Lecture 3 : Python Basics (Part II)

Data Science, DST, UIC

In this lecture, we will cover the following topics:

1. Flow control statements: if, for, while, pass...
2. List
3. Tuple
4. String
5. Sequence
6. Set
7. Dictionary

1. Flow control statements

More frequently used flow control statements are the following:

1. if
2. for
3. while
4. pass

1.1 if

Note: in python we use ： and indentation to define code blocks instead of {} , therefore, use ： to enclose the statements of **if, for, while...**

- Each ： must be accompanied by **identation** in the next line, usually we use 4 blank spaces for indentation, sometimes the editor can do it automatically for you when you type "enter" after a ：, like in Jupyter Notebook.

The general Python syntax for a simple **if** statement is:

```
if condition :
    indentedStatementBlock
```

- If the condition is true, then do the indented statements. If the condition is not true, then skip the indented statements.

General Python **if-else** syntax is:

```
if condition :
    indentedStatementBlockForTrueCondition
else:
    indentedStatementBlockForFalseCondition
```

- These statement blocks can have any number of statements, and can include about any kind of statement.

In [1]:
```
a = 1
b = 2

if a<b:
    print("a is less than b")
else:
    print("a is not less than b")
```

```
a is less than b
```

Nested **if-else** statements:

In [2]:
```
a = 1
b = 1

if a<=b:
    if a<b:
        print("a is less than b")
    else:
        print("a equals to b")
else:
    print("a is greater than b")
```

```
a equals to b
```

If you have multiple **if** and **else** tests, you can use **if-elif** statements. The most elaborate syntax for an **if-elif-else** statement is indicated in general below:

```
 if condition1 :
      indentedStatementBlockForTrueCondition1
elif condition2 :
      indentedStatementBlockForTrueCondition2
elif condition3 :
      indentedStatementBlockForTrueCondition3
else:
      indentedStatementBlockForEachConditionFalse
```

The *if*, each *elif*, and the final *else* lines are all aligned. There can be any number of *elif* lines, each followed by an indented block. With this construction exactly **one** of the indented blocks is executed. It is the one corresponding to the first *True* condition, or, if all conditions are False, it is the block after the final *else* line.

In [3]:
```
a=2
b=1

if a<b:
    print("a is less than b")
elif a==b:
    print("a equals to b")
else:
    print("a is greater than b")
```

```
a is greater than b
```

A special format of **if...else** statement in assignment.

- Ternery operator or conditional expression

In [4]:
```
x = 1
result = "Y" if x>0 else "N"  # ternery operator;
result
```

Out[4]: 'Y'

There are two types of loops in Python, **for** and **while**.

1.2 for

For loops iterate over a given sequence, e.g., tuple, list and string.

Syntax of for loops:

```
for iterator_var in sequence:
    indentedStatementBlock
```

In [5]:
```
for i in (1,2,3): # iterate over a tuple; i=1; i=2; i=3
    print(i)
```

```
1
2
3
```

In [6]:
```
for i in [1,2,3]: # iterate over a list
    print(i)
```

```
1
2
3
```

For loops can iterate over a sequence of numbers using the `range` functions. Range function returns a new list with numbers of that specified range.

- `range(start, stop, step)` function returns an object that produces a sequence of integers from start (inclusive) to stop (exclusive) by step (default to 1). Start from 0 if start is not given explicitly.

In [7]:
```
for i in range(3):  # default start is 0, step is 1 range(0,3,1)
    print(i)
```

```
0
1
2
```

In [8]:
```
for i in range(1,3): # range(1,3,1)
    print(i)
```

```
1
2
```

In [9]:
```
for i in range(1,10,2):
    print(i)
```

```
1
3
5
7
9
```

Use for loop to iterate over index of a list.

In [10]:
```
a_list = list(range(1,5)) # a_list=[1,2,3,4]
for i in [1,3]:
    print(a_list[i]) # a_list[1] => 2; a_list[3]=4
```

```
2
4
```

break and **continue** in for loops

- `break` is used to exit a for loop, whereas `continue` is used to skip the current block, and return to the `for` statement.

