Initiation à l'informatique

MSI 102, Université Bordeaux 1

Equipe enseignante MSI 102

2 septembre 2010

Table des matières **1**

Chapitre 1. Premiers pas en Python

1.1 Affectation et expressions

Python permet tout d'abord de faire des calculs. On peut évaluer des expressions (arithmétiques ou booléennes, par exemple). Il faut pour cela respecter la syntaxe du langage. On peut sauvegarder des valeurs dans des variables. Chaque variable a un nom. Par exemple, pour sauvegarder la valeur 123×45 dans une variable qui s'appelle `x`, on tape l'instruction

```
x = 123 * 45
```

On peut ensuite réutiliser `x`, par exemple calculer x^2 en tapant `x*x`. Contrairement à sa signification mathématique, le symbole `=` signifie « *calculer la valeur à droite du signe* `=` *et mémoriser le résultat dans la variable dont le nom est à gauche du signe* `=` ».

Exercice 1.1.1 Que contiennent les variables `x`, `y`, `z` après les instructions suivantes ?

```
x = 6
y = x + 3
z = 2 * y - 7
```

Exercice 1.1.2 L'instruction `i = i + 1` a-t-elle un sens ; si oui lequel ? Et `i + 1 = i` ?

Exercice 1.1.3 Que contiennent les variables `x`, `y`, `z` après les instructions suivantes ?

```
x = 6
y = 7
z = x
z = z + y
```

Exercice 1.1.4 Quel est le résultat de l'instruction `x = 2 * x` ? Si la valeur initiale de x est 1, donner les valeurs successives de x après une, deux, trois, etc. exécutions de cette instruction.

Exercice 1.1.5 Écrire une suite d'instructions permettant d'échanger le contenu de deux variables `a` et `b` (sans utiliser le raccourci Python `a, b = b, a`).

Exercice 1.1.6 Écrire une instruction qui affecte à la variable `m` le résultat du calcul de la moyenne de deux réels `a` et `b` pondérés par les coefficients respectifs `coef1` et `coef2`.

Expressions booléennes

Une expression booléenne est une expression qui n'a que deux valeurs possibles, *True* (vrai) ou *False* (faux) ; les tests `x == y` (égalité), `x != y` (inégalité), `x < y`, `x <= y`, etc. sont des expressions booléennes. On peut combiner des expressions booléennes avec les opérateurs `and`, `or` et `not`.

Exercice 1.1.7 Écrire une expression qui vaut *True* si `x` appartient à $[0, 5[$ et *False* dans le cas contraire.

Divisions entières

Le quotient entier q de deux entiers a et b positifs, et le reste r de la division sont définis par :

$$a = bq + r \qquad \text{avec} \quad 0 \leq r < b$$

Par exemple le quotient et le reste de la division de 17 par 5 sont 3 et 2 car $17 = 5 \times 3 + 2$; en Python le quotient entier de a par b est noté `a//b`, et le reste `a%b`.

Exercice 1.1.8 Écrire une expression qui vaut *True* si l'entier n est pair et *False* dans le cas contraire.

Premiers pas avec l'interprète Python.

Exercice 1.1.9 Taper les expressions suivantes et expliquer précisément les résultats obtenus.

```
11 * 34
13.4 - 6 # ceci est un commentaire
13 / 4
13 // 4 # division entiere
13 % 2
14 % 3
i = 5
i = i + 4
i
j = 0
k
i < j
i != 9
i == 9
i, j, i*i
```

1.2 Fonctions

Exemple de définition de fonction :

```
def f(x):
    return x*x + x + 1
```

Ne pas oublier les deux points à la fin de la première ligne. Les lignes suivantes (dans ce premier exemple il y en a une seule) contiennent des instructions qui constituent le *corps* de la définition ; ces instructions seront exécutées chaque fois qu'on utilisera la fonction, et doivent être *indentées*, c'est-à-dire décalées pour indiquer à Python qu'elles font partie de la définition de la fonction.

Ne pas confondre *définir* et *utiliser* une fonction : en jargon informatique on distingue définition et *appel* de fonction. De même ne pas confondre les *paramètres* dans la définition (ici il y en a un seul, nommé x), et les *arguments* utilisés lors de l'appel. Le premier exercice ci-dessous illustre ces notions.

La plupart des fonctions servent, comme en mathématiques, à *calculer* un résultat, indiqué dans l'instruction `return`.

Exercice 1.2.1 Après avoir défini la fonction `f` comme ci-dessus, taper les expressions suivantes et expliquer les résultats obtenus.

```
f(2)
t=4
f(t)
x = (3*t + 1) / 2
x, f(x)
x = (3*t + 1) // 2
x, f(x)
f (x - t)
```

Exercice 1.2.2 Écrire une fonction `moyenne(a,b)` qui calcule la moyenne de deux nombres a et b.

Exercice 1.2.3 Écrire une fonction `pair(n)` qui teste si n est pair ; la valeur calculée doit être *booléenne*, c'est-à-dire égale à *True* ou *False*.

1.3 Conditionnelles

L'instruction `if ... elif ... else` permet de tester des conditions pour exécuter ou non certaines instructions. Exemple :

```
h = 0
stop = False
if x < 5:
    y = x
    stop = True
elif x % 2 == 0:
    y = x // 2
    h = h - 1
else:
    y = (3*x+1) // 2
    h = h + 1
```

Ne pas oublier les deux points à la fin des lignes `if, elif, else`. Les instructions à exécuter dans chacun des cas doivent être *indentées* et rigoureusement alignées verticalement (ici il y a quatre blancs au début de chaque instruction indentée) — en fait l'utilisateur est aidé dans cette tâche par le logiciel de programmation, qui insère les blancs à sa place. On peut avoir plusieurs `elif` (abréviation de *else if*) ; inversement `elif` et `else` ne sont pas obligatoires : leur absence indique qu'il n'y a rien à faire dans ces cas.

Exercice 1.3.1 Donner les valeurs des variables $h, y, stop$ après exécution de l'exemple, pour chacune des valeurs de x comprises entre 1 et 8.

Même question si on supprime la ligne `elif` et les deux instructions suivantes. Idem si on supprime la ligne `else` et les deux instructions suivantes. Idem enfin en supprimant toutes les lignes à partir de `elif`.

Que se passe-t-il si l'instruction $h = h - 1$ n'est pas indentée ? Même question avec l'instruction finale $h = h + 1$.

Exercice 1.3.2 Ecrire et tester une fonction `compare(a,b)` qui retourne -1 si $a < b$, 0 si $a = b$, et 1 si $a > b$.

Exercice 1.3.3 Écrire une fonction `max2(x,y)` qui calcule le maximum de deux nombres x et y. Attention : bien appeler cette fonction *max2*, car la fonction *max* est prédéfinie en Python.

Exercice 1.3.4 Écrire une fonction `max3(x,y,z)` qui calcule le maximum de trois nombres x, y, z. Donner plusieurs versions de cette fonction dont une utilise la fonction `max2`.

Exercice 1.3.5 Écrire une fonction `uneMinuteEnPlus` qui calcule l'heure une minute après celle passée en paramètre sous forme de deux entiers que l'on suppose cohérents. Exemples :

- `uneMinuteEnPlus(14,32)` vaut (14, 33).
- `uneMinuteEnPlus(14,59)` vaut (15, 0).

Ne pas oublier le cas de minuit.

Exercice 1.3.6 Le service de reprographie propose les photocopies avec le tarif suivant : les 10 premières coûtent 20 centimes l'unité, les 20 suivantes coûtent 15 centimes l'unité et au-delà de 30 le coût est de 10 centimes. Écrire une fonction `coutPhotocopies(n)` qui calcule le prix à payer pour n photocopies.

1.4 Listes et boucles *for*

Python permet de manipuler les listes d'éléments, par exemple `[2,8,5]` est une liste d'entiers. On peut *tester* si un élément appartient à une liste, par exemple `8 in [2,8,5]` vaut *True*, tandis que `4 in [2,8,5]` vaut *False*. La fonction `len` (abréviation de *length*) retourne la *longueur* d'une liste (on dit aussi sa *taille*), par exemple `len([2,8,5])` vaut 3.

La boucle `for` permet de *parcourir* les éléments d'une liste ; ne pas oublier les deux points à la fin de la ligne, et les lignes suivantes doivent être indentées (décalées). La fonction `range` permet de construire et de parcourir des suites d'éléments consécutifs.

Exercice 1.4.1 Entrer l'instruction suivante et analyser ce qui est affiché :

```
for i in [3, 8, 5]:
    i, i * i
```

Entrer les deux instructions suivantes et analyser les réponses de Python :

```
range (10)
list (range (10))
```

Continuer de même avec les instructions suivantes (attention à bien apppuyer deux fois sur *Entrée* pour déclencher l'exécution d'une boucle) :

```
for j in range(10):
    j, j * j

list (range (3, 8))

for k in range(3, 8):
    k, 2*k + 1
    2 ** k
```

Les variables i, j, k sont-elles définies après exécution de ces instructions ? Si oui quelles sont leurs valeurs ?

Exercice 1.4.2 Tester chacune des instructions suivantes et analyser les résultats comme dans l'exercice précédent :

```
list (range(2, 20, 3))
u = [2, 6, 1, 10, 6]
for x in u:
    for y in u:
        print (x, y, x + y, x == y)

semaine = ["lundi", "mardi", "mercredi", "jeudi", "vendredi"]
for jour in semaine:
    print (jour, end = '␣')
```

Exercice 1.4.3 On considère la définition suivante :

```
def sigma (n):
    s = 0
    for i in range (1, n+1):
        s = s + i
    return s
```

Quelles sont les valeurs de `sigma(3)`, de `sigma(5)` et de `sigma(n)` ? Connaissez-vous une formule qui permet de calculer le même résultat ?

Que calcule la fonction `sigma` si par malheur on indente trop la dernière ligne, comme ci-dessous ?

```
def sigma (n):
    s = 0
    for i in range (1, n+1):
        s = s + i
        return s    # gros bug !
```

Exercice 1.4.4 Écrire une fonction `sommeCarres(n)` qui calcule $\sum_{i=1}^{n} i^2$.

