Tuples:

Tuple is exactly same as List except that it is immutable. i.e. once we create Tuple object, we cannot perform any changes in that object.

Hence Tuple is Read Only Version of List.

- Duplicates are allowed.
- Heterogeneous objects are allowed.
- Items in a tuple are immutable.

We can preserve insertion order and we can differentiate duplicate objects by using index. Hence index will play very important role in Tuple also.

Syntax – tuple_name = (item1, item2, item3….)

- **Creation of a tuple:**

A tuple is created by placing items inside parenthesis '()' separated by comma. Parenthesis are optional but recommended to use.

E.g. cars = ('Audi', 'Mercedes', 'BMW')

The tuple () constructor provides an alternative way to create a tuple.

E.g. cars = tuple (('Audi', 'Mercedes', 'BMW'))

- Creating tuple with one item:

Note: We have to take special care about single valued tuple.

Compulsory the value should ends with comma, otherwise it is not treated as tuple.

tuple_name = (item1,)

- **Accessing elements of a tuple:**

We can access characters of a tuple by using the following ways:

1. By using index
2. By using slice operator

- **Mathematical operators for list:** '+' for concatenation and '*' for repetition.

Some other important functions of a tuple: **len (), count (), index (), sorted (), min () and max ()**

- **Tuple packing and Unpacking:** We can create a tuple by packing a group of variables.

E.g.:

a=10

b=20

c=30

d=40

t=a, b, c, d

print(t) # (10, 20, 30, 40)

Here a, b, c, d are packed into a tuple t. This is nothing but tuple packing.

Tuple unpacking is the reverse process of tuple packing. We can unpack a tuple and assign its values to different variables

E.g.:

t= (10,20,30,40)

a, b, c, d = t

print ("a=", a, "b=", b, "c=", c, "d=", d) #o/p - a= 10 b= 20 c= 30 d= 40

Note: At the time of tuple unpacking the number of variables and number of values should be same., otherwise we will get Value Error.

To resolve this problem, we can use '*' symbol in front of any variable.

E.g. – t = (10,20,30,40)

a, b, *c = t # 30 and 40 both values will be assigned to variable c.

Dictionary:

We can use List, Tuple and Set to represent a group of individual objects as a single entity.

If we want to represent a group of objects as key-value pairs then we should go for Dictionary.

E.g.: roll no----name

phone number--address

Ip address---domain name

- Duplicate keys are not allowed but values can be duplicated.
- Heterogeneous objects are allowed for both key and values.
- Insertion order is not preserved
- Dictionaries are mutable
- Dictionaries are dynamic in size.
- Indexing and slicing concepts are not applicable
- **Creating a dictionary:**

d= {} or d=dict ()

we can create empty dictionary. We can add entries as follows

d [100] ="Python"

d [200] ="Java"

d [300] ="C"

print(d) # {100: 'Python', 200: 'Java', 300: 'C'}

If we know data in advance then we can create dictionary as follows

d = {100:'Python' ,200:'Java', 300:'C'}

d = {key: value, key: value}

- **Accessing Elements of a dictionary:**

We can access data by using keys.

d = {100:'python' ,200:'java', 300:'c'}

print (d [100]) #python

print (d [300]) #c

If the specified key is not available then we will get Key Error.

print (d [400]) # Key Error: 40

- **How to update dictionaries?**

d[key]=value

--If the key is not available then a new entry will be added to the dictionary with the specified key-value pair.

--If the key is already available then old value will be replaced with new value.

E.g.:

d={100:"python",200:"java",300:"c"}

print(d)

d [400] ="c++"

print(d) # {100:"python",200:"java",300:"c",400:" c++"}

d [100] ="sunny"

print(d)

- **How to delete elements from dictionary?**

del d[key]

It deletes entry associated with the specified key. If the key is not available then we will get Key Error.

E.g.: d={100:"durga",200:"ravi",300:"shiva"}

print(d)

del d [100]

print(d)

del d [400]

d. clear () - To remove all entries from the dictionary

del d - To delete total dictionary.

