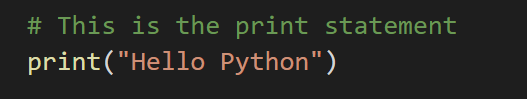
**Basic Fundamentals – Comments**

**Comments**

Python Comment is an essential tool for the programmers. Comments are generally used to explain the code. We can easily understand the code if it has a proper explanation. A good programmer must use the comments because in the future anyone wants to modify the code as well as implement the new module; then, it can be done easily.

In the other programming language such as C++, It provides the // for single-lined comment and /\*.... \*/ for multiple-lined comment, but Python provides the single-lined Python comment. To apply the comment in the code we use the hash(#) at the beginning of the statement or code.

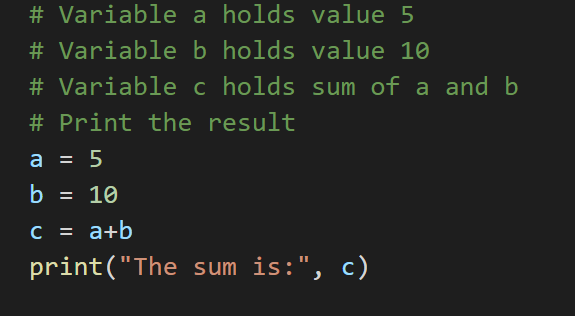
Let's understand the following example.



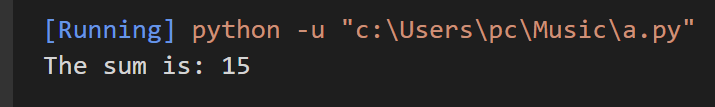
Multiline Python Comment

We must use the hash (#) at the beginning of every line of code to apply the multiline Python comment. Consider the following example.

**Example:**



**OUTPUT**



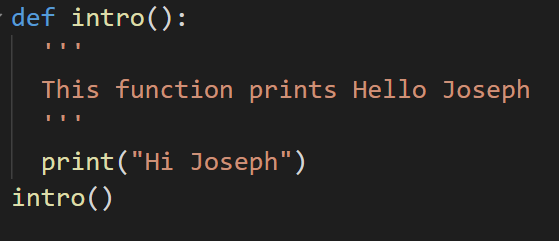
The above code is very readable even the absolute beginners can under that what is happening in each line of the code. This is the advantage of using comments in code.

We can also use the triple quotes ('''''') for multiline comment. The triple quotes are also used to string formatting. Consider the following example.

**Docstrings Python Comment**

The docstring comment is mostly used in the module, function, class or method. It is a documentation Python string.

**Example:**



**Operators**

Just like in mathematics, programming languages like Python have operators. Operators in python are used to perform operations between variables. The operator can be defined as a symbol which is responsible for a particular operation between two operands. Operators are the pillars of a program on which the logic is built in a specific programming language. Python provides a variety of operators.

**Types of Operators**

1. **Arithmetic Operators**

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

----------------------------------------------------------------------------------------------------------------

**+ (Addition)**

It is used to add two operands. For example, if a = 20, b = 10 => print(a+b) = 30

**- (Subtraction)**

It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value results negative. For example,

    a = 20, b = 10 =>print( a – b) = 10

**/ (divide)**

It returns the quotient after dividing the first operand by the second operand. Remember the result will be float.

           For example, if a = 20, b = 10 => print(a/b) = 2.0

**\* (Multiplication)**

It is used to multiply one operand with the other.

          For example, if a = 20, b = 10 => print(a \* b) = 200

**% (reminder)**

It returns the reminder after dividing the first operand by the second operand.

           For example, if a = 20, b = 10 => print(a%b) = 0

**\*\* (Exponent)**

It is an exponent operator represented as it calculates the first operand power to the second operand.

       For example, if a=2, b=3  ==>  print(a\*\*b) = 2\*\*3= 8

**// (Floor division)**

It gives the floor value of the quotient produced by dividing the two operands. Remember result will be int.

    For example, if a=3, b=2  ===> print(a//b) = 3//2 = 1

1. **Assignment Operators**

The assignment operators are used to assign the value of the right expression to the left operand.

**=**

It assigns the value of the right expression to the left operand.

**+=**

It increases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 10, b=20 => a+ =b will be equal to a = a+b and therefore a = 30

**-=**

It decreases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 20, b = 10 => a- = b will be equal to a = a- b and therefore, a = 10.

**\*=**

It multiplies right operand with the left operand and assign the result to left operand.

c \*= a is equivalent to c = c \* a

**%=**

It takes modulus using two operands and assign the result to left operand.

c %= a is equivalent to c = c % a

**\*\*=**

Performs exponential (power) calculation on operators and assign value to the left operand.

c \*\*= a is equivalent to c = c \*\* a

**//=**

It performs floor division on operators and assign value to the left operand.

c //= a is equivalent to c = c // a

1. **Comparison Operators**

Comparison operators are used to comparing the value of the two operands and returns Boolean true or false accordingly.

**== (is equal to)**

If the values of two operands are equal, then the condition becomes true.

**!= (is not equal to)**

If values of two operands are not equal, then condition becomes true.

**<= (is lesser than or equal to)**

If the value of left operand is less than or equal to the value of right operand, then condition becomes true.

**>= (is greater than or equal to)**

If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.

