Informatique 1 : programmation, bases de l'algorithmique et logique

4TPU146U/4TPU147U

Université de Bordeaux

Équipe enseignante initinfo

https://moodle.u-bordeaux.fr/course/view.php?id=14677

2024–2025

Table des matières **3**

1 Premiers pas en python **8**
1.1 Affectation et expressions . . . 8
1.2 Fonctions . . . 10
1.3 Conditionnelles . . . 13
1.4 Exercices de révisions et compléments . . . 16
1.5 L'essentiel du chapitre . . . 16

2 Listes et boucles `for` **18**
2.1 Listes et boucles `for` . . . 18
2.2 Utilisation de `range` dans des boucles for . . . 19
2.3 Exercices de révisions et compléments . . . 20
2.4 L'essentiel du chapitre . . . 21

3 Compléments (typage, complexité) et exercices de révisions **22**
3.1 Indications de typage . . . 22
3.2 Complexité . . . 22
3.3 Expressions booléennes . . . 24
3.4 Exécution conditionnelle . . . 24
3.5 Utilisation de `range` dans des boucles for . . . 24
3.6 Pour aller plus loin : Analyse de données . . . 25
3.7 Pour aller plus loin : notion de complexité en moyenne . . . 26
3.8 L'essentiel du chapitre . . . 27

4 Accès indexé **29**
4.1 Accès indexé dans les listes . . . 29
4.2 Exercices de révisions et compléments . . . 30
4.3 Pour aller plus loin : Les chaînes de caractères . . . 31
4.4 L'essentiel du chapitre . . . 32

5 Dessin d'images **35**
5.1 Mise en jambes . . . 36
5.2 Tracés de segments . . . 37
5.3 Exercices de révisions et compléments . . . 39
5.4 L'essentiel du chapitre . . . 41

6 Manipulations d'images **43**
6.1 Manipulations d'images . . . 43
6.2 Exercices de révisions et compléments . . . 44
6.3 L'essentiel du chapitre . . . 48

7 Boucle conditionnelle, représentation des nombres **49**
7.1 Boucle conditionnelle . . . 49
7.2 Recherche à l'aide d'une boucle conditionnelle . . . 50
7.3 Représentation des nombres . . . 51
7.4 Exercices de révisions et compléments . . . 53
7.5 L'essentiel du chapitre . . . 57

8 Sommets d'un graphe **59**
8.1 Échauffement : coloriages . . . 61
8.2 Voisins . . . 62
8.3 Calculs sur les degrés . . . 63
8.4 Exercices de révisions et compléments . . . 64
8.5 L'essentiel du chapitre . . . 65

9 Logique de base **66**
9.1 Notions plus ou moins connues 66
9.2 Exercices de révision et compléments . . . 71
9.3 Applications pratiques . . . 72
9.4 Exercices de révisions et compléments . . . 73
9.5 L'essentiel du chapitre . . . 76

10 Formule des poignées de mains, graphes simples **77**
10.1 Formalisation pour les graphes . . . 77
10.2 Formule des poignées de mains . . . 77
10.3 Graphes simples . . . 78
10.4 Exercices de révision et compléments . . . 80
10.5 L'essentiel du chapitre . . . 81

11 Chaînes et connexité **82**
11.1 Chaînes . . . 82
11.2 Démonstration par récurrence . . . 82
11.3 Connexité . . . 83
11.4 Exercices de révisions et compléments . . . 84
11.5 Algorithmes . . . 85
11.6 Exercices et notions complémentaires . . . 88
11.7 L'essentiel du chapitre . . . 90

12 Dictionnaires (hors-programme) **91**
12.1 Introduction . . . 91
12.2 Exercices . . . 92
12.3 L'essentiel du chapitre . . . 92

A Palettes **95**

B Aide-mémoire **96**
B.1 Environnement de TP . . . 96
B.2 Comprendre les messages d'erreur . . . 97
B.3 Utilisation de la bibliothèque de graphes . . . 106
B.4 Utilisation de la bibliothèque d'images . . . 109
B.5 Rappel de la syntaxe . . . 110

Bibliographie **112**

Introduction et utilisation du fascicule

L'objectif de ce cours universitaire de licence semestre 1 est de donner un aperçu de ce qu'est l'informatique en tant que science telle qu'elle est enseignée dans le cursus « informatique » à l'université. C'est également l'occasion pour tous les étudiants d'apprendre à programmer, ce qui leur sera utile quelles que soient les études scientifiques poursuivies. L'enseignement de la partie technique de l'informatique est l'objet du cours de Culture et Compétences numérique.

L'informatique couvrant de très nombreux domaines (bases de données, calcul, image, langages, réseau, sécurité, son, ...), il est impossible de donner un aperçu complet horizontal en un seul semestre. Nous allons donc plutôt approfondir deux notions qui sont utilisées en informatique, les notions d'image et de graphe, et dérouler verticalement les différents types de travaux que l'on effectue typiquement en informatique : des programmes travaillant dessus, bien sûr, mais aussi des preuves théoriques, et des algorithmes permettant d'écrire des programmes plus efficaces que des programmes naïfs.

Ce cours ne nécessite pas de pré-requis en *informatique*. Il n'utilise que quelques pré-requis de *mathématiques* : logique de base, preuve par récurrence.

L'ensemble des ressources électroniques (*e.g*, PDF du polycopié, annales d'examens) est sur le site web https://moodle.u-bordeaux.fr/course/view.php?id=14677

Il y aura un DS intermédiaire en novembre, et un DS terminal en fin de semestre. Il y aura également un TP noté ainsi que des tests.

Le langage de programmation python™

Les exercices de programmation de ce fascicule sont basés sur le langage python™ version 3.0 ou supérieure (*cf.* http://python.org). Attention, l'ancienne version 2 de python est légèrement différente et incompatible.

python est utilisé très largement dans l'industrie, que ce soit pour des sites web (par exemple youtube, pinterest, instagram, divers journaux, ...), du traitement de données (par exemple Google Groups, Gmail, Google maps, dropbox, openstack, ...), des scripts (par exemple les scripts 3D de blender), des applications pour smartphone, des jeux (Civilization IV, Battlefield 2, EVE Online, ...), des applications de calcul scientifique (à la NASA), etc.

