Unit I : Introduction and Basics of Python **Hours: 15**

Computer algorithms-The process of computational problem-solving Python programming language - Literals - Variables and Identifiers - Operators – Expressions- Operator Precedence, Operator Associativity - Data types - Age in Seconds Program.

COMPUTER ALGORITHMS

Python can use a wide variety of algorithms, but some of the most well-known are tree traversal, sorting, search and graph algorithms. Tree traversal algorithms are designed to visit all nodes of a tree graph, starting from the root and traversing each node according to the instructions laid out.

What Are the Types of Algorithms in Python?

- **Tree traversal algorithms** are designed to visit all nodes of a tree graph, starting from the root and traversing each node according to the instructions laid out. Traversal can occur in order, with the algorithm traversing the tree from node to edge (branches), or from the edges to the root.
- **Sorting algorithms** provide various ways of arranging data in a particular format, with common algorithms including bubble sort, merge sort, insertion sort and shell sort.
- **Searching algorithms** check and retrieve elements from different data structures, with variations including linear search and binary search.
- **Graph algorithms** traverse graphs from their edges in a depth-first (DFS) or breadth-first (BFS) manner.

HOW TO WRITE A PYTHON ALGORITHM: 6 CHARACTERISTICS

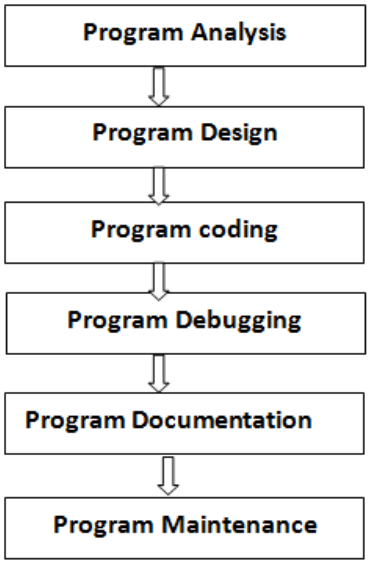
1. It is unambiguous and has clear steps.
2. The algorithm has zero or more well-defined inputs.
3. It must have one or more defined outputs.
4. The algorithm must terminate after a finite number of steps.
5. It must be feasible and exist using available resources.
6. The algorithm should be written independently of all programming code.

The process of computational problem-solving Python programming language

general process of problem solving involves the following steps.

1. Understanding the problem
2. Describing the problem in a clear, complete and error free from
3. Designing a solution to the problem by algorithm
4. Developing a computer solution to the problem
5. Testing the solution

Steps of Programming:



1. Program Analysis:

Analyze the problem and what desire output you want. Analysis and gathering knowledge about the requirement from problem.

Try to identify what requirement you need as a solution.Clarify define what the program is to do.

Clarify output and processing tasks. Define your program techniques and financial, legal feasible or not. And document the analysis.

2.Program Designing:

all the functionality converted into graphical form.

Designing is a process of problem solving and planning for a solution developer will design to develop a plan for solution using diagrammatic representation using flowchart, data flow diagram, decision tree all these tools are used.

3. Coding:

After designing coding steps comes. The goal of coding is to translate the design of the system into code in a given programming language.

It's a process of developing the program well written code can reduce the testing and maintenance effort.

4. program debugging:

Bug means "Error" and debugging means correct the error.
In this step developer test the program and try to find out any error and if error occurs then fix it.

After testing program to ensure that all modules working correctly or not.

5. Program documentation:

For programmer it is easy to understand program and code but general user cannot understand programming code and program working how to operate so it's a responsibility of developer to create a program with user manual documents.

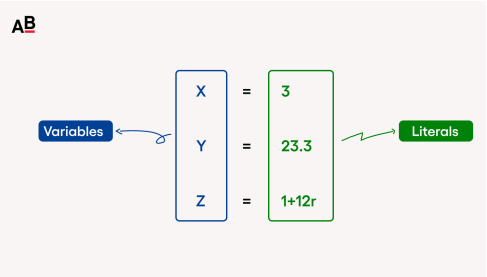
It is a instruction manual book or guide help user to understand working or program start to end. This manual guide is called program documentation.

6. Maintenance:

Used to correct and upgrade program.

Python Literals

Python Literals can be defined as data that is given in a variable or constant.



