Chapter 2

Python Tokens

The smallest units of Python language are the characters used to write Python tokens. These characters are defined by the *Unicode* character set, an emerging standard that tries to create characters. The Unicode is a 16-bit character coding system and currently support more than 34,000 defined characters derived from 24 languages from America, Europe, Middle East, Africa and Asia (including India). However, most of us use only the basic ASCII characters, which include letters, digits and punctuation marks, used in normal English. ASCII character set is a subset of Unicode character set.

Character Set

Character set is a set of valid characters that a language can recognize. A character represents any letter, digit or any other symbol.

character set Category	**Example**
Letters	A-Z, a-z
Digits	0-9
Special Symbols	Space + - * / ** \ () [] {} // = != == < > . „ " „ "" """ , ; : % ! & # <= >= @ >>> << >> _(underscore)
White Spaces	Blank space, tabs, carriage return, new line, form-feed
Other Characters	All other ASCII and Unicode characters

A Python program is made up of various statements and a statement is made up of Variables, Constants, Operators etc.

Tokens are the various Python program elements which are identified by the compiler. *A token is the smallest element of a program that is meaningful to the*

compiler. Tokens supported in Python include keywords, variables, constants, special characters, operations etc.

When you compile a program, the compiler scans the text in your source code and extracts individual tokens. While tokenizing the source file, the compiler recognizes and subsequently removes whitespaces (spaces, tabs, newline and form feeds) and the text enclosed within comments. Now let us consider a program

Python Token

Python language includes five types of tokens and they are:

- **Keyword**
- **Identifiers (Names)**
- **Literals**
- **Operators**
- **Punctuators**

Keywords or Reserved Word

Keywords are the reserved words in Python. There are as many as 33 such keywords in Python, each serving a different purpose.

We cannot use a keyword as variable name, function name, class or any other identifier. Here's a list of all keywords in Python Programming

False	**class**	**finally**	**is**	**return**
None	**continue**	**for**	**lambda**	**try**
True	**def**	**from**	**nonlocal**	**while**
and	**del**	**global**	**not**	**with**
as	**elif**	**if**	**or**	**yield**
assert	**else**	**import**	**pass**	
break	**except**	**in**	**raise**	

To print the keyword list of Python type the command as:

```
$ python
>>>
>>> import keyword

>>> print keyword.kwlist

['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else',
'except', 'exec', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is',
'lambda', 'not', 'or', 'pass', 'print', 'raise', 'return', 'try', 'while', 'with',
'yield']
```

Identifiers

An Identifier is used to identify the program element. They are used for naming class, array, function, file or variable. Pyhton Identifiers follow the following rules:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-Z,a-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- Python Identifiers can also not have special characters **[‘.’, ‘!’, ‘@’, ‘#’, ‘$’, ‘%’]** in their formation. These symbols are forbidden.
- Python doc says that you can **have an identifier with unlimited length**. But it is just the half truth.
- Using a large name (more than 79 chars) would lead to the violation of a rule set by the **PEP-8** standard.

PEP 8, sometimes spelled **PEP8** or **PEP-8**, is a document that provides **guidelines** and best practices on how to write **Python code**. It was written in 2001 by Guido van Rossum, Barry Warsaw, and Nick Coghlan. The primary focus of **PEP 8** is to improve the readability and consistency of **Python code**.

Python Variables

Variables are nothing but reserved memory locations to store values. It means that when you create a variable, you reserve some space in the memory.

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

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.

In programming language like C and many other languages, the concepts of variable is connected to memory location. Variables are nothing but reserved memory locations to store values. In all languages, a variable is imagined as a storage container which can store some values. e.g.

a=1

Some memory is allocated with the name 'a' and there the value '1' is stored.

a

If we change the Value of the variable, then the container will be updated with the new value. For example:

a =2

then the new value '2' is stored into the same box, as shown below:

a

When we assign one variable to another variable as:

int b = a # in 'C' language

This is how variables are visualized in other language. However in Python, a variable is seen as a tag that is tied to some value. For example:

a=1

means the value '1' is created first in memory and then a tag by the name 'a' is created to show that value:

a → 1

Python considers the values as 'objects'. If we change the value of 'a' to a new value as:

a=2

then the tag is simply changes to the new value (or object) as shown below:

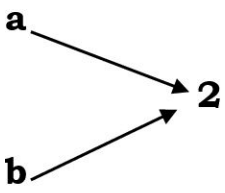
a⇒ 2 1

Since the value '1' becomes unreferenced object, it is removed by garbage collector.

