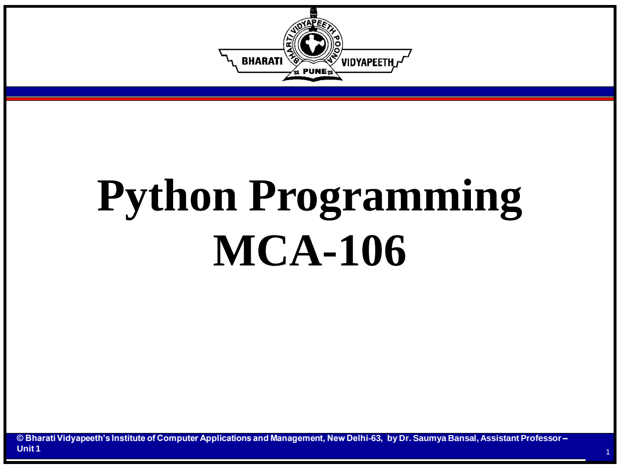
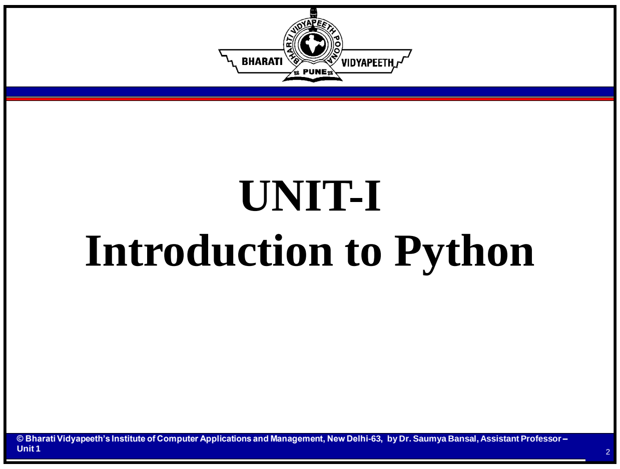
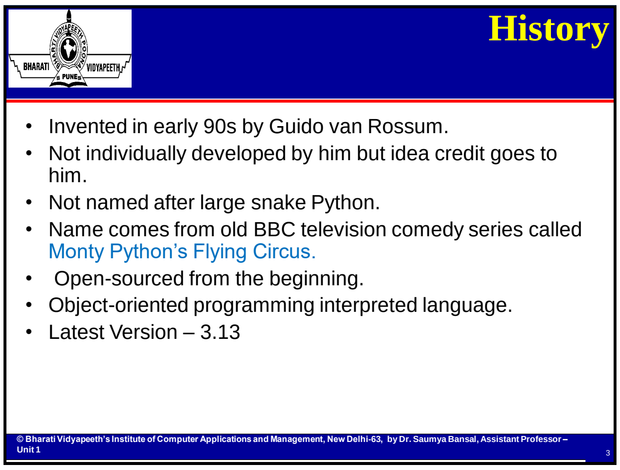


# Python Programming
# MCA-106

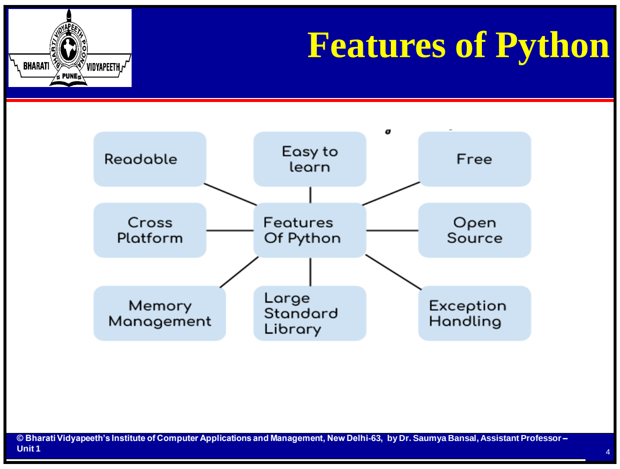


# UNIT-I
# Introduction to Python

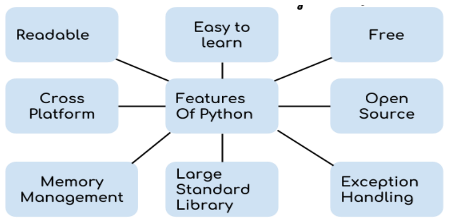


## History

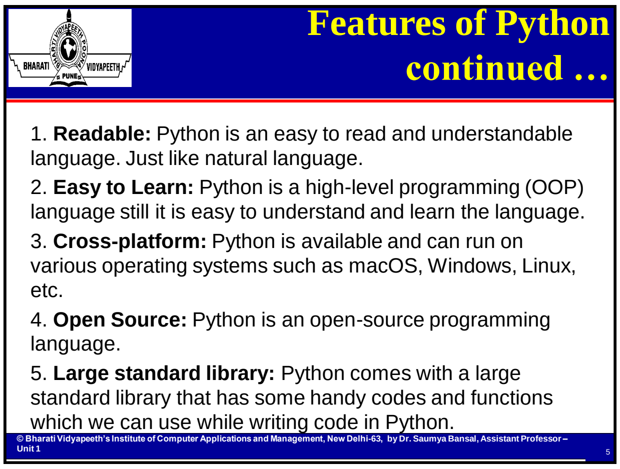
- Invented in early 90s by Guido van Rossum.
- Not individually developed by him but idea credit goes to him.
- Not named after large snake Python.
- Name comes from old BBC television comedy series called Monty Python's Flying Circus.
- Open-sourced from the beginning.
- Object-oriented programming interpreted language.
- Latest Version – 3.13



## Features of Python



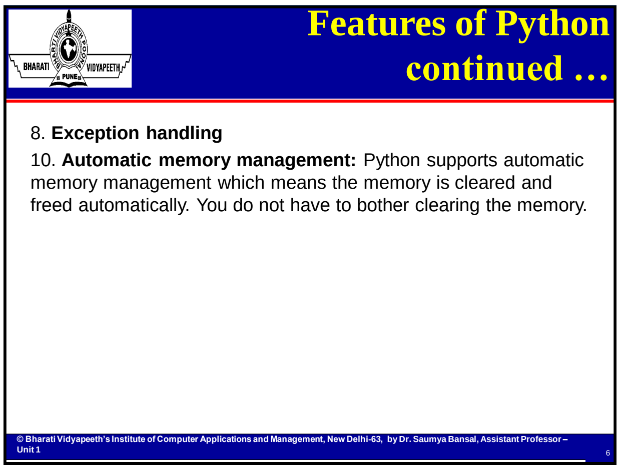
4



## Features of Python continued …

1. **Readable:** Python is an easy to read and understandable language. Just like natural language.
2. **Easy to Learn:** Python is a high-level programming (OOP) language still it is easy to understand and learn the language.
3. **Cross-platform:** Python is available and can run on various operating systems such as macOS, Windows, Linux, etc.
4. **Open Source:** Python is an open-source programming language.
5. **Large standard library:** Python comes with a large standard library that has some handy codes and functions which we can use while writing code in Python.

