Cours 2 : chaînes, ensembles et dictionnaires

Chaînes de caractères

Une chaîne est une suite finie de caractères (ou *string*) entourée par des apostrophes ou des guillements. Elle n'est pas mutable (on ne peut pas changer la valeur d'un caractère).

In [2]:
```
s='Ceci est une chaîne'
s
```

Out [2]:
```
'Ceci est une chaîne'
```

In [3]:
```
s[3]
```

Out [3]:
```
'i'
```

In [4]:
```
s[3]='a'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-1d84d5b091e8> in <module>()
----> 1 s[3]='a'

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

On doit utiliser un caractère d'échappement pour utiliser certains symboles, on concatène avec + et `len()` donne la longueur

In [3]:
```
t=', et l\'été est fini'
s+t
len(s+t)
```

Out [3]:
```
38
```

La chaîne est un objet *itérable*, i.e. dont on peu parcourir les valeurs, p.e. dans un `for`.

In [22]:
```
for c in s:
    print(c, end='')
```

```
Ceci est une chaîne
```

On peut en extraire une tranche *slice*, c'est un mécanisme *très* puissant en python

In [23]:
```
t[7:10]+t[-5:]
```

Out [23]:
```
'été fini'
```

`replace()` permet de remplacer un caractère ou une sous-chaîne, `count()` compte le nombre de caractères.

In [24]:
```
t.replace('été','automne')[5:15]+'commence'
```

Out [24]:
```
"l'automne commence"
```

In [40]:
```
s.count('e')
```

Out [40]:
```
4
```

Beaucoup de méthodes sont associées, p.e. `lower()` et `upper()` pour changer la casse

In [12]:
```
s.lower()+t[0:7]+t[7:10].upper()+t[10:]
```

Out [12]:
```
"ceci est une chaîne, et l'ÉTÉ est fini"
```

On peut convertir les caractères numériques en nombre avec la fonction `ord()` dont la fonction réciproque est `chr()`

In [39]:
```
print(ord('A'))
print(chr(65+25))
```

```
65
Z
```

C'est utile pour ramener (et recoder) {a,...,z} dans {0,..25} !

In [57]:
```
u=s.lower().replace(' ','').replace('\'','').replace('î','i')
for c in u:
    print(ord(c)-ord('a'),end=',')
```