Travail en TP

Lors des travaux pratiques, vous pourrez travailler de trois manières :

- Avec Python tutor, surtout au début pour de petits programmes, car il vous montre exactement ce qui se passe dans la mémoire de l'ordinateur pour comprendre pas à pas le fonctionnement de python.
- Avec un environnement de développement intégré plus professionnel comme Spyder.
- Avec les activités sur moodle, qui permettent de faire tester vos programmes sur différents cas, et avoir donc un retour sur la correction de votre code.

Contenu du fascicule

Ce fascicule contient différents chapitres correspondant aux différents objets et propriétés d'images et de graphes que nous allons étudier pendant le semestre.

Un grand nombre d'exercices sont proposés, parmi ceux-ci seuls les exercices placés dans les sections intitulées *Exercices de révisions et compléments* sont facultatifs : ils serviront à bien se préparer aux examens voire à occuper les étudiants les plus rapides.

Les exercices sont en général faisables à la fois sur papier en TD, et sur machine en TP. Certains exercices ne sont réellement intéressants qu'en TP, ils sont marqués avec l'étiquette TP .

Chaque chapitre se termine par une section *L'essentiel du chapitre* qui peut servir de fiche de révision.

À la suite des chapitres constituant le cours on trouvera deux énoncés de devoir surveillés.

On trouvera ensuite un aide-mémoire de la syntaxe de python et des fonctions de nos bibliothèques d'images et de graphes, et, à la fin du fascicule, une description des messages d'erreur et comment les traiter.

Un support plus fourni mais plus ancien est disponible sur Internet sous forme d'un livre :
http://dept-info.labri.fr/ENSEIGNEMENT/INITINFO/initinfo/supports/book/index.html
(Initiation à l'Informatique, par *Irène Durand* et *Robert Strandh*)

Mise en page

Pour les retrouver facilement dans le fascicule, les introductions de notions sont surlignées et les fonctions prédéfinies sont encadrées :

Prototype	Description et exemple
`listeSommets(G)`	retourne la *liste* des *sommets* de G, par exemple : `L = listeSommets(G)`

On trouve à gauche ce que l'on appelle le *prototype* de la fonction, c'est-à-dire son nom et la liste des paramètres que l'on doit lui fournir. La partie droite décrit son comportement.

Seule la liste des fonctions des bibliothèques d'images et de graphes est fournie lors des épreuves. Toutes les notions surlignées doivent être *apprises*.

Les *codes modèles* illustrant des constructions courantes en programmation sont mis en valeur à l'aide d'une double ligne verticale :

```
# définition de la fonction f
def f(x):
  a = x+1
  return a*a + x + 1

# appel de la fonction f avec l'argument 5
y = f(5)

# appel de la fonction f avec l'argument y + 1
z = f(y + 1)
```

Typographie

L'informatique étant une science faisant le lien entre théorie et programmation, dans ce fascicule on utilise les conventions de typographie suivantes pour bien distinguer les deux :

- les énoncés théoriques sont écrits en *italique*. Il ne s'agit donc pas de code python, mais d'énoncés mathématiques décrivant les objets de manière abstraite, permettant ainsi d'écrire des preuves, par exemple.
- les énoncés python sont écrits en `style machine à écrire` (police à chasse fixe), il ne s'agit donc pas d'énoncés mathématiques, mais de code python, interprété par la machine.

Il est important de bien distinguer les deux, car ils ne s'écrivent pas de la même façon et n'ont pas exactement les mêmes détails de signification – Par exemple $i=i+1$ est essentiellement toujours faux en mathématiques, mais `i=i+1` est très utilisé en python ! Certains exercices incluent par exemple un énoncé mathématique qu'il s'agira de reformuler en python.

Chapitre 1. Premiers pas en python

Voir l'annexe B pour une description de l'environnement Python Tutor.

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 12×5 dans une variable qui s'appelle `x`, on tape l'instruction

```
x = 12 * 5
```

On peut ensuite réutiliser `x`, et sauvegarder la valeur de $\texttt{x}^2$ dans la variable de nom `y` en tapant l'instruction `y = x * x`. Contrairement à sa signification mathématique, le symbole `=` signifie « *calculer la valeur à droite du signe* `=` *puis mémoriser le résultat dans la variable dont le nom est à gauche du signe* `=` », il y a donc deux étapes bien distinctes : calculer d'abord, et stocker le résultat ensuite.

Les instructions sont exécutées une à une et dans l'ordre dans lequel elles sont écrites : elles sont exécutées en *séquence*. Le tableau suivant illustre l'évolution des valeurs des variables `x`, `y` et `z` lors de l'exécution séquentielle des quatre instructions suivantes :

```
x = 12 * 5
y = x * x
z = x + 1
x = y + z
z = x + 1
```

	étape 1	*étape 2*	*étape 3*	*étape 4*	*étape 5*
`x`	**60**	60	60	**3661**	3661
`y`		**3600**	3600	3600	3600
`z`			**61**	61	**3662**

$\longrightarrow$ *temps*

On observe que dans chaque colonne de ce tableau, une seule case est en gras ; elle correspond à la variable affectée à l'étape correspondante (l'étape *1* correspond à l'exécution de la première instruction, etc . . .), les autres variables ne sont pas affectées. On notera également que l'affectation de la variable `x` à l'étape *4* n'a pas d'effet sur les variables `y` et `z`. La valeur finale d'une variable est la valeur dans la dernière colonne. La valeur finale de `x` est donc 3661, celle de `y` est 3600 et celle de `z` est 3662.

Pour améliorer la lecture et faciliter l'écriture de ce style de tableau, il est intéressant d'écrire uniquement les valeurs des variables affectées, étape par étape. La valeur finale d'une variable est maintenant la valeur la plus à droite sur sa ligne, il s'agit toujours de la valeur calculée lors la dernière affectation de la variable.

