Institut Villebon - Georges Charpak
T.P. 1

Savoir faire :	• Connaitre les différents types de représentation d'un graphe.
	• Savoir implémenter une structure de graphe en Python.
	• Savoir effectuer un parcours de graphe.
	• Savoir implémenter l'algorithme de calcul des composantes connexes.

1 Condition de rendu du T.P.

Ce T.P. sera à rendre par mail à olivier.bouillot@villebon-charpak.fr, au plus tard le mardi 24 novembre à 23h59. Il sera à réaliser dans un premier temps seul, pendant la séance, puis à finir en binôme avec un membre de l'autre groupe.

Votre mail d'envoi aura pour sujet : [UE 5i4] Rendu de TP 1 + `prenom_1` + `prenom_2`, où `prenom_1` et `prenom_2` seront bien entendu remplacé par les prénoms des membres du binômes. Il contiendra une et une seule pièce jointe qui sera une archive nommée `nom_1_nom_2_-_TP_1.zip`, où `nom_1` et `nom_2` correspondent aux noms de familles des membres du groupe, `nom_1` étant avant `nom_2` dans le dictionnaire.

Veillez bien à ne pas modifier les noms donnés dans l'énoncé, car une partie de votre T.P. sera corrigé automatiquement : si vous ne respectez pas cette consigne, vous aurez alors 0, car aucun des tests que votre code subira ne passera...

Enfin, n'oubliez pas de respecter des règles d'hygiènes correctes pour votre code : cela sera hautement pris en compte dans la notation. Voir §8 et §9.

2 Représentation d'un graphe

Un graphe orienté est un couple d'ensemble constitué de l'ensemble de ses sommets et de l'ensemble de ses arêtes.En machine, pour représenter un graphe, on pourrait alors simplement stocker ces deux ensembles. Cependant, cela entraîne un problème d'efficacité :

⇝ l'accès aux arêtes incidentes à un sommet demandera systématiquement une recherche ;

⇝ l'accès aux sommets adjacents à un sommet demandera systématiquement une recherche ;

D'ailleurs, dans un graphe, ce sont les relations d'incidence et d'adjacence qui sont en fait importantes. Il y a alors plusieurs structures de données possibles pour stocker en mémoire un graphe :

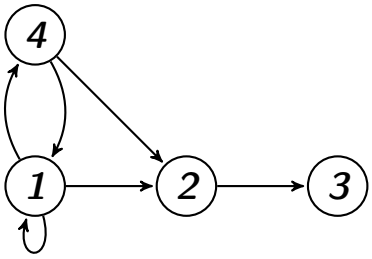
- la matrice d'adjacence ;
- la matrice d'incidence ;
- la liste d'adjacence ;
- la liste d'incidence.

2.1 A partir de la matrice d'adjacence

Un graphe $G = (S, A)$ orienté à n sommets (assimilés comme étant les entiers de 1 à n) peut être représenté par une matrice M de taille $n \times n$, appelée *matrice d'adjacence* et définie par :

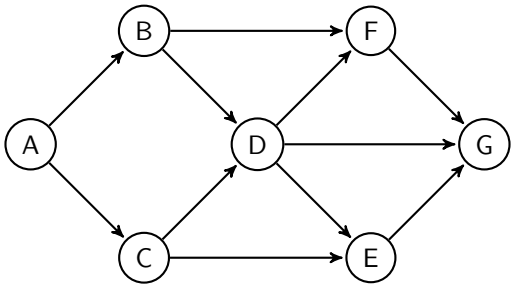
- les indices des lignes et des colonnes représentent les sommets de G.
- $M_{i,j}$ est égale au nombre d'arêtes reliant les sommets i et j.

Exemple 1. *La matrice d'adjacence du graphe suivant vaut :*



$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

▸ **Exercice 1.** Donner la matrice d'adjacence du graphe suivant :



Le résultat sera noté dans un fichier nommé `matrice_adjacence.txt`, une ligne de la matrice sur une ligne du fichier, les nombres étant simplement espacés et sans espace à la fin d'une ligne, et les sommets classé alphabétiquement.

Le contenu du fichier serait le suivant si la matrice était celle de l'exemple précédent :

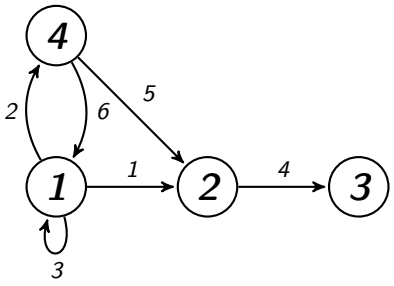
```
1 1 0 1
0 0 1 0
0 0 0 0
1 1 0 0
```

2.2 A partir de la matrice d'incidence

Un graphe $G = (S, A)$ orienté à n sommets (assimilés comme étant les entiers de 1 à n) et p arêtes (assimilées comme étant les entiers de 1 à p) peut être représenté par une matrice M de taille $n \times p$, appelée *matrice d'incidence* et définie par :

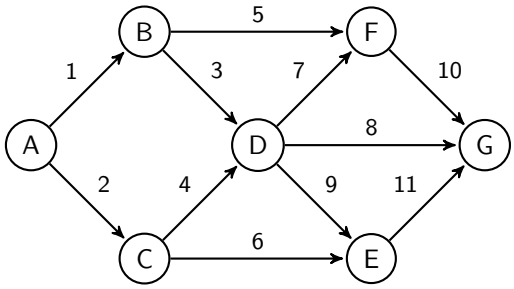
- les indices des lignes et des colonnes représentent respectivement les sommets de G et les arêtes de G.
- $M_{i,j}$ vaut : $\begin{cases} 1 & \text{lorsque le sommet } i \text{ est l'origine de l'arête } j. \\ -1 & \text{le sommet } i \text{ est la destination de l'arête } j. \\ 0 & \text{sinon.} \end{cases}$

Exemple 2. *La matrice d'incidence du graphe suivant vaut :*



$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 1 \end{pmatrix}$$

▸ **Exercice 2.** Donner la matrice d'incidence du graphe suivant :



Le résultat sera noté dans un fichier nommé `matrice_incidence.txt`, une ligne de la matrice sur une ligne du fichier, les nombres étant simplement espacés et sans espace à la fin d'une ligne !

