The `while` syntax

The `while` syntax is used as a more basic method for iteration compared to the `for` syntax. With the

```
while EXPRESSION:
    command_1
    command_2
    ...
    command_n
```

We declare that all the commands "below" `while` (`command_1` , ... `command_n`) will run until the `<EXPRESSION>` becomes `False` .

In [1]:
```
# Print all numbers from 0 to 9 (9 is included)
a=0
while a<10:
    print (a)
    a += 1
```

```
0
1
2
3
4
5
6
7
8
9
```

Print all **odd** numbers from 1 to 50

In [3]:
```
a=1

while a<50:
    if a%2 == 1:
        print (a)
    a+=1
```

```
1
3
5
7
9
11
13
15
17
19
21
23
25
27
29
31
33
35
37
39
41
```

```
43
45
47
49
```

print the multiplication table of 8

In [4]:
```
a = 1
while a<=10:
    print (a*8)
    a+=1
```

```
8
16
24
32
40
48
56
64
72
80
```

A little bit better:

In [3]:
```
a = 1
while a<=10:
    print (a, 'times 8 is', a*8)
    a+=1
```

```
1 times 8 is 8
2 times 8 is 16
3 times 8 is 24
4 times 8 is 32
5 times 8 is 40
6 times 8 is 48
7 times 8 is 56
8 times 8 is 64
9 times 8 is 72
10 times 8 is 80
```

Print the multiplication table of all numbers from 1 to 10

In [4]:
```
a = 1
while a<=10:
    b=1
    while b<=10:
        print (a, 'times ', b, ' is', a*b)
        b+=1
    a+=1
```

```
1 times  1  is 1
1 times  2  is 2
1 times  3  is 3
1 times  4  is 4
1 times  5  is 5
1 times  6  is 6
1 times  7  is 7
1 times  8  is 8
1 times  9  is 9
1 times  10  is 10
2 times  1  is 2
2 times  2  is 4
2 times  3  is 6
2 times  4  is 8
```

2 times 5 is 10
2 times 6 is 12
2 times 7 is 14
2 times 8 is 16
2 times 9 is 18
2 times 10 is 20
3 times 1 is 3
3 times 2 is 6
3 times 3 is 9
3 times 4 is 12
3 times 5 is 15
3 times 6 is 18
3 times 7 is 21
3 times 8 is 24
3 times 9 is 27
3 times 10 is 30
4 times 1 is 4
4 times 2 is 8
4 times 3 is 12
4 times 4 is 16
4 times 5 is 20
4 times 6 is 24
4 times 7 is 28
4 times 8 is 32
4 times 9 is 36
4 times 10 is 40
5 times 1 is 5
5 times 2 is 10
5 times 3 is 15
5 times 4 is 20
5 times 5 is 25
5 times 6 is 30
5 times 7 is 35
5 times 8 is 40
5 times 9 is 45
5 times 10 is 50
6 times 1 is 6
6 times 2 is 12
6 times 3 is 18
6 times 4 is 24
6 times 5 is 30
6 times 6 is 36
6 times 7 is 42
6 times 8 is 48
6 times 9 is 54
6 times 10 is 60
7 times 1 is 7
7 times 2 is 14
7 times 3 is 21
7 times 4 is 28
7 times 5 is 35
7 times 6 is 42
7 times 7 is 49
7 times 8 is 56
7 times 9 is 63
7 times 10 is 70
8 times 1 is 8
8 times 2 is 16
8 times 3 is 24
8 times 4 is 32
8 times 5 is 40
8 times 6 is 48
8 times 7 is 56
8 times 8 is 64

```
8 times  9  is 72
8 times  10  is 80
9 times  1  is 9
9 times  2  is 18
9 times  3  is 27
9 times  4  is 36
9 times  5  is 45
9 times  6  is 54
9 times  7  is 63
9 times  8  is 72
9 times  9  is 81
9 times  10  is 90
10 times  1  is 10
10 times  2  is 20
10 times  3  is 30
10 times  4  is 40
10 times  5  is 50
10 times  6  is 60
10 times  7  is 70
10 times  8  is 80
10 times  9  is 90
```

Find how many digits a number has:

In [5]:
```
a=51234123
# 1st method (fast and better)
len(str(a))
```

Out[5]: 8

In [8]:
```
# 2nd method
# Notice that the integer division with 10 removes the last digit from a nu
# 51234123 // 10 --> 5123412

a=51234123
c=0
remainder = a
while remainder != 0:
    remainder = remainder // 10
    c += 1
print (c)
```

8

How many times does a occurs in a string?

