UNIT - I

Syllabus:

Python Basics, Objects- Python Objects, Standard Types, Other Built-in Types, Internal Types, Standard Type Operators, Standard Type Built-in Functions, Categorizing the Standard Types, Unsupported Types
Numbers - Introduction to Numbers, Integers, Floating Point Real Numbers, Complex Numbers, Operators, Built-in Functions, Related Modules
Sequences - Strings, Lists, and Tuples, Mapping and Set Types

---

Python Basics:

1. Statements & Syntax
2. Variable Assignment
3. Identifiers & Keywords
4. Basic Style Guidelines
5. Memory Management
6. Basic Python Program

1. Statements & Syntax:

Some rules and certain symbols are used with regard to statements in Python:

- **Hash mark (#)** indicates Python comments - A comment can begin anywhere on a line. All characters following the # to the end of the line are ignored by the interpreter.

- **NEWLINE (\n)** is the standard line separator (one statement per line)

- **Backslash (\ )** continues a line - Single statements can be broken up into multiple lines by use of the backslash. The backslash symbol (\ ) can be placed before a NEWLINE to continue the current statement onto the next line.

- **Semicolon (;)** joins two statements on a line - The semicolon (;) allows multiple statements on a single line given that neither statement starts a new code block. Here is a sample snip using the semicolon:

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

- **Colon (:)** separates a header line from its suite - Compound or complex statements, such as **if**, **while**, **def**, and **class**, are those that require a header line and a suite. Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines that make up the suite.

- **Suites delimited via indentation** - Python employs indentation as a means of delimiting blocks of code. Code at inner levels are indented via spaces or tabs. Indentation requires exact indentation; in other words, all the lines of code in a suite must be indented at the exact same level (e.g., same number of spaces). Indented lines starting at different positions or column numbers are not allowed; each line would be considered part of another suite and would more than likely result in syntax errors.

- **Python files organized as modules** - Each Python script is considered a module. Modules have a physical presence as disk files. When a module gets large enough or has diverse enough functionality, it may make sense to move some of the code out to another module.

2. Variable Assignment:

1. Assignment Operator
2. Augmented Assignment
3. Multiple Assignment
4. Multuple Assignment

Assignment Operator:

The equal sign (=) is the main Python assignment operator. assignment does not explicitly assign a value to a variable, although it may appear that way from your experience with other programming languages. In Python, objects are referenced, so on assignment, a reference (not a value) to an object is what is being assigned, whether the object was just created or was a pre-existing object.

Example:

a=20 (assigning an integer value 20 to the object a)
str="college" (assigning the string value "college" to the object str)
b=26.30 (assigning an float value 26.30 to the object b)
c="JBREC' + "College"

Augmented Assignment:

The equal sign can be combined with an arithmetic operation and the resulting value reassigned to the existing variable is known as augmented assignment.

Example:

x = x + 1 (or) x+=1

Augmented assignment refers to the use of operators, which imply both an arithmetic operation as well as an assignment. (+=, -=, *=, /=, %=, **=, //=, &=, >>=, <<=)

Python does not support pre-/post-increment nor pre-/post-decrement operators such as x++ or --x.

Multiple Assignment:

This is the process of assigning a single object to multiple variables.

Example:

```
x = y = z = 1
```

In the above example, an integer object (with the value 1) is created, and x, y, and z are all assigned the same reference to that object.

Multuple Assignment:

This is another way of assigning multiple variables. This is not an official Python term, but we use "multuple" here because when assigning variables this way, the objects on both sides of the equal sign are tuples.

Example:

```
x, y, z = 12.5, 200, "College"
```

In the above example, two integer objects (with values 1 and 2) and one string object are assigned to x, y, and z respectively. Parentheses are normally used to denote tuples, and although they are optional, we recommend them anywhere they make the code easier to read:

```
(x, y, z) = (1, 2, 'a string')
```

3. Identifiers & Keywords:

Identifiers:

The rules for Python identifier strings are like most other high-level programming languages.

- First character must be a letter or underscore (_)
- Any additional characters can be alphanumeric or underscore
- Case-sensitive

No identifiers can begin with a number, and no symbols other than the underscore are ever allowed. The easiest way to deal with underscores is to consider them as alphabetic characters.

Keywords:

- **Keywords** are the reserved words in **Python**.
- We cannot **use** a **keyword** as a variable name, function name or any other identifier.
- They are used to define the syntax and structure of the **Python** language.
- In **Python**, **keywords** are case sensitive.

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

4. Basic Style Guidelines:

a. **Comments**
b. **Documentation**
c. **Indentation**
d. **Choosing Identifiers Names**

a. Comments:

In Python, there are two ways to annotate the code.
The first is to include comments that detail or indicate what a section of code – or snippet – does. The second makes use of multi-line comments or paragraphs that serve as documentation for others reading your code.

Single-line comments are created simply by beginning a line with the hash (#) character, and they are automatically terminated by the end of line.
Example: # This is single line comment

Multiline comments are created by adding a delimiter (“”””) on each end of the comment.
Example:

""" Multi line comment
Created in Python """

b. Documentation:

Python provides a mechanism whereby documentation strings can be retrieved dynamically through the __doc__ special variable. The first unassigned string in a module, class declaration, or function declaration can be accessed using the attribute *obj*.__doc__ where *obj* is the module, class, or function name.

c. Indentation:

Indentation refers to the spaces at the beginning of a code line. Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important. Python uses indentation to indicate a block of code.

Example:

```
if 5 > 2:
        print("Five is greater than two!")
```

d. Choosing Identifiers Name:

Identifiers are names given to identify something. There are some rules you have to follow for naming identifiers:

- The first character of the identifier must be a letter of the alphabet (upper or lowercase) or an underscore ('_').
- The rest of the identifier name can consist of letters (upper or lowercase), underscores ('_') or digits (0-9).
- Identifier names are case-sensitive. For example, myname and myName are **not** the same. Note the lowercase n in the former and the uppercase N in te latter.
- Examples of valid identifier names are i, __my_name, name_23 and a1b2_c3.
- Examples of invalid identifier names are 2things, this is spaced out and my-name.

5. Memory Management:

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the Python memory manager. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

a. Variable Declarations (or Lack Thereof)
b. Dynamic Typing
c. Memory Allocation
d. Reference Counting
e. Garbage Collection

a. Variable Declarations:

In most compiled languages, variables must be declared before they are used. In fact, C is even more restrictive: variables have to be declared at the beginning of a code block and before any statements are given. Other languages, like C++ and Java, allow "on-the-fly" declarations, i.e., those which occur in the middle of a body of code but these name and type declarations are

still required before the variables can be used. In Python, there are no explicit variable declarations. Variables are "declared" on first assignment. Like most languages, however, variables cannot be accessed until they are (created and) assigned:

Example:

Once a variable has been assigned, you can access it by using its name:

```
>>> x = 4
>>> y = 'this is a string'
>>> x
      4
>>> y
    'this is a string
```

b. Dynamic Typing:

In Python, the type and memory space for an object are determined and allocated at runtime. Although code is byte-compiled, Python is still an interpreted language. On creation that is, on assignment the interpreter creates an object whose type is dictated by the syntax that is used for the operand on the right-hand side of an assignment. After the object is created, a reference to that object is assigned to the variable on the left-hand side of the assignment.