Assigning one variables to another variable makes a new tag bound to the same value. For example,

b=a

Here, we are store 'a' value into 'b'. A new tag 'b' will be created that refers to '2' as:



In other language when we assign a value of one variable to other they created to separate memory locations for both variables. But in Python, there is only

one memory reference by two names(tags). It means Python is using memory efficiently.

Declaring Variable and Assigning Values

Python does not bound us to declare variable before using in the application. It allows us to create variable at 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.

Eg:

```
counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string
print (counter)
print (miles)
print (name)
```

Here, 100, 1000.0 and "John" are the values assigned to counter, miles, and name variables, respectively. This produces the following result –

```
100
1000.0
John
```

Multiple Assignment

Python allows you to assign a single value to several variables simultaneously.

For example –

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all the 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, "john"
```

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

LITERALS

Literals are data items that have a fixed values.

String Literals

- Single Line String
- Multi Line String
- Unicode String

Single Line String	“hello world”
Multi Line String	“””Allahabad Uttar Pradesh”””
Character	"C"
Unicode string	u"\u00dcnic\u00f6de“, u"\u0930\u093E\u091C\u0940 \u0935"

Numeric Literals

Numeric Literals are immutable (unchangeable). Numeric literals can belong to 3 different numerical types Integer, Float and Complex.

- Integer
- Float
- Complex

EXAMPLE

```
a = 0b1010 #Binary Literals

b = 100 #Decimal Literal

c = 0o310 #Octal Literal

d = 0x12c #Hexadecimal Literal

#Float Literal

float_1 = 10.5
```

```
float_2 = 1.5e2

#Complex Literal

x = 3.14j

print(a, b, c, d)

print(float_1, float_2)

print(x, x.imag, x.real)
```

Boolean Literals

There are two kinds of Boolean literal: **True** and **False**. Boolean literal contains 1 as true value and 0 as false value.

•Example:-

```
x = (1 == True)
y = (1 == False)
a = True + 4
b = False + 10
```

Literal Collection

- List
- Tuple
- Dictionary
- Set

Datatypes in Python

A datatypes represents the type of data stored into a variable or memory. The datatypes which are already available in python language are called *Built-in or standard* datatypes. The datatypes which can be created by the programmers are called *User-defined* datatypes.

Python has seven standard data types –

- Numbers

- String
- List
- Tuple
- Dictionary
- Sets
- Range

Numbers

Number stores numeric values. Python creates Number objects when a number is assigned to a variable. For example;

```
1. a = 3 , b = 5  #a and b are number objects
```

Python supports 4 types of numeric data.

1. int (signed integers like 10, 2, 29, etc.)
2. long (long integers used for a higher range of values like 908090800L, -0x1929291L, etc.)
3. float (float is used to store floating point numbers like 1.9, 9.902, 15.2, etc.)
4. complex (complex numbers like 2.14j, 2.0 + 2.3j, etc.)

Python allows us to use a lower-case L to be used with long integers. However, we must always use an upper-case L to avoid confusion.

A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts and suffix J represent the square root value of -1).

String

The string can be defined as the sequence of characters represented in the quotation marks. In python, we can use single, double, or triple quotes to define a string.

String handling in python is a straightforward task since there are various inbuilt functions and operators provided.

In the case of string handling, the operator + is used to concatenate two strings as the operation *"hello"+" python"* returns *"hello python"*.

The operator * is known as repetition operator as the operation "Python " *2 returns "Python Python ".

The following example illustrates the string handling in python.

```
1. str1 = 'hello GEHU' #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 are similar to arrays in C. A list represent a group of elements. The main difference between a list and an array is that a list can store different types of elements but an array can store only one type of elements. Also, lists can grow dynamically in memory. But the size of array is fixed and they cannot grow at runtime.

The items stored in the list are separated with a comma (,) and enclosed within square brackets [].