In [9]:
```
# 1st method (better / faster)
a = 'zabarakatranemia'
a.count('a')
```

Out[9]: 6

In [10]:
```
# 2nd method
index = 0
c = 0
while index < len(a):
    if a[index] == 'a':
        c += 1
    index += 1
print (c)
```

6

Invert a string.

In [94]:
```
# 1st method (better / faster)
a = 'zabarakatranemia'
a[::-1]
```

Out[94]:
```
'aimenartakarabaz'
```

In [11]:
```
# 2nd method
index = len(a)-1
anapodo = ''
while index >= 0:
    anapodo = anapodo + a[index]
    index -= 1
print (anapodo)
```

```
aimenartakarabaz
```

The sum of all numbers from 1 to 20

In [102…
```
s = 0
c = 0
while c < 20:
    c += 1
    s += c
print (s)
```

```
210
```

In [103…
```
s = 0
c = 1
while c <= 20:
    s += c
    c += 1
print (s)
```

```
210
```

In [98]:
```
sum(range(1,21))
```

Out[98]:
```
210
```

As with `for` we are allowed to use `break` and `continue`. With `break` we exit the `while` syntax completely, whereas with `continue` we move execution to the beginning of the while where the estimation of the logical expression takes place.

In [12]:
```
a=0
while a<10:
    a += 1
    if a== 5:
        continue

    print (a)
```

```
1
2
3
4
6
7
8
9
10
```

Note that 5 does not exist.

In [107…
```
a=0
while a<10:
    a += 1
    if a== 5:
        break

    print (a)
```

```
1
2
3
4
```

And here when a becomes 5 then it comes out completely from while.

Something that is rarely used but is especially useful is `else` after `while` . It enters the `else` syntax only if a `break` has **not** occurred in the while.

In [13]:
```
a=0
while a<10:
    a += 1
    if a== 5:
        break

    print (a)
else:
    print ('No break happened')
```

```
1
2
3
4
```

In [14]:
```
a=0
while a<10:
    a += 1
    #if a== 5:
    #    break

    print (a)
else:
    print ('No break happened')
```

```
1
```

```
2
3
4
5
6
7
8
9
10
No break happened
```

The `while` syntax is commonly used when we want to do an iteration but we do not know how many times this interation should be done. For example: let a ball fall from 1 meter. Each time it bounces it reaches 90% of its initial height. After 5 bounces to what height will it have reached? Here we know how many iteration we have to do, so we will use the `for` syntax:

In [112…
```
height=1
for i in range(5):
    height -= 0.1*height
print (height)
```

```
0.5904900000000001
```

Let us now look at another problem: After how many bounces will the ball reach a height of less than 0.5 meters? Now we do not know the number of repetitions (exactly this is what is required), so it is "convenient" to use the `while` syntax:

In [15]:
```
height = 1
bounces = 0
while height > 0.5:
    bounces += 1
    height -= 0.1*height
print (bounces)
```

```
7
```

Another example: The following function checks whether a number is prime or not:

In [17]:
```
def is_prime(n):
    for x in range(2, int(n**0.5)+1):
        if n%x==0:
            return False

    return True
```

If we start suming up all prime numbers starting from 1, to which priment number will this sum exceed 1,000,000?

In [18]:
```
s = 0
c = 1
while s < 1_000_000:
    if is_prime(c):
        s += c
    c += 1
print (c-1)
```

```
3943
```

Tuples

Tuples are data structures that are similar to lists. The difference is that in tuples we can not change a value. Or else tuples are immutable data types. To define a tuple, instead of brackets (`[1,2,3]`) we use parentheses (`(1,2,3)`)

In [19]:
```
a = (1,2,3)
type(a)
```

Out[19]:
```
tuple
```

In [20]:
```
a[2] = 7 #This is an error. We cannot modofy the valuea of a tuple
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-21aa96e85905> in <module>
----> 1 a[2] = 7 #This is an error. We cannot modofy the valuea of a tuple

TypeError: 'tuple' object does not support item assignment
```

In [21]:
```
b = [1,2,3]
b[2] = 7 # This is fine. A is a list
```

Although in tuples we can not add or remove an item, we can change items in lists or dictionaries that they may contain:

In [22]:
```
a = (1,[4,5],10)
a[1].append(6)
print (a)
```

```
(1, [4, 5, 6], 10)
```

Except from modifying their elements, we can use tuples just as we use lists. For example we can iterate on tuples and apply the functions `min` , `max` , `sort` , ...