Exercice 1.4.5 La fonction factorielle peut être définie de la manière suivante :

$$\begin{cases} 0! & = 1, \\ n! & = 1 \times 2 \times \cdots \times n = \prod_{k=1}^{n} k \text{ pour } n \geq 1. \end{cases}$$

Écrire une fonction `factorielle(n)` qui utilise cette définition pour calculer $n!$; tester en affichant les factorielles des nombres de 0 à 16.

1.5 Boucles *while*

La boucle `while` (tant que) permet de répéter une partie de programme tant qu'une condition est vraie.

Exercice 1.5.1 On considère la fonction suivante :

```
def mystere (n):
    c = []
    while n > 0:
        c = [n % 2] + c
        n = n // 2
    return c
```

La variable `c` contient une *liste*, vide au départ (première instruction), et l'opérateur `+`, appliqué à des *listes*, les *concatène*. Par exemple :

$$[1,1,0] + [1,0] = [1,1,0,1,0]$$

Attention : cet usage de l'opérateur `+` n'est pas commutatif (la concaténation n'est pas commutative), par exemple :

$$[1,0] + [1,1,0] = [1,0,1,1,0]$$

Effectuer le calcul de `mystere (43)`. De façon générale, que calcule cette fonction ?

Exercice 1.5.2 Écrire une fonction `logarithme(a, n)` qui retourne le plus petit entier k tel que $a^k > n$ (on supposera $a > 1$). *Note :* Python sait calculer a^k (notation `a**k`), mais il existe une solution simple et (légèrement) plus efficace qui utilise seulement la multiplication.

Comment modifier cette fonction pour qu'elle calcule le plus grand entier k tel que $a^k \leq n$ (c'est la définition habituelle du logarithme *entier*) ?

Exercice 1.5.3 (facultatif, hors programme) L'objectif est d'écrire un *test de primalité*, c'est à dire une fonction `premier(n)` qui retourne *True* si l'entier naturel $n > 1$ est premier et *False* sinon. Pour cela on parcourt les nombres supérieurs à 2 pour chercher un diviseur d de n : dès qu'on en trouve un on arrête les calculs, n n'est pas premier.

1. Si on ne trouve pas de diviseur $d \leq \sqrt{n}$, on est sûr que n est premier : pourquoi ? Comment effectuer le test $d \leq \sqrt{n}$ sans utiliser de fonction "racine carrée" ?

2. Ecrire la fonction `premier(n)`.

3. Tester cette fonction en affichant tous les nombres premiers inférieurs à 1000.

4. Améliorer `premier(n)` en traitant à part le cas où n est pair ; dans le cas où n est impair, il suffit ensuite de chercher un diviseur d impair.

Exercice 1.5.4 On suppose dans cet exercice qu'on dispose de la fonction `premier(n)` décrite dans l'exercice précédent. Le code de cette fonction est sans importance, on suppose seulement que son temps de calcul est proportionnel à $\sqrt{n}$ lorsque n est premier.

1. Ecrire une fonction `premierSuivant(n)` qui calcule le plus petit nombre premier $p > n$.

2. Sachant que le calcul de `premierSuivant(10**12)` prend environ une seconde, quel est l'ordre de grandeur maximal de n pour que le calcul de `premierSuivant(n)` dure moins d'une minute ? moins d'une heure ? moins d'une journée ?

   Quelle est la durée approximative du calcul de `premierSuivant(2**40)` ? Même question pour 2^{50}.

   Note : pour le traitement de cette question on suppose que le temps de calcul de `premier(n)` est négligeable lorsque n n'est pas premier, ce qui est très souvent le cas (la plupart des entiers possèdent un petit facteur, découvert très vite lors de l'exécution de `premier(n)`).

Remarque : il existe des tests de primalité sophistiqués *radicalement* plus efficaces que le test naïf décrit dans l'exercice 1.5.3 ; ils permettent de tester en quelques secondes la primalité d'un nombre dont l'écriture décimale comporte plusieurs centaines de chiffres.

1.6 Suites récurrentes

Exercice 1.6.1 Une suite de Syracuse est définie par récurrence par

$$\begin{cases} u_0 & = a \\ u_{n+1} & = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair,} \\ (3u_n+1)/2 & \text{sinon.} \end{cases} \end{cases}$$

où a est un entier naturel strictement positif.

1. Calculer les premiers termes de la suite pour $a = 5$ puis pour $a = 7$.
2. Ecrire une fonction `syracuse(a, n)` qui calcule le terme de rang n de cette suite lorsque le premier terme u_0 est égal à a. Tester cette fonction pour $a = 5$ puis pour $a = 7$.
3. Que se passe-t-il lorsque pour une valeur de n, u_n est égal à 1 ?
4. On conjecture (consulter par exemple Wikipedia) qu'une suite de Syracuse finit toujours par atteindre la valeur 1. Écrire une fonction `longueur(a)` qui calcule la première valeur de n telle que $u_n = 1$ lorsque le premier terme u_0 vaut a.
5. Vérifier la conjecture pour tous les entiers $a < 100$. Parmi ces valeurs de a, quelle est celle qui fournit une suite de longueur maximale ?
6. Utiliser la fonction *syracuse* de la question 2 pour écrire la fonction *longueur* de la question 4 est une idée naturelle. Etudier dans ce cas combien de fois on exécute le calcul :

   $$u_{n+1} = u_n/2 \quad \text{si } u_n \text{ est pair,} \quad u_{n+1} = (3u_n+1)/2 \quad \text{sinon,}$$

   pour calculer longueur(27). Améliorer le code de la fonction `longueur(a)` pour éviter les calculs inutiles.
7. Ecrire la fonction `listeSyracuse(a)` qui calcule la *liste*

   $$[u_0 = a, u_1, u_2, ..., u_n = 1]$$

   où u_n désigne le premier terme égal à 1. Par exemple, avec $a = 7$ on obtient la liste [7, 11, 17, 26, 13, 20, 10, 5, 8, 4, 2, 1].
8. Écrire une fonction `hauteur(a)` qui calcule la valeur maximale de u_n lorsque le premier terme u_0 vaut a ; par exemple `hauteur(7)` vaut 26.

Chapitre 2. Sommets d'un graphe

Un graphe est une modélisation d'un ensemble d'objets reliés : les objets sont appelés *noeuds*, et les liens sont appelés *arêtes*.

Python permet de définir la *classe* des graphes, ainsi que les autres classes (noeuds et arêtes) nécessaires. Ces définitions se trouvent dans le module `graphV3` écrit par des enseignants de MSI102 ; ce module comporte aussi une vingtaine de fonctions qui permettent de manipuler graphes, sommets et arêtes sans connaître les détails des classes correspondantes. Un autre module `bibV3` contient un certain nombre de graphes construits en utilisant `graphV3` ; pour utiliser un module il faut commencer par l'*importer*, et toute session de travail sur les graphes doit commencer par la phrase magique :

```
from bibV3 import *
```

Le module `bibV3` importe à son tour le module `graphV3` ; les noms de ces modules se terminent par "V3" pour indiquer qu'ils sont adaptés à la Version 3 du langage Python.

Voici l'un des graphes inclus dans le module `bibV3` : les sommets sont quelques grandes villes françaises, et une arête entre deux villes indique que des rames TGV circulent entre elles. Le suffixe 2005 indique que ce graphe correspond à la situation cette année-là : Strasbourg est un sommet isolé, car la ligne TGV Est n'est pas encore en service. De toute façon ces détails pratiques sont sans importance (de même que la position géographique étrange de Nantes), la seule chose à savoir est que le graphe `tgv2005` comporte les 9 sommets et les 20 arêtes de la figure ci-contre.

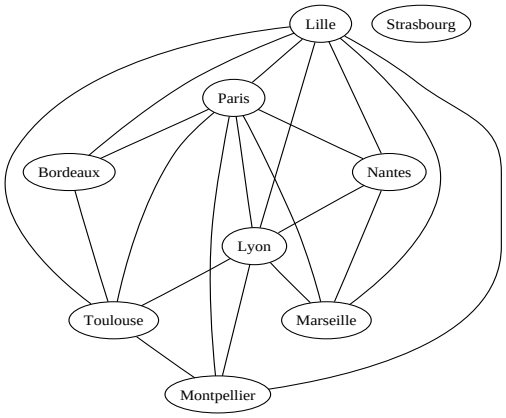


FIGURE 2.1: Le graphe `tgv2005`

Si pendant une session Python on tape simplement `tgv2005`, ou `print(tgv2005)`, l'interprète (en jargon : *Python Shell*) répond : `<graphe: 'tgv2005'>` pour indiquer que cet *objet* est un graphe. Pour désigner le sommet *Bordeaux* on pourrait croire qu'il suffit de taper `Bordeaux`, hélas l'interprète répond rouge de colère : `NameError: name 'Bordeaux' is not defined.`

En effet seuls les noms des graphes sont définis dans le module `bibV3` — les sommets sont beaucoup trop nombreux. On peut obtenir la liste complète des sommets d'un graphe G avec la fonction `listeSommets(G)`, par exemple en tapant `listeSommets(tgv2005)` on obtient :

```
[<sommet: 'Lille', None, False>, <sommet: 'Paris', None, False>, ...
 <sommet: 'Bordeaux', None, False>, ... <sommet: 'Strasbourg', None, False>]
```

Chaque sommet est affiché avec confirmation de sa classe, suivie du nom du sommet et de sa couleur — pour l'instant *None*, qui signifie que le sommet n'est pas coloré ; la dernière composante est booléenne et indique si le sommet est marqué ou non : ces marques seront utilisées section 2.4, pour l'instant *False* est affiché partout car aucun sommet n'est marqué.