Example:

```
# This will store 6 in the memory and binds the name x to it. After it runs, type of x will
   be int.

x = 6
print(type(x))

# This will store 'hello' at some location int the memory and binds name x to it. After it
  runs type of x will be str.

x = 'hello'
print(type(x))
```

Output:

```
<class 'int'>
<class 'str'>
```

c. Memory Allocation:

The good thing about Python is that everything in Python is an object. This means that Dynamic Memory Allocation underlies Python Memory Management. When objects are no longer needed, the Python Memory Manager will automatically reclaim memory from them.

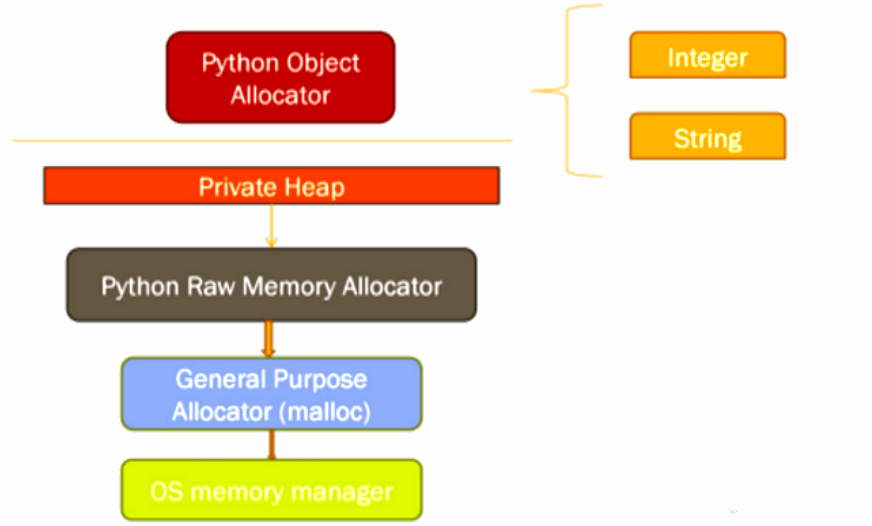
The Python memory manager manages Python's memory allocations. There's a private heap that contains all Python objects and data structures. The Python memory manager manages the Python heap on demand. The Python memory manager has object-specific allocators to allocate memory distinctly for specific objects such as int, string, etc… Below that, the raw

memory allocator interacts with the memory manager of the operating system to ensure that there's space on the private heap.

The Python memory manager manages chunks of memory called "Blocks". A collection of blocks of the same size makes up the "Pool". Pools are created on Arenas, chunks of 256kB memory allocated on heap=64 pools. If the objects get destroyed, the memory manager fills this space with a new object of the same size.

Objects and instance variables are created in Heap memory. As soon as the variables and functions are returned, dead objects will be garbage collected.

It is important to note that the Python memory manager doesn't necessarily release the memory back to the Operating System, instead memory is returned back to the python interpreter. Python has a small objects allocator that keeps memory allocated for further use. In long-running processes, you may have an incremental reserve of unused memory.



d. Reference Counting:

Reference counting is a simple technique in which objects are deallocated when there is no reference to them in a program.

Every variable in Python is a reference (a pointer) to an object and not the actual value itself. For example, the assignment statement just adds a new reference to the right-hand side. A single object can have many references (variable names).

Example:

This code creates two references to a single object:

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

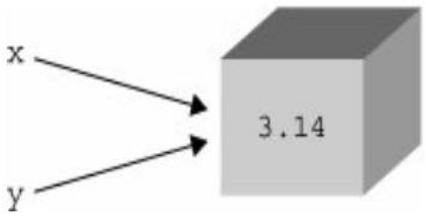


Fig: An Object with two references.

An assignment statement itself (everything on the left) never copies or creates new data. To keep track of references, every object (even integer) has an extra field called reference count that is increased or decreased when a pointer to the object is created or deleted.

Examples, where the reference count increases:

- assignment operator
- argument passing
- appending an object to a list (object's reference count will be increased).

If the reference counting field reaches zero, Python automatically calls the object-specific memory deallocation function. If an object contains references to other objects, then their reference count is automatically decremented too. Thus other objects may be deallocated in turn. For example, when a list is deleted, the reference count for all its items is decreased. If another variable references an item in a list, the item won't be deallocated.

e. Garbage Collection:

Python deletes unwanted objects (built-in types or class instances) automatically to free the memory space. The process by which Python periodically frees and reclaims blocks of memory that no longer are in use is called Garbage Collection.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.

An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary). The object's reference count decreases when it's deleted with del, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically.

Example:

```
a = 40      # Create object <40>
b = a       # Increase ref. count  of <40>
c = [b]     # Increase ref. count  of <40>
del a       # Decrease ref. count  of <40>
b = 100     # Decrease ref. count  of <40>
c[0] = -1   # Decrease ref. count  of <40>
```

6. Basic Python Program

Python interpreter treats a piece of text terminated by hard carriage return (new line character) as one statement. It means each line in a Python script is a statement.

Example:

```
msg="Hello World"
code=123
name="Python"
```

Text spread over more than one lines to be a single statement by using the backslash (\) as a continuation character.

Example:

```
msg="Hello World \
        Welcome to Python Programming \
        Multi Line format "
```

use the semicolon ; to write multiple statements in a single line.

Example:

```
msg="Hello World";code=123;name="Python"
```

Python uses uniform indentation to denote a block of statements. When a block is to be started, type the colon symbol (:) and press Enter. Any Python-aware editor (like IDLE) goes to the next line leaving an additional whitespace (called indent). Subsequent statements in the block follow the same level of indent. In order to signal the end of a block, the whitespace is de-dented by pressing the backspace key. If your editor is not configured for Python, you may have to ensure that the statements in a block have the same indentation level by pressing the spacebar or Tab key. The Python interpreter will throw an error if the indentation level in the block is not same.

Example:

```
if True:
        print ("Answer")
        print ("True")

else:
        print "(Answer")
        print ("False")
```

input() function is a part of the core library of standard Python distribution. input() function always reads the input as a string, even if comprises of digits.

Example:

```
name=input("Enter your name: ")
```

To read the input as integer or float or complex values, we have to make typecasting to the input which is read from the user.

Example:

```
value=int(input("Enter the value: "))
```

print() serves as an output statement in Python. It echoes the value of any Python expression on the Python shell.

Example:

```
print("Hello World")
```

Multiple values can be displayed by the single print() function separated by comma.

Example:

```
name="Ram"
age=21
print("Name:",name,"Age:",age)
```

By default, a single space ' ' acts as a separator between values. However, any other character can be used by providing a sep parameter. In the following example, (,) is used as a separator character.

Example:

```
name="Ram"
 age=21
print(name,age)
```

(or)

```
print(name,age,sep=",")
```

Example Program:

```
import math
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))
num3 = float(input("Enter third number: "))

if (num1 >= num2) and (num1 >= num3):
        largest = num1
elif (num2 >= num1) and (num2 >= num3):
         largest = num2
else:
         largest = num3
print("The largest number between",num1,",",num2,"and",num3,"is",largest)
```

PYTHON OBJECTS:

Python uses the object model abstraction for data storage. Any construct that contains any type of value is an object.

All Python objects have the following three characteristics:

1. Identity,
2. Type, and
3. Value.

Identity:

Unique identifier that differentiates an object from all others. Any object's identifier can be obtained using the id() built-in function (BIF). This value is as close as you will get to a "memory address" in Python.

