Now Let's Dive deeply in python's Built-in datatypes

Numeric Type

- Python’s numeric datatype includes...
  - Integer and floating-point objects
  - Complex number objects
  - Decimal: fixed-precision objects
  - Fraction: rational number objects
  - Sets: collections with numeric operations
  - Booleans: true and false
  - Built-in functions and modules: round, math, random, etc.
  - Expressions; unlimited integer precision; bitwise operations; hex, octal, and binary formats

Python provides...

- integers :- which are positive and negative whole numbers
- floating-point numbers :- which are numbers with a fractional part (sometimes called “floats” for verbal economy).
- Python also allows us to write integers using hexadecimal, octal, and binary literals.
- complex number type
- Decimals :- unlimited precision integers.they can grow to have as many digits as your memory space allows.

Numeric literals Table

Interpretation	Literal
Integer	1234, −24, 0
Floating-point numbers	1.23, 3.14e-10
Octal literals	0o177
hex literals	0x9ff
binary literals	0b101010
Complex number literals	3+4j, 3.0+4.0j, 3J
Decimal extension type	Decimal('1.0')
Fraction extension type	Fraction(1, 3)

1. Integer and floating-point literals
   - Integers are written as strings of decimal digits.
   - Floating-point numbers have a decimal point and/or an optional signed exponent introduced by an e or E and followed by an optional sign.
   - If you write a number with a decimal point or exponent, Python makes it a floating-point object and uses floating-point by default (not integer).

2. In Python 2.X...
   - there are two integer types

there are two integer types
   A. normal (often 32 bits)
   B. long (unlimited precision)

3. In Python 3.X...
   - the normal and long integer types have been merged—there is only integer

Basic Operations

Numbers in Python support the normal mathematical operations. for Ex...

- the plus sign `+` performs addition, a star `*` is used for multiplication, and two stars `**` are used for exponentiation

In [41]:

```
3 + 5
```

Out[41]:

```
8
```

In [43]:

```
5.8 + 24.7
```

Out[43]:

```
30.5
```

Mixed types Operations

- we can add an integer to a floating-point number
- for Ex. :- 40 + 3.14
- this creates an another question in our mind...
- what type is the result—integer or floating point?
  - The answer is simple Python first converts operands up to the type of the most complicated operand and then performs the operation on same-type operands.therefor when an integer is mixed with a floating point as in the preceding example, the integer is converted up to a floating-point value first then firther operatoion is performed.

In [1]:

```
40 + 3.14
```

Out[1]:

```
43.14
```

Type Cating

- we can also able to conert the type of object(data).

- there are two possibilities of converting datatypes...
  1. converting a less complicated operator in more complicated operator.
     - this type of operaton python performs itself and we don't need to specify.
     - for Ex. :- int to float or complex datatpye conversation.
  2. converting a more complicated operator in less complicated operator.
     - this is also called as a type casting.
     - we need to perform this type of operation forcefully and manually.
     - Casting in done using constructor functions like int(),float(),str() etc.
     - for Ex. :- float to int datatpye conversation.
     - This leads towards a loss of the data.
     - we cannot convert complex numbers into another number type.

In [2]:

```
int(3.1415) # Truncates float to integer
```

Out[2]:

```
3
```

In [3]:

```
float(3) # Converts integer to float
```

Out[3]:

```
3.0
```

In [46]:

```
a = 3
b = 4
c,d = 5,8
a + 1, a - 1,c * 3, d / 2
```

Out[46]:

```
(4, 2, 15, 4.0)
```

In [7]:

```
a % 2, b ** 2
```

Out[7]:

```
(1, 16)
```

- If we use a different variable that has not yet been assigned then Python reports an error rather than filling in some default value.

In [9]:

```
f * 2
```

```
---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
Input In [9], in <cell line: 1>()
----> 1 f * 2

NameError: name 'f' is not defined
```

- in python...
  - we don’t need to predeclare variables but we must have to assign a variablev at least once before we can use them.
  - this means we have to initialize counters to zero before we can use for add them.initialize lists to an empty list before you can append item to them etc.

Numeric Display Formats

In [14]:

```
num = 10000000000 / 3.0
print(num) # Print explicitly
num # Auto-echoes
```

```
3333333333.3333335
```

Out[14]:

```
3333333333.3333335
```

In [15]:

```
'%4.2f' % num # Alternative floating-point format
```

Out[15]:

```
'3333333333.33'
```

In [16]:

```
'{0:4.2f}'.format(num) # String formatting method
```

Out[16]:

```
'3333333333.33'
```

difference between built-in repr and str functions

- the difference between default interactive echoes and print corresponds to the difference between the built-in repr and str functions.
- Both of these convert arbitrary objects to their string representations
  - repr (and the default interactive echo) produces results that look as though they were code.

- str (and the print operation) converts to a typically more user-friendly format.

