

Practical File
DATA MINING AND ANALYSIS
B.E.(CSE) 6th Semester



Submitted to:
Prof. Sarabjeet Singh

Submitted by:
Amanpreet Kaur
UE223117
CSE Section-1

DEPARTMENT OF COMPUTER SCIENCE

University Institute of Engineering & Technology
Panjab University, Chandigarh

Index

PRACTICAL -1

Objective: Overview of Python, Basic Syntax, Variable Types, Basic Operators, Numbers

PRACTICAL -1

Objective: Overview of Python, Basic Syntax, Variable Types, Basic Operators, Numbers

OVERVIEW OF PYTHON

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

- **Python is Interpreted** – Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive** – You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** – Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language** – Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Python is an open-source and cross-platform programming language. It is available for use under **Python Software Foundation License** (compatible to GNU General Public License) on all the major operating system platforms Linux, Windows and Mac OS.

To facilitate new features and to maintain that readability, the Python Enhancement Proposal (PEP) process was developed. This process allows anyone to submit a PEP for a new feature, library, or other addition.

The design philosophy of Python emphasizes on simplicity, readability and unambiguity. Python is known for its batteries included approach as Python software is distributed with a comprehensive standard library of functions and modules.

Python's design philosophy is documented in the **Zen of Python**. It consists of nineteen aphorisms such as –

- Beautiful is better than ugly
- Explicit is better than implicit
- Simple is better than complex
- Complex is better than complicated

Pythonic Code Style

Python leaves you free to choose to program in an object-oriented, procedural, functional, aspect-oriented, or even logic-oriented way. These freedoms make Python a great language to write clean and beautiful code.

Pythonic Code Style is actually more of a design philosophy and suggests to write a code which is:

- Clean
- Simple
- Beautiful
- Explicit
- Readable

BASIC SYNTAX

The Python syntax defines a set of rules that are used to create a Python Program. The Python Programming Language Syntax has many similarities to Perl, C, and Java Programming Languages. However, there are some definite differences between the languages.

Python has two primary modes of execution: **Interactive Mode** and **Script Mode**.

1. Interactive Mode

- In **Interactive Mode**, Python executes commands one at a time.
- You can enter this mode by opening a Python shell or using an IDE's interactive console.
- Suitable for quick testing, debugging, and learning.
- Starts by running python or python3 in the terminal.

```
C:\Users\AMANPREET KAUR>python
Python 3.11.9 (main, Apr 12 2024, 09:55:27) [GCC UCRT 13.2.0 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> x=5
>>> y=2
>>> print(x+y)
7
>>> |
```

2. Script Mode

- In **Script Mode**, Python executes a pre-written script saved in a .py file.
- You write the entire program in a file and then execute it.
- Suitable for writing complex programs and automating tasks.

```
C: > Users > AMANPREET KAUR > OneDrive > Desktop > mining > prog.py > [x] x
1  x=2
2  y=5
3  print(x+y)
```

```
C:\Users\AMANPREET KAUR\OneDrive\Desktop\mining>python prog.py
7

C:\Users\AMANPREET KAUR\OneDrive\Desktop\mining>|
```

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, \$, and % within identifiers.

Here are naming conventions for Python identifiers –

- Python Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is **private** identifier.
- Starting an identifier with two leading underscores indicates a strongly **private** identifier.
- If the identifier also ends with two trailing underscores, the identifier is a **language-defined** special name.

Python Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

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

return	True	try
while	with	yield

Python Lines and Indentation

Python programming provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by **line indentation**, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example –

```
if True:
    print("True")
else:
    print('False')
```

True

Python Multi-Line Statements

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example –

```
> <
days=['monday','tuesday','wednesday',
      'Thursday', 'Friday']

for i in days:
    print(i)
```