Type:

An object's type indicates what kind of values an object can hold, what operations can be applied to such objects, and what behavioral rules these objects are subject to. We can use the type() Built in Function to reveal the type of a Python object.

Value:

Data item that is represented by an object.

PYTHON STANDARD TYPES:

Python standard types are referred as "primitive data types" because these types represent the primitive data types that Python provides.

- Numbers
  - ✓ Integer
  - ✓ Floating Point Real Number
  - ✓ Complex Number
- String
- List
- Tuple
- Dictionary

OTHER BUIT IN TYPES:

- Type
- Null Object
- File
- Set / Frozen Set
- Function / Method
- Module
- Class

Type:

Type built in function is used to find out the type of an object by calling type() with that object.

Example:

```
type(42)
```

O/P: <type 'int'>

```
a=40.5
print(type(a))
```

O/P: <type 'float>

NULL Object:

Python has a special type known as the Null object or NoneType. It has only one value, None. The type of None is NoneType. It does not have any operators or BIFs.

All standard type objects can be tested for truth value and compared to objects of the same type. Objects have inherent True or False values. Objects take a False value when they are empty, any numeric representation of zero, or the Null object None.

The following are defined as having false values in Python:

- None
- False (Boolean)
- 0 (integer)
- 0.0 (float)
- 0L (long integer)
- 0.0+0.0j (complex)

- " " (empty string)
- [ ] (empty list)
- ( ) (empty tuple)
- { } (empty dictionary)

Any value for an object other than those above is considered to have a true value, i.e., non-empty, non-zero, etc.

File:

A file is a named location on disk to store related information. We can access the stored information (non-volatile) after the program termination.

Files are treated in two modes as text or binary. The file may be in the text or binary format, and each line of a file is ended with the special character. Hence, a file operation can be done in the following order:

- Open a file
- Read or write - Performing operation
- Close the file

Set / Frozen Set:

Set: A Python set is the collection of the unordered items. Each element in the set must be unique, immutable, and the sets remove the duplicate elements. Sets are mutable which means we can modify it after its creation.

The set can be created by enclosing the comma-separated immutable items with the curly braces {}. Python also provides the set() method, which can be used to create the set by the passed sequence.

Example:

```
Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}
print(Days)
```

Frozen Set: Frozen set is just an immutable version of a Python set object. While elements of a set can be modified at any time, elements of the frozen set remain the same after creation.

A frozen set in Python is a set whose values cannot be modified. This means that it is immutable, unlike a normal set. Frozen sets help serve as a key in dictionary key-value pairs.

Frozen sets can be obtained using the frozenset() method. This function takes any iterable items and converts it to immutable.

Example:

```
a={1, 2,5, 4.6, 7.8, 'r', 's'}
b=frozenset(a)
print(b)
```

Function / Method:

Function:

A function is a block of code to carry out a specific task, will contain its own scope and is called by name. All functions may contain zero(no) arguments or more than one arguments. On exit, a function can or cannot return one or more values.

Syntax:

```
def functionName( arg1, arg2,….):
        …….
        # Function_body
        ……..
```

Example:

```
def sum(num1, num2):
        return (num1 + num2)
```

The function allows us to implement code reusability. There are three kinds of functions:

- **Built-in functions:** As the name suggests, these functions come with the Python language, for example, help() to ask for any help, max()- to get maximum value, type()- to return the type of an object and many more.
- **User-defined functions:** These are the functions created by the user using **def** keyword.
- **Anonymous Functions:** These functions are created by using **lambda** keyword.

Method:

A method in python is somewhat similar to a function, except it is associated with object/classes. Methods in python are very similar to functions except for two major differences.

- The method is implicitly used for an object for which it is called.
- The method is accessible to data that is contained within the class.

Syntax:

```
class ClassName:
        def method_name():
                ……..
                # Method_body
                ……
```

Example:

```
class Book(object):
        def my_method(self):
                print("Python Programming By Wesley")
Python = Book()
Book.my_method()
```

Module:

Modules in Python are simply Python files with a .py extension. The name of the module will be the name of the file. A Python module can have a set of functions, classes or variables defined and implemented.

Creating a Module:

module name: name.py

```
def greeting(name):
        print("Hello, "+ name)
```

Using Module:

```
import name
name.greeting("JBREC")
```

Class:

Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Syntax:

```
class ClassName:
        <statement-1>
        .
        .
        .
        <statement-N>
```

Example:

```
class Complex:
        def __init__(self, realpart, imagpart):
            self.r = realpart
            self.i = imagpart
x = Complex(3.0, -4.5)
x.r, x.i
```

INTERNAL TYPES:

Internal Python objects are hard to use in general and even the programmer will not directly interact with these types. The following are different internal types associated with python:

- Code Objects
- Frame Objects
- Traceback Objects
- Slice Objects
- Ellipsis Objects
- Xrange Objects

Code Objects:

Code objects are executable pieces of Python source that are byte-compiled, usually as return values from calling the compile() built in function. Such objects are appropriate for execution by either exec or by the eval() built in function. Code objects themselves do not contain any information regarding their execution environment, but they are at the heart of every user-defined function, all of which do contain some execution context. Along with the code object, a function's attributes also consist of the administrative support that a function requires, including its name, documentation string, default arguments, and global namespace.

Frame Objects:

These are objects representing execution stack frames in Python. Frame objects contain all the information the Python interpreter needs to know during a runtime execution environment. Some of its attributes include a link to the previous stack frame, the code object that is being executed, dictionaries for the local and global namespaces, and the current instruction.

Traceback Objects:

When you make an error in Python, an exception is raised. If exceptions are not caught or "handled," the interpreter exits with some diagnostic information similar to the output shown below:

```
Traceback (innermost last):
        File "<stdin>", line N?, in ???
ErrorName: error reason
```

The traceback object is just a data item that holds the stack trace information for an exception and is created when an exception occurs. If a handler is provided for an exception, this handler is given access to the traceback object.

Slice Objects:

Slice objects are created using the Python extended slice syntax. This extended syntax allows for different types of indexing. These various types of indexing include stride indexing, multi-dimensional indexing, and indexing using the Ellipsis type. The syntax for multi-dimensional indexing is sequence[start1 : end1, start2 : end2], or using the ellipsis, sequence [..., start1 : end1]. Slice objects can also be generated by the slice() Built in function. Stride indexing for sequence types allows for a third slice element that allows for "step"-like access with a syntax of sequence[starting_index : ending_index : stride].

Ellipsis Objects:

Ellipsis objects are used in extended slice notations as demonstrated above. These objects are used to represent the actual ellipses in the slice syntax. Like the Null object None, ellipsis objects also have a single name, Ellipsis, and have a Boolean True value at all times.

XRange Objects:

XRange objects are created by the built in function xrange(), and similar that of the range() Built in function, and used when memory is limited and when range() generates an unusually large data set.

STANDARD TYPE OPERATORS

- Object Value Comparison Operator
- Object Identity Comparison Operator
- Boolean Operator

Object Value Comparison Operator:

Comparison operators are used to determine equality of two data values between members of the same type. These comparison operators are supported for all built-in types. Comparisons yield Boolean True or False values, based on the validity of the comparison expression.