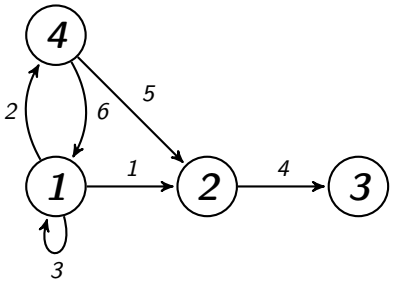
Le contenu du fichier serait le suivant si la matrice était celle de l'exemple précédent :

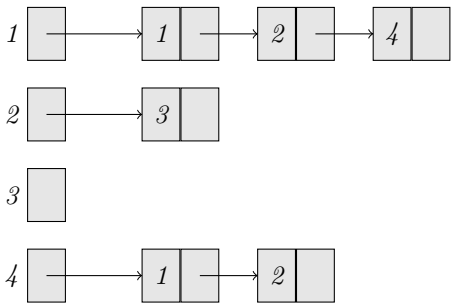
```
1 1 0 0 0 -1
-1 0 0 1 -1 0
0 0 0 -1 0 0
0 -1 0 0 1 1
```

2.3 A partir de la liste d'adjacence

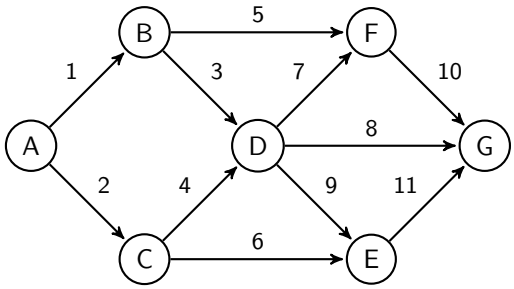
Un graphe $G = (S, A)$ orienté peut être représenté par la liste chainée des voisins de chaque sommet. Celle-ci s'appelle la liste d'adjacence de G.

Exemple 3. *La liste d'adjacence du graphe suivant vaut :*





▸ **Exercice 3.** Donner la liste d'adjacence du graphe suivant :



Le résultat sera noté dans un fichier nommé `liste_adjacence.txt`, une ligne par liste chainée, la liste chainée étant représentée par le sommet concernée suivi immédiatement de : puis de la suite des sommets que la liste chainée représente, ceux-ci étant étant simplement espacés et sans espace à la fin d'une ligne !

Le contenu du fichier serait le suivant si la liste d'adjacence était celle de l'exemple précédent :

1:1 2 4
2:3
3:
4:1 2

N.B. : L'ordre, à l'intérieur des listes chainées est l'ordre alphabétique, ou l'ordre sur des entiers, selon que les sommets soient nommés par des chaines de caractères ou des entiers.

parler de liste d'incidence

comparer les complexités des fonctions traditionnelles

3 Implémentation des graphes

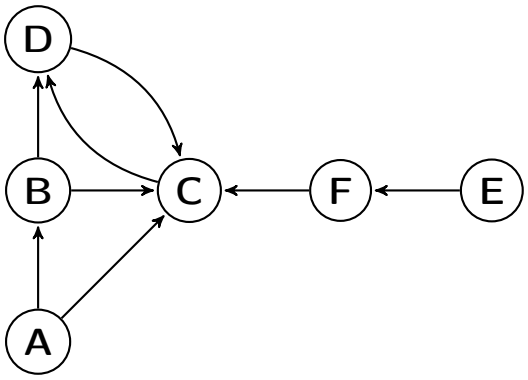
On opte ici pour une écriture des graphes sous forme de liste d'incidence. Pour cela, on suivra le conseil de Guido Van Rossum, le créateur de Python :

> "Few programming languages provide direct support for graphs as a data type, and Python is no exception. However, graphs are easily built out of lists and dictionaries. For instance, here's a simple graph (I can't use drawings in these columns, so I write down the graph's arcs):
>
> A –> B
> A –> C
> B –> C
> B –> D
> C –> D
> D –> C
> E –> F
> F –> C
>
> This graph has six nodes (A-F) and eight arcs. It can be represented by the following Python data structure:

```
graph = {'A': ['B', 'C'],
         'B': ['C', 'D'],
         'C': ['D'],
         'D': ['C'],
         'E': ['F'],
         'F': ['C']}
```

> This is a dictionary whose keys are the nodes of the graph. For each key, the corresponding value is a list containing the nodes that are connected by a direct arc from this node. This is about as simple as it gets (even simpler, the nodes could be represented by numbers instead of names, but names are more convenient and can easily be made to carry more information, such as city names)."

Le graphe considéré par Guido est le suivant :



▸ **Exercice 4.** Ecrire une classe `Graph` ayant trois attributs :

- un ensemble `nodes` représentant l'ensemble des noeuds du graphe ;
- une liste `edges` représentant la liste des arêtes du graphe ;
- un dictionnaire `adjacency_list` représentant la liste d'adjacence du graphe (comme décrit par Guido van Rossum).