5

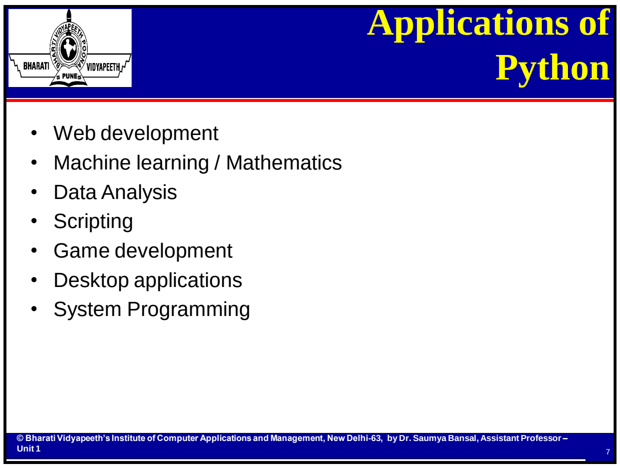


## Features of Python continued …

8. **Exception handling**

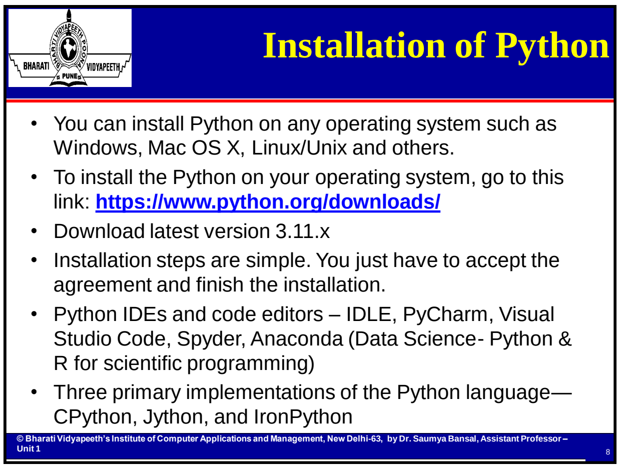
10. **Automatic memory management:** Python supports automatic memory management which means the memory is cleared and freed automatically. You do not have to bother clearing the memory.

6



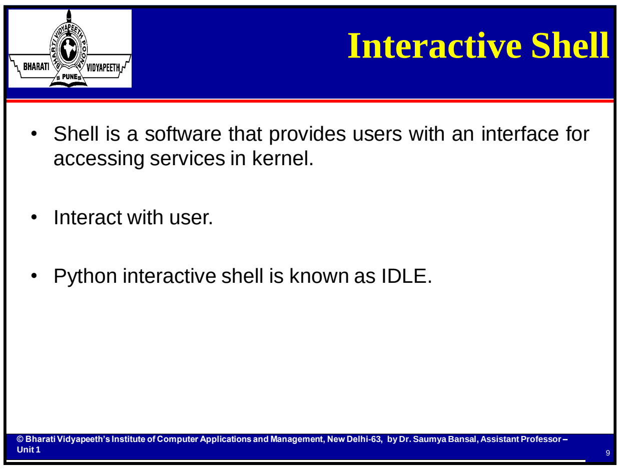
## Applications of Python

- Web development
- Machine learning / Mathematics
- Data Analysis
- Scripting
- Game development
- Desktop applications
- System Programming



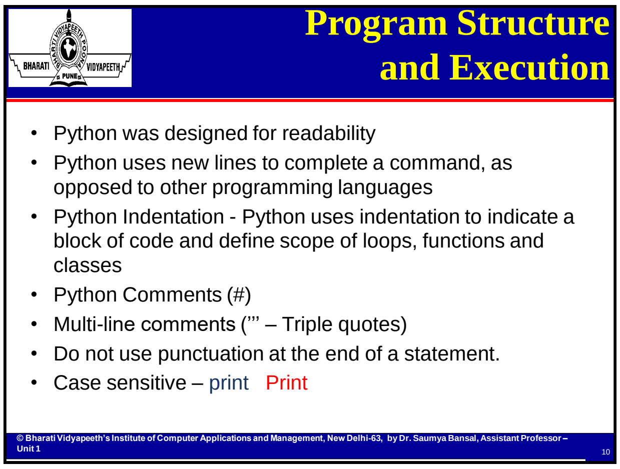
## Installation of Python

- You can install Python on any operating system such as Windows, Mac OS X, Linux/Unix and others.
- To install the Python on your operating system, go to this link: **https://www.python.org/downloads/**
- Download latest version 3.11.x
- Installation steps are simple. You just have to accept the agreement and finish the installation.
- Python IDEs and code editors – IDLE, PyCharm, Visual Studio Code, Spyder, Anaconda (Data Science- Python & R for scientific programming)
- Three primary implementations of the Python language—CPython, Jython, and IronPython



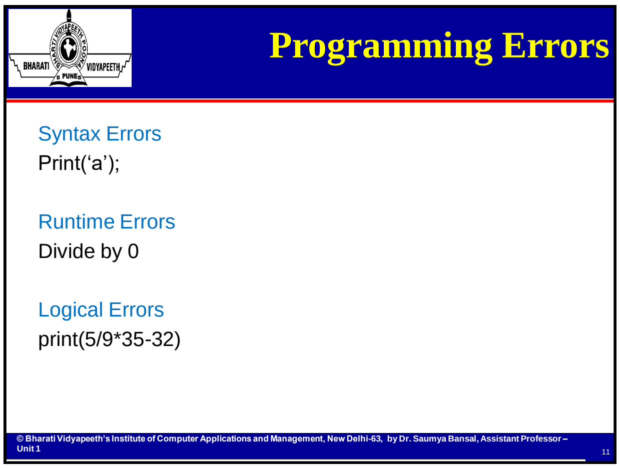
## Interactive Shell

- Shell is a software that provides users with an interface for accessing services in kernel.
- Interact with user.
- Python interactive shell is known as IDLE.



## Program Structure and Execution

- Python was designed for readability
- Python uses new lines to complete a command, as opposed to other programming languages
- Python Indentation - Python uses indentation to indicate a block of code and define scope of loops, functions and classes
- Python Comments (#)
- Multi-line comments (''' – Triple quotes)
- Do not use punctuation at the end of a statement.
- Case sensitive – print   Print

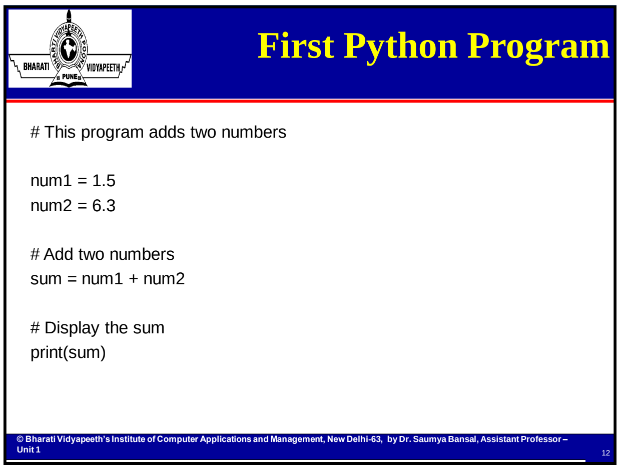


## Programming Errors

Syntax Errors
Print('a');

Runtime Errors
Divide by 0

Logical Errors
print(5/9*35-32)



## First Python Program

```
# This program adds two numbers

num1 = 1.5
num2 = 6.3

# Add two numbers
sum = num1 + num2

# Display the sum
print(sum)
```

## Identifiers

- **Identifier** is a user-defined name given to a variable, function, class, module, etc.
- The identifier is a combination of character digits and an underscore.
- They are case-sensitive.
- ‘num’ and ‘Num’ and ‘NUM’ are three different identifiers in python.

**Rules for Naming Python Identifiers**

- It cannot be a reserved python keyword.
- It should not contain white space.
- It can be a combination of A-Z, a-z, 0-9, or underscore.
- It should start with an alphabet character or an underscore ( _ ).
- It should not contain any special character other than an underscore ( _ ).