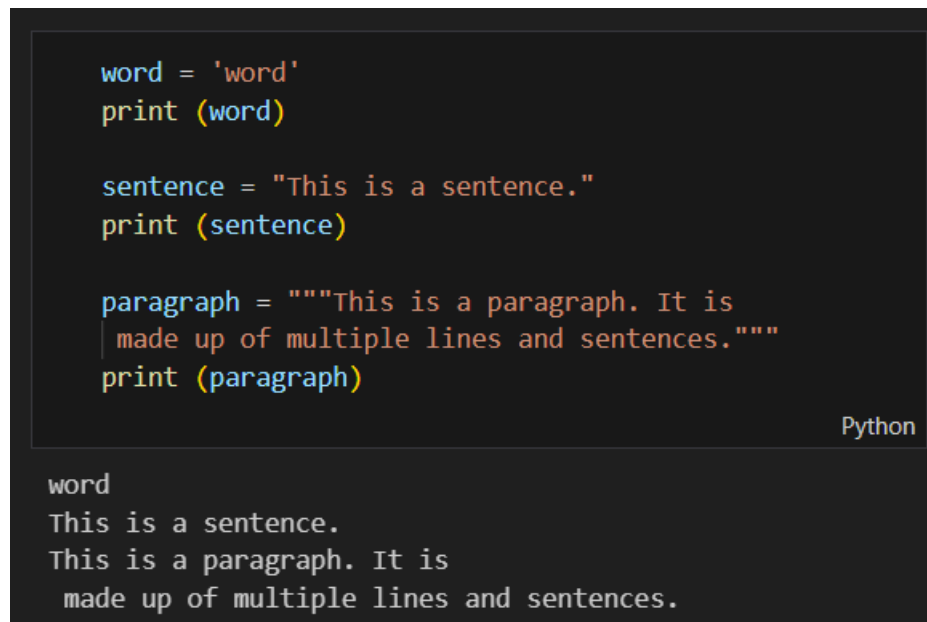
[4]

```
... monday
    tuesday
    wednesday
    Thursday
    Friday
```

Quotations in Python

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal –



```
word = 'word'
print (word)

sentence = "This is a sentence."
print (sentence)

paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
print (paragraph)
```

Python

word
This is a sentence.
This is a paragraph. It is
made up of multiple lines and sentences.

Comments in Python

A comment is a programmer-readable explanation or annotation in the Python source code. They are added with the purpose of making the source code easier for humans to understand, and are ignored by Python interpreter

Just like most modern languages, Python supports single-line (or end-of-line) and multi-line (block) comments. Python comments are very much similar to the comments available in PHP, BASH and Perl Programming languages.

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

```
#Comment
print("Hello World")

name = "Aman" # This is again comment

'''Multiline
comments'''

Python

Hello World
```

Using Blank Lines in Python Programs

A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it.

In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.

Multiple Statements on a Single Line

The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon –

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')

[8] Python
.. foo
.. 4
```

Multiple Statement Groups as Suites

A group of individual statements, which make a single code block are called **suites** in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite. For example –

```
if expression :  
| suite  
elif expression :  
| suite  
else :  
| suite
```

PYTHON VARIABLES

Python variables are the reserved memory locations used to store values with in a Python Program. This means that when you create a variable you reserve some space in the memory. Based on the data type of a variable, Python interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to Python variables, you can store integers, decimals or characters in these variables.

Memory Addresses

Data items belonging to different data types are stored in computer's memory. Computer's memory locations are having a number or address, internally represented in binary form. Data is also stored in binary form as the computer works on the principle of binary representation. In the following diagram, a string **May** and a number **18** is shown as stored in memory locations.

```
"May"
id("May")

2470345714656

18
id(18)

140735105469400
```

Creating Python Variables

Python variables do not need explicit declaration to reserve memory space or you can say to create a variable. A Python variable is created automatically when you assign a value to it. The equal sign (=) is used to assign values to variables.

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

Example to Create Python Variables

This example creates different types (an integer, a float, and a string) of variables.

```
▶ counter = 100          # Creates an integer variable
  miles  = 1000.0        # Creates a floating point variable
  name   = "Aman"        # Creates a string variable

#printing variables
print (counter)
print (miles)
print (name)

[1] ✓ 0.0s

... 100
    1000.0
    Aman
```

Deleting Python Variables

You can delete the reference to a number object by using the del statement. The syntax of the del statement is –

```
var=10
print(var)
del var
print(var)
```

[28]

... 10

... -----

NameError Traceback (most recent call last)

Cell In[28], line 4

2 print(var)

3 del var

----> 4 print(var)

NameError: name 'var' is not defined

Getting Type of a Variable

You can get the data type of a Python variable using the python built-in function type() as follows.

Example: Printing Variables Type

```
x = "Aman"
y = 10
z = 10.10

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

[2] ✓ 0.0s

... <class 'str'>

<class 'int'>

<class 'float'>

Casting Python Variables

You can specify the data type of a variable with the help of casting as follows:

Example

This example demonstrates case sensitivity of variables.

```
x = str(10)    # x will be '10'
y = int(10)    # y will be 10
z = float(10)  # z will be 10.0

print( "x =", x )
print( "y =", y )
print( "z =", z )

0]
· x = 10
  y = 10
  z = 10.0
```