S.No	Operator	Function	Example
1	= =	expr1 = = expr2 expr1 is equal to expr2	2= =2 True
2	!=	expr1 != expr2 expr1 is not equal to expr2	2!=2 False
3	< >	expr1 != expr2 expr1 is not equal to expr2	2!=2 False
4	>	expr1 > expr2 expr1 greater than expr2	2 > 3 False
5	<	expr1 < expr2 expr1 less than expr2	2 < 3 True
6	>=	expr1 >= expr2 expr1 is greater than or equal to expr2	4>=4 True
7	<=	expr1<=expr2 expr1 is less than or equal to expr2	2<=3 True

Object Identity Comparison Operator:

Python supports the notion of directly comparing objects themselves. Objects can be assigned to other variables (by reference). Because each variable points to the same (shared) data object, any change effected through one variable will change the object and hence be reflected through all references to the same object.

Example: "a & b reference the same object"

```
a=b=10
```

The above statement from the value point of view, it appears that you are performing a multiple assignment and assigning the numeric value of 4.3 to both the 'a' and 'b' variables.

Example: "a & b reference the same object"

a=15

b=a

A numeric object with value '15' is created, then assigned to one variable. When b = a occurs, 'b' is directed to the same object as 'a' since Python deals with objects by passing references. 'b' then becomes a new and additional reference for the original value. So both 'a' and 'b' now point to the same object.

Example: "a & b reference the different object"

a=20

b=12+8

In this example, First, a numeric object is created, then assigned to 'a'. Then a second numeric object is created, and this time assigned to 'b'. Although both objects are storing the exact same value, there are indeed two distinct objects in the system, with 'a' pointing to the first, and 'b' being a reference to the second.

S.No	Operator	Function	Example
1	is	obj1 **is** obj2	a = [5, 'hat', -9.3] b = a a **is** b O/p: True
2	is not	obj1 **is not** obj2	a = [5, 'hat', -9.3] b = a a **is not** b O/p: False

Boolean Operator:

Expressions may be linked together or negated using the Boolean logical operators **and, or**, and **not**, all of which are Python keywords. The **not** operator has the highest precedence and is immediately one level below all the comparison operators. The **and** and **or** operators follow, respectively.

S.No	Operator	Function	Example
1	not	not expr Logical NOT of expr (negation)	x, y = 3.1415926536, -1024 x < 5.0 O/p: True
2	and	expr1 and expr2 Logical AND of expr1 and expr2 (Conjunction)	x, y = 3.1415926536, -1024 (x < 5.0) **or** (y > 2.718281828) O/p: True
3	or	expr1 or expr2 Logical OR of expr1 and expr2 (Disjunction)	x, y = 3.1415926536, -1024 (x < 5.0) **and** (y > 2.718281828) O/p: True

STANDARD TYPE BUILT-IN FUNCTIONS

Python provides some built in functions that can be applied to all the basic object types:

- type()
- cmp()
- repr()
- str()

type():

The type() is an in built function which takes an object and returns its type. The return value is a type object.

Syntax:

type(object)

Example:

#int type	**#String Type**	**#Type Type**
type(4)	type("Hello World")	type(type(4))
O/p: <type 'int'>	O/p: <type 'string'>	O/p: <type 'type'>

In the examples above, we considered an integer and a string and obtain their types using the type() Built In Function; in order to also verify that types themselves are types, we call type() on the output of a type() call.

cmp():

The cmp() Built In Function compares two objects, say, obj1 and obj2.

This returns,

a negative number (integer) if obj1is less than obj2,

a positive number if obj1 is greater than obj2, and

zero if obj1 is equal to obj2.

Example:

a, b = - 4, 12	a,b = 10, 20	a,b = 20, 20
cmp(a,b)	cmp(b,a)	cmp(a,b)
O/p: -1	O/p: 1	O/p: 0

repr():

The repr() function returns a printable representation of the given object. It takes a single parameter.

Syntax:

repr(obj)

Example:

```
var="College"
print(repr(var))
```

O/p: 'College'

str():

The str() function returns the string version of the given object.

Syntax:

```
str(object, encoding='utf-8', errors='strict')
```

- **object** - The object whose string representation is to be returned. If not provided, returns the empty string
- **encoding** - Encoding of the given object. Defaults of **UTF-8** when not provided.
- **errors** - Response when decoding fails. Defaults to 'strict'.

There are six types of errors:

- **strict** - default response which raises a UnicodeDecodeError exception on failure
- **ignore** - ignores the unencodable Unicode from the result
- **replace** - replaces the unencodable Unicode to a question mark
- **xmlcharrefreplace** - inserts XML character reference instead of unencodable Unicode
- **backslashreplace** - inserts a \uNNNN espace sequence instead of unencodable Unicode
- **namereplace** - inserts a \N{...} escape sequence instead of unencodable Unicode

Example:

```
re=str(10)
print(re)
```
O/p: 10

```
re=str("college")
print(re)
```
O/p: college

```
re=  str(b'Python!')
print(re)
```
O/p: b'Python!'

CATEGORIZING THE STANDARD TYPES

The following are the three different models used to categorize the standard types. These models helps us to obtain the better understanding of how the types are related.

1. Storage Model
2. Update Model
3. Access Model

Storage Model:

Categorizing the types can be based on the storage model, i.e., how many objects can be stored in an object of this type. Python can hold either single or multiple values.

Atomic / Scalar Storage - A type which holds a single literal object.

Container Storage - A type which can hold multiple objects.

S.No	Storage Model Category	Python Type
1	Scalar / Atom	Numbers (all numeric types), Strings (all are literals)
2	Container	Lists, Tuples, Dictionaries.

Update Model:

Another way of categorizing the standard types is based on whether the object once created can be changed or not. Python allows certain types to update the values and some types do not allow to update the values.

Mutable Objects - Objects whose values can be changed.

Immutable Objects - Objects whose values cannot be chnaged.

S.No	Update Model Category	Python Type
1	Mutable	Lists, dictionaries
2	Immutable	Numbers, strings, tuples

Access Model:

Accessing the values of the stored data can be done based on the access model. There are three categories under the access models:

1. Direct
2. Sequence
3. Mapping

Direct:

Direct types indicate single-element, non-container types. All numeric types fit into this category.

Sequence:

Sequence types are those whose elements are sequentially accessible via index values starting at 0. Accessed items can be either single elements or in groups, better known as slices. Types that fall into this category include strings, lists, and tuples.

Mapping:

Mapping types are similar to the indexing properties of sequences, except instead of indexing on a sequential numeric offset, elements (values) are unordered and accessed with a key, thus making mapping types a set of hashed key-value pairs.

S.No	Access Model Category	Python Type
1	Direct	Numbers
2	Sequence	Strings, Lists, Tuples
3	Mapping	Dictionaries

The following table shows all the standard types, the three different models we use for categorization, and where each type fits into these models.

S.No	Data Type	Storage Model	Update Model	Access Model
1	Numbers	Scalar	Immutable	Direct
2	Strings	Scalar	Immutable	Sequence
3	Lists	Container	Mutable	Sequence
4	Tuples	Container	Immutable	Sequence
5	Dictionaries	Container	Mutable	Mapping

UNSUPPORTED TYPES

Unsupported types are nothing but the types that are not supported by python. The following are the list of types that are not supported by python:

1. char or byte
2. Pointer
3. int Vs short Vs long
4. float Vs double

char or byte:

Python does not have a char or byte type to hold either single character or 8-bit integers. Use strings of length one for characters and integers for 8-bit numbers.

