The split and join

If we want to "break" a string into a list of many strings then we can use the `split` command:

In [57]:
```
"a+b+c".split('+')
```

Out[57]:
```
['a', 'b', 'c']
```

In [58]:
```
"hello world".split(' ')
```

Out[58]:
```
['hello', 'world']
```

In [59]:
```
"I like to move it move it".split('move')
```

Out[59]:
```
['I like to ', ' it ', ' it']
```

In [60]:
```
a = '''
ἄνδρα μοι ἔννεπε, μοῦσα, πολύτροπον, ὃς μάλα πολλὰ
πλάγχθη, ἐπεὶ Τροίης ἱερὸν πτολίεθρον ἔπερσεν·
πολλῶν δ᾽ ἀνθρώπων ἴδεν ἄστεα καὶ νόον ἔγνω,
πολλὰ δ᾽ ὅ γ᾽ ἐν πόντῳ πάθεν ἄλγεα ὃν κατὰ θυμόν,
ἀρνύμενος ἥν τε ψυχὴν καὶ νόστον ἑταίρων.
ἀλλ᾽ οὐδ᾽ ὣς ἑτάρους ἐρρύσατο, ἱέμενός περ·
αὐτῶν γὰρ σφετέρῃσιν ἀτασθαλίῃσιν ὄλοντο,
νήπιοι, οἳ κατὰ βοῦς Ὑπερίονος Ἠελίοιο
ἤσθιον· αὐτὰρ ὁ τοῖσιν ἀφείλετο νόστιμον ἦμαρ.
'''
a.split('\n')
```

Out[60]:
```
['',
 'ἄνδρα μοι ἔννεπε, μοῦσα, πολύτροπον, ὃς μάλα πολλὰ',
 'πλάγχθη, ἐπεὶ Τροίης ἱερὸν πτολίεθρον ἔπερσεν·',
 'πολλῶν δ᾽ ἀνθρώπων ἴδεν ἄστεα καὶ νόον ἔγνω,',
 'πολλὰ δ᾽ ὅ γ᾽ ἐν πόντῳ πάθεν ἄλγεα ὃν κατὰ θυμόν,',
 'ἀρνύμενος ἥν τε ψυχὴν καὶ νόστον ἑταίρων.',
 'ἀλλ᾽ οὐδ᾽ ὣς ἑτάρους ἐρρύσατο, ἱέμενός περ·',
 'αὐτῶν γὰρ σφετέρῃσιν ἀτασθαλίῃσιν ὄλοντο,',
 'νήπιοι, οἳ κατὰ βοῦς Ὑπερίονος Ἠελίοιο',
 'ἤσθιον· αὐτὰρ ὁ τοῖσιν ἀφείλετο νόστιμον ἦμαρ.',
 '']
```

A call to `split` without any argument removes all types of spaces (space, tabs and new lines) between the words in a string:

In [61]:
```
"hello                world".split()
```

Out[61]:
```
['hello', 'world']
```

The `join` method does the opposite. Takes a list of strings and joins them into a string:

In [62]:
```
'+'.join(['a','b','c'])
```

Out[62]:
```
'a+b+c'
```

In [64]:
```
' '.join(['hello', 'world'])
```

Out[64]:
```
'hello world'
```

In [65]:
```
print ('\n'.join(['line 1', 'line 2']))
```

```
line 1
line 2
```

Functions `all` and `any`

`all` returns `True` if **all** the items in a list are `True`

In [49]:
```
all([True, True, True])
```

Out[49]:
```
True
```

In [50]:
```
all([True, False, True])
```

Out[50]:
```
False
```

In [51]:
```
all([3,4,5,4,5])
```

Out[51]:
```
True
```

In [52]:
```
all([3,4,5,'',4,5])
```

Out[52]:
```
False
```

`any` returns `True` if any of the items (even one) in the list is `True` :

In [53]:
```
any([False, False, False])
```

Out[53]:
```
False
```

In [54]:
```
any([False, False, False, "mitsos"])
```

Out[54]:
```
True
```

Attention!:

- The `all` value of the empty list is `True`
- The `any` value of the empty list is `False`

In [55]:
```
all([])
```

Out[55]:
```
True
```

In [56]:
```
any([])
```

Out[56]:
```
False
```

The `range` function

The `range` function creates something (*) that represents a numeric sequence.

