Python Variables

Variable is a name that is used to refer to memory location. Python variable is also known as an identifier and used to hold value.

In Python, we don't need to specify the type of variable because Python is a infer language and smart enough to get variable type.

Variable names can be a group of both the letters and digits, but they have to begin with a letter or an underscore.

## **Identifier Naming**

Variables are the example of identifiers. An Identifier is used to identify the literals used in the program. The rules to name an identifier are given below.

* The first character of the variable must be an alphabet or underscore ( \_ ).
* All the characters except the first character may be an alphabet of lower-case(a-z), upper-case (A-Z), underscore, or digit (0-9).
* Identifier name must not contain any white-space, or special character (!, @, #, %, ^, &, \*).
* Identifier name must not be similar to any keyword defined in the language.
* Identifier names are case sensitive; for example, my name, and MyName is not the same.
* Examples of valid identifiers: a123, \_n, n\_9, etc.
* Examples of invalid identifiers: 1a, n%4, n 9, etc.

## **Declaring Variable and Assigning Values**

Python does not bind us to declare a variable before using it in the application. It allows us to create a variable at the required time.

We don't need to declare explicitly variable in Python. When we assign any value to the variable, that variable is declared automatically.

The equal (=) operator is used to assign value to a variable.

A = 5

## **Object References**

It is necessary to understand how the Python interpreter works when we declare a variable. The process of treating variables is somewhat different from many other programming languages.

Python is the highly object-oriented programming language; that's why every data item belongs to a specific type of class. Consider the following example.

1. print("Ravi")

The Python object creates an integer object and displays it to the console. In the above print statement, we have created a string object. Let's check the type of it using the Python built-in **type()** function.

1. type("Ravi")

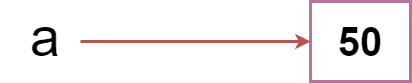
**Output:**

<class 'str'>

In Python, variables are a symbolic name that is a reference or pointer to an object. The variables are used to denote objects by that name.

Let's understand the following example

1. a = 50



In the above image, the variable **a** refers to an integer object.

Suppose we assign the integer value 50 to a new variable b.

a = 50

b = a

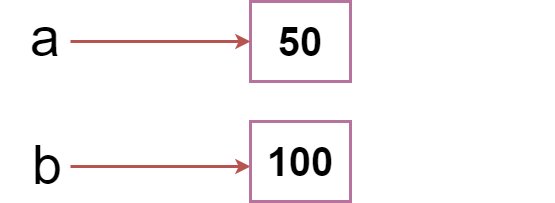


The variable b refers to the same object that a point to because Python does not create another object.

Let's assign the new value to b. Now both variables will refer to the different objects.

a = 50

b =100



Python manages memory efficiently if we assign the same variable to two different values.

## **Object Identity**

In Python, every created object identifies uniquely in Python. Python provides the guaranteed that no two objects will have the same identifier. The built-in **id ()** function, is used to identify the object identifier. Consider the following example.

1. a = 50
2. b = a
3. print(id(a))
4. print(id(b))
5. # Reassigned variable a
6. a = 500
7. print(id(a))

**Output:**

140734982691168

140734982691168

2822056960944

We assigned the **b = a, a** and **b** both point to the same object. When we checked by the **id()** function it returned the same number. We reassign **a** to 500; then it referred to the new object identifier.

## **Variable Names**

We have already discussed how to declare the valid variable. Variable names can be any length can have uppercase, lowercase (A to Z, a to z), the digit (0-9), and underscore character (\_). Consider the following example of valid variables names.

1. name = "Devansh"
2. age = 20
3. marks = 80.50
5. print(name)
6. print(age)
7. print(marks)

**Output:**

Devansh

20

80.5

The multi-word keywords can be created by the following method.

* **Camel Case -** In the camel case, each word or abbreviation in the middle of begins with a capital letter. There is no intervention of whitespace. For example - nameOfStudent, valueOfVaraible, etc.
* **Pascal Case -** It is the same as the Camel Case, but here the first word is also capital. For example - NameOfStudent, etc.
* **Snake Case -** In the snake case, Words are separated by the underscore. For example - name\_of\_student, etc.

## **Multiple Assignment**

Python allows us to assign a value to multiple variables in a single statement, which is also known as multiple assignments.

We can apply multiple assignments in two ways, either by assigning a single value to multiple variables or assigning multiple values to multiple variables. Consider the following example.

**1. Assigning single value to multiple variables**

**Eg:**

1. x=y=z=50
2. print(x)
3. print(y)
4. print(z)

**Output:**

50

50

50

**2. Assigning multiple values to multiple variables:**

**Eg:**

1. a,b,c=5,10,15
2. print a
3. print b
4. print c

**Output:**

5

10

15

The values will be assigned in the order in which variables appear.