3.1415 * 2 # repr: as code (Pythons less than 2.7 and 3.1)
>>> 6.2830000000000004
print(3.1415 * 2) # str: user-friendly
>>> 6.283

- first form is known as an object's as-code repr
- second is known as a its user-friendly str

In [44]:

```
repr('spam') # Used by echoes: as-code form
```

Out[44]:

```
"'spam'"
```

In [18]:

```
str('spam') # Used by print: user-friendly form
```

Out[18]:

```
'spam'
```

Normal and Chained Comparisons

- Python objects also supports a comparisions.
- we are able to compare two or more python object using assignment operators.

In [45]:

```
a,b,c = 23,21.8,23
print(a < b) # Less than
print(c >= b) # Greater than or equal: mixed-type 1 converted to 1.0
print(a == c) # Equal value
print(b != a) # Not equal value
```

```
False
True
True
True
```

- Python also allows us to chain multiple comparisons together to perform range tests.
- Chained comparisons are a sort of shorthand for larger Boolean expressions.

In [21]:

```
X,Y,Z = 2,4,6
X < Y < Z # Chained comparisons: range tests
```

Out[21]:

```
True
```

In [22]:

```
X < Y and Y < Z
```

Out[22]:

```
True
```

In [23]:

```
1.1 + 2.2 == 3.3 # Shouldn't this be True?...
```

Out[23]:

```
False
```

In [24]:

```
1.1 + 2.2 # Close to 3.3, but not exactly: limited precision
```

Out[24]:

```
3.3000000000000003
```

In [27]:

```
round(1.1 + 2.2,4) == float(3.3)
```

Out[27]:

```
True
```

Types of Division

- there are mainly three types of Division.
  - Classic Division
  - Floor Division
  - True Division
- X / Y :- Classic and true division
  - In Python 2.X this operator performs classic division.
  - division which truncates the result for integers and keeping remainders or fractional parts for floating-point numbers is called as a classic division.
  - for Ex. 3/2 = 1 and 3.0/2.0 = 1.5 is provided as output.
  - In Python 3.X it always performs a true division.
  - division which always keeping remainders in results regardless of types is called as a true division.
- X // Y :- Floor division.
  - this operator always truncates fractional remainders down to their floor regardless of types.

In [44]:

```
77/3,77//3
```

Out[44]:

```
(25.666666666666668, 25)
```

In [43]:

```
10/4,10//4,10//4.0
```

Out[43]:

```
(2.5, 2, 2.0)
```

- as we know that...when two or more different datatypes are there in expression, python performs typecasting first then produces the result.
- that's why 10///4.0 returns a 2.0 instead 2, as it casts 2 in 2.0

Floor versus truncate

- the `//` operator is informally called truncating division.
- it's more accurate to refer to it as floor division as it truncates the result down to its floor, which means the closest whole number below the true result. -6

In [47]:

```
import math
print(math.floor(2.5))# Closest number below value
print(math.trunc(2.5)) # Truncate fractional part (toward zero)
```

```
2
2
```

In [48]:

```
print(math.floor(-2.5))# Closest number below value
print(math.trunc(-2.5)) # Truncate fractional part (toward zero)
```

```
-3
-2
```

In [49]:

```
A = -27//4
A
```

Out[49]:

```
-7
```

In [50]:

```
A = -27/4
print(math.floor(A))
print(math.trunc(A))
```

```
-7
-6
```

Complex Numbers

- very less used datatype than other datatypes.
- complex numbers are a core object type in Python.
- They are typically used in engineering and science applications.
- complex literals are written as `realpart+imaginarypart` where the imaginarypart is terminated with a j or J.
- The realpart is technically optional so the imaginarypart may appear on its own.
- Complex numbers may also be created with the complex(real, imag) built-in call.

In [51]:

```
complex(2,3)
```

Out[51]:

```
(2+3j)
```

In [52]:

```
complex(2,0)
```

Out[52]:

```
(2+0j)
```

In [53]:

```
complex(0,0)
```

Out[53]:

```
0j
```

In [54]:

```
2 + 1j * 3
```

Out[54]:

```
(2+3j)
```

In [55]:

```
(2 + 1j) * 3
```

Out[55]:

```
(6+3j)
```

In [56]:

```
1j
```

Out[56]:

```
1j
```

In [62]:

```
1j + 1J
```

Out[62]:

```
2j
```

In [66]:

```
(2 + 1j) * (2 + 1J)
# 2*2 + 2j + 2J + 1jJ (J = j - 1 and j*J = 1)
# 4 + 2j + 2(j-1) + 1(1)
# 4 + 2j + 2j - 2 + 1
# 3 + 4j
```

Out[66]:

```
(3+4j)
```

In [69]:

```
(8 + 7j) * (4 + 3j)
# 32 + 28j + 24j + 21(-1)  (j*j = -1)
# 11 + 52j
```