(*) This "something" is called a generator and we will talk more about it in the next

lecture

In [22]:
```
range(1,10)
```

Out[22]:
```
range(1, 10)
```

If we apply the `list` function in the object returned from range then we can generate a list of sequential elements:

In [1]:
```
list(range(10)) # From 0 to 10 (without 10)
```

Out[1]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [2]:
```
list(range(5,10)) # From 5 to 10 (without 10)
```

Out[2]:
```
[5, 6, 7, 8, 9]
```

In [3]:
```
list(range(1,11,2)) # From 1 to 11 (without 11) with step 2
```

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

This arithmetic progression can also be in a descending order:

In [28]:
```
list(range(11,1,-2))
```

Out[28]:
```
[11, 9, 7, 5, 3]
```

In [29]:
```
list(range(10,1,-1))
```

Out[29]:
```
[10, 9, 8, 7, 6, 5, 4, 3, 2]
```

The `list(range(...))` returns a list with which we can perform operations normally as we have seen:

In [31]:
```
a = list(range(100, 120, 5)) + ["mitsos"]
print (a)
```

```
[100, 105, 110, 115, 'mitsos']
```

Why whenever we see `"XYZ"[a:b]` , `[1,2,3][a:b]` , `range(a,b)` this means from `a` to `b` **WITHOUT** `b` ? This story goes back a long way. Generally, in computing when we want N elements, then based on an old contract, the first element has index 0, the second 1, etc. So when we say `range(10)` the python generates a list from 0 to 9. When we type range(5,7) then in essence we ask python for a list of 2 items (7-5). The first according to the same convention will be "where the numbering starts" ie 5. Since the list must have 2 items then the second will be the next one ie 6. This numbering is very convenient for some mathematical calculations as well. Some additional examples:

In [32]:
```
list(range(3,10,2))
```

Out[32]:
```
[3, 5, 7, 9]
```

In [33]:
```
list(range(3,11,2))
```

Out[33]:
```
[3, 5, 7, 9]
```

In [34]:
```
list(range(3,12,2))
```

Out[34]:
```
[3, 5, 7, 9, 11]
```

In [35]:
```
list(range(10)) # list(range(0,10))
```

Out[35]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [36]:
```
list(range(5,7)) # list(range(a,b)) # b-a
```

Out[36]:
```
[5, 6]
```

The `zip` function

With `zip` we can 'join' two lists into one list of sublists:

In [70]:
```
list(zip([1,2,3], ['a', 'b', 'c']))
```

Out[70]:
```
[(1, 'a'), (2, 'b'), (3, 'c')]
```

The `enumerate` function

The `enumerate` function takes as an argument a list and creates another list which contains both the indexes and the elements of the first list:

In [109…
```
a = ["python", "mitsos", "Crete"]
print (list(enumerate(a)))
```

```
[(0, 'python'), (1, 'mitsos'), (2, 'Crete')]
```

The `enumerate` functions does exactly the same as this combination of `range` and `zip` :

In [110…
```
a = ["python", "mitsos", "Crete"]
print (list(zip(range(len(a)),a)))
```

```
[(0, 'python'), (1, 'mitsos'), (2, 'Crete')]
```

The `for` syntax

With the `for` syntax we can repeat commands for each element in a list

In [37]:
```
for x in [1,4,6]:
    print (x)
```

```
1
4
6
```

In [38]:
```
for x in [1,4,6]:
    print ("The number is:", x)
```

```
The number is: 1
The number is: 4
The number is: 6
```

In [39]:
```
for x in range(1,10):
    print ("The number is:", x)
```

```
The number is: 1
The number is: 2
```

```
The number is: 3
The number is: 4
The number is: 5
The number is: 6
The number is: 7
The number is: 8
The number is: 9
```

In [40]:
```
for i in range(1,10,3):
    print ("Hello:", i)
```

```
Hello: 1
Hello: 4
Hello: 7
```

If the commands we want to repeat are more than 1 then it is **MANDATORY** to put them on the next line and further inside. This is called mandatory indentation or off-side rule !