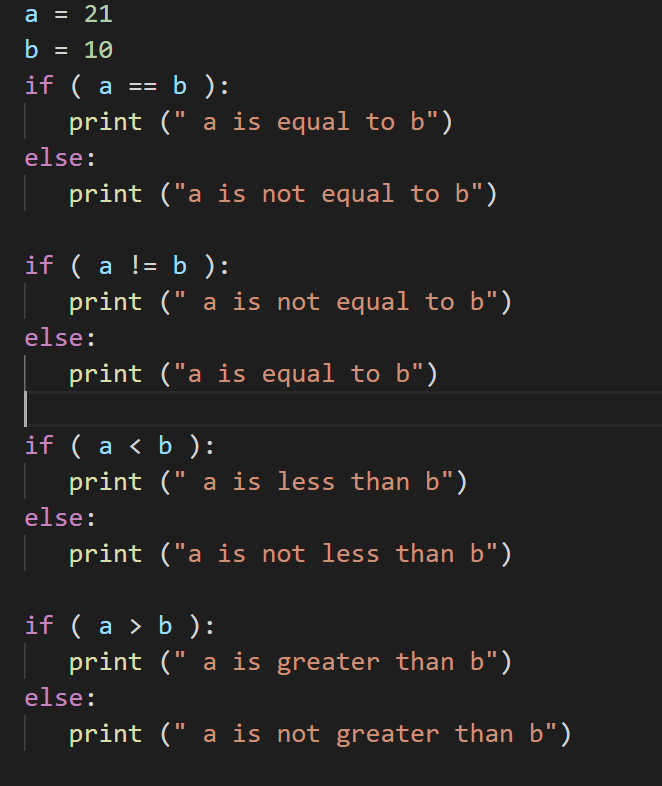
**>  ( is greater than)**

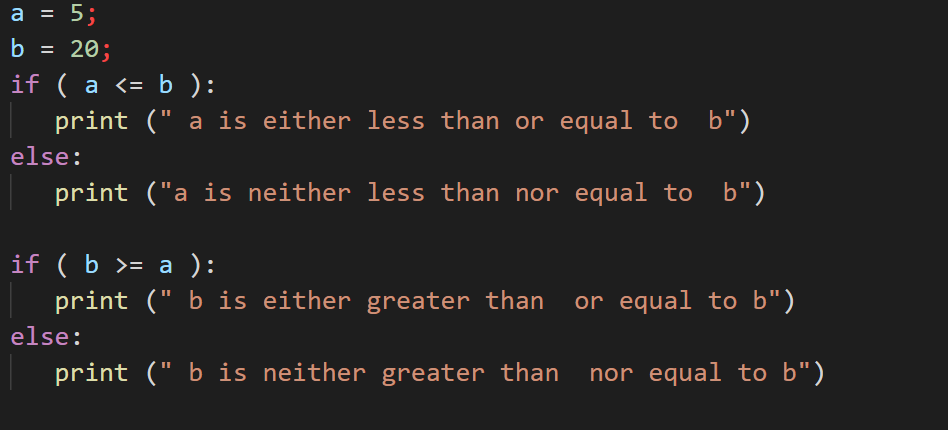
If the value of left operand is greater than the value of right operand, then condition becomes true.

**<  (is lesser than)**

If the value of left operand is less than the value of right operand, then condition becomes true.

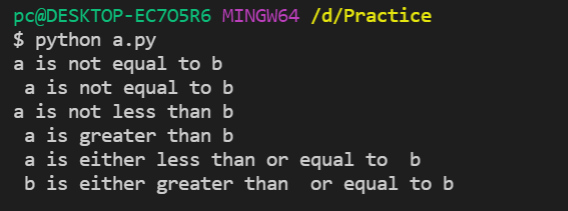
**Example:**





**OUTPUT**

When you execute the above program it produces the following result −

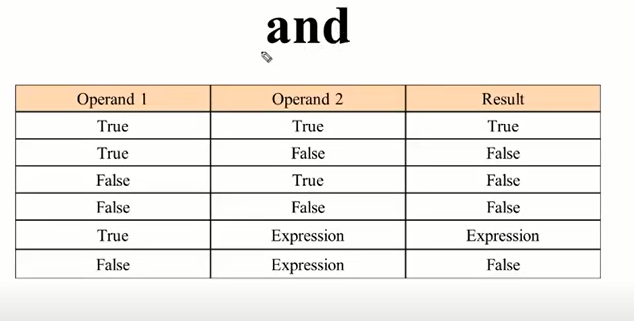


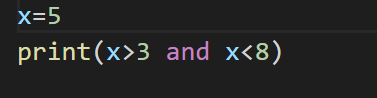
1. **Logical Operators**

The logical operators are used primarily in the expression evaluation to make a decision. Python supports the following logical operators.

and  - if both expression are true, then condition becomes true

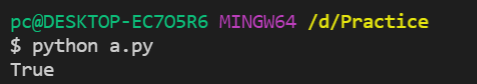
        a=true, b=true, a and b=true

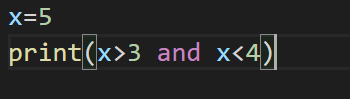




**OUTPUT**

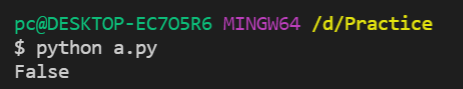
When you execute the above program it produces the following result –





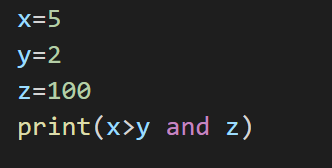
**OUTPUT**

When you execute the above program it produces the following result –

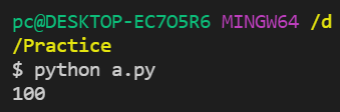


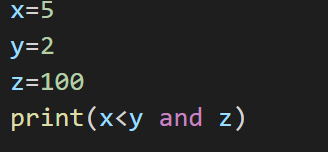
**Expression:**

 If you ask Python to print an expression, the interpreter evaluates the expression and displays the result.

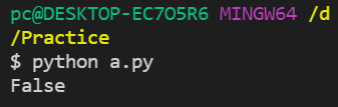


**OUTPUT**



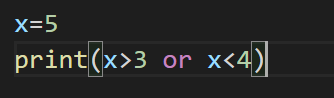


**OUTPUT**



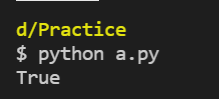
or - if one expression is true, then the condition becomes true.

