associated with a loop statement.

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

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



```
count = 0
while count < 5:
      print (count, " is less than 5")
   count = count + 1
```



```
else:
     print( count, " is not less than 5")
```

Loop control statements:

Loop control statements are:

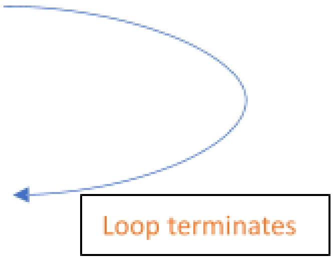
1. Break statement

2. Continue statement

Break statement: The break statement enables a program to skip over a part of the code. A break statement terminates the very loop it lies within. Execution resumes at the statement immediately following the body of the terminated statement.

The following fig explains the working of the break statement: while

```
<test-condition>:
     satement1
     if <condition>:
          break
     satement2
     satement3
satement4
satement5
```



example:

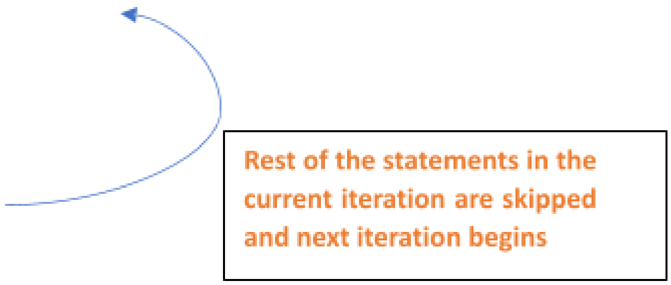
```
a=b=c=0
for i in range(1,21): a=int(input("Enter
   number1:"))
```

```
    b=int(input("Enter number2:")) if

    b==0:

        print("division by zero error!Aborting!") break

    else:

        c=a//b print("quotient=",c)

print("program error")
```

Continue statement : The continue statement is another jump statement like break as both the statements skip over a part of the code. But the continue statement is somewhat different from break. Instead of termination, the continue statement “**forces the next iteration of the loop to take place**”, skipping any code in between.

The following fig explains the working of the break statement: while

```
        <test-condition>:
            satement1
            if <condition>:
                continue
            satement2
            satement3
        satement4
        satement5
```



Rest of the statements in the current iteration are skipped and next iteration begins

Example:

```
a=b=c=0

for i in range(1,21): a=int(input("Enter

    number1:")) b=int(input("Enter

    number2:")) if b==0:

        print("division by zero error!Aborting!")

        continue

    else:

        c=a//b print("quotient=",c)

print("program error")
```

UNIT - III

Strings

String is a sequence which is made up of one or more UNICODE characters. Here the character can be a letter, digit, whitespace or any other symbol. A string can be created by enclosing one or more characters in single, double or triple quote.

Example :

```
>>> str1 = 'Hello World!'

>>> str2 = "Hello World!"

>>> str3 = """Hello World!"""

 >>> str4 = '''Hello World!''
```

str1, str2, str3, str4 are all string variables having the same value 'Hello World!'. Values stored in str3 and str4 can be extended to multiple lines using triple codes as can be seen in the following example:

```
>>> str3 = """Hello World! welcome to the world of Python"""

>>> str4 = '''Hello World! welcome to the world of Python'''
```

Accessing Characters in a String: Each individual character in a string can be accessed using a technique called indexing. The index specifies the character to be accessed in the string and is written in square brackets ([ ]).

The index of the first character (from left) in the string is 0 and the last character is n- l where n is the length of the string. If we give index value out of this range then we get an *IndexError*. The index must be an integer (positive, zero or negative).

```
#initializes a string strl
          >>> strl = 'Hello World!'

#gives the first character of strl
        >>>strl[0] 'H'

#gives seventh character of strl
        >>> strl[6]

#gives last character of strl
        >>> strl[11]

#gives error as index is out of range
        >>> strl[15]
IndexError: string index out of range
```

The index can also be an expression including variables and operators but the expression must evaluate to an integer.

```
#an expression resulting in an integer index

#so gives 6x character of strl

>>> strl[2+4]

#gives error as index must be an integer
>>> strl[1.5]
TypeError: string indices must be integers
```

Python allows an index value to be negative also. Negative indices are used when we want to access the characters of the string from right to left.

Starting from right hand side, the first character has the index as -1 and the last character has the index —n where n is the length of the string.

```
>>> strl[-1] #gives flrst character from right
>>> strl[-12]#giveslast character fromright'H'
```

positive Indices	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**		**9**	**10**	**11**
String	H	E	1	1	o		W	o		1	d	
negative Indices	-12	-11	-10	-9	-8	-7	-6	-5		-3	-2	-1

String is Immutable:

A string is an immutable data type. It means that the contents of the string cannot be changed after it has been created. An attempt to do this would lead to anerror.

```
>>> str1 = "Hello World!"
#if we try to replace character 'e' with 'a'
>>> str1[1] = 'a'
TypeError: 'str' object does not support item assignment
```

STRING OPERATIONS

As we know that string is a sequence of characters. Python allows certain operations on string data type, such as concatenation, repetition, membership and slicing. These operations are explained in the following subsections with suitable examples.

Concatenation:

To concatenate means to join. Python allows us to join two strings using concatenation operator plus which is denoted by symbol +.

```
>>> strl = 'Hello'
>>> str2 = 'World!'
>>> strl + str2 'HelloWorld!'
>>> strl 'Hello'
>>> str2 'World!'
```

Replication :

Python allows us to repeat the given string using repetition operator which is denoted by symbol *

```
#assign string 'Hello' to strl
>>> strl = 'Hello'
#repeat the value of strl 2 times
>>> strl * 2 'HelloHello' #repeat the
value of strl 5 times
>>> strl * 5 'HelloHelloHelloHelloHello'
```

Note: str1 still remains the same after the use of repetition operator.

