Cours_Python_SRI_1_base

September 9, 2019

1 Programmation avancée Python & C++

. ### Philippe Muller & Christine Regis

- Présentation des langages orientés objet Python/C++ et leur connexion
- Introduction aux concepts avancés dans ces langages
- Présentation de quelques outils et bibliothèques scientifiques

```
In [ ]:
```

1.1 Organisation

Deux parties avec : - Python - 5x2h de cours/TD - 4x2h de TP - C++ (Christine Régis) - 5x2h de cours/TD - 4x2h de TP

Evaluation : - un examen sur feuille pour chaque partie - un TP à rendre pour chaque partie

Cours Python en ligne sur ma page web http://www.irit.fr/~Philippe.Muller (statique+interactif) et/ou sur la page Moodle du cours

Pour utiliser la version interactive, il vous faut l'interpréteur ipython, avec le notebook (jupyter)

Vous pouvez installer tout d'un coup avec la distribution Anaconda (version python 3.xx) :
https://www.anaconda.com/distribution/

1.2 Plan du cours

- Motivations / Présentation de python
- Syntaxe de base
- Programmation objet
- Concepts avancés: itérateurs, programmation fonctionnelle, décorateurs
- Développement en python: IDE, test, debugging
- Présentation des outils scientifiques, notamment pour SRI
  - traitement du signal/audio/image
  - apprentissage automatique
  - middleware pour programmation de robot

1.3 Motivations

Pourquoi python ? Productivité + performances

- un langage devenu très populaire dans la communauté scientifique (et au-delà)
- langage de haut niveau: permet un développement rapide - syntaxe simple - gestion de la mémoire implicite - librairie standard importante
- autres avantages - portable / multi-plateformes - permet d'utiliser des librairies dans d'autres langages facilement

[cf aussi intro cours scipy/varoquaux]

1.4 Caractéristiques de base

- langage interprété (avec bytecode)
- gestion mémoire transparente (ou presque)
- typage implicite
- typage dynamique
- orienté objet (pas nécessaire mais utile)

2 Syntaxe de base

- typage et types simples
- structure contrôle
- types natifs structurés
- fonctions

2.1 ## Types de base

- entiers (integer) 1, -3
- nombre à virgule flottante (float) 2.0, 5.45
- chaines de caractères (str) "hello" ou 'hello'
- booléen (bool) True False

On dispose des **opérateurs** classiques sur ces types de données.

2.2 ## Typage

le typage est implicite en python :

```
In [73]: a = 0
         b = "hello"
         c = 1.2

In [74]: print(c)

1.2
```

```
In [75]: print(a,b,c)
```

```
0 hello 1.2
```

```
In [76]: print(type(a))
         a = a + c
         print(a)
         print(type(a))
```

```
<class 'int'>
1.2
<class 'float'>
```

2.3 Structures de contrôles

Les structures imbriquées sont indiquées par des **tabulations**

```
In [5]: # Affectations
        b = 2
        a = 12 + b
        # Tests: tabulations obligatoires
        if a>b:
            print(a)
            print("?")
        else:
            print(b)
```

```
14
?
```

```
In [6]: if a==1:
            print ("   ok")
        elif a==2:
            print("toujours ok")
        else:
            print("là c'est trop" )
```

```
là c'est trop
```

2.4 Boucles

```
In [77]: # le corps de la boucle est indiquée par des tabulations
         a = 5
         while a>0:
             a = a - 1
             print(a,end=" ")
         print("fini")
```

```
4 3 2 1 0 fini
```

```
In [8]: for i in range(10):
            print(i,end=" ")
```

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

"for" itère sur une séquence de valeurs

2.5 Fonctions

le corps des fonctions est indiqué par des **tabulations**

```
In [9]: def max(a,b):
            if a>b:
                return a
            else:
                return b

        print(max("fgjsd","jkjk"))
```

```
jkjk
```