Le nom d'un sommet, par exemple `'Bordeaux'`, est une simple chaîne de caractères utilisée pour identifier ce sommet à l'intérieur du graphe lorsqu'on affiche le sommet ou lorsqu'on

dessine le graphe. Comme on l'a expliqué ci-dessus, ce n'est pas le nom d'un objet au sens du langage Python, et lorsqu'il faut distinguer les deux on dira que `'Bordeaux'` est une *étiquette*. Il faut utiliser la fonction `sommetNom(G,etiquette)` pour accéder au sommet par son étiquette : `sommetNom (tgv2005, 'Bordeaux')` est une expression correcte pour désigner ce sommet, et on peut ensuite nommer cet objet au sens du langage Python en utilisant une affectation : `bx = sommetNom (tgv2005, 'Bordeaux')`. Pour bien distinguer les deux nous avons choisi ici un nom d'objet `bx` distinct de l'étiquette *Bordeaux*, mais souvent on utilise le même symbole — un lecteur attentif aura noté que l'étiquette du graphe `tgv2005` est ... `'tgv2005'`.

`listeSommets(G)`	retourne la *liste* des *sommets* de G
`nbSommets(G)`	retourne le *nombre* de sommets de G, c'est-à-dire la *taille* de la liste précédente
`sommetNom(G,etiquette)`	retourne le *sommet* de G désigné par son *nom* (étiquette), par exemple : `sommetNom (tgv2005, 'Bordeaux')`

2.1 Echauffement : coloriages

Un sommet s peut être colorié avec une couleur c par la fonction `colorierSommet(s,c)`, où c est une simple chaîne de caractères. La fonction `dessiner(G)` tient compte des couleurs des sommets si celles-ci font partie d'une liste prédéfinie ; par exemple `'red', 'green', 'blue'` sont des couleurs reconnues par le programme de dessin, et la liste complète se trouve à l'adresse http://www.graphviz.org/doc/info/colors.html — vous y trouverez entre autres `'chocolate'`, `'orange'` et `'tomato'` !

`colorierSommet(s,c)`	colorie le *sommet* s avec la *couleur* c (par exemple `'red'`)
`couleurSommet(s)`	retourne la *couleur* du *sommet* s.
`dessiner(G)`	dessine le *graphe* G

Exercice 2.1.1

1. La fonction `degre(s)` fournit le degré d'un sommet s. Ecrire l'expression Python qui fournit le degré du sommet *Paris* dans le graphe `tgv2005`.
2. Ecrire l'instruction Python qui colorie en vert ce sommet.

Exercice 2.1.2 Écrire une fonction `toutColorier(G,c)` qui colorie tous les sommets du graphe G avec la couleur c. Utiliser cette fonction pour colorier en rouge tous les sommets du graphe `tgv2005` ; vérifier le résultat de deux façons :

a) en affichant la liste des sommets du graphe,

b) en dessinant le graphe.

Un sommet non coloré a pour couleur *None* (attention, ce n'est pas une chaîne de caractères). Ecrire l'instruction qui permet d'annuler l'opération précédente ; vérifier que les sommets du graphe `tgv2005` sont bien décolorés.

Exercice 2.1.3 Écrire une fonction `existeCouleur(G,c)` qui renvoie *True* s'il existe au moins un sommet de couleur c dans le graphe G et *False* sinon.

Exercice 2.1.4

1. Écrire une fonction `nbSommetsCouleur(G,c)` qui compte les sommets du graphe G qui ont la couleur c.

2. Écrire une fonction `nbSommetsColores(G)` qui compte les sommets colorés de G (c'est-à-dire les sommets qui ont une couleur différente de *None*) en *utilisant* la fonction précédente et la fonction `nbSommets(G)`.

2.2 Voisins

Deux sommets s et t sont appelés *voisins* s'il existe une arête e ayant s et t comme extrémités ; on dit que l'arête e est *incidente* à chacun des sommets s et t.

Par exemple, les sommets A et B du graphe ci-contre sont voisins, ainsi que A et C, tandis que les sommets A et D ne sont pas voisins. Il peut exister plusieurs arêtes entre deux sommets, par exemple ici entre A et B. Un sommet peut aussi être voisin de lui-même, si les deux extrémités d'une arête sont confondues — dans ce cas on dit que l'arête est une *boucle*. Sur le graphe ci-contre, il y a une boucle autour du sommet B et deux boucles autour du sommet D.

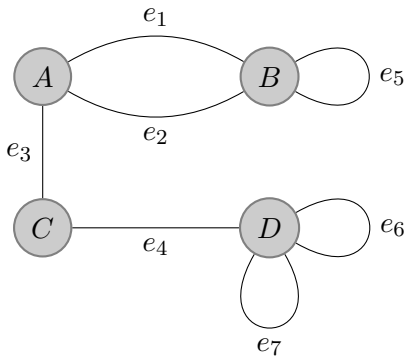


FIGURE 2.2: un graphe avec une arête double et trois boucles

La fonction `listeVoisins(s)` retourne la liste des voisins du sommet s, obtenue en suivant chacune des arêtes incidentes. Un même sommet peut se retrouver plusieurs fois dans cette liste : par exemple B apparaît deux fois dans la liste des voisins de A. Une boucle autour du sommet s est par convention deux fois incidente à s : la liste des voisins de B contient deux fois le sommet B lui-même, à cause de la boucle autour de B.

Le *degré* d'un sommet s est le nombre d'arêtes incidentes, et la fonction `degre(s)` calcule sa valeur ; c'est aussi le nombre de *brins* issus de s lorsqu'on regarde le dessin — une boucle compte pour deux dans le calcul du degré. *Attention* : jamais de lettre accentuée dans le nom d'une *fonction Python*.

`listeVoisins(s)`	retourne la *liste* des *voisins* du sommet s
`degre(s)`	retourne le *degré* du sommet s, qui est aussi la *taille* de la liste précédente

Exercice 2.2.1 Pour chaque sommet du graphe de la figure 2.2, donner son degré et écrire la liste de ses voisins (leur ordre dans la liste est sans importance). Écrire l'instruction Python qui permet de vérifier le résultat en TP (ce graphe est disponible sous le nom `fig22`).

Exercice 2.2.2 Comment utiliser la fonction `listeVoisins` pour obtenir la liste des villes voisines de *Nantes* dans le graphe `tgv2005` ? *Note* : en TP on peut améliorer l'affichage de ce genre de liste en utilisant la fonction `nomSommet(s)` qui retourne ... le nom du sommet s, c'est-à-dire son étiquette ; à ne pas confondre avec la fonction *sommetNom*, qui elle retourne un sommet d'un graphe à partir de son étiquette.

Exercice 2.2.3 Écrire une fonction `colorierVoisins(s,c)` qui colorie tous les voisins du sommet s avec la couleur c. Utiliser cette fonction pour colorier en vert les voisins de *Bordeaux* dans le graphe `tgv2005`. Vérifier le résultat de deux façons : afficher la liste des sommets du graphe, et le dessiner.

Exercice 2.2.4 Écrire une fonction `sontVoisins(s1,s2)` qui teste si les sommets $s1$ et $s2$ sont voisins, c'est-à-dire qui renvoie *True* si c'est le cas et *False* sinon.

Tester cette fonction sur tous les couples de sommets du graphe `fig22`.

Exercice 2.2.5 Ecrire une fonction `listeVoisinsCommuns (s1,s2)` qui calcule la liste des voisins communs à deux sommets $s1$ et $s2$.

2.3 Boucles

On testera les fonctions de cette section sur le graphe de la figure 2.2, disponible sous le nom `fig22`.

Exercice 2.3.1 Écrire une fonction `existeBoucle(s)` qui teste si le sommet s possède une boucle incidente, c'est-à-dire une arête qui relie le sommet s à lui-même — on dit aussi dans ce cas qu'il existe « une boucle autour de s ».

Par exemple sur le graphe de la figure 2.2, la fonction doit retourner *True* pour les sommets B et D, et *False* pour les sommets A et C.

Exercice 2.3.2 Écrire une fonction `nbBoucles(s)` qui compte le nombre de boucles autour d'un sommet s.

Exercice 2.3.3 Écrire une fonction `sansBoucle(G)` qui teste si un graphe G est sans boucle. Appliquer cette fonction à tous les graphes de la liste `graphes`.

2.4 Graphes simples

Un graphe est dit *simple* s'il n'a ni boucle, ni arête multiple ; autrement dit les extrémités d'une arête sont toujours distinctes, et il existe au plus une arête qui relie deux sommets s et t distincts. Par exemple le graphe `tgv2005` est simple, et celui de la figure 2.2 ne l'est pas ; les graphes simples sont de loin les plus fréquents en pratique.

Dans un graphe simple la liste des voisins d'un sommet s ne comporte jamais de répétition. On se propose d'utiliser cette propriété caractéristique pour écrire une fonction `estSimple(G)` qui teste si le graphe G est simple.

Une méthode efficace pour tester si tous les sommets d'une liste sont distincts est de marquer chaque sommet en parcourant la liste ; si l'on rencontre un sommet déjà marqué pendant ce parcours on sait que ce sommet est présent plusieurs fois dans la liste.

Cette méthode comporte un piège, car un sommet marqué le reste ! En appliquant deux fois l'algorithme à la même liste on va trouver, lors de la seconde passe, que le premier sommet est marqué, et on va en déduire bêtement qu'il s'agit d'un sommet répété. Il faut donc commencer par démarquer tous les sommets avant d'appliquer l'algorithme. Les fonctions disponibles pour manipuler les marques sur les sommets sont :

`marquerSommet(s)`	marque le sommet s
`demarquerSommet(s)`	démarque le sommet s
`estMarqueSommet(s)`	retourne `True` si s est marqué, `False` sinon

Exercice 2.4.1

1. Écrire une fonction `demarquerVoisins(s)` qui démarque tous les voisins du sommet s.

2. Écrire une fonction `voisinsDistincts(s)` qui teste si tous les voisins du sommet s sont distincts — le résultat est *True* ou *False*. Tester la fonction sur chaque sommet s du graphe `fig22`, et afficher la liste des voisins de s après chaque test pour repérer les sommets marqués et vérifier que ce sont bien les sommets prévus.