Le constructeur doit uniquement permettre de créer le graphe vide.

Rappels : La classe `Graph` doit contenir une docstring, tout comme son constructeur et toutes les méthodes qu'elle contiendra. Celles-ci devront être en **anglais**.

▸ **Exercice 5.** Ajouter à la classe `Graph` une méthode `add_a_node(self, node_name)` permettant d'ajouter le sommet `node_name` au graphe courant.
On fera attention à ne pas recréer de sommet si celui-ci est déjà crée mais aussi à mettre à jour la liste d'adjacence.

▸ **Exercice 6.** Ajouter à la classe `Graph` une méthode `add_an_edge(self, from_node, to_node):` permettant d'ajouter une arête au graphe courant entre les sommets `from_node` et `to_node`.
On fera attention de mettre à jour la liste d'adjacence du graphe après ajout de l'arête, mais surtout à ne pas créer d'arête si les sommets passés en paramètres ne sont pas des sommets du graphe : dans ce cas une exception de type `NameError` se devra d'être levée.

▸ **Exercice 7.** Dans un fichier `graph_examples.py`, créer le graphe correspondant au graphe mentionné par Guido Van Rossum.

N'oubliez pas de commenter votre fichier de tests.

Le fichier `graph_examples.py` sera systématiquement mis à jour, et puisque examples se trouve être au pluriel, il contiendra d'autres exemple que celui de Guido van Russo.

Faire faire des tests avant...

▸ **Exercice 8.** Ecrire la méthode `__str__(self)` permettant de représenter visuellement un graphe à la manière dont Guido Van Rossum l'a fait dans la citation précédente.

Précisément, si le graphe créée dans l'exercice précédent est disponible dans la variable `guido_s_graph`, alors la commande suivante, placée dans le fichier `graph_examples.py`,

```
print(guido_s_graph)
```

devra afficher précisément en console :

```
************************
* Display of the graph *
************************
Nodes:
------

A, E, C, D, F, B

Edges:
------

A ---> B
A ---> C
B ---> D
B ---> C
C ---> D
D ---> C
E ---> F
F ---> C
============================
```

Le fichier `graph_examples.py` sera mis à jour afin de faire afficher le graphe de Guido van Russo.

4 Parcours d'un graphe

Un *parcours de graphe* est un algorithme consistant à explorer les sommets d'un graphe de proche en proche à partir d'un sommet initial. Ces algorithmes reviennent à de nombreuses reprises en théorie des graphes, notamment dans : la résolution des problèmes suivant :

- Connexité et forte connexité ;
- Existence d'un circuit ou d'un cycle ;
- Calcul des plus courts chemins (notamment l'algorithme de Dijkstra).

Il existe de nombreux algorithmes de parcours, les plus classiques étant le *parcours en largeur* et le *parcours en profondeur*. Pour pouvoir les implémenter simplement, nous allons avoir besoin de deux structures de données extrêmement importante en informatique : les *piles* et les *files*.

4.1 Notion de pile et de file

Les notions de *pile* et de *file* sont deux structures de données abstraites importantes en informatique. On limite la présentation de ces notions aux besoins des parcours de graphes envisagés ci-après.

La structure de pile (aussi appelée structure LIFO : last in, first out) est celle d'une pile d'assiettes:

- Pour ranger les assiettes, on les empile les unes sur les autres.
- Lorsqu'on veut utiliser une assiette, c'est l'assiette qui a été empilée en dernier qui est utilisée.

La structure de file (aussi appelée structure FIFO : first in, first out) est celle d'une file d'attente :

- Les nouvelles personnes qui arrivent se rangent à la fin de la file d'attente.
- La personne servie est celle qui est arrivée en premier dans la file.

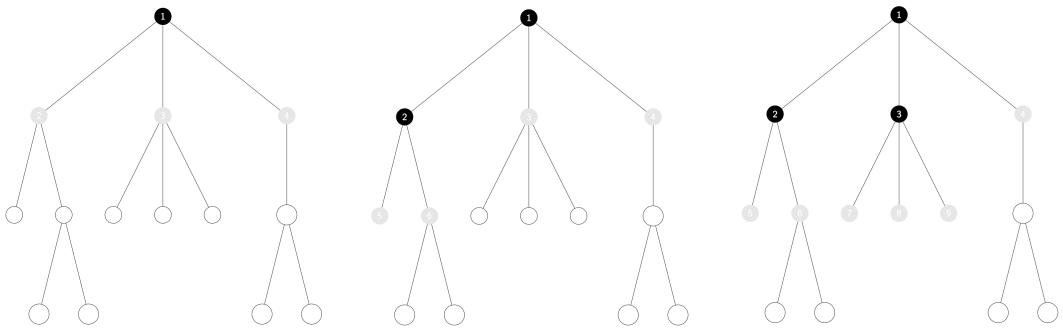
4.2 Parcours en largeur ou "breadth first search"