Python supports the following literals:

1. String literals:
String literals can be formed by enclosing a text in the quotes. We can use both single as well as double quotes to create a string.

Ex: "Aman" , '12345'

ypes of Strings:

There are two types of Strings supported in Python:

a) Single-line String- Strings that are terminated within a single-line are known as Single line Strings.

Example:

1. text1='hello'

b) Multi-line String - A piece of text that is written in multiple lines is known as multiple lines string.

There are two ways to create multiline strings:

1) Adding black slash at the end of each line.

Example:

```
text1='hello\
user'
print(text1)
```

```
'hellouser'
```

2) Using triple quotation marks:-

Example:

```
str2="""welcome
to
SSSIT"'
print str2
```

Output:

```
welcome
to
SSSIT
```

II. Numeric literals:
Numeric Literals are immutable. Numeric literals can belong to following four different numerical types.

Int(signed integers)	Long(long integers)	float(floating point)	Complex(complex)
Numbers(can be both positive and negative) with no fractional part.eg: 100	Integers of unlimited size followed by lowercase or uppercase L eg: 87032845L	Real numbers with both integer and fractional part eg: -26.2	In the form of a+bj where a forms the real part and b forms the imaginary part of the complex number. eg: 3.14j

```
x = 0b10100 #Binary Literals
y = 100 #Decimal Literal
z = 0o215 #Octal Literal
u = 0x12d #Hexadecimal Literal

#Float Literal
float_1 = 100.5
float_2 = 1.5e2

#Complex Literal
```

```
a = 5+3.14j

print(x, y, z, u)
print(float_1, float_2)
print(a, a.imag, a.real)
```

Output:

```
20 100 141 301
100.5 150.0
(5+3.14j) 3.14 5.0
```

III. Boolean literals:

A Boolean literal can have any of the two values: True or False.

```
x = (1 == True)
y = (2 == False)
z = (3 == True)
a = True + 10
b = False + 10

print("x is", x)
print("y is", y)
print("z is", z)
print("a:", a)
print("b:", b)
```

Output:

```
x is True
y is False
z is False
a: 11
b: 10
```

V. Literal Collections.

Python provides the four types of literal collection such as List literals, Tuple literals, Dict literals, and Set literals.

List:

- List contains items of different data types. Lists are mutable i.e., modifiable.
- The values stored in List are separated by comma(,) and enclosed within square brackets([]). We can store different types of data in a List.

Example - List literals

```
list=['John',678,20.4,'Peter']
list1=[456,'Andrew']
print(list)
print(list + list1)
```

Output:

```
['John', 678, 20.4, 'Peter']
['John', 678, 20.4, 'Peter', 456, 'Andrew']
```

Dictionary:

- Python dictionary stores the data in the key-value pair.
- It is enclosed by curly-braces {} and each pair is separated by the commas(,).

Example

```
dict = {'name': 'Pater', 'Age':18,'Roll_nu':101}
print(dict)
```

Output:

```
{'name': 'Pater', 'Age': 18, 'Roll_nu': 101}
```

Tuple:

- Python tuple is a collection of different data-type. It is immutable which means it cannot be modified after creation.
- It is enclosed by the parentheses () and each element is separated by the comma(,).

Example

```
tup = (10,20,"Dev",[2,3,4])
print(tup)
```

Output:

```
(10, 20, 'Dev', [2, 3, 4])
```

Set:

- Python set is the collection of the unordered dataset.
- It is enclosed by the {} and each element is separated by the comma(,).

Example: - Set Literals

```
set = {'apple','grapes','guava','papaya'}
print(set)
```

Output:

```
{'guava', 'apple', 'papaya', 'grapes'}
```

IDENTIFIERS

Identifier is a name used to identify a variable, function, class, module, etc. The identifier is a combination of character digits and underscore.

Rules for writing Identifiers in Python

1. The identifier is a combination of character digits and underscore and the character includes
letters in lowercase (a-z), letters in uppercase (A-Z), digits (0-9), and an underscore (_).
2. An identifier cannot begin with a digit. If an identifier starts with a digit, it will give a Syntax
error.
3. In Python, keywords are the reserved names that are built-in to Python, so a keyword cannot
be used as an identifier - they have a special meaning and we cannot use them as identifier names.
4. Special symbols like !, @, #, $, %, etc. are not allowed in identifiers.
5. Python identifiers cannot only contain digits.
6. There is no restriction on the length of identifiers.
7. Identifier names are case-sensitive.

