Introduction à la Programmation Python

Le langage [1/3]

- Variables
- Types de données
- Fichiers

Formation permanente du CNRS, Délégation Alsace
Février - Mars 2017

Auteurs :

- Vincent Legoll (vincent.legoll@iphc.cnrs.fr)
- Matthieu Boileau (matthieu.boileau@math.unistra.fr)

Langage python et sa syntaxe

Variables

Pour accéder aux données, on leur donne des noms que l'on appelle des variables.

Pour donner un nom à une donnée, on utilise l'opérateur d'assignation '='.

Exécutez la cellule suivante (SHIFT+ENTER) pour définir trois variables nommées : age, prénom et taille.

```
In [1]: # Par exemple: donnons la valeur 23 à la variable 'age'
        age = 23
        # Les variables peuvent se référer à divers types de données: des chaînes d
        prenom = 'Julien'
        # Des nombres réels, etc...
        taille = 1.83
```

Pour se servir de la donnée référencée par une variable, il suffit d'utiliser le nom cette variable.

La fonction `print()` affiche à l'écran ce qui lui est passé en paramètre. On peut lui donner plusieurs paramètres séparés par des virgules et print les affichera tous, séparés par un espace.

```
In [2]: print('Julien', 23, 1.83)
        print(prenom, 'a', age, 'ans, et mesure', taille, 'mètre')

Julien 23 1.83
Julien a 23 ans, et mesure 1.83 mètre
```

Les variables peuvent changer et ainsi se référer à une autre donnée.

```
In [3]: age = 23
        print(age)
        age = 24
        print(age)

23
24
```

Les variables n'ont pas de type propre: c'est la donnée qui est typée, pas la variable qui la référence. Elles peuvent donc faire référence à une donnée d'un autre type après une nouvelle assignation.

La fonction `type()` retourne le type effectif de la donnée passée en paramètre.

```
In [4]: # Des nombres
        age = 23
        print(type(age))
        # Les variables peuvent aussi changer de type : chaîne de caractères
        age = 'vingt quatre ans'
        print(type(age))
        # Sans limites...
```

```
        age = 24.5
        print(type(age))
        # Attention aux pièges...
        age = '25'
        print(type(age))
```

```
<class 'int'>
<class 'str'>
<class 'float'>
<class 'str'>
```

Une variable peut être initialisée avec des valeur constantes, comme vu précédement, mais aussi à partir d'autres variables ou des valeurs de retour de fonctions, etc...

La fonction max() retourne le plus grand de ses paramètres.

```
In [5]: a = 1
        b = a
        c = a * 2
        d = max(a, 2, 3, c)
        print(a, b, c, d)
```

```
1 1 2 3
```

Deux variables peuvent référencer le même objet, les modifications faites par l'intermédiaire d'une des variables sont visibles par toutes les autres.

```
In [6]: a = []
        b = a
        print(a, b)
        a.append(1)
        print(a, b)
```

```
[] []
[1] [1]
```

Les variables, une fois définies dans une cellule **exécutée**, continuent d'exister dans les suivantes.

Note

En cas de redémarrage du notebook, toutes les variables existantes sont détruites, il faut donc ré-exécuter les cellules qui les définissent si l'on veut de nouveau pouvoir les utiliser.

```
In [7]: abcd = 1234
```

Ici, la variable nommée 'abcd' survit, d'une cellule à la suivante...

```
In [8]: print(abcd)

1234
```

Si on veut faire disparaître une variable, on peut utiliser le mot clé interne `del`.

```
In [9]: # Cette cellule génère une erreur
        a = 2
        print(a)
        del(a)
        print(a)

2
```

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

        NameError                                 Traceback (most recent call last)

        <ipython-input-9-f69312804df2> in <module>()
          3 print(a)
          4 del(a)
    ----> 5 print(a)


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

Note : `del` est aussi utilisé pour enlever des éléments dans des conteneurs modifiables (listes, dictionnaires). Nous aborderons ce sujet plus tard.

Types de données

Types de base

- None
- Booléens
- Numériques
  - entiers
  - flottants
  - complexes

Séquences

- Chaines de caractères
- listes
- tuples

Conteneurs

- Dictionnaires
- Ensembles

Fichiers

Types de base

None

- Il existe dans python un type de données particulier : `None`.
- `None` représente un objet sans valeur. On s'en sert comme valeur de retour en cas d'erreur ou pour représenter un cas particulier.
- `None` est équivalent à NULL en java, C.

```
In [10]: a = None
         print(a)
         print(type(a))

None
<class 'NoneType'>
```

Booléens

Les booléens ne peuvent avoir que deux valeurs :

`True, False`

On peut utiliser la fonction `bool()` pour construire un booléen.
Dans un contexte booléen, toutes ces valeurs sont équivalentes à `False` :

- `None`
- le zero des types numériques, par exemple : `0, 0.0, 0j`.
- les séquences vides, par exemple : `'', (), []`.
- les dictionnaires vides, par exemple, `{}`.

Tout le reste est équivalent a `True` :

- une valeur numérique différente de zéro : `2, 3.14, 0.5j`
- une séquence non vide : `'abc', (1, 0.5, 'toto'), [None]`
- un dictionnaire non vide : `{'ane': True, 'chat': True}`

Quelques exemples de constructions de valeurs booléennes à partir d'autres types

```
In [11]: print(bool(None), bool())
         print(bool(0), bool(1))
         print(bool(0.0), bool(0.5))
         print(bool(0j), bool(3j))
         print(bool(''), bool('abc'))
         print(bool(()), bool((1, 0.5, 'toto')))
         print(bool([]), bool([None]))
         print(bool({}), bool({'ane': True, 'chat': True}))
```

```
False False
False True
False True
False True
False True
False True
False True
False True
```

Exemple d'utilisation d'un contexte booléen

```
In [12]: Am_I_OK = True
         if Am_I_OK:
             print('OK')
         else:
             print('KO')
         print(Am_I_OK)
         print(type(Am_I_OK))
```

```
OK
True
<class 'bool'>
```

> **Exercice :** 1. Supposons que vous soyez malade, mettez `Am_I_OK` à la valeur `False` puis réexecutez la cellule. 2. Essayez avec d'autres types : listes, `None`, nombres, etc...