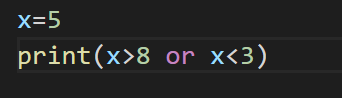
      a=true, b=false, a or b=true



**OUTPUT**

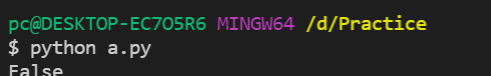
When you execute the above program it produces the following result −

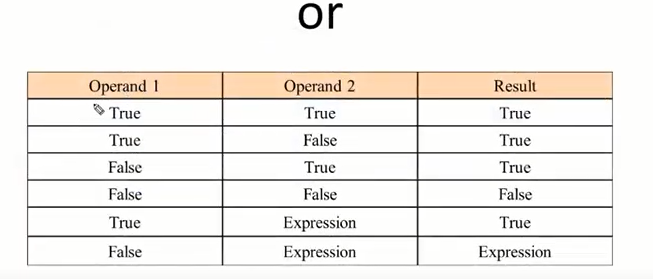


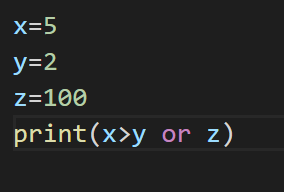


**OUTPUT**

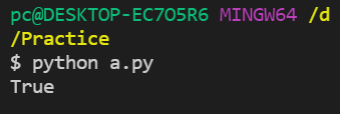
When you execute the above program it produces the following result −

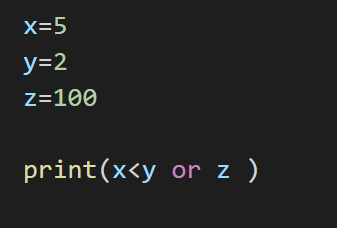




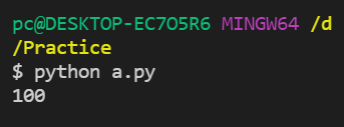


**OUTPUT**





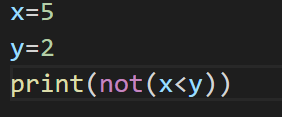
**OUTPUT**



not - if an expression a is true, then not(a) will be false and vice versa

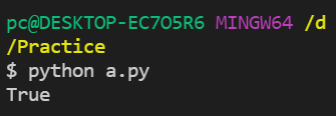
    a=true, nota=false

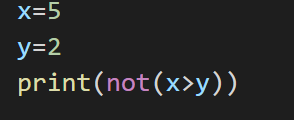




**OUTPUT**

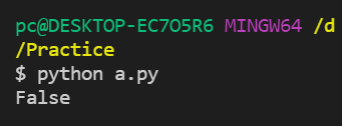
When you execute the above program it produces the following result −





**OUTPUT**

When you execute the above program it produces the following result −

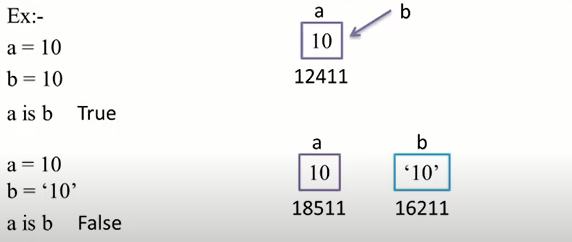


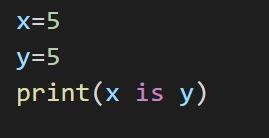
1. **Identity Operators**

The identity operators compare the memory locations of two objects. Hence, it is possible to know whether two objects are same or not. There are two types of identity operators:

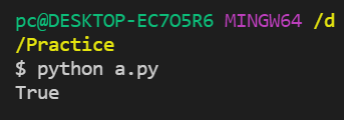
**is**

This operator is used to compare whether two objects are same or not. It is evaluated to be true if the reference present at both sides point to the same object.





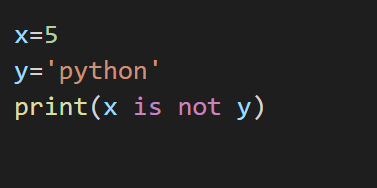
**OUTPUT**



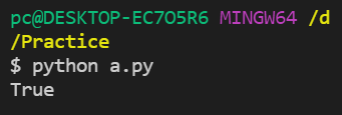
**is not**

This operator works in reverse manner for is operator. It is evaluated to be true if the reference present at both sides do not point to the same object.





**OUTPUT**



1. **Membership Operators**

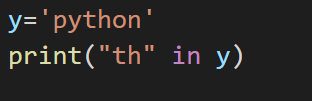
Python membership operators are used to check the membership of value inside a Python data structure. If the value is present in the data structure, then the resulting value is true otherwise it returns false. There are two types of Membership Operator:

**in**

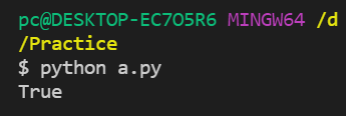
This operator is used to find an element in a specified sequence. It is evaluated to be true if element is found in the specified sequence else it returns false.

a = “Welcome to python”

“to” in a **True**

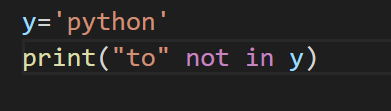


**OUTPUT**

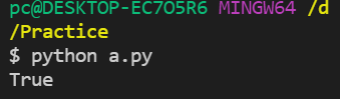


**not in**

This operator works in reverse manner for in operator. It is evaluated to be true if element is not found in the specified sequence else it returns false if element is found.