We can use slice [:] operators to access the data of the list. The concatenation operator (+) and repetition operator (*) works with the list in the same way as they were working with the strings.

Consider the following example.

```
1. l  = [1, "hi", "python", 2]
2. print (l[3:])
3. print (l[0:2])
4. print (l)
5. print (l + l)
6. print (l * 3)
   l[2] = “Hello”
```

Output:

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

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

Tuple

A tuple is similar to the list in many ways. Like lists, tuples also contain the collection of the items of different data types. The items of the tuple are separated with a comma (,) and enclosed in parentheses ().

A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple. Let's see a simple example of the tuple.

```
1. t  = ("hi", "python", 2)
2. print (t[1:]);
3. print (t[0:1]);
4. print (t);
5. print (t + t);
6. print (t * 3);
7. print (type(t))
8. t[2] = "hi";
```

Output:

```
('python', 2)
('hi',)
('hi', 'python', 2)
('hi', 'python', 2, 'hi', 'python', 2)
('hi', 'python', 2, 'hi', 'python', 2, 'hi', 'python', 2)
<type 'tuple'>
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    t[2] = "hi";
TypeError: 'tuple' object does not support item assignment
```

Dictionary:

- A dictionary is a collection which is unordered, changeable and indexed.
- It works like an associated array where no two keys can be same.
- Dictionaries are enclosed by curly braces ({}) and values can be retrieved by square bracket([]).

Eg:

```
1. d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'};
2. print("1st name is "+d[1]);
3. print("2nd name is "+ d[4]);
4. print (d);
5. print (d.keys());
6. print (d.values());
```

Output

1st name is Jimmy
2nd name is mike
{1: 'Jimmy', 2: 'Alex', 3: 'john', 4: 'mike'}
[1, 2, 3, 4]
['Jimmy', 'Alex', 'john', 'mike']

Set Datatype

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 elements. To create a set, we should enter the elements separated by commas inside curly braces {}.

```
s = {5,10,15,10, 35}

print(s)  # may display {35,5,15,10)
```

Please observe that the set 's' is not maintain the order of the elements. We entered the elements in the order 5,10,15,10 and 35. But it is showing another order. Also, we repeated the element 10 in the set, but it has stored only one 10. We can use the set() function to create a set as:

```
ch = set("Hello")

print(ch)  # may display {'H', 'e', 'l', 'o'}
```

We can convert a list into set using the set() function as:

```
lst = [1,2,3,4,5]

s = set(lst)

print(s)
```

Since sets are unordered, we cannot retrieve the elements using indexing or slicing operations. For example, the following statements will give error messages:

print(s[0]) # indexing display 0th element

print(s[0:2]) # slicing display from 0 to 1st elements

The update() method is used to add elements to a set as:

s.update([50,60]

print(s) # may display {1,2,3,4,5,50,60}

on the other hand, the remove() method is used to remove any particular element from a set as:

s.remove(50)

print(s) # may display{1,2,3,4,5,60}

range datatypes

The range datatypes represents a sequence of numbers. The numbers I the range are not modifiable. Generally, range is used for repeating a for loop for a specific number of times. To create a range of numbers, we can simply write:

r = range(10)

here, the range object is created with the numbers starting from 0 to 9. We can display these number using a for loop as:

for I in r: print(i)

The above statement will display numbers starting from 0 to 9. We can use a starting number, and ending number and a step value in the range object as:

r = range(30,40,2)

This will create a range object with a starting number 30 and an ending number 39. The step size is 2. It means the numbers in the range will increase by 2 every time. So, the for loop

For i in r: print(i)

Will display numbers: 30, 32, 36, 38.

Operators

Python operators are symbols that are used to perform mathematical or logical manipulations. Operands are the values or variables with which the operator is applied to, and values of operands can manipulate by using the operators.

6 + 2=8, where there are two operands and a plus (+) operator, and the result turns 8.

Here a single operator is used to manipulate the values. The +, -, *, / and ** does addition, subtraction, multiplication, division & exponentiation respectively.

Types of Operators

Python programming language is rich with built-in operators.