```
2,4,2,8,4,18,19,20,13,4,2,7,0,8,13,4,
```

Fonctions ou méthodes utiles pour la suite:

- retirer les caractères accentués (comme î) ou entrelacés (comme œ) en changeant le codage des caractères par la méthode `replace()`

In [42]:
```
import unicodedata
def remove_accents(input_str):
    input_str = input_str.replace('œ', 'oe').replace('æ', 'ae')
    nfd_form = unicodedata.normalize('NFD', input_str).encode('ASCII','ignore').deco
    return(nfd_form)
```

In [43]:
```
remove_accents('c\'était un œuf')
```

Out [43]:
```
"c'etait un oeuf"
```

- récupérer les lettres d'une chaîne sans les occurrences multiples (avec changement de type)

In [46]:
```
"".join(list(set('aabccccccd')))
```

Out [46]:
```
'cdba'
```

In [47]:
```
list('aabccccccd')
```

Out [47]:
```
['a', 'a', 'b', 'c', 'c', 'c', 'c', 'c', 'c', 'd']
```

Et si ça ne suffit pas, regardez les méthodes de chaînes ou votre cours de L1 (cours 3?)

Ensembles

On peut utiliser `set` de `Python` ou la classe `FiniteSet`de `SymPy`

Création d'un ensemble

Avec la structure `set`de base de `Python3`

In [7]:
```
s={1,2,3}
s
```

Out [7]:
```
{1, 2, 3}
```

on peut aussi changer le type d'un élément liste en un ensemble

In [63]:
```
liste=[4,5,6]
set(liste)
```

Out [63]:
```
{4, 5, 6}
```

En utilisant `FiniteSet`de `Sympy`, de façon analogue

In [8]:
```
from sympy import FiniteSet
s1=FiniteSet(1,2,3)
s1
```

Out [8]:
```
{1, 2, 3}
```

In [64]:
```
FiniteSet(*liste)
```

Out [64]:
```
{4, 5, 6}
```

Ci-dessus on a utilisé `f(*liste)` qui permet d'appeler `f` avec comme arguments les éléments de `liste`; p.e. `f(*(1,2,3))` est synonyme de `f(1,2,3)`

Deux notations pour l'ensemble vide:

- en `python`

In [66]:
```
e=set()
e
```

Out [66]:
```
set()
```

- avec `FiniteSet`, on utilise `EmptySet`

In [68]:
```
from sympy import EmptySet
e=EmptySet()
e
```

Out [68]:
```
EmptySet()
```

Cardinalité et appartenance

La fonction `len()` compte le nombre d'éléments

In [69]:
```
len(s)
```

Out [69]:
```
3
```

In [70]:
```
len(s1)
```

Out [70]:
```
3
```

Et l'appartenance se teste par l'instruction `in`

In [71]:
```
1 in s
```

Out [71]:
```
True
```

In [72]:
```
4 in s1
```

Out [72]:
```
False
```

Opérations sur les ensembles

Les méthodes `union` et `intersection` fonctionnent dans les deux cas

In [75]:
```
print(s)
t={3,4,5}
s.union(t)
```

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

Out [75]:
```
{1, 2, 3, 4, 5}
```

In [76]:
```
t1=FiniteSet(3,4,5)
s1.union(t1)
```

Out [76]:
```
{1, 2, 3, 4, 5}
```

In [77]:
```
s.intersection(t)
```

Out [77]:
```
{3}
```

In [78]:
```
s1.intersection(t1)
```

Out [78]:
```
{3}
```

La différence entre deux ensembles fonctionne dans les deux cas

In [86]:
```
s-t
```

Out [86]:
```
{1, 2}
```

In [87]:
```
s1-t1
```

Out [87]:
```
{1, 2}
```

La différence principale entre `set` et `FiniteSet` se fait pour le produit cartesien, plus difficile sans `sympy` car on a besoin de la librairie `itertools`

In [6]:
```
import itertools
itertools.product(s,t) # qui retourne un itérateur
```

Out [6]:
```
<itertools.product at 0x10e38a288>
```

In [83]:
```
list(itertools.product(s,t))
```

Out [83]:
```
[(1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 3), (3, 4), (3, 5)]
```

tandis qu'avec `FiniteSet`:

In [84]:
```
s1*t1
```

Out [84]:
```
{1, 2, 3} x {3, 4, 5}
```

In [85]:
```
list(s1*t1)
```

Out [85]:
```
[(1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 3), (3, 4), (3, 5)]
```

Et il est bien plus facile de calculer l'ensemble des parties d'un ensemble avec `FiniteSet`

In [89]:
```
s1.powerset()
```

Out [89]:
```
{EmptySet(), {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}}
```

juste pour info, voici ce qu'il faut faire sans `FiniteSet`

In [90]:
```
from itertools import *
def subsets(iterable):
    xs=list(iterable)
    return chain.from_iterable(combinations(xs,n)for n in range(len(xs)+1))
```

In [92]:
```
list(map(set,subsets(s)))
```

Out [92]:
```
[set(), {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
```

Construction par compréhension

Un ensemble peut aussi se construir par **compréhension** (ici les carrés multiples de 3)

In [96]:
```
E={x*x for x in range(10) if x%3==0}
E
```

Out [96]: {0, 9, 36, 81}

La primitive `filter(function,iterable)` permet de parcourir un objet itérable (`str, list, tuple, range, set, dict`) et d'obtenir un itérateur sur les objets retenus. Quitte ensuite à le parcourir, le transformer en liste, en ensemble, etc. Attention, une fois utilisé, il est épuisé.

In [103]:
```
F=filter(lambda x:x%2==0,{3,2,5,4,1,6,9,0})
F
```

Out [103]: <filter at 0x1074c7978>

In [98]:
```
list(F)
```

Out [98]: [0, 2, 4, 6]

In [101]:
```
set(F)
```

Out [101]: {0, 2, 4, 6}

Attention `FiniteSet(F)` retourne une erreur !

Conclusion

La classe `FiniteSet` apporte des améliorations au type `set` de base. Une alternative est d'utiliser les `frozenset` qui permet de créer des ensembles de listes ou d'ensembles mais la classe `FiniteSet` de `sympy` est plus accessible.

Les dictionnaires

Un dictionnaire est une collection non numérotée de couples `clé:valeur` où `clé` est un objet non mutable et `valeur` n'importe quelle valeur. **Attention** toutes les clés doivent être distinctes.

Création d'un dictionnaire

In [107]:
```
stock={'poires':51,'pommes':243}
stock
```

Out [107]: {'poires': 51, 'pommes': 243}

In [108]:
```
len(stock)
```

Out [108]: 2

In [109]:
```
'pommes' in stock
```

Out [109]: True

In [110]:
```
'pêches' in stock
```

Out [110]: False

In [111]:
```
stock['poires']
```

Out [111]: 51

Le dictionnaire vide se note `{}` ou `dict()`

On peut modifier la valeur associée à une clé ou ajouter un nouveau couple clé:valeur

In [113]:
```
stock['pommes']=100
stock['bananes']=23
stock
```

Out [113]: {'poires': 51, 'pommes': 100, 'bananes': 23}

Accès aux éléments d'un dictionnaire

Il est facile d'accéder à la valeur associée à une clé. C'est le principal intérêt de cette structure. La recherche est à sens unique. On va de la clé vers la valeur.

In [115]:
```
stock['bananes']
```

Out [115]: 23

La clé est non mutable. Donc principalement des chaînes et des nombres mais pas de listes.

In [14]:
```
dico={1:'one',2:'two',3:'three'}
```

Modifier un dictionnaire

Un dictionnaire est **mutable** par la méthode `update()` qui permet d'ajouter un dictionnaire à un dictionnaire existant

In [15]:
```
dico.update({4:'four',5:'five'})
dico
```

Out [15]: {1: 'one', 2: 'two', 3: 'three', 4: 'four', 5: 'five'}

On peut retirer un élément du dictionnaire par la méthode `pop()` qui retourne la valeur `val` associée à la clé et qui supprime le couple `clé:val` associé

In [16]:
```
dico.pop(5)
dico
```

Out [16]: {1: 'one', 2: 'two', 3: 'three', 4: 'four'}

Accéder aux clés ou aux valeurs

on peut obtenir la liste de toutes les clés ou de toutes les valeurs par les méthodes `keys()` et `values()` qui retournent des **vues** *views* sur les clés ou les valeurs

In [17]:
```
dico.keys()
```

Out [17]:
```
dict_keys([1, 2, 3, 4])
```

In [18]:
```
dico.values()
```

Out [18]:
```
dict_values(['one', 'two', 'three', 'four'])
```

Les vues sont des itérables qu'on peut parcourir par un `for`, ou transformer en liste ou en ensemble, les sommer,...

In [19]:
```
list(dico.values())
```

Out [19]:
```
['one', 'two', 'three', 'four']
```

In [20]:
```
list(dico.keys())
```

Out [20]:
```
[1, 2, 3, 4]
```

In [21]:
```
sum(dico.keys())
```

Out [21]:
```
10
```

Avec un `for` par défaut, on itère sur les clés

In [22]:
```
for k in dico:
    print(k)
```

