# 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**
  - introduce more **datatypes**:
    - Integers (int)
    - real numbers (float)
    - boolean (bool)
    - strings (str) ⇐
    - tuples (tuple) ⇐
    - set (set) ⇐
    - dictionaries (dict) ⇐
    - Lists ⇐
    - Range ⇐
- Introduction to **datastructures**
- **Loops**

# Preliminary remark : Function vs. Method

# Functions :

A function is a block of code to preform a specific task.

All functions have a numbere $n$ of arguments ($n \geq 0$).

On exit, a function can or can not return one or more values.

We have already seen some functions like input and print.

# Methods :

Methods in python are very similar to functions except for some major differences.

- A method is called on an object (or a class, when we will introduce them)
- The method is used for the object for which it is called.
- The method can alter the object's state. Functions usually do not do it.

In [13]:

```
st='ertl'
c=len(st)# Function.
# len is called, and st is the argument of the function len.
print(c)
st.index('r')  # Method.
# index is called associated to the string st, and it is asked to give the posit
ion of 'r' inside the string st
```

```
4
```

Out[13]:

```
1
```

Later we will learn to define new functions and new methods.

Remember that functions are defined as independant objects whereas a method is always defined within a type of objects, and does not "exist" out of it.

Functions and methods may return a value (like **len**) or do something (like **print**).

They can also modify some objects.

Neither functions or methods need to be "functions" in the mathematical sense, i.e. objects that take some parameters and return a value.

# Special Types: Containers

We have considered **integers** and **realnumbers**, and introduced **strings** of symbols. Now we consider an advanced class of object types: the **containers**.

A container is simply an object that contains other objects. The object types that are considered containers are:

- strings (str) ⇐
- tuples (tpl) ⇐
- sets (set) ⇐
- dictionaries (dict) ⇐
- Lists ⇐
- Range ⇐

We are going to review some features of the most used containers in Python.

# Strings of characters

A string of characters is an ordered sequence of characters.

$s =' xyz \ldots'$

The $i$-th element of the string $c$ can be accessed (provided that the index is not out of range) by writing $c[i]$.

**Remember**:

- The indexing of a string starts from the position 0.

That is, the first element of a string is in position 0.

In [6]:

```
c='Introduction'
print(c[1],c[2],c[3],c[4],c[5])# The indexation starts at 0.
print(c[-1]) # We can go backward by using negative indices. They do not start f
rom 0, but from -1.
```

```
n t r o d
n
```

In [8]:

```
c='Example'
print(' ',c[0],c[1],c[2],c[3],c[4],c[5],c[6])
print(' ',0,1,2,3,4,5,6)
print('-',7,6,5,4,3,2,1)
```

```
  E x a m p l e
  0 1 2 3 4 5 6
- 7 6 5 4 3 2 1
```

In [7]:

```
len(c) #The len function gives the length (cardinal) of the container.
```

Out[7]:

```
12
```

In [16]:

```
c='Example'
print(len(c)) #The len function gives the length (cardinal) of the container.
print(c[0],c[1],c[2],c[3],c[4],c[5],c[6])
print(c[-len(c)],c[-len(c)+1],c[-len(c)+2])
```

```
7
E x a m p l e
E x a
```

In [10]:

```
c='Example'
c+=' of union of strings' #The + command for strings means
                         #concatenation.
print(c)
```

Out[10]:

```
'Example of union of strings'
```

You may have noticed that to define strings I sometimes use single quote ' or double quote ".

There is no difference between both in terms of formal interpretation.

Convention: one should only use double quotes when you want to directly print the sentence as an output in order to speak to the operator (in the **input** or **print** functions) and always use single quotes when your are simply defining a genuine string of characters not to be printed as an output.

If you use triple quotes, the output will respect the line breaks in the code.

In [18]:

```
#one, two, three single quotes or double quotes.
c='a string'
print(c)
c="a string" # single and double quotes are the same.
print(c)

c='''a string very very very veryvery very very very
very very very very very very very very
very very very very very very very very long ''' #Triple allows you to directly
 print line break.
print(c)
c="""a string very very very veryvery very very very
very very very very very very very very
very very very very very very very very long """
print(c)
```

```
a string
a string
a string very very very veryvery very very very
very very very very very very very very
very very very very very very very very long
a string very very very veryvery very very very
very very very very very very very very
very very very very very very very very long
```

We can check whether some simbol or sub-string occurs in a string.

In [19]:

```
#Contains
c='a string'
print('s' in c)
print('z' in c)
print('stri' in c)
```

```
True
False
True
```

You can select a sub-string of a string specifying an interval [n:m] of indexes.

