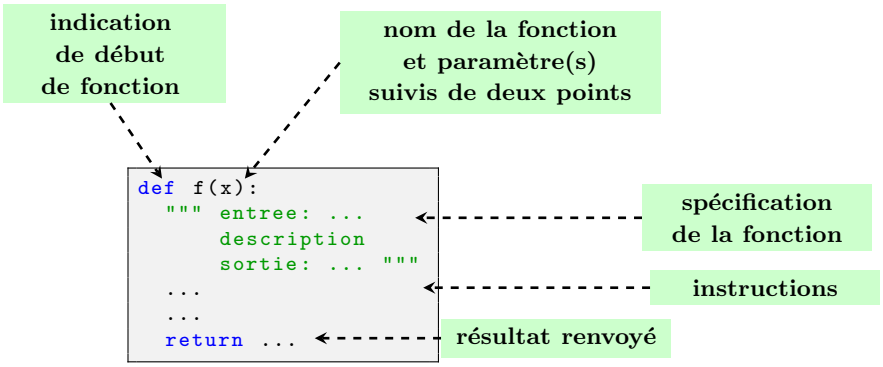
CHAPITRE 1

RAPPELS DE PYTHON – SYNTAXE ET STRUCTURES DE DONNÉES

I - Syntaxe des instructions de base

On rappelle que l'on définit une ***fonction*** de la façon suivante :



Le mot clé `return` peut apparaître plusieurs fois mais dès qu'il est «rencontré» une fois, cela met un terme à l'exécution des instructions qui suivent. Une fonction peut également agir sur les paramètres (on parle d'*effet de bord*) voire ne faire que cela et ne rien renvoyer (on dit alors souvent que cette fonction est une *procédure*).

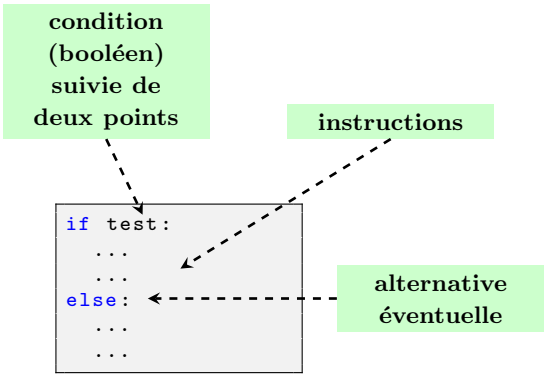
Exemple

Définissons en Python les fonctions $f : \mathbb{R} \to \mathbb{R}, x \mapsto x^2 + x - 1$ et $g : \mathbb{R} \to \mathbb{R}, x \mapsto \mathrm{e}^{x+1}$.

Exemple

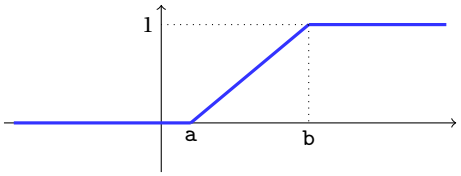
Définissons en Python une fonction qui prend en argument une chaîne de caractères et qui renvoie la valeur du dernier caractère.

Il convient de maîtriser les structures conditionnelles à l'aide de la syntaxe suivante :

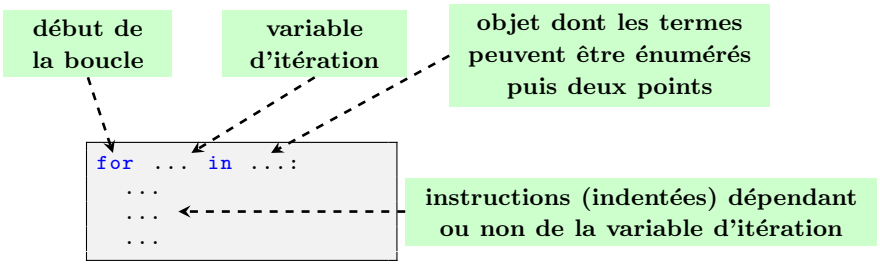


Exemple

Définissons la fonction `f` d'arguments `a`, `b` (avec `a` < `b`) et `x` et qui renvoie l'image de `x` par la fonction représentée ci-dessous :

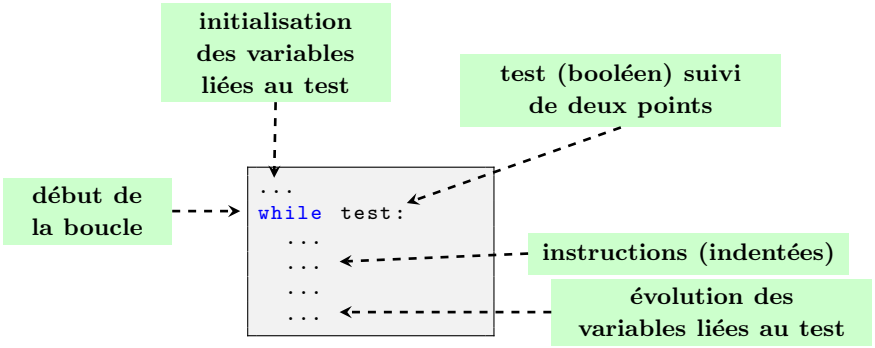


Il existe en Python deux types de structures itératives. Un premier type de ***boucle*** concerne le cas où le nombre d'itérations est déterminé et dépend de l'ensemble qui est décrit par la variable d'itération. La syntaxe générale est de la forme suivante :



L'objet dont les éléments sont énumérables peut être de la forme `range(n)` (ce qui correspond aux entiers de `0` à `n-1`), une chaîne de caractères, une liste, etc.