Attention ! les paramètres ne sont pas typés : attention au sens des opérateurs

```
In [10]: print(max("bonjour","hello"))
         print(max([1,2],[1,3]))
```

```
hello
[1, 3]
```

```
In [11]: print(max("hello",34))
```

```
        ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)

        <ipython-input-11-12298ae126bc> in <module>
    ----> 1 print(max("hello",34))


        <ipython-input-9-a0de9a245bba> in max(a, b)
          1 def max(a,b):
    ----> 2     if a>b:
          3         return a
          4     else:
```

```
      5         return b


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

2.6 Types structurés natifs

En plus des types de base, python peut manipuler directement les types de données structurés suivants :

- tuples
- listes
- tables associatives (appelées dictionnaires)

Dans tous les cas, on accède aux éléments avec un index, signalé entre []
Dans le cas des listes et tuples, c'est un entier donnant la position (commence à 0)
Dans le cas des tables, l'index fait partie de la définition de la table

```
In [80]: # tuple ... on peut mélanger les types
         t=(1,3,4,"hello")
         print(t[2])
```

```
4
```

```
In [13]: (1,2,3)+(4,5)
```

```
Out[13]: (1, 2, 3, 4, 5)
```

```
In [14]: # ne  peut être modifié "immutable"
         t[0] = 0
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-14-46b9e9cc11de> in <module>
  1 # ne  peut être modifié "immutable"
----> 2 t[0] = 0


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

```
In [81]: # listes ... on peut mélanger les types, mais cela n'a pas trop de sens
         l=[1,2,3,4,"hello"]
         print(l[1])
```

```
2
```

```
In [16]: [1,2,3]+[4,5]

Out[16]: [1, 2, 3, 4, 5]

In [17]: # mutable
         l[4] = 6
         print(l)

[1, 2, 3, 4, 6]
```

2.7 Indexation des séquences

valable pour tuple, liste, et chaines de caractères

```
In [18]: # indexation négatives depuis la fin
         l[-1]
         print(l)

[1, 2, 3, 4, 6]


In [19]:  # "tranches": deuxième borne non incluse
         print(l[2:5])
         print(l[:2])
         print(l[1:-2])
         print(l[3:4])
         print(l[:3]+l[3:])

[3, 4, 6]
[1, 2]
[2, 3]
[4]
[1, 2, 3, 4, 6]


In [20]: # avec des sauts
         print(l[::2])
         print(l[::-1])

[1, 3, 6]
[6, 4, 3, 2, 1]
```

La variable liste est en fait un nom pour une donnée (un peu comme un pointeur)

```
In [21]: l = [1,2]
         a = l
         a[0]=8
         print(a)
         print(l)
```

```
[8, 2]
[8, 2]
```

Mais les sous-listes sont copiées:

```
In [22]: l = [1,2,3]
         a = l[:]
         b = l[::-1]
         a[0] = 8
         b[0] = -2
         print(l)
```

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

2.8 Dictionnaires (Tables associatives)

```
In [23]: #
         d = {"a":(1,2), (1,2):"a", "bxxxxxz":35, 45:"bon exemple, merci"}
         print(d[45])
```

```
bon exemple, merci
```

là encore pas trop de contraintes, mais attention à ce que ça veut dire
Les entrées sont créées dynamiquement :

```
In [24]: d["rt"] = 0
         print(d)
```

```
{'a': (1, 2), (1, 2): 'a', 'bxxxxxz': 35, 45: 'bon exemple, merci', 'rt': 0}
```

Tout type immuable peut servir d'index (cela exclut donc listes et dictionnaires)

```
In [25]: d2 = {1:3, 4:5, 1:123}
         d3 = {(1,2):"?",(4,5):"."}
         # seule contrainte: index immutable
```

2.8.1 Structures vides

```
In [28]: ()  # ---> tuple
         []  # ---> liste
         {}  # ---> dictionnaire

         print((2,)) # tuple à un seul élément
         print((2))
```

```
(2,)
2
```

2.9 Itérations

Les types structurés sont prévus pour être l'objet d'itérations

```
In [29]: for x in [1,2,3,4,5]:
             print(x,end=" ")


         print()
         for x in ("a",3,-1.5):
             print(x,end=" ")
```

```
1 2 3 4 5
a 3 -1.5
```

```
In [30]: d = {1:2,10:3,100:45}
         for x in d:
             print(x, d[x])
```

```
1 2
10 3
100 45
```

```
In [31]: d =  {1:2,10:3, 100 :45}
         for cle in d:
             print(d[cle],end=" ")
```

```
2 3 45
```