Pointer:

Python manages memory automatically, there is no need to access pointer addresses. The closest to an address that you can get in Python is by looking at an object's identity using the id() built in function. Since we don't have control over this value, it's a moot point.

int Vs short Vs long:

Python's plain integers are the universal "standard" integer type, obviating the need for three different integer types, e.g., C's int, short, and long. Only need to use a single type, the Python integer. Even when the size of an integer is exceed, for example, multiplying two very large numbers, Python automatically gives you a long back instead of overflowing with an error.

float Vs double:

C has both a single precision float type and double-precision double type. Python's float type is actually a C double. Python does not support a single-precision floating point type because its benefits are outweighed by the overhead required to support two types of floating point types. For those wanting more accuracy and willing to give up a wider range of numbers, Python has a decimal floating point number too, but you have to import the decimal module to use the Decimal type. Floats are always estimations. Decimals are exact and arbitrary precision. Decimals make sense concerning things like money where the values are exact. Floats make sense for things that are estimates anyway, such as weights, lengths, and other measurements.

STRINGS

Python string is the collection of the characters surrounded by single quotes (' '), double quotes (" "), or triple quotes ('" "').

Creating String:

String can be created by enclosing the characters in single-quotes or double- quotes. Python also provides triple-quotes to represent the string, but it is generally used for multiline string or **docstrings**.

Example:

```
#Using single quotes
        str1 = 'Hello Python'
        print(str1)
#Using double quotes
        str2 = "Hello Python"
        print(str2)

#Using triple quotes
        str3 = '''Triple quotes are generally used for
                  represent the multiline or
                 docstring'''
        print(str3)
```

O/P:

```
Hello Python
Hello Python
Triple quotes are generally used for
    represent the multiline or
    docstring
```

Strings indexing and splitting:

The indexing of the Python strings starts from 0. For example, The string "HELLO" is indexed as given in the below figure.

str="HELLO"

H	E	L	L	O
0	**1**	**2**	**3**	**4**

str[0] = H
str[1] = E
str[2] = L
str[3] = L
str[4] = O

Example:

```
str="HELLO"
print(str[0])
print(str[1])
print(str[2])
print(str[3])
print(str[4])
print(str[7])     #This statement will return an Index error because 7th index does exists.
```

O/P:

```
H
E
L
L
O
IndexError: string index out of range
```

The **slice operator** [ ] is used to access the individual characters of the string. However, we can use the **colon (:)** operator in Python to access the substring from the given string. Consider the following example:

H	E	L	L	O
0	**1**	**2**	**3**	**4**

str[0] = H
str[1] = E
str[2] = L
str[3] = L
str[4] = O

str[:] = "HELLO"
str[0:] = "HELLO"
str[:5] = "HELLO"
str[:3] = "HEL"
str[0:2] = "HE"
str[1:4] = "ELL"

Example:

```
# Given String
        str = "JBREC COLLEGE"
# Start 0th index to end
        print(str[0:])
# Starts 1th index to 4th index
```

```
        print(str[1:5])
# Starts 2nd index to 3rd index
        print(str[2:4])
# Starts 0th to 2nd index
        print(str[:3])
#Starts 4th to 6th index
        print(str[4:7])
```

O/P:

```
JBREC COLLEGE
BREC
RE
JBR
C C
```

Negative Slicing:

Negative slicing can be performed over the string; it starts from the rightmost character, which is indicated as -1. The second rightmost index indicates -2, and so on.

H	E	L	L	O
-5	**-4**	**-3**	**-2**	**-1**

```
str[-1] = O          str[-3:-1] = "LL"
str[-2] = L          str[-4:-1] = "ELL"
str[-3] = L          str[-5:-3] = "HE"
str[-4] = E          str[-4:] = "ELLO"
str[-5] = H          str[::-1] = "OLLEH"
```

Example:

```
str = 'JBREC COLLEGE'
print(str[-1])
print(str[-3])
print(str[-2:])
print(str[-4:-1])
print(str[-7:-2])
# Reversing the given string
print(str[::-1])
print(str[-12])   #This statement will return an Index error because 12th index does exists.
```

O/P:

E
E
GE
LEG
COLLEG
EGELLOC CERBJ
IndexError: string index out of range

String Operators:

S.No	Operator	Description	Example
1	+	'+' operator is called concatenation operator used to join the strings given either side of the operator.	str='Hello' str1='World' print(str+str1) **o/p:** Hello World
2	*	'*' operator is called repetition operator. It concatenates the multiple copies of the same string.	str='Hello' print(str*3) **o/p:** HelloHelloHello
3	[ ]	'[ ]' operator is called as slice operator. It is used to access the sub-strings of a particular string.	str="Python" print(str[3] **o/p:** h
4	[:]	'[:]' operator is called range slice operator. It is used to access the characters from the specified range.	str="Python" print(str[1:4] **o/p:** yth
5	in	'in' is called as membership operator. It returns true if a particular sub-string is present in the specified string.	str="python" print('t' in str) **o/p:** True
6	not in	'not in' is called as membership operator. It returns true if a particular sub-string is not present in the specified string.	str="python" print('t' not in str) **o/p:** False
7	R / r	It is used to specify the raw string. Raw strings are used in the cases	print(r'C://python37')

		where we need to print the actual meaning of escape characters such as "C://python". To define any string as a raw string, the character r or R is followed by the string.	**o/p:** C://python37
8	%	It is used to perform string formatting. It makes use of the format specifiers %d or %f to map their values in python.	str="Hello" print("The string str : %s"%(str)) **o/p:** The string str : Hello

Python String Formatting

An escape sequence is a sequence of characters that does not represent itself when used inside a character or string literal, but is translated into another character or a sequence of characters that may be difficult or impossible to represent directly.

All escape sequences consist of two or more characters, the first of which is the backslash, \ (called the "Escape character"); the remaining characters determine the interpretation of the escape sequence. For example, \newline is an escape sequence that denotes a newline character.

Example:

```
# using triple quotes
print("""They said, "What's there?""")

# escaping single quotes
print('They said, "What\'s going on?"')

# escaping double quotes
print("They said, \"What's going on?\"")
```

O/P:

```
They said, "What's there?"
They said, "What's going on?"
They said, "What's going on?"
```

The list of an escape sequence is given below:

S.No	Escape Sequence	Description	Example
1	\newline	It ignores the new line.	print("Python1 \ Python2 \ Python3") **Output:** Python1 Python2 Python3
2	\\	Backslash	print("\\") **Output:** \
3	\'	Single Quotes (or) Double Quotes	print('\'') print("\"") **Output:** ' "
6	\b	ASCII Backspace(BS)	print("Hello \b World") **Output:** Hello World
7	\n	ASCII Linefeed	print("Hello \n World!") **Output:** Hello World!
8	\r	ASCII Carriege Return(CR)	print("Hello \r World!") **Output:** World!
9	\t	ASCII Horizontal Tab	print("Hello \t World!") **Output:** Hello World!
10	\v	ASCII Vertical Tab	print("Hello \v World!") **Output:** Hello World!
11	\ooo	Character with octal value	print("\110\145\154\154\157") **Output:** Hello
12	\xHH	Character with hex value.	print("\x48\x65\x6c\x6c\x6f") **Output:** Hello

format() method:

The **format()** method is the most flexible and useful method in formatting strings. The curly braces {} are used as the placeholder in the string and replaced by the **format()** method argument.

