Type Casting

We can convert one type value to another type. This conversion is called Typecasting or Type coersion.
The following are various inbuilt functions for type casting.

1. int()
2. float()
3. complex()
4. bool()
5. str()

1.int():

We can use this function to convert values from other types to int

Eg:

```
1)  >>> int(123.987)
2)  123
3)  >>> int(10+5j)
4)  TypeError: can't convert complex to int
5)  >>> int(True)
6)  1
7)  >>> int(False)
8)  0
9)  >>> int("10")
10) 10
11) >>> int("10.5")
12) ValueError: invalid literal for int() with base 10: '10.5'
13) >>> int("ten")
14) ValueError: invalid literal for int() with base 10: 'ten'
15) >>> int("0B1111")
16) ValueError: invalid literal for int() with base 10: '0B1111'
```

Note:

1. We can convert from any type to int except complex type.
2. If we want to convert str type to int type, compulsary str should contain only integral value and should be specified in base-10

2. float():

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

```
1)  >>> float(10)
2)  10.0
3)  >>> float(10+5j)
4)  TypeError: can't convert complex to float
5)  >>> float(True)
6)  1.0
7)  >>> float(False)
8)  0.0
9)  >>> float("10")
10) 10.0
11) >>> float("10.5")
12) 10.5
13) >>> float("ten")
14) ValueError: could not convert string to float: 'ten'
15) >>> float("0B1111")
16) ValueError: could not convert string to float: '0B1111'
```

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 compulsary str should be either integral or floating point literal and should be specified only in base-10.

3.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.

Eg:

```
1)  complex(10)==>10+0j
2)  complex(10.5)===>10.5+0j
3)  complex(True)==>1+0j
4)  complex(False)==>0j
5)  complex("10")==>10+0j
6)  complex("10.5")==>10.5+0j
7)  complex("ten")
8)      ValueError: complex() arg is a malformed string
```

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.

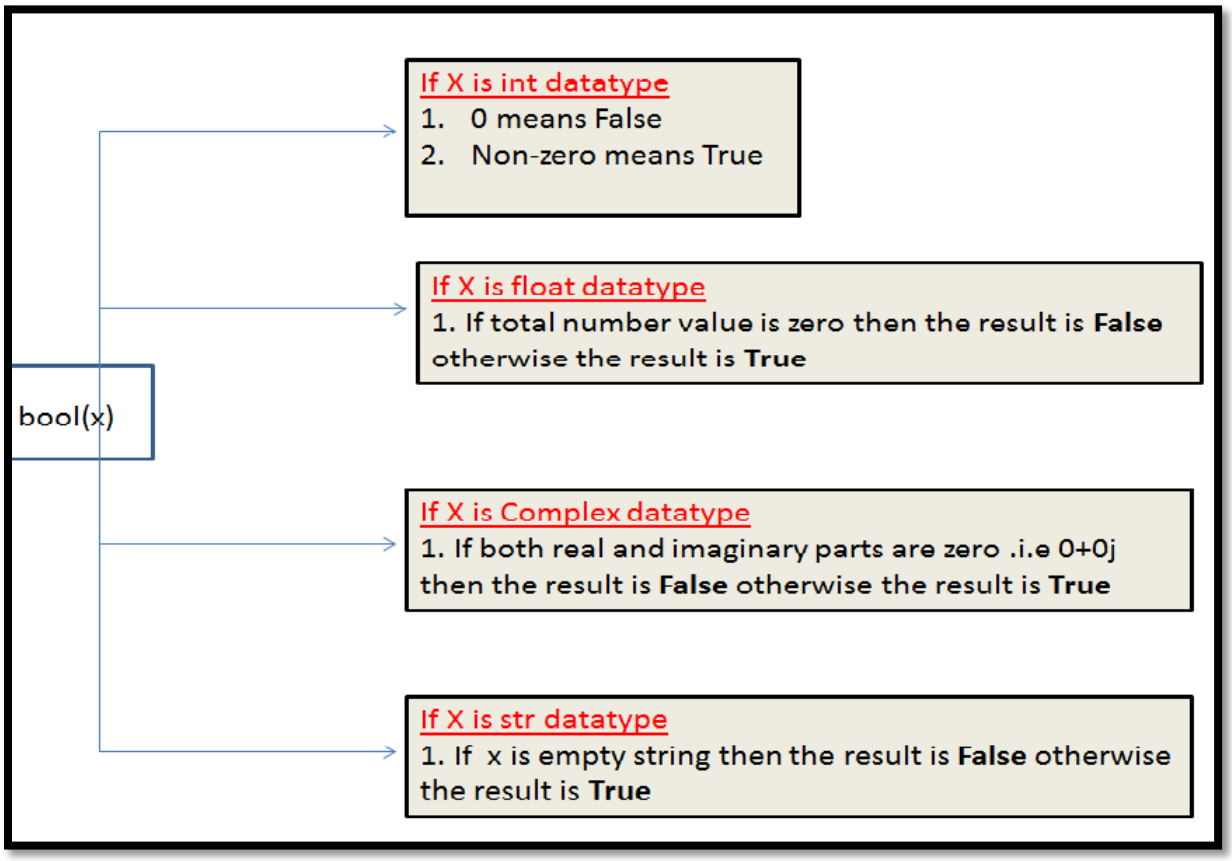
Eg: complex(10,-2)==>10-2j
complex(True,False)==>1+0j