Out[69]:

```
(11+52j)
```

In [78]:

```
a = 5.0 + 0j
a.conjugate()
#  complex.conjugate() -> complex
# Return the complex conjugate of its argument. (3-4j).conjugate() == 3+4j.
```

Out[78]:

```
(5-0j)
```

In [93]:

```
a,b = - 5.0,5
a.conjugate(),b.conjugate()
# conjugate() method of float instance and int instance Return self,the complex conjugat
```

Out[93]:

```
(-5.0, 5)
```

Hex, Octal, Binary Conversions

- Integers may be coded in...
  - decimal (base 10)
  - hexadecimal (base 16)
  - octal (base 8)
  - binary (base 2)
- Hexadecimals start with a leading 0x or 0X followed by a string of hexadecimal digits (0–9 and A–F).
- Hex digits may be coded in lower- or uppercase.
- Octal literals start with a leading 0o or 0O (zero and lower- or uppercase letter o) followed by a string of digits (0–7).
- Binary literals begin with a leading 0b or 0B followed by binary digits (0–1).
- The built-in calls hex(I), oct(I), and bin(I) convert an integer to its representation string in these three bases.
- int(str, base) converts a runtime string to an integer per a given base.

In [96]:

```
A = 170
bin(A),oct(A),hex(A)
```

Out[96]:

```
('0b10101010', '0o252', '0xaa')
```

In [98]:

```
# binary to decimal
A = '0b1011101011'
int(A,2)
```

Out[98]:

```
747
```

In [99]:

```
# binary to octral
A = '0b1011101011'
oct(int(A,2))
```

Out[99]:

```
'0o1353'
```

In [102]:

```
# binary to hexa-decimal
A = '0b1011101011'
hex(int(A,2))
```

Out[102]:

```
'0x2eb'
```

In [119]:

```
# measure a lenght of binary number..
X = 99
len(bin(X)) - 2 , X.bit_length() # -2 because 0b,0x and 0o
```

Out[119]:

```
(7, 7)
```

In [103]:

```
# octtral to decimal
A = '0o1716'
int(A,8)
```

Out[103]:

```
974
```

In [104]:

```
# octtral to binary
A = '0o1716'
bin(int(A,8))
```

Out[104]:

```
'0b1111001110'
```

In [105]:

```
# octtral to hexa-decimal
A = '0o1716'
hex(int(A,8))
```

Out[105]:

```
'0x3ce'
```

In [106]:

```
# hexa-decimal to decimal
A = '0x6ea'
int(A,16)
```

Out[106]:

```
1770
```

In [107]:

```
# hexa-decimal to binary
A = '0x6ea'
bin(int(A,16))
```

Out[107]:

```
'0b11011101010'
```

In [108]:

```
# hexa-decimal to octral
A = '0x6ea'
oct(int(A,16))
```

Out[108]:

```
'0o3352'
```

- The eval function treats strings as a Python code.
- Therefore it has a similar effect but usually runs more slowly.
- it actually compiles and runs the string as a piece of a program
- eval function assumes that the string being run comes from a trusted source so a clever user might be able to submit a string that deletes files on your machine so be careful with this call.

In [111]:

```
eval('64'), eval('0o1750'), eval('0x4d'), eval('0b1011000')
```

Out[111]:

```
(64, 1000, 77, 88)
```

Bitwise Operations Besides the Arithmetic operations (addition, subtraction, and so on) Python also supports Bitwise Operation.

In [112]:

```
x = 10 # 10 decimal is 1010 in bits
x << 2 # Shift left 2 bits: 101000
```

Out[112]:

```
40
```

In [113]:

```
x >> 2 # Shift right 2 bits: 10
```

Out[113]:

```
2
```

In [115]:

```
# Bitwise OR (both 0 then only 0 else 1)
x | 2  # 1010 | 0010 = 1010
```

Out[115]:

```
10
```

In [116]:

```
# Bitwise AND (both bits=1 then only 1 else 0)
x & 1 # 1010 & 0001 = 0000
```