Numériques

entiers (précision illimitée)

```
In [13]: # Il n'y a plus de distinction entre les entiers "courts" et les entiers "
         entier = 4
         print(entier)
         print(type(entier))
         # Ce nombre nécéssite 131 bits
         entier = 14159265358979323846264338327950288419 71
         print(entier)
         print(type(entier))
```

```
4
<class 'int'>
14159265358979323846264338327950288419 71
<class 'int'>
```

> **Note :** En python version < 3.0, il existait deux types distincts `int` et `long`...

On peut utiliser la fonction interne `int()` pour créer des nombres entiers. Elle peut créer des entiers a partir de leur représentation sous forme de chaîne de caractères. On peut aussi spécifier la base.

```
In [14]: entier = int(12)
         print(entier)
         print(type(entier))
         entier = int('13')
         print(entier)
         print(type(entier))
         entier = int('0xFF', 16)
         print(entier)
         print(type(entier))
```

```
12
<class 'int'>
13
<class 'int'>
255
<class 'int'>
```

Flottants (réels 64 bits)

Pour créer des nombres réels (en virgule flottante), on peut utiliser la fonction interne `float()`.

La précision est limitée à la 16ème décimale.

```
In [15]: pi_approx = 3.1415926535897932
         print(pi_approx)
         print(type(pi_approx))
         print('{:.16f}'.format(pi_approx))
         tropgros = float('Infinity')
         print(type(tropgros))
         print(tropgros)
```

```
3.141592653589793
<class 'float'>
3.1415926535897931
<class 'float'>
inf
```

Attention ! Python autorise un affichage plus long que la précision des flottants mais tous les chiffres après le 16ème chiffre significatif sont faux :

```
In [16]: # On demande 20 chiffres après la virgule, alors que 16 seulement sont exa
         print('{:.20f}'.format(pi_approx))
```

```
3.14159265358979311600
```

Attention à ne pas effectuer des opérations interdites...

```
In [17]: # Cette cellule génère une erreur
         print(3.14 / 0)
```

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

ZeroDivisionError                         Traceback (most recent call last)

<ipython-input-17-db1124d4427a> in <module>()
      1 # Cette cellule génère une erreur
----> 2 print(3.14 / 0)


ZeroDivisionError: float division by zero
```

```
In [18]: # Cette cellule génère une erreur
         print(34 % 0)
```

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

ZeroDivisionError                         Traceback (most recent call last)

<ipython-input-18-605ca9502a37> in <module>()
      1 # Cette cellule génère une erreur
----> 2 print(34 % 0)


ZeroDivisionError: integer division or modulo by zero
```

```
In [19]: # Cette cellule génère une erreur
         print(27 // 0.0)
```

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

ZeroDivisionError                         Traceback (most recent call last)

<ipython-input-19-58586591e7ef> in <module>()
      1 # Cette cellule génère une erreur
----> 2 print(27 // 0.0)


ZeroDivisionError: float divmod()
```

```
In [20]: # Cette cellule génère une erreur
         print(1 + None)
```

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

<ipython-input-20-e749d4639af0> in <module>()
  1 # Cette cellule génère une erreur
----> 2 print(1 + None)


TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

Complexes

Les nombres complexes, a deux dimensions, peuvent être créés en utilisant le suffixe `j` à la fin d'un entier ou réel ou en utilisant la fonction interne: `complex()`

```
In [21]: complexe = 1 + 2j
         print('représentation :', complexe)
         print(type(complexe))
         c = .3j
         print(c)
```

```
représentation : (1+2j)
<class 'complex'>
0.3j
```

Séquences

Les séquences sont des conteneurs d'objets où les objets référencés sont ordonnés.

Python supporte nativement trois types de séquences :

- les chaînes de caractères
- les listes
- les tuples

Note : De nouveaux types de conteneurs peuvent être créés.

Chaînes de caractères

Pour le traitement de données textuelles, python utilise les chaînes de caractères.

Il existe plusieurs façons de définir une chaîne de caractères.

Pour délimiter le texte on utilise des guillemets (double-quote) ou des apostrophes (single-quote)

```
In [22]: mois1 = 'janvier'
         mois2 = "février"
         print(mois1, mois2)
```

```
janvier février
```

Les deux formes sont très utiles car elles permettent d'utiliser des guillemets ou des apostrophes dans des chaîne de caractères de manière simple.

```
In [23]: arbre = "l'olivier"
         print(arbre)
         cuisson_pates = "huit minutes (8')"
         print(cuisson_pates)
         record_100m = 'neuf secondes et 58 dixièmes (9"58)'
         print(record_100m)
```

```
l'olivier
huit minutes (8')
neuf secondes et 58 dixièmes (9"58)
```

Sinon, pour avoir une apostrophe ou un guillemet dans une chaîne de caractères, il faut le faire précéder d'un \ (backslash). Ce qui est beaucoup moins lisible, mais parfois obligatoire (par exemple une chaîne avec à la fois des guillemets et des apostrophes).

```
In [24]: arbre = 'l\'alisier'
         pates = '8\''
         record = "9\"58"
         print(arbre, pates, record)
         duel = 'guillemet: " et apostrophes: \' peuvent être utilisés dans une mêm
         print(duel)
         multi = '''Dans une "triplequoted" (voire plus loin), on peut (presque) to
         print(multi)
```

```
l'alisier 8' 9"58
guillemet: " et apostrophes: ' peuvent être utilisés dans une même chaîne...
Dans une "triplequoted" (voire plus loin), on peut (presque) tout utiliser: `"" ”,'
```

Caractères spéciaux

Il est possible de définir des chaînes de caractères qui contiennent des caractères spéciaux. Ils sont introduits par des séquences de deux caractères dont le premier est un (backslash). On l'appelle le caractère d'échappement.

- retour à la ligne : `\n`
- tabulation : `\t`
- backslash : `\\`
- un caractère unicode avec son code : `\uXXXX` (où les `XXXX` sont le code hexadécimal représentant ce caractère)

Plus d'information dans la documentation officielle

```
In [25]: print("Une belle présentation, c'est:")
         print('\t- bien ordonné')
         print('\t- aligné')
         print("\t- m\u00EAme s'il y en a plein\nEt c'est plus joli.")