Example:

```
# Using Curly braces
print("{} and {} both are the best friend".format("Devansh","Abhishek"))

#Positional Argument
print("{1} and {0} best players ".format("Virat","Rohit"))

#Keyword Argument
print("{a},{b},{c}".format(a = "James", b = "Peter", c = "Ricky"))
```

Output:

```
Devansh and Abhishek both are the best friend
Rohit and Virat best players
James,Peter,Ricky
```

Python String Formatting Using % Operator:

Python provides an additional operator %, which is used as an interface between the format specifiers and their values. In other words, we can say that it binds the format specifiers to the values.

Example:

```
Integer = 10;
Float = 1.290
String = "Devansh"
print("Hi I am Integer ... My value is %d\nHi I am float ... My value is %f\nHi I am
      string ... My value is %s"%(Integer,Float,String))
```

Output:

```
Hi I am Integer ... My value is 10
Hi I am float ... My value is 1.290000
Hi I am string ... My value is Devansh
```

String functions:

S.No	Method	Description	Example
1	capitialize()	Converts the first character to upper case	txt = "hello, and welcome to my world." x = txt.capitalize() print (x) **O/P:** Hello, and welcome to my world.
2	casefold()	Converts string into lower case	txt = "Hello, And Welcome To My World!" x = txt.casefold() print(x) **O/P:** hello, and welcome to my world!
3	count()	Returns the number of times a specified value occurs in a string	txt = "I love apples, apple are my favorite fruit" x = txt.count("apple") print(x) **O/P:** 2
4	endswith()	Returns true if the string ends with the specified value	txt = "Hello, welcome to my world." x = txt.endswith(".") print(x) **O/P:** True
5	index()	Searches the string for a specified value and returns the position of where it was found	txt = "Hello, welcome to my world." x = txt.index("welcome") print(x) **O/P:** 7
6	isalpha()	Returns True if all characters in the string are in the alphabet	txt = "CompanyX" x = txt.isalpha() print(x) **O/P:** True
7	isdigit()	Returns True if all characters in the string are digits	txt = "50800" x = txt.isdigit() print(x) **O/P:** True

S.No	Method	Description	Example
8	islower()	Returns True if all characters in the string are lower case	txt = "hello world!" x = txt.islower() print(x) **O/P:** True
9	isupper()	Returns True if all characters in the string are upper case	txt = "THIS IS NOW!" x = txt.isupper() print(x) **O/P:** True
10	isnumeric()	Returns True if all characters in the string are numeric	txt = "565543" x = txt.isnumeric() print(x) **O/P:** True
11	lower()	Converts a string into lower case	txt = "Hello my FRIENDS" x = txt.lower() print(x) **O/P:** hello my friends
12	Upper()	Converts a string into upper case	txt = "Hello my friends" x = txt.upper() print(x) **O/P:** HELLO MY FRIENDS
13	strip()	Returns a trimmed version of the string	txt = ",,,,,rrttgg.....banana....rrr" x = txt.strip(",.grt") print(x) **O/P:** banana
14	rstrip()	Returns a right trim version of the string	txt = " banana " x = txt.rstrip() print("of all fruits", x, "is my favorite") **O/P:** Of all fruits banana is my favorite
15	swapcase()	Swaps cases, lower case becomes upper case and vice versa	txt = "Hello My Name Is PETER" x = txt.swapcase() print(x) **O/P:** hELLO mY nAME iS peter

LISTS

A list in Python is used to store the sequence of various types of data. Python lists are mutable type its mean we can modify its element after it created.

A list can be defined as a collection of values or items of different types. The items in the list are separated with the comma (,) and enclosed with the square brackets [ ].

Syntax:

li=[Item_1, Item_2, Item_3,....., Item_N]

Example:

fruits=[Apple,Grapes,Banana]

Characteristics of Lists:

Lists have the following Characteristics:

- The lists are ordered.
- The element of the list can access by index.
- The lists are mutable types.
- A list can store the number of various elements.

List Indexing and Splitting:

The elements of the list can be accessed by using the slice operator [].

The index starts from 0 and goes to length - 1. The first element of the list is stored at the 0^{th} index, the second element of the list is stored at the 1st index, and so on.

Example:

list=[0,1,2,3,4,5]

1	2	3	4	5
0	**1**	**2**	**3**	**4**

list[0] = 1

list[1] = 2

list[2] = 3

list[3] = 4

list[4] = 5

list[0:] = [1,2,3,4,5]

list[:] =[1,2,3,4,5]

list[2:4] = [3,4]

list[1:4] = [2,3,4]

list[:4] = [1,2,3,4]

We can get the **sub-list** of the list using the following syntax:

list_varible(start:stop:step)

- The start denotes the starting index position of the list.
- The stop denotes the last index position of the list.
- The step is used to skip the nth element within a start:stop

Example:

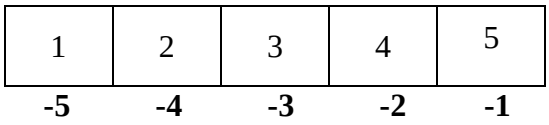
```
list = [1,2,3,4,5,6,7]
print(list[0])
print(list[1])
print(list[2])
print(list[3])
# Slicing the elements
print(list[0:6])
print(list[:])
print(list[2:5])
print(list[1:6:2])
```

Output:

```
1
2
3
4
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, 7]
[3, 4, 5]
[2, 4, 6]
```

Negative Indexing:

Python provides the flexibility to use the negative indexing also. The negative indices are counted from the right. The last element (rightmost) of the list has the index -1; its adjacent left element is present at the index -2 and so on until the left-most elements are encountered.



Example:

```
list = [1,2,3,4,5]
print(list[-1])
print(list[-3:])
print(list[:-1])
print(list[-3:-1])
```

Output:

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

Updating List Values:

since they are mutable, and their values can be updated by using the slice and assignment operator. Python also provides append() and insert() methods, which can be used to add values to the list.

Example:

```
list = [1, 2, 3, 4, 5, 6]
print(list)
# It will assign value to the value to the second index
list[2] = 10
print(list)
# Adding multiple-element
list[1:3] = [89, 78]
print(list)
# It will add value at the end of the list
list[-1] = 25
print(list)
```

Output:

[1, 2, 3, 4, 5, 6]
[1, 2, 10, 4, 5, 6]
[1, 89, 78, 4, 5, 6]
[1, 89, 78, 4, 5, 25]

Deleting List Elements:

The list elements can also be deleted by using the **del** keyword. Python also provides us the **remove(), pop()** methods if we do not know which element is to be deleted from the list.

del[a : b] - This method **deletes all the elements in range** starting from index 'a' till 'b' mentioned in arguments.

List Operators:

- **Standard Type Operator**
- **Sequence Type Operator**
- **Membership Operator**
- **Concatenation Operator**
- **Repetition Operator**

Standard Type Operator:

The comparison operators is same as the cmp() built-in function. This basically works like : the elements of both lists are compared until there is a determination of a winner.