Out[116]:

```
0
```

Built-in Numeric Tools

In [120]:

```
import math
math.pi, math.e
```

Out[120]:

```
(3.141592653589793, 2.718281828459045)
```

In [121]:

```
math.sin(2 * math.pi / 180)
```

Out[121]:

```
0.03489949670250097
```

In [122]:

```
pow(2, 4), 2 ** 4, 2.0 ** 4.0 # Exponentiation (power)
```

Out[122]:

```
(16, 16, 16.0)
```

In [126]:

```
abs(-42.0) # Absolute value
```

Out[126]:

```
42.0
```

In [127]:

```
sum((1, 2, 3, 4)) # summation
```

Out[127]:

```
10
```

In [128]:

```
min(3, 1, 2, 4) # Minimum
```

Out[128]:

```
1
```

In [129]:

```
max(3, 1, 2, 4) # maximum
```

Out[129]:

```
4
```

In [130]:

```
round(2.567804567)
```

Out[130]:

```
3
```

In [131]:

```
round(2.567, 2)
```

Out[131]:

```
2.57
```

- there are three ways to compute square roots in Python

In [132]:

```
import math
math.sqrt(144) # using Module
```

Out[132]:

```
12.0
```

In [133]:

```
144 ** .5 # using Expression
```

Out[133]:

```
12.0
```

In [134]:

```
pow(144, .5) # using Built-in function
```

Out[134]:

```
12.0
```

Decimal Type

- In Python 2.4 new core numeric type the decimal object is introduced formally known as Decimal.
- we able to create decimals by calling a function within an imported module rather than running a literal expression.
- decimals are like floating-point numbers but they have a fixed number of decimal points.
- Hence, decimals are also called as a fixed-precision floating-point values.
- For Ex. :- with decimals we can have a floating-point value that always retains just two decimal digits.
- decimal type is still relatively rare in practice.

In [136]:

```
print(0.1 + 0.1 + 0.1 - 0.3)
```

```
5.551115123125783e-17
```

In [1]:

```
from decimal import Decimal
Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
```

Out[1]:

```
Decimal('0.0')
```

In [2]:

```
Decimal('0.10000') + Decimal('0.10000') + Decimal('0.10000') - Decimal('0.30000')
```

Out[2]:

```
Decimal('0.00000')
```

In [3]:

```
Decimal('0.20000') + Decimal('0.10') + Decimal('0.100') - Decimal('0.3')
```

Out[3]:

```
Decimal('0.10000')
```

In [4]:

```
Decimal(1) / Decimal(7)
```

Out[4]:

```
Decimal('0.14285714285714285714285714429')
```

In [5]:

```
from decimal import getcontext
getcontext().prec = 4 # Fixed precisionDecimal(1) / Decimal(7)
```

Out[5]:

```
Decimal('0.1429')
```

In [8]:

```
float(Decimal(1.78))
```

Out[8]:

```
1.78
```

Fraction Type

- In Python 2.6 new core numeric type the fraction object is introduced formally known as Fraction.
- Fraction implements a rational number object.
- Fraction keeps both a numerator and a denominator to avoid some of the inaccuracies and limitations of floating-point math.
- Fraction resides in a module.
- we able to create Fraction by calling a function within an imported module rather than running a literal expression.

In [9]:

```
from fractions import Fraction
x = Fraction(1, 3)
y = Fraction(4, 6)
```

In [15]:

```
x
```

Out[15]:

```
Fraction(1, 3)
```

In [16]:

```
print(x)
```

```
1/3
```

In [17]:

```
y
```

Out[17]:

```
Fraction(2, 3)
```

In [18]:

```
print(y)
```

```
2/3
```

In [19]:

```
x + y
```

Out[19]:

```
Fraction(1, 1)
```

In [22]:

```
x - y
```

Out[22]:

```
Fraction(-1, 3)
```

In [23]:

```
x * y
```

Out[23]:

```
Fraction(2, 9)
```

- Fraction objects can also be created from floating-point number strings.

In [24]:

```
Fraction('.25'),Fraction('1.25')
```

Out[24]:

```
(Fraction(1, 4), Fraction(5, 4))
```

In [25]:

```
Fraction('.25') + Fraction('1.25')
```

Out[25]:

```
Fraction(3, 2)
```

In [26]:

```
0.1 + 0.1 + 0.1 - 0.3
```

Out[26]:

```
5.551115123125783e-17
```

In [30]:

```
Fraction('0.1') + Fraction('0.1') + Fraction('0.1') - Fraction('0.3')
```

Out[30]:

```
Fraction(0, 1)
```

In [31]:

```
print(Fraction('0.1') + Fraction('0.1') + Fraction('0.1') - Fraction('0.3'))
```

```
0
```

In [33]:

```
A = (2.5).as_integer_ratio() # other way to create a fracton
A
```

Out[33]:

```
(5, 2)
```

In [35]:

```
print(A)
```

```
(5, 2)
```

In [37]:

```
z = Fraction(*(2.5).as_integer_ratio()) # Convert float -> fraction: two args
z
```

Out[37]:

```
Fraction(5, 2)
```

In [38]:

```
print(z)
```

```
5/2
```

In [40]:

```
Fraction.from_float(1.75) # Convert float -> fraction: other way
```

Out[40]:

```
Fraction(7, 4)
```

Set Datatype

- Sets are a recent addition to the language that are neither mappings nor sequences.
- Sets are used to store multiple items in a single variable.
- Sets are unordered (unindexed) collections of unique and immutable (unchangeable) objects.
- Set items are unchangeable, but you can remove items and add new items

- we can able to create sets by calling the built-in set function or using new set literals.
- the choice of new {...} syntax for set literals makes sense.
- since, sets are much like the keys of a valueless dictionary that is the reason that set can't contain duplicate data.
- The values True and 1 are considered the same value in sets and are treated as duplicates.