- **Important functions of dictionary:**
- dict (): To create a dictionary
- d=dict () ===>It creates empty dictionary
- d=dict({100:"durga",200:"ravi"}) ==>It creates dictionary with specified elements.
- d=dict([(100,"durga"), (200,"shiva"), (300,"ravi")]) ==>It creates dictionary with the given list of tuple elements.

len (), clear (), get (), pop (), popitem (), copy (),

- get () - To get the value associated with the key

d. get (key). If the key is available then returns the corresponding value otherwise returns None. It won't raise any error.

- pop () - d. pop(key)

It removes the entry associated with the specified key and returns the corresponding value. If the specified key is not available then we will get Key Error.

- popitem (): It removes an arbitrary item(key-value) from the dictionary and returns it.

syntax – dict_name. popitem ()

- keys (): It returns all keys associated with dictionary
- values (): It returns all values associated with the dictionary
- update (): d. update (x) All items present in the dictionary x will be added to dictionary d.
- **Data Type conversion: -**

In type conversion, the Python interpreter automatically converts one data type to another. Since Python handles the implicit data type conversion, the programmer does not have to convert the data type into another type explicitly.

The data type to which the conversion happens is called the destination data type, and the data type from which the conversion happens is called the source data type.

- **Type casting: -**

In type casting also known as type coercion, the programmer has to change the data type as per their requirement manually. In this, the programmer explicitly converts the data type using predefined functions like int (), float (), str (), etc. There is a chance of data loss in this case if a particular data type is converted to another data type of a smaller size.

The in-built Python Functions used for the conversion are given below, along with their descriptions:

Function	Description
int (x, base)	Converts any data type x to an integer with the mentioned base
float (x)	Converts x data type to a floating-point number
complex (real, imag)	converts a real number to a complex number
str (x)	converts x data type to a string
tuple (x)	converts x data type to a tuple
list (x)	converts x data type to a list
set (x)	converts x data type to a set
dict (x)	creates a dictionary and x data type should be a sequence of (key, value) tuples
ord (x)	converts a character x into an integer
hex (x)	converts an integer x to a hexadecimal string
oct (x)	converts an integer x to an octal string
chr (x)	converts a number x to its corresponding ASCII (American Standard Code for Information Interchange) character

1. int (x, base) - We can use this function to convert values from other types to int.

E.g. int (10.55) #10

int (True) #1

int ("12") #12

int ('10.5') #Value Error

int ('ten') #Value Error

Note:

1. We can convert from any type to int except complex type.

2. If we want to convert str type to int type, compulsory str should contain only integral value and should be specified in base-10.

1. **float ():** We can use float () function to convert other type values to float type.

Note:

1. We can convert any type value to float type except complex type.

2. Whenever we are trying to convert str type to float type compulsory str should be either integral or floating-point literal and should be specified only in base-10.

1. **complex ():** We can use complex () function to convert other types to complex type.

Form-1: complex(x)

We can use this function to convert x into complex number with real part x and imaginary part 0.

Form-2: complex (x, y)

We can use this method to convert x and y into complex number such that x will be real part and y will be imaginary part.

1. **str ():** We can use this method to convert other type values to str type.

Operators:

Operator is a symbol that performs certain operations.

Python provides the following set of operators.

1. Arithmetic Operators

2. Relational Operators or Comparison Operators

3. Logical operators

4. Bitwise operators

5. Assignment operators

6. Special operators

Membership and Identity operators

1. **Arithmetic Operators:**

+ ==>Addition

- ==>Subtraction

* ==>Multiplication

/ ==>Division operator

% ===>Modulo operator

// ==>Floor Division operator

** ==>Exponent operator or power operator

Note: / operator always performs floating point arithmetic. Hence it will always return float value.

But Floor division (//) can perform both floating point and integral arithmetic. If arguments are int type then result is int type. If at least one argument is floating type,

then result is floating type.

Note: We can use +, * operators for str type also. If we want to use + operator for str type then compulsory both arguments should be str type only otherwise we will get error.

If we use * operator for str type then compulsory one argument should be int and other argument should be str type.

Note: For any number x, x/0 and x%0 always raises "ZeroDivisionError".