L'autre type de boucle a un nombre d'itérations qui n'est pas déterminé à l'avance ; la poursuite d'une telle boucle est liée à la réalisation d'une condition. La syntaxe générale est de la forme suivante :



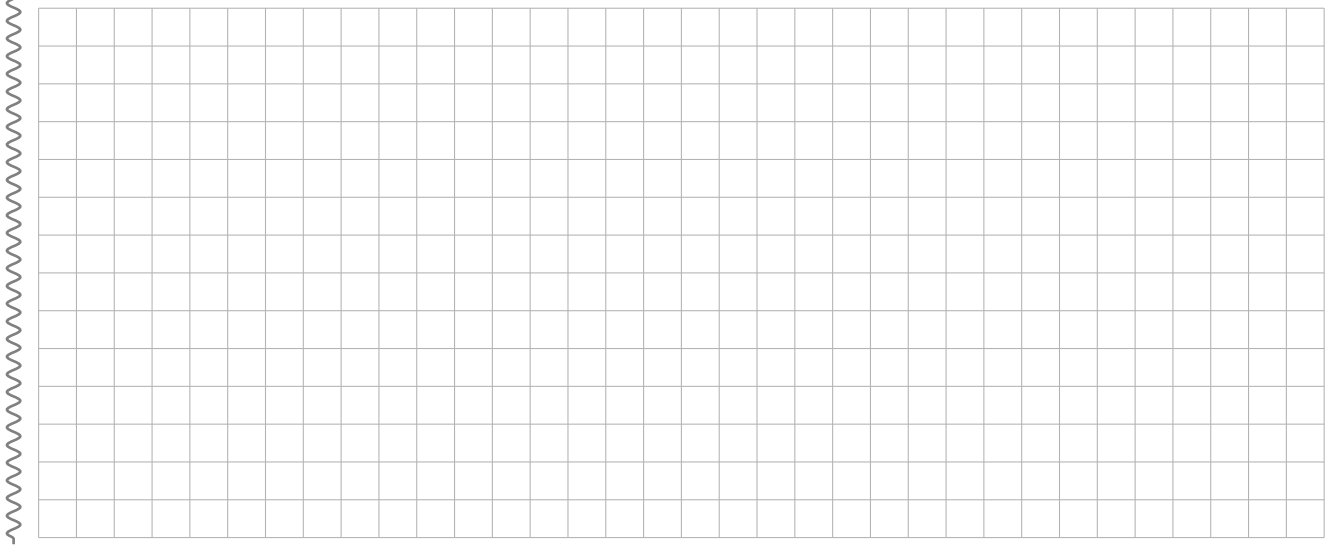
La question de la *terminaison* se pose alors (la boucle est-elle «infinie» ou non ?).

Exemple

Écrivons une fonction d'argument un entier naturel `n` et qui renvoie la somme des `n` premiers nombres impairs.

Exemple

Écrivons une fonction d'argument un entier naturel `n` et qui renvoie la plus grande puissance de 2 inférieure ou égale à `n`.



Exercice 1

1. À l'aide de la fonction `bin`, écrire une fonction `long` qui a pour argument `n` et renvoie le nombre de chiffres dans l'écriture en base 2 de `n` :

```
>>> long(13) # 13 = 8 + 4 + 1
4
>>> long(43) # 43 = 32 + 8 + 2 + 1
6
>>> long(1)
1
```

2. Définir de façon récursive la fonction `f` correspondant à la fonction $f : \mathbb{N}^* \to \mathbb{N}$ définie par $f(1) = 0$ et :

$$\forall n \in \mathbb{N},\ f(2n) = 2f(n) + 1 \text{ et } f(2n+1) = 2f(n).$$

3. On admet qu'en appliquant un certain nombre de fois la fonction f à partir d'un entier n, on obtient 0 :

$$f\left(f\left(\cdots f\left(f(n)\right)\cdots\right)\right) = 0.$$

Écrire une fonction `iter` d'argument un entiet n et qui renvoie le plus petit entier k tel que que k itérations de la fonction f à partir de n donnent 0.

```
>>> f(12)
3
>>> f(3)
0
>>> iter(12)
2
```

4. Déterminer le maximum de la fonction `iter` sur l'intervalle $[\![1, 100]\!]$.

II - Rappels sur les listes

Une liste est un ensemble ordonné (chaque élément à un numéro d'ordre) d'éléments (de types non nécessairement homogènes). Elles sont définies entre crochets et les éléments sont séparés par une virgule. Voici, au travers d'exemples, quelques façons de définir des listes :

```
>>> L = [1, 2, 3] # définition directe
>>> L = list(range(5)); print(L) # définition à partir d'un objet itérable
[0, 1, 2, 3, 4]
>>> L = list(range(4)) + [1, 5, 42]; print(L) # par concaténation de listes
[0, 1, 2, 3, 1, 5, 42]
>>> L = [0]*5; print(L) # par concaténation de copies d'une même liste
[0, 0, 0, 0, 0]
>>> L1 = [2*k+1 for k in range(5)]; print(L1) # en compréhension
[1, 3, 5, 7, 9]
>>> L2 = [k**2 for k in L1 if k>4]; print(L2) # en compréhension avec condition
[25, 49, 81]
```