```
1
2
3
4
```

Mais on peut décider d'itérer sur les valeurs

In [23]:
```
for k in dico.values():
    print(k)
```

```
one
two
three
four
```

Effacer un dictionnaire

Par l'instruction `del`

In [132]:
```
del dico
```

In [133]:
```
dico
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-133-efcd661f264a> in <module>()
----> 1 dico
```

```
NameError: name 'dico' is not defined
```

Une étude de cas

On veut tracer l'histogramme des fréquences des lettres qui apparaissent dans le dictionnaire français (ici le Littré compressé au format `zip`)

In [24]:
```
from sympy import *
```

In [25]:
```
init_printing()
```

In [26]:
```
def dictionnaire(fichier):
    import zipfile
    import sys
    f=zipfile.ZipFile(fichier,'r')
    l=f.namelist()       # retourne la liste des éléments de l'archive par nom
    if len(l) != 1:
        print('pas bon')
        sys.exit(1)
    r = f.read(l[0]).decode(encoding="UTF-8", errors="strict").split("\n")
    return [m for m in r if len(m) != 0]
```

In [27]:
```
littre=dictionnaire("littre.zip")
```

In [28]:
```
littre[112]       # regardons le contenu à l'index 112
```

Out [28]:
```
'abondance'
```

On nettoie les symboles. Le dictionnaire contient des caractères accentués, spéciaux, etc... Regardons son alphabet :

In [35]:
```
alphabet=set()
for m in littre:
    alphabet=alphabet.union(set(m))
print(sorted(alphabet))
```