Set Creation

In [1]:

```
# creating a set with built-in function
X = set('spam')
X
```

Out[1]:

```
{'a', 'm', 'p', 's'}
```

In [2]:

```
# creating a set with set literals
Y = {'h', 'a', 'm'}
Y
```

Out[2]:

```
{'a', 'h', 'm'}
```

In [3]:

```
#  Length of a Set :- To determine how many items a set len() function is used.
len(X),len(Y)
```

Out[3]:

```
(4, 3)
```

In [4]:

```
# sets are defined as objects with the data type 'set'
type(X),type(Y)
```

Out[4]:

```
(set, set)
```

In [5]:

```
set1 = {"X", "Y", "Z", True, 1, 0,False}
set1
```

Out[5]:

```
{0, True, 'X', 'Y', 'Z'}
```

- we can't able to create a empty set using {}.
- {} is considerd as an empty dictionary in all Pythons.
- Empty sets must be created with the set built-in constructor and print the same way.

In [6]:

```
S1 = set()
S1
```

Out[6]:

```
set()
```

In [7]:

```
type({}) # Because {} is an empty dictionary
```

Out[7]:

```
dict
```

Access Items

- we cannot access items in a set by referring to an index or a key as set is unordered.

In [8]:

```
S = set("Hitesh")
S
```

Out[8]:

```
{'H', 'e', 'h', 'i', 's', 't'}
```

In [9]:

```
# it is wrong and raisde an error that 'set' object is not subscriptable
for i in range(len(S)):
    print(S[i])
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
Input In [9], in <cell line: 2>()
      1 # it is wrong and raisde an error that 'set' object is not subscri
ptable
      2 for i in range(len(S)):
----> 3     print(S[i])

TypeError: 'set' object is not subscriptable
```

In [10]:

```
#  in-order to acess items in set we need to use membership operator with for loop
for i in S:
    print(i)
```

```
i
t
h
s
e
H
```

Set Tests

In [11]:

```
# membership test
S = set('purvi')
print('i' in S,'h' in S,'r' not in S)
```

```
True False False
```

In [12]:

```
# object identity test
S1 = set('Ramesh')
S2 = set('Reshma')
print(S1 is S2)
```

```
False
```

In [13]:

```
# object identity test
S1 = set('Reshma')
S2 = set('Reshma')
print(S1 is S2)
```

```
False
```

Add Items

- Once a set is created we cannot change its items but we can add new items.
- To add one new item to a set use the add() method.

In [14]:

```
S1 = {'a','b','c','d'}
S1.add('e')
S1
```

Out[14]:

```
{'a', 'b', 'c', 'd', 'e'}
```

- To add more than one item from another collection type into the current set use the update() method.

In [15]:

```
S1 = set('Purvi')
S2 = set('Reshma')
S1.update(S2)
S1
```

Out[15]:

```
{'P', 'R', 'a', 'e', 'h', 'i', 'm', 'r', 's', 'u', 'v'}
```

In [16]:

```
S1 = set('Purvi')
S2 = list('Reshma')
S1.update(S2)
S1
```

Out[16]:

```
{'P', 'R', 'a', 'e', 'h', 'i', 'm', 'r', 's', 'u', 'v'}
```

Remove Item

- To remove an item in a set use the remove() or the discard() method.

In [17]:

```
S1 = {'a','b','c','d'}
S1.remove('b')
S1
```

Out[17]:

```
{'a', 'c', 'd'}
```

- If the item to remove does not exist, remove() will raise an error.

In [18]:

```
S1.remove('b')
S1
```

```
---------------------------------------------------------------------------
-
KeyError                                  Traceback (most recent call las
t)
Input In [18], in <cell line: 1>()
----> 1 S1.remove('b')
      2 S1

KeyError: 'b'
```

- If the item to remove does not exist, discard() will NOT raise an error.

In [19]:

```
S1 = {'a','b','c','d'}
S1.discard('b')
S1
```

Out[19]:

```
{'a', 'c', 'd'}
```

In [20]:

```
S1.discard('b')
S1
```

Out[20]:

```
{'a', 'c', 'd'}
```

- we can also use the pop() method to remove an item but this method will remove a random item.
- so we cannot be sure what item that gets removed.
- The return value of the pop() method is the removed item.

In [21]:

```
S1 = {'a','b','c','d'}
poped = S1.pop()
S1,poped
```

Out[21]:

```
({'a', 'c', 'd'}, 'b')
```

In [22]:

```
S1.pop()
S1
```

Out[22]:

```
{'a', 'd'}
```

In [23]:

```
# The clear() method empties the set
S1 = {'a','b','c','d'}
S1.clear()
S1
```

Out[23]:

```
set()
```

In [24]:

```
# The del keyword will delete the set completely
S1 = {'a','b','c','d'}
del S1
S1
```

```
---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
Input In [24], in <cell line: 4>()
      2 S1 = {'a','b','c','d'}
      3 del S1
----> 4 S1

NameError: name 'S1' is not defined
```

Basic Methematical operations

Join Two Sets

- we able to use the union() or update() method for Join Two Sets.