On peut directement accéder à tous les éléments de la liste et les modifier :

```
>>> L=[2, 5, 42, 6, 3, -2, 4, 6, 8, 9, 12]
>>> L[2]; L[-3]
42
8
>>> L[1:5] # sous-liste de l'élément n°1 à l'élément n°5 non compris
[5, 42, 6, 3]
>>> L[1:10:2] # idem avec un pas
[5, 6, -2, 6, 9]
>>> L[2:] # de l'indice 2 jusqu'à la fin
[42, 6, 3, -2, 4, 6, 8, 9, 12]
>>> L[2::3] # idem avec un pas
[42, -2, 8]
>>> L[:8] # du début jusqu'à l'indice 8 non compris
[2, 5, 42, 6, 3, -2, 4, 6]
>>> L[:8:2] # idem avec un pas
[2, 42, 3, 4]
>>> L[:-2:3] # fonctionne également avec des n° négatifs
[2, 6, 4]
>>> L[0] = 76; L # remplacement d'une valeur
[76, 5, 42, 6, 3, -2, 4, 6, 8, 9, 12]
>>> L[1:6] = [0]; L # remplacement d'une tranche
[76, 0, 4, 6, 8, 9, 12]
```

Rappelons quelques opérations et méthodes sur les listes :

```
>>> L=[1, 6, 9, 23, 21, 0, 3]
>>> len(L) # longueur de la liste
7
>>> L.append(5); L # ajout d'une valeur en tête de liste
[1, 6, 9, 23, 21, 0, 3, 5]
>>> L.append(8); L
[1, 6, 9, 23, 21, 0, 3, 5, 8]
>>> L.insert(2, 8); L # insertion d'une valeur à un emplacement spécifié
[1, 6, 8, 9, 23, 21, 0, 3, 5, 8]
>>> L.pop() # retourne et supprime la tête de liste
8
>>> L
[1, 6, 8, 9, 23, 21, 0, 3, 5]
>>> L.pop(5) # idem avec un emplacement spécifié
21
>>> L
[1, 6, 8, 9, 23, 0, 3, 5]
```

Enfin, rappelons le «danger» (et la source d'erreurs !) résidant dans la copie d'une liste :

```
>>> from copy import copy
>>> a=[1, 2, 3, 4, 5]
>>> b = a; c = copy(a)
>>> a[0] = 42; a; b; c
[42, 2, 3, 4, 5]
[42, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

Quelques situations classiques

1. Écrire une fonction `appartient(a, L)` renvoyant un booléen indiquant si l'élément `a` appartient à la liste `L`.

2. Modifier la fonction précédente pour traiter le cas où `L` est une liste de nombres triés dans l'ordre croissant.

3. Écrire une fonction `tous(L)`, d'argument une liste `L` et renvoyant un booléen indiquant si tous les éléments de `L` sont positifs ou nuls et écrire une fonction `aumoins(L)` renvoyant un booléen indiquant si au moins un élément de `L` est positif ou nul.

4. Écrire une fonction `seuil(L, x)` renvoyant la liste formée par les éléments de la liste `L` supérieurs ou égaux à `x`.

5. Écrire une fonction `extremes(L)` renvoyant le couple formé du minimum et du maximum d'une liste `L` de nombres.

6. Écrire une fonction `indmax(L)` renvoyant l'indice de la dernière occurence du maximum d'une liste `L` de nombres.

7. Écrire une fonction `nombre(lis, x, n)`, d'argument une liste `lis`, un objet `x` et un entier `n`, qui renvoie `True` si l'objet `x` apparaît exactement `n` fois dans `lis` (et qui renvoie `False` sinon).

8. Écrire une fonction `deuxieme(lis)`, d'argument une liste d'entiers `lis` ayant au moins 2 éléments et dont tous les «éléments» sont différents, qui renvoie le deuxième plus grand entier de la liste.

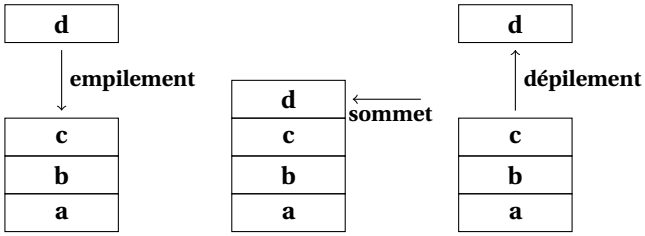
9. Écrire une fonction `distmin(lis)`, d'argument une liste de flottants `lis`, qui renvoie le plus petit écart entre deux éléments distincts de la liste.

III - Notion de pile

Une ***pile*** est une structure de données qui reprend l'idée d'une pile d'assiettes : on peut poser une assiette sur cette pile ou reprendre la dernière assiette posée. Les données vont donc être «empilées» et seule la dernière entrée est directement récupérable – l'accès aux éléments inférieurs de la pile n'est pas possible aussi rapidement. On parle de structure LIFO (*last in, first out*).
Les piles sont omniprésentes en informatique. Des exemples simples sont par exemple l'historique d'un navigateur web, le stockage des actions dans un éditeur de texte pour offrir la possibilité à l'utilisateur de revenir en arrière,...

Les fonctions usuelles de manipulation sont :

- la *création* d'une pile vide ;
- le test indiquant si l'on a une *pile vide* ;
- l'*empilement* d'un nouvel élément (en anglais, *push*) ;
- le *dépilement* de l'élément situé au sommet de la pile (en anglais, *pop*).

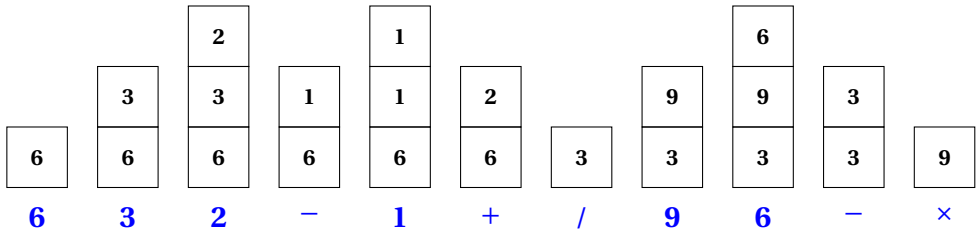


Exemple : évaluation d'une expression arithmétique

De façon usuelle, on note l'opérateur entre les deux opérandes (on écrit $2+3$ pour la somme des entiers 2 et 3) mais il existe d'autres notations. La notation polonaise inverse consiste à écrire l'opérateur après (on écrit 2 3 + pour la somme des entiers 2 et 3). L'intérêt de cette notation est qu'elle fait l'économie des parenthèses ; par exemple, au lieu d'écrire $(1+2)\times(3-4)$, on écrit 1 2 + 3 4 − ×.
Pour évaluer une expression écrite en notation polonaise inverse, on peut utiliser une pile :

- on lit de façon linéaire la succession de symboles ;
- on empile les entiers que l'on rencontre ;
- lorsque l'on rencontre un opérateur, on dépile les deux derniers entiers, on applique l'opérateur et on empile le résultat ;
- à la fin de la lecture de la succession de symboles, la pile contient exactement un entier qui est le résultat de l'évaluation de l'expression.

Illustrons cela avec l'expression 6 3 2 − 1 + / 9 6 − × :



On peut réaliser une pile en Python à l'aide du type `list`. En effet, il suffit de considérer les fonctions données ci-dessous :