	étape 1	*étape 2*	*étape 3*	*étape 4*	*étape 5*
`x`	**60**			**3661**	
`y`	—	**3600**			
`z`	—	—	**61**		**3662**

$\longrightarrow$ *temps*

Dans le tableau ci-dessus une seule valeur apparaît par colonne car dans le programme correspondant, comme dans tous ceux de ce chapitre, une affectation ne concerne qu'une seule variable à la fois. Dans les exercices qui suivent, nous vous recommandons fermement d'utiliser un tel tableau pour montrer l'évolution des valeurs des variables. On veillera à ne faire apparaître qu'une seule affectation par colonne pour bien mettre en valeur la chronologie des affectations.

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

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

	étape 1	*étape 2*	*étape 3*	*étape 4*
x				
y				
z				

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
```

	étape 1	*étape 2*	*étape 3*	*étape 4*
x				
y				
z				

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.

	départ	*étape 1*	*étape 2*	*étape 3*	*étape 4*	*étape 5*	*étape 6*	...	*étape n*
x	1								

Exercice 1.1.5 Parmi les codes suivants, quels sont les programmes python qui s'exécutent sans causer d'erreur ? Indiquer les erreurs dans les codes erronés.

```
x = 1
x = y + 1
y = 2
```

```
y = 2
x = 1
x = y +- 2
```

```
y = 2
x = 1
x = y + 3
```

```
x = 1
y = 0
x + y = 1
```

```
y = 2
x = 1
x = y +/- 1
```

```
y = 2
x = 1
y = x -+ 1
```

Exercice 1.1.6 Écrire une suite d'instructions permettant d'échanger le contenu de deux variables `a` et `b`.

Divisions entières (ou Euclidiennes)

En termes simples, la division entière (aussi appelée Euclidienne) est une division arrondie « par défaut », c'est-à-dire que l'on ne conserve pas les chiffres après la virgule dans le résultat (le quotient) : $17/5 = 3,4$ mais on ne conserve que 3. Il y a alors un reste : $17 - 5 * 3 = 2$

Plus formellement, 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` (aussi appelé modulo).

Exercice 1.1.7 TP Taper les instructions suivantes et prenez le temps d'expliquer précisément l'évolution des variables pendant l'exécution pas à pas de ces instructions. Les parties à droite des dièses `#` ne sont que des commentaires pour vous, l'ordinateur ne les interprètera pas même si vous les tapez.

```
x = 11 * 34
x = 13.4 - 6
y = 13 / 4    # division avec virgule
y = 13 // 4   # division entiere, notez la difference
z = 13 % 2    # pourquoi cela vaut-il 1 ?
x = 14 % 10   # quel sens donner au reste d'une divison par 10 ?
y = 14 // 10
i = x + y
```

x															
y															
z															
i															

1.2 Fonctions

La plupart des fonctions servent à *calculer* un résultat, comme en mathématiques, et à le *retourner* (on dit aussi *renvoyer*). La syntaxe python pour la définition d'une fonction est la suivante :

```
def nom_fonction(liste, de, parametres):
  corps de la fonction
```

La définition d'une fonction, effectuée en général une seule fois, permet de définir les paramètres de la fonction et le corps de la fonction. Remarquez les deux points à la fin de la première ligne. Les instructions qui constituent le corps de la fonction doivent être indentées par rapport au mot clef `def`, c'est-à-dire décalées, pour indiquer à python qu'elles font partie de la définition de la fonction.

L'instruction `return` termine l'exécution de la fonction, elle peut être suivie d'une expression pour indiquer la valeur renvoyée par la fonction.

Dans l'exemple ci-dessous, la fonction `f` a un seul paramètre, nommé `x`, et le corps de la fonction `f` est constitué de deux instructions.

Exemple de définition et d'appels de fonction :

```
# définition de la fonction f
def f(x):
  a = x+1
  return a*a + x + 1

# appel de la fonction f avec l'argument 5
y = f(5)

# appel de la fonction f avec l'argument y + 1
z = f(y + 1)
```

Une fonction doit être appelée pour être exécutée et peut être appelée autant de fois que l'on veut. Un appel de fonction fournit les arguments de cet appel et il y a autant d'arguments qu'il y a de paramètres dans la définition de la fonction.

Comme pour l'instruction d'affectation, l'appel d'une fonction se fait en plusieurs étapes bien distinctes : les valeurs des arguments passés à la fonction sont d'abord calculées. La fonction est alors appelée avec le résultat de ces calculs. Le corps de la fonction est alors exécuté, les paramètres contenant alors les résultats des calculs des arguments. La fonction se termine au

premier `return` exécuté qui désigne la valeur à retourner. L'exécution revient alors à l'endroit où l'on a effectué l'appel à la fonction, et c'est la valeur retournée par la fonction qui y est utilisée.

```
# definition de la fonction g contenant du code mort
def g(x):
  a = x+1
  return a*a + x + 1
  # code mort - erreur de programmation a eviter
  b = a + 1
```

La définition de fonction ci-dessus contient du code mort : les instructions qui suivent une instruction `return` ne sont jamais exécutées, c'est une erreur de programmation.

Exercice 1.2.1 TP Taper les instructions suivantes qui appellent la fonction `f` en lui passant différents arguments et prenez le temps d'expliquer en détail les résultats obtenus.

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

y = f(2)
t = 4
y = f(t)              # on passe la valeur d'une variable
y = f(1) + f(2)       # on effectue deux appels
x = 0
z = x+1
y = f(z)
y = f(x+1)            # on passe directement la valeur d'une expression
z = f(x-t)
t = f(t)              # on peut meme passer la variable qui servira a
                      # stocker le resultat
x = f(f(1))           # on peut combiner deux appels, le resultat de
                      # l'un est passe en parametre a l'autre
```

Exercice 1.2.2 Parmi les codes suivants, quels sont les programmes python qui ne comportent pas d'erreur ? Rayer les codes erronés.

```
Def fun(x):
  return x + 1

y = fun(3)
```

```
def fun(x):
return x + 1

y = fun(3)
```

```
def fun(x)
  return x + 1

y = fun(3)
```

```
def fun(y):
  return x + 1

y = fun(3)
```

```
def fun(x):
  return x + 1

y = fun(3)
```

```
def fun():
  return 42

y = fun(3)
```

