Data Science and Generative AI

by SATISH @

https://sathyatech.com

Python Programming

First Python Program

In this Section, we will discuss the basic syntax of Python, we will run a simple program to print **Hello World** on the console.

Python provides us the two ways to run a program:

- Using Interactive interpreter prompt
- Using a script file

Let's discuss each one of them in detail.

Interactive interpreter prompt

Here, we get the message **"Hello World !"** printed on the console.

Using a script file (Script Mode Programming)

The interpreter prompt is best to run the single-line statements of the code. However, we cannot write the code every-time on the terminal. It is not suitable to write multiple lines of code.

Using the script mode, we can write multiple lines code into a file which can be executed later. For this purpose, we need to open an editor like notepad, create a file named and save it with **.py** extension, which stands for **"Python".** Now, we will implement the above example using the script mode.

```
print ("hello world");

#here, we have used print() function to print the message on the console.
```

To run this file named as first.py, we need to run the following command on the terminal.

Step - 1: Open the Python interactive shell, and click **"File"** then choose **"New",** it will open a new blank script in which we can write our code.

Step -2: Now, write the code and press **"Ctrl+S"** to save the file.

Step - 3: After saving the code, we can run it by clicking "Run" or "Run Module". It will display the output to the shell.

The output will be shown as follows.

Step - 4: Apart from that, we can also run the file using the operating system terminal. But, we should be aware of the path of the directory where we have saved our file.

- Open the command line prompt and navigate to the directory.
- We need to type the **python** keyword, followed by the file name and hit enter to run the Python file.

Basic Syntax of Python

Indentation and Comment in Python

Indentation is the most significant concept of the Python programming language. Improper use of indentation will end up **"IndentationError"** in our code.

Indentation is nothing but adding whitespaces before the statement when it is needed. Without indentation Python doesn't know which statement to be executed to next. Indentation also defines which statements belong to which block. If there is no indentation or improper indentation, it will display "**IndentationError**" and interrupt our code.

In Python, statements that are the same level to the right belong to the same block. We can use four whitespaces to define indentation. Let's see the following lines of code.

Example -

```
1. list1 = [1, 2, 3, 4, 5]
2. for i in list1:
3.     print(i)
4.     if i==4:
5.         break
6. print("End of for loop")
```

Output:

```
1
2
3
4
End of for loop
```

Explanation:

In the above code, for loop has a code blocks and if the statement has its code block inside for loop. Both indented with four whitespaces. The last **print()** statement is not indented; that's means it doesn't belong to for loop.

Comments in Python

Comments are essential for defining the code and help us and other to understand the code. By looking the comment, we can easily understand the intention of every line that we have written in code. We can also find the error very easily, fix them, and use in other applications.

In Python, we can apply comments using the # hash character. The Python interpreter entirely ignores the lines followed by a hash character. A good programmer always uses the comments to make code under stable. Let's see the following example of a comment.

```
1. name  = "Thomas"   # Assigning string value to the name variable
```

We can add comment in each line of the Python code.

```
1. Fees = 10000      # defining course fees is 10000
2. Fees = 20000      # defining course fees is 20000
```

It is good idea to add code in any line of the code section of code whose purpose is not obvious. This is a best practice to learn while doing the coding.

Types of Comment

Python provides the facility to write comments in two ways- single line comment and multi-line comment.

Single-Line Comment - Single-Line comment starts with the hash # character followed by text for further explanation.

```
1. # defining the marks of a student
2. Marks = 90
```

We can also write a comment next to a code statement. Consider the following example.

```
1. Name = "James"   # the name of a student is James
2. Marks = 90          # defining student's marks
3. Branch = "Computer Science"   # defining student branch
```

Multi-Line Comments - Python doesn't have explicit support for multi-line comments but we can use hash # character to the multiple lines. **For example -**

```
1. # we are defining for loop
2. # To iterate the given list.
```

```
3. # run this code.
```

We can also use another way.

```
1. " " "
2. This is an example
3. Of multi-line comment
4. Using triple-quotes
5. " " "
```

This is the basic introduction of the comments. Visit our **Python Comment** tutorial to learn it in detail.

Python Identifiers

Python identifiers refer to a name used to identify a variable, function, module, class, module or other objects. There are few rules to follow while naming the Python Variable.

- A variable name must start with either an English letter or underscore (_).
- A variable name cannot start with the number.
- Special characters are not allowed in the variable name.
- The variable's name is case sensitive.