## **Python Variable Types**

There are two types of variables in Python - Local variable and Global variable. Let's understand the following variables.

### **Local Variable**

Local variables are the variables that declared inside the function and have scope within the function. Let's understand the following example.

**Example -**

1. # Declaring a function
2. **def** add():
3. # Defining local variables. They has scope only within a function
4. a = 20
5. b = 30
6. c = a + b
7. **print**("The sum is:", c)
9. # Calling a function
10. add()

**Output:**

The sum is: 50

**Explanation:**

In the above code, we declared a function named **add()** and assigned a few variables within the function. These variables will be referred to as the **local variables** which have scope only inside the function. If we try to use them outside the function, we get a following error.

1. add()
2. # Accessing local variable outside the function
3. **print**(a)

**Output:**

The sum is: 50

print(a)

NameError: name 'a' is not defined

We tried to use local variable outside their scope; it threw the **NameError.**

### **Global Variables**

Global variables can be used throughout the program, and its scope is in the entire program. We can use global variables inside or outside the function.

A variable declared outside the function is the global variable by default. Python provides the **global** keyword to use global variable inside the function. If we don't use the **global** keyword, the function treats it as a local variable. Let's understand the following example.

**Example -**

1. # Declare a variable and initialize it
2. x = 101
4. # Global variable in function
5. **def** myFunction():
6. # printing a global variable
7. **global** x
8. **print**(x)
9. # modifying a global variable
10. x = 'Welcome To MCA CLASS'
11. **print**(x)
13. myFunction()
14. **print**(x)

**Output:**

101

Welcome To MCA CLASS'

Welcome To MCA CLASS'

**Explanation:**

In the above code, we declare a global variable **x** and assign a value to it. Next, we defined a function and accessed the declared variable using the **global** keyword inside the function. Now we can modify its value. Then, we assigned a new string value to the variable x.

Now, we called the function and proceeded to print **x**. It printed the as newly assigned value of x.

## **Delete a variable**

We can delete the variable using the **del** keyword. The syntax is given below.

**Syntax -**

1. **del** <variable\_name>

In the following example, we create a variable x and assign value to it. We deleted variable x, and print it, we get the error **"variable x is not defined"**. The variable x will no longer use in future.

**Example -**

1. # Assigning a value to x
2. x = 6
3. **print**(x)
4. # deleting a variable.
5. **del** x
6. **print**(x)

**Output:**

6

Traceback (most recent call last):

File "C:/Users/DEVANSH SHARMA/PycharmProjects/Hello/multiprocessing.py", line 389, in

print(x)

NameError: name 'x' is not defined

## **Maximum Possible Value of an Integer in Python**

Unlike the other programming languages, Python doesn't have long int or float data types. It treats all integer values as an **int** data type. Here, the question arises. What is the maximum possible value can hold by the variable in Python? Consider the following example.

**Example -**

1. # A Python program to display that we can store
2. # large numbers in Python
4. a = 10000000000000000000000000000000000000000000
5. a = a + 1
6. **print**(type(a))
7. **print** (a)

**Output:**

<class 'int'>

10000000000000000000000000000000000000000001

As we can see in the above example, we assigned a large integer value to variable **x** and checked its type. It printed **class <int>** not long int. Hence, there is no limitation number by bits and we can expand to the limit of our memory.

Python doesn't have any special data type to store larger numbers.

### **Print Single and Multiple Variables in Python**

We can print multiple variables within the single print statement. Below are the example of single and multiple printing values.

**Example - 1 (Printing Single Variable)**

1. # printing single value
2. a = 5
3. **print**(a)
4. **print**((a))

**Output:**

5

5

**Example - 2 (Printing Multiple Variables)**

1. a = 5
2. b = 6
3. # printing multiple variables
4. **print**(a,b)
5. # separate the variables by the comma
6. Print(1, 2, 3, 4, 5, 6, 7, 8)

**Output:**

5 6

1 2 3 4 5 6 7 8

## **Basic Fundamentals:**

This section contains the fundamentals of Python, such as:

**i)Tokens and their types.**

**ii) Comments**

* 1. **Tokens:**
* The tokens can be defined as a punctuator mark, reserved words, and each word in a statement.
* The token is the smallest unit inside the given program.

There are following tokens in Python:

* Keywords.
* Identifiers.
* Literals.
* Operators.

Python Keywords

Python Keywords are special reserved words that convey a special meaning to the compiler/interpreter. Each keyword has a special meaning and a specific operation. These keywords can't be used as a variable. Following is the List of Python Keywords.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| True | False | None | and | as |
| asset | def | class | continue | break |
| else | finally | elif | del | except |
| global | for | if | from | import |
| raise | try | or | return | pass |
| nonlocal | in | not | is | lambda |

Consider the following explanation of keywords.