2.10 Opérations sur les types structurés

```
In [32]: #affectation : on peut décomposer
         (a,b,c) = (1,2,3)
         print(a,b)

         (a,b) = (b,a)
```

```
        # moins utile mais possible
        l = [1,2,3]
        [a,b,c] = l
        print(b)
```

```
1 2
2
```

```
In [33]: # dans les itérations
         l = [(1,2),(1,3),(4,5)]
         for (i,j) in l:
             print(i+2*j)
```

```
5
7
14
```

```
In [34]: for x in l:
             print(x[0]+2*x[1])
```

```
5
7
14
```

```
In [35]: # application  : échange de variables
         (a,b) = ( b,a)
         print(a,b)
```

```
2 1
```

```
In [36]: # une séquence utile sur les entiers
         print(range(1,10))
         print(list(range(1,10)))
```

```
range(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2.11 ## Fonctions, opérateurs et méthodes

Les types de données en python sont en fait des **objets**, et de nombreuses opérations que l'on peut faire dessus sont définies comme des méthodes de ces objets.

La syntaxe pour ces opérations suit le format suivant:

objet.methode(paramètres)

Exemple

```
In [37]:   # définissons une liste
           l = [3,12,-6,4]
           # append est le  nom de la méthode qui ajoute un élément à une liste
           l.append(20)
           print(l)
```

```
[3, 12, -6, 4, 20]
```

2.12 ## Quelques méthodes et opérateurs utiles

Pour les listes:

- append: ajoute un élément en fin de liste
- extend: ajoute une liste à la fin d'une liste (cf aussi opérateur +)
- sort: trie sur place par ordre croissant, si les valeurs sont comparables
- reverse: renverse sur place
- "+" (opérateur) concatène 2 listes pour faire une 3e liste
- sorted (fonction) trie en copiant

cf la documentation python pour les listes

```
In [38]: l = [1,2]
         l.extend([3,4])
         print(l+[4,5])
         print(len(l))
         l.append(4)
         print(l)

         print([].append(3))
```

```
[1, 2, 3, 4, 4, 5]
4
[1, 2, 3, 4, 4]
None
```

Pour les dictionnaires:

- keys: liste des "clefs" ou valeurs de l'index de la table
- values: liste des valeurs
- items: listes des couples (clef,valeur)
- get: récupère une valeur avec la clef, ou une valeur par défaut
- initialisation à partir de couples: dict

cf la documentation python pour les dict
Pour les chaines:

- "+" (opérateur) pour concaténer 2 chaines
- join pour concaténer une liste de chaine avec un séparateur
- split pour diviser une chaine selon un séparateur

cf la documentation python pour les chaines
Pour tous:

- len: **fonction** donnant le nombre d'éléments
- in: **opérateur** (infixe), teste l'appartenance (dans le cas d'un dictionnaire, teste la présence d'une clef)

Pour les dictionnaires:

- del d[clef] supprime une entrée

```
In [39]: d = {"a":23, "b":35, "x":27}
         print(d)
         print(d.keys())
         print(d.values())
         print(d.items())
         print(len(d))
         print(d.get('a',0))
         print(d.get('d',0))
         del d['a']
         print(d)
         print(d.get("y",0))
         print(dict([("a",1),("b",3)]))
         print('x' in d)
```

```
{'a': 23, 'b': 35, 'x': 27}
dict_keys(['a', 'b', 'x'])
dict_values([23, 35, 27])
dict_items([('a', 23), ('b', 35), ('x', 27)])
3
23
0
{'b': 35, 'x': 27}
0
{'a': 1, 'b': 3}
True
```

```
In [40]: print("hello"+"world")
         print("hello world !".split())
         print(" ".join(["hello","world","!"]))
```

```
helloworld
['hello', 'world', '!']
hello world !
```

3 Itérations : alternatives

```
In [41]: # iterations abrégées
         [x**2 for x in range(10)]

         l1 = [1,2,3,9,100]
         l2 = [x**2 for x in l1]
         print(l1)
         print(l2)
```

```
[1, 2, 3, 9, 100]
[1, 4, 9, 81, 10000]
```

```
In [42]: # itération avec l'indice:
         for (i,x) in enumerate(range(100,110)):
             print(i,"->",x,end=" ")
```

```
0 -> 100 1 -> 101 2 -> 102 3 -> 103 4 -> 104 5 -> 105 6 -> 106 7 -> 107 8 -> 108 9 -> 109
```

```
In [43]: # itération abrégée des dictionnaires:
         d = {1:2,3:4,5:9}
         d2 = {x:d[x]**2 for x in d}
         d2
```

```
Out[43]: {1: 4, 3: 16, 5: 81}
```

3.0.1 Entrées sorties:

L'affichage de base se fait avec la fonction "print", quel que soit le type.