Let's understand the following example.

Example -

```
1. number = 10
2. print(num)
3.
4. _a = 100
5. print(_a)
6.
7. x_y = 1000
8. print(x_y)
```

Output:

```
10
100
1000
```

Python Keywords

Every scripting language has designated words or keywords, with particular definitions and usage guidelines. Python is no exception. The fundamental constituent elements of any Python program are Python keywords.

Python has a set of keywords that are reserved words that cannot be used as variable names, function names, or any other identifiers:

Keyword	Description
and	A logical operator
as	To create an alias
assert	For debugging
break	To break out of a loop
class	To define a class
continue	To continue to the next iteration of a loop
def	To define a function
del	To delete an object
elif	Used in conditional statements, same as else if
else	Used in conditional statements
except	Used with exceptions, what to do when an exception occurs
False	Boolean value, result of comparison operations
finally	Used with exceptions, a block of code that will be executed no matter if there is an exception or not

for	To create a for loop
from	To import specific parts of a module
global	To declare a global variable
if	To make a conditional statement
import	To import a module
in	To check if a value is present in a list, tuple, etc.
is	To test if two variables are equal
lambda	To create an anonymous function
None	Represents a null value
nonlocal	To declare a non-local variable
not	A logical operator
or	A logical operator
pass	A null statement, a statement that will do nothing
raise	To raise an exception
return	To exit a function and return a value
True	Boolean value, result of comparison operations
try	To make a try...except statement
while	To create a while loop
with	Used to simplify exception handling
yield	To return a list of values from a generator

Python Data Structures

Python offers four built-in data structures:lists, **tuples**, **sets**, and **dictionaries** that allow us to store data in an efficient way. Below are the commonly used data structures in Python, along with example code:

1. Lists

- Lists are **ordered collections** of data elements of different data types.
- Lists are **mutable** meaning a list can be modified anytime.
- Elements can be **accessed using indices**.
- They are defined using square bracket '**[]**'.

Example:

```
# Create a list
fruits = ['apple', 'banana', 'cherry']
print("fuirts[1] =", fruits[1])

# Modify list
fruits.append('orange')
print("fruits =", fruits)

num_list = [1, 2, 3, 4, 5]
# Calculate sum
sum_nums = sum(num_list)
print("sum_nums =", sum_nums)
```

Output:

```
fuirts[1] = banana
fruits = ['apple', 'banana', 'cherry', 'orange']
sum_nums = 15
```

2. Tuples

- Tuples are also **ordered collections** of data elements of different data types, similar to Lists.
- Elements can be **accessed using indices**.
- Tuples are **immutable** meaning Tuples can't be modified once created.
- They are defined using open bracket '**()**'.

Example:

```
# Create a tuple
```

```
2. point = (3, 4)
3. x, y = point
4. print("(x, y) =", x, y)
5.
6. # Create another tuple
7. tuple_ = ('apple', 'banana', 'cherry', 'orange')
8. print("Tuple =", tuple_)
```

Output:

```
(x, y) = 3 4
Tuple = ('apple', 'banana', 'cherry', 'orange')
```

3. Sets

- Sets are **unordered** collections of immutable data elements of different data types.
- Sets are **immutable**.
- Elements can't be accessed using indices.
- Sets **do not contain duplicate elements**.
- They are defined using curly braces '**{}**'

Example:

```
1. # Create a set
2. set1 = {1, 2, 2, 1, 3, 4}
3. print("set1 =", set1)
4.
5. # Create another set
6. set2 = {'apple', 'banana', 'cherry', 'apple', 'orange'}
7. print("set2 =", set2)
```

Output:

```
set1 = {1, 2, 3, 4}
set2 = {'apple', 'cherry', 'orange', 'banana'}
```

4. Dictionaries

- Dictionary are **key-value pairs** that allow you to associate values with unique keys.
- They are defined using curly braces '**{}**' with key-value pairs **separated by colons ':'**.
- Dictionaries are **mutable**.
- Elements can be accessed using keys.