Python Valid Identifiers Example

·abc123
·abc_de
·_abc
·ABC
·abc

Python Invalid Identifiers Example

·123abc
·abc@
·123
·for

KEYWORDS

Keywords are some predefined and reserved words in python that have special meanings. Keywords are used to define the syntax of the coding.

The keyword cannot be used as an identifier, function, and variable name.

All the keywords in python are written in lower case except True and False. There are 33 keywords in Python 3.7.

Python Keywords List

False	await	else	import
None	break	except	in
True	class	finally	is
and	continue	for	lambda
as	def	from	nonlocal
assert	del	global	not
async	elif	if	or

VARIABLES

Variables are containers for storing data values.

Creating Variables
Python has no command for declaring a variable.
A variable is created the moment you first assign a value to it.

Example
x = 5 y = "John" print(x) print(y)
Variables do not need to be declared with any particular *type*, and can even change type after they have been set.

Example
x = 4 # x is of type int x = "Sally" # x is now of type str print(x)

Case-Sensitive
Variable names are case-sensitive.

Example
This will create two variables:
a = 4 A = "Sally"

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

Example
# An integer assignment
age = 45
# A floating point salary = 1456.8
# A string name = "John"
print(age) print(salary) print(name)

Output:
45
1456.8
John

Declare the Variable
# declaring the var
Number = 100
# display print(Number)

Output:
100

Re-declare the Variable
We can re-declare the python variable once we have declared the variable already.

Example
```
# declaring the var
Number = 100
# display
print("Before declare: ", Number)
# re-declare the var Number = 120.3
print("After re-declare:", Number)
```

Output:
Before declare: 100 After re-declare: 120.3

Assigning a single value to multiple variables
Also, Python allows assigning a single value to several variables simultaneously with "=" operators. **For example:**
a = b = c = 10 print(a) print(b) print(c)

Output:
10
10
10

Assigning different values to multiple variables
Python allows adding different values in a single line with ","operators.

Example
a, b, c = 1, 20.2, "GeeksforGeeks" print(a)
print(b) print(c)

Output:
1
20.2
GeeksforGeeks

PYTHON OPERATORS

Operators are used to perform operations on variables and values.
·OPERATORS: Are the special symbols. Eg- + , * , /, etc.
·OPERAND: It is the value on which the operator is applied.
Python divides the operators in the following groups:

Types of Python Operators
Python language supports the following types of operators.
· Arithmetic Operators
· Comparison (Relational) Operators
· Assignment Operators

· Logical Operators
· Bitwise Operators
· Membership Operators
· Identity Operators

Arithmetic Operators

Python arithmetic operators are used to perform mathematical operations on numerical values.

These operations are Addition, Subtraction, Multiplication, Division, Modulus, Expoents and Floor Division.

Operator	Name	Example
+	Addition	10 + 20 = 30
-	Subtraction	20 – 10 = 10
*	Multiplication	10 * 20 = 200
/	Division	20 / 10 = 2
%	Modulus	22 % 10 = 2
**	Exponent	4**2 = 16
//	Floor Division	9//2 = 4

Example

```
a = 21
b = 10
print ("a + b : ", a + b)
print ("a - b : ", a - b)
print ("a * b : ", a * b)
print ("a / b : ", a / b)
print ("a % b : ", a % b)
print ("a ** b : ", a ** b)
print ("a // b : ", a // b)
```

Output

```
a + b : 31
a - b : 11
a * b : 210
a / b : 2.1
a % b : 1
a ** b : 16679880978201
a // b : 2
```

Comparison Operators

Python comparison operators compare the values on either sides of them and decide the relation among them.

They are also called relational operators. These operators are equal, not equal, greater than, less than, greater than or equal to and less than or equal to.

Operator	Name	Example
==	Equal	4 == 5 is not true.
!=	Not Equal	4 != 5 is true.
>	Greater Than	4 > 5 is not true.
<	Less Than	4 < 5 is true.
>=	Greater than or Equal to	4 >= 5 is not true.
<=	Less than or Equal to	4 <= 5 is true.