Membership:

Python has two membership operators 'in' and 'not in'. The 'in' operator takes two strings and returns True if the first string appears as a substring in the second string, otherwise it returns False.

```
>>> str1 = 'Hello World!'
 >>> 'W' in str1 True
 >>> 'Wor' in str1 True
  >>> 'My' in str1 False
```

The 'not in' operator also takes two strings and returns True if the first string does not appear as a substring in the second string, otherwise returns False.

```
 >>> str1 = 'Hello World!'
 >>> 'My' not in str1          True
   >>> 'Hello' not in str1 False
```

Slicing:

In Python, to access some part of a string or substring, we use a method called slicing. This can be done by specifying an index range. Given a string str1, the slice operation str1[n:m] returns the part of the string str1 starting from index n (inclusive) and ending at m (exclusive). In other words, we can say that str1[n:m] returns all the characters starting from str1[n] till str1[m-1]. The numbers of characters in the substring will always be equal to difference of two indices m and n, i.e., (m-n).

```
>>> str1 = 'Hello World!'
#gives substring starting from index 1 to 4
                    >>> str1[1:5]

'ello'

#gives substring starting from 7 to 9
                    >>> str1[7:10]
'orl'
#index that is too big is truncated down to #the end
of the string
                    >>> str1[3:20]
                    'lo World!'
#first index > second index results in an #empty ''
string
 >>> str1[7:2]
```

If the first index is not mentioned, the slice starts from index. #gives
substring from index 0 to 4

```
>>> str1[:5]
'Hello'
```

If the second index is not mentioned, the slicing is done till the length of the string. #gives substring
from index 6 to end

```
                    >>> str1[6:]
'World!'
```

The slice operation can also take a third index that specifies the 'step size'. For example, str1[n:m:k], means every kth character has to be extracted from the string str1 starting from n and ending at m-1. By default, the step size is one.

```
>>> str1[0:10:2]
'HloWr'
>>> str1[0:10:3]
'HlWl'
```

Negative indexes can also be used for slicing. #characters at index
 -6,-5,-4,-3 and -2 are sliced

```
>>> str1[-6:-1]
   'World'
```

TRAVERSING A STRING:

We can access each character of a string or traverse a string using for loop and while loop.

(A) String Traversal Using for Loop:

```
 >>> str1 = 'Hello World!'
>>> for ch in str1:
        print(ch,end = '')
```

Hello World!   #output of for loop

In the above code, the loop starts from the first character of the string str1 and automatically ends when the last character is accessed.

(B) String Traversal Using while Loop:

```
>>> str1 = 'Hello World!'
>>> index = 0
#len(): a function to get length of string
```

```
>>> while index < len(str1): print(str1[index],end = '')

        index += 1

Hello World!              #output of while loop
```

Here while loop runs till the condition index < len(str) is True, where index varies from 0 to len(str1) -1.

Write a program with a user defined function to count the number of times a character (passed as argument) occurs in the given string.

```
def
     charCount
     (ch,st):
     count = 0

  for character in st:

            if character ==

                 ch: count

                 += 1

                  return count #end of function

st = input("Enter a string: ")
ch = input("Enter the character to be searched: ")
count = charCount(ch,st)
print("Number of times character",ch,"occurs in the string is:",count)
```

Program: check whether the given string is palindrome or not

```
str1=input("enter a string:")
str2=str1[::
-1]
print(str1)
print(str2)
if str1==str2:
    print("Entered string is
palindrome") else:
    print("Entered string is not a palindrome")
```

Python – Lists

The most basic data structure in Python is the **sequence**. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

Python has six built-in types of sequences, but the most common ones are lists and tuples. There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

Lists:

The list is a sequence 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.

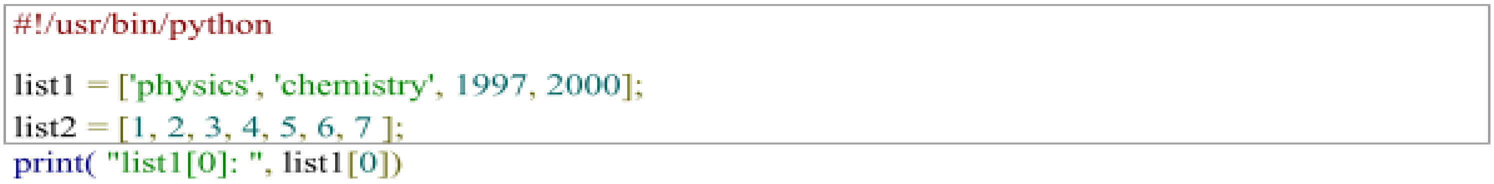
Creating a list is as simple as putting different comma-separated values between square brackets. For example −

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

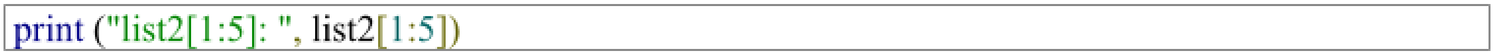
Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

Accessing Values in Lists

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −



```
#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];
print( "list1[0]: ", list1[0])
```



```
print ("list2[1:5]: ", list2[1:5])
```

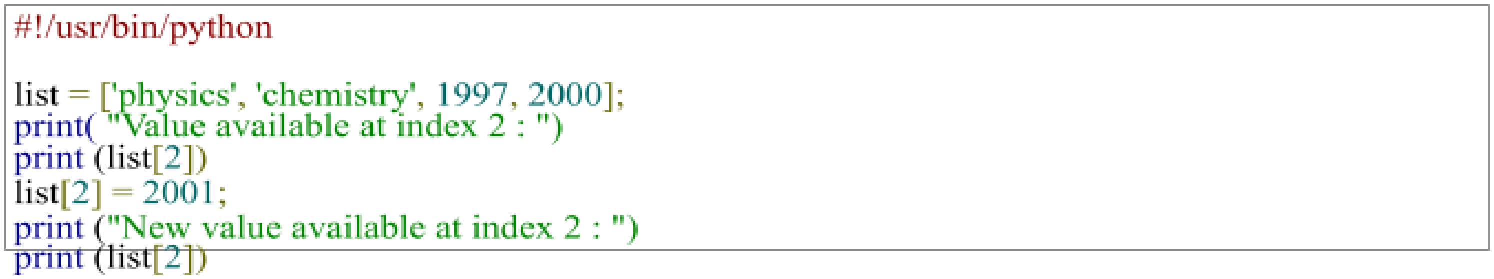
When the above code is executed, it produces the following result −

```
list1[0]: physics
list2[1:5]: [2, 3, 4, 5]
```

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.