In [11]:
```
for i in range(0,10,2): #[0,2,4,6,8]
    if i==6:
        break
    print(i)
```

```
0
2
4
```

In [12]:
```
for i in range(0,10,2):
    if i==6:
        continue
    print(i)
```

```
0
2
4
8
```

1.3 While

In python, **while** loop is used to execute a block of statements repeatedly until a given condition is satisfied. And when the condition becomes false, the line immediately after the loop in program is executed.

Syntax:

```
while expression:
   indentedStatementBlock
```

In [13]:
```
i=1
sum=0
while i<=10:
    sum += i  # 0+1+2+3+...+10 = 55
    i += 1
    # i++ not allowed in python

print(sum)
```

```
55
```

We can use **else** in loops. When the loop condition of `for` or `while` statement fails then code part in `else` is executed.

- If `break` statement is executed inside for loop then the `else` part is skipped.
- `else` part is executed even if there is a `continue` statement.

In [14]:
```
i=1
sum=0
while i<=10:
    sum += i
    i += 1
    if i == 6:
        continue
    print(i,sum)
else:
    print("here is else")
```

```
2 1
3 3
4 6
5 10
7 21
8 28
9 36
10 45
11 55
here is else
```

In [15]:
```
i=1
sum=0
while i<=10:
    sum += i
    i += 1
    if i==6:
        continue
    if i == 9:
        break
    print(i, sum)
else:
    print("here is else")
```

```
2 1
3 3
4 6
5 10
7 21
8 28
```

1.4 Pass

Pass means empty statement, the program will do nothing to this statement and go to next statement directly. It is useful as a placeholder when a statement is required syntactically, but no code needs to be execute.

In [16]:
```
a= 1
b= 0
if (a<=b):
    pass
else:
    print(b)
```

```
0
```

2. List

2.1 Definition

List is a mutable container (similar to ArrayList in Java), each element in the list has an index.

There are three ways to define a list.

1. Use **[ ]**

```
In [17]: test_list = [1,2,3,4,5,6,7,8,9]
         test_list
```

```
Out[17]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

1. Use Assignment

```
In [18]: test_list_2 = test_list
         test_list_2
```

```
Out[18]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

1. Use explicit conversion

```
In [19]: test_list_3 = list("I love data science")
         test_list_3
         # If no argument is given, the constructor creates a new empty list.
         # The argument must be iterable if specified.
         # b = list()
         # b
```

```
Out[19]: ['I',
          ' ',
          'l',
          'o',
          'v',
          'e',
          ' ',
          'd',
          'a',
          't',
          'a',
          ' ',
          's',
          'c',
          'i',
          'e',
          'n',
          'c',
          'e']
```

You can check the length of a list using **len()**

```
In [20]: len(test_list)
```

```
Out[20]: 9
```

2.2 Index

You can refer to an element in the list using index.

Note: the index of a list starts from 0 (which means the first element in the list) and ends at len(your_list)-1

```
In [21]: test_list
Out[21]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [22]: print(test_list[0])
         print(test_list[8])
1
9
```

```
In [23]: # test_list[9]  # list index out of range
```

You can even use negative integers in the index, for example, -1 which means start counting from the last element, etc

```
In [24]: print(test_list[-1])
         print(test_list[-5])
9
5
```

2.3 Slicing

We can get a part of lists using python's slicing operator (:) which has following syntax:

```
test_list[start:stop:step]
```

- which means slice the test_list from(and including) the **start** index, end at (but not including) the **stop** index, and the step size is **step**.
- either **start, stop, step** can be omitted and can also be negative integers.

```
In [25]: test_list
Out[25]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [26]: test_list[1:6:1]
Out[26]: [2, 3, 4, 5, 6]
```

```
In [27]: test_list[1:6] # step default to 1
Out[27]: [2, 3, 4, 5, 6]
```

```
In [28]: test_list[1:6:2]
Out[28]: [2, 4, 6]
```

Omit the **start** and **step** arguments, the default value of **start** is 0, **step** is 1

```
In [29]: test_list[:6] # test_list[0:6:1]
Out[29]: [1, 2, 3, 4, 5, 6]
```