In [42]:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}
set3 = set1.union(set2)
print(set3)
```

```
{'b', 1, 2, 'c', 3, 'a'}
```

In [92]:

```
{"a", "b" , "c"}.union([1, 2, 3])
```

Out[92]:

```
{1, 2, 3, 'a', 'b', 'c'}
```

In [93]:

```
{"a", "b" , "c"}.union((1, 2, 3))
```

Out[93]:

```
{1, 2, 3, 'a', 'b', 'c'}
```

In [95]:

```
{"a", "b" , "c"}.union(range(1,4))
```

Out[95]:

```
{1, 2, 3, 'a', 'b', 'c'}
```

| is a symbol for union operation.

In [50]:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}
set3 = set1 | (set2)
print(set3)
```

```
{'b', 1, 2, 'c', 3, 'a'}
```

In [46]:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}
set1.update(set2) # this is a inplace operation so if we assign it to the new variable a
set1
```

Out[46]:

```
{1, 2, 3, 'a', 'b', 'c'}
```

In [48]:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}
set3 = set1.update(set2) # this is a inplace operation so if we assign it to the new var
# it returns a None
print(set3)
```

```
None
```

intersection

- The intersection() method will return a new set that only contains the items that are present in both sets.

In [52]:

```
S1 = set('Hitesh')
S2 = set('Reshma')
S3 = S1.intersection(S2)
S3
```

Out[52]:

```
{'e', 'h', 's'}
```

& is a symbol for intersection.

In [53]:

```
S1 = set('Hitesh')
S2 = set('Reshma')
S3 = S1 & S2
S3
```

Out[53]:

```
{'e', 'h', 's'}
```

intersection_update

- The intersection_update() method will Removes the items in this set that are not present in other specified set.
- it is a inplace operation
- for Ex. :- S1.intersection_update(S2) means it removes the items from S1 which are not in S2 modifies the S1.

In [81]:

```
S1 = {'apple','banana','cat','dog'}
S2 = {'apple','banana','cat','eelephant'}
S2.intersection_update(S1) # inplace operaton
S2
```

Out[81]:

```
{'apple', 'banana', 'cat'}
```

In [60]:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
x.intersection_update(y)
print(x)
```

```
{'apple'}
```

difference

- the s1.difference(s2) method Return a set that contains the items that only exist in set s1 and not in set s2.

In [73]:

```
S1 = set('Hitesh')
S2 = set('Reshma')
S3 = S1.difference(S2)
S3
```

Out[73]:

```
{'H', 'i', 't'}
```

In [74]:

```
S1 = set('Hitesh')
S2 = set('Reshma')
S3 = S2.difference(S1)
S3
```

Out[74]:

```
{'R', 'a', 'm'}
```

- symbol is used for Difference

In [82]:

```
S1 = set('Hitesh')
S2 = set('Reshma')
S3 = S2 - S1
S3
```

Out[82]:

```
{'R', 'a', 'm'}
```

difference_update

- the difference_update() method removes the items that exist in both sets.
- The difference_update() method is different from the difference() method, because the difference() method returns a new set, without the unwanted items, and the difference_update() method removes the unwanted items from the original set.

In [75]:

```
S1 = set('Hitesh')
S2 = set('Reshma')
S1.difference_update(S2) # inplace operation
S1
```

Out[75]:

```
{'H', 'i', 't'}
```

In [77]:

```
S1 = set('Hitesh')
S2 = set('Reshma')
S2.difference_update(S1) # inplace operation
S2
```

Out[77]:

```
{'R', 'a', 'm'}
```

symmetric_difference

- symmetric_difference() method will return a new set that contains only the elements that are NOT present in both sets.

In [69]:

```
S1 = set('Hitesh')
S2 = set('Reshma')
S3 = S1.symmetric_difference(S2)
S3
```

Out[69]:

```
{'H', 'R', 'a', 'i', 'm', 't'}
```

^ symbol is used for Symmetric difference

In [83]:

```
S1 = set('Hitesh')
S2 = set('Reshma')
S3 = S1^S2
S3
```

Out[83]:

```
{'H', 'R', 'a', 'i', 'm', 't'}
```

symmetric_difference_update

- The symmetric_difference_update() method will keep only the elements that are NOT present in both sets.
- it is a inplace operation.
- it first removes the comman items from both sets and join that sets in first set.
- Ex. :- S1.symmetric_difference_update(S2) means it removes the comman items from both sets S1 and S2 and join that sets in S1 set where S2 set remains same as privious.

In [78]:

```
S1 = set('Hitesh')
S2 = set('Reshma')
S1.symmetric_difference_update(S2) # inplace operation
S1
```

Out[78]:

```
{'H', 'R', 'a', 'i', 'm', 't'}
```

In [80]:

```
S1 = set('Hitesh')
S2 = set('Reshma')
S2.symmetric_difference_update(S1) # inplace operation
S2
```

Out[80]:

```
{'H', 'R', 'a', 'i', 'm', 't'}
```

Subset

- Returns whether another set contains this set or not.
- The issubset() method returns True if all items in the set exists in the specified set, otherwise it returns False.
- result of this operation is always a boolean.

In [84]:

```
x = {"a", "b", "c"}
y = {"f", "e", "d", "c", "b"}
z = x.issubset(y)
z
```

Out[84]:

```
False
```

In [85]:

```
x = {"a", "b", "c"}
y = {"f", "a", "d", "c", "b"}
z = x.issubset(y)
z
```

Out[85]:

```
True
```

< is a symbol of subset

In [86]:

```
x = {"a", "b", "c"}
y = {"f", "a", "d", "c", "b"}
z = x < y
z
```

Out[86]:

```
True
```

In [90]:

```
{1, 2, 3}.issubset(range(-5, 5))
True
```

Out[90]:

```
True
```

Superset

- The issuperset() method returns True if all items in the specified set exists in the original set otherwise it returns False.
- result of this operation is always a boolean.

In [87]:

```
x = {"a", "b", "c"}
y = {"f", "a", "d", "c", "b"}
z = x.issuperset(y)
z
```

Out[87]:

```
False
```

In [88]:

```
x = {"a", "b", "c"}
y = {"f", "a", "d", "c", "b"}
z = y.issuperset(x)
z
```

Out[88]:

```
True
```

> is a symbol of superset

In [89]:

```
x = {"a", "b", "c"}
y = {"f", "a", "d", "c", "b"}
z = y > x
z
```

Out[89]:

```
True
```

Immutable constraint

- Sets are powerful and flexible objects, but they do have one constraint...
  - sets can only contain immutable (a.k.a. “hashable”) object types. Hence, lists and dictionaries cannot be embedded in sets, but tuples can if you need to store compound values.

In [98]:

```
S = set()
S.add([1, 2, 3]) # Only immutable objects work in a set
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
Input In [98], in <cell line: 2>()
      1 S = set()
----> 2 S.add([1, 2, 3])

TypeError: unhashable type: 'list'
```

In [99]:

```
S = set()
S.add({'a':1})
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
Input In [99], in <cell line: 2>()
      1 S = set()
----> 2 S.add({'a':1})

TypeError: unhashable type: 'dict'
```

In [100]:

```
S.add((1, 2, 3))
S # No list or dict, but tuple OK
```

Out[100]:

```
{(1, 2, 3)}
```

Frozen Set

- Sets themselves are mutable too, and so cannot be nested in other sets directly.
- if you need to store a set inside another set the frozenset built-in call works just like set but creates an immutable set that cannot change and thus can be embedded in other sets.

Set Comprehension

- Set Comprehension represents a creation of new set from an object that satisfy a given condition.
- Syntax - { expression for item in iterable object if_statement }

In [103]:

```
S1 = {i+1 for i in range(20)}
S1
```

Out[103]:

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
```

In [108]:

```
# similar for loop code
S1 = set()
for i in range(20):
    S1.add(i+1)
S1
```

Out[108]:

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
```

In [105]:

```
S2 = {i+1 for i in range(20) if i%2==0 }
S2
```

Out[105]:

```
{1, 3, 5, 7, 9, 11, 13, 15, 17, 19}
```

In [109]:

```
# similar for loop code
S1 = set()
for i in range(20):
    if i%2 == 0:
        S1.add(i+1)
S1
```

Out[109]:

```
{1, 3, 5, 7, 9, 11, 13, 15, 17, 19}
```

In [106]:

```
S2 = {i+1 for i in range(20) if i%2==0 if i%5==0}
S2
```

Out[106]:

```
{1, 11}
```

In [110]:

```
# similar for loop code
S1 = set()
for i in range(20):
    if i%2 == 0 and i%5==0:
        S1.add(i+1)
S1
```

Out[110]:

```
{1, 11}
```

In [111]:

```
{x for x in 'spam'} # Same as: set('spam')
```

Out[111]:

```
{'a', 'm', 'p', 's'}
```

In [112]:

```
{c * 4 for c in 'spam'} # Set of collected expression results
```

Out[112]:

```
{'aaaa', 'mmmm', 'pppp', 'ssss'}
```

Set comprehension with if-else statements

- Syntax :- { expression if_statement else expression for item in iterablle object }

In [118]:

```
S1 = { i if i%2==0 else i*100 for i in range(10)}
S1
```

Out[118]:

```
{0, 2, 4, 6, 8, 100, 300, 500, 700, 900}
```

In [121]:

```
S1 = { i if i==j else 'No' for i in range(3) for j in range(10)}
S1
```

Out[121]:

```
{0, 1, 2, 'No'}
```

Nested set comprehension

In [125]:

```
{(i,j) for i in range(5) for j in range(6) if i > 2 and j > 3}
```

Out[125]:

```
{(3, 4), (3, 5), (4, 4), (4, 5)}
```

In [126]:

```
{(i,j) for i in range(5) for j in range(6) if i > 2 if j > 3}
```

Out[126]:

```
{(3, 4), (3, 5), (4, 4), (4, 5)}
```

In [128]:

```
{(i,j) if i > j else (i+j) for i in range(5) for j in range(6)}
```

Out[128]:

```
{(1, 0),
 (2, 0),
 (2, 1),
 (3, 0),
 (3, 1),
 (3, 2),
 (4, 0),
 (4, 1),
 (4, 2),
 (4, 3),
 0,
 1,
 2,
 3,
 4,
 5,
 6,
 7,
 8,
 9}
```

Set Usecases

- sets useful for common tasks such as...
  - filtering out duplicates
  - isolating differences
  - performing order-neutral equality tests without sorting—in lists, strings, and all other iterable objects

Removing Duplicates

In [40]:

```
list1 = [2,3,4,5,2,6,4,6,4,2,1,4,1,6,3]
string1 = 'hiteshvaghela'
tuple1 = (2,3,4,5,2,5,4,6,5,2,1,4,1,6,3)
```

In [41]:

```
list1 = list(set(list1))
list1
```

Out[41]:

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

In [42]:

```
string1 = ''.join(list(set(string1)))
string1
```

Out[42]:

```
'giathsevl'
```

In [43]:

```
tuple1 = tuple(set(tuple1))
tuple1
```

Out[43]:

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

Isolate Difference

In [47]:

```
set(dir(list)) - set(dir(tuple))
```

Out[47]:

```
{'__delitem__',
 '__iadd__',
 '__imul__',
 '__reversed__',
 '__setitem__',
 'append',
 'clear',
 'copy',
 'extend',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort'}
```

In [49]:

```
set('HITESH') - set('HARDIK')
```

Out[49]:

```
{'E', 'S', 'T'}
```

Oreder Nuteral Equality Test

In [51]:

```
set('spam') == set('asmp')
```

Out[51]:

```
True
```

In [53]:

```
set('Reshma') == set('Ramesh')
```

Out[53]:

```
True
```

Boolean

- Boolean is also a python's core datatype.
- Boolean has a predefined True and False objects that are essentially just the integers 1 and 0 with custom display logic.
- Boolean also has a long supported special placeholder object called None.

In [54]:

```
1 > 2, 1 < 2 # Booleans
```

Out[54]:

```
(False, True)
```

- The bool() function allows you to evaluate any value, and give you True or False in return.

In [55]:

```
bool('spam') # Object's Boolean value
```

Out[55]:

```
True
```

- Any string is True, except empty strings.
- Any number is True, except 0.
- Any list, tuple, set, and dictionary are True, except empty ones.
- there are not many values that evaluate to False except empty values.such as...

- `()`, `[]`, `{}`, `""`, the number `0`, and the value `None`.
- `False` evaluates to False.

In [63]:

```
bool(False),bool(None),bool(0),bool(""),bool(()),bool([]),bool({})
```

Out[63]:

```
(False, False, False, False, False, False, False)
```

- Python has built-in function isinstance() that return a boolean value.
- isinstance() is used to determine if an object is of a certain data type or not.

In [64]:

```
x = 200
print(isinstance(x, int))
```

```
True
```

None object Usecases

- None object commonly used to initialize names and objects.

In [56]:

```
X = None # None placeholder
print(X)
```

```
None
```

In [57]:

```
L = [None] * 10 # Initialize a list of 10 Nones
L[9] = 'Reshma'
L
```

Out[57]:

```
[None, None, None, None, None, None, None, None, None, 'Reshma']
```

type object

- it is returned by the type built-in function.
- it is an object that gives the type of another object.

In [58]:

```
L = [1,2,3]
D = {'a':1,'b':2,'c':3}
S = {1,2,3,4}
T = (1,2,7,9)
St = 'Hitesh'
N = 818
B = True
X = None
type(L),type(D),type(S),type(T),type(St),type(N),type(B),type(X)
```

Out[58]:

```
(list, dict, set, tuple, str, int, bool, NoneType)
```

In [59]:

```
type(type(L))
```

Out[59]:

```
type
```

Type objcet Usecases

- most practical application of type object is...it allows code to check the types of the objects it processes.
- there are three ways to check the types of the objects in a Python script.

In [60]:

```
if type(L) == type([]): # Type testing
    print('yes')
```

```
yes
```

In [61]:

```
if type(L) == list: # Using the type name
     print('yes')
```

```
yes
```

In [62]:

```
if isinstance(L, list): # Object-oriented tests
     print('yes')
```

```
yes
```

In [ ]:

```

```