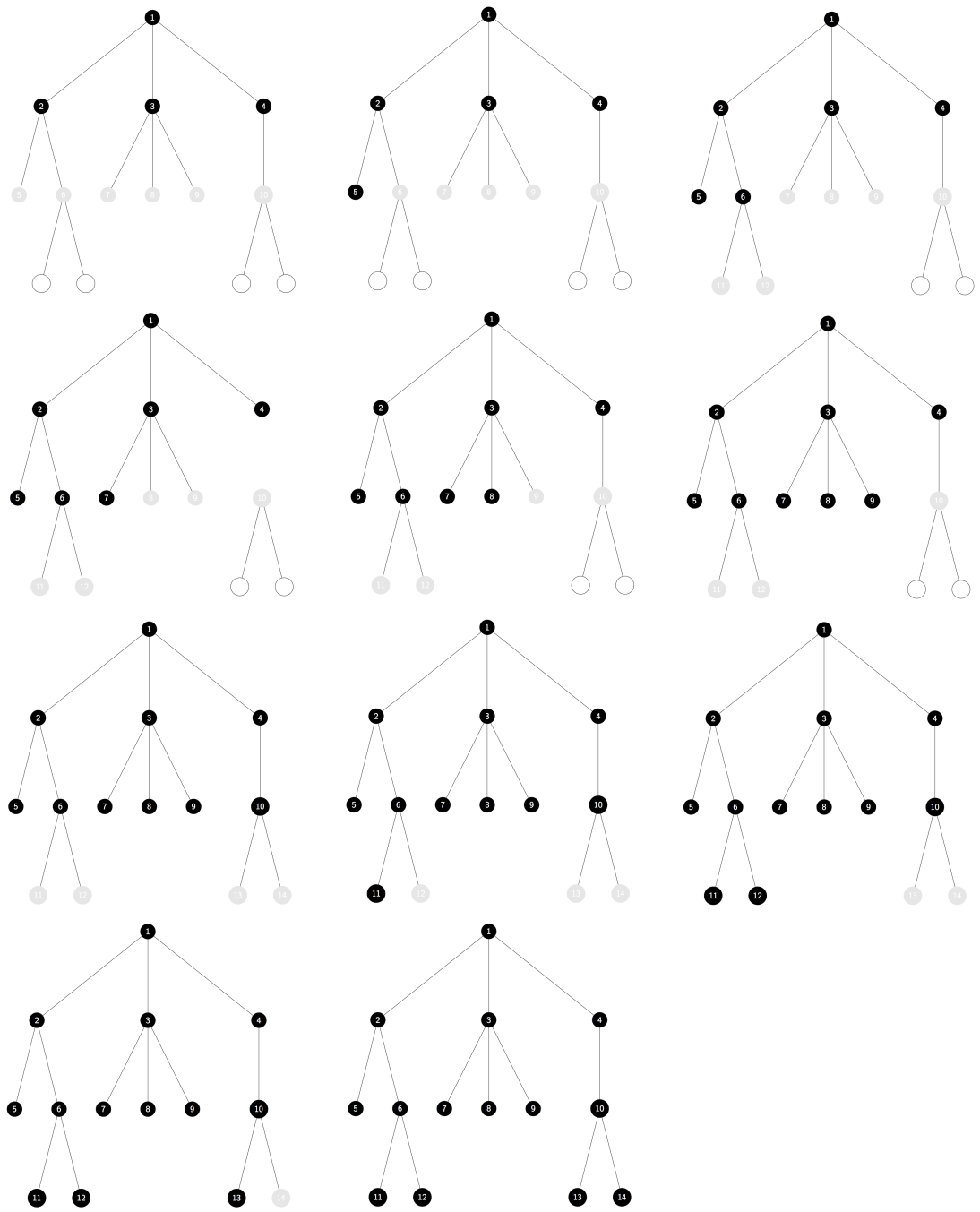
Lors d'un parcours en largeur d'un graphe, on traite en priorité les sommets découverts en premier. On utilisera donc une structure de file :

1. On enfile le sommet de départ.
2. On visite les voisins de la tête de file. On les enfile (en les numérotant au fur et à mesure de leur découverte) s'ils ne sont pas déjà présents dans la file, ni déjà passés dans la file.
3. On défile (c'est à dire que l'on supprime de la file l'élément en tête de la file).
4. On recommence les points 2 et 3 tant que c'est possible (c'est à dire tant que la file n'est pas vide).

L'algorithme de parcours en largeur va visiter en premier lieu tous les sommets du graphe à distance 1 du départ, puis tous les sommets à distance 2 du départ, puis tous les sommets à distance 3, $\cdots$ (c'est en fait cette propriété qui donne son nom à ce type de parcours).

Voici un exemple de parcours en largeur d'un arbre. En noir se trouvent les sommets traités, en gris sont les sommets situés sur la pile, et en blanc les sommets encore non découvert par le parcours.





▸ **Exercice 9.** Ecrire une méthode `breadth_first_search(departure)` dans la classe `Graph` qui effectue un parcours en largeur du graphe courant en partant du sommet `departure` et renvoie un dictionnaire `parents` tel qu'en fin de parcours, pour tout sommet `s` du graphe, autre que `departure`, `parents[s]` sera le père de `s`, c'est-à-dire le sommet à partir duquel le sommet `s` a été découvert lors du parcours. `parents[departure]` vaudra `None`.

On utilisera les variables locales suivantes :

- un dictionnaire `colors` tel que pour tout sommet `s`, `colors[s]` vaut `'white'` si le sommet `s` n'est pas passé dans la file, `'grey'` s'il est dans la file, `'black'` lorsqu'il est sorti de la file.