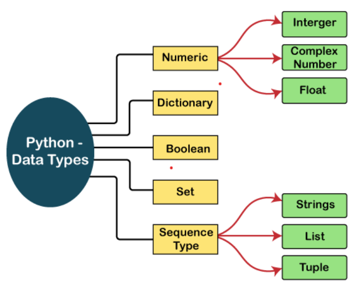
## Keywords

- Predefined and reserved words in Python that have special meanings
- The identifier is a combination of character digits and an underscore.
- The keyword cannot be used as an identifier, function, or variable name.
- All the keywords in Python are written in lowercase except True and False.
- There are 35 keywords in Python 3.11.

## Escape Sequences

- An escape sequence is a sequence of characters with special meaning when used inside a string or a character.
- **Syntax:** The characters need to be preceded by a backslash character
- The characters that we can't insert into a string are called **Illegal characters,** and these characters modify the string.
- The function of escape sequences is to insert such characters into the string without modifying the string.
- **Example:** \n, \t, \’, \” etc.

## Data Types



## Python Variable

Variables are containers for storing data values.

- Python has no command for declaring a variable.
- A variable is created the moment you first assign a value to it.
- Variables do not need to be declared with any particular *type*, and can even change type after they have been set.

```
x = 4         # x is of type int
x = "Arjun" # x is now of type str
print(x)
```

- If you want to specify the data type of a variable, this can be done with casting.

```
x = str(3)    # x will be '3'
y = int(3)    # y will be 3
z = float(3)  # z will be 3.0
```

- You can get the data type of a variable with the type() function.

## Python Variable continued …

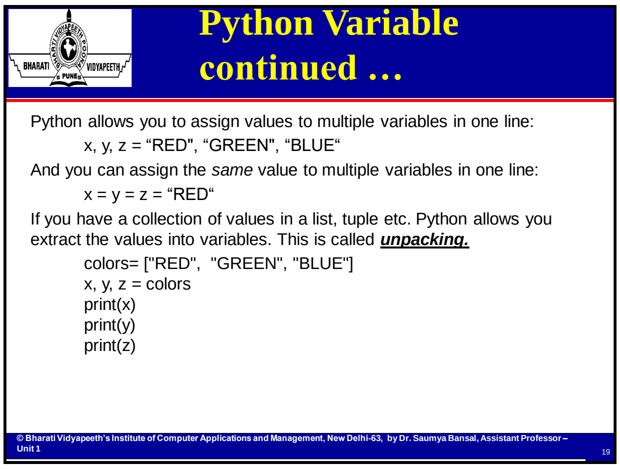
String variables can be declared either by using single or double quotes

```
x = "Arjun"
# is the same as
x = 'Arjun'
```

**Variable names are case-sensitive.**

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). 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)



## Python Variable continued …

Python allows you to assign values to multiple variables in one line:

```
x, y, z = “RED", “GREEN", “BLUE“
```

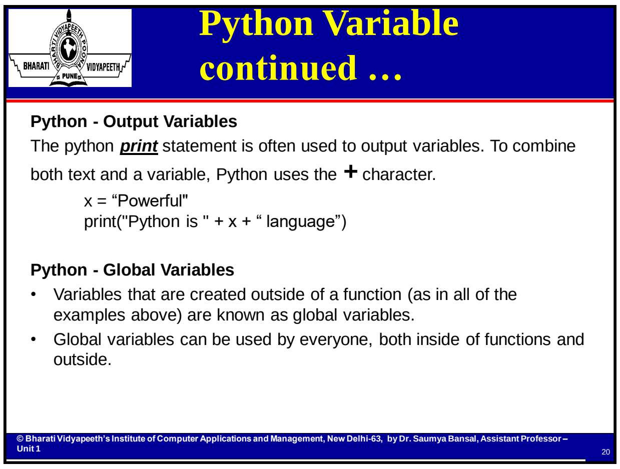
And you can assign the *same* value to multiple variables in one line:

```
x = y = z = “RED“
```

If you have a collection of values in a list, tuple etc. Python allows you extract the values into variables. This is called ***unpacking.***

```
colors= ["RED",  "GREEN", "BLUE"]
x, y, z = colors
print(x)
print(y)
print(z)
```

19



## Python Variable continued …

**Python - Output Variables**

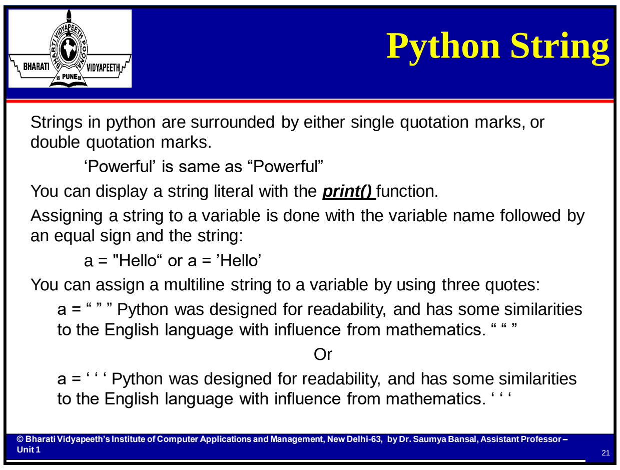
The python ***print*** statement is often used to output variables. To combine both text and a variable, Python uses the **+** character.

```
x = “Powerful"
print("Python is " + x + “ language”)
```

**Python - Global Variables**

- Variables that are created outside of a function (as in all of the examples above) are known as global variables.
- Global variables can be used by everyone, both inside of functions and outside.

20



## Python String

Strings in python are surrounded by either single quotation marks, or double quotation marks.

‘Powerful’ is same as “Powerful”

You can display a string literal with the ***print()*** function.

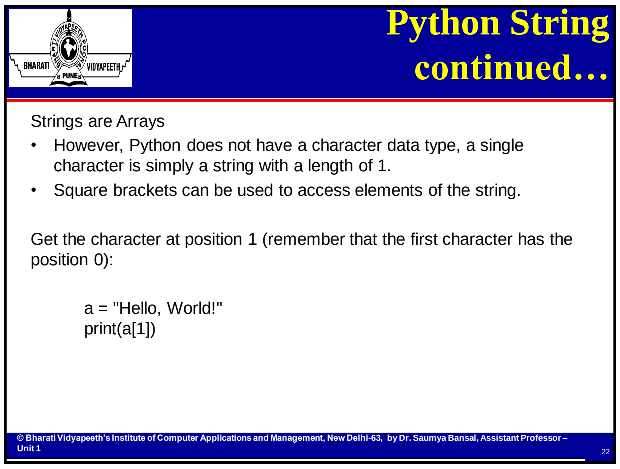
Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

```
a = "Hello“ or a = ’Hello’
```

You can assign a multiline string to a variable by using three quotes:

```
a = “ ” ” Python was designed for readability, and has some similarities
to the English language with influence from mathematics. “ “ ”
                                   Or
a = ‘ ‘ ‘ Python was designed for readability, and has some similarities
to the English language with influence from mathematics. ‘ ‘ ‘
```

21



## Python String continued…

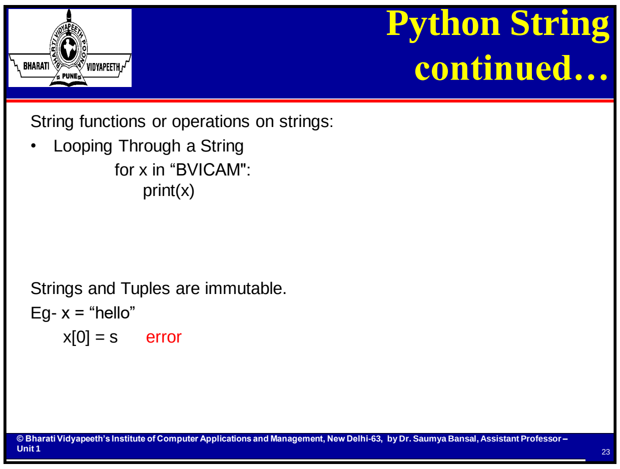
Strings are Arrays

- However, Python does not have a character data type, a single character is simply a string with a length of 1.
- Square brackets can be used to access elements of the string.

