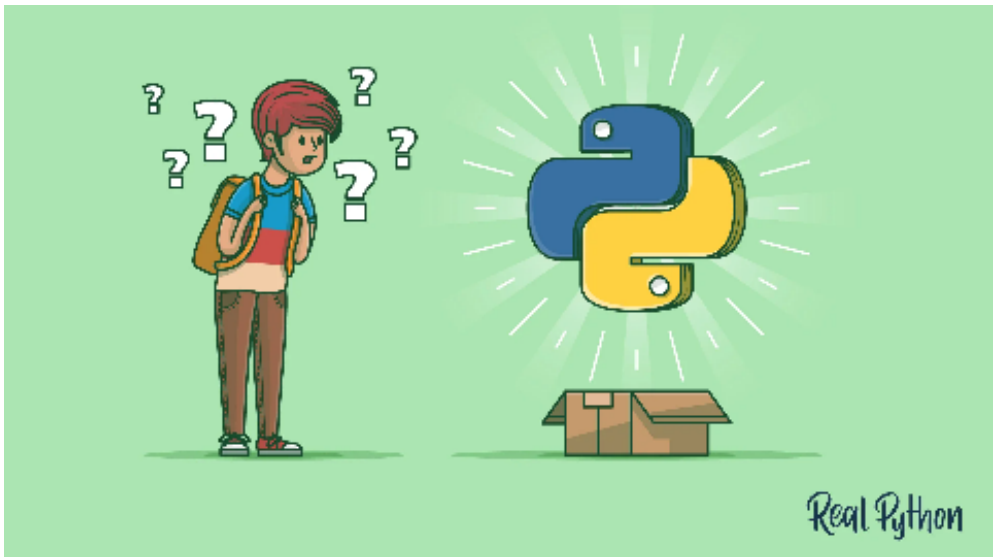
# NOIDA INSTITUTE OF ENGINEERING & TECHNOLOGY, GR. NOIDA

(An Autonomous Institute), Estd. 2001

*Approved by AICTE and Affiliated to Dr APJ Abdul Kalam Technical University, Lucknow*




# python™ Workshop: Day-1

# About Speaker

| | |
|---|---|
| Name | **Mr. RAJU CHAUHAN** |
| Designation | Assistant Professor |
| Department | Computer Science (Data Science) |
| Teaching Experience | 7+ Years |
| Qualification | MTech (Computer Engineering) |

# Outline of Lecture

- To understand the Programming Cycle for Python
- To understand Python IDE
- Elements of Python
  - Keywords
  - Identifiers
  - Variables
  - Data Types
  - Type Conversion
- Operators in Python
- Operators precedence and associativity
- Expressions in Python
- Various print format

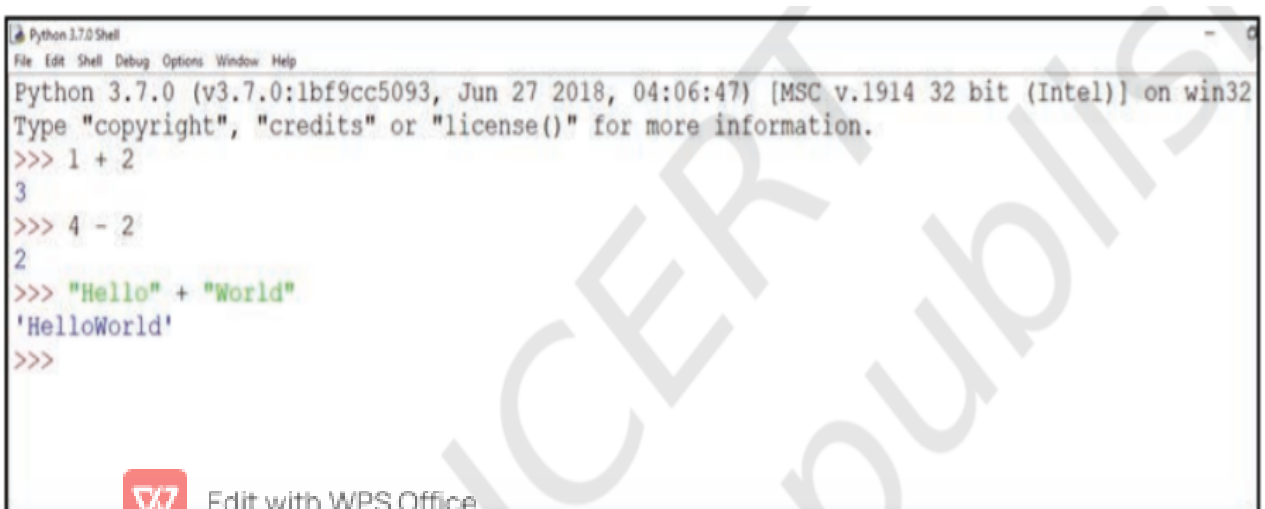
# Features of Python

- Python is a High Level Language. It is a free and open source language.
- It is an interpreted language, as Python programs are executed by an interpreter.
- Python programs are easy to understand as they have a clearly defined syntax and relatively simple structure.
- Python is case-sensitive. For example, *NUMBER* and *number* are not same in Python.
- Python is portable and platform independent, means it can run on various operating systems and hardware platforms.
- Python has a rich library of predefined functions.
- Python is also helpful in web development. Many popular web services and applications are built using Python.
- Python uses indentation for blocks and nested blocks.