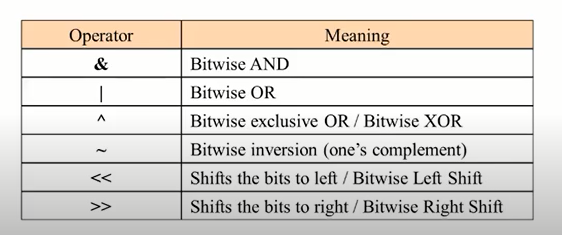


**OUTPUT**

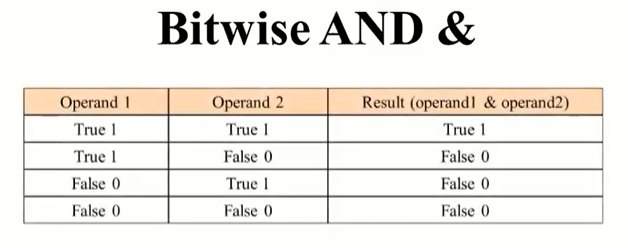


1. **Bitwise Operators**

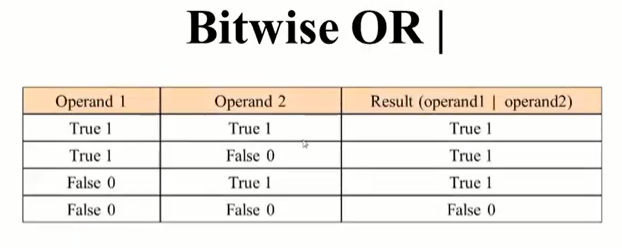
Bitwise operators are used to perform operations at binary digit level. The bitwise operators perform bit by bit operation on the values of the two operands.



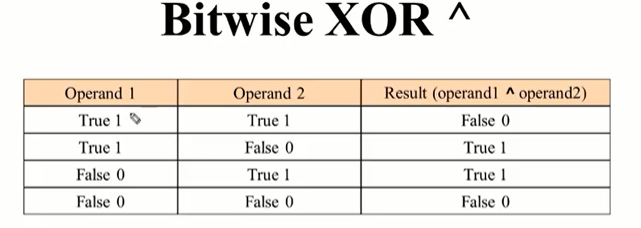
**& Bitwise AND**



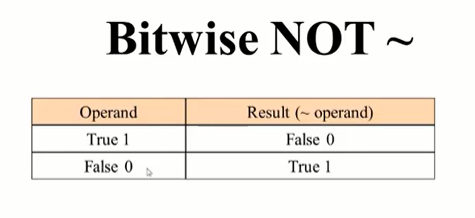
**| Bitwise OR**



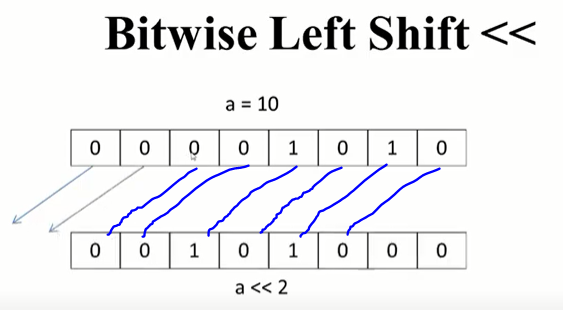
**^ Bitwise XOR**



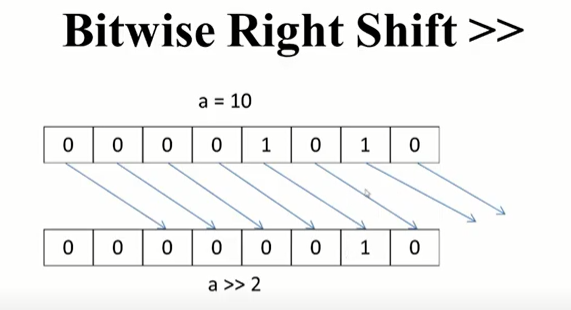
~ **Bitwise inversion**



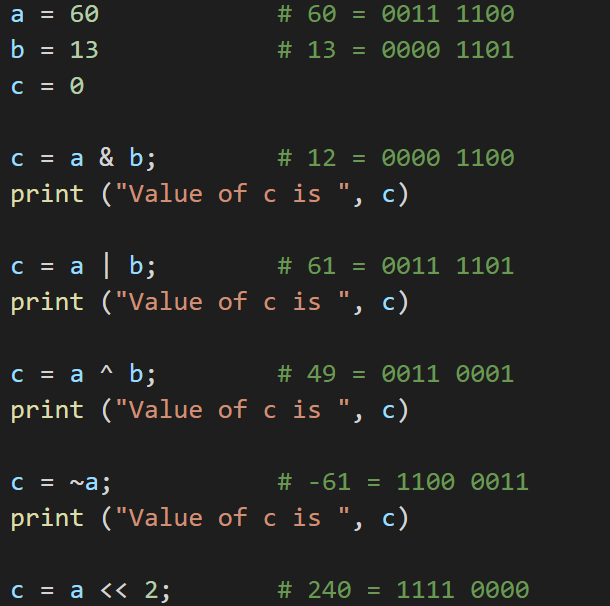
**<< Bitwise Left Shift**

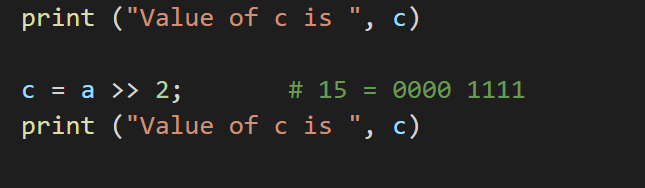


**>> Bitwise Right Shift**



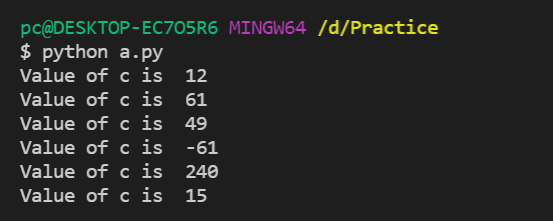
**Example:**





**OUTPUT**

When you execute the above program it produces the following result –



**Data Types**

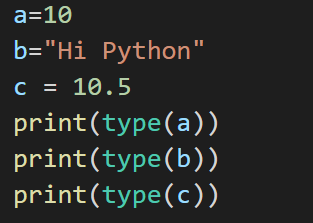
Datatype represents the type of data stored into a variable or memory. Variables can hold values, and every value has a data-type. Python is a dynamically typed language; hence we do not need to define the type of the variable while declaring it. The interpreter implicitly binds the value with its type.