Get the character at position 1 (remember that the first character has the position 0):

```
a = "Hello, World!"
print(a[1])
```

22



## Python String continued…

String functions or operations on strings:

- Looping Through a String

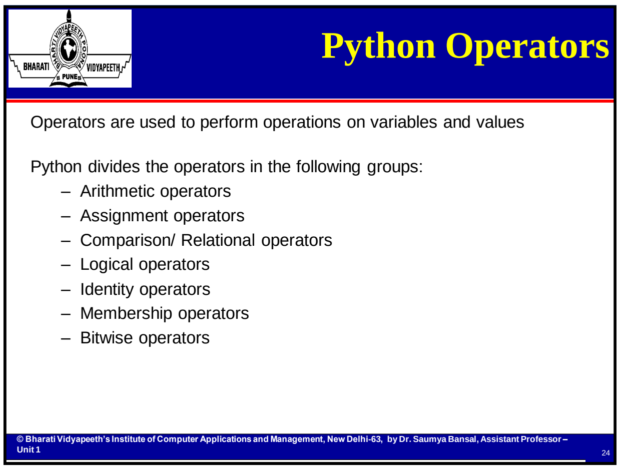
```
for x in "BVICAM":
    print(x)
```

Strings and Tuples are immutable.

Eg- x = “hello”

x[0] = s      error

23



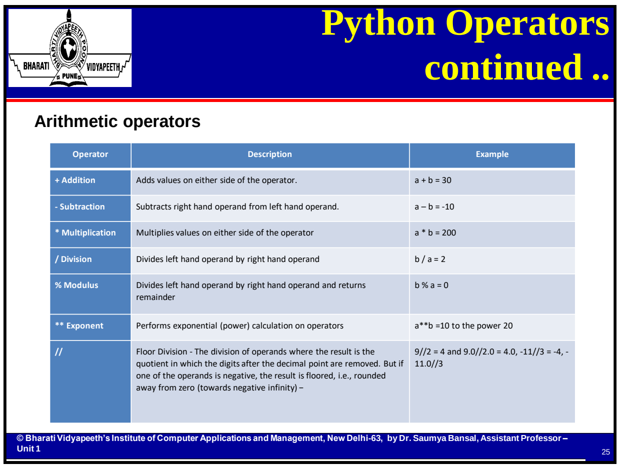
## Python Operators

Operators are used to perform operations on variables and values

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison/ Relational operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

24

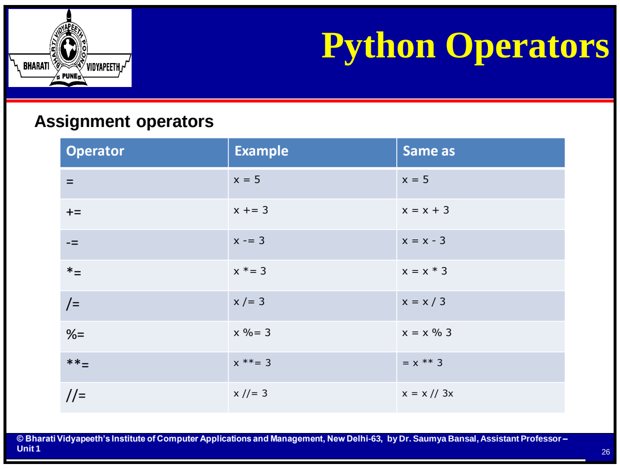


## Python Operators continued ..

### 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. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) – | 9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 |

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Saumya Bansal, Assistant Professor – Unit 1 25

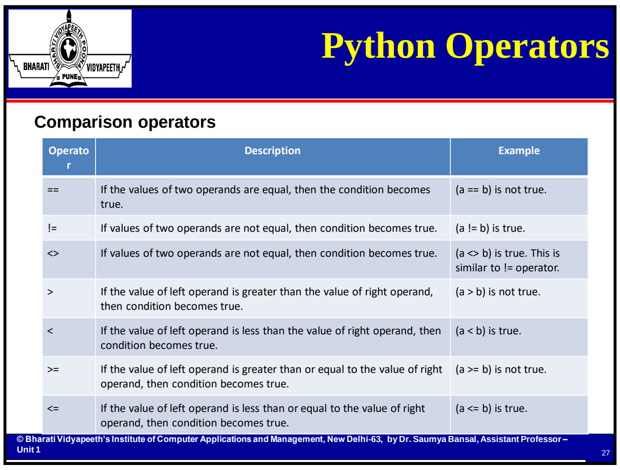


## Python Operators

### Assignment operators

| 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 ** 3 |
| //= | x //= 3 | x = x // 3x |

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Saumya Bansal, Assistant Professor – Unit 1 26

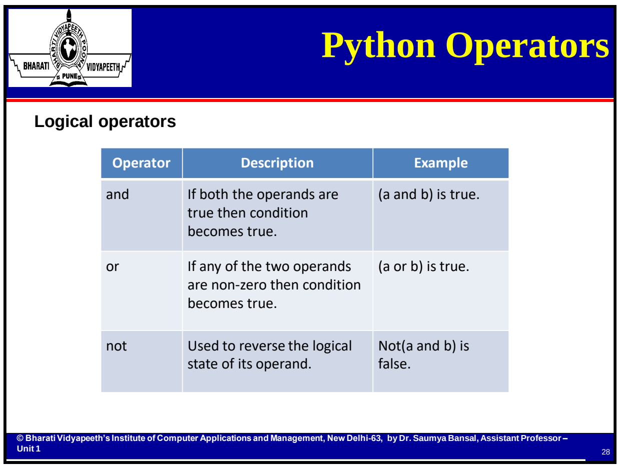


## Python Operators

### Comparison operators

| Operato r | 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. |

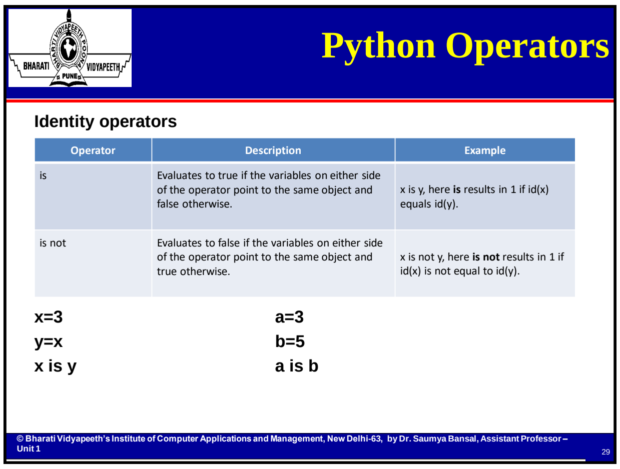
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Saumya Bansal, Assistant Professor – Unit 1 27