Case-Sensitivity of Python Variables

Python variables are case sensitive which means **Age** and **age** are two different variables:

```
age = 20
Age = 30

print( "age =", age )
print( "Age =", Age )

31]
·· age = 20
   Age = 30
```

Python Variables - Multiple Assignment

Python allows to initialize more than one variables in a single statement. In the following case, three variables have same value.

```
32] a=10
    b=10
    c=10
    print(a,b,c)
.. 10 10 10

33] a,b,c = 10,20,30
    print (a,b,c)
.. 10 20 30

34] a = b = c = 100
    print (a)
    print (b)
    print (c)
.. 100
   100
   100
```

Python Variables - Naming Convention

Every Python variable should have a unique name like a, b, c. A variable name can be meaningful like color, age, name etc. There are certain rules which should be taken care while naming a Python variable:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number or any special character like \$, (, * % etc.
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Python variable names are case-sensitive which means Name and NAME are two different variables in Python.
- Python reserved keywords cannot be used naming the variable.

If the name of variable contains multiple words, we should use these naming patterns

—

- **Camel case** – First letter is a lowercase, but first letter of each subsequent word is in uppercase. For example: kmPerHour, pricePerLitre
- **Pascal case** – First letter of each word is in uppercase. For example: KmPerHour, PricePerLitre
- **Snake case** – Use single underscore (_) character to separate words. For example: km_per_hour, price_per_litre

```

counter = 100
_count = 100
name1 = "Zara"
name2 = "Nuha"
Age = 20
zara_salary = 100000

print (counter)
print (_count)
print (name1)
print (name2)
print (Age)
print (zara_salary)

```

100
100
Zara
Nuha
20
100000

Invalid Naming Conventions

```

1counter = 100
$count = 100
zara-salary = 100000

print (1counter)
print ($count)
print (zara-salary)

```

Cell In[38], line 1
1counter = 100
^
SyntaxError: invalid decimal literal

Python Local Variables

Python Local Variables are defined inside a function. We can not access variable outside the function.

A Python functions is a piece of reusable code and you will learn more about function in Python - Functions tutorial.

```
def sum(x,y):  
    sum = x + y  
    return sum  
print(sum(5, 10))
```

15

Python Global Variables

Any variable created outside a function can be accessed within any function and so they have global scope.

Example

Following is an example of global variables –

```
x = 5  
y = 10  
def sum():  
    sum = x + y  
    return sum  
print(sum())
```

15

Python vs C/C++ Variables

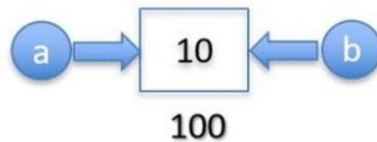
The concept of variable works differently in Python than in C/C++. In C/C++, a variable is a named memory location. If a=10 and also b=10, both are two different memory locations. Let us assume their memory address is 100 and 200 respectively.



If a different value is assigned to "a" - say 50, 10 in the address 100 is overwritten.



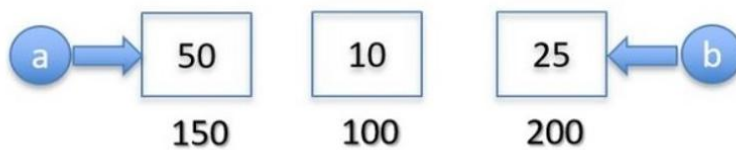
A Python variable refers to the object and not the memory location. An object is stored in memory only once. Multiple variables are really the multiple labels to the same object.



The statement `a=50` creates a new **int** object 50 in the memory at some other location, leaving the object 10 referred by "b".



Further, if you assign some other value to b, the object 10 remains unreferred.




```
[41] a=b=10
      a is b
...  True

[42] id(a),id(b)
...  (140735105469144, 140735105469144)
```

Python Operators

Python operators are special symbols used to perform specific operations on one or more operands. The variables, values, or expressions can be used as operands. For example, Python's addition operator (+) is used to perform addition operations on two variables, values, or expressions.

The following are some of the terms related to **Python operators**:

- **Unary operators:** Python operators that require one operand to perform a specific operation are known as unary operators.
- **Binary operators:** Python operators that require two operands to perform a specific operation are known as binary operators.
- **Operands:** Variables, values, or expressions that are used with the operator to perform a specific operation.

Python Arithmetic Operators

Python Arithmetic operators are used to perform basic mathematical operations such as addition, subtraction, multiplication, etc.