In [41]:
```
for i in range(1,10,3):
    print ("command A:", i)
    print ("command B:", i)
```

```
command A: 1
command B: 1
command A: 4
command B: 4
command A: 7
command B: 7
```

If there is a `for` inside another `for` then the following lines must be entered even further:

In [42]:
```
for i in range(1,5):
    for j in range(1,5):
        print (i,j)
```

```
1 1
1 2
1 3
1 4
2 1
2 2
2 3
2 4
3 1
3 2
3 3
3 4
4 1
4 2
4 3
4 4
```

Example: The multiplication table:

In [43]:
```
for i in range(1,11):
    for j in range(1,11):
        print ("{} X {} = {}".format(i,j,i*j))
    print ("=" * 10)
```

```
1 X 1 = 1
1 X 2 = 2
1 X 3 = 3
1 X 4 = 4
1 X 5 = 5
```

1 X 6 = 6
1 X 7 = 7
1 X 8 = 8
1 X 9 = 9
1 X 10 = 10
==========
2 X 1 = 2
2 X 2 = 4
2 X 3 = 6
2 X 4 = 8
2 X 5 = 10
2 X 6 = 12
2 X 7 = 14
2 X 8 = 16
2 X 9 = 18
2 X 10 = 20
==========
3 X 1 = 3
3 X 2 = 6
3 X 3 = 9
3 X 4 = 12
3 X 5 = 15
3 X 6 = 18
3 X 7 = 21
3 X 8 = 24
3 X 9 = 27
3 X 10 = 30
==========
4 X 1 = 4
4 X 2 = 8
4 X 3 = 12
4 X 4 = 16
4 X 5 = 20
4 X 6 = 24
4 X 7 = 28
4 X 8 = 32
4 X 9 = 36
4 X 10 = 40
==========
5 X 1 = 5
5 X 2 = 10
5 X 3 = 15
5 X 4 = 20
5 X 5 = 25
5 X 6 = 30
5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
5 X 10 = 50
==========
6 X 1 = 6
6 X 2 = 12
6 X 3 = 18
6 X 4 = 24
6 X 5 = 30
6 X 6 = 36
6 X 7 = 42
6 X 8 = 48
6 X 9 = 54
6 X 10 = 60
==========
7 X 1 = 7
7 X 2 = 14
7 X 3 = 21

```
7 X 4 = 28
7 X 5 = 35
7 X 6 = 42
7 X 7 = 49
7 X 8 = 56
7 X 9 = 63
7 X 10 = 70
==========
8 X 1 = 8
8 X 2 = 16
8 X 3 = 24
8 X 4 = 32
8 X 5 = 40
8 X 6 = 48
8 X 7 = 56
8 X 8 = 64
8 X 9 = 72
8 X 10 = 80
==========
9 X 1 = 9
9 X 2 = 18
9 X 3 = 27
9 X 4 = 36
9 X 5 = 45
9 X 6 = 54
9 X 7 = 63
9 X 8 = 72
9 X 9 = 81
9 X 10 = 90
==========
10 X 1 = 10
10 X 2 = 20
10 X 3 = 30
10 X 4 = 40
10 X 5 = 50
10 X 6 = 60
10 X 7 = 70
10 X 8 = 80
10 X 9 = 90
10 X 10 = 100
==========
```

We can also repeat using a string instead of a list:

In [44]:
```
for letter in "python":
    print (letter)
```

```
p
y
t
h
o
n
```

If a list has sub-lists with more than 1 item then we can use more than 1 variable in `for` :

In [45]:
```
a = [[2, "Crete"], [3, "Cyprus"], [4, "Majiorca"]]
for x, y in a:
    print ("Number: {} Island: {}".format(x,y))
```

```
Number: 2 Island: Crete
Number: 3 Island: Cyprus
Number: 4 Island: Majiorca
```

Of course the same can be done if it has sub-lists with 3 items etc ..

In [46]:
```
a = [[1,2,3], ["a", "b", "c"]]
for x,y,z in a:
    print ("{} {} {}".format(x,y,z))
```

```
1 2 3
a b c
```

The `if` syntax (and any other python syntax) can be inside a `for` :

In [47]:
```
for i in range(1,10):
    if i>5:
        print (i)
```

```
6
7
8
9
```

In [ ]:
```
for i in range(1,10):
    if i>=5:
        print (i)
```

break and continue

With the `break` command we can "stop" the iteration inside a for. When the computer "sees" `break` then it directly exits `for` :

In [67]:
```
for i in range(1,10):
    print (i)
    if i>5:
        break # Get out from for !!!
```

```
1
2
3
4
5
6
```

With the `continue` command we can ignore all the resr of the commands in an iteration and move on to the next item in the iteration. Or else, we can move on, or.. continue to the next item in the iteration.