Omit the **stop** and **step** arguments, the default value of **end** is length of the list, **step** is 1

```
In [30]: test_list[1:]
Out[30]: [2, 3, 4, 5, 6, 7, 8, 9]
```

Omit all arguments

```
In [31]: test_list[:]
Out[31]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Negative arguments

```
In [32]: test_list[:-1]  # start from 0, end at last index, step is 1
Out[32]: [1, 2, 3, 4, 5, 6, 7, 8]
```

Perform inverse traversal using slicing operator and `reversed()` function.

- Note: Slicing will not affect the original list

```
In [33]: test_list[::-1] # step=-1, reverse direction step is 1
Out[33]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
In [34]: list(reversed(test_list))   # the same as above, but use reversed()
Out[34]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
In [35]: test_list
Out[35]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

However, the following built-in function for list will change the list itself to its reverse

```
In [36]: test_list.reverse()
         test_list
Out[36]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

2.4 extend and append

In Python, use list methods `append()` and `extend()` to add items to a list or combine other lists. You can also use `+` to combine lists.

You can add an item to the end of a list with with `append()` .

In [37]:
```
test_list_1 = [1,2,3]
test_list_2 = [5,6,7]
test_list_1.append(4)
print(test_list_1)
```

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

If the argument is a list object, **append()** will append the list with another list as a whole.

In [38]:
```
test_list_1.append(test_list_2)
test_list_1
```

Out[38]:
```
[1, 2, 3, 4, [5, 6, 7]]
```

You can combine another list at the end with `extend()`. Different with `append()` method, all itemss are added to the end of the original list.

In [39]:
```
test_list_1 = [1,2,3]
test_list_2 = [4,5,6]
test_list_1.extend(test_list_2)  # the argument in extend method must be iterable
test_list_1
```

Out[39]:
```
[1, 2, 3, 4, 5, 6]
```

It is also possible to combine using the `+` operator instead of `extend()`. In the case of the `+` operator, a new list is returned. You can also add to the existing list with `+=`.

In [40]:
```
test_list_1 = [1,2,3]
test_list_2 = [4,5,6]
test_list_1 += test_list_2
test_list_1
```

Out[40]:
```
[1, 2, 3, 4, 5, 6]
```

2.5 List comprehensions

List comprehensions provide a concise way to create lists.

- It consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses. The expressions can be anything, meaning you can put in all kinds of objects in lists.
- The following is the basic structure of a list comprehension:

```
output_list = [expression for var in input_list if (var satisfies this condition)]
```

In [41]:
```
clist1 = [1 for i in range(10)] #[1,1,1.....,1]
clist1
```

Out[41]:
```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

In [42]:
```
clist2 = [i for i in range(1,10)] # [1,2,.....,9]
clist2
```

Out[42]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [43]:
```
clist3 = [i for i in range(1,10,2)]
clist3
```

Out[43]: [1, 3, 5, 7, 9]

In [44]:
```
type_list = [type(elements) for elements in [1,1.0,'1',True]]
type_list
```

Out[44]: [int, float, str, bool]

In [45]:
```
["my student id is %d"%i for i in range(5)] # string format specifier "%d %s"%(var1,var2), simi
lar to printf in C
```

Out[45]:
```
['my student id is 0',
 'my student id is 1',
 'my student id is 2',
 'my student id is 3',
 'my student id is 4']
```

2.6 insert and delete

`insert()` : Inserts an element before/at specified index.

Syntax: `list.insert(index, element)`

In [46]:
```
L = [1,2,3]
L.insert(1,0)
L
```

Out[46]: [1, 0, 2, 3]

`pop()` : Delete an element, index is not a necessary parameter, if not mentioned takes the last index.

Syntax: `list.pop([index])`

In [47]:
```
L = [1,2,3]
L.pop(2)        # delete by index
L
```

Out[47]: [1, 2]

In [48]:
```
L =[1,2,3]
L.pop()     # delete the last element
L
```

Out[48]: [1, 2]

`del` : Element to be deleted is mentioned using list name and index.