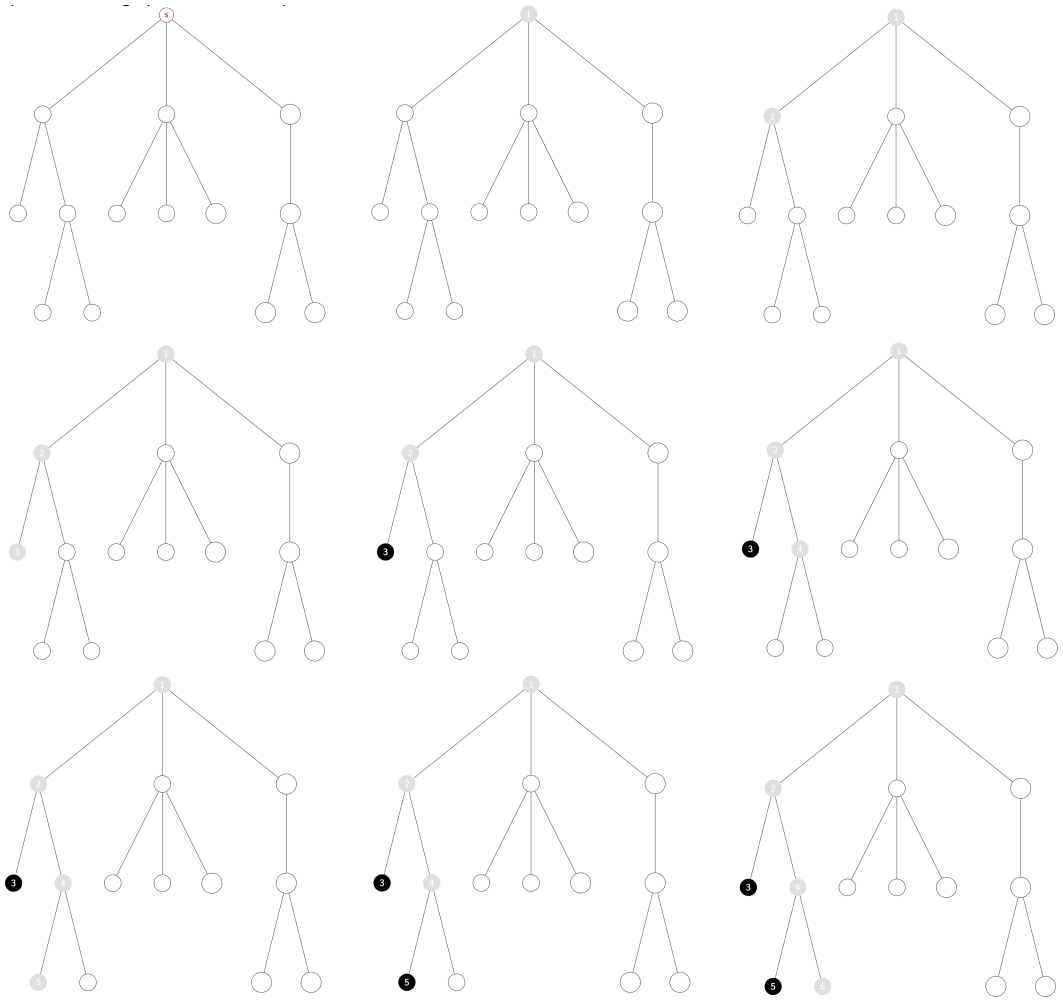
- une liste `fifo` utilisée comme une file : on ajoute en queue un sommet du graphe lorsqu'il est découvert, on le défile lorsque son traitement est terminé : c'est le traitement prioritaire des sommets découverts au plus tôt.

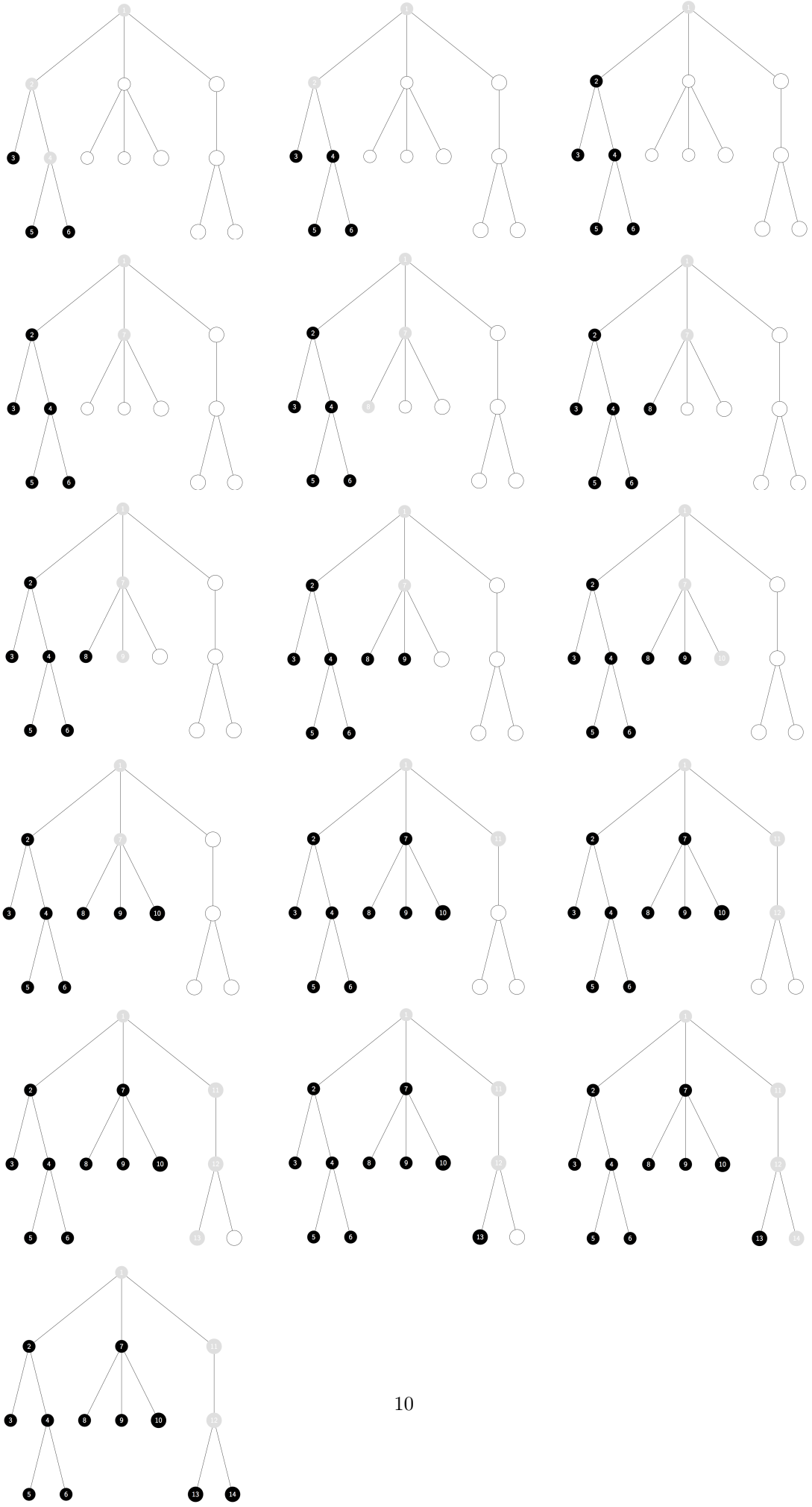
4.3 Parcours en profondeur : depth first search