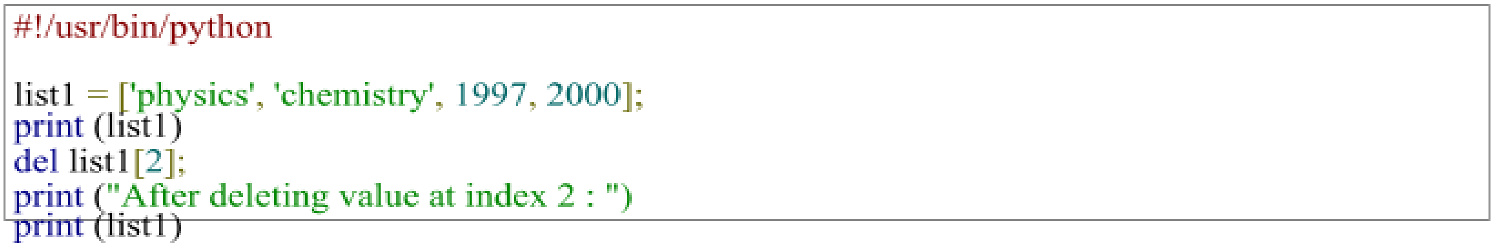
For example −



```
#!/usr/bin/python

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])
```

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 −



```
#!/usr/bin/python

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

Basic List Operations:

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

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

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

[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
[1,2] * 4	[1,2,1,2,1,2,1,2]	Repetition
3 in [1, 2, 3] 4 not in [1,2,3]	True True	Membership

for x in [1, 2, 3]:
print(x)
	1 2 3	Iteration

Indexing, Slicing, and Matrixes

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

L = ['spam', 'Spam', 'SPAM!']

Python Expression	Results	Description
L[2]	SPAM!	Offsets start at zero
L[-2]	Spam	egative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

Python - Tuples

A tuple is a collection of objects which ordered and immutable. 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.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example −

tup1 = ('physics', 'chemistry', 1997, 2000);

tup2 = (1, 2, 3, 4, 5);
tup3 = "a", "b", "c", "d";

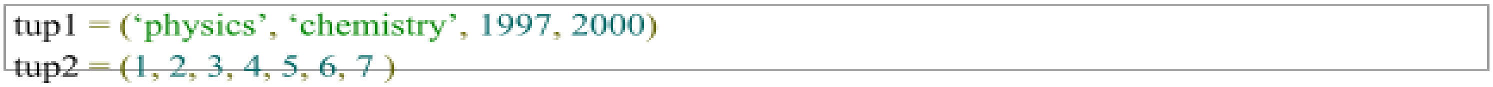
The empty tuple is written as two parentheses containing nothing − tup1 = ();

To write a tuple containing a single value you have to include a comma, even though there is only one value −

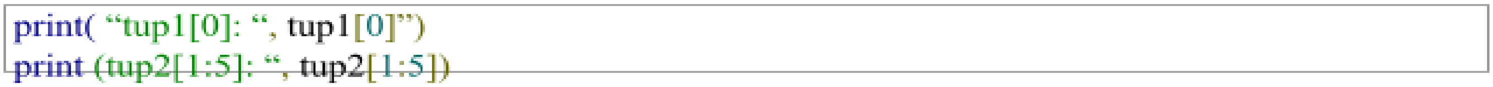
tup1 = (50,);

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

Accessing Values in Tuple: 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 –



```
tup1 = (‘physics’, ‘chemistry’, 1997, 2000)
tup2 = (1, 2, 3, 4, 5, 6, 7 )
```



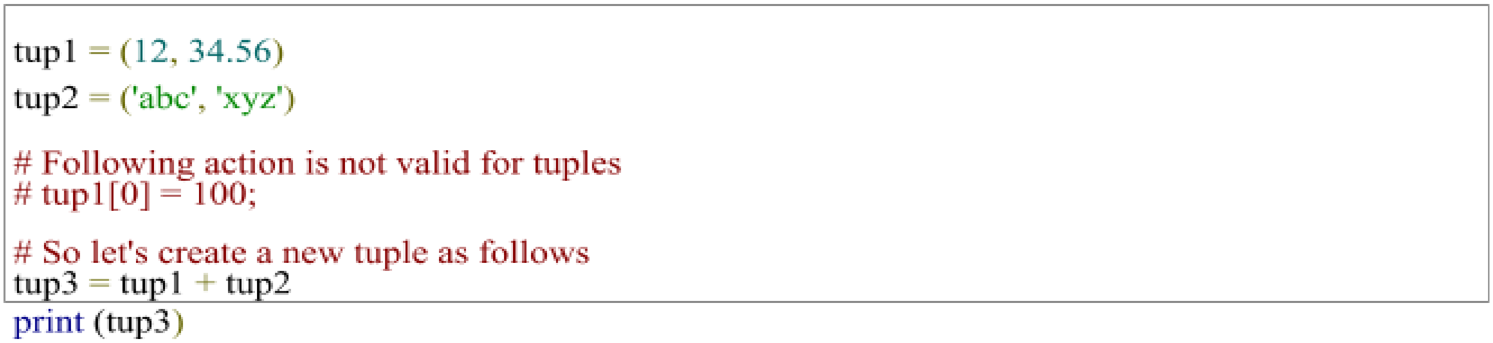
```
print( “tup1[0]: “, tup1[0]”)
print (tup2[1:5]: “, tup2[1:5])
```

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

```
tup1[0]: physics
tup2[1:5]: [2, 3, 4, 5]
```

Updating Tuples:

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates −



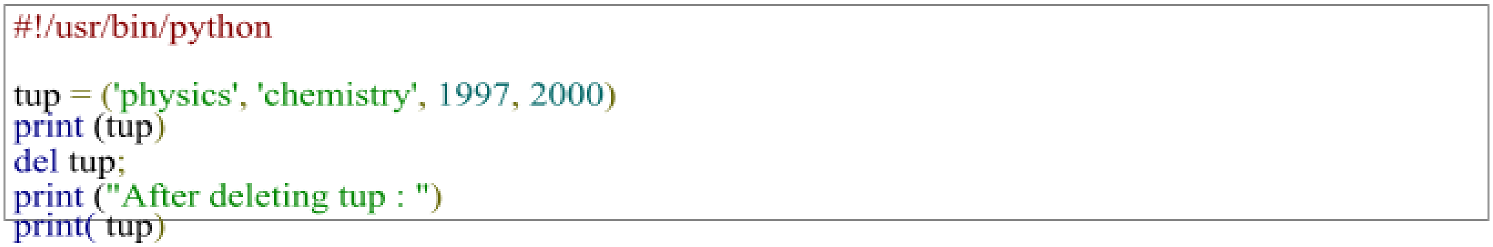
```
tup1 = (12, 34.56)
tup2 = ('abc', 'xyz')

# Following action is not valid for tuples
# tup1[0] = 100;

# So let's create a new tuple as follows
tup3 = tup1 + tup2
print (tup3)
```

When the above code is executed, it produces the following result − (12, 34.56, 'abc', 'xyz')

Delete Tuple Elements:

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the **del** statement. For example −