# Execution mode in Python

- There are two execution mode in Python
  1. **Interactive Mode**
     - In interactive mode, instruction is types one-by-one at Python Prompt (>>>)
     - It is convenient to test single line code for instant execution.
     - But, in interactive mode we can't save statement for future use

Figure 1



# Execution mode in Python

- There are two execution mode in Python

## 2. Script Mode

- In the script mode, a Python program is written in a file, save it and then use the interpreter to execute it.
- Python scripts are saved as files where file name has extension ".py"
- By default, the Python scripts are saved in the Python installation folder.

- ***To execute a script***
  - Type the file name along with the path at the prompt. For example, if the name of the file is *myFile.py*, we type *myFile.py*. We can otherwise open the program directly from IDLE as shown in Figure 2.
  - While working in the script mode, after saving the file, click [Run]-> [Run Module] from the menu as shown in Figure 3.
  - The output appears on shell as shown in Figure 4.

# Execution mode in Python


Figure 2

Figure 3

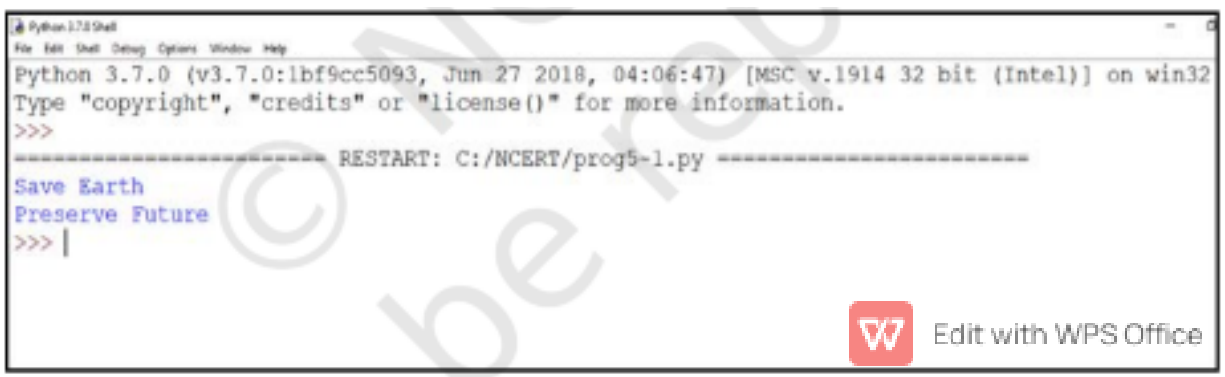
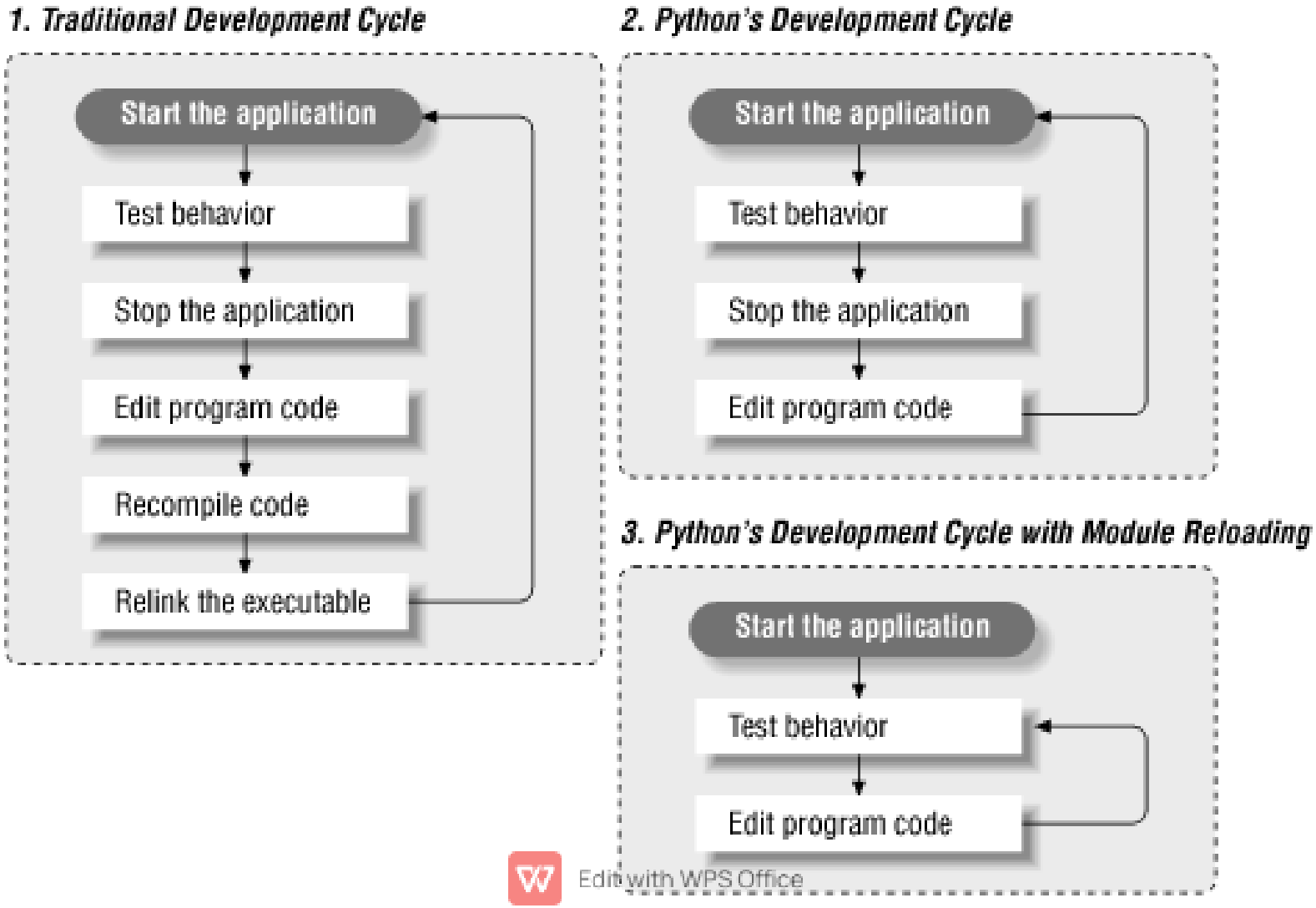