Lors d'un parcours en profondeur d'un graphe, on traite en priorité les sommets découverts en dernier. On utilisera donc une pile :

1. On empile le sommet de départ.
2. Si le sommet de la pile possède des voisins qui ne sont pas dans la pile, ni déjà passés dans la pile, alors on sélectionne l'un de ces voisins et on l'empile (en le numérotant au fur et à mesure des découvertes), sinon on dépile (c'est-à-dire que l'on supprime l'élément du sommet de la pile).
3. On recommence au point 2 tant que la pile n'est pas vide.

Voici un exemple de parcours en profondeur d'un arbre. En noir se trouvent toujours les sommets traités, en gris sont les sommets situés sur la pile, et en blanc les sommets encore non découvert par le parcours.





▸ **Exercice 10.** Ecrire une méthode `depth_first_search(departure)` dans la classe `Graph` qui effectue un parcours en profondeur du graphe courant en partant du sommet `departure` et renvoie un dictionnaire `parents` tel qu'en fin de parcours, pour tout sommet `s` du graphe, `parents[s]` sera le père de `s`, c'est-à-dire le sommet à partir duquel le sommet `s` a été découvert lors du parcours. `parents[departure]` vaudra `None`.

On utilisera les variables locales suivantes :

- un dictionnaire `colors` tel que pour tout sommet `s`, `colors[s]` vaut `'white'` si le sommet `s` n'est pas passé dans la file, `'grey'` s'il est dans la file, `'black'` lorsqu'il est sorti de la file.
- une liste `lifo` utilisée comme une pile : on empile un sommet du graphe lorsqu'il est découvert, on le défile lorsque son traitement est terminé : c'est le traitement prioritaire des sommets découverts au plus tard.

5 Applications à la connexité des graphes non-orientés

L'objectif de cette partie est de pouvoir déterminer simplement si un graphe est connexe, et sinon de déterminer ses composantes connexes. Mais cette notion n'est valide que pour les graphes non-orientés... Or, nous avons implémenté une classe pour les graphes non-orientés...

Bien évidemment, pour chacune des méthodes écrites dans cette partie (comme ailleurs, d'ailleurs...!), le fichier `graph_examples.py` sera mis à jour en utilisant correctement les rattrapages d'exceptions.

▸ **Exercice 11.** Compléter la classe `Graph` en écrivant une méthode `is_non_oriented(self)` renvoyant `True` si le graphe est non-orienté et `False` s'il est orienté.

▸ **Exercice 12.** Compléter la classe `Graph` en écrivant une méthode `connected_components(self)` renvoyant une liste dont les éléments sont les listes des sommets des composantes connexes du graphe courant.

On veillera bien à n'appliquer cette méthode qu'à des graphes non-orienté en levant une exception de type `TypeError` dans les cas d'impossibilité d'application de l'algorithme permettant de déterminer une composante connexe dans un graphe non-orienté.

▸ **Exercice 13.** Compléter la classe `Graph` en écrivant une méthode `is_connected(self)` renvoyant `true` ou `false` selon que le graphe courant est connexe ou non.

Encore une fois, on fera attention à ce que cette méthode ne concerne que les graphes non-orienté en levant une exception de type `TypeError` dans le cas contraire.

6 Bonus : Compléments possibles

▸ **Exercice 14.**

1. Proposer un algorithme décidant si un graphe est biparti ou non.
2. Implémenter cet algorithme dans une méthode `is_bipartite(self)` en expliquant en détail dans la docstring l'algorithme déployé.

Les parcours en largeur ou en profondeur d'un graphe produisent un arbre dont la racine est le point de départ du parcours. Il n'est qlors pas difficile de se convaincre qu'un sommet `s` d'un graphe est un point d'articulation de celui-ci si et seulement si la racine de l'arbre obtenu par l'un de ces parcours possède deux fils, ou plus.

Ainsi, un algorithme naïf pour déterminer les points d'articulations d'un graphe consiste à déterminer les parcours issus de chacun des sommets du graphe, et de compter le nombre de fils de la racine de l'arbre obtenu au cours du parcours.

Bien évidemment, on sait faire cela de manière beaucoup plus efficace, mais l'algorithme est loin d'être aussi simple...