3. Utiliser les fonctions précédentes pour écrire la fonction `estSimple(G)` qui teste si un graphe G est simple.

Tester la fonction `estSimple(G)` sur tous les graphes de la liste `graphes`. Après exécution du test sur un graphe G, dessiner G et/ou afficher la liste de ses sommets pour repérer ceux qui sont marqués ; interpréter le résultat, en particulier pour les graphes simples.

Chapitre 3. Degrés

On rappelle que le *degré d'un sommet* est le nombre d'arêtes incidentes à ce sommet. Une boucle compte pour deux dans le degré. Par exemple dans le graphe de la figure 2.2 (reproduite ci-dessous) :

- le degré de A est 3, car A est extrémité de e_1, e_2 et e_3 ;
- le degré de B est 4, car B est extrémité de e_1, e_2 et e_5 (qui contribue pour 2 unités au degré, car c'est une boucle autour de B).

3.1 Calculs sur les degrés, sommets isolés

Exercice 3.1.1 Soit la fonction suivante :

```
def mystere(G):
    n = nbSommets(G)
    x = 0
    for s in listeSommets(G):
        x = x + degre(s)
    return x/n
```

1. Quelle est la valeur calculée si G est le graphe de la figure ci-contre ?
2. Que calcule la fonction `mystere` dans le cas général ?

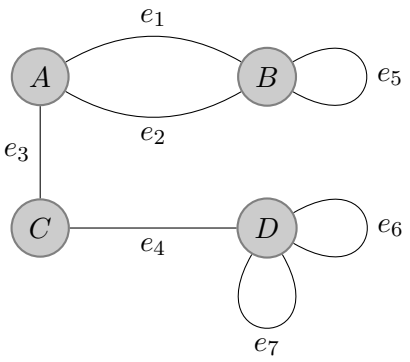


Figure 2.2 (copie)

Exercice 3.1.2

1. Écrire une fonction `degreMax(G)` qui calcule le degré maximum des sommets d'un graphe G.
2. Écrire une fonction `degreMin(G)` qui calcule le degré minimum des sommets d'un graphe G. Pourquoi cette question est-elle un peu plus difficile que la précédente ?

Note : la fonction `sommetNumero(G,i)` peut être utile pour la seconde question de cet exercice, cf. aide-mémoire page 32 en fin de fascicule.

Exercice 3.1.3 Écrire une fonction `nbSommetsDegre(G,d)` qui calcule le nombre de sommets du graphe G ayant pour degré d.

Exercice 3.1.4 On dit qu'un sommet est *isolé* s'il n'a aucune arête incidente. Écrire une fonction `existeIsole(G)` qui teste si un graphe G a au moins un sommet isolé.

3.2 Graphes simples

Rappel : un graphe est *simple* s'il n'a ni boucle, ni arête multiple.

Exercice 3.2.1

1. Essayer de construire un graphe simple ayant 4 sommets, et tel que les degrés de sommets soient tous distincts. Que peut-on en déduire ?

2. On se propose de démontrer par l'absurde qu'il n'existe pas de graphe simple de n sommets ($n \geq 2$) tel que tous ses sommets soient de degrés distincts. Supposons qu'un tel graphe G existe.

   a) Montrer que G ne peut comporter de sommet de degré supérieur ou égal à n.

   b) En déduire les degrés possibles pour les n sommets.

   c) Montrer que ceci entraîne une absurdité (existence simultanée d'un sommet de degré 0 et d'un sommet de degré $n-1$).

3. Application : peut-on dire que dans un groupe de n personnes, il y en a toujours deux qui ont le même nombre d'amis présents ?

3.3 Formule des poignées de mains

Exercice 3.3.1 Un graphe est dit *cubique* si tous ses sommets sont de degré 3.

1. Dessiner un graphe cubique ayant 4 sommets ; même question avec 3 sommets, 5 sommets. Que constate-t-on ?

2. Quel est le nombre d'arêtes d'un graphe cubique de n sommets ?

3. En déduire qu'un graphe cubique a un nombre pair de sommets.

Exercice 3.3.2 Écrire une fonction `cubique(G)` qui teste si le graphe G est cubique.

Exercice 3.3.3 On dispose de 235 machines, qu'on souhaite relier en réseau. Est-il possible de relier chaque machine à exactement 17 autres machines ? Justifier la réponse, soit en expliquant comment relier les machines, soit en expliquant pourquoi ce n'est pas possible.

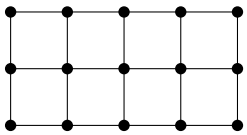
Exercice 3.3.4

1. Dessiner un graphe à 6 sommets tel que la liste des degrés des sommets soit :

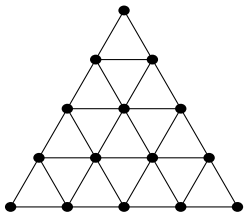
   `[2, 1, 4, 0, 2, 1]`.

2. Même question avec `[2, 1, 3, 0, 2, 1]`.

Exercice 3.3.5 Une *grille* (rectangulaire) $m \times n$ est constituée de m lignes horizontales et n lignes verticales, dont les croisements forment les sommets du graphe ; une grille 3×5 est représentée ci-contre.



1. Compter les sommets de degré 2 (respectivement 3 et 4) et en déduire la somme des degrés des sommets en fonction de m et n.
2. Compter les arêtes et comparer avec le résultat de la question précédente.



Exercice 3.3.6 La grille *triangulaire* T_5 est représentée ci-contre. Adapter les calculs de l'exercice précédent au cas de T_n.

Exercice 3.3.7 Écrire une fonction `nbAretes(G)` qui calcule le nombre d'arêtes d'un graphe G.

Dès qu'on l'aura comprise on notera dans l'encadré ci-dessous la formule générale des poignées de mains, qui fait partie des résultats à mémoriser :

Chapitre 4. Chaînes et connexité

Une *chaîne* dans un graphe est une suite alternée de sommets et d'arêtes :

$$[s_0, a_1, s_1, a_2, s_2 \dots a_n, s_n]$$

où l'arête a_i relie le sommet s_{i-1} qui la précède et le sommet s_i qui la suit ; on dit que cette chaîne relie les sommets s_0 et s_n, ou qu'elle représente un *chemin* de s_0 vers s_n.

Un *cycle* est une chaîne dont les deux extrémités coïncident ($s_0 = s_n$) ; on peut choisir n'importe quel sommet du cycle comme sommet de départ.

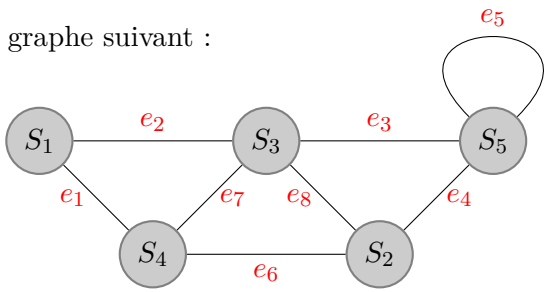
Le nombre d'arêtes n est la *longueur* de la chaîne (ou du cycle).

Une chaîne est *simple* si tous ses sommets sont *distincts* ; dans ce cas toutes ses arêtes sont distinctes — expliquer pourquoi. Dans le cas d'un cycle simple il faut adapter la définition : les sommets sont tous distincts, *sauf* les deux extrémités (qui par définition coïncident) ; les arêtes doivent rester distinctes, et il faut donc éliminer les allers-retours de la forme $[s, a, t, a, s]$ (où a désigne une arête qui relie les sommets s et t).

Un graphe est *connexe* si, par définition : *pour tous sommets s et t, il existe une chaîne qui relie s et t.*

4.1 Echauffement

Exercice 4.1.1 Soit le graphe suivant :

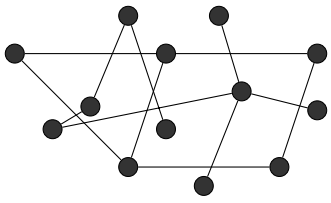


1. Donner plusieurs exemples de chaînes simples entre S_1 et S_5.
2. Donner un exemple de chaîne non simple entre S_1 et S_5.
3. Donner les cycles de longueur 4 de ce graphe.

Exercice 4.1.2 Soit Γ une chaîne d'extrémités s et t ; on suppose que Γ *n'est pas* simple :

1. montrer que Γ contient un cycle ;
2. en déduire qu'il existe une chaîne plus courte que Γ qui relie s et t ;
3. en déduire qu'il existe une chaîne *simple* qui relie s et t.

Exercice 4.1.3 Le graphe suivant est-il connexe ?



Exercice 4.1.4 Indiquer si les propositions suivantes sont vraies ou fausses en justifiant votre réponse :

1. Si un graphe n'a pas de sommet isolé, alors il est connexe.
2. Si un graphe possède un sommet isolé, alors il n'est pas connexe.
3. Pour qu'un graphe soit connexe il est nécessaire que tous ses sommets soient de degré supérieur ou égal à 1.

Exercice 4.1.5

1. La proposition suivante :

   « un graphe est connexe s'il existe un sommet s_0 qui peut être relié (par des chaînes) à tous les autres sommets »

   est-elle vraie ? Justifier ou donner un contre-exemple.

2. Même question pour la proposition suivante :

   « un graphe est connexe si et seulement si il existe une chaîne passant (au moins une fois) par chaque sommet du graphe ».

4.2 Arbres

Théorème 4.2 *Un graphe connexe avec n sommets possède au moins $n-1$ arêtes.*

Exercice 4.2.1 Peut-on en déduire les propositions suivantes ?

1. Si un graphe de n sommets a plus de n arêtes, alors il n'est pas connexe.
2. Si un graphe de n sommets a plus de n arêtes, alors il est connexe.
3. Si un graphe de n sommets a $n-2$ arêtes, alors il n'est pas connexe.

Définitions

Un graphe connexe avec n sommets et $n-1$ arêtes est appelé un *arbre* ; un sommet de degré 1 dans un arbre est appelé une *feuille*.