a=50

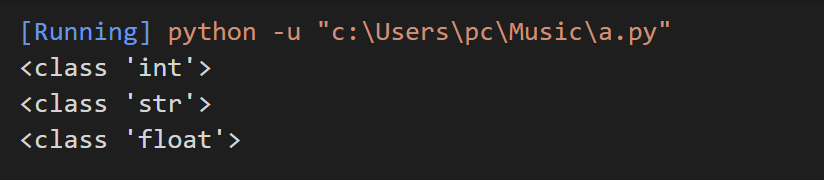
The variable **a** holds integer value five and we did not define its type. Python interpreter will automatically interpret variables **a** as an integer type.

Python enables us to check the type of the variable used in the program. Python provides us the **type()** function, which returns the type of the variable passed.

Consider the following example to define the values of different data types and checking its type.



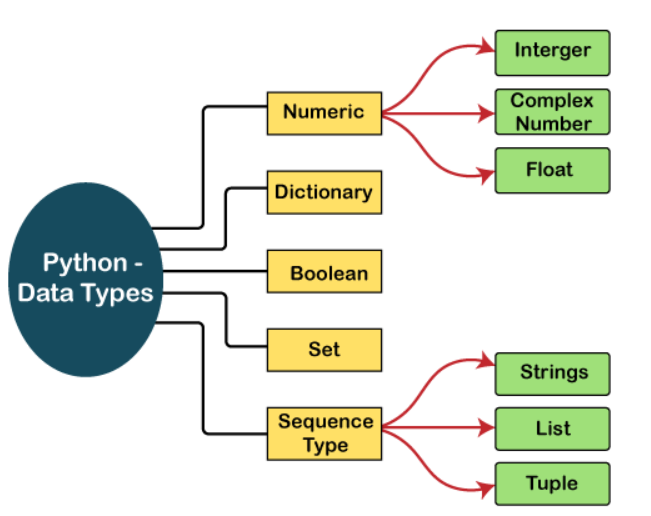
**OUTPUT**



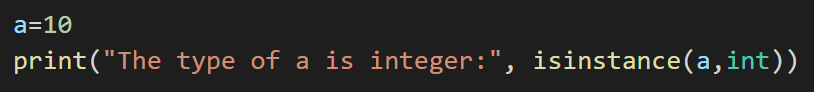
Standard data types

A variable can hold different types of values. For example, a person's name must be stored as a string whereas its id must be stored as an integer.

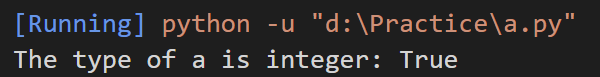
Python provides various standard data types that define the storage method on each of them. The data types defined in Python are given below.

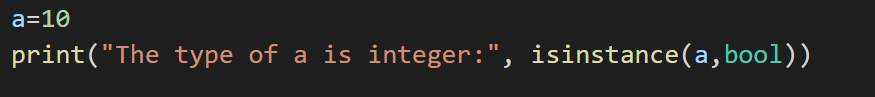


**1.Numeric:** Number stores numeric values. Python provides the **type()** function to know he datatype of the variable. Similarly, the **isinstance()** function is used to check an object belongs to a particular class.

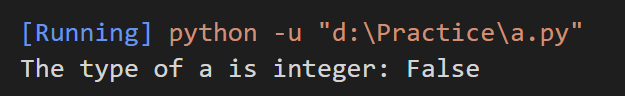


**OUTPUT**



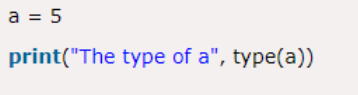


**OUTPUT**



**Following are the numeric Data type:**

**Integer:** The int datatype represents an integer number. An integer number without any decimal point or fraction part. In python, it is possible to store very large integer number as there is no limit for the size of an int datatype. **Example:** 20, 10, -50, -1002



**Complex Number:** A complex number is a number that is written in the form of a+bj or a+bj. Where,

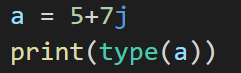
a = Real part of the number

b = Imaginary part of the number

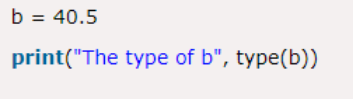
j or J = Square root value of -1

a and b may contain integer or float number

**Example:** 2+1j, 2+4j



**Float:** The float datatype represents floating point numbers. A floating-point number is a number that contains a decimal point. **Example:** 25.56, 10.5, -45.69, -0.8

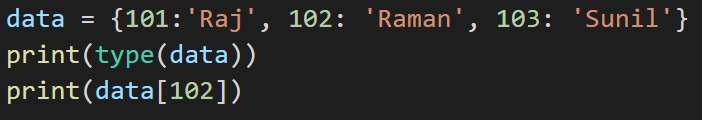


**2.Dictionary:** It is also called Mapping Type or dict. A map represents a group of elements in the form of key value pairs.