4. bool():

We can use this function to convert other type values to bool type.

Eg:

```
1)  bool(0)==>False
2)  bool(1)==>True
3)  bool(10)===>True
4)  bool(10.5)===>True
5)  bool(0.178)==>True
6)  bool(0.0)==>False
7)  bool(10-2j)==>True
8)  bool(0+1.5j)==>True
9)  bool(0+0j)==>False
10) bool("True")==>True
11) bool("False")==>True
12) bool("")==>False
```



5. str():

We can use this method to convert other type values to str type

Eg:

```
1) >>> str(10)
2) '10'
3) >>> str(10.5)
4) '10.5'
5) >>> str(10+5j)
6) '(10+5j)'
7) >>> str(True)
8) 'True'
```

Fundamental Data Types vs Immutability:

All Fundamental Data types are immutable. i.e once we creates an object,we cannot perform any changes in that object. If we are trying to change then with those changes a new object will be created. This non-chageable behaviour is called immutability.

In Python if a new object is required, then PVM wont create object immediately. First it will check is any object available with the required content or not. If available then existing object will be reused. If it is not available then only a new object will be created. The advantage of this approach is memory utilization and performance will be improved.

But the problem in this approach is,several references pointing to the same object,by using one reference if we are allowed to change the content in the existing object then the remaining references will be effected. To prevent this immutability concept is required. According to this once creates an object we are not allowed to change content. If we are trying to change with those changes a new object will be created.

Eg:

```
1)  >>> a=10
2)  >>> b=10
3)  >>> a is b
4)  True
5)  >>> id(a)
6)  1572353952
7)  >>> id(b)
8)  1572353952
9)  >>>
```

```
>>> a=10
>>> b=10
>>> id(a)
1572353952
>>> id(b)
1572353952
>>> a is b
True
```

```
>>> a=10+5j
>>> b=10+5j
>>> a is b
False
>>> id(a)
15980256
>>> id(b)
15979944
```

```
>>> a=True
>>> b=True
>>> a is b
True
>>> id(a)
1572172624
>>> id(b)
1572172624
```

```
>>> a='durga'
>>> b='durga'
>>> a is b
True
>>> id(a)
16378848
>>> id(b)
16378848
```

bytes Data Type:

bytes data type represens a group of byte numbers just like an array.

Eg:

```
1)  x = [10,20,30,40]
2)      b = bytes(x)
3)      type(b)==>bytes
4)      print(b[0])==> 10
5)      print(b[-1])==> 40
6)      >>> for i in b : print(i)
7)
8)        10
9)        20
10)       30
11)       40
```

Conclusion 1:

The only allowed values for byte data type are 0 to 256. By mistake if we are trying to provide any other values then we will get value error.

Conclusion 2:

Once we creates bytes data type value, we cannot change its values,otherwise we will get TypeError.

Eg:

```
1)  >>> x=[10,20,30,40]
2)  >>> b=bytes(x)
3)  >>> b[0]=100
4)   TypeError: 'bytes' object does not support item assignment
```

bytearray Data type:

bytearray is exactly same as bytes data type except that its elements can be modified.