```
#!/usr/bin/python

tup = ('physics', 'chemistry', 1997, 2000)
print (tup)
del tup;
print ("After deleting tup : ")
print( tup)
```

Basic Tuples Operations

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

In fact, tuples respond to all of the general sequence operations we used on strings in the prior chapter −

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

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

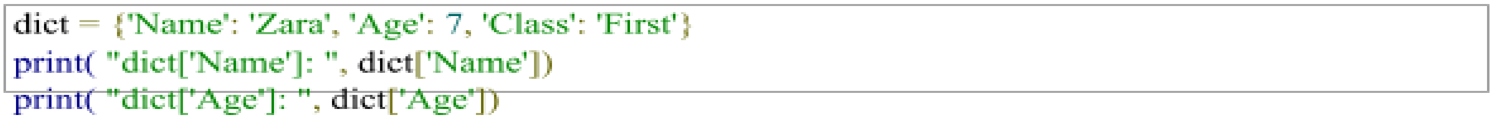
Python - Dictionary

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.

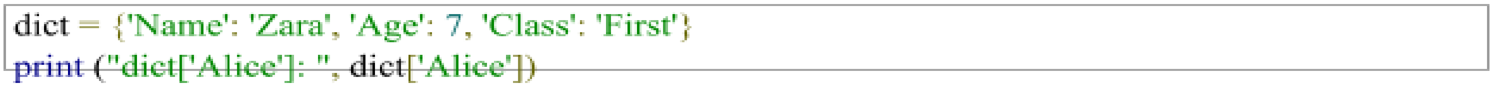
Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example −



```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print( "dict['Name']: ", dict['Name'])
print( "dict['Age']: ", dict['Age'])
```

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error as follows −

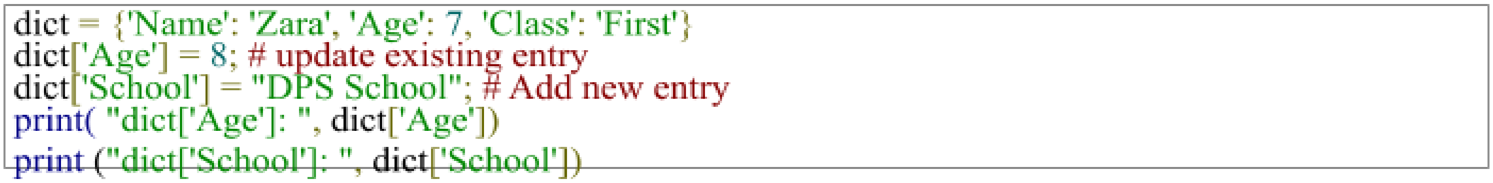


```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print ("dict['Alice']: ", dict['Alice'])
```

When the above code is executed, it produces the following result − dict['Alice']:
Traceback (most recent call last): File "test.py",
   line 4, in <module>
      Print("dict['Alice']: ", dict['Alice']) KeyError:
'Alice'

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 −

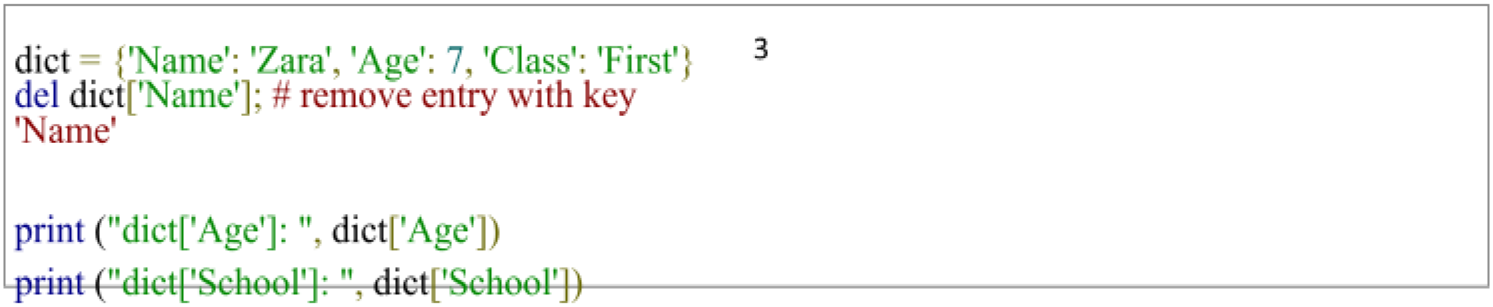
dict['Age']: 8

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'


print ("dict['Age']: ", dict['Age'])
print ("dict['School']: ", dict['School'])
```

UNIT - IV

Python - Functions

Definition : A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions.*

Defining a Function:
There are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax:

```
def functionname( parameters ): “documentation string”
        Body of the function return statement
```

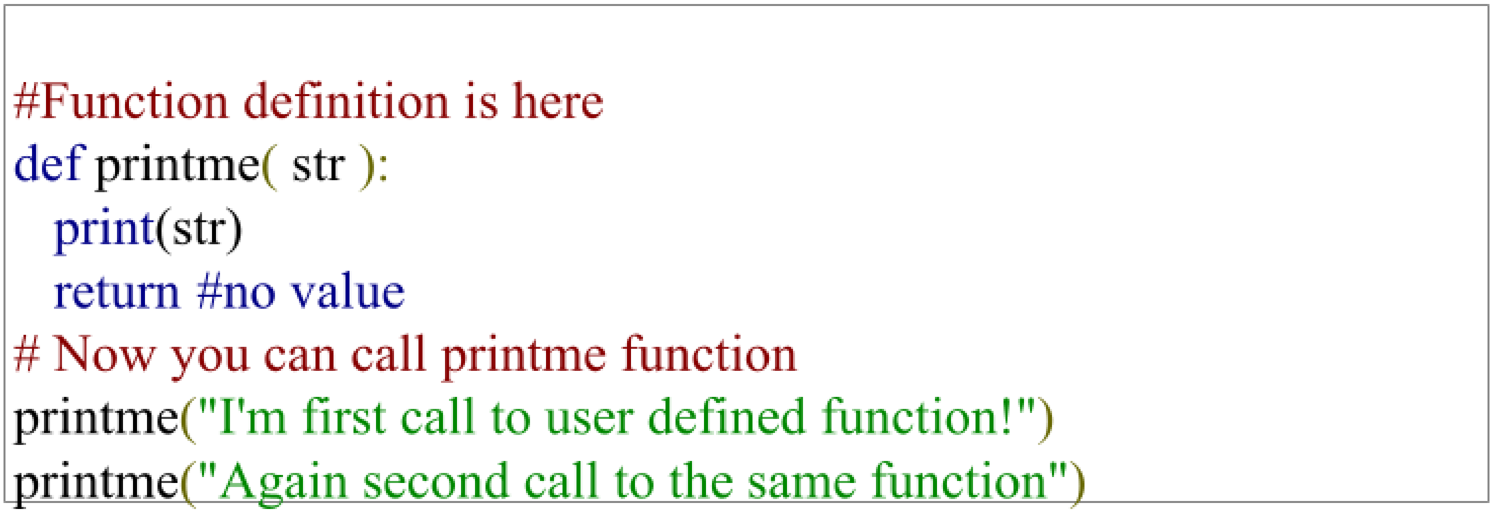
Calling a Function:

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example:
Syntax:

```
Function_name(parameters_list)
```

Example:



```
#Function definition is here
def printme( str ):
   print(str)
   return #no value
# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

Document string or Docstring :

The first string after the function is called the Document string or **Docstring** in short. This is used to describe the functionality of the function. The use of docstring in functions is optional but it is considered a good practice.

Syntax: print(function_name._doc_)

Example: Adding Docstring to the function

```
     # A simple Python function to check whether x is even o def evenOdd(x):
        """Function to check if the number is even or odd""" if (x % 2 == 0):
                print("even") else:
                print("odd")
      print(evenOdd.____________doc_____) # code to call the function
```

Output

Function to check if the number is even or odd

The return statement:

The function return statement is used to exit from a function and go back to the function caller and return the specified value or data item to the caller.

Syntax: return [expression_list]

The return statement can consist of a variable, an expression, or a constant which is returned to the end of the function execution. If none of the above is present with the return statement a None object is returned.

Example: Python Function Return Statement

```
    def square_value(num):
         """This function returns the square value of the ente mber"""
         return num**2 print(square_value(2)._____doc
    ____________________________________)

    print(square_value(4))
```

Output:

```
4

16
```

Arguments of a Function:

Arguments are the values passed inside the parenthesis of the function. A function can have any number of arguments separated by a comma.

Example: Python Function with arguments

In this example, we will create a simple function to check whether the number passed as an argument to the function is even or odd.

```
# A simple Python function to check whether x is even or def evenOdd(x):
        if (x % 2 == 0) print("even")
        else:
                print("odd") # call the
function evenOdd(2) evenOdd(3)
```

Output:

```
even odd
```

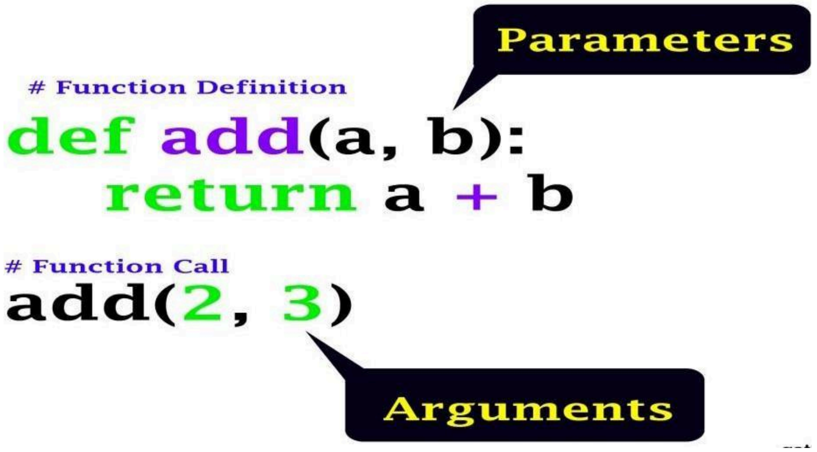
Parameters vs Arguments:

It is recommended to understand what are arguments and parameters before proceeding further. Vocabulary parameters and arguments are not limited to python but they are same across different programming languages.

- **Arguments** are values that are passed into function(or method) when the calling function
- **Parameters** are variables(identifiers) specified in the (header of) function definition

Following image shows difference between parameters and arguments.

Function Parameters VS Arguments



```
# Function Definition
def add(a, b):
    return a + b

# Function Call
add(2, 3)
```

Types of Arguments:

Python supports various types of arguments they are:

1. Default arguments
2. Keyword arguments
3. Positional arguments
4. Arbitrary positional arguments
5. Arbitrary keyword arguments

1. Default arguments:

A default argument is a parameter that assumes a default value, if a value is not provided in the function call for that argument. The following example illustrates Default arguments.

```
# Python program to demonstrate default arguments def myFun(x, y=50):
        print("x: ", x)

        print("y: ", y)

 myFun(10)              # We call myFun() with only one argument
```

2. Keyword arguments:

The idea is to allow the caller to specify the argument name with values so that caller does not need to remember the order of parameters.

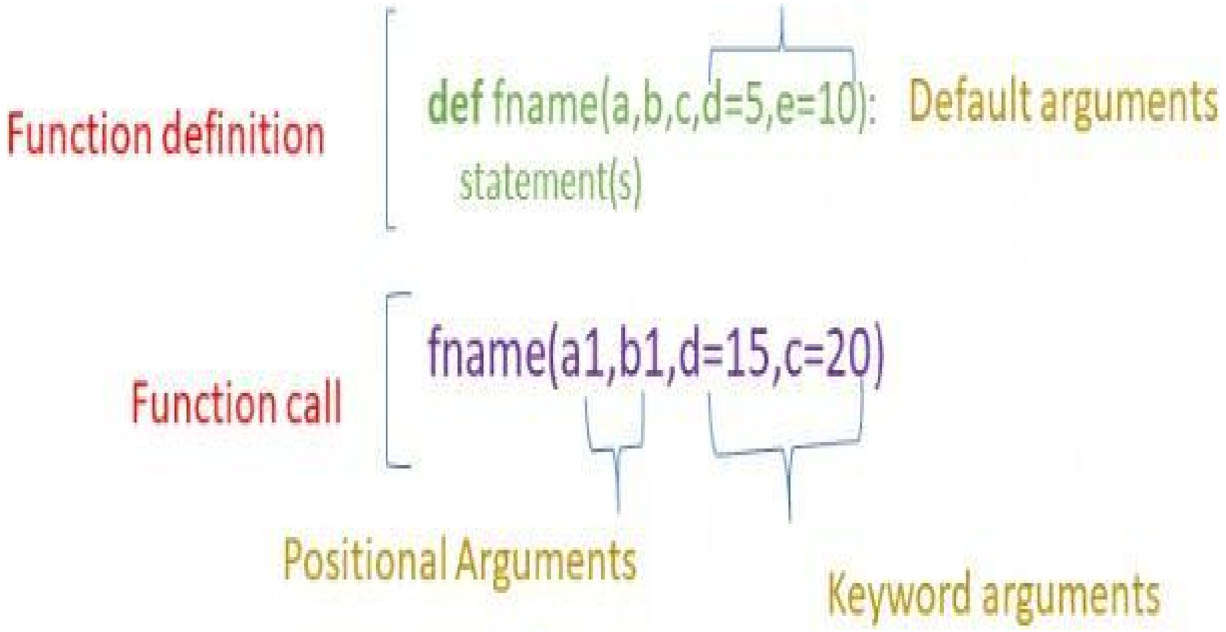
```
# Python program to demonstrate Keyword Arguments def student(firstname, lastname):
     print(firstname, lastname)
student(firstname='Geeks', lastname='Practice') # Keywo guments
student(lastname='Practice', firstname='Geeks') # Keywo guments
```

3. Positional arguments:

During a function call, values passed through arguments should be in the order of parameters in the function definition. This is called **positional arguments.**

Keyword arguments should follow positional arguments only.

default vs positional vs keyword arguments:



4. **<u>Variable-length arguments:</u>**

In Python, we can pass a variable number of arguments to a function using special symbols. There are two special symbols:

- *args (Non-Keyword Arguments)/Arbitrary positional arguments
- **kwargs (Keyword Arguments)/ Arbitrary keyword arguments

Example 1: Variable length non-keywords arguments