## Python Operators

**Logical operators**

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

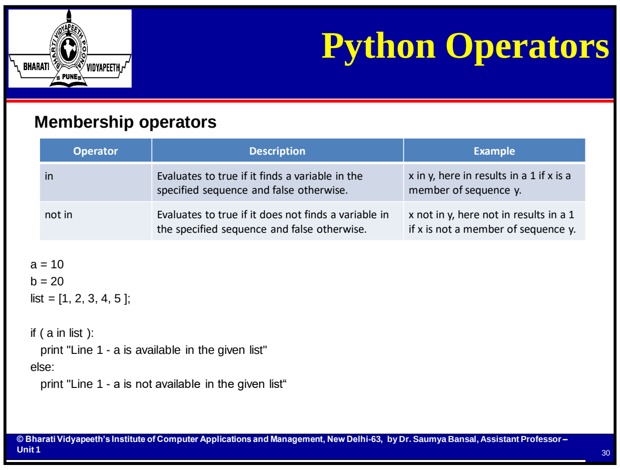


## Python Operators

**Identity operators**

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

```
x=3        a=3
y=x        b=5
x is y     a is b
```



## Python Operators

**Membership operators**

| 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. |

```
a = 10
b = 20
list = [1, 2, 3, 4, 5 ];

if ( a in list ):
   print "Line 1 - a is available in the given list"
else:
   print "Line 1 - a is not available in the given list"
```

## Python Operators

**Bitwise operators –** Consider a = 0011 1100 and b = 0000 1101

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

## Operators Precedence

| S.No. | Operator & Description |
|---|---|
| 1 | ** Exponentiation (raise to the power) |
| 2 | ~ + - Complement, unary plus and minus (method names for the last two are +@ and -@) |
| 3 | * / % // Multiply, divide, modulo and floor division |
| 4 | + - Addition and subtraction |
| 5 | >> << Right and left bitwise shift |
| 6 | & Bitwise 'AND' |
| 7 | ^ \| Bitwise exclusive `OR' and regular `OR' |
| 8 | <= < > >= Comparison operators |
| 9 | <> == != Equality operators |
| 10 | = %= /= //= -= += *= **= Assignment operators |
| 11 | **is or is not** Identity operators |
| 12 | **in not in** Membership operators |
| 13 | **not or and** Logical operators |

## Type conversion in Python

Python defines type conversion functions to directly convert one data type to another which is useful in day to day and competitive programming.

There are two types of Type Conversion in Python:

- ***Implicit Type Conversion***

  In Implicit type conversion of data types in Python, the Python interpreter automatically converts one data type to another without any user involvement.

```
x = 10
print("x is of type:",type(x))
y = 10.6
print("y is of type:",type(y))
x = x + y
print(x)
print("x is of type:",type(x))
```

## Type conversion in Python

- ***Explicit Type Conversion***

  In Explicit Type Conversion in Python, the data type is manually changed by the user as per their requirement.

```
#convert from int to float:
x = float(1)
#convert from float to int:
y = int(2.8)
#convert from int to complex:
z = complex(x)
print(x)
print(y)
print(z)

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

## Short Circuit Evaluation in Python

Short-circuit evaluation means that when evaluating Boolean expression like AND and OR, you can stop as soon as you find the first condition that satisfies or negates the expression.

| Operation | Result | Description |
|---|---|---|
| x or y | If x is false, then y else x | Only evaluates the second argument(y) if the firs one is false |
| x and y | If x is false, then x else y | Only evaluates the second argument(y) if the first one(x) is True |
| | | |

## Lazy Evaluation in Python

Lazy evaluation is an evaluation strategy which holds the evaluation of an expression until its value is needed. It avoids repeated evaluation.

**Lazy Evaluation – Advantages**

- It allows the language runtime to discard sub-expressions that are not directly linked to the final result of the expression.
- It reduces the time complexity of an algorithm by discarding the temporary computations and conditionals.
- It allows the programmer to access components of data structures out-of-order after initializing them, as long as they are free from any circular dependencies.
- It is best suited for loading data which will be infrequently accessed.

## Lazy Evaluation in Python

**Lazy Evaluation − Drawbacks**

- It forces the language runtime to hold the evaluation of sub-expressions until it is required in the final result by creating **thunks** (delayed objects).
- Sometimes it increases space complexity of an algorithm.
- It is very difficult to find its performance because it contains thunks of expressions before their execution.

**Lazy Evaluation in Python**

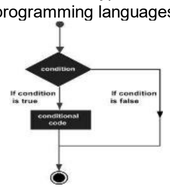
- The **range** method in Python follows the concept of Lazy Evaluation. It returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

```
r = range(start, stop, step)
r= range(6)
r= range(3,6)
r= range(3,20,2)
```

## Conditional Statements in Python

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.

- Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.
- Following is the general form of a typical decision making structure found in most of the programming languages −



## Conditional Statements in Python

- 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.
- Python programming language provides following types of decision making statements. Click the following links to check their detail.

| Sr.No. | Statement & Description |
|---|---|
| 1 | if statements An **if statement** consists of a boolean expression followed by one or more statements. |
| 2 | if...else statements An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE. |
| 3 | nested if statements You can use one **if** or **else if** statement inside another **if** or **else if** statement(s). |

## Conditional Statements in Python

**Python Conditions and If statements**

Python supports the usual logical conditions from mathematics:

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
elif a== b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

## Conditional Statements in Python

**Short Hand If**

If you have only one statement to execute, you can put it on the same line as the if statement : Example

```
if a > b: print("a is greater than b")
```

**Short Hand If ... Else**

- If you have only one statement to execute, one for if, and one for else, you can put it all on the same line: Example

```
a = 2
b = 330
print("A") if a > b else print("B")
```

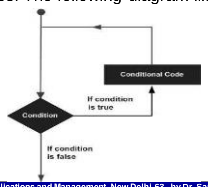
**The pass Statement**

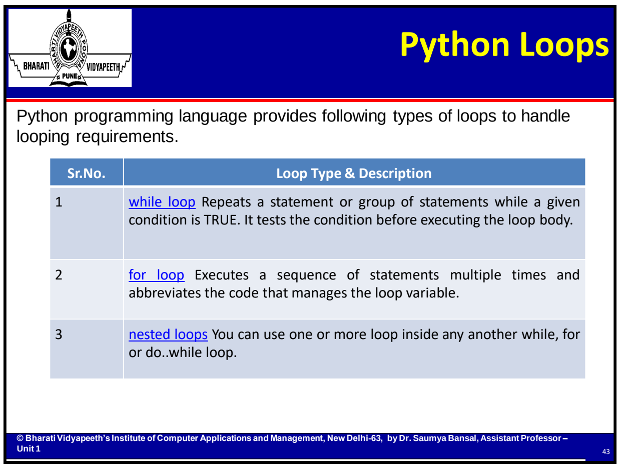
If statements cannot be empty, but if you for some reason have an if statement with no content, put in the statement to avoid getting an error.