Exercice 1.2.3

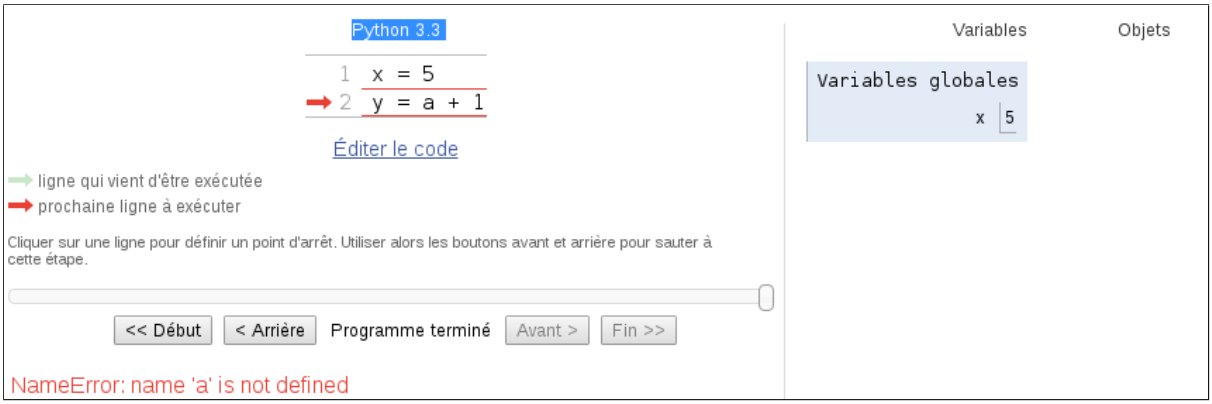
1. Soient `x` et `y` deux variables contenant chacune un nombre. Écrire l'expression python qui stocke dans une variable `m` le calcul de la moyenne des deux nombres `x` et `y`.
2. Écrire une fonction `moyenne(a,b)` qui retourne la moyenne des deux nombres `a` et `b`. Testez-la avec les arguments 42 et 23.

3. Écrire une fonction `moyennePonderee(a, coef_a, b, coef_b)` qui retourne la moyennne avec des coefficients `coef_a` pour la note `a` et `coef_b` pour la note `b`. Testez-la en appelant `moyennePonderee(5,2,12,3)`.

4. Utilisez votre fonction `moyennePonderee(a, coef_a, b, coef_b)` pour écrire une autre version de la fonction `moyenne(a,b)` *(question* 2(question 2.)*.)* qui fait simplement un appel à `moyennePonderee`. Testez-la.

Compléments de programmation

Une erreur classique de programmation consiste à utiliser une variable qui n'est pas définie. C'est le cas de la variable `a` dans le code ci-dessous. Cette erreur est signalée par un message et provoque l'arrêt de l'exécution du programme.



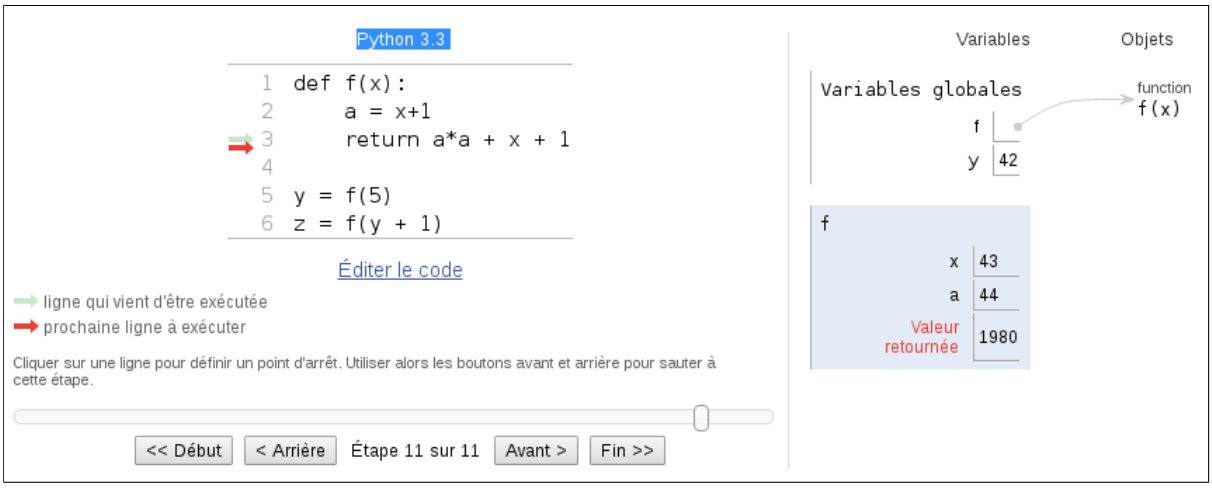
La figure ci-dessous illustre la distinction entre les variables globales et locales :

- les variables globales sont définies en dehors de toute fonction ; par exemple, la variable `y` de valeur 42.

- les variables locales à une fonction sont les paramètres de la fonction et les variables définies dans son corps ; par exemple, les variables `x` et `a` pour la fonction `f`. Ces variables n'existent que pour la durée de l'exécution de la fonction (lors d'un appel).

Il est possible d'utiliser (lire) une variable globale dans le corps d'une fonction. Pour des raisons de lisibilité du code, nous n'utiliserons pas cette possibilité : dans le corps des fonctions que nous écrirons, nous utiliserons uniquement les variables locales à cette fonction.

De plus, à notre niveau, il est préférable d'éviter d'avoir des variables globales et locales de même nom.



1.3 Conditionnelles

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`, `x >= y`, etc. sont des expressions booléennes. On peut combiner des expressions booléennes avec les opérateurs `and`, `or` et `not`.