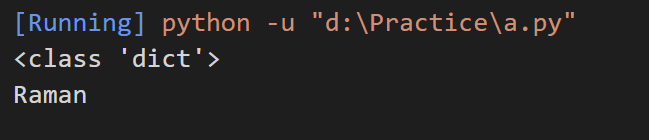
**Example:**

data = {101: ‘Raj’, 102: ‘Raman’, 103: ‘Sunil’}

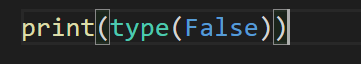
data = {‘Raj’: 101, ‘Raman’: 102, ‘Sunil’: 103}

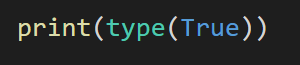


**OUTPUT**



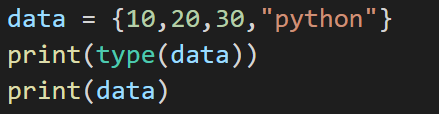
**3.Boolean:** The bool datatype represents Boolean value **True** or **False**. Python internally represents **True** as 1 and **False** as 0. **Example:** True, False



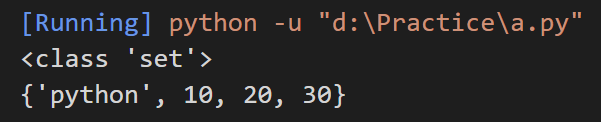


**4.Set:** A set is an unordered collection of elements much like a set-in mathematics. The order of elements is not maintained in the sets. It means the elements may not appear in the same order as they are entered into the set. A set does not accept duplicate elements. Sets are unordered so we cannot access its element using index. Sets are represented using curly brackets { }.

**Example:** data = {10,20,30,”python”,”Raj”,40}



**OUTPUT**



**5.Sequence Type:** Following are sequence type:

**String:** String represents group of characters. Strings are enclosed in double quotes or single quotes.

Ex: - “Hello”, ‘Sahil’

**Creating String**

**Single Quotes**

Ex: - a = ‘Python’

**Double Quotes**

Ex: - a = “Python”

**Triple Single Quotes**

Ex: - a = ‘’’Python’’’

**Triple Double Quotes**

Ex: - a = “””Python”””

**Double Quote inside Single Quotes**

Ex: - a = ‘Python “is a programming language”’

**Single Quote inside Double Quotes**

Ex: - a = “Python ‘is a programming’ language “

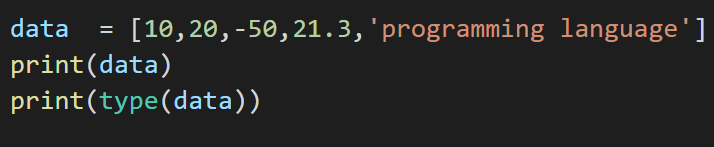
**Using Escape Characters**

Ex: - a = ‘Python \n is a programming language’

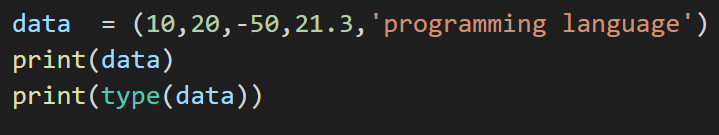
**Raw String -** is used to nullify the effect of escape characters

Ex: - a = r‘Python \n is a programming language’

**List:** A list represents a group of elements. A list can store different types of elements which can be modified. Lists are dynamic which means size is not fixed. Lists are represented using square bracket [ ]. **Example:** data = [10, 20, -50, 21.3, ‘Programming language’]

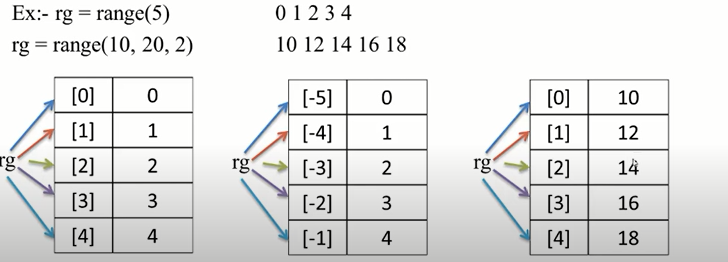


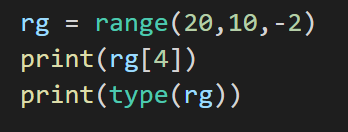
**Tuple:** A tuple contains a group of elements which can be different types. It is similar to List but Tuple are read-only which means we cannot modify its element. Tuples are represented using parentheses ( ). **Example:** data = (10,20,-50,21.3, ‘Programming language’)

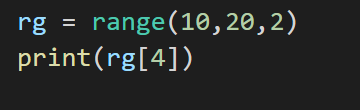


**Range:** Range represents a sequence of numbers. The numbers in the range are not modifiable.

**Example:**







**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.

**Example: “Sunil”, ‘Raman’**

**Types of Strings:**

There are two types of Strings supported in Python:

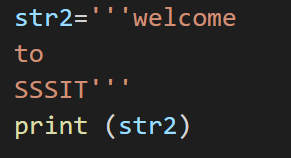
**a) Single-line String**- Strings that are terminated within a single-line are known as Single line Strings.

**Example:**

text1='hello'

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

**Using triple quotation marks: -**



**2. Numerical Literals**

Numeric literals can belong to the following four different numerical types.