```
if b > a:
  pass
```

## Python Loops

- In general, statements are executed sequentially: But, there may be a situation when you need to execute a block of code several number of times.
- Programming languages provide various control structures that allow for more complicated execution paths.
- A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −

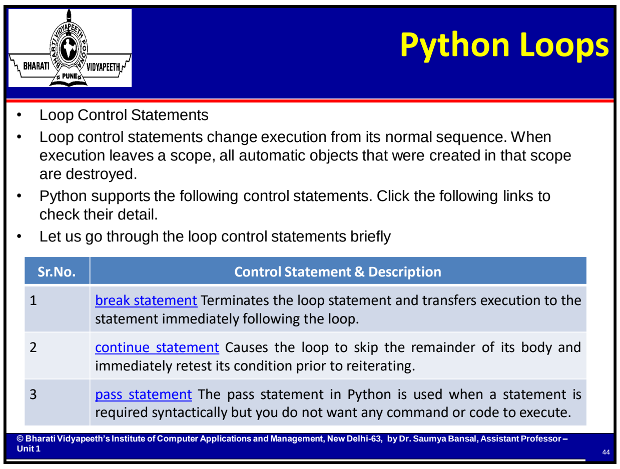




## Python Loops

Python programming language provides following types of loops to handle looping requirements.

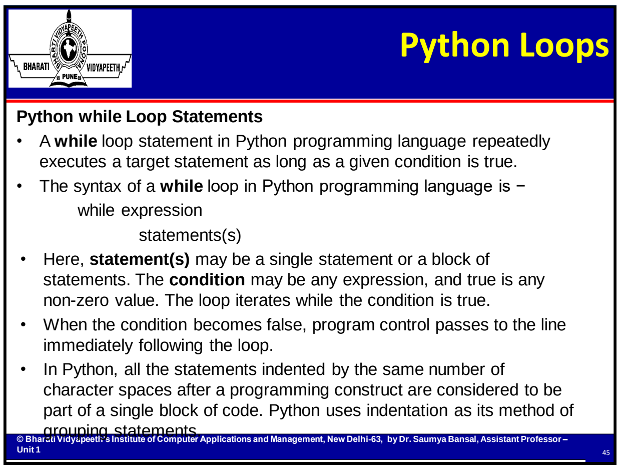
| Sr.No. | Loop Type & Description |
|---|---|
| 1 | while loop Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body. |
| 2 | for loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| 3 | nested loops You can use one or more loop inside any another while, for or do..while loop. |



## Python Loops

- Loop Control Statements
- Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
- Python supports the following control statements. Click the following links to check their detail.
- Let us go through the loop control statements briefly

| Sr.No. | Control Statement & Description |
|---|---|
| 1 | break statement Terminates the loop statement and transfers execution to the statement immediately following the loop. |
| 2 | continue statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| 3 | pass statement The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute. |



## Python Loops

**Python while Loop Statements**

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

```
while expression
     statements(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.

## Python Loops

**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 "
```

## Python Loops

**Python for Loop Statements**

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

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

Example:

```
        for letter in "BVICAM"
                print 'Current Letter:' , letter

        colors = ['RED', 'GREEN', 'BLUE']
        for color in colors:
                print 'Current Color:' , color
```

## Functions

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.

Basically, we can divide functions into the following two types:

- **Built-in functions** - Functions that are built into Python.
- For example – abs(), complex(), dict(), float(), format(), id()

- **User-defined functions** - Functions defined by the users themselves.

## User-defined functions

**Advantages of user-defined functions**

- User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug. (Modular Programming)
- If repeated code occurs in a program, functions can be used to include those codes and execute when needed by calling that function.
- Programmers working on large project can divide the workload by making different functions.

## User-defined functions

**Defining a *user-defined functions***

- 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.
- 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. A return statement with no arguments is the same as return None.
- **pass** statement can be used in case function has an empty body.

## Functions

```
def function_name( paramenters ) :
        "Function_docstring"
        Fucntion_Body
        return [expression]
```

Example :

```
def print_me( str ):
        "This prints a passed string into this function"
        print str
        return

print_me("Test String")
```

## Arguments

**Arguments**

Information can be passed into functions as arguments.

You can add as many arguments as you want, just separate them with a comma.

**Number of Arguments**

By default, a function must be called with the correct number of arguments.

**Arbitrary Arguments, *args**

- If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.
- This way the function will receive a *tuple* of arguments, and can access the items accordingly

```
def my_function(*students)
          print(“The topper of the class is ” + students[3])
```

## Arguments

**Keyword Arguments**

- You can also send arguments with the *key* = *value* syntax.
- This way the order of the arguments does not matter.

```
def my_function(stud1, stud2, stud3, stud4)
           print(“The topper of the class is ” + students[3])

my_function(stud2=‘Amit’, stud3=‘Suman’, stud1= ‘Nikita’, stud4=‘Parth’)
```

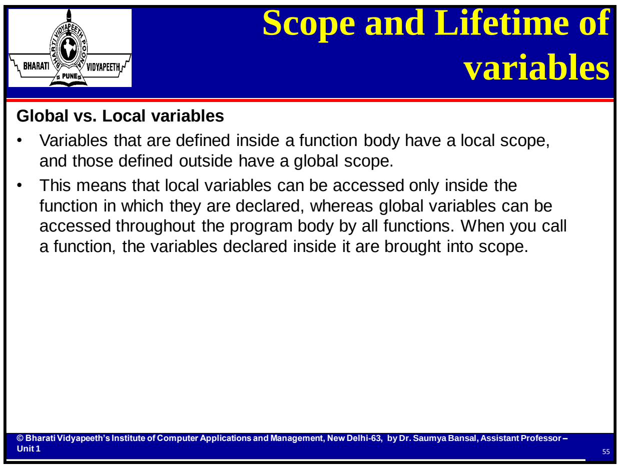
**Default Parameter Value**

```
def my_function(country = ‘INDIA’)
           print(“I am from “ + country)
my_fucntion(“Canada”)
my_fucntion ()
```

## Scope and Lifetime of variables

- Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a local scope.
- The lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.
- They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

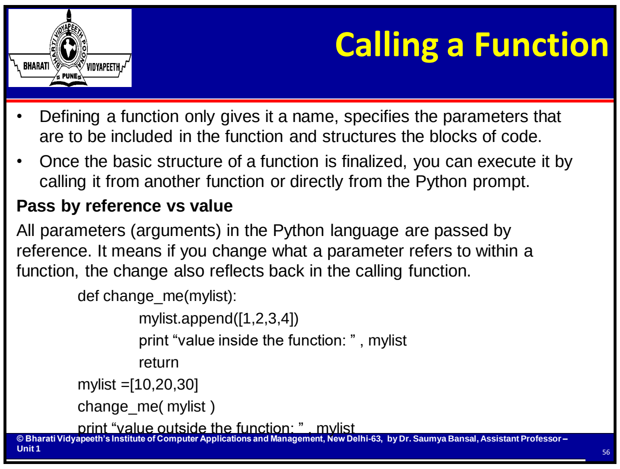
```
def my_function():
          x=10
          print(“Value of x inside function = ” , x)
my_function()
x=20
print(“Value of x outside function = ” , x)
```



## Scope and Lifetime of variables

**Global vs. Local variables**

- Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.
- This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope.



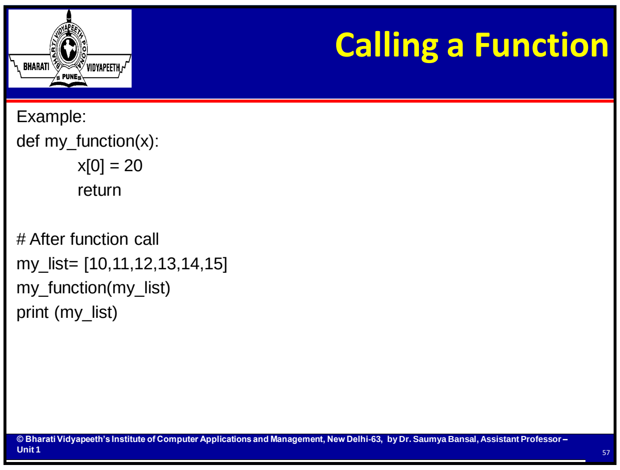
## Calling a Function

- 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.

**Pass by reference vs value**

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

```
def change_me(mylist):
        mylist.append([1,2,3,4])
        print “value inside the function: ” , mylist
        return
mylist =[10,20,30]
change_me( mylist )
print “value outside the function: ” , mylist
```



## Calling a Function

Example:

```
def my_function(x):
        x[0] = 20
        return

# After function call
my_list= [10,11,12,13,14,15]
my_function(my_list)
print (my_list)
```

## Calling a Function

When we pass a reference and change the received reference to something else, the connection between the passed and received parameter is broken.