In [21]:

```
#You can access to a substring out of a string.
c='''a string very very very veryvery very very very
very very very very very very very very
very very very very very very very very long '''
print(c)
print(c[2:10])
```

```
a string very very very veryvery very very very
very very very very very very very very
very very very very very very very very long
string v
```

In [22]:

```
#No assignment is possible.
c[2]='s'
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-22-cf50fef9b208> in <module>
      1 #No assignment is possible.
----> 2 c[2]='s'

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

We add some methods concerning strings that you may find useful (call $\mathbf{help}(\mathbf{str})$ or read the Python documentation for more information).

Let $c$ be a string.

- $c * n$, where $n$ is an integer, returns a string with n concatened copies of $c$. It is exactly the same as $\underbrace{c + \cdots + c}_{n\text{times}}$.
- $c.\mathbf{find}('car')$ returns the first index $i$ (if it exists) where the substring $c[i] =' car'$ starts.
- $c.\mathbf{count}(st)$ count the number of (non-overlapping) occurences of the string $st$ in $c$.
- $c.\mathbf{replace}(st1, st2)$ returns a string which is a copy of $c$ where every occurence of the string st1 has been replaced by st2.

In [23]:

```
c=3*'121'
print(c)
print(c.count('12'))
```

```
121121121
3
```

In [81]:

```
c=3*'121'
c=c.replace('11','ab')
print(c)
```

```
12ab2ab21
```

We can compare strings by lexicographic order.

In [46]:

```
c='Alfred'
d='Ann'
e='Alfred2'
print(c<d)
print(d<c)
print(c<e)
```

```
True
False
True
```

We can add values to a string using the **format** method

In [17]:

```
#The "format" method for strings.

c="Les entiers {1} et {0} sont pairs. {1} "

print(c.format(2,4)) # Insert values in a string, the place is given by the numb
er into braces.
c="Les entiers {} et {} sont pairs. {} "
print(c.format(0,2,'truc'))# Insert values in a string.
```

```
Les entiers 4 et 2 sont pairs. 4
Les entiers 0 et 2 sont pairs. truc
```

# Tuples

A tuples is a finite sequence of objects of possibly different types.

You can define a tuple as a sequence of objects between **parentheses** and separated by **commas**.

This data structure is rigid: you cannot modify a tuple.

However, you can:

- consider the concatenation of tuples.
- access the length of a tuple by applying the **len** function.
- access the elements with an index.
- compare two tuple using the lexicographic order

The comparison of tuples is trickier than for strings, since all the compared elements of the tuples must be of the same type.

Mathematically it is the equivalent of the usual tuple.

In [27]:

```
#Definition of a tuple.
x=(0,1,2,'obj',(1,2))
#Type
type(x)
```

Out[27]:

```
tuple
```

In [32]:

```
x=(0,1,2,'obj',(1,2))
# We cannot change the tuple assigning new values
x[1]='newvalue'
```

```
---------------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-32-ecdd8800034d> in <module>
      1 x=(0,1,2,'obj',(1,2))
      2 # We cannot change the tuple assigning new values
----> 3 x[1]='newvalue'

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

In [30]:

```
x=(0,1,2,'obj',(1,2))
#Concatenation
x=x+(2,3,4)
x
```

Out[30]:

```
(0, 1, 2, 'obj', (1, 2), 2, 3, 4)
```

In [28]:

```
x=(0,1,2,'obj',(1,2))
#Computation of the length
len(x)
```

Out[28]:

```
5
```

In [29]:

```
x=(0,1,2,'obj',(1,2))
#Accessing index
x[3]
```

Out[29]:

```
'obj'
```

In [39]:

```
(0,1,3)<(1,1,0) #Comparability
```

Out[39]:

```
True
```

In [40]:

```
('obj',1,3)<('1',1,0) #Comparability
```

Out[40]:

```
False
```

In [41]:

```
('obj',1,3)<(1,1,0)
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-41-bf9544542433> in <module>
----> 1 ('obj',1,3)<(1,1,0)

TypeError: '<' not supported between instances of 'str' and 'int'
```

There are not much specific methods associated to tuples.

- $x.\textbf{count}(obj)$ returns the number of occurences of the object $obj$ in $x$.
- $x.\textbf{index}(obj)$ returns the first index of occurence of the object $obj$ in $x$.

In [42]:

```
x=(0,1,2,'obj',(1,2))
print(x.count(1))
print(x.index('obj'))
```

```
1
3
```

# Sets

A set is a finite collection of objects between **curly brackets** and separated by **commas**.

$A = \{x, y, z, \dots\}$

In [25]:

```
#Definition of a set.
A={1,2,4,5,6}
#type.
type(A)
```

Out[25]:

```
set
```

It is equivalent to a mathematical set: repetitions and orders do not count.

Consequently also indexing does not work.

Set are comparable by the inclusion relation.

The function **len** gives the cardinality of the set.