Example:

```
>>> list1 = ['abc', 123]
>>> list2 = ['xyz', 789]
>>> list3 = ['abc', 123]
>>> 1ist1 < list2
```

Output: True

```
>>> list2 < list3
```

Output: False

```
>>> list2 > list3 and list1 == list3
```

Output: True

Sequence Type Operator:

slices of lists pull out an object or a group of objects that are elements of the list operated on. Slicing operators obey the same rules regarding positive and negative indexes, starting and ending indexes, as well as missing indexes, which default to the beginning or to the end of a sequence.

Example:

```
list = [1,2,3,4,5,6,7]
print(list[0])
print(list[1])
print(list[2])
print(list[3])
# Slicing the elements
print(list[0:6])
# By default the index value is 0 so its starts from the 0th element and go for index -1.
print(list[:])
print(list[2:5])
print(list[1:6:2])
```

Output:

```
1
2
3
4
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, 7]
[3, 4, 5]
[2, 4, 6]
```

Membership Operator:

Membership Operators are **in** and **not in**. With lists , we can check whether an object is a member of a list or not.

Example:

```
l1 = [1, 2, 3, 4]
l2 = [5, 6, 7, 8]
print(2 in l1)
```

Output: True

```
Print(6 not in l2)
```

Output: False

Concatenation Operator:

The concatenation operator allows us to join multiple lists together. you can concatenate only objects of the same type. You cannot concatenate two different types even if both are sequences. **'+'** is used as concatenation operator.

Example:

```
l1 = [1, 2, 3, 4]
l2 = [5, 6, 7, 8]
print(l1+l2)
```

Output: [1,2,3,4,5,6,7,8]

Repetition Operator:

The repetition operator enables the list elements to be repeated multiple times. **'*'** is used as concatenation operator.

Example:

```
l1 = [1, 2, 3, 4]
l2 = [5, 6, 7, 8]
print(l1*2)
```

Output: [1,2,3,4,1,2,3,4]

List Built in Functions:

Python provides the following built-in functions, which can be used with the lists.

S. No	Function	Description	Example
1	len(list)	It is used to calculate the length of the list.	L1 = [1,2,3,4,5,6,7,8] print(len(L1)) Output: 8
2	max(list)	It returns the maximum element of the list.	L1 = [12,34,26,48,72] print(max(L1)) Output: 72
3	min(list)	It returns the minimum element of the list.	L1 = [12,34,26,48,72] print(min(L1)) Output: 12
4	list(seq)	It converts any sequence to the list.	str = "Johnson" s = list(str) print(type(s)) Output: <class list>

TUPLES:

Python Tuple is used to store the sequence of immutable Python objects. The tuple is similar to lists since the value of the items stored in the list can be changed, whereas the tuple is immutable, and the value of the items stored in the tuple cannot be changed.

Creating a tuple:
A tuple can be written as the collection of comma-separated (,) values enclosed with the small () brackets.

Syntax:

tup = (item1, item2, item3, ……, itemN)

Example:

Tup=(100,"College", 12.5, "Hyderabad")

Tuple Indexing & Slicing:

The indexing in the tuple starts from 0 and goes to length(tuple) - 1.
The items in the tuple can be accessed by using the index [ ] operator. Python also allows us to use the colon operator to access multiple items in the tuple.

tup=(0,1,2,3,4,5)

1	2	3	4	5
0	**1**	**2**	**3**	**4**

tup[0] = 1
tup[1] = 2
tup[2] = 3
tup[3] = 4
tup[4] = 5

tup[0:] = (1,2,3,4,5)
tup[:] = (1,2,3,4,5)
tup[2:4] = (3,4)
tup[1:4] = (2,3,4)
tup[:4] = (1,2,3,4)

We can get the **sub-list** of the tuple by using the following syntax:

tuple_varible(start:stop:step)

- The start denotes the starting index position of the tuple.
- The stop denotes the last index position of the tuple.
- The step is used to skip the nth element within a start:stop

Example:

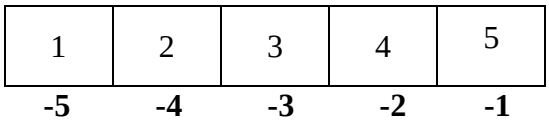
```
tup = (1,2,3,4,5,6,7)
print(tup[0])
print(tup[1])
print(tup[2])
print(tup[3])
# Slicing the elements
print(tup[0:6])
print(tup[:])
print(tup[2:5])
print(tup[1:6:2])
```

Output:

```
1
2
3
4
(1, 2, 3, 4, 5, 6)
(1, 2, 3, 4, 5, 6, 7)
(3, 4, 5)
(2, 4, 6)
```

Negative Indexing:

Python provides the flexibility to use the negative indexing also. The negative indices are counted from the right. The last element (rightmost) of the list has the index -1; its adjacent left element is present at the index -2 and so on until the left-most elements are encountered.



Example:

```
tup = (1,2,3,4,5)
print(list[-1])
print(list[-3:])
print(list[:-1])
print(list[-3:-1])
```

Output:

```
 5
(3, 4, 5)
(1, 2, 3, 4)
(3, 4)
```

Deleting Tuple:

The tuple items cannot be deleted by using the del keyword as tuples are immutable. To delete an entire tuple, we can use the **del** keyword with the tuple name.

Example:

```
Tup=(1,2,3,4,5,6)
del Tup
```

Operators:

S.No	Operator	Description	Example
1	+	'+' operator is called concatenation operator used to join the lists given either side of the operator.	Tup1 =(1,2,3,4) Tup2=(5,6,7,8) print(Tup1+Tup2) **o/p:** (1,2,3,4,5,6,7,8)
2	*	'*' operator is called repetition operator. It concatenates the multiple copies of the same list.	Tup1 =(1,2,3,4) print(str*2) **o/p:** (1,2,3,4,1,2,3,4)
5	in	'in' is called as membership operator. It returns true if a particular itme is present in the specified tuple.	Tup1 =(1,2,3,4) print(3 in Tup1) **o/p:** True
6	not in	'not in' is called as membership operator. It returns true if a particular item is not present in the specified list.	Tup1 =(1,2,3,4) print(3 not in Tup1) **o/p:** False

Tuple Built In Functions:

S. No	Function	Description	Example
1	len(tuple)	It is used to calculate the length of the tuple.	tup = (1,2,3,4,5,6,7,8) print(len(tup)) Output: 8
2	max(tuple)	It returns the maximum element of the tuple.	tup = (12,34,26,48,72) print(max(tup)) Output: 72
3	min(tuple)	It returns the minimum element of the tuple.	tup = (12,34,26,48,72) print(min(tup)) Output: 12
4	tuple(seq)	It converts any sequence to the tuple.	str = "college" s = tuple(str) print(type(s)) Output: <class tuple>

DIFFERENCE BETWEEN LIST & TUPLE:

S. No	Parameter	List	Tuple
1	Type	A list is mutable in nature.	A tuple is immutable in nature.
2	Consumption of Memory	It is capable of consuming more memory.	It is capable of consuming less memory.
3	Time Consumption	The list iteration is more time-consuming. It is comparatively much slower than a tuple.	The tuple iteration is less time-consuming. It is comparatively much faster than a list.
4	Methods	It comes with multiple in-built methods.	These have comparatively lesser built-in methods in them.
5	Appropriate Usage	It is very helpful in the case of deletion and insertion operations.	It is comparatively helpful in the case of read-only operations, such as accessing elements.
6	Prone to Error	The list operations are comparatively much more error-prone. Some unexpected changes and alterations may take place.	Any such thing is hard to take place in a tuple. The tuple operations are very safe and not very error-prone.