▸ **Exercice 15.** Ecrire une méthode `articulation_points(self)` dans la classe `Graph` retournant l'ensemble des sommets du graphe courant qui se trouve être des points d'articulation.

7 Annexe A : Les exceptions en python !

Une erreur, qui n'est pas une erreur due au programmeur peut être détectée au moment de l'exécution du programme. Ces erreurs sont appelées des **exceptions** et sont systèmatiquement fatales !

De même que les différents objets utilisés en Python possèdent un type, une exception en possède aussi un. Les types des exceptions sont multiples, comme le montre l'exemple suivant :

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam * 3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> 2 + '2'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

A chaque fois, la dernière ligne du message d'erreur (du `Traceback`) indique ce qu'il s'est passée. Dans la première instruction, nous avons tenté de faire une division par 0 ; dans la seconde, nous avons utilisé une variable non déclarée ; enfin, dans la troisième, nous avons tenté de faire une addition entre deux objets de type différents...

Voyons désormais comment rattraper des exceptions dans un programme. Pour cela nous allons utiliser un `try/except`. L'exemple suivant permet d'effectuer une saisie controllée, et plus précisément ici de tester si l'utilisateur entre bien une chaine de caractères représentant un entier ou non.

```
try:
    nb_str = int(input("Donnez␣moi␣un␣entier␣svp.\n"))
    print("Bravo␣!␣Vous␣avez␣reussi␣a␣ecrire␣un␣nombre␣!!!")
except ValueError:
    print("Ooups!␣␣Ceci␣ne␣semble␣pas␣etre␣un␣nombre␣valide...")
print("Merci␣d'avoir␣participe␣a␣cette␣experience.")
```

L'instruction `try` fonctionne comme cela :

- Au commencement, ce qui se trouve entre les mots clés `try` et `except` est exécuté.
- Si aucune exception n'apparaît, ce qui suit le mot clé `except` n'est pas exécuté. L'exécution reprends donc au bloc suivant. Dans l'exemple précédent, il s'agira de l'instruction

```
print("Merci d'avoir participe a cette experience.")
```

- Si une exception apparaît pendant l'exécution des commandes situées entre les mots clés `try` et `except`, les instructions suivantes ne seront pas exécutées. Cette fois, si l'exception levée est de l'un des types décrit après le mot clé `except` (on dit que **l'exception a été rattrapée**), la clause `except` est alors exécutée. Après l'exécution de cette portion de code, on continue sur le bloc suivant. Ici, il s'agit toujours de l'instruction

```
print("Merci d'avoir participe a cette experience.")
```

- Si une exception apparaît pendant l'exécution des commandes situées entre les mots clés `try` et `except`, mais n'est pas de l'un des types décrit après le mot clé `except`, **l'exception n'est pas rattrappée**, l'exécution du programme s'interrompt avec la levée de cette exception.

Il ne reste plus qu'à voir une chose sur les exceptions. Au moment où l'on se rend compte que les lignes de code que l'on écrit ne pourront pas s'effectuer raisonnablement (par exemple, ajouter une arête dans un graphe entre deux sommets qui n'existent pas encore...), on va arrêter le programme pour signaler à l'utilisateur le problème : on dit qu'on *lève une exception*

Pour cela, on utilise un nouveau mot-clé pour lever une exception : le mot-clé `raise`, qui s'utilise comme suit :

```
raise TypeDeLException("text to print")
```

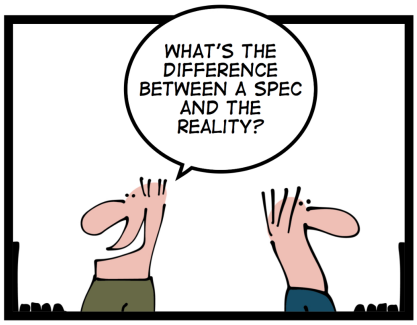
Notamment, on lèvera une exception à chaque fois que les conditions d'utilisations d'une méthode ne sont pas satisfaites.

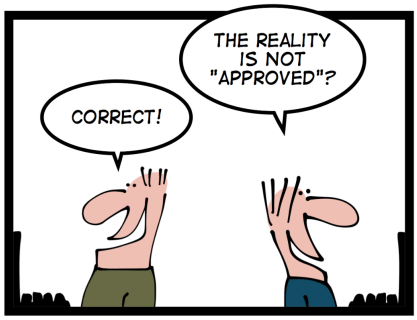
Voici un exemple de fonction ne pouvant s'appliquer que sur des nombres positif ou nul. Il s'agit de calculer la partie entière de la racine carrée d'un nombre :

```
def integer_square_root(x):
    if x < 0:
        raise ValueError, "How dare you to give me a negative number???"
    else:
        n = 0
        while n * n < x:
            n += 1
        return n
```

8 Annexe B : Une bonne hygiène, c'est toujours mieux !

Pour faciliter la lecture du code par d'autres que son auteur, il existe des règles d'hgiènes typographiques à respecter. Il s'agit de spécifications. Pour python, on s'appuiera sur le fameux "PEP 0008 – Style Guide for Python Code". On y trouve les recommandations suivantes (et beaucoup d'autres aussi)





- Les noms de variables sont construits avec des lettres minuscules, et les mots séparés par des “underscores”, afin d’augmenter la lisibilité :

  ma_variable : oui
  pasMa_variable : Bouhhhhh !!! Pas beau

- Les constantes sont elles nommées en lettres majuscules, les mots étant séparés par des “underscores” :

  MA_CONSTANTE : oui
  pas_ma_constante : Bouhhhhh !!! Pas beau

- Les noms de fonctions sont construits avec des lettres minuscules, et les mots sont sépars par des “underscores” afin d’augmenter la lisibilité :

  nom_de_fonction_valide oui
  noMde_fonctionInvalide: non ! Trop moche !