1. **True -** It represents the Boolean true, if the given condition is true, then it returns "True". Non-zero values are treated as true.
2. **False -** It represents the Boolean false; if the given condition is false, then it returns "False". Zero value is treated as false
3. **None -** It denotes the null value or void. An empty list or Zero can't be treated as **None**.
4. **and -** It is a logical operator. It is used to check the multiple conditions. It returns true if both conditions are true. Consider the following truth table.

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **A and B** |
| True | True | True |
| True | False | False |
| False | True | False |
| False | False | False |

1. **or** - It is a logical operator in Python. It returns true if one of the conditions is true. Consider the following truth table.

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **A or B** |
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

1. **not** - It is a logical operator and inverts the truth value. Consider the following truth table.

|  |  |
| --- | --- |
| **A** | **Not A** |
| True | False |
| False | True |

**7. assert -** This keyword is used as the debugging tool in Python. It checks the correctness of the code. It raises an **AssertionError** if found any error in the code and also prints the message with an error.

**Example:**

1. a = 10
2. b = 0
3. **print**('a is dividing by Zero')
4. **assert** b != 0
5. **print**(a / b)

**Output:**

a is dividing by Zero

Runtime Exception:

Traceback (most recent call last):

File "/home/40545678b342ce3b70beb1224bed345f.py", line 4, in

assert b != 0, "Divide by 0 error"

AssertionError: Divide by 0 error

**8. def -** This keyword is used to declare the function in Python. If followed by the function name.

1. **def** my\_func(a,b):
2. c = a+b
3. **print**(c)
4. my\_func(10,20)

**Output:**

30

**9. class -** It is used to represents the class in Python. The class is the blueprint of the objects. It is the collection of the variable and methods. Consider the following class.

1. **class** Myclass:
2. #Variables……..
3. **def** function\_name(self):
4. #statements………

**10. continue -** It is used to stop the execution of the current iteration. Consider the following example.

1. a = 0
2. **while** a < 4:
3. a += 1
4. **if** a == 2:
5. **continue**
6. **print**(a)

**11. break -** It is used to terminate the loop execution and control transfer to the end of the loop. Consider the following example.

**Example**

1. **for** i **in** range(5):
2. **if**(i==3):
3. **break**
4. **print**(i)
5. **print**("End of execution")

**Output:**

0

1

2

End of execution

**12. If -** It is used to represent the conditional statement. The execution of a particular block is decided by if statement.

**13. else -** The else statement is used with the if statement. When if statement returns false, then else block is executed.

**elif -** This Keyword is used to check the multiple conditions. It is short for **else-if**. If the previous condition is false, then check until the true condition is found. Condition the following example.

**Example:**

1. marks = int(input("Enter the marks:"))
2. **if**(marks>=90):
3. **print**("Excellent")
4. **elif**(marks<90 **and** marks>=75):
5. **print**("Very Good")
6. **elif**(marks<75 **and** marks>=60):
7. **print**("Good")
8. **else**:
9. **print**("Average")

**15. del -** It is used to delete the reference of the object. Consider the following example.

**Example:**

1. a=10
2. b=12
3. **del** a
4. **print**(b)
5. # a is no longer exist
6. **print**(a)

**16. try, except -** The try-except is used to handle the exceptions. The exceptions are run-time errors. Consider the following example.

1. a = 0
2. **try**:
3. b = 1/a
4. **except** Exception as e:
5. **print**(e)

**Output:**

division by zero

**17. raise -** The raise keyword is used to through the exception forcefully. Consider the following example.

**Example**

1. a = 5
2. **if** (a>2):
3. **raise** Exception('a should not exceed 2 ')

**Output:**

Exception: a should not exceed 2

**18. finally -** The **finally** keyword is used to create a block of code that will always be executed no matter the else block raises an error or not. Consider the following example.

**Example:**

1. a=0
2. b=5
3. **try**:
4. c = b/a
5. **print**(c)
6. **except** Exception as e:
7. **print**(e)
8. **finally**:
9. **print**('Finally always executed')

**Output:**

division by zero

Finally always executed

**19. for, while -** Both keywords are used for iteration. The **for** keyword is used to iterate over the sequences (list, tuple, dictionary, string). A while loop is executed until the condition returns false.

**20. import -** The import keyword is used to import modules in the current Python script. The module contains a runnable Python code.