Syntax:

```
del list[index]
```

In [49]:
```
L = [1,2,3]
del L[1]        # delete by index
L
```

Out[49]: [1, 3]

`remove()` : Element to be deleted is mentioned using list name and element, not index.

Syntax: `list.remove(element)`

In [50]:
```
L = [1,2,3]
L.remove(2)  # compare with the pop function
L
```

Out[50]: [1, 3]

In [51]:
```
L = ['a','b','c']
L.remove('a')      # delete by value
L
```

Out[51]: ['b', 'c']

In [52]:
```
L = ['a','a','b','c','d']
L.remove('a')      # delete by value, only delete the first 'a', how to delete all 'a' in the li
st?
L
```

Out[52]: ['a', 'b', 'c', 'd']

In [53]:
```
L = ['a','a','b','c','d']
L = [x for x in L if x!='a']
L
```

Out[53]: ['b', 'c', 'd']

3. Tuple

Think about **tuple** as immutable **list**. The operations of **tuple** are similar to **list** except the elements in a tuple cannot be modified directly

In [54]:
```
test_tuple = 2,4,6,8,0
test_tuple
```

Out[54]: (2, 4, 6, 8, 0)

In [55]:
```
test_tuple = (2,4,6,8,0)  # the same as above
test_tuple
```

Out[55]: (2, 4, 6, 8, 0)

In [56]:
```
test_tuple = 2,4,6,8,0
# test_tuple[2] = 5
```

Exercise: check all the operations of **list** mentioned in previous sections and see if they can be applied to **tuple**

4. String

In Python, Strings are arrays of bytes representing Unicode characters. However, Python does not have a character data type, a single character is simply a string with a length of 1.

Strings in Python can be created using **single quotes** or **double quotes** or even **triple quotes**.

In [57]:
```
print('abc')                    #you can either use ' or "  to quote the strings
print("abc")
print("abc loves 'data science'")
print('abc loves "data science"')

# if you need strings with \n, use '''
print('''
        abc
        loves
        data
        science
        ''')
```

```
abc
abc
abc loves 'data science'
abc loves "data science"

        abc
        loves
        data
        science
```

Square brackets can be used to access elements of the string. To access a range of characters in the String, method of slicing is used.

In [58]:
```
str1= 'test'
str1[2]          # access single character of the string
```

Out[58]: 's'

In [59]:
```
str1[1:3]       # access a range of characters use slicing
```

Out[59]: 'es'

In Python, updation or deletion of characters from a String is not allowed. The following example will cause an error because item assignment or item deletion from a String is not supported.

In [60]:
```
str1='test'
# str[1]='a'
```

In Python, there are a few ways to concatenate – or combine - strings. In order to merge two strings into a single string, you may use the `+` operator.

In [61]:
```
'I'+'love'+'DS'
```

Out[61]: 'IloveDS'

The `join` method is used to concatenate a list of strings.

In [62]:
```
print('**'.join(['I','love','DS']))  # reverse string using join method

print(''.join(reversed("hello")))
```

```
I**love**DS
olleh
```

`strip` method will delete all the spaces, tabs, etc before or after letters, but not the ones in the middle

In [63]:
```
"      I  love  \tz   \t DS \n".strip()
```

Out[63]: 'I love \tz   \t DS'

In [64]:
```
str(1234) # return the string format of the object
```

Out[64]: '1234'

In [65]:
```
'data science'.upper()
```

Out[65]: 'DATA SCIENCE'

In [66]:
```
'DATA SCIENCE'.lower()
```

Out[66]: 'data science'

Special charactor and paths.

In [67]:
```
my_file_path = 'C:\windows\desktop' # as \ is escaping character, special character \ will be es
caped to \\
my_file_path
```

Out[67]: 'C:\\windows\\desktop'

In [68]:
```
my_url='http://www.uic.edu.hk'
my_url
```

Out[68]: 'http://www.uic.edu.hk'

Strings in Python can be formatted with the use of `format()` method which is very versatile and powerful tool for formatting of Strings.

- Format method in String contains curly braces `{}` as placeholders which can hold arguments according to **position** or **keyword** to specify the order.

