INFO-H-600 - Computing foundations of data sciences

Session 3
Introduction to Python
Simple data structures

Université libre de Bruxelles
École polytechnique de Bruxelles

2019-2020

Plan

We will see different types of data structures :

1. Immutable - Sequence :
   - Strings
   - Tupples

2. Mutable - Sequence :
   - Lists

3. Mutable - unordered (next week) :
   - Sets
   - Dictionaries

Indices and slicing

Sequences elements can be accessed via square brackets

```
>>> mot = "HelloWorld"
>>> mot[0] + ' ' + mot[5]
'H W'
```

A negative index can be used to acces the elements starting from the end

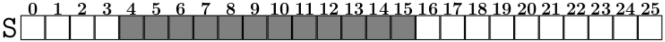
```
>>> mot = "HelloWorld"
>>> mot[-1] + ' ' + mot[-3]
'd r'
```

The positive (negative) indices are in range 0 to n-1 (-1 to -n)

```
           0       1     2       3
>>> s = ('Hello', ' ', 'World', '!')
          -4      -3    -2      -1
```

Indices and slicing

The slicing $s[a, b]$ allows to acces the elements of a sequence s which indices go from a included to b **not included**



S[4 : 16]

Examples :

```
>>> date = "18/06/2017"
>>> month = date[3:5]
>>> print(month)
06
>>> date[-5:-1]
'/201'
```

Indices and slicing

If the beginning or the end is not precised, the extremity is used

```
>>> date = "18/06/2017"
>>> date[:5]
'18/06'
```

Don’t hesitate to try by yourself

```
>>> date[6:2]
???
>>> date[-1: -5]
???
>>> date[3:50]
???
```

Tuples

In Python, a tuple is a sequence of values formed by separating these values with commas. The usage of paranthesis is recomanded.

```
>>> point = (1,2)
>>> print(point)
(1, 2)
```

```
>>> course = ('INFO','H',600)
>>> print(course)
('INFO','H',600)
```

The values are indexed

```
>>> print(point[0])
1
```

We can assign the values of a tuple to some other tuple containing the same number of values.

```
>>> (x,y) = point        # extraction
>>> print(x)
1
>>> print(y)
2
```

Tuples

A function can take a tuple as a parameter.

```
>>> def distance_origine(point):
      (x,y) = point
      return math.sqrt(x**2 + y**2)

>>> distance_origine((0,1))
1.0
```

Be careful :

```
>>> distance_origine(0,1)
...
TypeError: distance_origine() takes 1 positional argument but 2 were
```

A function can also return a tuple, which allows to return several values

```
>>> def divise_modulo(a, b):
      return (a // b, a % b)

>>> (quotient, reste) = divise_modulo(5,2)
>>> print(quotient, reste)
2 1
```

Strings : some useful functions

```
>>> s = "  \n  Foo\nBar spam\n"
>>> s
'  \n  Foo\nBar spam\n'

>>> s.upper()            # Capitalize
'  \n  FOO\nBAR SPAM\n'
>>> s.lower()
'  \n  foo\nbar spam\n'

>>> s.strip()            # Clean the beginning and the end
'Foo\nBar spam'

>>> s.replace("\n", "-") # Replace
'  -  Foo-Bar spam-'

>>> s.replace("\n", "")  # Suppress
'    FooBar spam'

>>> s.split()            # cut at whitespaces
['Foo', 'Bar', 'spam']

>>> s.split("\n")
['  ', '  Foo', 'Bar spam', '']
```

Strings : some useful functions

Careful, *strings* are immutables. These functions built modified copies.

```
>>> a = 'TesT'
>>> a.upper()
TEST
>>> a
TesT
```

Lists

In Python, a list is a sequence of elements that can be of different types.

Square brackets can be used to initialise empty lists

```
>>> li1 = []       # empty list []
>>> type(li1)
<type 'list'>
>>> li1
[]
>>> li2 = [1,2,3,4]
>>> li2
[1, 2, 3, 4]
>>> li2[2]
3
>>> li2[-1]
4
>>> li3 = ["SPAM", True, ('eggs', 42)] # different types of elements
>>> print(li3[2][0][3])
s
```

Operations on lists

Lists and strings being sequences, some operations are similar :

```
>>> li1 = [1,2,3,4]
>>> li2 = [5,6,7,8]
>>> li1 + li2
[1, 2, 3, 4, 5, 6, 7, 8]
>>> li1[1:] + li2[:-1]
[2, 3, 4, 5, 6, 7]
>>> len(li1)
4
>>> 5 in li1                # very useful
False
>>> li2 = li2 * 3
>>> li2
[5, 6, 7, 8, 5, 6, 7, 8, 5, 6, 7, 8]
>>> li2.index(7)
2
>>> li2.count(8)
3
```

Lists are mutable sequences

In opposition to strings and tuples lists are mutable, it means that they can be modified.

```
>>> my_list = [1,7]
>>> my_list[1] = 2
>>> my_list
[1, 2]
```