```
In [44]: print(3)
         print("a")
         print([1,2])
         print({"a":2})
         print("a")
```

```
3
a
[1, 2]
{'a': 2}
a
```

On peut enchainer les impressions, avec une tabulation

```
In [45]: print(a,b)
```

```
2 1
```

```
In [46]: # argument de fin de ligne
         print(a,end="///")
         print(b)

2///1
```

3.1 Retour sur les chaines de caractères

les chaines sont des séquences

```
In [47]: print("hello"[4])
         print("hello"[2:5])

o
llo
```

Il n'y a pas de type caractères: une chaine de longueur 1 est une chaine

```
In [48]: print(type("hello"))
         print(type("hello"[0]))

<class 'str'>
<class 'str'>
```

```
In [49]: #plusieurs façons de décrire une chaine
         "blabla"
         ' ou blabla '

         # guillemets triples pour étaler sur plusieurs lignes

         """ encore plus
         de blabla"""

Out[49]: ' encore plus \nde blabla'
```

```
In [50]: #Quelques méthodes
         a = "Hello  world !"# deux espaces entre hello et world
         print(a.upper())
         print(a.lower())
         print(a.replace(" ","---").replace("-","?"))
         items = a.split()
         print(items)
         print("---".join(items))
         print(a.split("l"))
```

```
HELLO  WORLD !
hello  world !
Hello??????world???!
['Hello', 'world', '!']
Hello---world---!
['He', '', 'o  wor', 'd !']
```

```
In [51]: a = "hello"
         a =  a.replace("l","r")
         print(a)
```

```
herro
```

Pour gérer les fichiers, on doit gérer l'ouverture d'un canal:

```
In [82]: f=open("monfichier.txt","w") # écriture
         f.write("blabla\n")
         f.write("bla")
         f.close()
```

```
In [53]: f=open("monfichier.txt") # lecture
         print(f.read())
         f.close()
         f=open("monfichier.txt") # lecture
         print(f.readlines())
         f.close()

         l = open("monfichier.txt").readlines()

         for line in open("monfichier.txt"):
             print(line)
```

```
blabla
bla
['blabla\n', 'bla']
blabla

bla
```

```
In [54]: "blabla\n".strip()
```

```
Out[54]: 'blabla'
```

En écriture

```
In [55]: f=open("monfichier.txt","w") # écriture
         print("blabla",file=f)
         print("bla",file=f)
         f.close()
```

4 Importer des modules

La librairie python contient de très nombreux modules que l'on peut charger à la demande.
Syntaxe :

```
In [56]: import math
         math.exp(1)

Out[56]: 2.718281828459045

In [57]: # alternativement
         from math import exp
         from math import * # ahem
         exp(1)

Out[57]: 2.718281828459045

In [58]: import math as fct_math
         print(fct_math.exp(1))
         from math import log as log2
         print(log2(1))

2.718281828459045
0.0
```

4.1 Définition avancée de fonctions

le format vu jusqu'ici pour les fonctions est en fait un schéma de base avec des arguments fixes: >> def nom_fonction(argument1,arg2,...,argn):

On peut aussi mettre des arguments optionnels de la façon suivante : >> def nom_fonction(arg1,...,argn, option1 = defaut1, option2 = defaut2, ...)

```
In [79]: def message(texte, signature, politesse = "cordialement"):
             print(texte)
             print(politesse+"," )
             print(signature)

         message("blabla","Emmanuel")

blabla
cordialement,
Emmanuel


In [80]: message("blabla","Emmnanuel Macron",politesse = "sincerement")

blabla
sincerement,
Emmnanuel Macron
```

```
In [3]: # on peut avoir un nombre d'argument variable en passant un pointeur sur toute la séquen
        def variable(*tous_les_args):
            for x in tous_les_args:
                print(x,end=" ")


        variable(3,"+",4,"=",3+4)
```

```
3 + 4 = 7
```

```
In [82]: # on peut même récupérer les options; on verra plus tard à quoi ça sert
         def illisible(*args,**options):
             print(args)
             print(options)


         illisible("euh","comment","dire",point=23,extra=2,bonus="craquage")
```

```
('euh', 'comment', 'dire')
{'bonus': 'craquage', 'extra': 2, 'point': 23}
```