Une belle présentation, c'est:
        - bien ordonné
        - aligné
        - même s'il y en a plein
Et c'est plus joli.
```

Chaînes multilignes

Pour écrire plusieurs lignes d'une façon plus lisible, il existe les chaînes multilignes :

```
In [26]: # L'équivalent est :
         print('''\
         Une belle présentation, c'est:
         \t- bien ordonné
         \t- aligné
         \t- même s'il y en a plein
         Et c'est plus joli.''')

Une belle présentation, c'est:
        - bien ordonné
        - aligné
        - même s'il y en a plein
Et c'est plus joli.
```

Exercice : enlevez le caractère de la 2ème ligne dans la cellule ci-dessus, et comparez le résultat à celui de la cellule de code précédente (3 cellules plus haut)

Les deux formes de délimiteurs sont aussi utilisables : guillemets triples ou apostrophes triples.

```
In [27]: multi1 = '''m
         a
         r
         s est un mois "multiligne"'''
         print(multi1, '\n')

         multi2 = """a
         v
         r
         i
         l l'est aussi"""
         print(multi2)
```

```
m
a
r
s est un mois "multiligne"

a
v
r
i
l l'est aussi
```

Unicode

En python 3 , toutes les chaînes de caractères sont unicode, et permettent d'utiliser des alphabets différents, des caractères accentués ou des pictogrammes, etc.

Pour une liste de caractères unicode, voir ici.

Chaînes spéciales

Il existe d'autres manières plus spécialisées de définir des chaînes de caractères.

Premièrement, des chaînes dans lesquelles les séquences d'échappement ne sont pas remplacées (raw strings = chaînes brutes): `r'...'`, `r"'...'"`, etc.

```
In [29]: normal_str = "chaîne normale: \n'est pas un retour à la ligne, \t pas une
         print(normal_str)

         raw_str = r"chaîne RAW: \n'est pas un retour à la ligne, \t pas une tabula
         print(raw_str)
```

```
chaîne normale:
'est pas un retour à la ligne,          pas une tabulation
chaîne RAW: \n'est pas un retour à la ligne, \t pas une tabulation
```

Plusieurs chaînes de caractères contigües sont rassemblées (concaténées)

```
In [30]: b = 'a' 'z' 'e' 'r' 't' 'y'
         print(b)
```

```
azerty
```

On peut mélanger les genres.

```
In [31]: a = 'une chaine ' u"qui est " r'''la somme de ''' """plusieurs morceaux...
         print(a)
```

```
une chaine qui est la somme de plusieurs morceaux...
```

On peut utiliser la fonction `str()` pour créer une chaîne de caractère à partir d'autres objets.

```
In [32]: a = 23
         ch_a = str(a)
         print(type(a), a)
         print(type(ch_a), repr(ch_a))
         a = 3.14
         ch_a = str(a)
         print(type(a), a)
         print(type(ch_a), repr(ch_a))
```

```
<class 'int'> 23
<class 'str'> '23'
<class 'float'> 3.14
<class 'str'> '3.14'
```

On ne peut pas mélanger les guillemets et les apostrophes pour délimiter une chaîne de caractères.

```
In [33]: # Cette cellule génère une erreur
         a = "azerty'
```

```
          File "<ipython-input-33-7cb25c6d338d>", line 2
        a = "azerty'
                    ^
    SyntaxError: EOL while scanning string literal
```

```
In [34]: # Cette cellule génère une erreur
         a = 'azerty"


          File "<ipython-input-34-28f8e986cea7>", line 2
        a = 'azerty"
                    ^
    SyntaxError: EOL while scanning string literal
```

Exercice : corrigez les deux cellules ci dessus.

Formatage

On peut formater du texte, c'est à dire utiliser une chaîne de caractères qui va servir de modèle pour en fabriquer d'autres. Il y a plusieurs manières de faire cela.

Méthode format() On utilise la méthode `format()` d'une chaîne de caractères qui va remplacer les occurences de `'{}'` par des valeurs qu'on lui spécifie en paramètre. Le type des valeurs passées n'est pas important, une représentation sous forme de chaîne de caractère sera automatiquement créée, avec la fonction `str()`.

```
'Bonjour {} !'.format('le monde')
```

```
In [35]: variable_1 = 27
         variable_2 = 'vingt huit'
         ch_modele = 'Une chaine qui contient un {} ou de multiples ici {} et là {}
         ch_modele.format('toto', variable_1, variable_2)

Out[35]: 'Une chaine qui contient un toto ou de multiples ici 27 et là vingt huit'
```

Opérateur % On peut aussi formater du texte avec l'opérateur `%`. Il va analyser la chaîne à la recherche de caractères '%' et remplacer les marqueurs par le contenu de variables.

Les marqueurs sont donc l'association d'un caractère '%' avec une sequence de caractères décrivant le type de la donnée à intégrer et la manière de formater sa représentation.

- %d pour des entiers
- %s pour des chaînes de caractères
- %f pour des nombres flottants
- %x pour un nombre entier affiché en base hexadécimale
- %o pour un nombre entier affiché en base octale
- %e pour un nombre affiché en notation exponentielle

```
In [36]: import math
         a = 27
         print('un nombre : %d, une chaîne : %s, un flottant avec une précision spé
         print('%x, %x, %x, %x' % (254, 255, 256, 257))
         print('%o, %o, %o, %o' % (254, 255, 256, 257))
         print('%e' % 2**64)
```

```
un nombre : 27, une chaîne : canal+, un flottant avec une précision spécifiée : 3.1
fe, ff, 100, 101
376, 377, 400, 401
1.844674e+19
```

Attention: Si le type de la donnée passée ne correspond pas à la séquence de formatage (%) python va remonter une erreur.

```
In [37]: # Cette cellule génère une erreur
         variable_3 = 'une chaine de caracteres'
         # Exemple d'erreur de type: il faut un entier, on a une chaîne de caractèr
         print('on veut un entier : %d' % variable_3)


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-37-6bb6742dd867> in <module>()
          2 variable_3 = 'une chaine de caracteres'
          3 # Exemple d'erreur de type: il faut un entier, on a une chaîne de carac
    ----> 4 print('on veut un entier : %d' % variable_3)


        TypeError: %d format: a number is required, not str
```

On peut se servir de cette fonctionnalité pour tabuler du texte de taille variable.