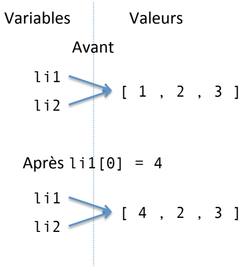
```
>>> message = "Welcome"
>>> message[0] = 'B'
TypeError: 'str' object does not support item assignment
```

Lists elements are pointers

```
>>> li1 = [1,2,3]
>>> li2 = li1
>>> li1[0] = 4
>>> print(li1)
[4, 2, 3]
>>> print(li2)
[4, 2, 3]
>>> li1 == li2
True
```

```
>>> li1 = [1,2,3]
>>> li2 = li1[:]
>>> li1[0] = 4
>>> print(li1)
[4, 2, 3]
>>> print(li2)
[1, 2, 3]
>>> li1 == li2
False
```

`li2 = li1`



Variables Valeurs

Avant

li1

li2

[1 , 2 , 3]

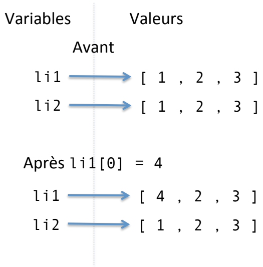
Après `li1[0] = 4`

li1

li2

[4 , 2 , 3]

`li2 = li1[:]`



Variables Valeurs

Avant

li1 [1 , 2 , 3]

li2 [1 , 2 , 3]

Après `li1[0] = 4`

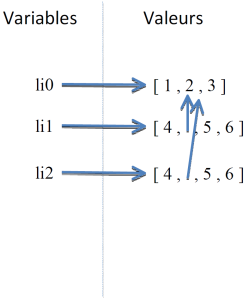
li1 [4 , 2 , 3]

li2 [1 , 2 , 3]

Lists elements are pointers

Copy is superficial

```
>>> li0 = [1, 2, 3]
>>> li1 = [4, li0, 5, 6]
>>> li2 = li1[:]          # make a copy
>>> print(li2)
[4, [1, 2, 3], 5, 6]
>>> li1[1][0] = 8         # change li1
>>> print(li2)
[4, [8, 2, 3], 5, 6]      # li2 changed too
```



Variables

Valeurs

li0

[1 , 2 , 3]

li1

[4 , , 5 , 6]

li2

[4 , , 5 , 6]

New operations on lists

Due to theire mutability, lists have new operations :

```
>>> li1 = [5,2,6,7,1]
>>> li1.append(9)             #add an item to the end
>>> li1
[5, 2, 6, 7, 1, 9]
>>> li1.sort()                #sort the items by ascending order
>>> li1
[1, 2, 5, 6, 7, 9]
>>> li1.insert(2,'eggs')      #insert an item at a given position
>>> print(li1)
[1, 2, 'eggs', 5, 6, 7, 9]
>>> del li1[4]                #remove an item
>>> li1
[1, 2, 'eggs', 5, 7, 9]
>>> list("SPAM")              #convert to list
['S', 'P', 'A', 'M']
>>> li1.extend([3, 4, 5])     #extend with another list
>>> li1
[1, 2, 'eggs', 5, 7, 9, 3, 4, 5]
>>> li1.append([6, 7, 8])     # append another list
>>> li1
[1, 2, 'eggs', 5, 7, 9, 3, 4, 5, [6, 7, 8]]
```

Sequences are iterators

```
# my_iterator : 1, 2, 3
>>> for x in my_iterator:
        print(x)

1
2
3
```

Any sequence can be used as iterator!

```
def sum_list(li):
    total = 0
    for item in li:
        total += item
    return total

ls =  [ 1, 2, 3 ]

print(sum_list(ls)) # -> 6
```

Range function builds an iterator

```
>>> for i in range(3):
        print(i)

0
1
2
```

```
>>> ls = [1, 2, 3]
>>> for i in range(len(ls)):
        print("pos:", i, ", val:", ls[i])

pos: 0 -> val: 1
pos: 1 -> val: 2
pos: 2 -> val: 3
```

Look at the documentation for other operations

Sequences are iterators

```
>>> mat = [ [ 1 , 2 ,  3 ,  4  ],
            [ 5 , 6 ,  7 ,  8  ],
            [ 9 , 10 , 11 , 12 ] ]
>>> len(mat)
3
>>> mat[1]
[5, 6, 7, 8]
>>> mat[1][0]
5
>>> for line in mat:
        print(sum_list(line))

10
26
42
```

```
def sum_list_of_lists(li):
    total = 0
    for line in li:
        for item in line:     # or total += sum_list(line)
            total += item     #
    return total

print(sum_list_of_lists(mat)) # -> 78
```

keyword in

The *in* keyword can be used to check whether an element is in a datastructure :

```
>>> l = [1, 2, 5]
>>> 2 in l
True
>>> "ok" in l
False
```