```
In [ ]: # on peut définir des fonctions de fonctions
        def deux_f(f):
            def fbis(x):
                return 2*f(x)
            return fbis

        from math import sin, cos
        f1 = deux_f(sin)
        f2 = deux_f(sin)
        print(f1)
        print(f2)

        print(deux_f(cos)(0.5))

        #compliqué ...
```

Il est souvent utile de définir un objet fonction jetable comme suit (fonction "anonyme") :

```
In [59]: lambda a,b: a+b #  -> la fonction addition de deux variables
```

```
Out[59]: <function __main__.<lambda>(a, b)>
```

```
In [10]: # on reprend l'exemple plus haut
         def deux_f(f):
             return lambda x: 2*f(x)
```

```
        sin2 =  deux_f(sin)
        sin2(0.5)
```

```
<function deux_f.<locals>.fbis at 0x10a5fae18>
<function deux_f.<locals>.fbis at 0x10a5faea0>
1.7551651237807455
```

```
Out[10]: 0.958851077208406
```

```
In [60]: # pourquoi faire des fonctions retournant des fonctions ? opérations plus générales
         def plus(f,g):
             return lambda x: f(x)+g(x)

         a = plus(cos,sin)
         a(0.5)
```

```
Out[60]: 1.3570081004945758
```

4.2 Passage des paramètres

il y a deux façons de passer des paramètres à une fonction: par **valeur** ou par **référence**

dans le premier cas le paramètre transmis est une valeur (éventuellement contenue dans une variable): le contenu du paramètre est localement copié.

dans le deuxième cas, le paramètre est une variable, qui reste accessible dans le contexte de la fonction.

En python, les fonctions passent toujours des références.

Mais il faut bien comprendre comment marchent les affectations: le paramètre formel d'une fonction "renomme" la valeur.

Exemple

```
In [61]: def test1(a):
             a="aa"
             return a

         b="zz"
         c = test1(b)
         print(b, c)

         # a et b réfèrent à la même valeur (1.2), puis a est "réaffecté" à 3
```

```
zz aa
```

Les types complexes sont passés aussi par référence mais on peut accéder à leurs éléments sans renommer quoi que ce soit. Une fonction peut donc éventuellement avoir accès à la variable de l'appel:

```
In [62]: def test2(l):
             l[0]=0
             return l

         l1 = [1,2,3,4]
         print(test2(l1))
         print(l1)
         print(test2([1,2]))

         # l1 et l réfèrent à la même liste de valeur [1,2,3,4]
```

```
[0, 2, 3, 4]
[0, 2, 3, 4]
[0, 2]
```

5 Types structurés (suite): les ensembles

Le type 'set' modélise l'équivalent d'un ensemble en mathématique: une collection d'items sans ordre spécifique.

```
In [63]: # initialisation et ajouts

         a = set([1,3,4,5,1,1,5])
         b = set()
         c = {1,2,3}
         b.add(4)
         b.add(6)

         print(a)
         print(b)
```

```
{1, 3, 4, 5}
{4, 6}
```

```
In [64]: set([1,2])==set([2,1])


         a.remove(3)
         print(a)
```

```
{1, 4, 5}
```

```
In [65]: # appartenance, intersection, union
         x = 3
```

```
         print(x in b)
         print(a,b)
         print(a & b)
         print(a | b )
```

```
False
{1, 4, 5} {4, 6}
{4}
{1, 4, 5, 6}
```

```
In [66]: # cardinal, inclusion , différence
         print(len(a))
         print(set([1,3]) <= a)
         print(a - b)
         print(b - a)
```

```
3
False
{1, 5}
{6}
```

```
In [67]: # suppression
         a.add(4)
         a.remove(4)
         b.clear()

         print(a)
         print(b)
         a.add(4)
```

```
{1, 5}
set()
```

5.1 Exceptions

Python définit un ensemble d'exceptions, qui sont un type à part et qui peuvent être aussi étendues, et bien sûr levées et rattrapées.

Quelques exemples:

```
In [68]: a = 0.
         b = 15
         print(b/a)
```

```
---------------------------------------------------------------------------
```

```
        ZeroDivisionError                         Traceback (most recent call last)