Figure 4

# Programming Cycle for Python

- Python's development cycle is dramatically shorter than that of traditional languages.
- In Python, there are no compile or link steps.
- Python programs simply import modules at runtime and use the objects they contain.
- Python programs run immediately after changes are made.
- And in cases where dynamic module reloading can be used, it's even possible to change and reload parts of a running program without stopping it at all.

# Programming Cycle for Python



# Python IDE

- The Jupyter Notebook is an open-source web application that allows  to create and share documents that contain live code, equations, visualizations and narrative text.
- Its Uses include data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.
- Download link as per OS version:

  https://www.anaconda.com/products/individual-d

- Download Procedure Link:

  https://www.youtube.com/watch?v=kbTSj5xTgNc

# Python IDE

- Spyder
  - https://www.spyder-ide.org/
- IDLE
  - https://docs.python.org/3/library/idle.html
- Sublime Text 3
  - https://www.sublimetext.com/3
- Jupyter
  - https://jupyter.org/install.html

# Elements of Python

- Keywords
- Identifiers
- Variables
- Data Types
- Type Conversion

# Keywords

- Keywords are reserved words.
- Each keyword has a specific meaning to the Python interpreter, and we can use a keyword in our program only for the purpose for which it has been defined.

```
help("keywords")
```

```
Here is a list of the Python keywords.  Enter any keyword to get more help.

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

# Identifiers

- identifiers are names used to identify a variable, function, or other entities in a program.
- The rules for naming an identifier in Python
  - The name should begin with an uppercase or a lowercase alphabet or an underscore sign (_).
  - an identifier cannot start with a digit.
  - It can be of any length. (However, it is preferred to keep it short and meaningful).
  - It should not be a keyword or reserved word.
  - We cannot use special symbols like !, @, #, $, %, etc., in identifiers.

| Valid Identifiers | Invalid Identifiers | Reason Invalid |
|---|---|---|
| totalSales | 'totalSales' | quotes not allowed |
| totalsales | total sales | spaces not allowed |
| salesFor2010 | 2010Sales | cannot begin with a digit |
| sales_for_2010 | _2010Sales | should not begin with an underscore |

# Variables

- A variable in a program is uniquely identified by a name (identifier).
- Variable in Python refers to an object — an item or element that is stored in the memory. Value of a variable can be a string (e.g., 'b', 'Global Citizen'), numeric (e.g., 345) or any combination of alphanumeric characters (CD67).
- In Python we can use an assignment statement to create new variables and assign specific values to them.
- Variable declaration and initialization (*Figure 5*) and a program to print the values of variables (*Figure 6*)

```
gender    = 'M'
message   = "Keep Smiling"
price     = 987.9
```

Figure 5

```
#Program 5-2
#To display values of variables
message = "Keep Smiling"
print(message)
userNo = 101
print('User Number is', userNo)
```

Figure 6

Output:

```
Keep Smiling
User Number is 101
```

# Variables

- Variable declaration is implicit in Python, means variables are automatically declared and defined when they are assigned a value the first time.
- Variables must always be assigned values before they are used in expressions as otherwise it will lead to an error in the program.
- Wherever a variable name occurs in an expression, the interpreter replaces it with the value of that particular variable.
- Example: Area of Rectangle (*Figure 7*)

Figure 7

```
#To find the area of a rectangle
length = 10
breadth = 20
area = length * breadth
print(area)
```

Output:



```
200
```

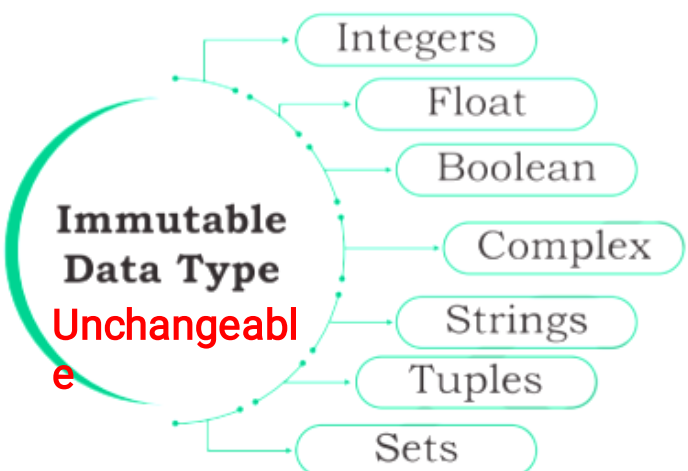
# Comments in Python

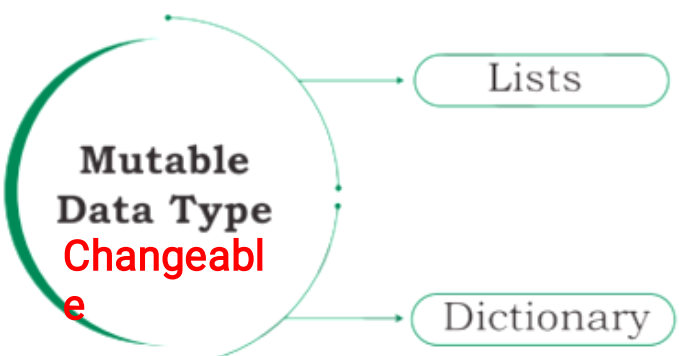
- Comments are used to add a remark or a note in the source code.
- Comments are not executed by interpreter.
- Comments are added with the purpose of making the source code easier for humans to understand.
- In Python, a comment starts with # (hash sign).
- Everything following the # till the end of that line is treated as a comment and the interpreter simply ignores it while executing the statement.

```
#Variable amount is the total spending on
#grocery
amount = 3400
#totalMarks is sum of marks in all the tests
#of Mathematics
totalMarks = test1 + test2 + finalTest
```

# Data Types

- **Mutable Data Type** : Variables whose values can be changed after they are created and assigned are called mutable
- **Immutable Data Type**: Variables whose values cannot be changed after they are created and assigned are called immutable.



# Data Types

- Data type identifies the type of data values a variable can hold and the operations that can be performed on that data.

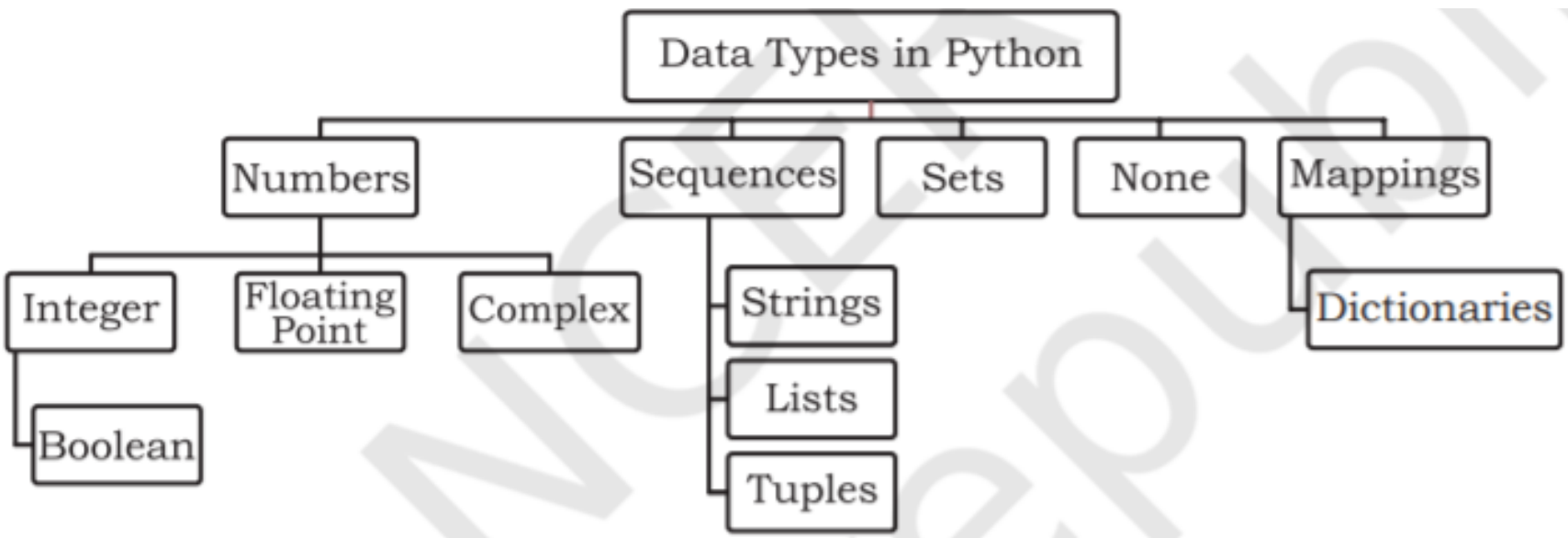



*Figure 8* Different types of Data Types

# Number

- Number data type stores numerical values only.
- It is further classified into three different types:
  - int,
  - float and
  - complex.
- Boolean data type (bool) is a subtype of integer.
  - It is a unique data type, consisting of two constants, True and False.
  - Boolean True value is non-zero, non-null and non-empty. Boolean False is the value zero.

| Type/ Class | Description | Examples |
|---|---|---|
| int | integer numbers | –12, –3, 0, 125, 2 |
| float | real or floating point numbers | –2.04, 4.0, 14.23 |
| complex | complex numbers | 3 + 4j, 2 – 2j |

