Introduction to programming.

Lecturers : Giovanni Casini (giovanni.casini@uni.lu), Xavier Parent (xavier.parent@uni.lu) The slides and the handout have been obtained modifying the materials by Clément Guérin

What we have seen in the first lecture:

- Some data types of objects:
  - Integers (int)
  - real numbers (float)
  - boolean (bool)
  - strings (str, just mentioned)
- Some functions:
  - input
  - print
- if elif else
- while

What we are going to see in this lecture:

- Distinction between **functions** and **methods**
- more **datatypes**:
  - Integers (int)
  - real numbers (float)
  - boolean (bool)
  - strings (str)
  - tuples (tuple)
  - set (set)
  - dictionaries (dict) ⇐
  - Lists ⇐
- Use of the container range ⇐
- **Loops** ⇐

Dictionaries.

A dictionary or associative table is a very particular container. It is a collection of items "key:value" where key and value can be any kind of objects, where the statements are put between **curlybrackets** and separated from each other by **commas**.

$c = \{x : y, s : t, \ldots\}$

In [1]:

```
dico1={'Jean Paul':'jeanpaul@trucmuch.lu',\
'Fanny':'fanny@trucmuch.lu',\
'Robert':'robert@trucmuch.lu',\
'Stephanie': (6812424239),\
0:2}

#Type
print(type(dico1))
```

```
<class 'dict'>
```

- The function **len** still gives the cardinality.
- Indexes cannot be used, but we can use the key to recall the correspondent values.
- It is possible to associate new values to the keys.

In [2]:

```
dico1={'Jean Paul':'jeanpaul@trucmuch.lu',\
'Fanny':'fanny@trucmuch.lu',\
'Robert':'robert@trucmuch.lu',\
'Stephanie': (6812424239),\
0:2}
#It is still possible to get the length of a dictionnary.
len(dico1)
#Indexes cannot be used with dictionaries. Instead you ask for a key.
dico1['Jean Paul']
dico1[0]
```

Out[2]:

```
2
```

In [3]:

```
dico1={'Jean Paul':'jeanpaul@trucmuch.lu',\
'Fanny':'fanny@trucmuch.lu',\
'Robert':'robert@trucmuch.lu',\
'Stephanie': (6812424239),\
0:2}
#Changing a value in a dictionary.
dico1['Stephanie']='stephanie@trucmuch.lu'
print(dico1)
```

```
{'Jean Paul': 'jeanpaul@trucmuch.lu', 'Fanny': 'fanny@trucmuch.lu',
'Robert': 'robert@trucmuch.lu', 'Stephanie': 'stephanie@trucmuch.l
u', 0: 2}
```

In [4]:

```
#What is the effect when you have two items with the same key?
dico2={0:7,'x':'x@trucmuch.lu',0:3}
print(dico2)
```

```
{0: 3, 'x': 'x@trucmuch.lu'}
```

Here are some methods that you can use with dictionnaries. Let dic be a dictionary.

- $dic.$ **item**() returns a list of the items in the dictionary (see below for the notion of list).
- $dic.$ **keys**() only returns the keys of the dictionary.
- $dic.$ **values**() only returns the values of the dictionary.
- $dic.$ **copy**() returns a dictionary which is a copy of dic.
- $dic.$ **pop**(key) take out of the dictionary the item which has key as a key and returns the value associated to key.
- $dic.$ **popitem**() takes out of the dictionnary the the last inserted item and returns such an item.
- $dic.$ **update**$(newdic)$ updates dic with the values of another dictionary $newdic$, it adds new items if the keys of $newdic$ are not in dic.

The expression

- $dic[key] = value$

assigns a new value to the key key if it already exists in the dictionary.

Otherwise we add the pair $(key : value)$ to the dictionary.