Exercice 1.3.1 TP Taper les instructions suivantes et prenez le temps d'expliquer précisément l'évolution des variables pendant l'exécution pas à pas de ces instructions.

```
i = 9
j = 0
b = i < j     # b ne contient donc pas un entier, mais True ou False
b = i != 9
b = i == 9
bi = i % 2 == 0 or i % 3 == 0
bj = j % 2 == 0 or j % 3 == 0
b = bi and bj
c = not b
```

Exercice 1.3.2 (À faire sur papier seulement)
Parmi les expressions suivantes, quelles sont celles qui valent `True` si les trois variables `a`, `b` et `c` ont des valeurs toutes différentes deux à deux, et `False` sinon ? Rayer les expressions incorrectes.

- `a != b and b != c`
- `a != b and b != c and a != c`
- `a != b or b != c`
- `a != b or b != c or a != c`

Exercice 1.3.3 Écrire une expression qui vaut `True` si `x` appartient à $[0, 5[$ et `False` dans le cas contraire.

Écrire une expression qui vaut `True` si `x` n'appartient **pas** à $[0, 5[$ et `False` dans le cas contraire. Proposez-en deux formes différentes : l'une avec une négation `not`, et l'autre sans.

Exercice 1.3.4 (À faire sur papier seulement)

Parmi les expressions suivantes, quelles sont celles qui valent `True` si l'entier n est pair, et `False` s'il est impair ? Rayer les mauvaises réponses.

- `n == 0 % 2`
- `1 != 2 % n`
- `n // 2 == 0`
- `n % 2 != 1`
- `0 != n % 2`
- `n % 2 == 0`

Exercice 1.3.5 Écrire une fonction `pair(n)` qui teste si son paramètre `n` est pair ; la valeur renvoyée doit être *booléenne*, c'est-à-dire égale à `True` ou `False`. Testez-la.

Exécution conditionnelle

La syntaxe `if ... else ...` permet de tester des conditions pour exécuter ou non certaines instructions.

```
if condition_1 :
    code a executer si condition_1 est vraie
elif condition_2 :
    code a executer si condition_2 est vraie
    et condition_1 est fausse
else :
    code a executer si aucune condition est vraie
```

La partie `else` est optionnelle et la partie `elif` peut apparaître autant de fois que nécessaire. Les exemples suivant détaillent l'utilisation de cette structure :

```
def prixCinema(age):
  prix = 10
  if age < 18:
    prix = 6.70
  return prix
```

```
def prixCinema(age):
  if age < 18:
    prix = 6.70
  else:
    prix = 10
  return prix
```

Ne pas oublier les deux points à la fin des lignes `if, 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. La partie `else` n'est pas obligatoire : son absence indique simplement qu'il n'y a rien à faire dans ce cas.

La syntaxe `if ... else ...` peut être imbriquée, par exemple cela permet de distinguer trois cas : $age \leq 14$, $14 < age < 18$ et $age \geq 18$. :

```
def prixCinema(age):
  if age <= 14:
    prix = 5
  else:
    if age < 18:
      prix = 6.70
    else:
      prix = 10
  return prix
```

On peut utiliser la syntaxe `if ... elif ... else ...` pour l'écrire de manière plus simple. On peut répéter la partie `elif` autant de fois que voulu. Sur cet exemple :

```
def prixCinema(age):
  if age <= 14:
    prix = 5
  elif age < 18:
    prix = 6.70
  else:
    prix = 10
  return prix
```

Cette fonction peut être écrite encore plus simplement en éliminant la variable locale `prix` grâce à l'instruction `return` :

```
def prixCinema(age):
  if age <= 14:
    return 5
  elif age < 18:
    return 6.70
  else:
    return 10
```

Finalement, comme l'instruction `return` *termine* l'exécution de la fonction, on peut plus encore simplifier ce code :

```
def prixCinema(age):
  if age <= 14:
    return 5
  if age < 18:
    return 6.70
  return 10
```

Supposons que l'on exécute `prixCinema(17)`. La condition (`age <= 14`) du premier test n'étant pas satisfaite, le deuxième test (`if age < 18`) est alors évalué. Comme la condition (`age < 18`) est vraie, l'instruction `return 6.70` est exécutée et a pour effet de terminer l'évaluation de la fonction. Dans ce cas, l'instruction `return 10` n'est pas executée.

Exercice 1.3.6 (À faire sur papier seulement)

Donner les valeurs des variables `x` et `y` après exécution de l'exemple suivant pour `x` valant 1, puis pour `x` valant 8.

```
if x % 2 == 0:
    y = x // 2
else:
    y = x + 1
x = x + 1
```

Même question pour chacun des codes suivants :

```
if x % 2 == 0:
    y = x // 2
    y = x + 1
x = x + 1
```

```
if x % 2 == 0:
    y = x // 2
else:
    y = x + 1
    x = x + 1
```

```
if x % 2 == 0:
y = x // 2
else:
    y = x + 1
x = x + 1
```

Exercice 1.3.7 Écrire une fonction `compare(a,b)` qui retourne -1 si $a < b$, 0 si $a = b$, et 1 si $a > b$. Testez-la. Donner plusieurs versions de cette fonction en vous inspirant de celles de la fonction `prixCinema`.

1.4 Exercices de révisions et compléments

Exercice 1.4.1 Écrire une fonction `max2(x,y)` qui calcule et retourne la plus grande valeur parmi deux nombres x et y. Attention : bien nommer cette fonction `max2`, et non `max`, car la fonction `max` est prédéfinie en python.

Exercice 1.4.2 Écrire une fonction `abs2(x)` qui retourne la valeur absolue de x. Attention : bien nommer cette fonction `abs2`, et non `abs`, car la fonction `abs` est prédéfinie en python.

Exercice 1.4.3 Écrire une fonction `nbJours(mois)` qui reçoit en paramètre le numéro d'un mois et qui renvoie le nombre de jours de ce mois (sans tenir compte des années bisextiles). Note : éviter de traiter un par un chacun des 12 cas ! :)

1.5 L'essentiel du chapitre

Affectation : `variable = expression`

Opérateurs mathématiques : opérateurs usuels `+,-,*,/`, division entière `//`, reste de la division entière `%` .