```
x = 10
y = 12.34
z = 2 + 3j
p = True
print("Data Type of x:",type(x))
print("Data Type of y:",type(y))
print("Data Type of z:",type(z))
print("Data Type of p:",type(p))
```

```
Data Type of x: <class 'int'>
Data Type of y: <class 'float'>
Data Type of z: <class 'complex'>
Data Type of p: <class 'bool'>
```

# Sequence

- A Python sequence is an ordered collection of items, where each item is indexed by an integer.
- The three types of sequence data types available in Python are
  - Strings,
  - Lists and
  - Tuples.

# String (“NIET”,’Hello’,”456”)

- String is a group of characters.
- These characters may be alphabets, digits or special characters including spaces.
- String values are enclosed either in single quotation marks (e.g., ‘Hello’) or in double quotation marks (e.g., “Hello”).
- The quotes are not a part of the string, they are used to mark the beginning and end of the string for the interpreter
- We cannot perform numerical operations on strings, even when the string contains a numeric value, as in str2.

```
firstName = "Raj"
secondName ='Chauhan'
age = "32"
print("Hello Mr."+firstName+" "+secondName+" your age is "+age+"year")
```

```
Hello Mr.Raj Chauhan your age is 32year
```

# List

- List is a sequence of items separated by commas and the items are enclosed in square brackets [ ].
- List is heterogeneous collection of data types

```
studentData = ["Raj Chauhan",35,7.6,"A-250"]
#show the item of list
print(studentData)
##Accessing item of list studentData
studentData[0]
```

```
['Raj Chauhan', 35, 7.6, 'A-250']
'Raj Chauhan'
```

# Tuple

- Tuple is a sequence of items separated by commas and items are enclosed in parenthesis ( ).
- Once created, we cannot change the tuple.

```
empData = ('Arjun Raj',"2K21F0012",35,7.6)
print(empData)
print(empData[2])
```

```
('Arjun Raj', '2K21F0012', 35, 7.6)
35
```

# Set

- Set is an unordered collection of items separated by commas and the items are enclosed in curly brackets { }.
- A set is similar to list, except that it cannot have duplicate entries. Once created, elements of a set cannot be changed.

```
manData = {12,2.3,"Arjun Raj",'A234'}
print(type(manData))
manData
```

```
<class 'set'>
{12, 2.3, 'A234', 'Arjun Raj'}
```

# None

- None is a special data type with a single value.
- It is used to signify the absence of value in a situation.
- None supports no special operations, and it is neither False nor 0 (zero).

```
Hello = None
print(type(Hello))
print(Hello)
```

```
<class 'NoneType'>
None
```

# Mapping

- Mapping is an unordered data type in Python.
- Currently, there is only one standard mapping data type in Python called **dictionary**.
- **Dictionary**
  - Dictionary in Python holds data items in key-value pairs.
  - Items in a dictionary are enclosed in curly brackets { }.
  - Dictionaries permit faster access to data.
  - Every key is separated from its value using a colon (:) sign.
  - The key : value pairs of a dictionary can be accessed using the key.
  - The keys are usually strings and their values can be any data type.
  - In order to access any value in the dictionary, we have to specify its key in square brackets [ ].

# Dictionary

```
resultAnalysis = {'Name':'Raj Chauhan','Age':32,'Designation':"Assistant Professor",'Department':"Data Science"}
resultAnalysis
```

```
{'Name': 'Raj Chauhan',
 'Age': 32,
 'Designation': 'Assistant Professor',
 'Department': 'Data Science'}
```

```
resultAnalysis['Name']
```

```
'Raj Chauhan'
```

# Type Conversion

- Explicit Conversion
- Implicit Conversion

# Explicit Conversion

- Forceful type conversion
- Programmer intentionally do this conversion
- General form: *(new_data_type) (expression)*
- This type of conversion has risk of losing information because programmer forcefully do this.
- For example:

```
x = 67.34
y = int(x)
print("Value of x after explicit conversion :",y)

Value of x after explicit conversion : 67
```

# Explicit Conversion

| Function | Description |
|---|---|
| `int(x)` | Converts x to an integer |
| `float(x)` | Converts x to a floating-point number |
| `str(x)` | Converts x to a string representation |
| `chr(x)` | Converts ASCII value of x to character |
| `ord(x)` | returns the character associated with the ASCII code x |

# Explicit Conversion

```
x = 10
y = 45.23
print("Sum before type conversion :",x+y)
print("Data Type of x :",type(x))
print("Data Type of y :",type(y))
z = x + int(y)
print("Data Type of z :",type(z))
print("Sum after type conversion :",z)
```