The following table contains all arithmetic operators with their symbols, names, and examples (assume that the values of **a** and **b** are 10 and 20, respectively) –

Operator	Name	Example
+	Addition	$a + b = 30$
-	Subtraction	$a - b = -10$
*	Multiplication	$a * b = 200$
/	Division	$b / a = 2$
%	Modulus	$b \% a = 0$
**	Exponent	$a ** b = 10 ** 20$
//	Floor Division	$9 // 2 = 4$

Example of Python Arithmetic Operators

```

a=21
b=10
c=0
print()
print("Addition")
c=a+b
print("a:{} b:{} a+b:{}".format(a,b,c))

print()
print("Subtraction")
c = a - b
print ("a: {} b: {} a-b: {}".format(a,b,c) )

print()
print("Multiplication")
c = a * b
print ("a: {} b: {} a*b: {}".format(a,b,c))

print()
print("Division")
c = a / b
print ("a: {} b: {} a/b: {}".format(a,b,c))

print()
print("Modulus")
c = a % b
print ("a: {} b: {} a%b: {}".format(a,b,c))

```

```

print()
print("Exponent")
a = 2
b = 3
c = a**b
print ("a: {} b: {} a**b: {}".format(a,b,c))

print()
print("Floor Division")
a = 10
b = 5
c = a//b
print ("a: {} b: {} a//b: {}".format(a,b,c))

```

```

Addition
a:21 b:10 a+b:31

Subtraction
a: 21 b: 10 a-b: 11

Multiplication
a: 21 b: 10 a*b: 210

Division
a: 21 b: 10 a/b: 2.1

Modulus
a: 21 b: 10 a%b: 1

Exponent
a: 2 b: 3 a**b: 8

Floor Division
a: 10 b: 5 a//b: 2

```

Python Comparison Operators

Python Comparison operators compare the values on either side of them and decide the relation among them. They are also called Relational operators.

The following table contains all comparison operators with their symbols, names, and examples (assume that the values of **a** and **b** are 10 and 20, respectively) –

Operator	Name	Example
==	Equal	(a == b) is not true.
!=	Not equal	(a != b) is true.
>	Greater than	(a > b) is not true.
<	Less than	(a < b) is true.
>=	Greater than or equal to	(a >= b) is not true.
<=	Less than or equal to	(a <= b) is true.

Example of Python Comparison Operators

```

a = 21
b = 10
if ( a == b ):
    print ("Line 1 - a is equal to b")
else:
    print ("Line 1 - a is not equal to b")

if ( a != b ):
    print ("Line 2 - a is not equal to b")
else:
    print ("Line 2 - a is equal to b")

if ( a < b ):
    print ("Line 3 - a is less than b" )
else:
    print ("Line 3 - a is not less than b")

if ( a > b ):
    print ("Line 4 - a is greater than b")
else:
    print ("Line 4 - a is not greater than b")

a,b=b,a #values of a and b swapped. a becomes 10, b becomes 21

if ( a <= b ):
    print ("Line 5 - a is either less than or equal to b")
else:
    print ("Line 5 - a is neither less than nor equal to b")

if ( b >= a ):
    print ("Line 6 - b is either greater than or equal to b")
else:
    print ("Line 6 - b is neither greater than nor equal to b")

```

```
... Line 1 - a is not equal to b
Line 2 - a is not equal to b
Line 3 - a is not less than b
Line 4 - a is greater than b
Line 5 - a is either less than or equal to b
Line 6 - b is either greater than or equal to b
```

Python Assignment Operators

Python Assignment operators are used to assign values to variables. Following is a table which shows all Python assignment operators.

The following table contains all assignment operators with their symbols, names, and examples –

Operator	Example	Same As
=	a = 10	a = 10
+=	a += 30	a = a + 30
-=	a -= 15	a = a - 15
*=	a *= 10	a = a * 10
/=	a /= 5	a = a / 5
%=	a %= 5	a = a % 5
**=	a **= 4	a = a ** 4
//=	a //= 5	a = a // 5
&=	a &= 5	a = a & 5
=	a = 5	a = a 5
^=	a ^= 5	a = a ^ 5
>>=	a >>= 5	a = a >> 5
<<=	a <<= 5	a = a << 5

Example of Python Assignment Operators

```

a = 21
b = 10
c = 0
print ("a: {} b: {} c : {}".format(a,b,c))
c = a + b
print ("a: {} c = a + b: {}".format(a,c))

c += a
print ("a: {} c += a: {}".format(a,c))

c *= a
print ("a: {} c *= a: {}".format(a,c))

c /= a
print ("a: {} c /= a : {}".format(a,c))

c = 2
print ("a: {} b: {} c : {}".format(a,b,c))
c %= a
print ("a: {} c %= a: {}".format(a,c))

c **= a
print ("a: {} c **= a: {}".format(a,c))

c //= a
print ("a: {} c //= a: {}".format(a,c))

```

Python Bitwise Operators

Python Bitwise operator works on bits and performs bit by bit operation. These operators are used to compare binary numbers.