```
def creerPile():
    return []

def estVide(p):
    return p == []

def empiler(p, x):
    p.append(x)
```

```
def depiler(p):
    if estVide(p):
        return "pile vide"
    else:
        return p.pop()
```

Malgré la pertinence des listes pour implémenter la notion de Pile en Python, il convient de s'exercer à n'utiliser que les quatre fonctions de base sans exploiter la structure sous-jacente.

Exemples

1 ► Écrire une fonction `sommet(p)` renvoyant le sommet d'une pile `p` sans la modifier.

2 ► Écrire une fonction qui prend une pile en argument et qui échange les deux éléments au sommet de la pile (sauf s'il y a moins de deux éléments dans la pile auquel cas elle est inchangée).

3 ► Écrire une fonction donnant la «taille» d'une pile sans la détruire.

IV - Notion de file

Une structure voisine de celle de pile est la structure de ***file*** qui reprend l'idée d'une file d'attente (sans priorité) : on entre dans la file et on attend que tous ceux arrivés avant passent. On parle de structure FIFO (*first in, first out*).
Les fonctions usuelles de manipulation sont :

- la *création* d'une file vide ;
- le test indiquant si l'on a une *file vide* ;
- l'*enfilement* d'un nouvel élément (le nouvel élément est ajouté en queue de file) ;
- le *défilement* de l'élément situé en tête de la file.

Exemple

On appelle *nombre de Hamming*, tout entier naturel n de la forme $2^a3^b5^c$ avec $(a,b,c) \in \mathbb{N}^3$; les premiers sont :

$$1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24.$$

Une méthode possible pour déterminer les premiers nombres de ce type est présentée dans le programme suivant :

```
def g(L2, L3, L5) :
    n = min(L2[0], L3[0], L5[0])
    if n == L2[0]:
        L2.pop(0)
    if n == L3[0]:
        L3.pop(0)
    if n == L5[0]:
        L5.pop(0)
    L2.append(2*n)
    L3.append(3*n)
    L5.append(5*n)
    return n

L2, L3, L5 = [1], [1], [1]
LH = []
for no in range(15) :
    LH.append(g(L2, L3, L5))
```

Dans ce qui précède, les listes `L2`, `L3` et `L5` sont utilisées comme des files qui «stockent» les doubles, les triples et les quintuples «en attente».

Comme dans l'exemple ci-dessus, les listes Python permettent d'implémenter la structure de file. En effet, il suffit de considérer les fonctions données ci-dessous :

```
def creerFile():
    return []

def estVide(f):
    return p == []

def enfiler(f, x):
    f.append(x)

def defiler(f):
    if estVide(f):
        return "file vide"
    else:
        return f.pop(0)
```

Il convient néanmoins de remarquer que la méthode `pop(0)` n'est pas de complexité anodine. On est préférable d'utiliser l'instruction `deque` du module `collection`. Il suffit par exemple de considérer les fonctions données ci-dessous :