In [43]:

```
# Repetition and order do not count.
A={1,2,3,4,5}
B={1,3,1,2,2,5,5,4,4,4,3}
A==B
```

Out[43]:

```
True
```

In [47]:

```
# Index
A={1,2,3,4,5}
print(A[1])
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-47-9045106314ed> in <module>
      1 # Index
      2 A={1,2,3,4,5}
----> 3 print(A[1])

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

In [28]:

```
#from strings to set.
c='a set'
set(c)
```

Out[28]:

```
{' ', 'a', 'e', 's', 't'}
```

In [48]:

```
# The length function computes the cardinality.
A={1,2,3,4,5}
len(A)
```

Out[48]:

```
5
```

Some methods that apply to sets.

The usual set computations hold.

- $A.\textbf{intersection}(B)$ returns the intersection of $A$ and $B$.
- $A.\textbf{union}(B)$ returns the union of $A$ and $B$.
- $A.\textbf{difference}(B)$ returns $A \setminus B$.
- $A.\textbf{copy}()$ returns a copy of $A$.

You may also directly change a set $A$:

- $A.\textbf{add}(a)$ replaces $A$ by $A \cup \{a\}$ (does nothing if $a$ is in $A$).
- $A.\textbf{discard}(a)$ replaces $A$ by $A$ without the object $a$ (does nothing if $a$ is not in $A$).
- $A.\textbf{difference\_update}(B)$ replaces $A$ by $A \cap B^c$.

In [30]:

```
A={1,2,4}
B=A.copy()
print(A,B)
B.discard(2)
print(A,B)
```

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

One useful way to define a set in mathematics is by comprehension i.e. $B := \{f(x) \mid x \in A\}$. You can do the same thing in Python. The syntax is the following :

$$B = \{\textbf{function}(x) \textbf{ for } x \textbf{ in } A\}$$

In [50]:

```
#You may also define a set by comprehension
A={1,2,3}
B={(x+1) for x in A}
B
```

Out[50]:

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

# 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 **curly brackets** and separated from each other by **commas**.

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

In [52]:

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

Out[52]:

```
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 [54]:

```
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[54]:

```
2
```

In [55]:

```
dico1={'Jean Paul':'jeanpaul@trucmuch.lu',\
             'Fanny':'fanny@trucmuch.lu',\
             'Robert':'robert@trucmuch.lu',\
       'Stephanie': (6812424239),\
       0:2}
#Changing a value in a dictionnary.
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 [36]:

```
#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 dictionnary.

- $dic$. **items**() returns the items of the dictionnary.
- $dic$. **keys**() only returns the keys of the dictionnary.
- $dic$. **values**() only returns the values of the dictionnary.
- $dic$. **copy**() returns a dictionnary which is a copy of $dic$.
- $dic$. **pop**($key$) takes out of the dictionnary 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 $newdic$, it adds new items if the keys of $newdic$ are not in $dic$.

In [78]:

```
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'}
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:
7, 'x': 'x@trucmuch.lu'}
{'Jean Paul': 'jeanpaul@trucmuch.lu', 'Fanny': 'fanny@trucmuch.lu',
'Robert': 'robert@trucmuch.lu', 'Stephanie': 6812424239, 0: 7, 'x':
'x@trucmuch.lu'}
```

# Lists

A list is an ordered container of possibly different types of objects.

It is a sequence of objects between **square brackets** and objects are separated by **commas**.

As ofr sets, we can define the content of lists by comprehension.

Most importantly, we can re-assign some value in a list. This is the main difference w.r.t. tuples.

In [40]:

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

Out[40]:

```
list
```

In [67]:

```
#Definition in comprehension
C={1,2,3}
L=[x**2 for x in C]# ''**'' is the power operator
L
```

Out[67]:

```
[1, 4, 9]
```

There are some ways 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 $L[i:\mathbf{len}(L)+1]$.
- $L[:j]$ is the same as $L[0:j]$.
- $L[i::n]$ is the list of elements from the $i$-th that you obtain by $n$ steps.

We can also concatenate lists.

In [71]:

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

Out[71]:

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

In [70]:

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

Out[70]:

```
[3, 6]
```

In [75]:

```
#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**$()$ reorders $L$ according to the lexicographic order of the lements. The elements should all be of the same type.

In [77]:

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

Out[77]:

```
['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**$(stop)$ and $start, step = 0, 1$ 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.

## Differences and links between 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.

In [47]:

```
#You can of course change a variable of type 'str' or 'tuple'.
x=(1,2,3)
x=x+x
print(x)
```

```
(1, 2, 3, 1, 2, 3)
```

- 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**. The good way to deal with this problem is to use the **copy** method instead.

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

In [83]:

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

Out[83]:

```
[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
```

Beware that **range** does not return a convenient type as a value.

Anyway you can still convert it as a list. It appears that if the value where a list the call above would be sligthly slower.

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 [53]:

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

```
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**.

The **break** statement can be used in both **while** and **for** loops.

If you are using nested loops, the break statement stops the execution of the innermost loop and start executing the next line of code after the block.

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
```