In [23]:
```
for x in (1,2,3):
    print (x)
```

```
1
2
3
```

In [24]:
```
sum((5,6,7))
```

Out[24]:
```
18
```

Functions that return more than 1 value

In python a function can return more than one value:

In [26]:
```
def f():
    return 1,2
```

In [27]:
```
a,b = f()
print (a)
print (b)
```

```
1
2
```

If we store in a single variable the result of a function that returns more than one value then the type of the returned value is a tuple.

In [28]:
```
a = f()
```

In [29]:
```
print (a)
```

```
(1, 2)
```

In [30]:
```
type(a)
```

Out[30]:
```
tuple
```

Dictionaries

So far we have seen the following types of variables:

In [31]:
```
a=0 # integer
a=True # boolean
a="324234" # strings
a=5.6 # floats
a=[2,4,4] # lists
a=None # None
```

Dictionaries are a new type of variable. Dictionaries have data in the form of key -> value. Each key is unique. For example:

In [32]:
```
a = {"mitsos": 50, "anna": 40}
```

In [33]:
```
print(a)
```

```
{'mitsos': 50, 'anna': 40}
```

In [34]:
```
a['mitsos']
```

Out[34]:
```
50
```

In [35]:
```
a['anna']
```

Out[35]:
```
40
```

The `keys` method returns a list of all dictionary keys

In [36]:
```
a.keys()
```

Out[36]:
```
dict_keys(['mitsos', 'anna'])
```

The `values` method returns a list of all dictionary values

In [37]:
```
a.values()
```

Out[37]:
```
dict_values([50, 40])
```

We can insert a new pair of KEY, VALUE, as follows:

In [38]:
```
a["kitsos"] = 100
```

In [39]:
```
print (a)
```

```
{'mitsos': 50, 'anna': 40, 'kitsos': 100}
```

We can also remove a KEY, VALUE pair with the command del:

In [40]:
```
del a['kitsos']
print (a)
```

```
{'mitsos': 50, 'anna': 40}
```

The key can be number, string and boolean and tuple. The value can be anything.

In [41]:
```
a[123] = 0.1
a[3.14] = "hello"
a[False] = [1,2,3]
a[(4,7)] = 4
```

In [42]:
```
print (a)
```

```
{'mitsos': 50, 'anna': 40, 123: 0.1, 3.14: 'hello', False: [1, 2, 3], (4,
7): 4}
```

In [43]:
```
# Attention ! False == 0 !
a[0]
```

Out[43]:
```
[1, 2, 3]
```

The key can NOT be a list:

In [141…
```
a[[1,2,3]] = 0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-141-6cebb9942dfe> in <module>
----> 1 a[[1,2,3]] = 0

TypeError: unhashable type: 'list'
```

The key can NOT be a dictionary either:

In [142…
```
a[{}] = 0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-142-b372ccb1b9be> in <module>
----> 1 a[{}] = 0

TypeError: unhashable type: 'dict'
```

In python we can have dictionaries in lists and lists in dictionaries without any restrictions

In [44]:
```
d = {"a": {2:"a"}, 3: ["hello", False, []], 3.1: True}
print (d)
```

```
{'a': {2: 'a'}, 3: ['hello', False, []], 3.1: True}
```

We can compose lists and dictionaries from other lists and dictionaries:

In [27]:
```
[d, d, d["a"]]
```

Out[27]:
```
[{'a': {2: 'a'}, 3: ['hello', False, []], 3.1: True},
 {'a': {2: 'a'}, 3: ['hello', False, []], 3.1: True},
 {2: 'a'}]
```

In [28]:
```
{"a": d, "b": d[3]}
```

Out[28]:
```
{'a': {'a': {2: 'a'}, 3: ['hello', False, []], 3.1: True},
```

```
'b': ['hello', False, []]
```

There is also the empty dictionary

In [29]:
```
a = {}
```

`len` returns the number of records of a dictionary:

In [46]:
```
person = {"name": "alex", "age": 50, "occupation": "master"}
```

In [47]:
```
len(person)
```

Out[47]: 3

In [48]:
```
len({})
```

Out[48]: 0

We can check if a key exists in a dictionary

In [49]:
```
"name" in person # Very fast..
```

Out[49]: True

In [50]:
```
"alex" in person # Very fast..
```

Out[50]: False

We can check if a value exists in a dictionary:

In [51]:
```
"alex" in person.values() # Attention! this is slow!
```

Out[51]: True

We can iterate in all the elements of a dictionary:

In [52]:
```
for i in person:
    print (i)
```

```
name
age
occupation
```

In [53]:
```
for i in person:
    print ("key: {}  Value: {}".format(i, person[i]))
```

```
key: name  Value: alex
key: age  Value: 50
key: occupation  Value: master
```

We can convert a list (or tuple) into a dictionary with the dict function. However, the list must consist of sublists, where each sublist has 2 items. In these sublists the first item will become the key and the second the value:

In [146…
```
a = [("mitsos", 1), ('maria', 2), ('elenh', 4) ]
dict(a)
```

Out[146… {'mitsos': 1, 'maria': 2, 'elenh': 4}

The opposite can also be done with the `items` method:

In [54]:
```
a = {'mitsos': 1, 'maria': 2, 'elenh': 4}
print(list(a.items()))
```

```
[('mitsos', 1), ('maria', 2), ('elenh', 4)]
```

Accessing data in dictionary

We can use `[][]` more than once to access an item:

In [38]:
```
person = {"name": "alex", "age": 50, "occupation": "master", "exper": ["py
```

In [39]:
```
print (person)
```

```
{'name': 'alex', 'age': 50, 'occupation': 'master', 'exper': ['python', 'ka
rate']}
```

In [40]:
```
print (person['exper'][0])
```

```
python
```

In [41]:
```
print (person['exper'][1])
```

```
karate
```

In [42]:
```
print (person['exper'])
```

```
['python', 'karate']
```

In [43]:
```
a = ["a", "b", {"name": "mitsos", "surnmae": "sdfsdfsdf"}]
```

In [45]:
```
a[0]
```

Out[45]:
```
'a'
```

In [47]:
```
a[1]
```

Out[47]:
```
'b'
```

In [48]:
```
a[2]
```

Out[48]:
```
{'name': 'mitsos', 'surnmae': 'sdfsdfsdf'}
```

In [49]:
```
a[2]['name']
```

Out[49]:
```
'mitsos'
```

The function `a.get(b,c)` checks if `b` exists in the dictionary `a`. If it exists then it returns the value: `a[b]`, otherwise it returns `c`:

In [157…
```
a = {"a": 1, "b": 2, "c": 3}
```

In [158…
```
a.get("mitsos", 50)
```

Out[158…
```
50
```

In [159…
```
a.get("a", 50)
```

Out[159…
```
1
```

Iteration in a dictionary

Suppose that a is a list and b is a dictionary:

In [55]:
```
a = [1,2,3]
b = {"a":1, "b":2, "c":3}
```

We can iterate a list as follows:

In [51]:
```
for x in a:
    print (x)
```

```
1
2
3
```

The same can be done in a dictionary:

In [52]:
```
for x in b:
    print (x, b[x])
```

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

We can get the key-value pairs of the dictionary as a list by using the `items()` method:

In [53]:
```
list(b.items())
```

Out[53]:
```
[('a', 1), ('b', 2), ('c', 3)]
```

So as we have seen before, we can iterate a dictionary and assign the key-value pairs of each record of the dictionary into two variables:

In [56]:
```
for x,y in b.items():
    print (x,y)
```

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

Examples with dictionary

Count the number of occurences of each item in a list:

In [145…
```
a = [3,2,3,2,4,5,4,3,6,5,7,9,1,2,8,9,9]
d = {}

for x in a:
    if not x in d:
        d[x] = 0
    d[x] += 1

print (d)
```

```
{3: 3, 2: 3, 4: 2, 5: 2, 6: 1, 7: 1, 9: 3, 1: 1, 8: 1}
```

Find the value that has the largest key:

In [57]:
```
a = {1:3, 5:2, 3:1} # Maximum key is 5. The value of 5 is 2.

max_key = max(a.keys())
print(a[max_key])
```

```
2
```

Find the key that has the highest value

In [59]:
```
a = {1:3, 5:2, 3:1} #  Maximum value is 3 belonging to key 1
max( (v,k)  for k,v in a.items())[1]
```