```
['!', "'", '(', ')', '.', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
```

In [19]:
```
import unicodedata
def remove_accents(input_str):
    input_str = input_str.replace('œ', 'oe').replace('æ', 'ae')
    nkfd_form = unicodedata.normalize('NFD', input_str).encode('ASCII','ignore').dec
    return(nkfd_form)
```

In [33]:
```
littre2=[remove_accents(m) for m in littre if len(m) != 0]
```

On crée un dictionnaire pour compter les occurrences de chaque lettre de l'alphabet réduit à {a,..z}. On parcourt le Littré nettoyé mot par mot et pour chaque mot, caractère par caractère. On incrémente la

valeur du dictionnaire `lettres` si on trouve le caractère dans l'alphabet.

In [66]:
```
def occurrences(dico):
    lettres = {chr(i):0 for i in range(ord('a'), ord('z') + 1)}
    for w in dico:
        for c in w:
            if c in lettres:
                lettres[c] = lettres[c] + 1
    return lettres
```

In [67]:
```
occur=occurrences(littre2)
```

In [68]:
```
occur['a'] # le nombre d'occurrences de la lettre a
```

Out [68]:
```
51230
```
51230

In [69]:
```
sum(occur.values()) # compte le nombre total de lettres comptées
```

Out [69]:
```
641234
```
641234

In [70]:
```
somme=0
for i in occur.values():
    somme+=i
print(somme)
```

```
641234
```

On transforme les valeurs du dictionnaire des occurrences en pourcentage d'apparition de la lettre de l'alphabet

In [74]:
```
def normalise(lettres):
    somme = sum(lettres.values())
    return {c:round(lettres[c] * 100 / somme, 1) for c in lettres}
occur=normalise(occur)
```

In [75]:
```
print(occur,end='')
```

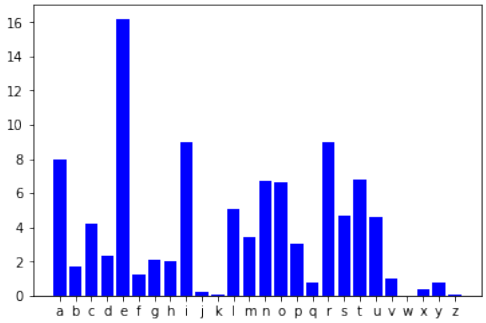
```
{'a': 8.0, 'b': 1.7, 'c': 4.2, 'd': 2.3, 'e': 16.2, 'f': 1.2, 'g': 2.1, 'h': 2.0,
```

On affiche enfin l'histogramme des fréquences des lettres en français (pour le dictionnaire Littré).

In [50]:
```
import matplotlib.pyplot as plt
```

In [76]:
```
plt.bar(list(occur.keys()),occur.values(),color='b')
```

Out [76]:
```
<BarContainer object of 26 artists>
<Figure size 432x288 with 1 Axes>
```



In [77]:
```
sum(occur.values())
```

Out [77]:
```
100.0
```
100.0

Répondons à la question : "quelles sont les lettres les plus fréquences en français?". On trace l'histogramme par ordre décroissant des fréquences.

In [78]:
```
occtri=sorted(occur.items(),key=lambda t:t[1],reverse=True)
print(occtri)
```

```
[('e', 16.2), ('i', 9.0), ('r', 9.0), ('a', 8.0), ('t', 6.8), ('n', 6.7), ('o', 6
```

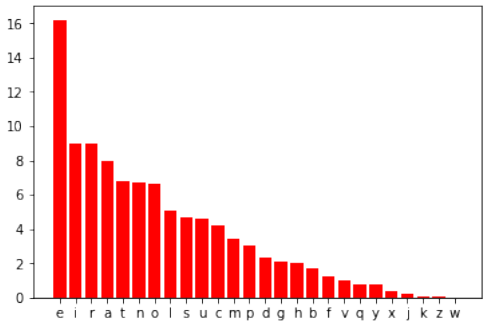
On a utilisé la fonction `sorted()` appliquée sur les valeurs du dictionnaire des occurrences (par `lamba t:t[1]`). Si on avait voulu trier sur les clés, on aurait utilisé (par `lamba t:t[0]`). Mais la sortie de la fonction est une liste et plus un dictionnaire ! On transforme la liste en dictionnaire par `dict()`.

In [79]:
```
occtrid=dict(occtri)
print(occtrid)
```

```
{'e': 16.2, 'i': 9.0, 'r': 9.0, 'a': 8.0, 't': 6.8, 'n': 6.7, 'o': 6.6, 'l': 5.1,
```

In [80]:
```
plt.bar(list(occtrid.keys()),occtrid.values(),color='r')
```

Out [80]:
```
<BarContainer object of 26 artists>
<Figure size 432x288 with 1 Axes>
```



Pourquoi cet histogramme est-il différent des fréquences des lettres en français, p.e. celui-ci ?

In [ ]:
```
```