```
In [38]: ani_mots = ('ane', 'becasse', 'chat', 'dinde', 'elephant')
         print('''
         Alignons a droite les animaux:
         Un animal : %08s.
         Un animal : %08s.
         Un animal : %18s, qui se croit plus malin que les autres.
         Un animal : %08s.
         Un animal : %08s.''' % ani_mots)
```

```
Alignons a droite les animaux:
Un animal :      ane.
Un animal :  becasse.
Un animal :                 chat, qui se croit plus malin que les autres.
Un animal :    dinde.
Un animal : elephant.
```

Exercice : remettez le chat à sa place.

Pour plus d'informations sur le formatage de chaînes de caractères, voir ici et pour la méthode `.format()` qui tend à remplacer l'utilisation de l'opérateur `%`, voir là.

Listes

Une liste est un objet pouvant contenir d'autres objets. Ces objets sont placés à l'intérieur de façon séquentielle, les uns à la suite des autres. C'est un conteneur dynamique qui peut accueillir de nouveaux éléments ou pouvant rétrécir si on en supprimme.

On crée un liste en délimitant par des crochets `[]` les éléments qui le composent :

```
In [39]: L = ['egg', 'spam', 'spam', 'spam', 'bacon']
         print(L, 'est de type', type(L))

['egg', 'spam', 'spam', 'spam', 'bacon'] est de type <class 'list'>
```

Une liste peut contenir n'importe quel type d'objets.

```
In [40]: L0 = [1, 2]
         L2 = [None, True, False, 0, 1, 2**64, 3.14, '', 0+1j, 'abc']
         L1 = []
         L3 = [[1, 'azerty'], L0]
         print(L0, L1)
         print(L2)
         print(L3)

[1, 2] []
[None, True, False, 0, 1, 18446744073709551616, 3.14, '', 1j, 'abc']
[[1, 'azerty'], [1, 2]]
```

On peut utiliser la fonction `list()` pour créer une liste a partir d'autres séquences ou objets.

```
In [41]: a = list()
         b = list('bzzzzzt')
         print(a)
         print(b)

[]
['b', 'z', 'z', 'z', 'z', 'z', 't']
```

On accède aux éléments d'une liste grâce à un indice. Le premier élément est **indicé 0.**

```
In [42]: print(L[0])
         print(L[4])

egg
bacon
```

Un dépassement d'indice produit une erreur :

```
In [43]: # Cette cellule génère une erreur
         print(L[10])
```

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

<ipython-input-43-b524a28c27dd> in <module>()
  1 # Cette cellule génère une erreur
----> 2 print(L[10])


IndexError: list index out of range
```

Les listes sont dites *mutables* : je peux modifier la séquence de ces éléments.
Je remplace le deuxième élément :

```
In [44]: L[1] = 'tomatoes'
         print(L)
         L[3] = 9
         print(L)
```

```
['egg', 'tomatoes', 'spam', 'spam', 'bacon']
['egg', 'tomatoes', 'spam', 9, 'bacon']
```

Méthodes associées aux listes

Méthodes ne modifiant pas la liste

- La longueur d'une liste est donnée par fonction `len()`
- `L.index(elem)`: renvoie l'indice de l'élément `elem` (le 1er rencontré)

Méthodes modifiant la liste

- `L.append()` : ajouter un élément à la fin
- `L.pop()` : renvoie le dernier élément et le retire de la liste
- `L.sort()` : trier
- `L.reverse()` : inverser

Plus d'infos ici.

```
In [45]: L = ['egg', 'spam', 'spam', 'spam', 'bacon']
         print('len() renvoie:', len(L))
         L.append('spam') # Ne renvoie pas de valeur
         print('Après append():', L)
         print('pop() renvoie:', L.pop())
```

```
        print('Après pop():', L)
        L.reverse() # Ne renvoie pas de valeur
        print('Après reverse():', L)
        print('index() renvoie:', L.index('egg'))
        L.remove('spam') # Ne renvoie pas de valeur
        print('Après remove:', L)

len() renvoie: 5
Après append(): ['egg', 'spam', 'spam', 'spam', 'bacon', 'spam']
pop() renvoie: spam
Après pop(): ['egg', 'spam', 'spam', 'spam', 'bacon']
Après reverse(): ['bacon', 'spam', 'spam', 'spam', 'egg']
index() renvoie: 4
Après remove: ['bacon', 'spam', 'spam', 'egg']
```

Pour obtenir la liste des méthodes associées aux listes, on peut utiliser la fonction interne `help()` :

```
In [46]: help(list) # ou aussi help([])

Help on class list in module builtins:

class list(object)
 |  list() -> new empty list
 |  list(iterable) -> new list initialized from iterable's items
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
```

```
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __iadd__(self, value, /)
|      Implement self+=value.
|
|  __imul__(self, value, /)
|      Implement self*=value.
|
|  __init__(self, /, *args, **kwargs)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mul__(self, value, /)
|      Return self*value.n
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __reversed__(...)
|      L.__reversed__() -- return a reverse iterator over the list
|
|  __rmul__(self, value, /)
|      Return self*value.
|
|  __setitem__(self, key, value, /)
|      Set self[key] to value.
|
|  __sizeof__(...)
|      L.__sizeof__() -- size of L in memory, in bytes
```

```
 |
 |  append(...)
 |      L.append(object) -> None -- append object to end
 |
 |  clear(...)
 |      L.clear() -> None -- remove all items from L
 |
 |  copy(...)
 |      L.copy() -> list -- a shallow copy of L
 |
 |  count(...)
 |      L.count(value) -> integer -- return number of occurrences of value
 |
 |  extend(...)
 |      L.extend(iterable) -> None -- extend list by appending elements from the it
 |
 |  index(...)
 |      L.index(value, [start, [stop]]) -> integer -- return first index of value.
 |      Raises ValueError if the value is not present.
 |
 |  insert(...)
 |      L.insert(index, object) -- insert object before index
 |
 |  pop(...)
 |      L.pop([index]) -> item -- remove and return item at index (default last).
 |      Raises IndexError if list is empty or index is out of range.
 |
 |  remove(...)
 |      L.remove(value) -> None -- remove first occurrence of value.
 |      Raises ValueError if the value is not present.
 |
 |  reverse(...)
 |      L.reverse() -- reverse *IN PLACE*
 |
 |  sort(...)
 |      L.sort(key=None, reverse=False) -> None -- stable sort *IN PLACE*
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __hash__ = None
```