Example

```
a = 4
```

```
b = 5
print ("a == b : ", a == b)
 print ("a != b : ", a != b)
print ("a > b : ", a > b)
print ("a < b : ", a < b)
print ("a >= b : ", a >= b)
print ("a <= b : ", a <= b)
```

Output

a == b : False a != b : True a > b : False
a < b : True a >= b : False a <= b : True

Assignment Operators

Python assignment operators are used to assign values to variables. These operators include simple assignment operator, addition assign, subtraction assign, multiplication assign, division and assign operators etc.

Operator	Name	Example
=	Assignment Operator	a = 10
+=	Addition Assignment	a += 5 (Same as a = a + 5)
-=	Subtraction Assignment	a -= 5 (Same as a = a - 5)
*=	Multiplication Assignment	a *= 5 (Same as a = a * 5)
/=	Division Assignment	a /= 5 (Same as a = a / 5)
%=	Remainder Assignment	a %= 5 (Same as a = a % 5)
**=	Exponent Assignment	a **= 2 (Same as a = a ** 2)
//=	Floor Division Assignment	a //= 3 (Same as a = a // 3)

Example

```
a = 10
a += 5
print("a += 5 : ", a)
a -= 5
print("a -= 5 : ", a)
a *= 5
print("a *= 5 : ", a)
a /= 5
print("a /= 5 : ",a)
a %= 3
print ("a %= 3 : ", a)
a **= 2
print ("a **= 2 : ", a)
a //= 3
print ("a //= 3 : ", a)
```

Output

a += 5 : 105
a -= 5 : 100
a *= 5 : 500
a /= 5 : 100.0
a %= 3 : 1.0
a **= 2 : 1.0
a //= 3 : 0.0

Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in the binary format their values will be 0011 1100 and 0000 1101 respectively.

Following table lists out the bitwise operators supported by Python language with an example each in those, we use the above two variables (a and b) as operands -

a = 0011 1100
b = 0000 1101
---------- a&b = 12 (0000 1100)
a|b = 61 (0011 1101)
a^b = 49 (0011 0001)
~a = -61 (1100 0011)
a << 2 = 240 (1111 0000)
a>>2 = 15 (0000 1111)

There are following Bitwise operators supported by Python language

Operator	Name	Example
&	Binary AND	Sets each bit to 1 if both bits are 1
|	Binary OR	Sets each bit to 1 if one of two bits is 1
^	Binary XOR	Sets each bit to 1 if only one of two bits is 1
~	Binary Ones Complement	Inverts all the bits
<<	Binary Left Shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Binary Right Shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Example

```
a = 60 # 60 = 0011 1100
b = 13 # 13 = 0000 1101
# Binary AND
c = a & b # 12 = 0000 1100
print ("a & b : ", c)
# Binary OR
c = a | b # 61 = 0011 1101
 print ("a | b : ", c)
# Binary XOR
c = a ^ b # 49 = 0011 0001
print ("a ^ b : ", c)
# Binary Ones Complement

c = ~a; # -61 = 1100 0011
print ("~a : ", c)
# Binary Left Shift
c = a << 2; # 240 = 1111 0000 print ("a << 2 : ", c)
# Binary Right Shift
```

c = a >> 2; # 15 = 0000 1111 print ("a >> 2 : ", c)

Output

a & b : 12
a | b : 61
a ^ b : 49
~a : -61
a >> 2 : 240
a >> 2 : 15

Logical Operators

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then

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

Example

a = 1
b = 6
print((a > 2) and (b >= 6)) print((a>2) or (b>=6)) print(not a)

Output

False
True
False

Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below

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.

Example

x = 24
y = 20
list = [10, 20, 30, 40, 50]

```
if (x not in list):
print("x is NOT present in given list") else:
print("x is present in given list")
if (y in list):
print("y is present in given list") else:
print("y is NOT present in given list")
```

Output

```
x is NOT present in given list
y is present in given list
```

Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators explained below

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).

Example

```
x = 5
y = 5 print(x is y) id(x)
id(y)
```

Output

```
True
```

PYTHON OPERATORS PRECEDENCE

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

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Ccomplement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction

>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Operator precedence affects the evaluation of an an expression.

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because the operator * has higher precedence than +, so it first multiplies 3*2 and then is added to 7.

Associativity of Python Operators

We can see in the above table that more than one operator exists in the same group. These operators have the same precedence.

When two operators have the same precedence, associativity helps to determine the order of operations.

Associativity is the order in which an expression is evaluated that has multiple operators of the same precedence. Almost all the operators have left-to-right associativity.