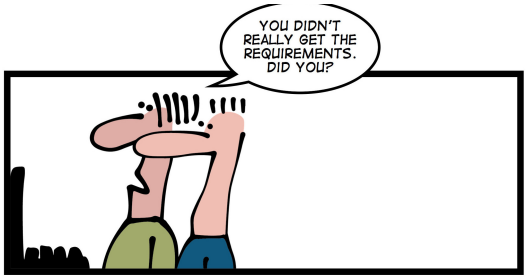
- Les noms d’exceptions (cf §7) sont construits en “CamelCase” : la première lettre de chaque mot est une majuscule et les mots sont collés les uns aux autres.

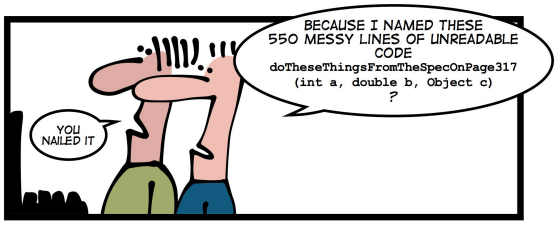
  MonException : oui
  pas_mon_Exception : non...

- Les noms de classe (on verra peut être plus tard ce que c’est) sont aussi construits en CamelCase.

  Dans tous ces cas, les seuls caractères autorisés sont :

  `a`, `b`, $\cdots$, `y`, `z`, `0`, `1`, $\cdots$, `9`, `_`





Enfin, il est important que toutes vos variables et fonctions aient un nom significatif qui permette immédiatement de savoir à quoi elles correspondent. Une règle simple qu’on vous conseille de suivre pour s’en assurer est de toujours utiliser des noms de variable avec au moins trois caractères. On pourra s’affrachir de cette règle dans le cas des variables d’itération (utilisées dans les boucles `for`).

Enfin, tous vos fichiers Python devront contenir une entête indiquant quelques informations :

- le contexte de création du fichier ;
- la date de création du fichier ;
- le nom du ou des auteurs du fichier ;
- la date de dernière modification.

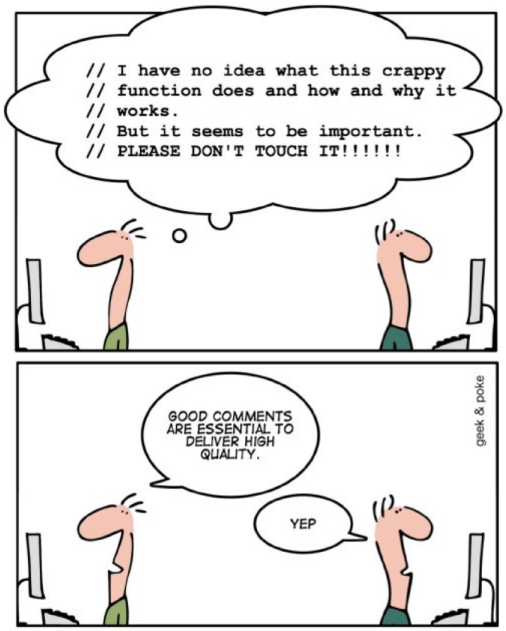
```
##############################
# Institut Villebon, UE 3.1
# Travaux pratique 1
# Auteur : O. Bouillot
# Date de creation : 09/11/15
# Date de derniere modification : 09/11/15
##############################
```

Dans le cas d'un projet d'ampleur sérieuse, il est aussi possible d'envisager d'avoir une entête un peu plus longue reprenant l'aspect d'un fichier `log` et indiquant, pour chaque date importante du développement, les modifications majeures effectuées.

9 Annexe C : Lisibilité du code et documentation

Pour rendre plus lisible un code, il y a deux techniques :

1. mettre des commentaires dans le code expliquant ce qui est fait dans tel partie du code ou détail technique ;
2. créer des fonctions réalisant telle ou telle partie du code.



Le fait de mettre des commentaires permet de mieux comprendre ce que le code fait, mais choisir des noms de variables et de fonctions adéquates aide encore plus...

Enfin, dans une fonction en python, il existe un style de documentation particulier : les **docstrings**. Voici la différence, pour une même fonction, entre avoir ou non une docstring :

```
def ajouter(a, b):
    return a + b
```

```
def ajouter(a, b):
    """␣Ajoute␣deux␣nombres␣l'un␣a␣l'autre␣et␣retourne␣le␣resultat.
␣␣␣␣␣␣␣␣>>>␣ajouter(2,␣3)
␣␣␣␣␣␣␣␣5
␣␣␣␣"""
    return a + b
```

Ecrire des docstrings offre de nombreux avantages :

- La fonction `help()` affiche cette documentation dans un shell. Les outils de programmation tels que les shells ou les IDE (Environnement de développement intégré) affichent cette documentation quand le développeur qui ne lit pas votre code, mais qui l'utilise, en a besoin.

```
>>> help(ajouter)
Help on function ajouter in module __main__:

ajouter(a, b)
    Ajoute deux nombres l'un a l'autre et
    retourne le resultat.
```

- On peut générer une bonne documentation du code avec des commandes qui extraient ces docstrings.
- C'est un mécanisme standardisé de documentation : tout le monde sait que si c'est là, et que ça a cette forme, c'est de la documentation.
- On peut mettre des exemples d'utilisation dans les docstrings, qui servent alors de tests.