On peut créer facilement des listes répétitives grâce a l'opération de multiplication.

```
In [47]: a = ['a', 1] * 5
         print(a)
```

```
['a', 1, 'a', 1, 'a', 1, 'a', 1, 'a', 1]
```

Mais on ne peut pas 'diviser' une liste.

```
In [48]: # Cette cellule génère une erreur
         a = [1, 2, 3, 4]
         print(a / 2)
```

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

TypeError                                 Traceback (most recent call last)

<ipython-input-48-046a8a1147ab> in <module>()
  1 # Cette cellule génère une erreur
  2 a = [1, 2, 3, 4]
----> 3 print(a / 2)


TypeError: unsupported operand type(s) for /: 'list' and 'int'
```

Exercice : Manipulez la liste `L` ci-dessous avec les méthodes associées aux listes.

```
In [49]: L = ['egg', 'spam', 'spam', 'spam', 'bacon']
         # Votre code ci-dessous
```

Vous trouverez la documentation complète sur les listes ici.

Tuples

Les Tuples (ou n-uplets en Français) sont des séquences non mutables. On ne peut plus les modifier après leur création.

On les initialise ainsi :

```
In [50]: T = ('a', 'b', 'c')
         print(T, 'est de type', type(T))
         T = 'a', 'b', 'c'  # une autre façon, en ommettant les parenthèses
         print(T, 'est de type', type(T))
         T = tuple(['a', 'b', 'c'])  # à partir d'une liste
         print(T, 'est de type', type(T))
         T = ('a')  # ceci n'est pas un tuple
         print(T, 'est de type', type(T))
         T = ('a',)  # Syntaxe pour initialiser un tuple contenant un seul élément
         print(T, 'est de type', type(T))
         T = 'a',  # Syntaxe alternative pour initialiser un tuple contenant un seu
         print(T, 'est de type', type(T))
```

```
('a', 'b', 'c') est de type <class 'tuple'>
('a', 'b', 'c') est de type <class 'tuple'>
('a', 'b', 'c') est de type <class 'tuple'>
a est de type <class 'str'>
('a',) est de type <class 'tuple'>
('a',) est de type <class 'tuple'>
```

Un tuple est dit *non-mutable* : on ne peut pas en modifier la séquence.

```
In [51]: T = ('a', 'b', 'c')
         print(T[1]) # On peut utiliser un élément
```

```
b
```

```
In [52]: # Cette cellule génère une erreur
         T[1] = 'z'  # mais pas le modifier
```

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

<ipython-input-52-c83551bd9547> in <module>()
      1 # Cette cellule génère une erreur
----> 2 T[1] = 'z'  # mais pas le modifier


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

Intérêt des tuples par rapport aux listes : - plus rapide à parcourir que les listes - non mutables donc 'protégés' - peuvent être utilisés comme clé de dictionnaires (cf. plus loin)

On peut créer des tuples a partir d'autres séquences ou objets grâce à la fonction `tuple()`.

```
In [53]: a = [1, 2, 3, 'toto']
         print(type(a), a)
         b = tuple(a)
         print(type(b), b)
         a = 'azerty'
         print(type(a), a)
         b = tuple(a)
         print(type(b), b)
```

```
<class 'list'> [1, 2, 3, 'toto']
<class 'tuple'> (1, 2, 3, 'toto')
<class 'str'> azerty
<class 'tuple'> ('a', 'z', 'e', 'r', 't', 'y')
```

Manipulation des tuples

Construire d'autres tuples par concaténation et multiplication

```
In [54]: T1 = 'a', 'b', 'c'
         print('T1 =', T1)
         T2 = 'd', 'e'
         print('T2 =', T2)
         print('T1 + T2 =', T1 + T2)
         print('T2 * 3 =', T2 * 3)

T1 = ('a', 'b', 'c')
T2 = ('d', 'e')
T1 + T2 = ('a', 'b', 'c', 'd', 'e')
T2 * 3 = ('d', 'e', 'd', 'e', 'd', 'e')
```

Note :

Etant *non-mutable* l'objet tuple ne peut pas être modifié. Toutefois, s'il est constitué d'éléments mutables, alors ces éléments-là peuvent être modifiés.

Illustration avec un tuple dont un des éléments est une liste :

```
In [55]: T = ('a', ['b', 'c']) # le deuxième élément est une liste donc il est muta
         print('T =', T)
         L = T[1]
         print('L =', L)
         L[0] = 'e'
         print('L =', L)
         print('T =', T)

T = ('a', ['b', 'c'])
L = ['b', 'c']
L = ['e', 'c']
T = ('a', ['e', 'c'])
```

```
In [56]: # Ici on fait exactement la même chose...
         T = ('a', ['b', 'c'])
         print('T =', T)
         T[1][0] = 'z'
         print('T après =', T)

T = ('a', ['b', 'c'])
T après = ('a', ['z', 'c'])
```

```
In [57]: # Cette cellule génère une erreur
         T[0] = 'A'  # Ici, on essaye de modifier le tuple lui même...
```

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

TypeError                                 Traceback (most recent call last)

<ipython-input-57-f81fe1e4ec24> in <module>()
  1 # Cette cellule génère une erreur
----> 2 T[0] = 'A'  # Ici, on essaye de modifier le tuple lui même...


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

Le slicing de séquences en Python

- Cela consiste à extraire une sous-séquence à partir d'une séquence.
- Le slicing fonctionne de manière similaire aux intervalles mathématiques : `[début:fin[`
- La borne de fin ne fait pas partie de l'intervalle sélectionné.
- La syntaxe générale est `L[i:j:k]`, où :
  - `i` = indice de début
  - `j` = indice de fin, le premier élément qui n'est pas sélectionné
  - `k` = le "pas" ou intervalle (s'il est omis alors il vaut `1`)
- La sous-liste sera donc composée de tous les éléments de l'indice `i` jusqu'à l'indice `j-1`, par pas de `k`.
- La sous-liste est un nouvel objet.

Dans le sens normal (le pas `k` est positif)

- Si `i` est omis alors il vaut `0`
- Si `j` est omis alors il vaut `len(L)`

Dans le sens inverse (le pas `k` est négatif)

- Si `i` est omis alors il vaut `-1`
- Si `j` est omis alors il vaut `-len(L)-1`

Illustrons ça en créant une liste à partir d'une chaîne de caractères.
La fonction `split()` découpe une chaîne de caractères en morceaux, par défaut en 'mots'.