For example, multiplication and floor division have the same precedence. Hence, if both of them are present in an expression, the left one is evaluated first.

```
Left-right associativity
# Output: 3
print(5 * 2 // 3)

# Shows left-right associativity
# Output: 0
print(5 * (2 // 3))
```

Output

```
3
0
```

Ex:

```
# This shows Left to right associativity
print(4 * 9 // 3)
# Using parenthesis this time to show left to right associativity
print(4 * (9 // 3))
```

Output:

```
12
12
```

Expression

It is a combination of operators and operands that is interpreted to produce some other value.

In any programming language, an expression is evaluated as per the precedence of its operators.

So that if there is more than one operator in an expression, their precedence decides which operation will be performed first.

We have many different types of expressions in Python. Let's discuss all types along with some exemplar codes :

```
# Constant Expressions

x = 15 + 1.3

print(x)
```

Output

16.3

2. **Arithmetic Expressions:**
   An arithmetic expression is a combination of numeric values, operators, and sometimes parenthesis.
   The result of this type of expression is also a numeric value.
   The operators used in these expressions are arithmetic operators like addition, subtraction, etc. Here are some arithmetic operators in Python:

Operators	**Syntax**	**Functioning**
+	x + y	Addition
–	x – y	Subtraction
*	x * y	Multiplication
/	x / y	Division
//	x // y	Quotient
%	x % y	Remainder
**	x ** y	Exponentiation

Example:

Let's see an exemplar code of arithmetic expressions in Python :

- Python3

```
# Arithmetic Expressions

x = 40

y = 12

 add = x + y
```

```
sub = x - y

pro = x * y

div = x / y

  print(add)

print(sub)

print(pro)

print(div)
```

Output

52

28

480

3.3333333333333335

3. Integral Expressions: These are the kind of expressions that produce only integer results after all computations and type conversions.

Example:

- Python3

```
# Integral Expressions

a = 13

b = 12.0

  c = a + int(b)

print(c)
```

Output

25

4. Floating Expressions: These are the kind of expressions which produce floating point numbers as result after all computations and type conversions.

Example:

- Python3

```
# Floating Expressions

a = 13

b = 5

 c = a / b

print(c)
```

Output

2.6

5. **Relational Expressions:**
   In these types of expressions, arithmetic expressions are written on both sides of relational operator (> , < , >= , <=).
   Those arithmetic expressions are evaluated first, and then compared as per relational operator and produce a boolean output in the end.
   These expressions are also called Boolean expressions.

Example:

- Python3

```
# Relational Expressions

a = 21

b = 13

c = 40

d = 37
```

```
p = (a + b) >= (c - d)

print(p)
```

Output

True

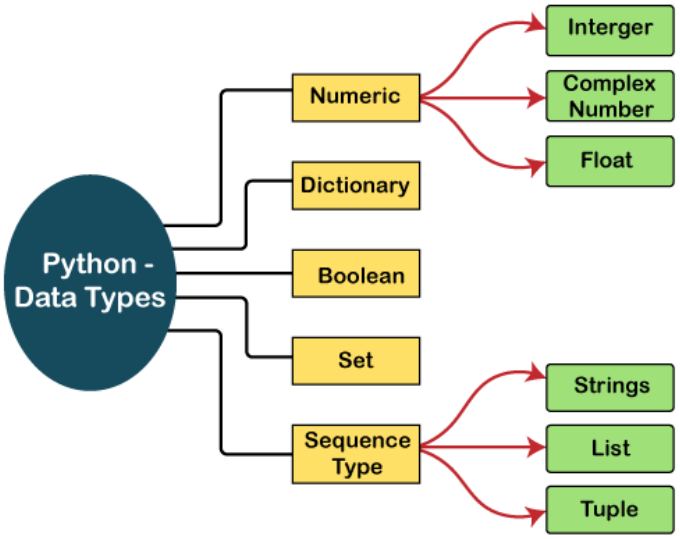
Data types

Standard data types

A variable can contain a variety of values. On the other hand, a person's id must be stored as an integer, while their name must be stored as a string.

The storage method for each of the standard data types that Python provides is specified by Python. The following is a list of the Python-defined data types.