In [68]:
```
for i in range(1,10):
    if i == 5: # ΔΕΝ ΤΥΠΩΝΕΙ ΤΟ 5
        continue
    print (i) # TO PRINT **DEN** EINAI MESA STHN IF! (einai mesa sth for)
```

```
1
2
3
4
6
7
8
9
```

Caution! anything that is below (and in the same indentation) with `continue` and `break` is ignored!

In [4]:
```
for i in range(1,10):
    if i == 5:
        continue
        print (i) #  This command is ignored
```

Also having as a last command in an iteration the `continue` command, is redundant and does not make any difference. Python does this automatically..

In [5]:
```
# Please don't do this!
for x in [1,2,3]:
    print (x)
    continue # <-- This is redundant
```

```
1
2
3
```

List Comprehensions

List Comprehension is a new syntax for creating a list from another list. The official description is here . The general form is:

```
a = [expression for variable in LIST]
```

Which is equivalent to:

```
a = []
for variable in LIST:
    a.append(expression)
```

For example:

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

In [2]:
```
b = [x+1 for x in a]
print (b)
```

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

The above is equivalent to:

In [3]:
```
a = [1,2,3]
b = []
for x in a:
    b.append(x+1)
print (b)
```

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

Some other examples:

In [5]:
```
a = ["a", "b", "c"]
["hello: " + x for x in a]
```

Out[5]:
```
['hello: a', 'hello: b', 'hello: c']
```

In [6]:
```
a = ["a", "b", "c"]
b  = [x * 3 for x in a]
print (b)
```

```
['aaa', 'bbb', 'ccc']
```

In [8]:
```
a = [1,2,3,4,5,6]
[x/2 for x in a]
```

Out[8]:
```
[0.5, 1.0, 1.5, 2.0, 2.5, 3.0]
```

We can also use the `if` syntax in a list comprehensions. The general form is:

```
a = [expression_1 for variable in LIST if expression_2]
```

Which is equivalent to:

```
a = []
for variable in LIST:
    if expression_2:
        a.append(expression_1)
```

Examples:

In [10]:
```
a = [1,2,3,4,5,6]
[x/2 for x in a if x>4]
```

Out[10]:
```
[2.5, 3.0]
```

This is equivalent to:

In [12]:
```
a = [1,2,3,4,5,6]
b = []
for x in a:
    if x>4:
        b.append(x/2)
print (b)
```

```
[2.5, 3.0]
```

Another example. Let the list be:

In [13]:
```
a = [1,2,3,4,5,4,3,5,6,7,8,7,6,5,5,4]
```

What are all the items that have a value of 4?

First method:

In [14]:
```
[i for i, x in enumerate(a) if x==4]
```

Out[14]:
```
[3, 5, 15]
```

Second method:

In [16]:
```
a = [1,2,3,4,5,4,3,5,6,7,8,7,6,5,5,4]
[x for x in range(len(a)) if a[x] == 4]
```

Out[16]:
```
[3, 5, 15]
```

In [18]:
```
list(range(len(a)))
```

Out[18]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Also a list comprehension can have more than 1 `for` :

In [19]:
```
a = [1,2,3]
b = ["a", "b", "c"]
["{}{}".format(x,y) for x in a for y in b]
```

Out[19]:
```
['1a', '1b', '1c', '2a', '2b', '2c', '3a', '3b', '3c']
```

This is equivalent to:

In [20]:
```
c = []
for x in a:
    for y in b:
        c.append("{}{}".format(x,y))
print (c)
```

```
['1a', '1b', '1c', '2a', '2b', '2c', '3a', '3b', '3c']
```

By using list comprehension we can produce a string that contains the whole multiplication table:

In [21]:
```
print ('\n'.join(["{} X {} = {}".format(x,y,x*y) for x in range(1,11) for y
```

```
1 X 1 = 1
1 X 2 = 2
1 X 3 = 3
1 X 4 = 4
1 X 5 = 5
1 X 6 = 6
1 X 7 = 7
1 X 8 = 8
1 X 9 = 9
1 X 10 = 10
2 X 1 = 2
2 X 2 = 4
2 X 3 = 6
2 X 4 = 8
2 X 5 = 10
2 X 6 = 12
2 X 7 = 14
2 X 8 = 16
2 X 9 = 18
2 X 10 = 20
3 X 1 = 3
3 X 2 = 6
3 X 3 = 9
3 X 4 = 12
3 X 5 = 15
3 X 6 = 18
3 X 7 = 21
3 X 8 = 24
3 X 9 = 27
3 X 10 = 30
4 X 1 = 4
4 X 2 = 8
4 X 3 = 12
4 X 4 = 16
4 X 5 = 20
4 X 6 = 24
4 X 7 = 28
4 X 8 = 32
4 X 9 = 36
4 X 10 = 40
5 X 1 = 5
5 X 2 = 10
```

```
5 X 3 = 15
5 X 4 = 20
5 X 5 = 25
5 X 6 = 30
5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
5 X 10 = 50
6 X 1 = 6
6 X 2 = 12
6 X 3 = 18
6 X 4 = 24
6 X 5 = 30
6 X 6 = 36
6 X 7 = 42
6 X 8 = 48
6 X 9 = 54
6 X 10 = 60
7 X 1 = 7
7 X 2 = 14
7 X 3 = 21
7 X 4 = 28
7 X 5 = 35
7 X 6 = 42
7 X 7 = 49
7 X 8 = 56
7 X 9 = 63
7 X 10 = 70
8 X 1 = 8
8 X 2 = 16
8 X 3 = 24
8 X 4 = 32
8 X 5 = 40
8 X 6 = 48
8 X 7 = 56
8 X 8 = 64
8 X 9 = 72
8 X 10 = 80
9 X 1 = 9
9 X 2 = 18
9 X 3 = 27
9 X 4 = 36
9 X 5 = 45
9 X 6 = 54
9 X 7 = 63
9 X 8 = 72
9 X 9 = 81
9 X 10 = 90
10 X 1 = 10
10 X 2 = 20
10 X 3 = 30
10 X 4 = 40
10 X 5 = 50
10 X 6 = 60
10 X 7 = 70
10 X 8 = 80
10 X 9 = 90
10 X 10 = 100
```

Some examples of use of the above

1. From a list take only those that have a specific

property

e.g. Take only odd numbers from a list

In [7]:
```
a = [1,2,3,4,5,6,7,8,9,10]
def f(x):
    return x%2==1
```

In [8]:
```
# 1st method
b = list(filter(f,a))
print (b)
```

```
[1, 3, 5, 7, 9]
```

In [9]:
```
# 2nd method
b = []
for x in a:
    if f(x):
        b.append(x)
print (b)
```

```
[1, 3, 5, 7, 9]
```

In [10]:
```
# 2rd method
b = []
for x in a:
    if not f(x):
        continue

    b.append(x)
print (b)
```

```
[1, 3, 5, 7, 9]
```

In [11]:
```
# 4th method
b = [x for x in a if f(x)]
print (b)
```

```
[1, 3, 5, 7, 9]
```

2. Count the number that has a specific property in a list

How many are odd numbers?

In [12]:
```
# 1st method
c = len(list(filter(f,a)))
print(c)
```

```
5
```

In [13]:
```
# 2nd method
c = sum(map(f,a))
print (c)
```

```
5
```

In [14]:
```
# 3rd method
c = 0
for x in a:
    if f(x):
        c += 1
print (c)
```

```
5
```

In [15]:
```
# 4th method
c = 0
for x in a:
    if not f(x):
        continue
    c += 1
print (c)
```

```
5
```

In [16]:
```
# 5th method
c = len([None for x in a if f(x)])
print (c)
```

```
5
```

In [17]:
```
# 6th method
c = sum(f(x) for x in a)
print (c)
```

```
5
```

3. Apply a function to all elements of a list

eg multiply with 10 all elements in a list

In [18]:
```
a = [1,2,3,4,5,6,7,8,9,10]
def f(x):
    return x*10
```

In [92]:
```
# 1st method
b = list(map(f,a))
print (b)
```

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

In [19]:
```
# 2nd method
b = []
for x in a:
    b.append(f(x))
print (b)
```

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

In [20]:
```
# 3rd method
b = [f(x) for x in a]
print (b)
```

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

4. A combination of all of the above

Find the sum of the multiplication with 10 of all the odd numbers in a list.