Eg 1:

```
1)  x=[10,20,30,40]
2)  b = bytearray(x)
3)  for i in b : print(i)
4)  10
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

```
5)  20
6)  30
7)  40
8)   b[0]=100
9)   for i in b: print(i)
10) 100
11) 20
12) 30
13) 40
```

Eg 2:

```
1)  >>> x =[10,256]
2)  >>> b = bytearray(x)
3)  ValueError: byte must be in range(0, 256)
```

list data type:

If we want to represent a group of values as a single entity where insertion order required to preserve and duplicates are allowed then we should go for list data type.

1. insertion order is preserved
2. heterogeneous objects are allowed
3. duplicates are allowed
4. Growable in nature
5. values should be enclosed within square brackets.

Eg:

```
1)  list=[10,10.5,'durga',True,10]
2)      print(list) # [10,10.5,'durga',True,10]
```

Eg:

```
1)  list=[10,20,30,40]
2)  >>> list[0]
3)  10
4)  >>> list[-1]
5)  40
6)  >>> list[1:3]
7)  [20, 30]
8)  >>> list[0]=100
9)  >>> for i in list:print(i)
10) ...
11) 100
12) 20
13) 30
```

```
14) 40
```

list is growable in nature. i.e based on our requirement we can increase or decrease the size.

```
1)  >>> list=[10,20,30]
2)  >>> list.append("durga")
3)  >>> list
4)  [10, 20, 30, 'durga']
5)  >>> list.remove(20)
6)  >>> list
7)  [10, 30, 'durga']
8)  >>> list2=list*2
9)  >>> list2
10) [10, 30, 'durga', 10, 30, 'durga']
```

Note: An ordered, mutable, heterogenous collection of eleemnts is nothing but list, where duplicates also allowed.

tuple data type:

tuple data type is exactly same as list data type except that it is immutable.i.e we cannot chage values.

Tuple elements can be represented within parenthesis.

Eg:

```
1)  t=(10,20,30,40)
2)  type(t)
3)  <class 'tuple'>
4)  t[0]=100
5)  TypeError: 'tuple' object does not support item assignment
6)  >>> t.append("durga")
7)  AttributeError: 'tuple' object has no attribute 'append'
8)  >>> t.remove(10)
9)  AttributeError: 'tuple' object has no attribute 'remove'
```

Note: tuple is the read only version of list

range Data Type:

range Data Type represents a sequence of numbers.
The elements present in range Data type are not modifiable. i.e range Data type is immutable.

Form-1: range(10)
generate numbers from 0 to 9

Eg:
r=range(10)
for i in r : print(i) 0 to 9

Form-2: range(10,20)

generate numbers from 10 to 19

r = range(10,20)
for i in r : print(i) 10 to 19

Form-3: range(10,20,2)

2 means increment value

r = range(10,20,2)
for i in r : print(i) 10,12,14,16,18

We can access elements present in the range Data Type by using index.
r=range(10,20)
r[0]==>10
r[15]==>IndexError: range object index out of range

We cannot modify the values of range data type

Eg:
r[0]=100
TypeError: 'range' object does not support item assignment

We can create a list of values with range data type

Eg:

```
1)  >>> l = list(range(10))
2)  >>> l
3)  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

set Data Type:

If we want to represent a group of values without duplicates where order is not important then we should go for set Data Type.

1. insertion order is not preserved
2. duplicates are not allowed
3. heterogeneous objects are allowed
4. index concept is not applicable
5. It is mutable collection
6. Growable in nature

Eg:

```
1)  s={100,0,10,200,10,'durga'}
2)  s # {0, 100, 'durga', 200, 10}
3)  s[0] ==>TypeError: 'set' object does not support indexing
4)
5)  set is growable in nature, based on our requirement we can increase or decrease the size.
6)
7)  >>> s.add(60)
8)  >>> s
9)  {0, 100, 'durga', 200, 10, 60}
10) >>> s.remove(100)
11) >>> s
12) {0, 'durga', 200, 10, 60}
```

frozenset Data Type:

It is exactly same as set except that it is immutable.
Hence we cannot use add or remove functions.

```
1)  >>> s={10,20,30,40}
2)  >>> fs=frozenset(s)
3)  >>> type(fs)
4)  <class 'frozenset'>
5)  >>> fs
6)  frozenset({40, 10, 20, 30})
7)  >>> for i in fs:print(i)
8)  ...
9)  40
10) 10
11) 20
12) 30
13)
14) >>> fs.add(70)
15) AttributeError: 'frozenset' object has no attribute 'add'
16) >>> fs.remove(10)
17) AttributeError: 'frozenset' object has no attribute 'remove'
```

dict Data Type:

If we want to represent a group of values as key-value pairs then we should go for dict data type.