```
from collections import deque

def creerFile():
    return deque()

def estVide(f):
    return f == deque([])
```

```
def enfiler(f, x):
    return f.append(x)

def defiler(f):
    return f.popleft()
```

Comme pour les files, il convient de s'exercer à n'utiliser que les quatre fonctions de bases sans exploiter la structure sous-jacente.

Exemple

Écrire une fonction `consultation(f)` renvoyant l'élément en tête d'une file sans modifier la file.

V - Notion de dictionnaire

Les dictionnaires sont des structures de données formées d'enregistrements repérés chacun par un élément particulier appelé *clé*, différent d'un enregistrement à l'autre, et qui doit être de type non mutable (un entier ou une chaîne de caractères par exemple, les clés peuvent être de types différents). L'enregistrement associé à une clé donnée peut, lui, être de n'importe quel type (entier, liste, tuple...) et peut être modifié (*i.e.* est mutable).
Contrairement à une liste où les données sont repérées par un indice numéroté à partir de 0, il n'y pas d'ordre prédéfini pour les différents enregistrements d'un dictionnaire, la machine organisant en interne les données pour accélérer les accès en fonction des clés. Pour l'utilisateur, un dictionnaire est une collection de données stockées «en vrac» dans laquelle il est facile d'ajouter ou de supprimer un élément.
À ce stade de l'année, on ne s'intéresse pas à la façon dont fonctionne un dictionnaire.

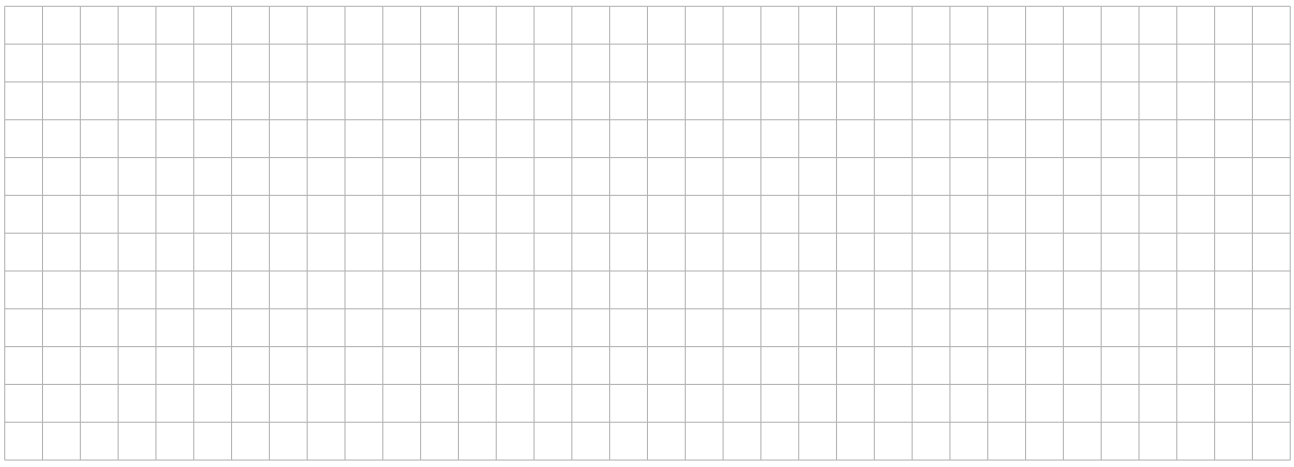
On crée un dictionnaire en mettant entre accolades `{ }` une suite d'enregistrements `clé:valeur` (et le dictionnaire vide s'obtient par {}) ou en transformant une liste `L` de listes à deux éléments à l'aide de `dict(L)`.
On accède à l'enregistrement associé à une clé `key` d'un dictionnaire `D` par `D[key]` (ce qui permet aussi éventuellement de créer l'enregistrement). Il convient de connaître également le moyen de tester si une clé apparaît dans le dictionnaire à l'aide de `in`. Enfin, il est utile de connaître les méthodes `items` et `values`.