1. Numbers
2. Sequence Type
3. Boolean
4. Set
5. Dictionary



Numbers

Numeric values are stored in numbers. The whole number, float, and complex qualities have a place with a Python Numbers datatype. Python offers the type() function to determine a variable's data type. The instance () capability is utilized to check whether an item has a place with a specific class.

```
a = 5
print("The type of a", type(a))
```

```
b = 40.5
print("The type of b", type(b))
c = 1+3j
print("The type of c", type(c))
print(" c is a complex number", isinstance(1+3j,complex))
```

Output:

```
The type of a <class 'int'>
The type of b <class 'float'>
The type of c <class 'complex'>
c is complex number: True
```

- **Int:** Whole number worth can be any length, like numbers 10, 2, 29, - 20, - 150, and so on. An integer can be any length you want in Python. Its worth has a place with int.
- **Float:** Float stores drifting point numbers like 1.9, 9.902, 15.2, etc. It can be accurate to within 15 decimal places.
- **Complex:** An intricate number contains an arranged pair, i.e., x + iy, where x and y signify the genuine and non-existent parts separately. The complex numbers like 2.14j, 2.0 + 2.3j, etc.

Sequence Type

String

The sequence of characters in the quotation marks can be used to describe the string. A string can be defined in Python using single, double, or triple quotes.

String dealing with Python is a direct undertaking since Python gives worked-in capabilities and administrators to perform tasks in the string.

```
str = "string using double quotes"
print(str)
s = """A multiline
string'''
print(s)
```

Output:

```
string using double quotes
A multiline
string
```

Tuple

In many ways, a tuple is like a list. Tuples, like lists, also contain a collection of items from various data types. A parenthetical space () separates the tuple's components from one another.

```
tup  = ("hi", "Python", 2)
# Checking type of tup
print (type(tup))
```

```
#Printing the tuple
print (tup)
# Tuple slicing
print (tup[1:])
print (tup[0:1])
# Tuple concatenation using + operator
print (tup + tup)
# Tuple repatation using * operator
print (tup * 3)
# Adding value to tup. It will throw an error.
t[2] = "hi"
```

Output:

```
<class 'tuple'>
('hi', 'Python', 2)
('Python', 2)
('hi',)
('hi', 'Python', 2, 'hi', 'Python', 2)
('hi', 'Python', 2, 'hi', 'Python', 2, 'hi', 'Python', 2)

Traceback (most recent call last):
  File "main.py", line 14, in <module>
    t[2] = "hi";
TypeError: 'tuple' object does not support item assignment
```

Dictionary

A dictionary is a key-value pair set arranged in any order. It stores a specific value for each key, like an associative array or a hash table. Value is any Python object, while the key can hold any primitive data type.

The comma (,) and the curly braces are used to separate the items in the dictionary.

Look at the following example.

```
d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'}
# Printing dictionary
print (d)
# Accesing value using keys
print("1st name is "+d[1])
print("2nd name is "+ d[4])
print (d.keys())
print (d.values())
```

Output:

```
1st name is Jimmy
2nd name is mike
{1: 'Jimmy', 2: 'Alex', 3: 'john', 4: 'mike'}
dict_keys([1, 2, 3, 4])
dict_values(['Jimmy', 'Alex', 'john', 'mike'])
```

Boolean

True and False are the two default values for the Boolean type. These qualities are utilized to decide the given assertion valid or misleading. The class book indicates this. False

can be represented by the 0 or the letter "F," while true can be represented by any value that is not zero.

Look at the following example.

```
# Python program to check the boolean type
print(type(True))
print(type(False))
print(false)
```

Output:

```
<class 'bool'>
<class 'bool'>
NameError: name 'false' is not defined
```

Set

The data type's unordered collection is Python Set. It is iterable, mutable(can change after creation), and has remarkable components.

The elements of a set have no set order; It might return the element's altered sequence. Either a sequence of elements is passed through the curly braces and separated by a comma to create the set or the built-in function set() is used to create the set.

It can contain different kinds of values.

Look at the following example.

```
# Creating Empty set
set1 = set()
set2 = {'James', 2, 3,'Python'}
#Printing Set value
print(set2)
# Adding element to the set
set2.add(10)
print(set2)
#Removing element from the set
set2.remove(2)
print(set2)
```

Output:

```
{3, 'Python', 'James', 2}
{'Python', 'James', 3, 2, 10}
{'Python', 'James', 3, 10}
```

Displaying age in seconds using Python3