Eg:

```
d={101:'durga',102:'ravi',103:'shiva'}
```

Duplicate keys are not allowed but values can be duplicated. If we are trying to insert an entry with duplicate key then old value will be replaced with new value.

Eg:

```
>>> d={101:'durga',102:'ravi',103:'shiva'}
>>> d[101]='sunny'
>>> d
{101: 'sunny', 102: 'ravi', 103: 'shiva'}

We can create empty dictionary as follows
d={ }
We can add key-value pairs as follows
d['a']='apple'
d['b']='banana'
print(d)
```

Note: dict is mutable and the order wont be preserved.

Note:

1. In general we can use bytes and bytearray data types to represent binary information like images,video files etc
2. In Python2 long data type is available. But in Python3 it is not available and we can represent long values also by using int type only.
3. In Python there is no char data type. Hence we can represent char values also by using str type.

Summary of Datatypes in Python3

Datatype	Description	Is Immutable	Example

Int	We can use to represent the whole/integral numbers	Immutable	>>> a=10
			>>> type(a)
			<class 'int'>

Float	We can use to represent the decimal/floating point numbers	Immutable	>>> b=10.5
			>>> type(b)
			<class 'float'>

Complex	We can use to represent the complex numbers	Immutable	>>> c=10+5j
			>>> type(c)
			<class 'complex'>
			>>> c.real
			10.0
			>>> c.imag
			5.0

Bool	We can use to represent the logical values(Only allowed values are True and False)	Immutable	>>> flag=True
			>>> flag=False
			>>> type(flag)
			<class 'bool'>

Str	To represent sequence of Characters	Immutable	>>> s='durga'
			>>> type(s)
			<class 'str'>
			>>> s="durga"
			>>> s='''Durga Software Solutions
			... Ameerpet'''
			>>> type(s)
			<class 'str'>

bytes	To represent a sequence of byte values from 0-255	Immutable	>>> list=[1,2,3,4]
			>>> b=bytes(list)
			>>> type(b)
			<class 'bytes'>

bytearray	To represent a sequence of byte values from 0-255	Mutable	>>> list=[10,20,30]
			>>> ba=bytearray(list)
			>>> type(ba)
			<class 'bytearray'>

range	To represent a range of values	Immutable	>>> r=range(10)
			>>> r1=range(0,10)
			>>> r2=range(0,10,2)

list	To represent an ordered collection of objects	Mutable	>>> l=[10,11,12,13,14,15]
			>>> type(l)
			<class 'list'>

tuple	To represent an ordered collections of objects	Immutable	>>> t=(1,2,3,4,5)
			>>> type(t)
			<class 'tuple'>

set	To represent an unordered collection of unique objects	Mutable	>>> s={1,2,3,4,5,6} >>> type(s)

			<class 'set'>

frozenset	To represent an unordered collection of unique objects	Immutable	>>> s={11,2,3,'Durga',100,'Ramu'}
			>>> fs=frozenset(s)
			>>> type(fs)
			<class 'frozenset'>

dict	To represent a group of key value pairs	Mutable	>>> d={101:'durga',102:'ramu',103:'hari'} >>> type(d) <class 'dict'>

None Data Type:

None means Nothing or No value associated.
If the value is not available,then to handle such type of cases None introduced.
It is something like null value in Java.

Eg:

```
def  m1():
  a=10

print(m1())
  None
```

Escape Characters:

In String literals we can use esacpe characters to associate a special meaning.

```
1)  >>> s="durga\nsoftware"
2)  >>> print(s)
3)  durga
4)  software
5)  >>> s="durga\tsoftware"
6)  >>> print(s)
7)  durga   software
8)  >>> s="This is " symbol"
9)    File "<stdin>", line 1
10)    s="This is " symbol"
11)                 ^
12) SyntaxError: invalid syntax
13) >>> s="This is \" symbol"
14) >>> print(s)
15) This is " symbol
```

The following are various important escape characters in Python

1) \n==>New Line
2) \t===>Horizontal tab
3) \r ==>Carriage Return
4) \b===>Back space
5) \f===>Form Feed
6) \v==>Vertical tab
7) \'===>Single quote
8) \"===>Double quote
9) \\===>back slash symbol
   ....

Constants:

Constants concept is not applicable in Python.
But it is convention to use only uppercase characters if we don't want to change value.

MAX_VALUE=10

It is just convention but we can change the value.

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com