Operator	Name	Example
&	AND	a & b
	OR	a b
^	XOR	a ^ b
~	NOT	~a
<<	Zero fill left shift	a << 3
>>	Signed right shift	a >> 3

The following table contains all bitwise operators with their symbols, names, and examples –

Example of Python Bitwise Operators

```
a = 20
b = 10

print ('a=',a,': ',bin(a), 'b=',b,': ',bin(b))
c = 0
```

```
a= 20 : 0b10100 b= 10 : 0b1010
```

```
c = a & b;
print ("result of AND is ", c,': ',bin(c))
```

```
result of AND is  0 : 0b0
```

```
c = a | b;
print ("result of OR is ", c,': ',bin(c))
```

```
result of OR is  30 : 0b11110
```

```
c = a ^ b;
print ("result of EXOR is ", c,': ',bin(c))
```

```
result of EXOR is  30 : 0b11110
```

```
✓
c = ~a;
print ("result of COMPLEMENT is ", c,':',bin(c))

result of COMPLEMENT is  -21 : -0b10101

c = a << 2;
print ("result of LEFT SHIFT is ", c,':',bin(c))

result of LEFT SHIFT is  80 : 0b1010000

c = a >> 2;
print ("result of RIGHT SHIFT is ", c,':',bin(c))

result of RIGHT SHIFT is  5 : 0b101
```

Python Logical Operators

Python logical operators are used to combine two or more conditions and check the final result. There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then

The following table contains all logical operators with their symbols, names, and examples –

Operator	Name	Example
and	AND	a and b
or	OR	a or b
not	NOT	not(a)


```
var = 5

print(var > 3 and var < 10)
print(var > 3 or var < 4)
print(not (var > 3 and var < 10))

[58]
... True
    True
    False
```

Python 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	Returns True if it finds a variable in the specified sequence, false otherwise.	a in b
not in	returns True if it does not finds a variable in the specified sequence and false otherwise.	a not in b

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

print ("a:", a, "b:", b, "list:", list)

if ( a in list ):
|   print ("a is present in the given list")
else:
|   print ("a is not present in the given list")

if ( b not in list ):
|   print ("b is not present in the given list")
else:
|   print ("b is present in the given list")

c=b/a
print ("c:", c, "list:", list)
if ( c in list ):
|   print ("c is available in the given list")
else:
|   print ("c is not available in the given list")
```

59]

```
.. a: 10 b: 20 list: [1, 2, 3, 4, 5]
a is not present in the given list
b is not present in the given list
c: 2.0 list: [1, 2, 3, 4, 5]
c is available in the given list
```

Python Identity Operators

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

Operator	Description	Example
is	Returns True if both variables are the same object and false otherwise.	a is b
is not	Returns True if both variables are not the same object and false otherwise.	a is not b

Example of Python Identity Operators

```
a = [1, 2, 3, 4, 5]
b = [1, 2, 3, 4, 5]
c = a

print(a is c)
print(a is b)

print(a is not c)
print(a is not b)
```

True
False
False
True

Python Numbers

Python has built-in support to store and process numeric data (**Python Numbers**). Most of the times you work with numbers in almost every Python application. Obviously, any computer application deals with numbers. This tutorial will discuss about different types of Python Numbers and their properties.

Python - Number Types

There are three built-in number types available in Python:

- integers (**int**)
- floating point numbers (**float**)
- **complex** numbers

Python also has a built-in Boolean data type called **bool**. It can be treated as a sub-type of **int** type, since its two possible values **True** and **False** represent the integers 1 and 0 respectively.

Python – Integer Numbers

In Python, any number without the provision to store a fractional part is an integer. (Note that if the fractional part in a number is 0, it doesn't mean that it is an integer. For example a number 10.0 is not an integer, it is a float with 0 fractional part whose numeric value is 10.) An integer can be zero, positive or a negative whole number. For example, 1234, 0, -55 all represent to integers in Python.

There are three ways to form an integer object. With (a) literal representation, (b) any expression evaluating to an integer, and (c) using **int()** function.

Literal is a notation used to represent a constant directly in the source code. For example –

```
[1] a = 10
    b = 20
    c = a + b

    print ("a:", a, "type:", type(a))
    print ("c:", c, "type:", type(c))
... a: 10 type: <class 'int'>
    c: 30 type: <class 'int'>
```

Binary Numbers in Python

A number consisting of only the binary digits (1 and 0) and prefixed with "**0b**" is a binary number. If you assign a binary number to a variable, it still is an int variable.