Python supports the following types of operators:

1. Arithmetic Operators
2. Assignment Operators
3. Comparison (Relational) Operators
4. Logical Operators
5. Identity Operators
6. Bitwise Operators
7. Membership Operators

1. Arithmetic Operators

With arithmetic operators, we can do various arithmetic operations like addition, subtraction, multiplication, division, modulus, exponent, etc. There are seven arithmetic operators available in Python. Since these operators act on two operands, they are called 'Binary Operator' also. Let's assume a =13 and b =5 and see the effect of various arithmetic operators.

The table below outlines the built-in arithmetic operators in Python.

Symbol	Operator Name	Description	Example	Result
+	Addition	Adds the values on either side of the operator and calculate a result.	a + b	18
-	Subtraction	Subtracts values of right side operand from left side operand.	a-b	8
*	Multiplication	Multiplies the values on both sides of the operator.	a * b	65
/	Division	Divides left side operand with right side operand.	a/b	2.6
%	Modulus	It returns the remainder by dividing the left side operand with right side operand	a%b	3
**	Exponent	Calculates the exponential power	2** 3	8
//	Integer Division or Floor Division	Here the result is the quotient in which the digits after decimal points are not taken into account. i.e. it performs division and gives only integer quotient.	a//b	2

2. Assignment Operators

These operators are useful to store the right side value into a left side variable. They can also be used to perform simple arithmetic operations like addition, subtraction etc. and then store the result into a variable. These operators are shown below:

(Let's assume the value a = 20 and b =10 and c = 5:

Symbol	Operator Name	Description	Example	Result
=	Equal	Assigns the values of right side operand to left side operand.	c = a + b	c = 15
+=	Add AND	Adds right side operand value to the left side operand value and assigns the results to the left operand.	c+=a	c = 25
-=	Subtract AND	Subtracts right side operand value to the left side operand value and assigns the results to the left operand.	c-=a	c = 15
*=	Multiply AND	Similarly does their respective operations and assigns the operator value to the left operand.	c*=a	c = 100
/=	Division AND		c/=a	c = 0.25
%=	Modulus AND		c%=a	c = 5
**=	Exponent AND		c**=b	c = 9765625
//=	Floor Division AND		c//=b	c = 0

3. Comparison (Relational) Operators

Relational operators are used to compare two values. We can understand that whether two values are same or which one is bigger or which one is lesser etc. These operator will result in True or False depending on the value compared as shown below:

(assume a = 1 and b = 2)

Symbol	Operator Name	Description	Example	Result
==	Double Equal	If the two value of its operands are equal, then the condition becomes true, otherwise false	a==b	False
!= or <>	Not Equal To	If two operands values are not equal, then condition becomes true. Both the operators define the same meaning and function	a != b	True
>	Greater Than	If the value of the left-hand operand is greater than the value of right-hand operand, the condition becomes true.	a>b	False
<	Less Than	If the value of the left-hand operand is less than the value of right operand, then condition becomes true.	a<b	True
<=	Less Than Equal To	If the value of the left-hand operand is less than or equal to the value of right-hand operand, the condition becomes true.	a<=b	True
>=	Greater Than Equal To	If the value of the left-hand operand is greater than or equal to the value of right-	a>=b	False

		hand operand, the condition becomes true.		

4. Logical Operator

Symbol	Operator Name	Description
or	Logical OR	If any of the two operands are non-zero, then the condition is true.
and	Logical AND	If both the operands are true, then the condition is true.
not	Logical NOT	It is used to reverse the logical state of its operand.

5. Bitwise Operators

Bitwise Python operators process the individual bits of integer values. They treat them as sequences of binary bits.

We can use bitwise operators to check whether a particular bit is set. For example, IoT applications read data from the sensors based on a specific bit is set or not. In such a situation, these operators can help.

Symbol	Operator Name	Description
&	Binary AND	This operator copies the bit to the result if it exists in both operands.
|	Binary OR	This operator copies the bit if it exists in either of the operands.

^	Binary XOR	This operator copies the bit if it is set in one operand but not both.
~	Binary 1s Complement	This is a unary operator and has the ability of 'flipping' bits.
<<	Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand using this operator.
>>	Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand using this operator.