**21. from -** This keyword is used to import the specific function or attributes in the current Python script.

**Example:**

1. **from** math **import** sqrt
2. **print**(sqrt(25))

**Output:**

5

**22. as -** It is used to create a name alias. It provides the user-define name while importing a module.

**Example:**

1. **import** calendar as cal
2. **print**(cal.month\_name[5])

**Output:**

May

**23. pass -** The **pass** keyword is used to execute nothing or create a placeholder for future code. If we declare an empty class or function, it will through an error, so we use the pass keyword to declare an empty class or function.

**Example:**

1. **class** my\_class:
2. **pass**
4. **def** my\_func():
5. **pass**

**24. return -** The **return** keyword is used to return the result value or none to called function.

**Example:**

1. **def** sum(a,b):
2. c = a+b
3. **return** c
5. **print**("The sum is:",sum(25,15))

**25. is -** This keyword is used to check if the two-variable refers to the same object. It returns the true if they refer to the same object otherwise false. Consider the following example.

**Example**

1. x = 5
2. y = 5
4. a = []
5. b = []
6. **print**(x **is** y)
7. **print**(a **is** b)

**Output:**

True

False

**26. global -** The global keyword is used to create a global variable inside the function. Any function can access the global. Consider the following example.

**Example**

1. **def** my\_func():
2. **global** a
3. a = 10
4. b = 20
5. c = a+b
6. **print**(c)
8. my\_func()
10. **def** func():
11. **print**(a)
13. func()

**Output:**

30

10

**27. nonlocal -** The **nonlocal** is similar to the **global** and used to work with a variable inside the nested function(function inside a function). Consider the following example.

**Example**

1. **def** outside\_function():
2. a = 20
3. **def** inside\_function():
4. nonlocal a
5. a = 30
6. **print**("Inner function: ",a)
7. inside\_function()
8. **print**("Outer function: ",a)
9. outside\_function()

**Output:**

Inner function: 30

Outer function: 30

**28. lambda -** The lambda keyword is used to create the anonymous function in Python. It is an inline function without a name. Consider the following example.

**Example**

1. a = **lambda** x: x\*\*2
2. **for** i **in** range(1,6):
3. **print**(a(i))

**Output:**

1

4

9

16

25

**29. yield -** The **yield** keyword is used with the Python generator. It stops the function's execution and returns value to the caller. Consider the following example.

**Example**

1. **def** fun\_Generator():
2. **yield** 1
3. **yield** 2
4. **yield** 3

7. # Driver code to check above generator function
8. **for** value **in** fun\_Generator():
9. **print**(value)

**Output:**

1

2

3

**30. with -** The **with** keyword is used in the exception handling. It makes code cleaner and more readable. The advantage of using **with**, we don't need to call **close()**. Consider the following example.

**Example**

1. with open('file\_path', 'w') as file:
2. file.write('hello world !')

**31. None -** The None keyword is used to define the null value. It is remembered that **None** does not indicate 0, false, or any empty data-types. It is an object of its data type, which is Consider the following example.

**Example:**

1. **def** return\_none():
2. a = 10
3. b = 20
4. c = a + b
6. x = return\_none()
7. **print**(x)

**Output:**

None

# 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:**

1. "Aman" , '12345'

**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:**

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:**

1. text1='hello\
2. user'
3. **print**(text1)

'hellouser'

**2) Using triple quotation marks:-**

**Example:**

1. str2='''''welcome
2. to
3. SSSIT'''
4. **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 |

**Example - Numeric Literals**

1. x = 0b10100 #Binary Literals
2. y = 100 #Decimal Literal
3. z = 0o215 #Octal Literal
4. u = 0x12d #Hexadecimal Literal
6. #Float Literal
7. float\_1 = 100.5
8. float\_2 = 1.5e2
10. #Complex Literal
11. a = 5+3.14j
13. **print**(x, y, z, u)
14. **print**(float\_1, float\_2)
15. **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.

**Example - Boolean Literals**

1. x = (1 == True)
2. y = (2 == False)
3. z = (3 == True)
4. a = True + 10
5. b = False + 10
7. **print**("x is", x)
8. **print**("y is", y)
9. **print**("z is", z)
10. **print**("a:", a)
11. **print**("b:", b)

**Output:**

x is True

y is False

z is False

a: 11

b: 10

### **IV. 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 - Special Literals**

1. val1=10
2. val2=None
3. **print**(val1)
4. **print**(val2)

**Output:**

10

None

### **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**

1. list=['John',678,20.4,'Peter']
2. list1=[456,'Andrew']
3. **print**(list)
4. **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**

1. dict = {'name': 'Pater', 'Age':18,'Roll\_nu':101}
2. **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**

1. tup = (10,20,"Dev",[2,3,4])
2. **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**

1. set = {'apple','grapes','guava','papaya'}
2. **print**(set)

**Output:**

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

References: https://www.javatpoint.com/python-literals