Example:

```
1. # Create a dictionary
2. person = {'name': 'Umesh', 'age': 25, 'city': 'Noida'}
3. print("person =", person)
4. print(person['name'])
5.
6. # Modify Dictionary
7. person['age'] = 27
8. print("person =", person)
```

Output:

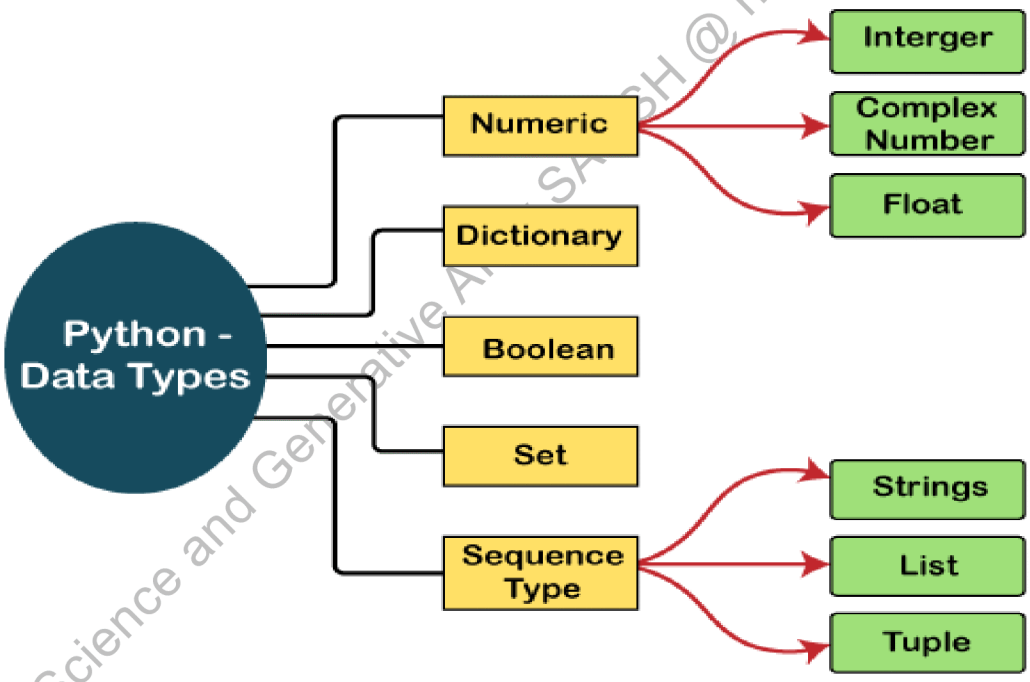
```
person = {'name': 'Umesh', 'age': 25, 'city': 'Noida'}
Umesh
```

Python 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



The data types will be briefly discussed in this tutorial section. We will talk about every single one of them exhaustively later in this instructional exercise.

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.

When a number is assigned to a variable, Python generates Number objects. For instance,

```
1. a = 5
2. print("The type of a", type(a))
3.
4. b = 40.5
5. print("The type of b", type(b))
6.
7. c = 1+3j
8. print("The type of c", type(c))
9. 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

Python supports three kinds of numerical data.

- **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.

When dealing with strings, the operation "hello"+" python" returns "hello python," and the operator + is used to combine two strings.

Because the operation "Python" *2 returns "Python," the operator * is referred to as a repetition operator.

The Python string is demonstrated in the following example.

Example - 1

```
1. str = "string using double quotes"
2. print(str)
3. s = '''A multiline
4. string'''
5. print(s)
```

Output:

string using double quotes
A multiline
string

Look at the following illustration of string handling.

Example - 2

```
1. str1 = 'hello SathyaTech' #string str1
2. str2 = ' how are you' #string str2
3. print (str1[0:2]) #printing first two character using slice operator
4. print (str1[4]) #printing 4th character of the string
5. print (str1*2) #printing the string twice
6. print (str1 + str2) #printing the concatenation of str1 and str2
```

List

Lists in Python are like arrays in C, but lists can contain data of different types. The things put away in the rundown are isolated with a comma (,) and encased inside square sections [].

To gain access to the list's data, we can use slice [:] operators. Like how they worked with strings, the list is handled by the concatenation operator (+) and the repetition operator (*).

Look at the following example.

Example:

```
1. list1  = [1, "hi", "Python", 2]
2. #Checking type of given list
3. print(type(list1))
4.
5. #Printing the list1
6. print (list1)
7.
8. # List slicing
9. print (list1[3:])
10.
11.# List slicing
12.print (list1[0:2])
13.
14.# List Concatenation using + operator
15.print (list1 + list1)
16.
17.# List repetation using * operator
18.print (list1 * 3)
```

Output:

```
[1, 'hi', 'Python', 2]
[2]
[1, 'hi']
[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]
[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]
```

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.

Because we cannot alter the size or value of the items in a tuple, it is a read-only data structure.

Let's look at a straightforward tuple in action.

Example:

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

```
4.
5.  #Printing the tuple
6.  print (tup)
7.
8.  # Tuple slicing
9.  print (tup[1:])
10. print (tup[0:1])
11.
12. # Tuple concatenation using + operator
13. print (tup + tup)
14.
15. # Tuple repatation using * operator
16. print (tup * 3)
17.
18. # Adding value to tup. It will throw an error.
19. 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.

```
1.  d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'}
2.
3.  # Printing dictionary
```

```
4. print (d)
5.
6. # Accesing value using keys
7. print("1st name is "+d[1])
8. print("2nd name is "+ d[4])
9.
10. print (d.keys())
11. 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.

```
1. # Python program to check the boolean type
2. print(type(True))
3. print(type(False))
4. 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.

```
1. # Creating Empty set
2. set1 = set()
3.
4. set2 = {'James', 2, 3,'Python'}
5.
6. #Printing Set value
7. print(set2)
8.
9. # Adding element to the set
10.
11.set2.add(10)
12.print(set2)
13.
14.#Removing element from the set
15.set2.remove(2)
16.print(set2)
```

Output:

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

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.

Python's built-in **id()** function returns the address where the object is stored.

```
a=10
print(id(a))

b='x'
print(id(b))

c=12.56
print(id(c))
```

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.

```
counter = 100        # Creates an integer variable

miles   = 1000.0     # Creates a floating point variable

name    = "SathyaTech"   # Creates a string variable
```

Deleting Python Variables

You can delete the reference to a number object by using the del statement

```
del var
delvar_a,var_b

Ex:
counter =100
print(counter)

del counter
print(counter)
100
Traceback (most recent call last):
  File "main.py", line 7, in <module>
    print (counter)
NameError: name 'counter' 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 ="Zara"
y =10
z =10.10

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

This will produce the following result:

```
<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 )
```

This will produce the following result:

```
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 )
```

This will produce the following result:

```
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.

```
>>> a=10
>>> b=10
>>> c=10
```

Instead of separate assignments, you can do it in a single assignment statement as follows –

```
>>> a=b=c=10
>>>print(a,b,c)
101010
```

In the following case, we have three variables with different values.

```
>>> a=10
>>> b=20
>>> c=30
```

These separate assignment statements can be combined in one. You need to give comma separated variable names on left, and comma separated values on the right of = operator.

```
>>>a,b,c=10,20,30
>>>print(a,b,c)
102030
```

Let's try few examples in script mode: –

```
a = b = c =100

print(a)
print(b)
print(c)
```

This produces the following result:

```
100
100
100
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example –

```
a,b,c=1,2,"Zara Ali"

print(a)
print(b)
print(c)
```

This produces the following result:

```
1
2
Zara Ali
```

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "Zara Ali" is assigned to the variable c.

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

Example

Following are valid Python variable names:

```
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)
```

This will produce the following result:

```
100
100
Zara
Nuha
20
100000
Example
```

Following are invalid Python variable names:

```
1counter =100
$_count  =100
```

```
zara-salary =100000

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

This will produce the following result:

```
File "main.py", line 3
   1counter = 100
       ^
SyntaxError: invalid syntax
```

Example

Once you use a variable to identify a data object, it can be used repeatedly without its id() value. Here, we have a variables height and width of a rectangle. We can compute the area and perimeter with these variables.

```
>>> width=10
>>> height=20
>>> area=width*height
>>> area
200
>>> perimeter=2*(width+height)
>>> perimeter
60
```

Use of variables is especially advantageous when writing scripts or programs. Following script also uses the above variables.

```
#! /usr/bin/python3
width =10
height =20
area = width*height
perimeter =2*(width+height)
print("Area = ", area)
print("Perimeter = ", perimeter)
```

Save the above script with .py extension and execute from command-line. The result would be –

```
Area = 200
Perimeter = 60
```

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

Example

Following is an example to show the usage of local variables:

```
defsum(x,y):
sum= x + y
returnsum
print(sum(5,10))
```

This will produce the following result –

```
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
defsum():
sum= x + y
returnsum
print(sum())
```

This will produce the following result –

```
15
```