```
Sum before type conversion : 55.23
Data Type of x : <class 'int'>
Data Type of y : <class 'float'>
Data Type of z : <class 'int'>
Sum after type conversion : 55
```

# Implicit Conversion

- This conversion is also known as automatic type conversion.
- There is no involvement of programmer.
- Example

```
a = 23
b = 2.5
print("Data Type of a :",type(a))
print("Data Type of b :",type(b))
c = a + b
print("Data Type of c :",type(c))
print(c)
```

```
Data Type of a : <class 'int'>
Data Type of b : <class 'float'>
Data Type of c : <class 'float'>
25.5
```

# Operators in Python

- An operator is used to perform specific mathematical or logical operation on values.
- The values that the operators work on are called operands.
- Operator supported by Python
  - Arithmetic Operators
  - Relational Operators
  - Assignment Operators
  - Logical Operators
  - Identity Operators
  - Mapping Operator

# Arithmetic Operator

```
x = 10
y = 7
print(" + Operator :",x+y)
print(" - Operator :",x-y)
print(" * Operator :",x*y)
print(" / Operator :",x/y)
print(" % Operator :",x%y)
print(" // Operator :",x//y)
print(" ** Operator :",x**y)
```

```
+ Operator : 17
- Operator : 3
* Operator : 70
/ Operator : 1.4285714285714286
% Operator : 3
// Operator : 1
** Operator : 10000000
```

| Operator | Operation | Description | Example (Try in Lab) |
|---|---|---|---|
| + | Addition | Adds the two numeric values on either side of the operator<br><br>This operator can also be used to concatenate two strings on either side of the operator | >>> num1 = 5<br>>>> num2 = 6<br>>>> num1 + num2<br>11<br>>>> str1 = "Hello"<br>>>> str2 = "India"<br>>>> str1 + str2<br>'HelloIndia' |
| | Subtraction | Subtracts the operand on the right from the operand on the left | >>> num1 = 5<br>>>> num2 = 6<br>>>> num1 - num2<br>-1 |
| * | Multiplication | Multiplies the two values on both side of the operator<br><br>Repeats the item on left of the operator if first operand is a string and second operand is an integer value | >>> num1 = 5<br>>>> num2 = 6<br>>>> num1 * num2<br>30<br>>>> str1 = 'India'<br>>>> str1 * 2<br>'IndiaIndia' |
| / | Division | Divides the operand on the left by the operand on the right and returns the quotient | >>> num1 = 8<br>>>> num2 = 4<br>>>> num2 / num1<br>0.5 |
| % | Modulus | Divides the operand on the left by the operand on the right and returns the remainder | >>> num1 = 17<br>>>> num2 = 5<br>>>> num1 % num2<br>2 |
| // | Floor Division | Divides the operand on the left by the operand on the right and returns the quotient by removing the decimal part. It is sometimes also called integer division. | >>> num1 = 17<br>>>> num2 = 4<br>>>> num1 // num2<br>4<br>>>> num2 // num1<br>0 |
| ** | Exponent | Performs exponential (power) calculation on operands. That is, raise the operand on the left to the power of the operand on the right | >>> num1 = 3<br>>>> num2 = 4<br>>>> num1 ** num2<br>81 |

# Relational Operator

```
p = 10
q = 20
r = 10
print("Relational Operators")
print(" == Operator :",p==r)
print(" != Operator :",p!=r)
print(" >= Operator :",p>=r)
print(" <= Operator :",p<=r)
print(" > Operator :",p>r)
print(" == Operator :",p<r)
```

```
Relational Operators
 == Operator : True
 != Operator : False
 >= Operator : True
 <= Operator : True
 > Operator : False
 == Operator : False
```

| Operator | Operation | Description | Example (Try in Lab) |
|---|---|---|---|
| == | Equals to | If the values of two operands are equal, then the condition is True, otherwise it is False | >>> num1 == num2<br>False<br>>> str1 == str2<br>False |
| != | Not equal to | If values of two operands are not equal, then condition is True, otherwise it is False | >>> num1 != num2<br>True<br>>>> str1 != str2<br>True<br>>>> num1 != num3<br>False |
| > | Greater than | If the value of the left-side operand is greater than the value of the right-side operand, then condition is True, otherwise it is False | >>> num1 > num2<br>True<br>>>> str1 > str2<br>True |
| < | Less than | If the value of the left-side operand is less than the value of the right-side operand, then condition is True, otherwise it is False | >>> num1 < num3<br>False<br>>>> str2 < str1<br>True |
| >= | Greater than or equal to | If the value of the left-side operand is greater than or equal to the value of the right-side operand, then condition is True, otherwise it is False | >>> num1 >= num2<br>True<br>>>> num2 >= num3<br>False<br>>>> str1 >= str2<br>True |
| <= | Less than or equal to | If the value of the left operand is less than or equal to the value of the right operand, then is True otherwise it is False | >>> num1 <= num2<br>False<br>>>> num2 <= num3<br>True<br>>>> str1 <= str2<br>False |

# Assignment Operators