```
def myFun(*argv):                    # *args for variable number of arguments
     for arg in argv: print(arg)
  myFun('Hello', 'Welcome', 'to', 'pyhton')
```

Output

```
Hello Welcome to
python
```

Example 2: Variable length keyword arguments

```
def myFun(**kwargs):                    # *kargs for variable number of keyword arguments
       for key, value in kwargs.items(): print( (key, value))
myFun(first='python', mid='program', last='functions')
```

Functions are first class objects :

Python’s functions are first-class objects. You can assign them to variables, store them in data structures, pass them as arguments to other functions, and even return them as values from other functions.
Properties of first class functions:

- A function is an instance of the Object type.

- You can store the function in a variable.
- You can pass the function as a parameter to another function.
- You can return the function from a function.
- You can store them in data structures such as hash tables, lists, …

Examples illustrating First Class functions in Python:

1. **Functions are objects:** Python functions are first class objects. we are assigning function to a variable. This assignment doesn't call the function. It takes the function object referenced by fun1 and creates a second name pointing to it, fun2.

```
# Python program to illustrate functions can be treated as jects
def fun1(text): return text.upper()
print(fun1('Hello')) fun2 = fun1
print(fun2('Hello'))
```

Output:

```
HELLO HELLO
```

2. **Functions can be passed as arguments to other functions:**

Because functions are objects we can pass them as arguments to other functions. Functions that can accept other functions as arguments are also called higher-order functions. In the example below, we have created a function **greet** which takes a function as an argument.

```
# Python program to illustrate functions can be passed as guments to other functions
def simple(text):

    return text.upper() def word(text):
    return text.lower()

def greet(func):
    # storing the function in a variable

    greeting = func("""Hi, I am created by a function passe argument.""")
    print (greeting)

greet(simple) greet(word)
```

3. **Functions can return another function:**

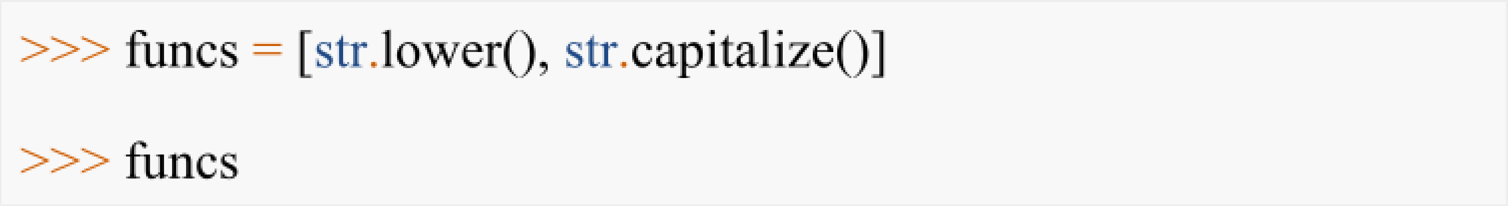
Functions are objects we can return a function from another function. In the below example, the create_adder function returns adder function.

```
# Functions can return another function def create_adder(x):
        def adder(y): return x+y
        return adder
```

```
add_15 = create_adder(15)

print (add_15(10))
```

4. Functions Can Be Stored In Data Structures:

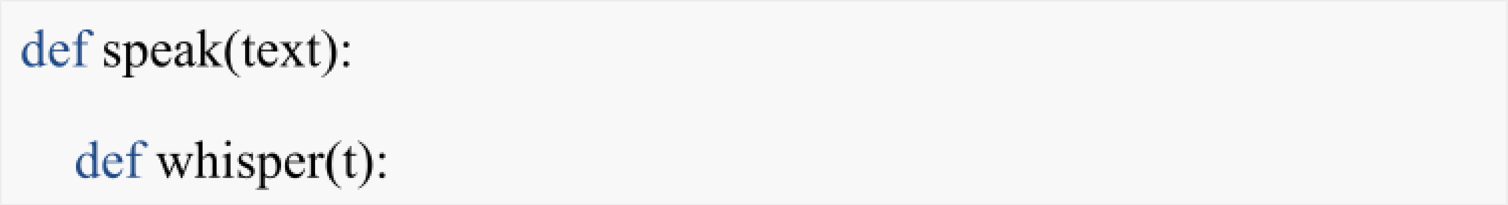
As functions are first-class citizens you can store them in data structures, just like you can with other objects. For example, you can add functions to a list:



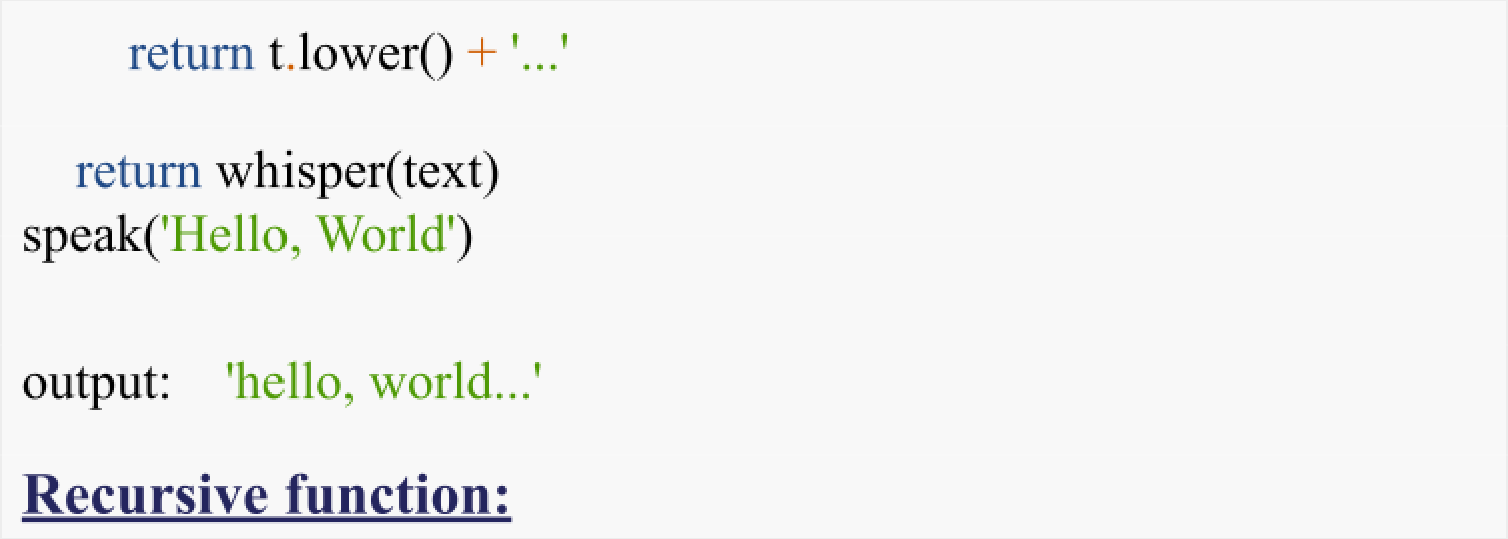
```
>>> funcs = [str.lower(), str.capitalize()]