```
In [58]: L = 'Dans le Python tout est bon'.split()
         print(L)

['Dans', 'le', 'Python', 'tout', 'est', 'bon']
```

Pour commencer, on extrait de la liste `L` un nouvel objet liste qui contient tous les éléments de `L` <=> copie de liste

```
In [59]: print(L[0:len(L):1]) # Cette notation est inutilement lourde car :
         print(L[::])         # i = 0, j=len(L) et k=1 donc i, j et k peuvent être
         print(L[:])          # on peut même ommettre le 2ème ":"

['Dans', 'le', 'Python', 'tout', 'est', 'bon']
['Dans', 'le', 'Python', 'tout', 'est', 'bon']
['Dans', 'le', 'Python', 'tout', 'est', 'bon']
```

On extrait une sous-liste qui ne contient que les 3 premiers éléments :

```
In [60]: print(L[0:3:1])  # Notation complète
         print(L[:3:1])  # Le premier indice vaut i=0, donc on peut l'omettre
         print(L[:3])  # Le pas de slicing vaut 1, donc on peut l'omettre, ainsi qu

['Dans', 'le', 'Python']
['Dans', 'le', 'Python']
['Dans', 'le', 'Python']
```

J'extrais une sous-liste qui exclut les trois premiers éléments :

```
In [61]: print(L[3:len(L):1])  # Cette notation est inutilement lourde car :
         print(L[3:])          # j et k peuvent être omis, ainsi que le ":"

['tout', 'est', 'bon']
['tout', 'est', 'bon']
```

Les indices peuvent être négatifs, ce qui permet traiter les derniers éléments :

```
In [62]: # Je veux exclure le dernier élément :
         print(L[0:-1:1])  # Notation complète
         print(L[:-1:1])  # Le premier indice vaut i=0, donc on peut l'omettre
         print(L[:-1])  # Le pas de slicing vaut 1, donc on peut l'omettre

['Dans', 'le', 'Python', 'tout', 'est']
['Dans', 'le', 'Python', 'tout', 'est']
['Dans', 'le', 'Python', 'tout', 'est']
```

On ne garde que les deux derniers éléments

```
In [63]: print(L[-2:])

['est', 'bon']
```

Note :

`L[1]` n'est pas équivalent à `L[1:2]`, ni à `L[1:]`, ni à `L[:1]`.

Illustration :

```
In [64]: a = L[1]
         print(type(a), a) # Je récupère le deuxième élément de la liste
         a = L[1:2]
         print(type(a), a) # Je récupère une liste composée du seul élément L[1]
         a = L[1:]
         print(type(a), a) # Je récupère une liste
         a = L[:1]
         print(type(a), a) # Je récupère une liste
```

```
<class 'str'> le
<class 'list'> ['le']
<class 'list'> ['le', 'Python', 'tout', 'est', 'bon']
<class 'list'> ['Dans']
```

> **Exercice :** Retourner une liste composée des éléments de `L` *en ordre inverse* **avec une opération de slicing**. Toute utilisation de `[].reverse()` ou `reversed()` est interdite.

```
In [65]: L = 'Dans le Python tout est bon'.split()
         print(L)
         # <- votre code ici
```

```
['Dans', 'le', 'Python', 'tout', 'est', 'bon']
```

Solution : exos/reverse.py

Le slicing peut être utilisé pour **modifier** une séquence **mutable**, gâce à l'opération d'assignation.

```
In [66]: L = 'Dans le Python tout est bon'.split()
         print(L)
         L[2:4] = ['nouvelles', 'valeurs', 'et encore plus...', 1, 2, 3]
         print(L)
```

```
['Dans', 'le', 'Python', 'tout', 'est', 'bon']
['Dans', 'le', 'nouvelles', 'valeurs', 'et encore plus...', 1, 2, 3, 'est', 'bon']
```

Le slicing peut être utilisé sur des chaînes de caractères.

```
In [67]: alphabet = 'abcdefghijklmnopqrstuvwxyz'
         print(alphabet[:4], alphabet[-4:], alphabet[0:12:3])
```

```
abcd wxyz adgj
```

Exercice: 1. Découpez l'alphabet en deux parties égales 2. Prenez une lettre sur deux

```
In [68]: # Votre code ici
```

Solution : exos/alphabet.py

Chaînes de caractères, le retour

Les chaînes de caractères sont considérées comme des séquences non mutables et l'on peut les manipuler comme telles.

```
In [69]: # Cette cellule génère une erreur
         non_mutable = 'abcdefgh'
         non_mutable[3] = 'D'
```

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

<ipython-input-69-7c3090bf77af> in <module>()
  1 # Cette cellule génère une erreur
  2 non_mutable = 'abcdefgh'
----> 3 non_mutable[3] = 'D'


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

Il faut construire une nouvelle chaîne de caractère. En concaténant des morceaux (slices) de la chaîne originale:

```
In [70]: nouvelle_chaine = non_mutable[:3] + 'D' + non_mutable[4:]
         print(nouvelle_chaine)
```

```
abcDefgh
```

Ou alors en utilisant une transformation en liste, puis a nouveau en chaîne:

```
In [71]: a = list(non_mutable)
         a[3] = 'D'
         print(''.join(a))
```

```
abcDefgh
```

On peut savoir si une chaîne se trouve dans une autre

```
In [72]: print('123' in 'azerty_123_uiop')
         print('AZE' in 'azerty_123_uiop')
         print('aze' in 'azerty_123_uiop')
```

```
True
False
True
```

La longueur d'une chaîne s'obtient avec `len()`.