Exercice 4.2.2

1. Dessiner tous les arbres avec 1, 2, 3 ou 4 sommets.
2. Montrer qu'un arbre possède au moins une feuille (et même deux).
3. Montrer qu'un arbre privé d'une feuille (et de l'arête issue de cette feuille) reste un arbre.
4. Dessiner tous les arbres avec 5 sommets.
5. Dessiner tous les arbres avec 6 sommets (il y en a six).

Exercice 4.2.3 On se propose de montrer le théorème 4.2 par induction sur le nombre de sommets.

1. Vérifier que le théorème est vrai pour tout graphe à un sommet — *trop cool*.

2. Induction : partant d'un graphe à $n+1$ sommets, on construit un ou plusieurs graphes à moins de n sommets pour appliquer l'hypothèse d'induction. On propose deux méthodes :

   a) $1^{\text{ère}}$ méthode : supprimer un sommet et toutes les arêtes qui lui sont incidentes. Dans ce cas, le graphe obtenu n'est plus nécessairement connexe. Donner un exemple où cela se produit. Appliquer l'hypothèse d'induction sur chacune des composantes connexes du graphe obtenu pour conclure.

   b) $2^{\text{ème}}$ méthode : utiliser la contraction d'une arête. Vérifier que le graphe obtenu reste connexe, et calculer son nombre de sommets et son nombre d'arêtes pour conclure.

Exercice 4.2.4 Dans cet exercice un *cycle* désigne un cycle *simple*.

1. Soit G un graphe connexe possédant un cycle Γ : montrer que la suppression de n'importe quelle arête de Γ laisse le graphe connexe.

2. En déduire qu'un arbre ne possède pas de cycle.

3. En déduire que pour tous sommets s et t d'un arbre il existe une seule chaîne *simple* qui relie s et t.

Note : chacune des deux dernières propriétés caractérise en fait les arbres parmi les graphes connexes, et sert souvent de définition pour la notion d'arbre.

4.3 Algorithmes

Accessibilité

On dit que le sommet t est *accessible* à partir du sommet s s'il existe une chaîne reliant s et t. Il est facile de construire progressivement l'ensemble A des sommets accessibles depuis s :

1. initialisation : $A = \{s\}$;

2. tant qu'il existe un sommet $x \notin A$ qui est voisin d'un sommet de A, ajouter x à A.

Pour savoir si le sommet t est accessible depuis s on stoppe la construction de A dès qu'à l'étape 2 on a $x = t$, et dans ce cas la réponse est affirmative ; par contre si cet événement $(x = t)$ ne se produit pas avant la fin de la construction de A (il n'existe plus de sommet en dehors de A qui soit voisin d'un sommet de A), la réponse est négative. Dans ce dernier cas A est une composante connexe du graphe G, et t appartient à une autre composante connexe.

Exercice 4.3.1 On travaille sur le graphe G de l'exercice 4.1.1 : appliquer l'algorithme de construction de l'ensemble A des sommets accessibles à partir de S_1 en choisissant toujours le sommet de plus petit numéro à l'étape 2. Recommencer en faisant varier le sommet de départ.

Connexité

Pour savoir si le graphe G est connexe, on choisit un sommet s, on construit l'ensemble A des sommets accessibles depuis s, puis on teste si tous les sommets de G appartiennent à A.

Exercice 4.3.2 Montrer que le résultat de cet algorithme ne dépend pas du choix du sommet initial s.

Programmation

Pour les exercices suivants il suffit de marquer les sommets accessibles depuis s au lieu de construire explicitement A. Cette méthode comporte un piège, car un sommet marqué le reste ! Les fonctions *accessible* ou *estConnexe* (voir ci-dessous) doivent commencer par démarquer tous les sommets du graphe, sinon les résultats deviennent imprévisibles lorsqu'on les applique plusieurs fois de suite au même graphe. Les fonctions disponibles pour manipuler les marques sur les sommets sont :

`marquerSommet(s)`	marque le sommet s
`demarquerSommet(s)`	démarque le sommet s
`estMarqueSommet(s)`	retourne `True` si s est marqué, `False` sinon

Exercice 4.3.3 Écrire les fonctions suivantes :

1. `toutDemarquer(G)` qui démarque tous les sommets du graphe G ;
2. `sommetAccessible(G)` qui retourne un sommet non marqué ayant un voisin marqué, et qui retourne *None* s'il n'existe pas de tel sommet ;
3. `accessible(G, sNom, tNom)` qui renvoie *True* si, dans le graphe G, le sommet dont le nom est `tNom` est accessible depuis le sommet de nom `sNom`, et *False* sinon. Tester cette dernière fonction sur le graphe `Europe`, avec les sommets de noms `"Belgique"` et `"Hongrie"`, puis `"Espagne"` et `"Angleterre"`. Après chaque test examiner quels sont les sommets marqués dans le graphe *Europe*.

Combien le graphe *Europe* possède-t-il de composantes connexes ? Pour chacune employer la fonction *accessible* de façon judicieuse afin de marquer tous les sommets de la composante ; vérifier en dessinant le graphe. *Note* : la Russie est absente du graphe, d'où une composante connexe surprenante — laquelle ?

Exercice 4.3.4 Ajouter les fonctions suivantes :

1. `sommetsTousMarques(G)` qui teste si tous les sommets du graphe G sont marqués ;
2. `estConnexe(G)` qui teste si le graphe G est connexe.

Arêtes

Pour traiter les arêtes le module de manipulation de graphes contient les fonctions suivantes :

`listeAretesIncidentes(s)`	retourne la liste des arêtes issues du sommet s
`sommetVoisin(s,a)`	retourne le voisin du sommet s en suivant l'arête a
`marquerArete(a)`	marque l'arête a
`demarquerArete(a)`	démarque l'arête a

Ainsi le fragment de code

```
for t in listeVoisins(s):
```

peut être remplacé par :

```
for a in listeAretesIncidentes(s):
    t = sommetVoisin(s,a)
```

ce qui est un peu plus compliqué, mais permet de traiter l'arête a qui relie les sommets s et t. *Note* : si le graphe n'est pas simple plusieurs arêtes incidentes à s peuvent mener au même voisin t, qui dans ce cas apparaît aussi plusieurs fois dans la liste des voisins de s.

Exercice 4.3.5 Modifier la fonction *sommetAccessible* de l'exercice précédent pour marquer l'arête qui relie ce sommet non marqué à son voisin marqué. Ne pas oublier de modifier aussi la fonction *toutDemarquer* pour démarquer les arêtes en même temps que les sommets.

Tester à nouveau la fonction *accessible* comme dans l'exercice 4.3.3, puis dessiner le graphe *Europe*. Les arêtes marquées apparaissent avec une épaisseur et une couleur différentes des arêtes non marquées : que remarque-t-on ? Comment repérer sur le dessin le chemin trouvé par l'algorithme entre le sommet de départ et le sommet d'arrivée (lorsqu'il existe) ? Le chemin est-il toujours le même lorsqu'on échange le sommet de départ et le sommet d'arrivée ? Pourquoi ?

Parcours aléatoire

Le module de manipulation de graphes contient la fonction :

`melange(u)`	retourne une copie de la liste u dont les éléments ont été permutés de façon aléatoire

Par exemple l'instruction :

```
for s in melange (listeSommets (G)):
```

parcourt la liste des sommets du graphe G dans un ordre aléatoire.

Exercice 4.3.6 Modifier la fonction *sommetAccessible* de l'exercice précédent pour que le résultat ne dépende plus de l'ordre dans lequel sont rangés les sommets du graphe, ni de l'ordre des arêtes incidentes à un sommet.

Tester à nouveau la fonction *accessible* comme dans l'exercice 4.3.3, puis dessiner le graphe *Europe* : le chemin entre la Belgique et la Hongrie (par exemple) devrait changer à chaque exécution de l'algorithme.

4.4 Complexité

Dans cette section on note S l'ensemble des sommets d'un graphe G, et A l'ensemble des arêtes ; $|S|$ désigne alors le nombre de sommets du graphe, et $|A|$ le nombre d'arêtes.

Exercice 4.4.1

1. Lorsqu'on exécute le code suivant :

```
for s in listeSommets(G):
    for t in listeVoisins(s):
        op(s,t)
```

   combien de fois l'opération $op(s,t)$ est-elle exécutée ?

2. En déduire que la complexité *au pire* de la fonction *sommetAccessible* de l'exercice 4.3.3 est *proportionnelle* à $|A|$. Note : les opérations de marquage des sommets (y compris les tests) sont des opérations élémentaires effectuées en temps constant.

3. Estimer de même la complexité (au pire) des fonctions *accessible* (exercice 4.3.3) et *estConnexe* (exercice 4.3.4).

   Remarque : il existe des méthodes plus sophistiquées pour programmer ces algorithmes, qui fournissent des fonctions de complexité (au pire) proportionnelle à $(|A| + |S|)$.

Chapitre 5. Graphes Eulériens

5.1 Définitions

Une chaîne est *Eulérienne* si elle passe par chaque arête une fois et une seule. Un graphe est *Eulérien* s'il possède une chaîne Eulérienne.

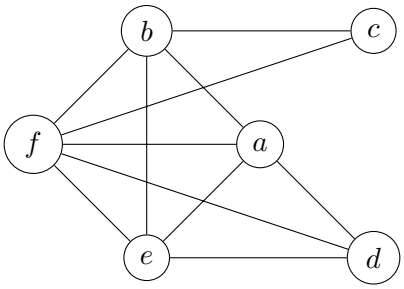
Un sommet *pair* (resp. *impair*) est un sommet de degré pair (resp. impair). *Rappel* : un *cycle* est une chaîne dont les deux extrémités sont égales.

Théorème 5.1 *Un graphe connexe G est* Eulérien *si et seulement si le nombre de sommets impairs vaut 0 ou 2 ; dans le premier cas G possède un cycle Eulérien, dans le second cas G possède une chaîne Eulérienne qui relie les deux sommets de degré impair.*

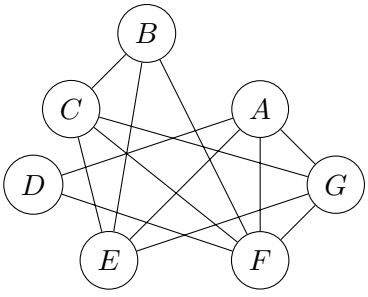
Exercice 5.1.1 Les propositions suivantes sont-elles vraies ou fausses ?