>>> funcs
```

5. Functions Can Be Nested:

Python allows functions to be defined inside other functions.
These are often called *nested functions* or *inner functions*. Here's an example:



```
def speak(text):

    def whisper(t):
```



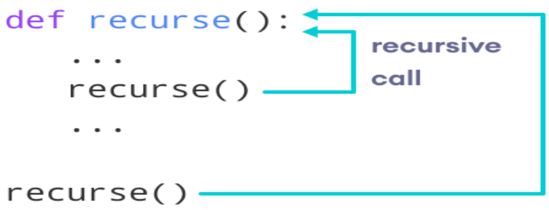
```
        return t.lower() + '...'

    return whisper(text)
speak('Hello, World')


output:    'hello, world...'
```

Recursive function:

Recursion is a programming method, in which a function calls itself one or more times in its body. Usually, it is returning the return value of this function call. If a function definition follows recursion, we call this function a recursive function.

A recursive function has to terminate to be used in a program. It terminates, if with every recursive call the solution of the problem is becomes smaller and moves towards a base case, where the problem can be solved without further recursion. A recursion can lead to an infinite loop, if the base case is not met in the calls.

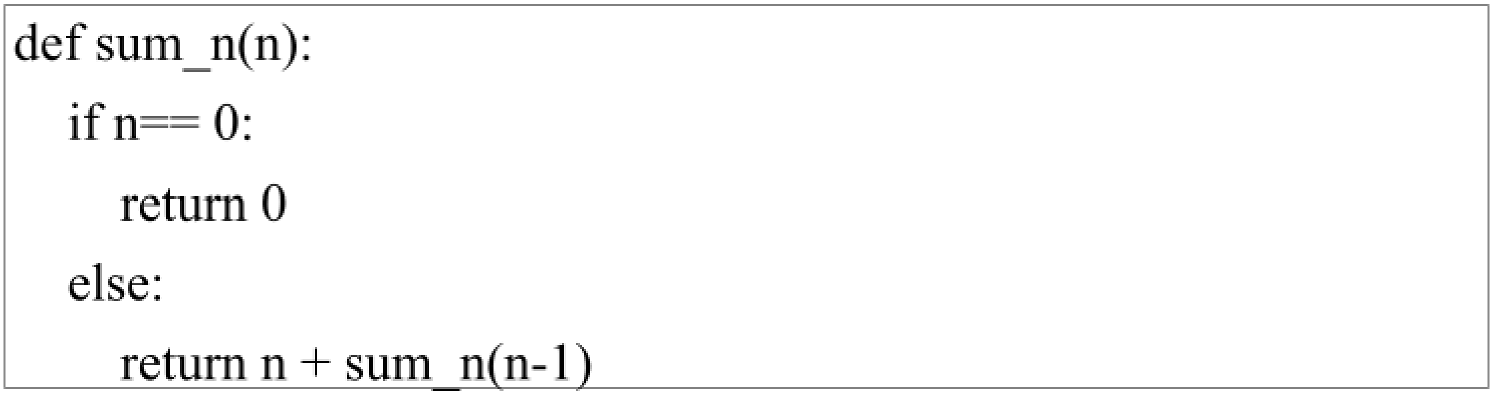
The following image shows the working of a recursive function called recurse.



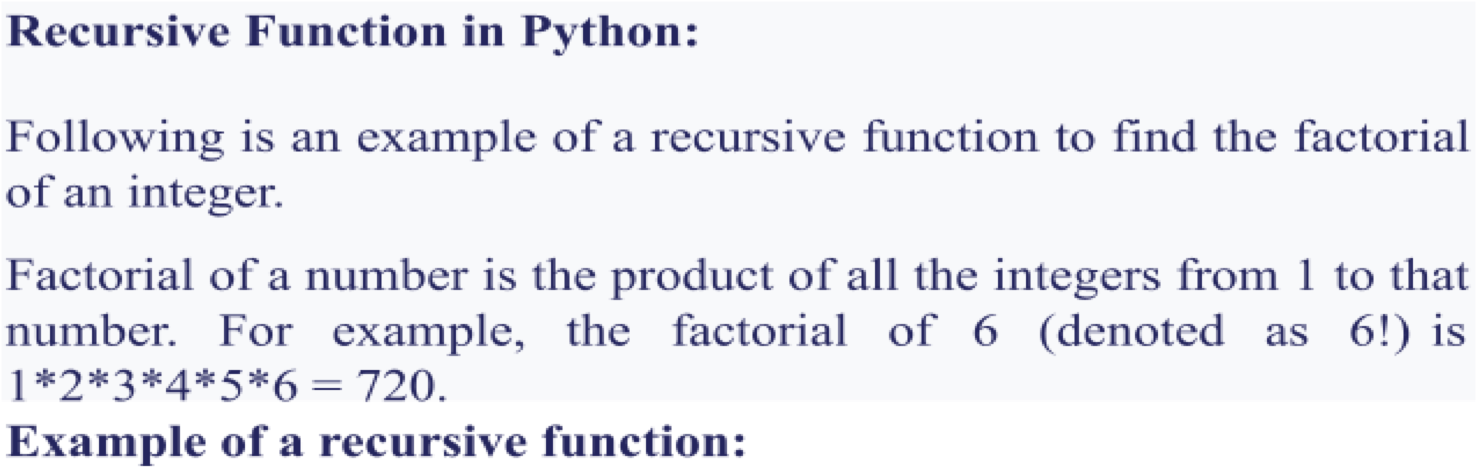
```
def recurse():
    ...
    recurse()
    ...

recurse()
```

Example

The following code returns the sum of first n natural numbers using a recursive python function.