Opérateurs booléens : comparaison `<,>,<=,>=`, égalité `==,!=`, combinaison `and,or,not`
Ci-dessous, les caractères "_____" représentent l'indentation obligatoire.

```
if (condition):
_____instructions exécutées quand
_____condition est vraie
```

Exemple :

```
if age <= age_reduction:
  prix = prix / 2
```

```
if (condition):
_____instructions exécutées quand
_____condition est vraie
else:
_____instructions exécutées quand
_____condition est fausse
```

Exemple :

```
if age <= age_reduction:
  prix = 5
else:
  prix = 10
```

```
if (condition1):
_____instructions exécutées quand
_____condition1 est vraie
elif (condition2):
_____instructions exécutées quand
_____condition2 est vraie
else:
_____instructions exécutées quand
_____condition1 et condition2 sont fausses
```

Exemple :

```
if age <= age_reduction1:
  prix = 5
elif age >= age_reduction2:
  prix = 7
else:
  prix = 10
```

Définition d'une fonction :

```
def fonction(parametres, avec, virgules):
_____instructions exécutées quand
_____fonction est appelée
_____return valeur
```

Exemple :

```
def f(n,m):
  if n < m:
    return n
  return m
```

Appel d'une fonction :

```
fonction(arguments,a,fournir)
```

Exemple :

```
f(1,3)
```

Note : la fonction ne doit pas décider elle-même des valeurs de ses paramètres, c'est au moment de l'appel de fonction que l'on fournit les arguments. Par exemple ci-dessus, c'est l'appel `f(1,3)` qui indique que la fonction travaille sur les entiers `1` et `3`.

Cela permet ainsi à la fonction d'être *générique* : une fois écrite, on n'a plus jamais besoin d'y toucher, on peut l'appeler plusieurs fois avec différents arguments.

Chapitre 2. Listes et boucles `for`

2.1 Listes et boucles `for`

python permet de manipuler les listes d'éléments, par exemple `[2,8,5]` est une liste d'entiers. 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 (on dit que la liste est *itérable*) ; en voici la syntaxe :

```
for variable in liste :
    corps de la boucle
```

Remarquez bien les deux points à la fin de la ligne et le fait que les instructions appartenant au corps de la boucle doivent être indentées (décalées) par rapport au mot clef `for`. L'évaluation par python d'une boucle `for` correspond à l'algorithme suivant :

1. calculer la valeur de la liste et la mémoriser ;
2. sortir de la boucle si tous les éléments de la liste ont été traités ;
3. affecter la valeur du premier élément non traité à la variable de boucle ;
4. exécuter le corps de la boucle ;
5. recommencer le traitement à partir de l'étape 2.

Exercice 2.1.1 Dans l'environnement Python Tutor, entrer l'instruction suivante et analyser ce qui est affiché :

```
for elt in [2, 8, 5]:
    print(elt, elt * elt)
```

Même question pour :

```
u = [2, 8, 5]
for elt in u:
    print(elt, elt * elt)
```

La variable `elt` est-elle définie après exécution de ces instructions ? Si oui, quelle est sa valeur ?

Exercice 2.1.2 On considère la définition suivante :

```
def sommeListe(L):
    s = 0
    for elt in L:
        s = s + elt
    return s
```

Que retournent les appels `sommeListe([1,3,13])`, `sommeListe([1])` et `sommeListe([])` ? Pour vous aider, remplissez le tableau ci-dessous.

	étape 1	*étape 2*	*étape 3*	*étape 4*	*étape 5*	*étape 6*	*étape 7*	*étape 8*	*étape 9*
`L`	[1,3,13]								
`elt`	-								
`s`	0								

L'instruction `return` termine l'exécution de la fonction qui la contient. Que calcule la fonction `somme` si par malheur on indente trop la dernière ligne, comme ci-dessous ?

```
def sommeListe(L):
    s = 0
    for elt in L:
        s = s + elt
        return s    # gros bug !
```

Exercice 2.1.3 Écrire les fonctions suivantes prenant en paramètre une liste de nombres `L`, et **testez** ces fonctions :

- une fonction `moyenneListe(L)` qui calcule et retourne la moyenne de ces nombres,
- une fonction `nbPairsListe(L)` qui compte et retourne combien de ces nombres sont pairs.
- une fonction `maximumListe(L)` qui calcule et retourne le maximum de ces nombres (supposés positifs).

2.2 Utilisation de `range` dans des boucles for

La fonction `range(debut, fin, pas)` permet de définir une suite arithmétique finie d'entiers de raison `pas` commençant par l'entier `debut` et bornée par `fin` (qui ne fait pas partie de la suite). Les syntaxes possibles de `range` sont :

```
range(debut, fin, pas)
range(debut, fin)       # pas vaut 1 par défaut
range(fin)              # debut vaut 0 par défaut
```

L'argument `pas` est donc optionnel, et vaut 1 par défaut. L'argument `debut` est également optionnel, et vaut 0 par défaut. La suite se termine **juste avant** d'atteindre `fin`.

Exercice 2.2.1 TP Entrer les instructions suivantes et analyser les réponses de python :

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

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

```
for k in range(3, 9, 2):
    print(k)
```

```
for i in range(10):
    print("Bonjour")
```

Exercice 2.2.2 TP Dans l'environnement Python Tutor, tester chacun des groupes d'instructions suivantes et analyser les résultats comme précédemment :

```
for a in [10,11,12]:
    for b in [1,2,3]:
        print(a, b)
```

```
u = [2,6,1,10,6]
v = [2,5,6,4]
for x in u:
    for y in v:
        print (x, y, x + y, x == y)
```

Exercice 2.2.3 On considère la définition suivante :

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

Que retournent les appels `mystere(2)`, `mystere(3)`, et `mystere(n)` en général ?
Connaissez-vous une formule qui permet de calculer le même résultat ?

Exercice 2.2.4 Écrire une fonction `sommeCarres(n)` qui calcule et retourne $\sum_{i=1}^{n} i^2$.

2.3 Exercices de révisions et compléments

Exercice 2.3.1 Multiplication à la main

Écrivez `produit(x,y)` qui calcule le produit de deux nombres x et y, mais sans utiliser l'opérateur `*` : à la place, utilisez une boucle qui réalise des additions successives.