In [15]:

```
dico1={'Jean Paul':'jeanpaul@trucmuch.lu',\
'Fanny':'fanny@trucmuch.lu',\
'Robert':'robert@trucmuch.lu',\
'Stephanie': (6812424239),\
0:2}
dico1['Stephanie']='stephanie@trucmuch.lu' #'Stephanie' was already in the dicti
onary,
#so there is a re-assignment of value.
dico1['Carl']='carl@trucmuch.lu'#'Carl' was not in the dictionary,
#so there is the addition of a new pair to the dictionary.
print(dico1)
```

```
{'Jean Paul': 'jeanpaul@trucmuch.lu', 'Fanny': 'fanny@trucmuch.lu',
'Robert': 'robert@trucmuch.lu', 'Stephanie': 'stephanie@trucmuch.l
u', 0: 2, 'Carl': 'carl@trucmuch.lu'}
```

In [17]:

```
dico1={'Jean Paul':'jeanpaul@trucmuch.lu',\
'Fanny':'fanny@trucmuch.lu',\
'Robert':'robert@trucmuch.lu',\
'Stephanie': (6812424239),\
0:2}
dico2={0:7,'x':'x@trucmuch.lu',0:3}
print(dico1,dico2)
dico1.update(dico2)
print(dico1)
```

```
{'Jean Paul': 'jeanpaul@trucmuch.lu', 'Fanny': 'fanny@trucmuch.lu',
'Robert': 'robert@trucmuch.lu', 'Stephanie': 6812424239, 0: 2} {0:
3, 'x': 'x@trucmuch.lu'}
{'Jean Paul': 'jeanpaul@trucmuch.lu', 'Fanny': 'fanny@trucmuch.lu',
'Robert': 'robert@trucmuch.lu', 'Stephanie': 6812424239, 0: 3, 'x':
'x@trucmuch.lu'}
```

Lists

A list is an ordered container of possibly different types of objects. It is defined between **brackets** and objects are separated by **comas**.

$L = [x, y, \ldots]$

In [40]:

```
#Definition
L=[2,3,4]
#Type
type(L)
```

Out[40]:

```
list
```

In [41]:

```
#Definition by comprehension
L=[x**2 for x in range(0,9)]
L
```

Out[41]:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64]
```

There are some ways of using indices that are very convenient to access to elements of a **list** (they work for any ordered container such as strings and tuples).

- $L[i]$ returns the i-th element of the list L.
- $L[i:j]$ returns the elements from the i-th (included) to the j-th (excluded). The result has the same type as L.
- $L[i:]$ is the same as $L[i:\mathbf{len}(L)]$.
- $L[:j]$ is the same as $L[0:j]$.
- $L[i::step]$ is the list of elements from the i-th that you obtain by step of $step$.

We can also concatenate lists using '+'.

In [6]:

```
#Changing the i-th element
L=[1,2,3,4,5,6]
L[3]=12
print(L)
```

```
[1, 2, 3, 12, 5, 6]
```

In [7]:

```
#Accessing to an element of the list using the index
L=[1,2,3,4,5,6]
L[2::3]
```

Out[7]:

```
[3, 6]
```

In [8]:

```
#Concatenation
L=[1,2,3,4,5,6]
M=['horse','dog']
N=L+M
print(N)
```

```
[1, 2, 3, 4, 5, 6, 'horse', 'dog']
```

Here we list some built-in methods to deal with lists.

- $L.$ **count**(obj) returns the number of occurences of the object obj.
- $L.$ **index**$(value)$ returns the first index i for which $L[i] = value$.
- $L.$ **insert**(i, obj) inserts the object obj at the i-th place, shifting the rest of the list to the right.
- $L.$ **remove**$(value)$ removes from L the first occurence of $value$.
- $L.$ **pop**$(index)$ returns the corresponding value and removes it from L.
- $L.$ **reverse**$()$ writes L backward (it changes L).
- $L.$ **sort**(L) reorders L according to the lexicographic order of the eements. The elements should all be of the same type.

In [10]:

```
L=['1','5','2','horse','3']
L.sort()
print(L)
```

```
['1', '2', '3', '5', 'horse']
```

Range

Ranges are very specific types of containers.

You typically create a range by calling **range**$(start, stop, step)$.

This will create a range of integer numbers from $start$ (included) to $stop$ (excluded) by steps of length $step$.

You can also call **range**$(start, stop)$ and $step = 1$ by defaut and you can also call **range**$(stop)$ with $start = 0$ and $step = 1$ by defaut.

In [46]:

```
#Wait, is this really working?
range(1,100,2)
```

Out[46]:

```
range(1, 100, 2)
```

It is not "really" an object **per se**, one should rather think about it as a potential list of integers. You can still ask if $something$ is in a range object.

In [19]:

```
#Is a value in the range?
r=range(1,100,2)
c=12
print(12 in r)
```

```
False
```

Differences and links among data structures

- A dictionnary is a very convenient way to store/update/erase some information about specific keys. However, it is a rather complicated object compared to the other data structures and should therefore be used wisely.
- A string of characters is a very specific object. It is the best way to communicate with the operator running the code. Using the **format** method.
- Strings and tuples are non-mutable objects. There is no built-in method to change their values.

- Sets, dictionnaries and lists are mutable objects.

There are plenty of built-in methods to change them.

Be careful, as we have seen before the "=" sign **is a re-assignment function in case of mutable objects**.

To copy a complex object we need to use the **copy** method instead.

- The counterpart of the mutability is a slightly slower access to the data.

In [13]:

```
# Effects of type functions.
L=[1,2,1,2,1,3,4,2,3]
c=list(set(L))
print(c)
```

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

"for" loops

When you have to do a repetitive task, it is very convenient to use a **for** loop. The standard statement is as follows.

for *variable* **in** *something*:

and then line break and your instructions. Like any ':' statement you will need to indent your instructions.

variable is any (non-protected) name for your variable and *something* is a container.

In [50]:

```
#First example of loop
for x in range(0,20):
    print(x,end=" ")
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

In [51]:

```
#Comparison with a while loop
x=0
while x<20:
    print(x,end=" ")
    x+=1
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

In [52]:

```
# Use of loops for different containers

for x in {0,1,2}:#Set
    print(x,end=' ')
print('')
for x in 'Introductiontoprogramming':#Strings
     print(x,end=' ')
print('')
for x in list('Introductiontoprogramming'):#Lists
     print(x,end=' ')
```

```
0 1 2
I n t r o d u c t i o n t o p r o g r a m m i n g
I n t r o d u c t i o n t o p r o g r a m m i n g
```

When going through a list L using a **for** loop you may want to have both the value and its index.

In [18]:

```
# First way to do it.
L=['1','5','2','horse','3']
for index in range(0,len(L)):
    print("L[{}]={}".format(index,L[index]))
```

```
L[0]=1
L[1]=5
L[2]=2
L[3]=horse
L[4]=3
```

You can also use the following notation **for** $index, variable$ **in** L:. In this case you will have $L[index] = variable$ during the execution of the loop.

In [54]:

```
for  index,variable in enumerate(L): # Use the enumerate function to access the
 tuples (i,L[i])
    print("L[{}]={}".format(index,variable))
```

```
L[0]=1
L[1]=2
L[2]=1
L[3]=2
L[4]=1
L[5]=3
L[6]=4
L[7]=2
L[8]=3
```

You can go out of a **for** loop. You do it using **break**.

In [55]:

```
for x in range(0,9):
    if x>6:
        break
    print(x,end=" ")
```

```
0 1 2 3 4 5 6
```

In [56]:

```
for y in range(0,9):
    for x in range(0,9):
        if x>6:
            break # You only break out of the loop you are in.
        print(10*y+x,end=" ")
```

```
0 1 2 3 4 5 6 10 11 12 13 14 15 16 20 21 22 23 24 25 26 30 31 32 33
34 35 36 40 41 42 43 44 45 46 50 51 52 53 54 55 56 60 61 62 63 64 65
66 70 71 72 73 74 75 76 80 81 82 83 84 85 86
```