In [69]:
```
s1 = "{} {} {}".format('I', 'love', 'DS')
print(" String in default order: ", s1)
```

```
 String in default order:  I love DS
```

In [70]:
```
s2 = "{0} {2} {1}".format('I','DS','love')
print("String in positional order: ", s2)
```

```
String in positional order:  I love DS
```

In [71]:
```
s3 = "{ffff} {s} {t}".format(ffff = 'I', s = 'love', t = 'DS')
print("String in order of Keywords: ", s3)
```

```
String in order of Keywords:  I love DS
```

5. Sequence

In python, sequence is not a specific data type, but a category of container data types, which contains a set of elements with specific order. Typical sequence includes **string**, **list**, and **tuple**. Particularly, **set** is not sequence, because the elements in **set** do not have order.

```
In [72]: mySeq1="I love data science"
         mySeq2=[5,2,0]
         mySeq3=(1,3,1,4)
```

```
In [73]: not_my_seq = {'a','c','b'}
```

There are some common operations on data types that are sequences

5.1 Slicing

Syntax of slicing on sequence: `my_seq[start:stop:step]`

```
In [74]: mySeq1[0:3],mySeq2[0:3],mySeq3[0:3]
Out[74]: ('I l', [5, 2, 0], (1, 3, 1))
```

5.2 Indexing

```
In [75]: my_str="I love data science"
         my_str[0]
Out[75]: 'I'
```

```
In [76]: my_list = [1,2,3,4,5]
         my_list[-1]
Out[76]: 5
```

```
In [77]: my_tuple = ('a','b','c','d')
         my_tuple[3]
Out[77]: 'd'
```

5.3 Iteration

Sequence is **iterable**, can be used after **in** keyword in **for** statements

In [78]:
```
my_str = 'I love data science'
for i in my_str:
    print(i)
```

```
I

l
o
v
e

d
a
t
a

s
c
i
e
n
c
e
```

In [79]:
```
my_index = [0,2,4,6,8]
my_list = [100,98,99,85,92,90,80,60,80,88]
for i in my_index:
    print(my_list[i])
```

```
100
99
92
80
80
```

5.4 Unpacking assignment

Sequence unpacking in python allows you to take objects in a collection and store them in variables for later use.

- In unpacking of a sequence, we extract items stored in the sequence into separate variables.
- The number of variables on left hand side should be equal to number of values in given sequence.

In [80]:
```
student_list = ['amy','bob','candy','david']
s1,s2,s3,s4 = student_list
print(s1,s2,s3,s4)
```

```
amy bob candy david
```

5.5 The repeating operator *

Repetition operator is denoted by a ＊ symbol and is useful for repeating sequences to a certain length.

In [81]:
```
my_tuple = (1,2,3)
my_tuple*3
```

Out[81]: (1, 2, 3, 1, 2, 3, 1, 2, 3)

In [82]:
```
my_str = 'i love data science '
my_str*3
```

Out[82]:
```
'i love data science i love data science i love data science '
```

5.6 Common functions

Functions that can be applied to all sequence type(tuple, list, string, etc) variables.

- len
- sorted
- reversed
- enumerate
- zip

In [83]:
```
my_str = 'i love data science'
my_list= [1,9,6,5,4,8]
my_tuple = [1,2,3,6,5,4]
len(my_str),len(my_list),len(my_tuple)    # return the length of a sequence
```

Out[83]:
```
(19, 6, 6)
```

In [84]:
```
sorted(my_str),sorted(my_list),sorted(my_tuple,reverse=True)  #return a sorted sequence of orig
inal sequence
```

Out[84]:
```
([' ',
  ' ',
  ' ',
  'a',
  'a',
  'c',
  'c',
  'd',
  'e',
  'e',
  'e',
  'i',
  'i',
  'l',
  'n',
  'o',
  's',
  't',
  'v'],
 [1, 4, 5, 6, 8, 9],
 [6, 5, 4, 3, 2, 1])
```

In [85]:
```
reversed(my_str), reversed(my_list),reversed(my_tuple)  #the function returns an iterator which
will be discussed later
```