In [21]:
```
a = [1,2,3,4,5,6,7,8,9,10]
def f(x):
    return x%2==1
def g(x):
    return x*10
```

In [97]:
```
# 1st method
c = sum(map(g, filter(f, a)))
print (c)
```

```
250
```

In [22]:
```
# 2nd method
c = 0
for x in a:
    if f(x):
        c += g(x)
print (c)
```

```
250
```

In [23]:
```
# 3rd method
c = 0
for x in a:
    if not f(x):
        continue
    c += g(x)
print (c)
```

```
250
```

In [25]:
```
# 4th method
c = sum(g(x) for x in a if f(x))
print (c)
```

```
250
```

5. Combining more than one list

We have two lists of the same size a, b. In each element in these lists we have one observation. Each observation has 2 "values". For example:

Suppose we have this list of cities:

In [28]:
```
cities = ['Heraklion', 'Athens', 'Thessaloniki']
```

and their **respective** populations:

In [27]:
```
pop = [200_000, 4_000_000, 500_000]
```

Get the cities with a population of less than 1,000,000:

In [29]:
```
# 1st method
solution = []
for city, p in zip(cities, pop):
    if p < 1_000_000:
        solution.append(city)
print (solution)
```

```
['Heraklion', 'Thessaloniki']
```

In [30]:
```
# 2nd method
solution = []
for i, p in enumerate(pop):
    if p < 1_000_000:
        solution.append(cities[i])
print (solution)
```

```
['Heraklion', 'Thessaloniki']
```

In [31]:
```
# 3rd method
solution = [city for city, p in zip(cities, pop) if p<1_000_000]
print (solution)
```

```
['Heraklion', 'Thessaloniki']
```

In [32]:
```
# 4th method
solution = [cities[i] for i,p in enumerate(pop) if p<1_000_000]
print (solution)
```

```
['Heraklion', 'Thessaloniki']
```

In [33]:
```
# 5th method (a bit ugly..)
def f(x):
    return x[1]<1_000_000

def g(x):
    return x[0]

def h(x):
    return cities[x]

solution = list(map(h, map(g, filter(f, enumerate(pop)))))
print (solution)
```

```
['Heraklion', 'Thessaloniki']
```

6. Convert a list of lists, a flat list

Suppose that we have the following list:

In [35]:
```
a = [ [1,2], ["a", "b"], [True, False] ]
```

Make a list that has all the elements of `a` in one level (without sublists)

In [36]:
```
# 1st method
b = []
for x in a:
    b.extend(x)
print (b)
```

```
[1, 2, 'a', 'b', True, False]
```

In [37]:
```
# 2nd method
b = []
for x in a:
    for y in x:
        b.append(y)
print (b)
```

```
[1, 2, 'a', 'b', True, False]
```

In [38]:
```
# 3rd method (more pythonic!)
b = [y for x in a for y in x]
print (b)
```

```
[1, 2, 'a', 'b', True, False]
```

7. Un-zipping!

Suppose that we have the following list:

In [120…
```
a = [ [1, "a"], [2, "b"], [3, "c"]]
```

Create two lists k, l which will have the first and second elements of each list respectively.

In [39]:
```
# 1st method
k = []
l = []
for x in a:
    k.append(x[0])
    l.append(x[1])
print (k)
print (l)
```

```
[1, 'a', True]
[2, 'b', False]
```

In [40]:
```
# 2nd method
k = []
l = []
for x,y in a:
    k.append(x)
    l.append(y)
print (k)
print (l)
```

```
[1, 'a', True]
[2, 'b', False]
```

In [41]:
```
# 3rd method
k = [x[0] for x in a]
l = [x[1] for x in a]
print (k)
print (l)
```

```
[1, 'a', True]
[2, 'b', False]
```

8. Check if there is a specific item in a list.

Is `3` in the list: `a = [1,2,3,4,5,6,7,8,9,10]` ?

In [42]:
```
a = [1,2,3,4,5,6,7,8,9,10]
```

In [43]:
```
# 1st method
b = 3 in a
print (b)
```

```
True
```

In [44]:
```
# 2nd method
b = False
for x in a:
    if x==3:
        b = True
        break
print (b)
```

```
True
```

In [45]:
```
# 3rd method
b = True
for x in a:
    if x==3:
        break
else:
    b = False
print (b)
```

```
True
```