```
>>> d1 = {'pcsi1':'DE004', 'pcsi2':'C204', 'mp2i': ['DE002','C205'], 'C011':['ECG11','ECG21']}
>>> d1['pcsi2']
'C204'
>>> 'pcsi1' in d1
True
>>> d1.items()
dict_items([('pcsi1', 'DE004'), ('pcsi2', 'C204'), ('mp2i', ['DE002', 'C205']), ('C011', ['
    ECG11', 'ECG21'])])
>>> d1.values()
dict_values(['DE004', 'C204', ['DE002', 'C205'], ['ECG11', 'ECG21']])
>>> d2 = dict([(i,chr(i)) for i in range(97,100)])
>>> print(d2)
{97: 'a', 98: 'b', 99: 'c'}
>>> d2[100] = 'd'
>>> print(d2)
{97: 'a', 98: 'b', 99: 'c', 100: 'd'}
```

Exemple

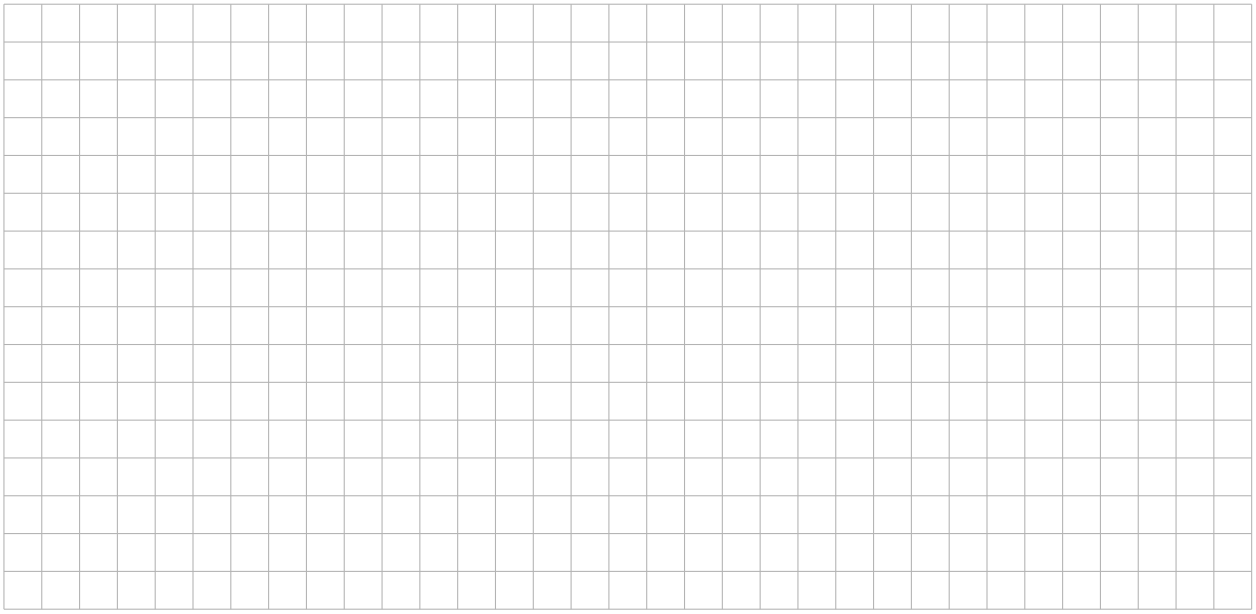
1 ► À l'aide d'un dictionnaire, écrire une fonction `compte(ch)` prenant en argument une chaîne de caractères `ch` et comptant le nombre d'occurrences de chaque caractère. Par exemple, on doit avoir :

```
>>> compte('sapristi, voici venir ici des ex-pcsi !')
{'s': 4, 'a': 1, 'p': 2, 'r': 2, 'i': 8, 't': 1, ',': 1, ' ': 6, 'v': 2, 'o': 1,
 'c': 3, 'e': 3, 'n': 1, 'd': 1, 'x': 1, '-': 1, '!': 1}
```



2 ► On considère un dictionnaire `D` représentant un dictionnaire de traduction *anglais vers français* : les clés sont des mots anglais et les valeurs sont la liste des traductions possibles des mots. Écrire le programme qui crée à partir de `D` le dictionnaire *français vers anglais* associé. On pourra tester avec :

```
>>> D = {'make' : ['faire', 'fabriquer', 'réaliser'], 'do' : ['faire'],
         'realize' : ['réaliser', 'prendre conscience', 'parvenir'],
         'produce' : ['réaliser', 'produire', 'fabriquer']}
```



Exercice 2

Écrire une fonction `trois` qui prend un argument une chaîne de caractères `ch` et qui renvoie la liste des caractères de `ch` qui apparaissent au moins trois fois.