```
firstValue = 10
secondValue = firstValue
print("= Operator :",secondValue)

num1 = 12
num2 = 10
num2 += num1
print("+= operator :",num2)

num1 = 12
num2 = 10
num2 -= num1
print("-= operator :",num2)

num1 = 12
num2 = 10
num2 *= num1
print("*= operator :",num2)
```

```
= Operator : 10
+= operator : 22
-= operator : -2
*= operator : 120
```

| Operator | Description | Example (Try in Lab) |
|---|---|---|
| = | Assigns value from right-side operand to left-side operand | >>> num1 = 2<br>>>> num2 = num1<br>>>> num2<br>2<br>>>> country = 'India'<br>>>> country<br>'India' |
| += | It adds the value of right-side operand to the left-side operand and assigns the result to the left-side operand<br>***Note:*** x += y is same as x = x + y | >>> num1 = 10<br>>>> num2 = 2<br>>>> num1 += num2<br>>>> num1<br>12<br>>>> num2<br>2<br>>>> str1 = 'Hello'<br>>>> str2 = 'India'<br>>>> str1 += str2<br>>>> str1<br>'HelloIndia' |
| -= | It subtracts the value of right-side operand from the left-side operand and assigns the result to left-side operand<br>***Note:*** x -= y is same as x = x - y | >>> num1 = 10<br>>>> num2 = 2<br>>>> num1 -= num2<br>>>> num1<br>8 |
| *= | It multiplies the value of right-side operand with the value of left-side operand and assigns the result to left-side operand<br>***Note:*** x *= y is same as x = x * y | >>> num1 = 2<br>>>> num2 = 3<br>>>> num1 *= 3<br><br>>>> num1<br>6<br>>>> a = 'India'<br>>>> a *= 3<br>>>> a<br>'IndiaIndiaIndia' |

# Assignment Operators

```
num1 = 12
num2 = 10
num2 /= num1
print("/= operator :",num2)

num1 = 12
num2 = 15
num2 //= num1
print("//= operator :",num2)

num1 = 2
num2 = 10
num2 **= num1
print("**= operator :",num2)

num1 = 7
num2 = 10
num2 %= num1
print("%= operator :",num2)
```

```
/= operator : 0.8333333333333334
//= operator : 1
**= operator : 100
%= operator : 3
```

| | | |
|---|---|---|
| /= | It divides the value of left-side operand by the value of right-side operand and assigns the result to left-side operand<br>***Note:*** x /= y is same as x = x / y | >>> num1 = 6<br>>>> num2 = 3<br>>>> num1 /= num2<br>>>> num1<br>2.0 |
| %= | It performs modulus operation using two operands and assigns the result to left-side operand<br>***Note:*** x %= y is same as x = x % y | >>> num1 = 7<br>>>> num2 = 3<br>>>> num1 %= num2<br>>>> num1<br>1 |
| //= | It performs floor division using two operands and assigns the result to left-side operand<br>***Note:*** x //= y is same as x = x // y | >>> num1 = 7<br>>>> num2 = 3<br>>>> num1 //= num2<br>>>> num1<br>2 |
| **= | It performs exponential (power) calculation on operators and assigns value to the left-side operand<br>***Note:*** x **= y is same as x = x ** y | >>> num1 = 2<br>>>> num2 = 3<br>>>> num1 **= num2<br>>>> num1<br>8 |

# Logical Operators

```
n = 1
m = 2
# and Operator
print("and operator ")
if n == 1 and m == 2:
    print("Both conditions are TRUE")
print("or operator ")
# or Operator
if n == 5 or m == 3:
    print("Any one condition is TRUE")
else:
    print("Both conditions are FALSE")
# not Operator
print("not operator")
if not m == 2:
    print("condition not MATCH")
else:
    print("condition MATCH")
```

```
and operator
Both conditions are TRUE
or operator
Both conditions are FALSE
not operator
condition MATCH
```

| Operator | Operation | Description | Example (Try in Lab) |
|---|---|---|---|
| and | Logical AND | If both the operands are True, then condition becomes True | >>> True and True<br>True<br>>>> num1 = 10<br>>>> num2 = -20<br>>>> bool(num1 and num2)<br>True<br>>>> True and False<br>False<br>>>> num3 = 0<br>>>> bool(num1 and num3)<br>False<br>>>> False and False<br>False |
| or | Logical OR | If any of the two operands are True, then condition becomes True | >>> True or True<br>True<br>>>> True or False<br>True<br>>>> bool(num1 or num3)<br>True<br>>>> False or False<br>False |
| not | Logical NOT | Used to reverse the logical state of its operand | >>> num1 = 10<br>>>> bool(num1)<br>True<br>>>> not num1<br>>>> bool(num1)<br>False |

# Identity Operators

```
num1 = 10
type(num1) is int

True


num1 = 10
type(num1) is float

False


num1 = 102
num2 = 103
num1 is not num2

True


num1 = 102
num2 = 102
num1 is not num2

False
```

| Operator | Description | Example (Try in Lab) |
|---|---|---|
| is | Evaluates `True` if the variables on either side of the operator point towards the same memory location and `False` otherwise. `var1` is `var2` results to `True` if id(`var1`) is equal to id(`var2`) | `>>> num1 = 5`<br>`>>> type(num1) is int`<br>`True`<br>`>>> num2 = num1`<br>`>>> id(num1)`<br>`1433920576`<br>`>>> id(num2)`<br>`1433920576`<br>`>>> num1 is num2`<br>`True` |
| is not | Evaluates to `False` if the variables on either side of the operator point to the same memory location and `True` otherwise. `var1` is not `var2` results to `True` if id(`var1`) is not equal to id(`var2`) | `>>> num1 is not num2`<br>`False` |

# Identity Operators