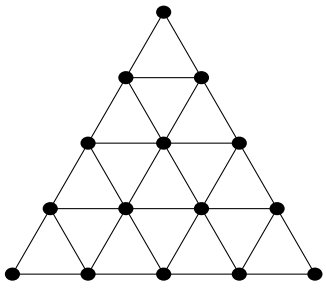
1. Tous les graphes Eulériens sont connexes.
2. Si un graphe n'est pas Eulérien, alors il n'est pas connexe.
3. Il existe des graphes connexes non Eulériens.

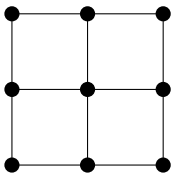
Exercice 5.1.2 Le graphe ci-dessous admet-il une chaîne Eulérienne ? Si oui, donner une telle chaîne, sinon, justifier. Admet-il un cycle Eulérien ?



Exercice 5.1.3 Parmi les 3 graphes suivants, lesquels sont Eulériens ?







Exercice 5.1.4 L'affirmation suivante :

> *« si un graphe G admet un cycle Eulérien alors toutes les chaînes Eulériennes de G sont des cycles »*

est-celle correcte ? Justifier ou donner un contre-exemple.

Exercice 5.1.5 Un graphe complet est un graphe où tout sommet est relié à chacun des autres par une arête. Pour quelles valeurs de n un graphe complet de n sommets (noté K_n) est-il Eulérien ?

Exercice 5.1.6 On considère un graphe G Eulérien dont les sommets sont A, B, C, D, E, F. On connaît une chaîne Eulérienne de ce graphe :

$$B, e_3, F, e_0, A, e_1, E, e_2, F, e_4, D, e_6, C, e_7, B, e_5, D.$$

1. Déduire de la chaîne Eulérienne le degré de chaque sommet.
2. Dessiner G sans oublier de porter les noms des sommets et des arêtes, et en évitant les croisements d'arêtes — c'est possible car G est *planaire*.

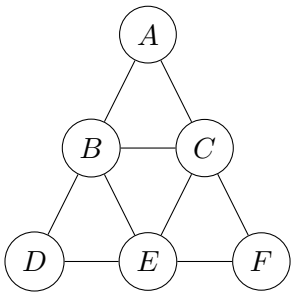
Exercice 5.1.7 Écrire une fonction `nbSommetsImpairs(G)` qui renvoie le nombre de sommets de degré impair du graphe G. Écrire une fonction `estEulerien(G)` qui teste si le graphe G est Eulérien ; on pourra utiliser la fonction *estConnexe* de l'exercice 4.3.4.

5.2 Construction d'un cycle Eulérien

Soit un graphe connexe G dont tous les sommets sont pairs, voici un algorithme qui construit un cycle Eulérien Γ :

1. choisir un sommet s arbitraire, et former la chaîne $\Gamma = \{s\}$;
2. tant que le dernier sommet de Γ possède une arête incidente a qui n'appartient pas à Γ, ajouter à la chaîne Γ l'arête a et son sommet extrémité ;
3. si toutes les arêtes de G sont dans Γ alors l'algorithme est terminé et Γ est un *cycle Eulérien* ;
4. sinon considérer le graphe G' constitué des arêtes qui ne sont pas dans Γ (et de leurs extrémités) ; choisir un sommet s' qui soit à la fois dans Γ et dans G' et construire dans G' un cycle Γ' d'origine s' avec le même algorithme ; enfin insérer Γ' dans Γ et retourner à l'étape précédente.

Exercice 5.2.1 Appliquer l'algorithme au graphe ci-contre, en choisissant toujours à l'étape 2 l'arête a dont l'extrémité porte le nom le plus petit dans l'ordre alphabétique, et en choisissant $s_0 = A$ à l'étape 1. Recommencer en faisant varier le point de départ s_0.



Exercice 5.2.2 Soit G un graphe quelconque auquel l'algorithme soit applicable — G est donc connexe et tous ses sommets sont pairs.

1. Montrer qu'à la fin de l'étape 2 la chaîne Γ est forcément un cycle.
2. Montrer qu'à l'étape 4 tous les sommets du graphe G' sont pairs, et qu'il existe toujours un sommet s' qui appartient à la fois à Γ et à G'.
3. Construire un exemple tel que le graphe G' obtenu à l'étape 4 ne soit pas connexe.

Le module de manipulation de graphes contient les fonctions suivantes :

- `sommetNumero(G,i)` calcule le sommet numéro i du graphe G — le premier sommet a pour numéro 0 ;
- `listeAretesIncidentes(s)` calcule la liste des arêtes issues du sommet s ;
- pour chaque arête a de la liste précédente, `sommetVoisin(s,a)` calcule le voisin du sommet s *par* l'arête a, c'est-à-dire le sommet t extrémité de l'arête a (qui relie donc s et t) ;
- `marquerArete(a)`, `demarquerArete(a)`, `estMarqueeArete(a)` — la dernière retourne *False* ou *True*.

Exercice 5.2.3

1. Écrire une fonction `toutDemarquer(G)` qui démarque toutes les *arêtes* de G.
2. Écrire une fonction `areteIncidenteNonMarquee(s)` qui retourne une arête non marquée d'extrémité s s'il en existe une — sinon la fonction retourne *None*.
3. Écrire une fonction `cycleSansIssue(s)` qui exécute les *deux premières étapes* de l'algorithme et retourne le cycle d'origine s ainsi construit. Pour cela marquer les arêtes au fur et à mesure qu'elles sont ajoutées à la chaîne Γ (représentée par une liste).

Tester cette fonction sur le premier sommet de la grille triangulaire T_5 — voir exercices 3.3.6 et 5.1.3 (graphe du milieu) — qu'on construira avec la fonction `construireTriangle(n)`. Avec de la chance le cycle calculé est Eulérien ; dans d'autres cas il manque des arêtes.

4. Combien d'arêtes le graphe T_5 possède-t-il ?
5. En déduire la taille d'un cycle Eulérien de T_5 (ne pas oublier qu'un cycle comporte alternativement des sommets et des arêtes).
6. Pour chaque choix du sommet de départ s afficher la taille de la liste u retournée par la fonction `cycleSansIssue(s)` — la fonction `len(u)` fournit la taille d'une liste u ; lorsqu'il manque des arêtes dessiner le graphe :
   a) les arêtes marquées apparaissent avec une épaisseur et une couleur différentes des arêtes non marquées : identifier le graphe G' (étape 4 de l'algorithme) et un sommet s' défini comme dans l'algorithme ;
   b) calculer `u2 = cycleSansIssue(s2)` *sans effacer le marquage* des arêtes de u — s2 désigne le sommet s' de la question précédente (s' n'est pas un nom valide pour Python) ;
   c) recommencer tant qu'il existe des arêtes non marquées !

Exercice 5.2.4 Utiliser la fonction *melange* comme dans l'exercice 4.3.6 pour modifier la fonction *areteIncidenteNonMarquee* afin de retourner une arête aléatoire parmi les arêtes incidentes non marquées.

Choisir un sommet arbitraire s du graphe T_5 (voir exercice précédent), par exemple le premier, et calculer plusieurs fois `cycleSansIssue(s)` ; après chaque calcul dessiner le graphe et identifier le graphe G' (étape 4 de l'algorithme). *Note* : ne pas oublier d'utiliser la fonction *toutDemarquer* avant de lancer un nouveau calcul.

Chapitre 6. Problèmes et algorithmes de coloration

6.1 Coloration d'un graphe. Nombre chromatique

Un graphe est dit *bien colorié* si deux sommets voisins ont toujours des couleurs différentes : dans ce chapitre toutes les colorations respectent cette contrainte. Un graphe ne peut pas être bien colorié s'il possède une boucle (il existerait alors un sommet voisin de lui-même), et le nombre d'arêtes qui relient deux sommets voisins est sans importance pour la coloration : on ne s'intéressera donc qu'aux graphes *simples*.

Le nombre minimal de couleurs nécessaires pour colorier un graphe G est appelé *nombre chromatique* de G.

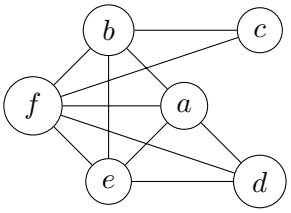
Dans les chapitres précédents on a vu des algorithmes efficaces pour tester si un graphe est connexe ou Eulérien, et pour construire des chemins dans des graphes ou des cycles Eulériens. Pour les problèmes de coloration la situation est différente : en général *on ne connaît pas* d'algorithme *efficace*, sauf pour le problème de coloration avec deux couleurs — voir section 6.2. En particulier on ne connaît pas d'algorithme qui, appliqué à un graphe G quelconque, calculerait rapidement son nombre chromatique ; on ne connait même pas de test efficace qui, appliqué à un graphe G quelconque, déterminerait si trois couleurs suffisent pour colorier G.

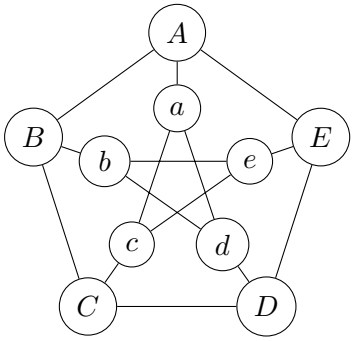
On conjecture depuis plusieurs dizaines d'années, sans savoir le démontrer, qu'en fait *il n'existe pas* d'algorithme efficace pour des problèmes tels que la coloration avec trois couleurs : c'est la conjecture $P \neq NP$. L'énoncé précis de cette conjecture repose entre autres sur une définition précise (et indépendante de la technologie) de la classe des algorithmes efficaces, et sort du cadre de MSI 102.

Exercice 6.1.1 Écrire une fonction `bienColorie(G)` qui teste si un graphe est bien colorié.