MAPPING & SET TYPES:

- Python Dictionary is used to store the data in a key-value pair format. The dictionary is the data type in Python, which can simulate the real-life data arrangement where some specific value exists for some particular key.
- It is the mutable data-structure.
- The dictionary is defined into element Keys and values.
    - Keys must be a single element
    - Value can be any type such as list, tuple, integer, etc.

Creating the dictionary:

The dictionary can be created by using multiple key-value pairs enclosed with the curly brackets {}, and each key is separated from its value by the colon (:).

Syntax:

Dict={“Key1”:”Value1”, “Key2”:”Value2”,……., “KeyN”:”ValueN”}

Example:

Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}

Python provides the built-in function **dict()** method which is also used to create dictionary.

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)

# Creating a Dictionary
# with dict() method
Dict = dict({1: 'Python', 2: 'Programming', 3:'College'})
print("\nCreate Dictionary by using  dict(): ")
print(Dict)

# Creating a Dictionary
# with each item as a Pair
Dict = dict([(1, 'College'), (2, 'Hyderabad')])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

Output:

```
Empty Dictionary:
{}

Create Dictionary by using dict():
{1: 'Python', 2: 'Programming', 3: 'College'}

Dictionary with each item as a pair:
{1: 'College', 2: 'Hyderabad'}
```

Accessing the dictionary values:

The values can be accessed in the dictionary by using the keys as keys are unique in the dictionary.

Example:

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
print(type(Employee))
print("printing Employee data .... ")
print("Name : %s" %Employee["Name"])
print("Age : %d" %Employee["Age"])
print("Salary : %d" %Employee["salary"])
print("Company : %s" %Employee["Company"])
```

Output:

```
<class 'dict'>
printing Employee data ....
Name : John
Age : 29
Salary : 25000
Company : GOOGLE
```

Adding dictionary values:

The dictionary is a mutable data type, and its values can be updated by using the specific keys. The value can be updated along with key **Dict[key] = value**. The update() method is also used to update an existing value.

Example:

```
# Creating an empty Dictionary
Dict = {}
```

```
print("Empty Dictionary: ")
print(Dict)

# Adding elements to dictionary one at a time
Dict[0] = 'ABC'
Dict[2] = 'XYZ'
Dict[3] = 'DEF'
print("\nDictionary after adding 3 elements: ")
print(Dict)

# Adding set of values
# with a single Key
# The Emp_ages doesn't exist to dictionary
Dict['Emp_ages'] = 20, 33, 24
print("\nDictionary after adding 3 elements: ")
print(Dict)

# Updating existing Key's Value
Dict[3] = 'JavaTpoint'
print("\nUpdated key value: ")
print(Dict)
```

Output:

```
Empty Dictionary:
{}

Dictionary after adding 3 elements:
{0: 'ABC', 2: 'XYZ', 3: ‘DEF'}

Dictionary after adding 3 elements:
{0: 'ABC', 2: 'XYZ', 3: 'DEF', 'Emp_ages': (20, 33, 24)}

Updated key value:
{0: 'ABC', 2: 'XYZ', 3: 'DEF', 'Emp_ages': (20, 33, 24)}
```

Deleting elements using del keyword:

The items of the dictionary can be deleted by using the **del** keyword as given below.

Example:

```
Employee = {"Name": "ABC", "Age": 29, "salary":25000,"Company":"GOOGLE"}
print(type(Employee))
print("printing Employee data .... ")
print(Employee)
print("Deleting some of the employee data")
del Employee["Name"]
del Employee["Company"]
print("printing the modified information ")
print(Employee)
print("Deleting the dictionary: Employee");
del Employee
print("Lets try to print it again ");
print(Employee)
```

Output:

```
<class 'dict'>
printing Employee data ....
{'Name': 'ABC', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}
Deleting some of the employee data
printing the modified information
{'Age': 29, 'salary': 25000}
Deleting the dictionary: Employee
Lets try to print it again
NameError: name 'Employee' is not defined
```

Using pop() method

The pop() method accepts the key as an argument and remove the associated value. Consider the following example.

```
# Creating a Dictionary
Dict = {1: 'ABC', 2: 'XYZ', 3: 'DEF'}
# Deleting a key
# using pop() method
pop_ele = Dict.pop(3)
print(Dict)
```

Output:

```
{1: 'ABC', 2: 'XYZ'}
```

Iterating Dictionary:

loop to print all the keys of a dictionary

```
Employee = {"Name": "ABC", "Age": 29, "salary":25000,"Company":"GOOGLE"}
for x in Employee:
  print(x)
```

Output:

```
Name
Age
salary
Company
```

loop to print all the values of the dictionary

```
Employee = {"Name": "ABC", "Age": 29, "salary":25000,"Company":"GOOGLE"}
for x in Employee:
  print(Employee[x])
```

Output:

```
ABC
29
25000
GOOGLE
```

loop to print the values of the dictionary by using values() method.

```
Employee = {"Name": "ABC", "Age": 29, "salary":25000,"Company":"GOOGLE"}
for x in Employee.values():
  print(x)
```

Output:

```
ABC
29
25000
GOOGLE
```

loop to print the items of the dictionary by using items() method.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
for x in Employee.items():
  print(x)
```

Output:

('Name', 'ABC')
('Age', 29)
('salary', 25000)
('Company', 'GOOGLE')

Properties of Dictionary keys:

1. In the dictionary, we cannot store multiple values for the same keys. If we pass more than one value for a single key, then the value which is last assigned is considered as the value of the key.

Example:

```
Employee={"Name":"ABC","Age":29,"Salary":25000,"Company":"GOOGLE","Name":"ABC"}
for x,y in Employee.items():
        print(x,y)
```

Output:

Name ABC
Age 29
Salary 25000
Company GOOGLE

2. In python, the key cannot be any mutable object. We can use numbers, strings, or tuples as the key, but we cannot use any mutable object like the list as the key in the dictionary.

Example:

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE",[100,201,301]:"Department ID"}
for x,y in Employee.items():
   print(x,y)
```

Output:

```
Traceback (most recent call last):
  File "dictionary.py", line 1, in

Employee={"Name":"ABC","Age":29"salary":25000,"Company":"GOOGLE",[100,201,301]:"Departme
nt ID"}
TypeError: unhashable type: 'list'
```

Built-in Dictionary functions:

S.No	Function	Description
1	len(dict)	It is used to calculate the length of the dictionary.
2	str(dict)	It converts the dictionary into the printable string representation.
3	type(variable)	It is used to print the type of the passed variable.

Built-in Dictionary methods:

S.No	Function	Description
1	clear()	It is used to delete all the items of the dictionary.
2	copy()	It returns a shallow copy of the dictionary.
3	get(key, default = "None")	It is used to get the value specified for the passed key.
4	has_key(key)	It returns true if the dictionary contains the specified key.
5	items()	It returns all the key-value pairs as a tuple.
6	setdefault(key,default= "None")	It is used to set the key to the default value if the key is not specified in the dictionary
7	update(dict2)	It updates the dictionary by adding the key-value pair of dict2 to this dictionary.
8	values()	It returns all the values of the dictionary.
9	popItem()	popitem() method removes an element from the dictionary.
10	pop()	pop() method removes an element from the dictionary. It removes the element which is associated to the specified key.