Example-

Let's consider the numbers 4 and 6 whose binary representations are '00000100' and '00000110'. Now, we'll perform the AND operation on these numbers.

```
a=4
b=6

#Bitwise AND: The result of 'a & b' is 4
print('a & b is',a & b)
```

Output-

```
a & b is 4
```

The above result is the outcome of following AND ('&') operation.

```
0 0 0 0 0 1 0 0 &
0 0 0 0 0 1 1 0
-----------------
0 0 0 0 0 1 0 0 (the binary representation of the number 4)
```

6. Identity Operator

These operator compare the memory locations of two objects. Hence, it is possible to know whether the two objects are same or not. The memory location of an object can be seen using the id() function. This function returns an integer number, called the identity number that internally represents the

memory location of the object. For example id(a) gives the identity of the object referred by the name 'a'. See the following statements:

a = 18

b = 18

In the first statement, we are assigning the name(or identifier) 'a' to the object 18. In the second statement, we are assigning another name 'b' to the same object 18. In Python, everything is considered as an object. Here, 18 is the object for which two names are given. If we display an identity number of these two variables, we will get same numbers as they refer to the same object.

id(a)

1567895490

id(b)

1567895490

There are two identity operators:

Symbol	Operator Name	Description
Is	**is**	The result becomes true if values on either side of the operator point to the same object and False otherwise.
is not	**is not**	The result becomes False if the variables on either side of the operator point to the same object

Example-

```
# Using 'is' identity operator
a = 7
if (type(a) is int):
 print("true")
else:
 print("false")
```

```
str = 'Python operators'
dict = {6:'June',12:'Dec'}
print('P' in str)
print('Python' in str)
print('python' not in str)
print(6 in dict)
print('Dec' not  in dict)
```

Output-

```
True
True
True
True
False
```

Let's take a group of strings in the list. We want to display the members of the list using a for loop where the 'in' operator is used. The list of names is given below:

names = ["Akshay", "Anil", "Sanjay", "Saloni", "Sakshi"]

Suppose we want to retrieve all the names from the list, we can use for loop:

```
for name in names: print(name)
```

Output

```
Akshay
Anil
Sanjjay,
Saloni
Saksay,
Saloni
Sakshi
```

In the for loop, the variable 'name' is used. This variable will store each and every value of the list 'names'. It means, 'name' values will be 'Akshay', 'Anil', etc. The operator 'in' will check whether each of these names is a member of the list or not. For example, 'Akshay' is a member of the list and hence, 'in' will return True. When 'True' is returned, the print() function in the for loop is executed and that name is displayed. In this way, all the names of the list are displayed.

Let's take another example where we want to display the elements of a dictionary. The dictionary 'dob' (date of birth) contains the names of the student and their date of birth. The student name become a key and the pin code will become its value. For example,

dob = {'Akshay': '21 Jan, 1998', 'Anil': '18 Mar, 1997', 'Saloni' : '15 May, 1998'}

Now, we want to retrieve each key and its corresponding value. For this purpose, we take a variable 'd' in for loop. Every time in the for loop 'd' will get the name of the student. To get the corresponding value, we can use dob[d]. So, the following for loop will display the student name and their date of birth from 'dob' dictionary:

```
for  d in dob: print(d, dob[d])
```

Output:

```
Akshay 21 Jan 1998
Anil 18 Mar, 1997
Saloni 15 May, 1998
```

Naming Styles (as per PEP-8)

The table below outlines some of the common naming styles in Python code and when you should use them:

Type	Naming Convention	Examples
Function	Use a lowercase word or words. Separate words by underscores to improve readability.	function, my_function
Variable	Use a lowercase single letter, word, or words. Separate words with underscores to improve readability.	x, var, my_variable
Class	Start each word with a capital letter. Do not separate words with underscores. This style is called camel case.	Model, MyClass
Method	Use a lowercase word or words.	class_method, method

	Separate words with underscores to improve readability.	
Constant	Use an uppercase single letter, word, or words. Separate words with underscores to improve readability.	CONSTANT, MY_CONSTANT, MY_LONG_CONSTANT
Module	Use a short, lowercase word or words. Separate words with underscores to improve readability.	module.py, my_module.py
Package	Use a short, lowercase word or words. Do not separate words with underscores.	package, mypackage