```
def my_function(x):
        x = [20, 30, 40]
        print (my_list)
        return

# After function call
my_list= [10,11,12,13,14,15]
my_function(my_list)
print (my_list)
```

## The *Anonymous* Functions

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

**Syntax**

The syntax of *lambda* functions contains only a single statement, which is as follows −

```
        Lambda [ agr1, [arg2, ….. agrn ]]: expression
Sum = lambda agr1, agr2 : arg1 +arg2
Print "Value of total = " , sum(10,10)
Print "Value of total = " , sum(20,20)
```

## Rercursion

Recursion is the process of defining something in terms of itself.

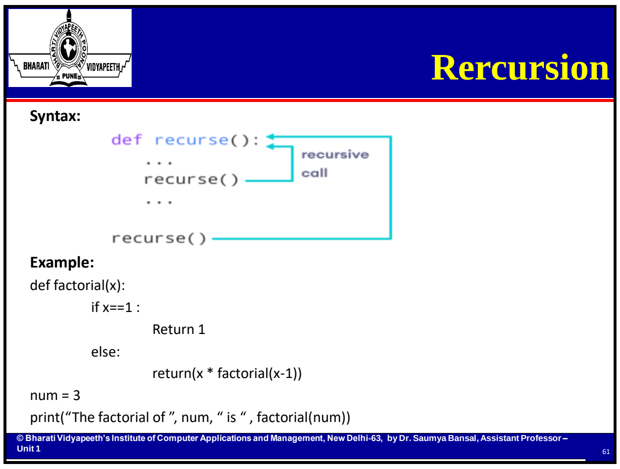
In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

**Advantages of using recursion**

- A complicated function can be split down into smaller sub-problems utilizing recursion.
- Sequence creation is simpler through recursion than utilizing any nested iteration.
- Recursive functions render the code look simple and effective.

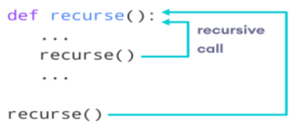
**Disadvantages of using recursion**

- A lot of memory and time is taken through recursive calls which makes it expensive for use.
- Recursive functions are challenging to debug.
- The reasoning behind recursion can sometimes be tough to think through.



## Rercursion

**Syntax:**

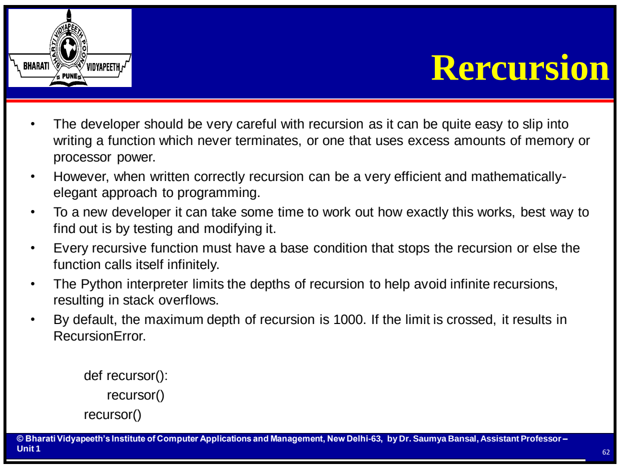


```
def recurse():
    ...
    recurse()
    ...

recurse()
```

**Example:**

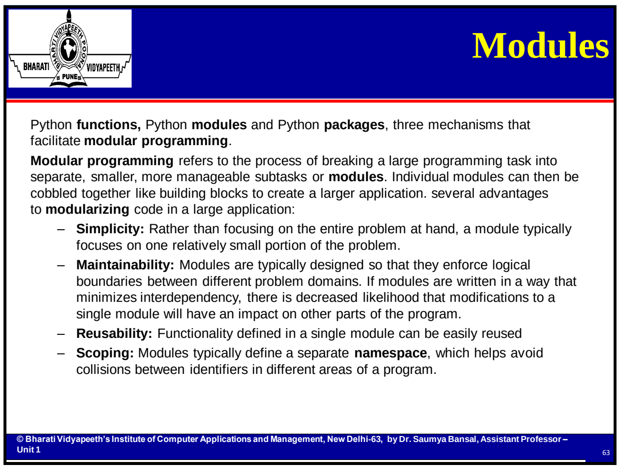
```
def factorial(x):
        if x==1 :
                Return 1
        else:
                return(x * factorial(x-1))
num = 3
print("The factorial of ", num, " is " , factorial(num))
```



## Rercursion

- The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power.
- However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.
- To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.
- 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.
- By default, the maximum depth of recursion is 1000. If the limit is crossed, it results in RecursionError.

```
def recursor():
    recursor()
recursor()
```

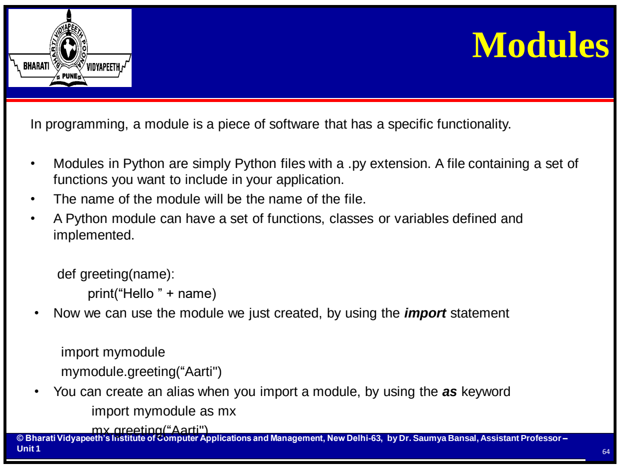


## Modules

Python **functions,** Python **modules** and Python **packages**, three mechanisms that facilitate **modular programming**.

**Modular programming** refers to the process of breaking a large programming task into separate, smaller, more manageable subtasks or **modules**. Individual modules can then be cobbled together like building blocks to create a larger application. several advantages to **modularizing** code in a large application:

- **Simplicity:** Rather than focusing on the entire problem at hand, a module typically focuses on one relatively small portion of the problem.
- **Maintainability:** Modules are typically designed so that they enforce logical boundaries between different problem domains. If modules are written in a way that minimizes interdependency, there is decreased likelihood that modifications to a single module will have an impact on other parts of the program.
- **Reusability:** Functionality defined in a single module can be easily reused
- **Scoping:** Modules typically define a separate **namespace**, which helps avoid collisions between identifiers in different areas of a program.



## Modules

In programming, a module is a piece of software that has a specific functionality.

- Modules in Python are simply Python files with a .py extension. A file containing a set of functions you want to include in your application.
- The name of the module will be the name of the file.
- A Python module can have a set of functions, classes or variables defined and implemented.