        <ipython-input-68-022aea3d6c79> in <module>
          1 a = 0.
          2 b = 15
    ----> 3 print(b/a)


        ZeroDivisionError: float division by zero
```

```
In [69]: l = []
         l[0]
```

```
        ---------------------------------------------------------------------------

        IndexError                                Traceback (most recent call last)

        <ipython-input-69-7985abb5df3c> in <module>
          1 l = []
    ----> 2 l[0]


        IndexError: list index out of range
```

```
In [70]: a = 0.5
         b = 15
         try:
             print(b/a)
         except ZeroDivisionError:
             print(0)

         l = []
         try:
             print(l[0])
         except IndexError:
             print("?")
```

```
30.0
?
```

```
In [71]: l = []
         try:
             print(l[0])
         except IndexError:
             raise IndexError("m'enfin ?!?!!!")
```

```
        except:
            print("erreur inattendue !")
            raise
        finally:
            l.append(1)
            print(l)
```

```
[1]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

<ipython-input-71-8e951cb9800e> in <module>
  2 try:
----> 3     print(l[0])
  4 except IndexError:


IndexError: list index out of range


During handling of the above exception, another exception occurred:


IndexError                                Traceback (most recent call last)

<ipython-input-71-8e951cb9800e> in <module>
  3     print(l[0])
  4 except IndexError:
----> 5     raise IndexError("m'enfin ?!?!!!")
  6 except:
  7     print("erreur inattendue !")


IndexError: m'enfin ?!?!!!
```

```
In [72]: l = []
        try:    # passage risqué
            print(l[0])
        except IndexError:  # raté !
            raise IndexError("m'enfin ?!?!!!")
            #raise IndexError
        else: # si c'est bon on s'assure de ça (peut lever une erreur aussi !)
            print(1/l[0])
```

```
        finally: # dans tous les cas on veut faire ça
            l.append(1)
            print(l)
```

```
[1]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

<ipython-input-72-09596d5d2ded> in <module>
  2 try:    # passage risqué
----> 3     print(l[0])
  4 except IndexError:  # raté !


IndexError: list index out of range


During handling of the above exception, another exception occurred:


IndexError                                Traceback (most recent call last)

<ipython-input-72-09596d5d2ded> in <module>
  3     print(l[0])
  4 except IndexError:  # raté !
----> 5     raise IndexError("m'enfin ?!?!!!")
  6     #raise IndexError
  7 else: # si c'est bon on s'assure de ça (peut lever une erreur aussi !)


IndexError: m'enfin ?!?!!!
```

Index des exercices:
Ecrire des fonctions faisant les opérations suivantes :

1. prendre une liste et enlever les doublons, dans un premier temps on supposera la liste triée, puis on considèrera le cas général
2. prendre une liste de listes d'entiers, retourner la liste "plate" des éléments
3. extraire d'une liste d'entiers les valeurs consécutives qui se suivent, par exemple [3,8,9,10,15,16] -> [(8,9),(9,10),(15,16)]
4. `Comptez (dans un dictionnaire) le nombre d'occurrence d'entiers contenus dans une liste. Pa`

5. Faire une table qui enregistre les combinaisons de deux dés à six faces, avec comme clefs la valeur de la somme des deux dés et comme valeurs les combinaisons de deux dés.

6. Faire le crible d'Eratosthène, enregistré dans un dictionnaire : on commence avec comme clefs tous les entiers de 1 à n (fixé), et finalement on ne garde que les clefs qui sont des nombres premiers.

7. Faire l'inversion (clefs/valeurs) d'une table

8. Faire une fonction qui teste si deux mots ont les mêmes lettres; en faire une autre qui teste si ce sont des anagrammes (c'est-à-dire qu'ils ont les mêmes lettres le même nombre de fois). Tester si une phrase contient toutes les lettres de l'alphabet (pangrammes).

9. Cryptage / décryptage

Problèmes:

- définir un ensemble de fonctions pour manipuler des représentations de polynômes
- définir un ensemble de fonctions manipulant des représentations de vecteurs "creux" de dimensions quelconque. Un vecteur creux est un vecteur dont beaucoup de coordonnées sont nulles, et pour lesquels on veut optimiser la place mémoire nécessaire. Ici on le fera avec des dictionnaires.

```
In [97]: pangram = "the quick brown fox (!) jumps over the lazy dog"
```