Exercice 6.1.2 Un graphe complet d'ordre n est un graphe de n sommets tel que chaque sommet est relié aux autres sommets par une arête ; on appelle K_n un tel graphe. Dessiner K_2, K_3, K_4 et K_5. Quel est le nombre chromatique de K_n ?

Exercice 6.1.3 En remarquant que le graphe ci-contre contient des sous-graphes complets (lesquels ?) montrer que son nombre chromatique est supérieur ou égal à 4. Vérifier qu'on peut effectivement le colorier avec 4 couleurs.





Exercice 6.1.4 Calculer le nombre chromatique du graphe de Petersen, représenté ci-contre à gauche.

Exercice 6.1.5 Quel est le nombre chromatique d'une grille triangulaire T_n (voir exercice 3.3.6 pour la définition) ?

6.2 Graphes 2-coloriables

On dit qu'un graphe est *2-coloriable* si son nombre chromatique vaut 2, autrement dit s'il est possible de le colorier avec deux couleurs. Un tel graphe G est aussi appelé *biparti*, ce qui signifie qu'on peut partitionner les sommets en deux ensembles S_1 et S_2 tels que les arêtes de G relient toujours un sommet de S_1 à un sommet de S_2 — on peut donc attribuer la couleur c_1 à tous les sommets de S_1, et la couleur c_2 à tous les sommets de S_2.

Un *cycle impair* est un cycle de longueur impaire (voir chapitre 4), autrement dit un cycle avec un nombre impair d'arêtes (et donc un nombre impair de sommets si l'on ne compte pas deux fois le sommet de départ qui est aussi le sommet d'arrivée) ; le cycle impair le plus simple est le triangle.

Théorème 6.2 *Un graphe G est 2-coloriable si et seulement si G ne possède pas de cycle impair.*

Exercice 6.2.1 Parmi les graphes de l'exercice 5.1.3 :

1. lequel ou lesquels sont-ils 2-coloriables ?
2. pour le ou les autres, calculer le nombre chromatique.

Algorithme

Pour tester si un graphe connexe G est coloriable avec deux couleurs données c_1 et c_2, on dispose d'un algorithme qui ressemble au test de connexité (voir section 4.3) :

1. choisir un sommet initial s et le colorier avec la couleur c_1 ;
2. tant qu'il existe un sommet t non coloré avec un voisin coloré :
   a) tester si les voisins de t déjà coloriés ont tous la même couleur ;
   b) si c'est le cas, colorier t avec l'autre couleur ;
   c) sinon, arrêter l'algorithme.

Cet algorithme prouve en même temps le théorème 6.2, c'est l'objet de l'exercice suivant.

Exercice 6.2.2

1. Montrer que lorsque cet algorithme colorie tous les sommets du graphe, celui-ci est bien colorié.
2. Montrer que lorsque l'exécution de l'algorithme se termine prématurément (étape 2.c), un cycle impair a été détecté — le graphe n'est donc pas 2-coloriable.

Exercice 6.2.3

1. Écrire une fonction `effacerCouleurs(G)` qui décolore tous les sommets de G.
2. Écrire une fonction `sommetColoriable(G)` qui renvoie un sommet de G non coloré ayant au moins un voisin coloré et *None* si un tel sommet n'existe pas.

3. Écrire une fonction `monoCouleurVoisins(s)` qui, si les voisins colorés du sommet s sont tous de la même couleur, renvoie cette couleur et dans le cas contraire (s a deux voisins de couleurs différentes) renvoie *None*.

4. Écrire une fonction `deuxColoration(G,c1,c2)` qui tente de colorier le graphe G avec les deux couleurs $c1$ et $c2$, et renvoie *True* en cas de succès, *False* sinon.

5. Tester la fonction sur :

   a) des grilles — utiliser la fonction `construireGrille(m,n)`,

   b) des arbres — fonction `construireArbre(degre,hauteur)`,

   c) des graphes bipartis complets — voir la définition ci-dessous (exercice 6.3.3) et la fonction `construireBipartiComplet(m,n)`,

   d) le graphe de Petersen,

   e) le cube, l'octaèdre et le dodécaèdre.

   Après chaque essai dessiner le graphe pour vérifier soit qu'il est bien colorié, soit qu'un sommet non coloré t possède deux voisins de couleurs distinctes ; dans ce dernier cas vérifier qu'un cycle impair relie t et certains des sommets déjà coloriés.

Conseils : lorsque la fonction *deuxColoration* découvre un sommet t non coloré avec deux voisins de couleurs différentes (cas d'un graphe non 2-coloriable), afficher ce sommet en ajoutant simplement une instruction `print(t)`.

Les graphes de cette section sont pour la plupart mieux dessinés en spécifiant l'algorithme de dessin comme suit : `dessiner(G,algo='neato')`. Voir `http://www.graphviz.org` pour une description rapide des algorithmes disponibles avec ce logiciel de dessin de graphes : *dot, neato, fdp, twopi* et *circo*.

Exercice 6.2.4 Comme dans l'exercice 4.3.5 modifier la fonction *sommetColoriable* pour marquer l'arête qui relie ce sommet non coloré à son voisin coloré ; ne pas oublier de modifier aussi la fonction *effacerCouleurs* pour démarquer les arêtes en même temps.

Tester à nouveau la fonction *deuxColoration* comme dans l'exercice précédent, et dessiner à chaque fois le graphe : que remarque-t-on à propos du sous-graphe formé par les arêtes marquées (et leurs extrémités) ? Lorsque le graphe n'est pas 2-coloriable vérifier qu'il existe toujours un cycle impair dont toutes les arêtes, sauf une, sont marquées.

Exercice 6.2.5 Utiliser la fonction *melange* comme dans l'exercice 4.3.6 pour modifier la fonction *sommetColoriable* afin que le résultat ne dépende plus de l'ordre dans lequel sont rangés les sommets du graphe, ni de l'ordre des arêtes incidentes à un sommet.

Une fois cette modification effectuée, appliquer plusieurs fois la fonction *deuxColoration* au dodécaèdre ; après chaque calcul dessiner le graphe et observer quel est le sommet non coloré t avec deux voisins de couleurs différentes qui a stoppé l'exécution de l'algorithme ; observer que le sommet t est situé sur un cycle impair dont toutes les arêtes, sauf une, sont marquées, et que ce cycle n'est pas toujours un pentagone (bonne chance).

6.3 Graphes planaires

Un graphe est dit *planaire* s'il *existe* une façon de le dessiner dans le plan sans que ses arêtes se croisent. Voici un théorème très célèbre, car facile à énoncer et très difficile à démontrer :

Théorème 6.3 (des quatre couleurs) *Tout graphe planaire peut être colorié avec 4 couleurs.*

Voir wikipedia pour une excellente présentation de ce théorème et de son histoire étonnante (origine de la conjecture, preuves percées, emploi d'ordinateurs pour les preuves correctes).

On appelle *carte planaire* tout dessin d'un graphe dans le plan sans croisement d'arêtes ; à un graphe planaire correspondent en général beaucoup de cartes planaires distinctes. En plus des sommets et des arêtes une carte possède des *faces*, car elle découpe le plan en composantes connexes disjointes, bordées par les arêtes. On vérifie facilement qu'un dessin dans le plan correspond à un dessin sur la sphère, et vice-versa ; la seule différence est qu'il semble y avoir une face de plus sur la sphère, mais celle-ci correspond à la *face externe* dans le plan (la composante connexe qui "part vers l'infini"), qu'il ne faut pas oublier quand on compte les faces.

Si S, A, F désignent respectivement l'ensemble des sommets, l'ensemble des arêtes et celui des faces, et si la carte est connexe, on a la célèbre *formule d'Euler* :

$$|S| - |A| + |F| = 2$$

Exercice 6.3.1 Soit K_4 le graphe complet à 4 sommets : dessiner K_4 de deux façons, l'une avec deux arêtes qui se croisent, l'autre sans croisement. K_4 est-il planaire ?

K_5 est-il planaire ? Quelles sont les valeurs de n pour lesquelles K_n est planaire ?

Exercice 6.3.2 Montrer que le graphe de l'exercice 6.1.3 est planaire en proposant plusieurs cartes planaires distinctes pour ce graphe. Vérifier la formule d'Euler sur ces exemples.

Exercice 6.3.3 On note K_{mn} le graphe biparti complet composé de deux ensembles de sommets A et B avec respectivement m et n éléments, et tel que tout sommet de A soit relié à tout sommet de B. Etudier quelles sont les valeurs du couple (m, n) pour lesquelles ce graphe est planaire.

Exercice 6.3.4 On appelle *degré* d'une face le nombre d'arêtes qui la bordent : énoncer une formule analogue à celle des poignées de main (voir section 3.3) ; vérifier cette formule sur les cartes de l'exercice 6.3.2.

Exercice 6.3.5 Démontrer la formule d'Euler par récurrence sur le nombre d'arêtes $|A|$; attention au cas particulier où la suppression d'une arête déconnecte la carte.

Exercice 6.3.6 Soit G un graphe planaire simple : G ne possède ni boucle ni arête multiple, ce qui n'est pas restrictif quand il s'agit de le colorier — cf.introduction de ce chapitre. Montrer que G possède au moins un sommet de degré au plus 5. *Conseils* : raisonner par l'absurde en supposant que tous les sommets sont de degré supérieur ou égal à 6, et appliquer la formule des poignées de main ; remarquer aussi que toute face (d'une carte planaire associée à G) est de degré au moins 3 (car le graphe est simple), ajouter une cuiller de formule d'Euler, et bien mélanger...

Dessiner un graphe planaire simple sans sommet de degré inférieur à 5 (bonne chance).

6.4 Heuristique

Heuristique est un terme dérivé d'une expression grecque signifiant « *discours propre à découvrir* », par opposition à « *discours propre à convaincre* », selon le *Trésor de la Langue Française* : http://atilf.atilf.fr. En sciences on désigne ainsi une méthode ou un principe (souvent simples et astucieux) qui permettent d'obtenir une solution approchée d'un problème, ici celui de la coloration d'un graphe, pour lequel on ne connaît pas de méthode à la fois efficace et rigoureuse.