Out[85]:
```
(<reversed at 0x2335d6a79b0>,
 <list_reverseiterator at 0x2335d6a7b00>,
 <list_reverseiterator at 0x2335d6a7b70>)
```

In [86]:
```
list(reversed(my_str)), list(reversed(my_list)),list(reversed(my_tuple))
```

Out[86]:
```
(['e',
  'c',
  'n',
  'e',
  'i',
  'c',
  's',
  ' ',
  'a',
  't',
  'a',
  'd',
  ' ',
  'e',
  'v',
  'o',
  'l',
  ' ',
  'i'],
 [8, 4, 5, 6, 9, 1],
 [4, 5, 6, 3, 2, 1])
```

Use **enumerate** to trace and show the index of each element in a sequence

In [87]:
```
list(enumerate(my_str))
#The enumerate object yields pairs containing a count (from start, which defaults to zero) and
# a value yielded by the iterable argument.
# enumerate?
```

Out[87]:
```
[(0, 'i'),
 (1, ' '),
 (2, 'l'),
 (3, 'o'),
 (4, 'v'),
 (5, 'e'),
 (6, ' '),
 (7, 'd'),
 (8, 'a'),
 (9, 't'),
 (10, 'a'),
 (11, ' '),
 (12, 's'),
 (13, 'c'),
 (14, 'i'),
 (15, 'e'),
 (16, 'n'),
 (17, 'c'),
 (18, 'e')]
```

`zip(sequence1,sequence2,..)` returns a zip object whose `.__next__()` method returns a tuple where the i^{th} element comes from the i^{th} iterable argument.

In [88]:
```
print(my_list,my_tuple)
list(zip(my_list, my_tuple))        # combine sequences element by element
```

```
[1, 9, 6, 5, 4, 8] [1, 2, 3, 6, 5, 4]
```

Out[88]:
```
[(1, 1), (9, 2), (6, 3), (5, 6), (4, 5), (8, 4)]
```

6. Set

Set in Python is a data structure equivalent to sets in mathematics. It may consist of **unordered** collections of **unique** elements; the order of elements in a set is undefined.

- You can add and delete elements of a set, you can iterate the elements of the set, you can perform standard operations on sets (union, intersection, difference).

In [89]:
```
my_set = {1,2,3,4,1,2,5,6} # create set using {}
my_set
```

Out[89]: {1, 2, 3, 4, 5, 6}

In [90]:
```
my_set_2 = my_set # create set by assignment
my_set_2
```

Out[90]: {1, 2, 3, 4, 5, 6}

In [91]:
```
my_list = [1,1,2,2,3,3]
my_set_3 = set(my_list)  # create set using explicit conversion
my_set_3
```

Out[91]: {1, 2, 3}

Once a set is created, you cannot chagne its items, but you can add new items using `add()` method. To add items from another set into the current set, use the `update` method.

In [92]:
```
my_set = {1,2,3}
my_set.add(4)
print(my_set)
my_set.update({'a','b','c'})
my_set
```

```
{1, 2, 3, 4}
```

Out[92]: {1, 2, 3, 4, 'a', 'b', 'c'}

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

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

In [93]:
```
my_set.remove(1)
print(my_set)
my_set.discard(1)
print(my_set)
# my_set.remove(1)
```

```
{2, 3, 4, 'b', 'a', 'c'}
{2, 3, 4, 'b', 'a', 'c'}
```

Use **in** to check if an element belongs to a set

In [94]:
```
1 in my_set_3
```

Out[94]: True

In [95]:
```
'1' in my_set_2
```

Out[95]:
```
False
```

Set does not support indexing, because it is unordered

In [96]:
```
# my_set[0]
```

In [97]:
```
my_set_a = {1,2,3,4,5}
my_set_b = {3,4,5,6,7}
```

In [98]:
```
3 in my_set_a, 3 not in my_set_b
```

Out[98]:
```
(True, False)
```

In [99]:
```
my_set_a == my_set_b, my_set_a != my_set_b
```

Out[99]:
```
(False, True)
```

In [100]:
```
{1,2,3} < my_set_a # is subset ?
# my_set_a.issubset?
{1,2,3}.issubset(my_set_a)
```