**a. Int(integers)-** numbers can be both positive and negative with no fractional part. e.g., 100

**b. float(floating point) -** Real numbers with both integer and fractional

part e.g., -26.2, 30.3

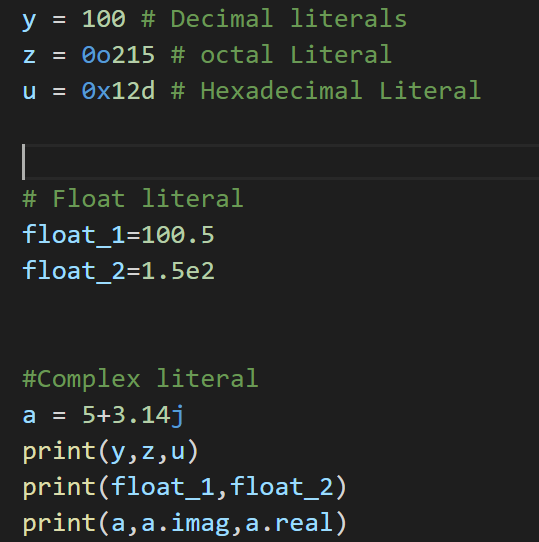
**c. Complex(complex)-** In the form of a+bj where a form the real part and b forms the imaginary part of the complex number e.g., 5+3.14j

Example- Numeric literals

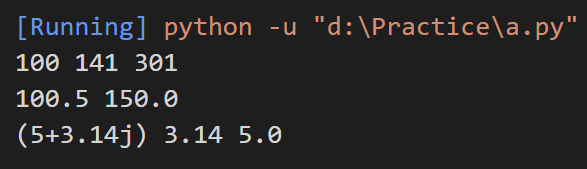
y = 100, 7684, 52 # Decimal literals (It consists of any sequence of digits is taken to be decimal literal unless it begins with 0(digit zero)).

z = 0o215, 0o56 # octal Literal (Any sequence of digits (from 0 to 7) starting with 0o (zero and letter o) is taken to be an octal integer).

u = 0x12d, 0x6A7 # Hexadecimal Literal (Any sequence of digits (from 0 to 9 digits and from a to f letters) preceded by 0X or 0x is taken to be a hexadecimal literal.



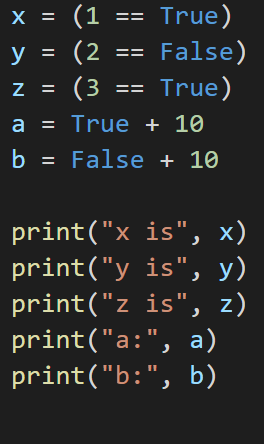
**OUTPUT**



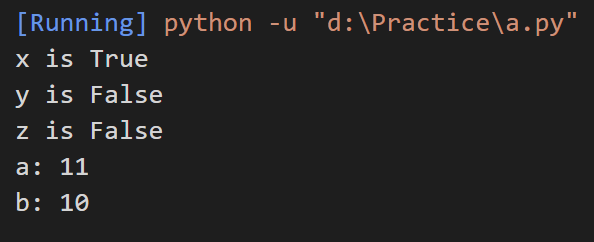
### **3. Boolean literals**

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

**Example - Boolean Literals**



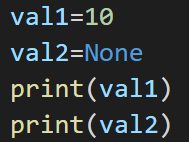
**OUTPUT**



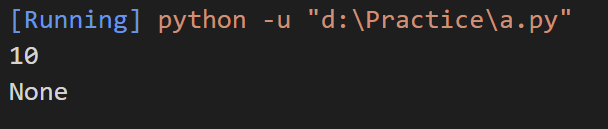
**4. Special literals:**

Python contains one special literal i.e., None. None is used to specify to that field that is not created. It is also used for the end of lists in Python.

Example:



**OUTPUT**



**5. Literal Collections:**

Python provides the four types of literal collection such as List literals, tuple literals, dict literals and set literals.

**list literals**: contains items of different data types. Lists are mutable. values stored in list are separated by comma(,) and enclosed within square brackets( [ ] ).

list=[1,2,3,'apple']

**tuple literals**: contains items of different data types. tuple is immutable. values stored in tuple are separated by comma( , ) and enclosed within parenthesis () but parenthesis is not compulsory

1,2.4,'mango' or (1,2.4, 'mango')

**set literals**: Values stored in set are separated by comma (,) and enclosed within curly braces

 {1,2.4,'mango'}

**dictionary literals**: Values stored in a set are separated by comma (,) and enclosed within curly braces but the important thing is that it comes in key and value pair. {key:value,key:value}

    {1:'roll',2:'name'}

  {'roll':100,'name:'shyam'}

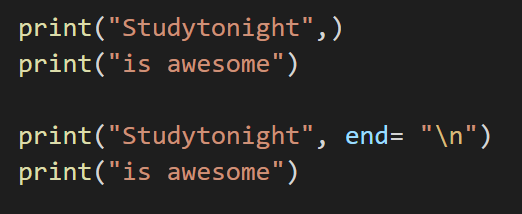
# **The 'sep' and 'end' parameters in Python print statement**

We will discuss how ‘**sep**’ and ‘**end**’ parameters can be used to change the way in which the contents of the print function are printed on the console.

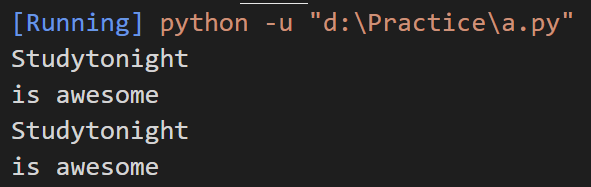
The **end** parameter:

The **end** parameter is used to append any string at the end of the output of the print statement in python. By default, the print method ends with a **newline**. This means there is no need to explicitly specify the parameter **end** as '**\n'**. Let us look at how changing the value of the **end** parameter changes the contents of the print statement onscreen.

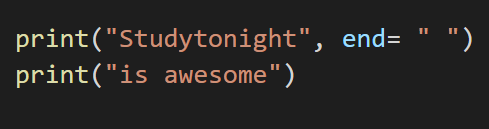
The below example demonstrates that passing **'\n'** or not specifying the **end** parameter both yield the same result. Execute 2 lines of code at a time to see the result.