```
 def greeting(name):
        print("Hello " + name)
```

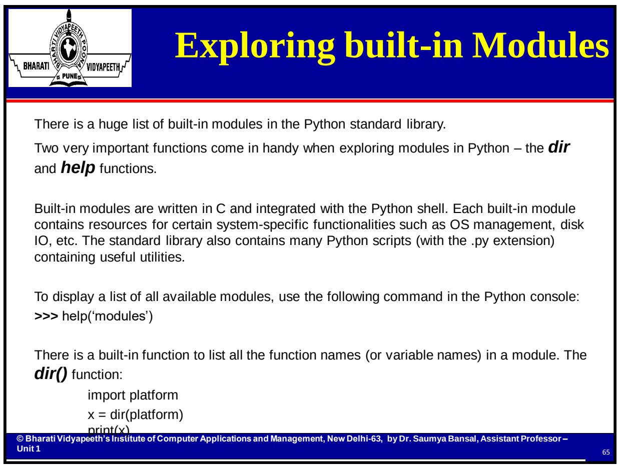
- Now we can use the module we just created, by using the ***import*** statement

```
 import mymodule
 mymodule.greeting("Aarti")
```

- You can create an alias when you import a module, by using the ***as*** keyword

```
        import mymodule as mx
        mx.greeting("Aarti")
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Saumya Bansal, Assistant Professor – Unit 1 64



## Exploring built-in Modules

There is a huge list of built-in modules in the Python standard library.

Two very important functions come in handy when exploring modules in Python – the ***dir*** and ***help*** functions.

Built-in modules are written in C and integrated with the Python shell. Each built-in module contains resources for certain system-specific functionalities such as OS management, disk IO, etc. The standard library also contains many Python scripts (with the .py extension) containing useful utilities.

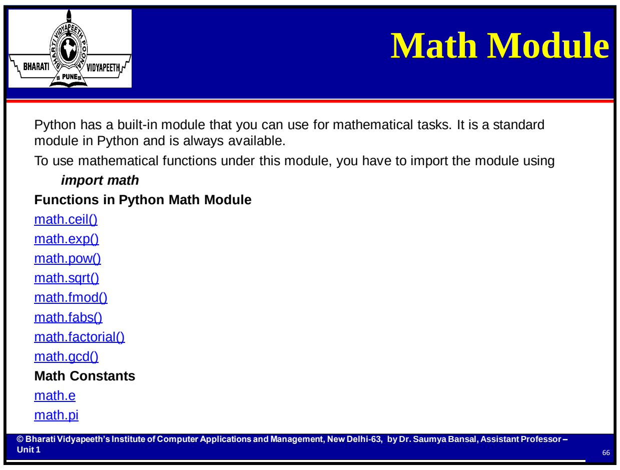
To display a list of all available modules, use the following command in the Python console:

```
>>> help('modules')
```

There is a built-in function to list all the function names (or variable names) in a module. The ***dir()*** function:

```
        import platform
        x = dir(platform)
        print(x)
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Saumya Bansal, Assistant Professor – Unit 1 65



## Math Module

Python has a built-in module that you can use for mathematical tasks. It is a standard module in Python and is always available.

To use mathematical functions under this module, you have to import the module using

***import math***

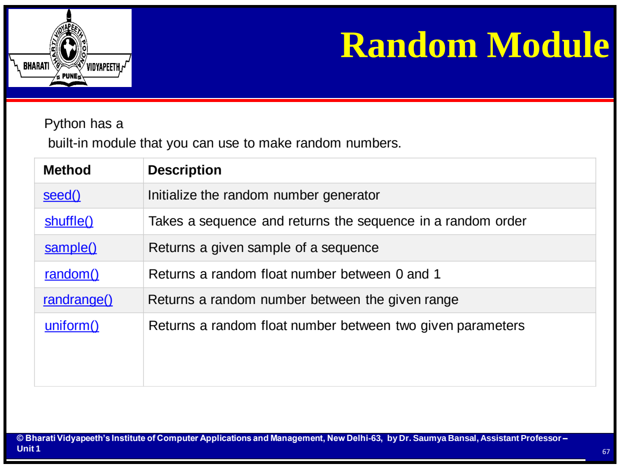
**Functions in Python Math Module**

math.ceil()
math.exp()
math.pow()
math.sqrt()
math.fmod()
math.fabs()
math.factorial()
math.gcd()

**Math Constants**

math.e
math.pi

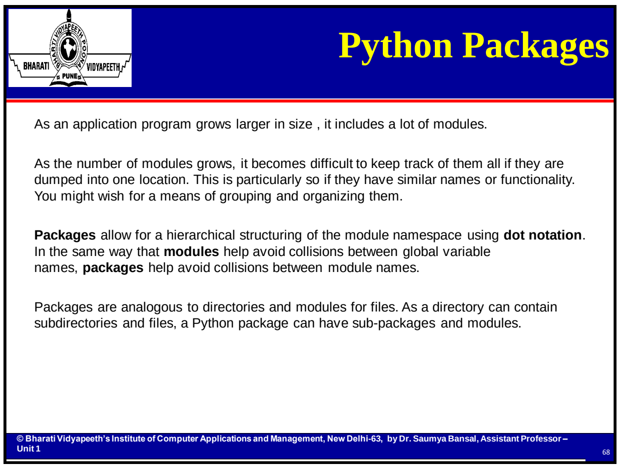
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Saumya Bansal, Assistant Professor – Unit 1 66



## Random Module

Python has a
built-in module that you can use to make random numbers.

| Method | Description |
|---|---|
| seed() | Initialize the random number generator |
| shuffle() | Takes a sequence and returns the sequence in a random order |
| sample() | Returns a given sample of a sequence |
| random() | Returns a random float number between 0 and 1 |
| randrange() | Returns a random number between the given range |
| uniform() | Returns a random float number between two given parameters |



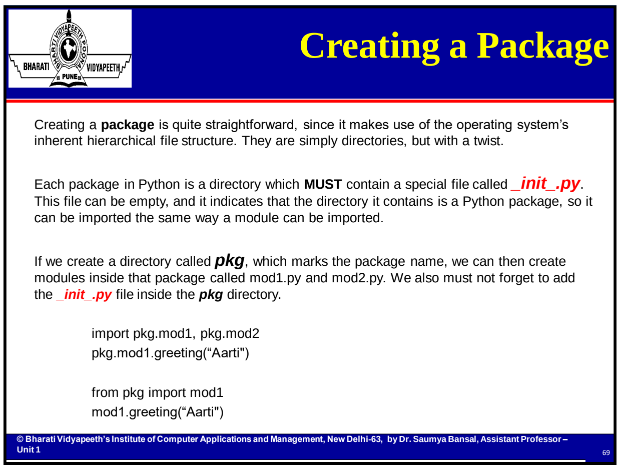
## Python Packages

As an application program grows larger in size , it includes a lot of modules.

As the number of modules grows, it becomes difficult to keep track of them all if they are dumped into one location. This is particularly so if they have similar names or functionality. You might wish for a means of grouping and organizing them.

**Packages** allow for a hierarchical structuring of the module namespace using **dot notation**. In the same way that **modules** help avoid collisions between global variable names, **packages** help avoid collisions between module names.

Packages are analogous to directories and modules for files. As a directory can contain subdirectories and files, a Python package can have sub-packages and modules.



## Creating a Package

Creating a **package** is quite straightforward, since it makes use of the operating system's inherent hierarchical file structure. They are simply directories, but with a twist.

Each package in Python is a directory which **MUST** contain a special file called ***_init_.py***. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.

If we create a directory called ***pkg***, which marks the package name, we can then create modules inside that package called mod1.py and mod2.py. We also must not forget to add the ***_init_.py*** file inside the ***pkg*** directory.

```
import pkg.mod1, pkg.mod2
pkg.mod1.greeting("Aarti")

from pkg import mod1
mod1.greeting("Aarti")
```