Out[59]:
```
1
```

Let's break the above in smaller steps:

In [58]:
```
b = list(a.items()) # Convert the list
print (b)

c = [(v,k) for k,v in b] # Swat key/value pairs
print(c)

d = max(c) # Get the tuple that has the maximum value
print(d)

e = d[1] # Get the second element of the maximum tuple. This happens to be
print (e)
```

```
[(1, 3), (5, 2), (3, 1)]
[(3, 1), (2, 5), (1, 3)]
(3, 1)
1
```

Dictionary Comprehension

In a previous lecture we introduced list comprehensions

In [55]:
```
# List comprehension
[x for x in [1,2,3,4] if x>2]
```

Out[55]:
```
[3, 4]
```

As a reminder, we talked about how the above is equivalent to:

In [56]:
```
a = []
for x in [1,2,3,4]:
    if x>2:
        a.append(x)
print (a)
```

```
[3, 4]
```

The same can be done with dictionaries:

In [58]:
```
{ x:x*10 for x in range(1,10)}
```

Out[58]:
```
{1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60, 7: 70, 8: 80, 9: 90}
```

This is equivalent to:

In [59]:
```
a={}
for x in range(1,10):
    a[x] = x*10
print (a)
```

```
{1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60, 7: 70, 8: 80, 9: 90}
```

Another example:

In [60]:
```
{ x:'hello {}'.format(x*10) for x in range(1,10)}
```

Out[60]:
```
{1: 'hello 10',
 2: 'hello 20',
 3: 'hello 30',
 4: 'hello 40',
 5: 'hello 50',
 6: 'hello 60',
 7: 'hello 70',
 8: 'hello 80',
 9: 'hello 90'}
```

Sets

A `set` is a data structure that models a set. Each item in a set can **only** exist once:

In [61]:
```
set([1,2,3])
```

Out[61]:
```
{1, 2, 3}
```

In [62]:
```
set([1,2,3,2])
```

Out[62]:
```
{1, 2, 3}
```

In [63]:
```
a = set(['a','b', 'a'])
a
```

Out[63]:
```
{'a', 'b'}
```

In [64]:
```
'b' in a
```

Out[64]:
```
True
```

In [65]:
```
set("Hello World!")
```

Out[65]:
```
{' ', '!', 'H', 'W', 'd', 'e', 'l', 'o', 'r'}
```

The operation `&` between two sets returns the intersection of the sets:

In [66]:
```
a = set([1,2,3,4])
b = set([3,4,5,6])
a & b
```

Out[66]:
```
{3, 4}
```

The same can be done with the intersection function:

In [67]:
```
a.intersection(b)
```

Out[67]:
```
{3, 4}
```

The operation | between two sets returns the union of the sets:

In [68]:
```
a | b
```

Out[68]:
```
{1, 2, 3, 4, 5, 6}
```

The same can be done with the union function:

In [69]:
```
a.union(b)
```

Out[69]:
```
{1, 2, 3, 4, 5, 6}
```

The operation - between two sets α and β returns the elements of α that do not exist in β :

In [70]:
```
a - b
```

Out[70]:
```
{1, 2}
```

In [71]:
```
b - a
```

Out[71]:
```
{5, 6}
```

In [75]:
```
(a - b) & (b-a)
```

Out[75]:
```
set()
```

We can add an element to a set with the add function:

In [81]:
```
a = {1,2,3}
a.add(10)
print (a)
```

```
{10, 1, 2, 3}
```

Another way to add an element is to use the | operator :

In [82]:
```
a = {1,2,3}
a = a | {10}
print (a)
```

```
{10, 1, 2, 3}
```

We can not add a list to a set. This is because we can only add items that do NOT change:

In [79]:
```
a.add([7,8,9])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-79-1706898a2cfb> in <module>
----> 1 a.add([7,8,9])

TypeError: unhashable type: 'list'
```

Sets are an additional type of data:

In [73]:
```
type(set([1,2,3]))
```

Out[73]:
```
set
```

In [74]:
```
a = set([1,2,3])
type(a) is set
```

Out[74]:
```
True
```

set comprehension

Just like with lists and dictionaries, we can have comprehensions with sets:

In [83]:
```
{x%4 for x in range(10)}
```

Out[83]:
```
{0, 1, 2, 3}
```