Out[100]:
```
True
```

In [101]:
```
# my_set_b.issuperset, my_set_b.union, my_set_b.difference
# my_set_a.symmetric_difference?
print(my_set_b > {5,6,7} ) # is superset ?
print(my_set_a | my_set_b )  #union
print(my_set_a & my_set_b ) #intersection
print(my_set_a - my_set_b ) #difference
print(my_set_a ^ my_set_b )# symmetric_difference,returns new set with elements in either s or t
but not both
```

```
True
{1, 2, 3, 4, 5, 6, 7}
{3, 4, 5}
{1, 2}
{1, 2, 6, 7}
```

set is mutable, **frozenset** is immutable. **frozenset** is used when you don't want the elements in a set to be changed accidentally

In [102]:
```
my_set = {1,2,3}
my_set.add(4)
my_set.remove(1)
my_set.update({3,5})
my_set.pop() # pop of set will delte an element at random
my_set
```

Out[102]:
```
{3, 4, 5}
```

In [103]:
```
my_frozenset = frozenset({1,2,3})
my_frozenset
```

Out[103]:
```
frozenset({1, 2, 3})
```

In [104]:
```
# my_frozenset.add(4)
```

7. Dictionary

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

- It is best to think of a dictionary as a set of `key: value` pairs, with the requirement that the keys are unique (within one dictionary).

Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any **immutable type** (e.g., numeric data type, string, tuple, frozenset).

The main operations on a dictionary are storing a value with some key and extracting the value given the key.

In [105]:
```
my_dict = {'id':123, 'name':'abc', 'age':18, 8:True}
my_dict
# dic = {{1,2,3}:'a'} # error
# dic = {(1,2,3):'a'}
# dic
```

Out[105]:
```
{'id': 123, 'name': 'abc', 'age': 18, 8: True}
```

Question: What are the keys and what are the values to each key in the above definition?

In [106]:
```
my_dict['id']
```

Out[106]:
```
123
```

Note: you can always use keys to access a particular value

In [107]:
```
my_dict[8]
```

Out[107]:
```
True
```

In [108]:
```
my_dict['name']= 'abcd'  # update dictionary
my_dict
```

Out[108]:
```
{'id': 123, 'name': 'abcd', 'age': 18, 8: True}
```

In [109]:
```
my_dict['gender'] = 'male' # add new item to dictionary
my_dict
```

Out[109]:
```
{'id': 123, 'name': 'abcd', 'age': 18, 8: True, 'gender': 'male'}
```

To update the dictionary with the items from the given arguments (must be a list of key:value pairs or another dictionary), you can use `update()` method.

In [110]:
```
newDict ={"1":"a","2":"b"}
my_dict.update(newDict)
my_dict
```

Out[110]:
```
{'id': 123,
 'name': 'abcd',
 'age': 18,
 8: True,
 'gender': 'male',
 '1': 'a',
 '2': 'b'}
```

To remove items from a dictionary, you can use the `pop()` method, which removes the item with the specified key name.

In [111]:
```
my_dict.pop("1")
my_dict
```

Out[111]:
```
{'id': 123, 'name': 'abcd', 'age': 18, 8: True, 'gender': 'male', '2': 'b'}
```

Dictionaries can be iterated over, just like a list. However, a dictionary, unlike a list, does not keep the order of the values stored in it. To iterate over key value pairs, use the following syntax:

In [112]:
```
for k in my_dict:
    print(k, my_dict[k])
```

```
id 123
name abcd
age 18
8 True
gender male
2 b
```

In [113]:
```
for k,v in my_dict.items():
    print("Key:%s; Value:%s"%(k,v))
```

```
Key:id; Value:123
Key:name; Value:abcd
Key:age; Value:18
Key:8; Value:True
Key:gender; Value:male
Key:2; Value:b
```

In [114]:
```
del my_dict['gender']; # It deletes only the key with the name 'gender'
print(my_dict)

my_dict.clear();# The above code removes all entries in dictionary & makes the dictionary empty
print(my_dict)
```

```
{'id': 123, 'name': 'abcd', 'age': 18, 8: True, '2': 'b'}
{}
```