```
In [73]: print(len(non_mutable))
```

```
8
```

Exercice, dans la cellule ci-dessous: 1. **Insérez** le caractère '#' au milieu de la chaîne donnée. 2. Idem mais **coupez** la chaine en 3 parties, et insérez le caractère '@' entre chacune d'elles. 3. **Insérez** le caractère '|' entre chaque caractère de la chaîne.

```
In [74]: chaine_donnee = 'azertyuioppoiuytreza'
         # Votre code ici
```

Solution : exos/chaine_donnee.py

Les dictionnaires

Les listes associatives sont des conteneurs où les objets ne sont **pas** ordonnés ni accessibles par un indice mais sont associés à une clé d'accès.

L'accès aux éléments se fait comme pour les listes ou tuples, avec les `[]`, mais vous pouvez utiliser (presque) n'importe quelle valeur comme clé.

```
dico = {<clé1>: <valeur1>[, <clé2>: <valeur2>]...}
```

Les clés doivent être des objets non mutables. Les listes ne peuvent pas servir de clés alors que les tuples le peuvent.

Dans `dico`, on accède à `valeur1` avec la syntaxe `dico[clé1]`.

Un exemple

```
In [75]: dic_animaux = {'ane': True, 'arbre': False, 'chat': True, 'lune': False, '
         cle = 'chat'
         valeur = dic_animaux[cle]
         print(valeur)
         print('{} est un animal: {}'.format(cle, valeur))
         # Ou encore
         print('{} est un animal: {}'.format('chat', dic_animaux['chat']))

True
chat est un animal: True
chat est un animal: True
```

Exercice : essayez de savoir si un arbre est un animal

Solution : exos/animarbre.py

Les différentes manières de créer des dictionnaires:

```
In [76]: a = {'un': 1, 'deux': 2, 'trois': 3}  # Les accolades comme syntaxe
         b = dict(un=1, deux=2, trois=3)  # La méthode dict()
         c = dict(zip(['un', 'deux', 'trois'], [1, 2, 3]))  # On "zippe" deux liste
         d = dict([('deux', 2), ('un', 1), ('trois', 3)])  # On transforme une list
         e = dict({'trois': 3, 'un': 1, 'deux': 2})
         a == b == c == d == e

Out[76]: True
```

La fonction interne [`dict()` permet de créer des dictionnaires de diverses manières.

Accéder à un élément qui n'est pas dans le dictionnaire génère une erreur. Il existe la méthode `{}.get()` ou l'opérateur `in` qui permettent de ne pas rencontrer ce problème.

```
In [77]: # Cette cellule génère une erreur
         err = a['quatre']
```

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

KeyError                                  Traceback (most recent call last)

<ipython-input-77-a2ea8ffbfaf9> in <module>()
      1 # Cette cellule génère une erreur
----> 2 err = a['quatre']


KeyError: 'quatre'
```

Ici on utilise `.get()` et cela ne remonte pas d'erreur.

```
In [78]: print(a.get('quatre'))     # La valeur par défaut est 'None' lorsque .get()
         print(a.get('quatre', 5)) # On peut spécifier une autre valeur par défaut
```

```
None
5
```

Un autre exemple

```
In [79]: tup_mois = ('jan', 'feb', 'mar', 'apr', 'mai', 'jun', 'jul', 'aug', 'sep',
         tup_long = (31, (28, 29), 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
         dic_mois = dict(zip(tup_mois, tup_long))
         mois_naissance = 'jun'
         print('Il y a %d jours dans votre mois de naissance' % dic_mois[mois_naiss
```

```
Il y a 30 jours dans votre mois de naissance
```

Exercice:

1. Modifiez la cellule précédente pour y changer mois_naissance (les 3 premiers caractères de votre mois de naissance, en anglais). Ré-exécutez la cellule et vérifiez la réponse.
2. Un problème s'est glissé dans la cellule, lequel ?
3. Corrigez-le. (il y a plusieur manières de faire)
4. Les noms de mois raccourcis sont en anglais et non en français, pourquoi ?
5. Utilisez `input('En quel mois êtes-vous né(e) ? ')` pour initialiser `mois_naissance` et constatez le gain de qualité en IHM (Interface Homme Machine, ergonomie).

Solution : exos/dict_anniv.py
Les dictionnaires sont mutables.

```
In [80]: ages = {'albert': 62, 'bob': 34, 'charlie': 1, 'daphne': 67}
         print(ages)
         # C'est l'anniversaire de charlie, il a un an de plus...
         ages['charlie'] += 1
         print(ages)
         print('Albert a %d ans.' % ages['albert'])
         # Bob est parti, enlevons-le
         del ages['bob']
         print(ages)

{'bob': 34, 'charlie': 1, 'albert': 62, 'daphne': 67}
{'bob': 34, 'charlie': 2, 'albert': 62, 'daphne': 67}
Albert a 62 ans.
{'charlie': 2, 'albert': 62, 'daphne': 67}
```

- Savoir si une clé est présente dans un dictionnaire est une opération rapide. On utilise, comme pour les séquences, l'opérateur `in`.
- La fonction interne `len()` est utilisable pour savoir combien d'objets sont référencés dans le dictionnaire.

```
In [81]: ages = {'albert': 62, 'bob': 34, 'charlie': 1, 'daphne': 67}
         print('Charlie est dedans ?', 'charlie' in ages)
         print('Zoé est dedans ?', 'zoé' in ages)
         print('Bob est dedans ?', 'bob' in ages)
         print('Il y a %d personnes.' % len(ages))
         # Bob est parti, enlevons-le
         del ages['bob']
         print('Bob est dedans ?', 'bob' in ages)
         print('Il y a %d personnes.' % len(ages))

Charlie est dedans ? True
Zoé est dedans ? False
Bob est dedans ? True
Il y a 4 personnes.
Bob est dedans ? False
Il y a 3 personnes.
```

On peut itérer sur les clés ou les objets référencés, ou vider un dictionnaire, en comparer deux, etc.

Pour plus d'informations sur les dictionnaires, voir ici.