```
def sum_n(n):
   if n== 0:
      return 0
   else:
      return n + sum_n(n-1)
```



Recursive Function in Python:

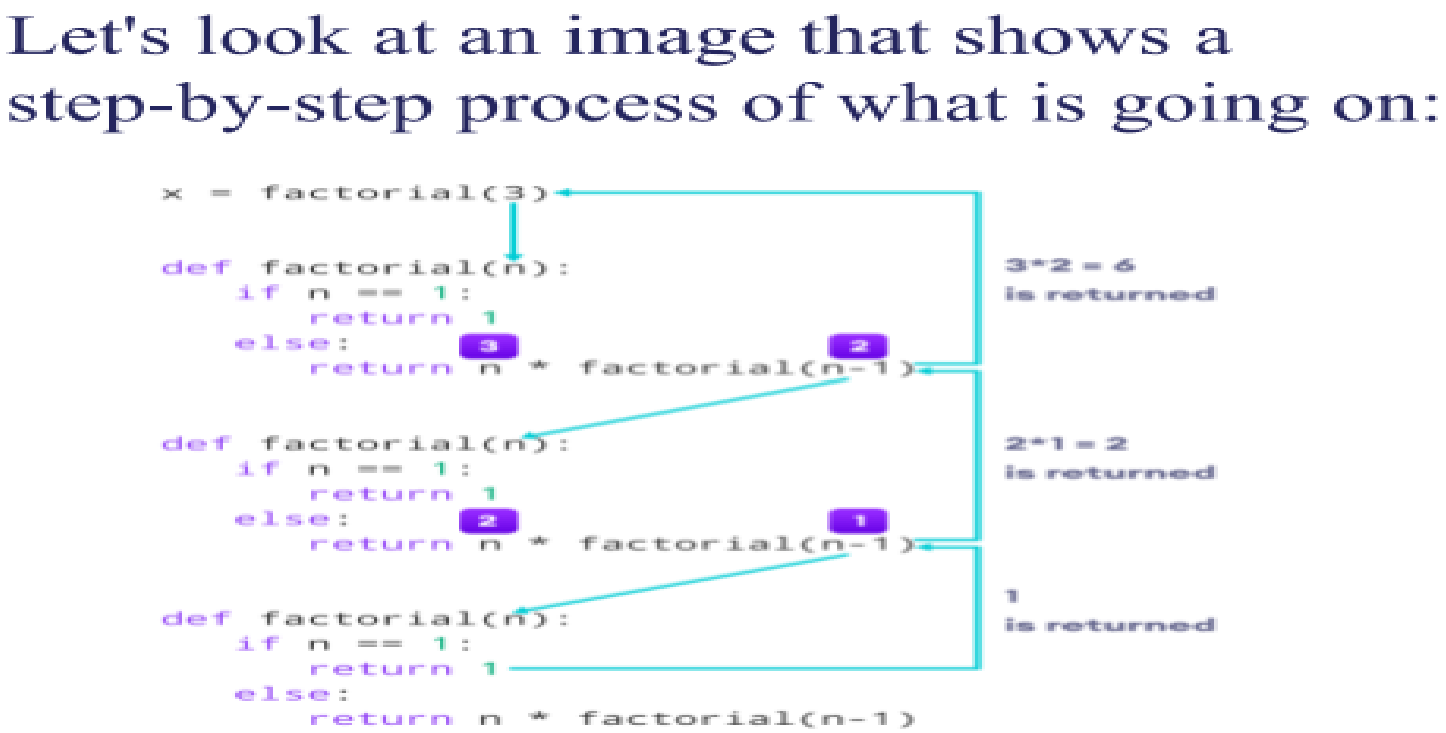
Following is an example of a recursive function to find the factorial of an integer.

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.

Example of a recursive function:

```
def tri_recursion(k):
      if(k > 0):
        result = k + tri_recursion(k - 1)
        print(result)
      else:
        result = 0
      return result

     print("Recursion Example Results:")
     tri_recursion(6)
```

- Our recursion ends when the number reduces to 1. This is called the base condition.
- Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.
- The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.



Let's look at an image that shows a step-by-step process of what is going on:

```
x = factorial(3)

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

UNIT - V

Classes and Objects in Python

1. Class Method and self-argument:

- **Class Method:** A method bound to the class and not the instance of the class. It receives the class itself as the first argument, conventionally named cls. Class methods are declared using the @classmethod decorator.

  Create a Class

  To create a class, use the keyword class:

  Example: Create a class named MyClass, with a property named x:

  ```
  class MyClass:
    x = 5
  ```

- **self-argument:** The first parameter of an instance method, representing the instance of the class on which the method is called. It allows access to the instance's attributes and other methods. The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class. It does not have to be named self, you can call it whatever you like, but it has to be the first parameter of any function in the class:

  Example :

  ```
  class Person:
    def __init__(myobject, name, age):
  ```

```
    myobject.name = name
    myobject.age = age
  def myfunc(abc):
    print("Hello my name is " + abc.name)
p1 = Person("John", 36)
p1.myfunc()
```

2. Class Variables and Object Variables:

- **Class Variables (Static Variables):** Variables defined within the class but outside any methods. They are shared by all instances of the class. Changes to a class variable affect all instances.
- **Object Variables (Instance Variables):** Variables defined within instance methods (typically in the __init__ constructor) using self.variable_name. Each instance has its own copy of object variables.

  Example: Create an object named p1, and print the value of x:

  ```
  p1 = MyClass()
  print(p1.x)
  ```

3. Public and Private Data Members:

- **Public Data Members:** Variables in a class that can be accessed and modified directly from outside the class. In Python, all members are public by default.
- **Private Data Members:** Variables intended for internal use within the class, not directly accessible from outside. In Python, privacy is conventionally indicated by prefixing the variable name with a double underscore (e.g., __private_var). This triggers name mangling, making direct access harder but not impossible.

4. Private Methods:

- Methods intended for internal use within the class, not meant to be called directly from outside. Similar to private data members, they are conventionally indicated by prefixing with a double underscore (e.g., __private_method()).

5. Built-in Class Attributes:

- Special attributes automatically provided by Python for every class.

  Examples include:

  - __doc__: The class's documentation string.
  - __name__: The class's name.
  - __module__: The module in which the class is defined.
  - __dict__: A dictionary containing the class's namespace.

6. Static Methods:

- Methods defined within a class but do not operate on the instance or the class itself. They do not receive self or cls as their first argument. Static methods are declared using the @staticmethod decorator and are essentially regular functions logically grouped within a class. They are useful for utility functions that relate to the class but don't require access to instance or class state.