```
[3] a=43
    b=bin(a)
    print ("Integer:",a, "Binary equivalent:",b)
... Integer: 43 Binary equivalent: 0b101011
```

Octal Numbers in Python

An octal number is made up of digits 0 to 7 only. In order to specify that the integer uses octal notation, it needs to be prefixed by "**0o**" (lowercase O) or "**0O**" (uppercase O). A literal representation of octal number is as follows –

```
▶ ~
a=00107
print (a, type(a))

[4]
... 71 <class 'int'>

a=int('20',8)
print (a, type(a))

[5]
... 16 <class 'int'>

a=0056
print ("a:",a, "type:",type(a))

b=int("0031",8)
print ("b:",b, "type:",type(b))

c=a+b
print ("addition:", c)

[6]
... a: 46 type: <class 'int'>
    b: 25 type: <class 'int'>
    addition: 71
```

Hexa-decimal Numbers in Python

As the name suggests, there are 16 symbols in the Hexadecimal number system. They are 0-9 and A to F. The first 10 digits are same as decimal digits. The alphabets A, B, C, D, E and F are equivalents of 11, 12, 13, 14, 15, and 16 respectively. Upper or lower cases may be used for these letter symbols.

For the literal representation of an integer in Hexadecimal notation, prefix it by **"0x"** or **"0X"**.

```
[7]: a=0xA2
     print (a, type(a))
... 162 <class 'int'>

[8]: a=int('0X1e', 16)
     print (a, type(a))
... 30 <class 'int'>
```

Python – Floating Point Numbers

A floating point number has an integer part and a fractional part, separated by a decimal point symbol (.). By default, the number is positive, prefix a dash (-) symbol for a negative number. A floating point number is an object of Python's float class. To store a float object, you may use a literal notation, use the value of an arithmetic expression, or use the return value of float() function.

Using literal is the most direct way. Just assign a number with fractional part to a variable. Each of the following statements declares a float object.

```
a=10.33
b=2.66
c=a/b

print ("c:", c, "type", type(c))

c: 3.8834586466165413 type <class 'float'>
```

Python – Complex Numbers

In this section, we shall know in detail about Complex data type in Python. Complex numbers find their applications in mathematical equations and laws in electromagnetism, electronics, optics, and quantum theory. Fourier transforms use complex numbers. They are Used in calculations with wavefunctions, designing filters, signal integrity in digital electronics, radio astronomy, etc.

A complex number consists of a real part and an imaginary part, separated by either "+" or "-". The real part can be any floating point (or itself a complex number) number. The imaginary part is also a float/complex, but multiplied by an imaginary number.

In mathematics, an imaginary number "i" is defined as the square root of -1 ($\sqrt{-1}$). Therefore, a complex number is represented as "x+yi", where x is the real part, and "y" is the coefficient of imaginary part.

Quite often, the symbol "j" is used instead of "I" for the imaginary number, to avoid confusion with its usage as current in theory of electricity. Python also uses "j" as the imaginary number. Hence, "x+yj" is the representation of complex number in Python.

Like int or float data type, a complex object can be formed with literal representation or using complex() function. All the following statements form a complex object.

```
a=complex(5.3,6)
b=complex(1.01E-2, 2.2E3)
print ("a:", a, "type:", type(a))
print ("b:", b, "type:", type(b))
```

```
a: (5.3+6j) type: <class 'complex'>
b: (0.0101+2200j) type: <class 'complex'>
```

```
a=complex(1+2j, 2-3j)
print (a, type(a))
```

```
a: (5.3+6j) type: <class 'complex'>
b: (0.0101+2200j) type: <class 'complex'>
```

```
a=complex(5.3)
print ("a:", a, "type:", type(a))
```

```
a: (5.3+0j) type: <class 'complex'>
```

```
a= "5.5+2.3j"
b=complex(a)
print ("Complex number:", b)
```

```
Complex number: (5.5+2.3j)
```