Exercice :

1. Créez un dictionnaire qui va traduire des chiffres (de 0 à 9) écrits en toutes lettres entre deux langues Par exemple : `trad_num['un'] -> 'one'`
2. Modifiez ce dictionnaire, pour qu'il fonctionne dans les deux sens de traduction (Fr -> En et En -> Fr)

3. Modifiez ce dictionnaire, pour qu'il fonctionne aussi avec les chiffres sous forme d'entiers Par exemple : `trad_num[1] -> 'un'`

```
In [82]: # Votre code ici
```

Solution : exos/trad_num.py

Les ensembles

Les ensembles sont des conteneurs qui n'autorisent pas de duplication d'objets référencés, contrairement aux listes et tuples.

On peut créer des ensembles de cette façon :

```
ensemble = set(<iterable>)
```

Où `<iterable>` peut être n'importe quel objet qui supporte l'itération : liste, tuple, dictionnaire, un autre set (pour en faire une copie), vos propres objets itérables, etc...

Tout comme pour les dictionnaires, l'opérateur `in` est efficace. La fonction `set()` permet de créer des ensembles.

```
In [83]: list1 = [1, 1, 2]
         tupl1 = ('un', 'un', 'deux', 1, 3)

         b = set(list1)
         a = set(tupl1)

         print(a, b)

         print('''La chaîne 'un' est elle dans l'ensemble ?''', 'un' in a)
         a.remove('un')
         print('''La chaîne 'un' est-elle toujours dans l'ensemble ?''', 'un' in a)

         print(a, b)
```

```
{1, 'deux', 3, 'un'} {1, 2}
La chaîne 'un' est elle dans l'ensemble ? True
La chaîne 'un' est-elle toujours dans l'ensemble ? False
{1, 'deux', 3} {1, 2}
```

Des opérations supplémentaires sont possibles sur des ensembles. Elles sont calquées sur les opérations mathématiques :

- union
- intersection
- etc...

```
In [84]: print('intersection :', a & b)
         print('union :', a | b)
```

```
intersection : {1}
union : {2, 1, 'deux', 3}
```

Pour plus d'informations sur les ensembles, voir ici

Fichiers

Ouverture

L'instruction :

```
f = open('interessant.txt', mode='r')
```

ouvre le fichier `interessant.txt` en mode lecture seule et le renvoie dans l'objet `f`.

On peut spécifier un chemin d'accès complet ou relatif au répertoire courant. Le caractère de séparation pour les répertoires peut être différent en fonction du système d'exploitation ('/' pour unix et " pour windows), voir le module `os.path`.

Modes d'ouverture communs

- `'r'` : lecture seule
- `'w'` : écriture seule
- `'a'` : ajout à partir de la fin du fichier

Note : Avec `'w'` et `'a'`, le fichier est créé s'il n'existe pas.

Pour plus d'informations sur les objets fichiers, voir ici, pour la documentation de la fonction `open()`, voir là.

Fermeture

On ferme le fichier `f` avec l'instruction :

```
f.close()
```

Méthodes de lecture

- `f.read()` : retourne tout le contenu de `f` sous la forme d'une chaîne de caractères.

```
In [85]: f = open('interessant.txt', mode='r')
         texte = f.read()
         print('"texte" est un objet de type', type(texte), 'de longueur', len(text
         print(texte)
         print('Contenu en raw string:')
         print(repr(texte))
         f.close()

"texte" est un objet de type <class 'str'> de longueur 74 caractères:
Si vous lisez ce texte alors
...
vous savez lire un fichier avec Python !

Contenu en raw string:
'Si vous lisez ce texte alors\n...\nvous savez lire un fichier avec Python !\n'
```

- `f.readlines()` : retourne toutes les lignes de `f` sous la forme d'une liste de chaînes de caractères.

```
In [86]: f = open('interessant.txt', mode='r')
         lignes = f.readlines()
         print('"lignes" est un objet de type', type(lignes), 'contenant', len(lign
         print(lignes)
         f.close()

"lignes" est un objet de type <class 'list'> contenant 3 éléments:
['Si vous lisez ce texte alors\n', '...\n', 'vous savez lire un fichier avec Python
```

Méthodes d'écriture

- `f.write('du texte')` : écrit la chaine `'du texte'` dans `f`

```
In [87]: chaine = 'Je sais écrire\n...\navec Python !\n'
         # mode 'w' : on écrase le contenu du fichier s'il existe
         # encoding UTF-8: on peut écrire des chaînes avec des caractères non ASCI
         f = open('pas_mal.txt', mode='w', encoding='utf8')
         f.write(chaine)
         f.close()
```

Note : du point de vue du système, rien n'est écrit dans le fichier avant l'appel de `f.close()`

- `f.writelines(ma_sequence)` : écrit la séquence `ma_sequence` dans `f` en mettant bout à bout les éléments

```
In [88]: sequence = ['Je sais ajouter\n', 'du texte\n', 'avec Python !\n']
         f = open('pas_mal.txt', mode='a')
         # mode 'a' : on ajoute à la fin du fichier
         # encoding par défaut: uniquement des caractères ASCII
         f.writelines(sequence)
         f.close()

In [89]: chaine = 'Pour ajouter du texte avec des caractères accentués: ï\n'
         f = open('pas_mal.txt', mode='a', encoding='utf8')
         # mode 'a' : on ajoute à la fin du fichier
         # encoding UTF-8: on peut écrire des chaînes avec des caractères non ASCI
         f.write(chaine)
         f.close()
```

Exercice :

1. écrire le contenu de la liste `mystere` dans le fichier `coded.txt` puis fermer ce dernier
2. lire le fichier `coded.txt` et le stocker dans une chaine `coded`
3. Décoder la chaîne `coded` avec les instructions suivantes:

```
import codecs
decoded = codecs.decode(coded, encoding='rot13')
```

4. écrire la chaine `decoded` dans le fichier `decoded.txt` et fermer ce dernier
5. visualiser le contenu du fichier `decoded.txt` dans un éditeur de texte

```
In [90]: mystere = ['Gur Mra bs Clguba, ol Gvz Crgref\n\n',
                    'Ornhgvshy vf orggre guna htyl.\n',
                    'Rkcyvpvg vf orggre guna vzcyvpvg.\n']
         # Votre code ci-dessous
```

Solution : exos/fichiers.py