```
>>> trois("la salle des PC est magnifiquement décorée")
['l', 'a', ' ', 's', 'e']
```

VI - Utilisation de listes ou de dictionnaires : l'exemple des graphes

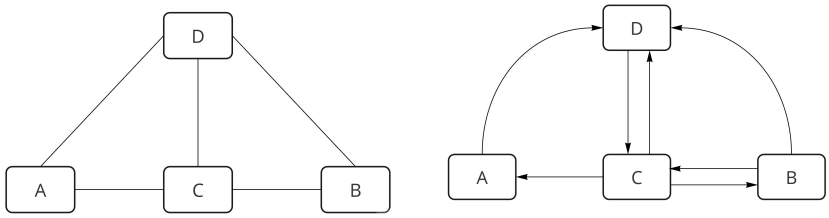
Un ***graphe*** G est un couple (S, A) où :

- S est un ensemble *fini et non vide* d'éléments appelés ***sommets*** ou ***nœuds***.
- A est un ensemble de couples ou de paires d'éléments de S appelées ***arêtes***.

Lorsque les arêtes sont des couples, on dit que le graphe est ***orienté*** et les arêtes sont appelées des ***arcs*** (et un arc ayant mêmes origine et arrivée est appelé une *boucle*).
Dans le cas contraire, on dit que le graphe est ***non orienté***.

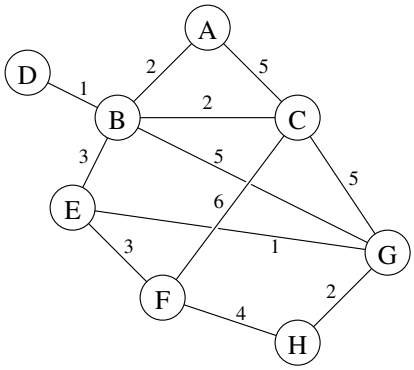
Exemple

Voici à gauche un exemple de graphe non orienté et à droite un exemple de graphe orienté.



Il peut être également utile d'ajouter des poids ou des étiquettes aux arêtes : on parle alors de graphe ***pondéré*** ou *étiqueté*.

Exemple



Il convient de maîtriser certains éléments de vocabulaire.

- Un graphe est dit *simple* lorsque, entre deux sommets, il n'y a pas plus d'une arête.
- Le nombre de sommets du graphe G s'appelle l'*ordre du graphe* et le nombre d'arêtes ou d'arcs s'appelle la *taille* du graphe G.
- Deux sommets reliées par une arête ou un arc sont dits *adjacents* ou *voisins*; ces sommets sont les *extrémités* ou les *sommets de l'arête* (dans le cas orienté, on parle d'extrémité initiale ou d'origine et d'extrémité finale ou d'arrivée de l'arc).

  Le *degré* d'un sommet x est le nombre de ses voisins. Un sommet de degré 0 est dit *isolé*.
- Un *chemin* de s_0 à s_n est une séquence $(s_0, s_1, \ldots, s_n)$ de sommets tels que deux consécutifs soient adjacents; ce chemin permet de relier s_0 à s_n, s_0 est le sommet de départ, s_n est le sommet d'arrivée; on emploie aussi le terme de *chaîne* mais plutôt dans le contexte des graphes orientés.

  La *longueur* d'un tel chemin $(s_0, \ldots, s_n)$ est le nombre d'arêtes utilisées pour aller du sommet s_0 au sommet s_n, il y en a donc n.

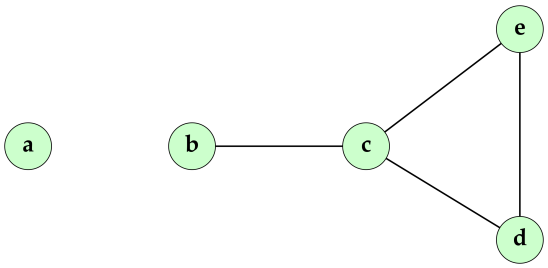
  La *distance* entre deux sommets est la longueur minimale d'un chemin reliant ces deux sommets.

  Un chemin ayant même origine que l'arrivée est appelé un *cycle*.
- On dit que le graphe est *connexe* si, pour tout couple de sommets distincts, il existe un chemin reliant ces deux sommets.

  Les sous-graphes connexes maximaux d'un graphe sont appelés ses *composantes connexes*.

Exemple

Considérons le graphe représenté ci-dessous :



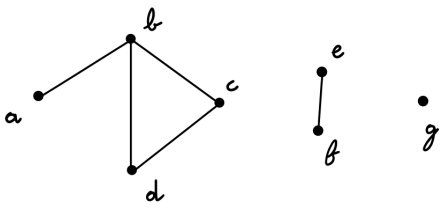
Le sommet **a** est isolé et il y a deux composantes connexes.

Un graphe peut être représenté de deux façons : à l'aide d'une *liste d'adjacence* ou à l'aide d'une *matrice d'adjacence.*

Une représentation par **liste d'adjacence** d'un graphe associe à chaque sommet du graphe la collection de ses voisins.

Exemple

Voici un graphe G ainsi que sa représentation par liste d'adjacence.



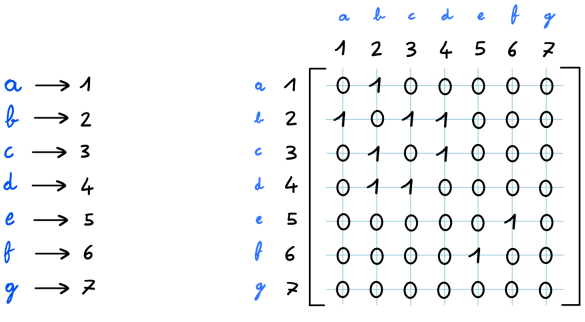
sommets	voisins
a	$\{b\}$
b	$\{a, c, d\}$
c	$\{b, d\}$
d	$\{b, c\}$
e	$\{f\}$
f	$\{e\}$
g	$\{\}$

La **matrice d'adjacence** d'un graphe d'ordre n est la matrice :

$$(a_{ij})_{(i,j)\in[\![1,n]\!]^2} \text{ avec } a_{ij} = \begin{cases} 1 & \text{si } i \text{ et } j \text{ sont voisins} \\ 0 & \text{si } i \text{ et } j \text{ ne sont pas voisins} \end{cases}$$

Exemple

En reprenant le graphe précédent, et en renommant ses sommets par les entiers de 1 à 7, on obtient la matrice d'adjacence suivante :



Notons que pour un graphe non orienté la matrice d'adjacence est toujours symétrique et la diagonale est toujours constituée de zéros.

La matrice d'adjacence M d'un graphe d'ordre n peut être représentée en Python par une liste de listes de la manière suivante :

- `M[i]` est la ligne d'indice `i` avec `i` $\in [\![0, n-1]\!]$;
- `M[i][j]` contient la valeur de a_{ij} avec `(i,j)` $\in [\![0, n-1]\!]^2$.

Exemple

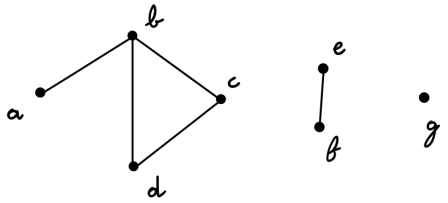
Toujours avec le même graphe, on a :