Exercice 2.3.2 Écart-type

1. En vous inspirant de la fonction `sommeListe` de l'exercice 2.1.2, écrivez une fonction `sommeCarresListe(L)` calculant la somme des carrés des éléments de la liste `L`.

2. L'écart-type d'une liste de nombres L permet d'estimer dans quelle mesure les éléments de L s'éloignent de la moyenne des éléments de L. Par exemple l'écart-type de la liste `[8, 8, 8, 12, 12, 12]` est de 2 (puisque tous les éléments sont à distance 2 de la moyenne 10).

   On peut le calculer en utilisant la somme des carrés des éléments de L et la somme des éléments de L (en notant n le nombre d'éléments de L obtenu avec la fonction `len`, et L_i les éléments de la liste L) :

$$EcartType(L) = \sqrt{\frac{\sum_i ({L_i}^2)}{n} - \left(\frac{\sum_i L_i}{n}\right)^2}$$

   Pour calculer une racine carrée, il faut ajouter `from math import sqrt` au début du fichier pour avoir accès à la fonction `sqrt` . En appelant les fonctions `sommeListe` et `sommeCarresListe` écrites précédemment, écrivez une fonction `ecartTypeListe(L)` qui calcule l'écart-type de la liste `L`.

2.4 L'essentiel du chapitre

```
for variable in une_liste:
    instructions exécutées avec variable
    contenant successivement les valeurs
    de une_liste
```

Exemple :

```
s = 0
for a in range(1,10):
  s = s + a
```

La fonction `range` permet de générer des listes d'entiers utilisables par la primitive `for` :

```
list(range(10))                    [0,1,2,3,4,5,6,7,8,9]
list(range(3,7))                              [3,4,5,6]
list(range(1,20,4))                      [1,5,9,13,17]
```

Note : Lorsque l'on a une fonction qui prend en paramètre une liste `L`, on n'utilise pas `range` puisque l'on a déjà une liste pour `for` :

```
def f(L):
  for x in L:
    ...
```

Inversement, si la fonction prend en paramètre un entier `n`, on a besoin d'utiliser `range` pour fabriquer une liste pour `for` :

```
def f(n):
  for i in range(n):
    ...
```

Chapitre 3. Compléments (typage, complexité) et exercices de révisions

3.1 Indications de typage

On a vu au chapitre précédent que l'on pouvait passer différents types de paramètres à une fonction, par exemple il y a une grande différence entre ces deux fonctions :

```
def mystere1(L):
    s = 0
    for elt in L:
        s = s + elt
    return s
```

```
def mystere2(n):
    s = 0
    for i in range(n):
        s = s + i
    return s
```

La fonction `mystere1` prend en paramètre une liste de nombres `L` et calcule la somme de ses éléments, alors que la fonction `mystere2` prend juste en paramètre un nombre, et calcule la somme des nombres de 0 à $n-1$. A priori, lorsque l'on commence à lire le code de la fonction, on ne sait pas vraiment quel type de paramètre la fonction s'attend à recevoir. En regardant le nom du paramètre, on peut se douter que `L` est probablement censé être une liste, mais on pourrait avoir n'importe quel nom de paramètre... Lorsque l'on lit la ligne `for elt in L`, on devient sûr que `L` est censé être une liste : `for` veut absolument une liste à droite du `in`[1].

Une bonne pratique est de *documenter* le type de paramètre qui est attendu par la fonction, en ajoutant l'indication du type dans la liste des paramètres, par exemple `int` pour un entier, `list` pour une liste. On peut également documenter le type de ce qui est retourné par la fonction avec `->`. On obtient ainsi :

```
def mystere1(L: list) -> int:
    s = 0
    for elt in L:
        s = s + elt
    return s
```

```
def mystere2(n: int) -> int:
    s = 0
    for i in range(n):
        s = s + i
    return s
```

Ainsi, les fonctions perdent un peu de leur mystère, elles sont plus faciles à comprendre !

Attention, ces indications de typage ne sont à écrire qu'au moment de la *définition* de la fonction, il ne faut pas les écrire au moment des *appels* à la fonction.

3.2 Complexité

3.2.1 Introduction

Supposons que l'on dispose de deux listes d'entiers, on souhaite déterminer combien il y a d'entiers qui apparaissent à la fois dans les deux listes (On suppose que les listes sont elles-même sans doublons).

1. ou du moins quelque chose d'*itérable*.

```
def nbDoublons(L1: list, L2: list) -> int:
  n = 0
  for elt1 in L1:
    for elt2 in L2:
      if elt2 == elt1:
        n = n + 1
  return n
```

On se pose la question de la complexité de cette fonction : combien d'opérations effectue-t-elle ? Par opération, on entend chaque calcul, chaque comparaison, chaque affectation, etc.

Ici, pour chaque élément de la liste `L1`, on parcourt toute la liste `L2` pour déterminer s'il apparaît dedans. Ainsi, si l'on note n_1 la longueur de la liste `L1` et n_2 la longueur de la liste `L2`, la comparaison `elt2 == elt1` est faite $n_1 \times n_2$ fois, et le calcul `n + 1` et l'affectation `n = n + 1` sont faites au plus $n_1 \times n_2$ fois.

On dira alors ici que la complexité de `doublon` est $\mathcal{O}(n_1 \times n_2)$.

Il est important de remarquer que c'est juste l'ordre de grandeur qui nous intéresse : le fait qu'il y ait *trois* opérations (ou une seule, selon le résultat de la comparaison) dans chaque tour de la boucle `for j` n'est pas vraiment intéressant du point de vue théorique : si l'on avait un ordinateur deux-trois fois plus rapide ce facteur trois disparaîtrait. C'est pour cela que l'on ne fait pas apparaître ce facteur 3 dans la complexité, et l'on ne fait apparaître que les tailles des listes : un programme qui est trois fois plus rapide ou trois fois plus lent, ce n'est pas très important d'un point de vue théorique. Par contre, selon que les listes manipulées ont pour taille 1000, 10 000, 1 000 000, cela change énormément le temps de calcul !