1. **Relational Operators:**

>,>=,<,<=

We can apply relational operators for str types also

Note: Chaining of relational operators is possible. In the chaining, if all comparisons return True then only result is True. If at least one comparison returns False then the result is False

E.g.:

1) 10 True

2) 10<20 True

3) 10<20<30 True

4) 10<20<3050 ==>False

Equality operators: ==, !=

We can apply these operators for any type even for incompatible types also.

1. **Logical Operators:**

and, or, not

We can apply logical operators for all types.

- For Boolean type behavior:

and ==>If both arguments are True then only result is True

or =====>If at least one argument is True then result is True

not ==>complement

True and False ==>False, True or False ====>True, not False ==>True

- For non-Boolean types behavior: 0 means False non-zero means True

Empty string is always treated as False

x **and** y: ==>if x is evaluating to false return x otherwise return

x **or** y: If x evaluates to True then result is x otherwise result is y

10 or 20 ==> 10

0 or 20 ==> 20

If x is evaluating to False then result is True otherwise False

not 10 ==>False

not 0 ==>True

1. **Bitwise Operators:**

&,|,^,~,<<,>>

We can apply these operators bitwise. These operators are applicable only for int and Boolean type.

& ==> If both bits are 1 then only result is 1 otherwise result is 0

| ==> If at least one bit is 1 then result is 1 otherwise result is 0

^ ==>If bits are different then only result is 1 otherwise result is 0

~ ==>bitwise complement operator 1==>0 & 0==>1

<< ==>Bitwise Left shift

>> ==>Bitwise Right Shift

1. **Assignment Operator:**

We can use assignment operator to assign value to the variable.

E.g.: x=10

We can combine assignment operator with some other operator to form compound assignment operator. E.g.: x+=10 ====> x = x+10

The following is the list of all possible compound assignment operators in Python

+=, -=, *=, /=, %=, //=, **=, &=, |=, ^=, >>=, <<=

1. **Ternary Operator:**

Syntax: x = first Value if condition else second Value

If condition is True then first Value will be considered else second Value will be considered.

E.g. min=a if a<b else b

1. **Special operators:**

Python defines the following 2 special operators

1. Identity Operators

2. Membership operators as special operators.

Identity Operators: We can use identity operators for address comparison.

r1 is r2 returns True if both r1 and r2 are pointing to the same object

r1 is not r2 returns True if both r1 and r2 are not pointing to the same object.

Note: We can use is operator for address comparison whereas == operator for content comparison.

- **Operators Precedence:**

If multiple operators present then which operator will be evaluated first is decided by operator precedence.

() ⌊ Parenthesis

** ⌊ exponential operator

~, - ⌊ Bitwise complement operator, unary minus operator

*, /, %, // ⌊ multiplication, division, modulo, floor division

+, - ⌊ addition, subtraction

<> ⌊ Left and Right Shift & ⌊ bitwise And

^ ⌊ Bitwise X-OR

| ⌊ Bitwise OR

>,>=, <, <=, ==, != ==>Relational or Comparison operators

=, +=, -=, *=... ==>Assignment operators

is, is not ⌊ Identity Operators

in, not in ⌊ Membership operators

not ⌊ Logical not

and ⌊ Logical and

or ⌊ Logical or

Mathematical Functions:

A Module is collection of functions, variables and classes etc.

Math is a module that contains several functions to perform mathematical operations.

If we want to use any module in Python, first we have to import that module.

import math

Once we import a module then we can call any function of that module.

E.g.

```
import math
print (math. sqrt (16))
print (math. pi)
#4.0
#3.141592653589793
```

We can create alias name by using as keyword.

E.g. import math as m

Once we create alias name, by using that we can access functions and variables of that module.

E.g.

```
import math as m
print (m. sqrt (16))
print (m. pi)
```

We can import a particular member of a module explicitly as follows:

from math import sqrt

from math import sqrt, pi

- Important functions of math module:

ceil(x), floor(x), pow (x, y), factorial(x), trunc(x), gcd(x, y), sin(x), cos(x), tan(x)