Voici un algorithme qui permet de colorer un graphe G avec peu de couleurs, mais qui ne garantit pas qu'on ne pourrait pas faire mieux, c'est-à-dire utiliser moins de couleurs ; G est supposé simple (cf. introduction de ce chapitre), ainsi le degré d'un sommet est égal à son nombre de voisins :

1. ordonner les couleurs disponibles dans un ordre arbitraire ;
2. choisir un sommet s de degré minimal d ;
3. retirer s (et toutes les arêtes issues s) du graphe G, ce qui fournit un graphe G' ;
4. colorier G' par la même méthode ;
5. s est alors entouré de d voisins colorés : colorier s avec la première couleur autorisée, c'est-à-dire avec la première couleur qui ne fait pas partie de l'ensemble des couleurs des voisins de s.

Exercice 6.4.1 Exécuter l'algorithme à la main sur le graphe de l'exercice 6.1.3. Autres exemples simples : grille rectangulaire 3×3, grille triangulaire T_3 (voir exercices 3.3.5 et 3.3.6), octaèdre.

Exercice 6.4.2 Soit G un graphe connexe 2-coloriable ; montrer par récurrence sur le nombre de sommets que l'algorithme ci-dessus colorie G avec deux couleurs, à condition que la suppression répétée de sommets de degrés minimaux ne déconnecte jamais le graphe.

Donner un exemple de graphe G connexe 2-coloriable et possédant un sommet s de degré minimal dont la suppression déconnecte G (le graphe G' n'est plus connexe) ; montrer que dans ce cas l'algorithme peut échouer à colorier G avec deux couleurs. Proposer une modification de l'algorithme qui corrige ce défaut.

Exercice 6.4.3 En utilisant l'exercice 6.3.6 montrer que l'algorithme de cette section colorie tout graphe planaire (simple) avec au plus six couleurs.

Exercice 6.4.4 Comme on ne dispose pas de moyen de supprimer un sommet dans un graphe, on va programmer cette heuristique en marquant les sommets, avec la convention qu'un sommet *marqué* sera considéré comme *supprimé* :

1. écrire une fonction `nbVoisinsNonMarques(s)` qui retourne le nombre de voisins du sommet s qui ne sont pas marqués ;
2. écrire une fonction `sommetNonMarqueMinimal(G)` qui retourne un sommet non marqué du graphe G ayant un nombre minimal de voisins non marqués ;
3. écrire une fonction `couleurLibre(s)` qui retourne la première couleur différente des couleurs actuelles des voisins de s ;
4. utiliser les fonctions *sommetNonMarqueMinimal* et *couleurLibre* pour écrire une fonction `degmin(G)` qui colorie un graphe G selon l'algorithme décrit dans cette section ;

5. terminer en écrivant une fonction `colorierGraphe(G)` qui efface les couleurs et les marques du graphe G avant de lancer *degmin* ;

6. améliorer la fonction précédente pour qu'elle retourne le nombre de couleurs employées.

Conseils :

a) Il est possible qu'il n'existe pas de sommet non marqué ! Ceci signifie que tous les sommets ont été supprimés, et dans ce cas la fonction *sommetNonMarqueMinimal* doit retourner *None*.

b) La fonction *degmin* supprime le sommet s retourné par la fonction *sommetNonMarqueMinimal*, c'est-à-dire le marque, après quoi elle appelle ... *degmin* pour colorier le graphe privé de s. Une fonction qui s'appelle elle-même est dite *récursive* ; pour toute fonction de ce type il faut spécifier un cas particulier sans appel récursif, sinon on entre dans une boucle sans fin. Ici le plus simple est de tester si s est égal à *None* : dans ce cas tous les sommets ont été supprimés et il n'y a rien à colorier.

c) Pour toute liste u constituée des éléments $u_0, u_1, u_2 \ldots$ et pour toute fonction f, la liste $[f(u_0), f(u_1), f(u_2) \ldots]$ s'écrit très simplement en Python : `[f(x) for x in u]` — en remplaçant les crochets par des accolades on obtient un ensemble au lieu d'une liste. On pourra utiliser cette construction pour calculer la liste (ou l'ensemble) des couleurs des voisins d'un sommet.

   Pour écrire la fonction `couleurLibre(s)` il reste à parcourir la liste des couleurs disponibles (voir annexe ci-dessous) et à retourner la première qui ne figure pas parmi les couleurs des voisins de s ; si ce n'est pas possible (toutes les couleurs sont prises) retourner *None*.

d) Utiliser le conseil précédent pour calculer le nombre de couleurs employées par la fonction *colorierGraphe* — la fonction `len(E)` compte le nombre d'éléments d'un *ensemble* E.

Tester la fonction *colorierGraphe* sur des grilles, sur le graphe de Petersen (voir exercice 6.1.4), et sur les graphes planaires réguliers (tétraèdre, cube, octaèdre, dodécaèdre et icosaèdre). Utiliser la fonction *melange* comme dans l'exercice 4.3.6 pour que la fonction *sommetNonMarqueMinimal* retourne un sommet aléatoire parmi ceux de degré minimal, lorsqu'il en existe plusieurs : on verra alors que le nombre de couleurs employées pour colorier certains graphes varie suivant l'ordre dans lequel on choisit les sommets de degré minimal.

Annexe : palettes

Du point de vue informatique, une *palette* de couleurs n'est rien d'autre qu'une *liste* de couleurs, qu'on peut construire manuellement :

```
p = ['red', 'green', 'blue'...]
```

— voir http://www.graphviz.org/doc/info/colors.html pour la liste des noms de couleurs reconnues par le programme de dessin de graphes (il y en a plusieurs centaines).

Mais construire des palettes *harmonieuses* est une autre histoire, c'est le travail des graphistes, et le même site propose plusieurs dizaines de palettes prédéfinies (avec de nombreuses variantes suivant le nombre de couleurs souhaitées). Le module *palettes*, à charger par :

```
from palettes import *
```

comporte quelques listes de couleurs empruntées à ce site ; la fonction `paletteNumero(n)` retourne l'une de ces palettes. Ces couleurs ne portent pas de noms, elles sont définies par leur code RGB : six chiffres hexadécimaux précédés du caractère # et encadrés par des apostrophes doubles (sinon le logiciel de dessin *graphviz* n'est pas content).

Chapitre 7. Aide-mémoire

Il faut commencer par importer les graphes définis dans le module *bibV3* :

```
from bibV3 import *
```

Cette phrase magique importe aussi les fonctions suivantes définies dans le module *graphV3* — attention à respecter la distinction entre minuscules et majuscules :

L'argument G est un graphe	
`listeSommets(G)`	retourne la *liste* des *sommets* de G.
`nbSommets(G)`	retourne le *nombre* de *sommets* de G.
`sommetNom(G,etiquette)`	retourne le *sommet* de G désigné par son *nom* (étiquette). Exemple : `sommetNom (Europe, 'Italie').`
`sommetNumero(G,i)`	retourne le *sommet* numéro i dans G ; la numérotation commence à 0.
`dessinerGraphe(G)` ou simplement `dessiner(G)`	demande (très poliment) au logiciel *Graphviz* de dessiner le graphe ; voir page suivante pour les détails.

L'argument s est un sommet	
`listeVoisins(s)`	retourne la *liste* des *voisins* de s.
`degre(s)`	retourne le *degré* de s.
`nomSommet(s)`	retourne le *nom* (étiquette) de s.
`colorierSommet(s,c)`	colorie s avec la couleur c. Exemples de couleurs : `'red'`, `'green'`, `'blue'`, `None`.
`couleurSommet(s)`	retourne la *couleur* de s.
`marquerSommet(s)` `demarquerSommet(s)`	marque ou démarque s.
`estMarqueSommet(s)`	retourne `True` si s est marqué, `False` sinon.
`listeAretesIncidentes(s)`	retourne la *liste* des arêtes *incidentes* à s.

L'argument a est une arête	
`nomArete(a)`	retourne le *nom* (étiquette) de a.
`marquerArete(a)` `demarquerArete(a)`	marque ou démarque a.
`estMarqueeArete(a)`	retourne `True` si a est marquée, `False` sinon.

Arguments : un sommet s et une arête a	
`sommetVoisin(s,a)`	retourne le voisin de s en suivant l'arête a.

L'argument est une liste	
`melange(u)`	retourne une copie mélangée aléatoirement de la liste u. Exemple : `melange(listeSommets(cube))`.

La liste `graphes` contient les graphes `tgv2005`, `fig22`, `Europe`, `Koenigsberg` et `Petersen`. La liste `graphesPlanairesReguliers` contient les graphes `tetraedre`, `cube`, `octaedre` (huit triangles), `dodecaedre` (douze pentagones) et `icosaedre` (vingt triangles). Les fonctions suivantes permettent de construire des graphes de taille variable :

`construireComplet(n)`	retourne le graphe complet K_n . Exemple : `K5 = construireComplet(5)`.
`construireBipartiComplet(m,n)`	retourne K_{mn} . Exemple : `K34 = construireBipartiComplet(3,4)`.
`construireArbre(d,h)`	retourne l'arbre de hauteur h dont chaque sommet possède d fils. Exemple : `arbre = construireArbre(2,3)`.
`construireGrille(m,n)`	retourne la grille rectangulaire avec m lignes et n colonnes. Exemple : `grille = construireGrille(4,6)`.
`construireTriangle(n)`	retourne la grille triangulaire d'ordre n. Exemple : `t5 = construireTriangle(5)`.

La fonction `dessiner` (abréviation de `dessinerGraphe`) comporte des paramètres facultatifs :

`dessiner(G,True)`	dessine le graphe G en affichant les étiquettes des arêtes.
`dessiner(G,algo='neato')`	dessine le graphe G en utilisant un algorithme où les arêtes sont traitées comme des ressorts ;
`dessiner(G,algo='circo')`	dessine le graphe G en utilisant un algorithme de placement des sommets sur un cercle.