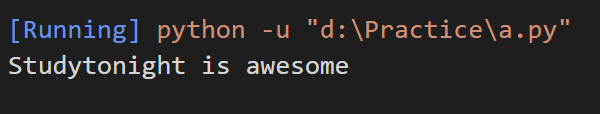
**OUTPUT**



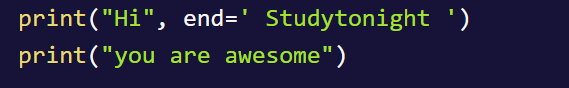
On the other hand, passing the whitespace to the **end** parameter indicates that the end character has to be identified by whitespace and not a newline (which is the default).



**OUTPUT**



The below example shows that any value can be passed to the **end** parameter and based on the content in the print statement, the end value gets displayed.



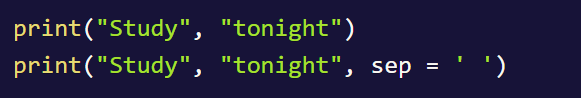
**OUTPUT**



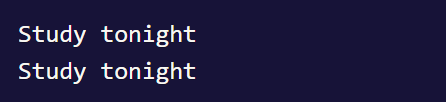
The **sep** parameter:

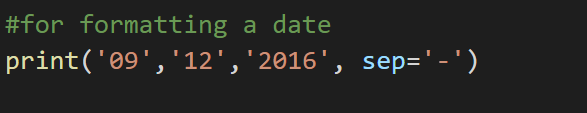
Sometimes, we may wish to print multiple values in a Python program in a readable manner. It adds a separator between strings to be printed. This is when the argument **sep** comes to play. The arguments passed to the program can be separated by different values. The default value for **sep** is whitespace. The **sep** parameter is primarily used to format the strings that need to be printed on the console and add a separator between strings to be printed. This feature was newly introduced in Python 3.x version.

The below example shows that passing **sep** parameter as whitespace or not passing the **sep** at all doesn't make a difference. Execute every line of code to see the result.

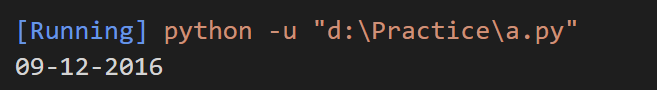


**OUTPUT**

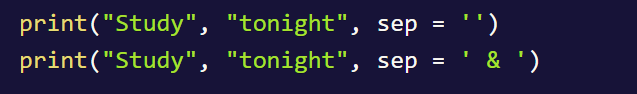




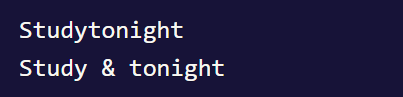
**OUTPUT**



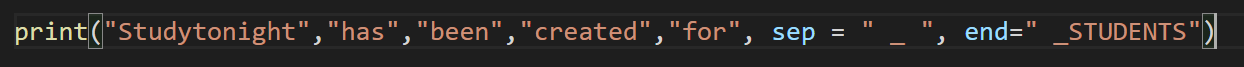
The below example shows different values that are passed to the **sep** parameter.



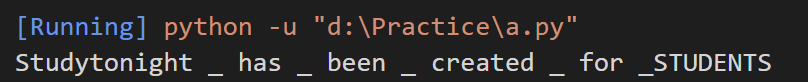
**OUTPUT**



The **sep** parameter when used with the [**end**](https://www.geeksforgeeks.org/gfact-50-python-end-parameter-in-print/) parameter it produces awesome results. Some examples by combining the **sep** and [**end**](https://www.geeksforgeeks.org/gfact-50-python-end-parameter-in-print/)parameters.

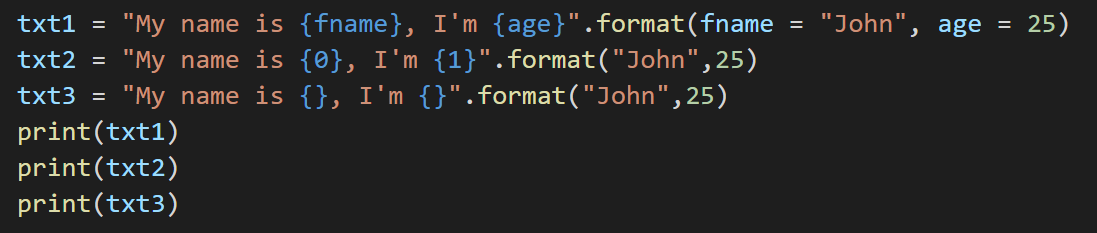


**OUTPUT**

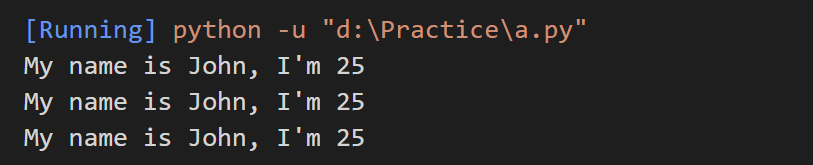


# **Python String format() Method**

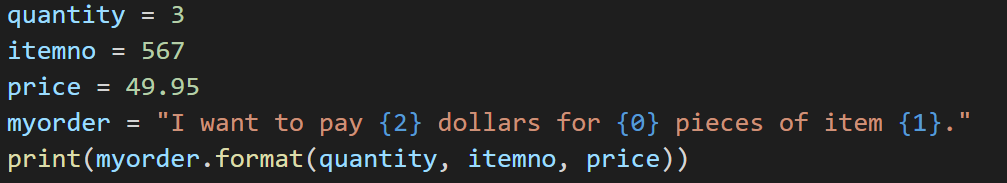
The **format()** method formats the specified value(s) and insert them inside the string's placeholder. The placeholder is defined using curly brackets: {}. Read more about the placeholders in the Placeholder section below. The **format()** method returns the formatted string.



**OUTPUT**



**Example 2:**



**OUTPUT**