```
list1 = [12,23,"Raj",34.5]
12 in list1

True
```

```
list1 = [12,23,"Raj",34.5]
10 in list1

False
```

```
list1 = [12,23,"Raj",34.5]
10 not in list1

True
```

| Operator | Description | Example (Try in Lab) |
|---|---|---|
| in | Returns `True` if the variable/value is found in the specified sequence and `False` otherwise | `>>> a = [1,2,3]`<br>`>>> 2 in a`<br>`True`<br>`>>> '1' in a`<br>`False` |
| not in | Returns `True` if the variable/value is not found in the specified sequence and `False` otherwise | `>>> a = [1,2,3]`<br>`>>> 10 not in a`<br>`True`<br>`>>> 1 not in a`<br>`False` |

# Bitwise Operators

```
a = 5 #101
b = 7 #111
print("Bitwise &(AND) operator: ", a&b)
print("Bitwise |(OR) operator: ", a|b)
print("Bitwise ~(NOT) operator: ", a|(~b))
print("Bitwise ^(XOR) operator: ", a^b)
print("Bitwise <<(LEFT SHIFT) operator: ", a<<1)
print("Bitwise <<(RIGHT SHIFT) operator: ", b>>1)
```

```
Bitwise &(AND) operator:  5
Bitwise |(OR) operator:  7
Bitwise ~(NOT) operator:  -3
Bitwise ^(XOR) operator:  2
Bitwise <<(LEFT SHIFT) operator:  10
Bitwise <<(RIGHT SHIFT) operator:  3
```

| a | b | a & b | a \| b | a ^ b | ~a |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 |

# Expressions in Python

- An expression is defined as a combination of constants, variables, and operators.
- An expression always evaluates to a value.
- A value or a standalone variable is also considered as an expression but a standalone operator is not an expression.

```
(i) 100
(ii) num
(iii) num - 20.4
```

```
(iv) 3.0 + 3.14
(v) 23/3 -5 * 7(14 -2)
(vi) "Global" + "Citizen"
```

# Operators precedence and associativity

| Order of Precedence | Operators | Description |
|---|---|---|
| 1 | ** | Exponentiation (raise to the power) |
| 2 | ~ ,+, - | Complement, unary plus and unary minus |
| 3 | * ,/, %, // | Multiply, divide, modulo and floor division |
| 4 | +, - | Addition and subtraction |
| 5 | <= , < , > , >=, == , != | Relational and Comparison operators |
| 6 | =, %=, /=, //=, -=, +=, *=, **= | Assignment operators |
| 7 | is, is not | Identity operators |
| 8 | in, not in | Membership operators |
| 9 | not | Logical operators |
| 10 | and | |
| 11 | or | |

# Operators precedence and associativity

- Parenthesis can be used to override the precedence of operators. The expression within () is evaluated first.
- For operators with equal precedence, the expression is evaluated from left to right.

```
20 + 10 - 30 * 125 / 5 ** 2
```

```
-120.0
```

```
20 + 10 - (30 * (125 / (5 ** 2)))
```

```
-120.0
```

# Debugging

- The process of identifying and removing such mistakes, also known as bugs or errors, from a program is called debugging
- Errors occurring in programs can be categorised as:
  - Syntax errors
  - Logical errors
  - Runtime errors

# Syntax Errors

- The interpreter interprets the statements only if it is syntactically (as per the rules of Python) correct.
- If any syntax error is present, the interpreter shows error message(s) and stops the execution there.
- For example, parentheses must be in pairs, so the expression *(10 + 12)* is syntactically correct, whereas *(7 + 11* is not due to absence of right parenthesis.

# Logical Errors

- A logical error is a bug in the program that causes it to behave incorrectly.
- A logical error produces an undesired output but without abrupt termination of the execution of the program.
- Since the program interprets successfully even when logical errors are present in it, it is sometimes difficult to identify these errors.
- The only evidence to the existence of logical errors is the wrong output.
- While working backwards from the output of the program, one can identify what went wrong.
- For example, if we wish to find the average of two numbers 10 and 12 and we write the code as 10 + 12/2, it would run successfully and produce the result 16. Surely, 16 is not the average of 10 and 12. The correct code to find the average should have been (10 + 12)/2 to give the correct output as 11.
- Logical errors are also called semantic errors as they occur when the meaning of the program (its semantics) is not correct.

# Run Time Errors

- A runtime error causes abnormal termination of program while it is executing.
- Runtime error is when the statement is correct syntactically, but the interpreter cannot execute it.
- Runtime errors do not appear until after the program starts running or executing.
- For example, we have a statement having division operation in the program. By mistake, if the denominator entered is zero then it will give a runtime error like “division by zero”.

# Thank You
# and
# Any
# Question?