```
M = [[0, 1, 0, 0, 0, 0, 0],
     [1, 0, 1, 1, 0, 0, 0],
     [0, 1, 0, 1, 0, 0, 0],
     [0, 1, 1, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 1, 0],
     [0, 0, 0, 0, 1, 0, 0],
     [0, 0, 0, 0, 0, 0, 0]]
```

Pour mettre en œuvre en langage Python la représentation par liste d'adjacence, il est pertinent d'utiliser un dictionnaire dont les clés sont les sommets et les valeurs sont les listes de voisins.
Dans le cas d'un graphe pondéré, on peut remplacer les sommets (dans les listes de voisins) par un couple constitué du nom du sommet et du points de l'arête correspondante (ou de l'arc).

Exemples

1 ► Considérons à nouveau le graphe :



Avec un dictionnaire, on définit un tel graphe par :

```
G = {'a' : ['b'], 'b' : ['a', 'c', 'd'], 'c' : ['b', 'd'], 'd' : ['b', 'c'], 'e' : ['f'
    ], 'f' : ['e'], 'g' : []}
```

2 ► Quel est le graphe représenté par le dictionnaire suivant ?

```
G = {'a': [('b', 8), ('c', 2), ('d', 1)],
     'b': [('a', 8), ('c', 4), ('d', 6), ('e', 1)],
     'c': [('a', 2), ('b', 4), ('e', 2)],
     'd': [('a', 1), ('b', 6)],
     'e': [('b', 1), ('c', 2)]
       }
```

3 ► Écrire une fonction python prenant un dictionnaire (représentant la liste d'adjacence d'un graphe non orienté) et renvoyant le nombre de sommets et le nombre d'arêtes.

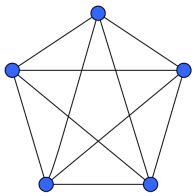
4 ► Écrire une fonction python prenant un dictionnaire (représentant la liste d'adjacence d'un graphe) et le nom d'un sommet et renvoyant son degré.

5 ► Écrire une fonction python prenant un dictionnaire (représentant la liste d'adjacence d'un graphe non orienté) et une liste `[a, b]` correspondant à deux sommets, et ajoutant une arête entre ces deux sommets.

6 ► Écrire une fonction python `aretes(gr)` prenant en argument un dictionnaire (représentant la liste d'adjacence d'un graphe non orienté) et qui renvoie une liste des arêtes du graphe `gr` sans répétition d'arête.

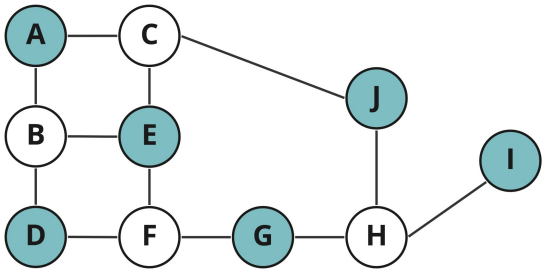
7 ► Écrire la fonction `listed(gr, d)` qui renvoie la liste classée par ordre croissant lexicographique des sommets du graphe `gr` dont le degré est `d`.

8 ► On s'intéresse maintenant à la création d'un *graphe complet*. Il s'agit d'un graphe tel que tout couple de sommets disjoints est relié par une arête. Le graphe complet d'ordre n est noté K_n. Par exemple, voici K_5 :



Écrire la fonction `graphe_complet(n)` qui renvoie l'implémentation en Python de la liste d'adjacence de K_n, les sommets étant numérotés de 0 à $n-1$.

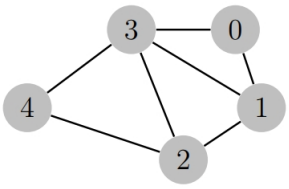
9 ► On dit qu'un graphe (non orienté et connexe) est ***biparti*** lorsque l'ensemble S des sommets de G peut être partitionné en deux sous-ensembles disjoints S_1 et S_2 tels que chaque arête de S ait une extrémité dans S_1 et l'autre dans S_2. Par exemple :



Écrire une fonction `biparti(gr, S)` qui prend pour argument un graphe `gr` et une liste `S` de sommets de ce graphe et qui renvoie un booléen indiquant si ce graphe est biparti avec l'un des deux sous-ensembles de sommets qui correspond à `S`.

Exercice 3

On considère un graphe non orienté G = (S, A).
On dit qu'un ensemble X ⊂ S de sommets de G vérifie la propriété (⋆) lorsqu'il n'y a aucune arête entre des sommets de X : pour tous u et v dans X, $(uv) \notin A$.
Par exemple, avec le graphe ci-dessous, les ensembles {0}, {0,2}, {1,4} vérifie cette propriété mais pas l'ensemble {0,1,2}.



1. Écrire une fonction `propstar` qui prend en argument un graphe et une liste `X` de sommets et qui renvoie `True` si `X` vérifie la propriété (⋆) et `False` sinon.
2. Écrire une fonction `propstarmax` qui prend en argument un graphe et une liste `X` de sommets et qui renvoie `True` si `X` vérifie la propriété (⋆) et n'est contenu dans aucun autre ensemble vérifiant la propriété (⋆), et `False` sinon.