3.2.2 Attention aux complexités cachées

Soient les fonctions :

```
def nbOccurrences(x: int, L: list) -> int:
  n = 0
  for elt in L:
    if elt == x:
      n = n + 1
  return n

def nbDoublons(L1: list, L2: list) -> int:
  n = 0
  for elt in L1:
    n = n + nbOccurrences(elt, L2)
  return n
```

Quelle est la complexité de la fonction `nbDoublons` ?

Attention à la complexité cachée dans l'appel à la fonction `nbOccurrences` ! Même si l'affectation a l'air simple, il faut compter tout le temps de calcul de l'expression utilisée. La fonction `nbOccurences` a en effet une complexité qui n'est pas triviale : si on note n la taille de la liste `L`, elle a pour complexité $\mathcal{O}(n)$.

Ainsi, dans cette version aussi `nbDoublons` a pour complexité $\mathcal{O}(n_1 \times n_2)$

Exercice 3.2.1 Quelles sont les complexités des fonctions `moyenneListe`, `nbPairsListe`, `maximumListe` ?

Exercice 3.2.2 Quelle est la complexité de la fonction suivante :

```
def nbPairs(L1: list, L2: list) -> int:
  n = 0
  for elt in L1:
    if elt % 2 == 0:
      n = n + 1
  for elt in L2:
    if elt % 2 == 0:
      n = n + 1
  return n
```

3.3 Expressions booléennes

Exercice 3.3.1 Parmi les expressions suivantes, quelles sont celles qui valent `True` si et seulement si les trois variables `a, b, c` ont toutes la même valeur ? Rayer les expressions incorrectes.

- `a == b and b == c`
- `a == b and b == c and a == c`
- `not (a != b or b != c)`
- `not (a != b or b != c or a != c)`

3.4 Exécution conditionnelle

Exercice 3.4.1 Écrire une fonction `max3(x: int, y: int, z:int) -> int` qui calcule et retourne le maximum de trois nombres x, y, z. Donner plusieurs versions de cette fonction dont une utilise la fonction `max2` de l'exercice 1.4.1

Exercice 3.4.2 Écrire une fonction `uneMinuteEnPlus(h: int, m: int)` qui calcule et retourne l'heure une minute après celle passée en paramètre, sous la forme d'un `tuple` de deux entiers correspondant à une heure et une minute valides. Exemples :

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

Ne pas oublier le cas de minuit.

Exercice 3.4.3 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: int) -> int` qui calcule et retourne le prix à payer pour `n` photocopies, en centimes.

3.5 Utilisation de `range` dans des boucles for

Exercice 3.5.1 La fonction `factorielle(n)` (notée mathématiquement « $n!$ ») peut être définie de la manière suivante (on suppose que $n \geq 1$) :

$$n! = 1 \times 2 \times \cdots \times n$$

Écrire une fonction `factorielle(n: int) -> int` qui utilise cette définition pour calculer et retourner $n!$. Tester votre fonction en affichant les factorielles des nombres de 0 à 100.

Exercice 3.5.2 (extrait DS 2015-16) Rappel : On dit que i est un diviseur de n si le reste de la division de n par i est égal à 0.

1. Écrire une fonction `estDiviseur(i: int, n: int) -> bool` qui retourne `True` si `i` est un diviseur de `n` et `False` sinon.

2. Un nombre est dit *premier* s'il n'a que 2 diviseurs : 1 et lui-même. Calculez à la main sur papier la liste des nombres premiers inférieurs à 15.

3. Écrire une fonction `estPremier(n: int)` qui retourne `True` si `n` est premier, `False` sinon (on profitera du fait que seuls les nombres strictement inférieurs à `n` peuvent être diviseurs de `n`). Quelle est la complexité de cette fonction, en fonction de `n` ?

4. En s'aidant de la fonction `estPremier`, écrire une fonction `nbPremiers(n: int)` qui retourne le nombre de nombres premiers strictement plus petits que n. Note : l'idée est que `nbPremiers` *appelle* `estPremier`, ce qui simplifie beaucoup son écriture, dans `nbPremiers` il y a seulement besoin d'une boucle for. Quelle est la complexité de `nbPremiers`, en fonction de `n` ?

3.6 Pour aller plus loin : Analyse de données

Nous vous fournissons un module `bibcsv.py` qui permet d'ouvrir des fichiers CSV contenant juste une liste de nombres. Ce module est disponible en téléchargement sur le site https://moodle.u-bordeaux.fr/course/view.php?id=14677. Utiliser un clic droit et "enregistrer sous" pour l'enregistrer à côté de vos autres fichiers python. Pour utiliser un module, il faut commencer par l'*importer*, et toute session de travail utilisant ce module doit commencer par la phrase magique :

```
from bibcsv import *
```

Vous disposez alors de la fonction

`ouvrirCSV(nom:str) -> list`	Ouvre le fichier *nom* et retourne la liste de nombres qu'il contient, par exemple : `l = ouvrirCSV("notes.csv")`

Exercice 3.6.1 Récupérer le fichier `notes.csv` depuis le site du cours, l'enregistrer de la même façon, et utiliser `ouvrirCSV` pour récupérer la liste des nombres stockée dans le fichier CSV :

```
maliste = ouvrirCSV("notes.csv")
```

et observer le contenu de la variable `maliste`.

1. Utilisez les fonctions `moyenneListe`, `ecartTypeListe`, `maximumListe`, pour analyser la liste de notes contenue dans `maliste`.

2. Vous pouvez créer votre propre fichier `.csv` avec LibreOffice. Dans une feuille de calcul, mettez les nombres à la suite dans la première colonne uniquement (ou bien en les copiant/collant depuis un document existant). Utilisez "Fichier", "Enregistrer sous", saisissez un nom de fichier en utilisant l'extension `.csv` et validez, confirmez que c'est bien le format CSV que vous désirez utiliser, et utilisez les options par défaut. Vous pouvez alors charger le fichier dans python à l'aide d'`ouvrirCSV` et effectuer les mêmes analyses.

3. Récupérez sur le site et ouvrez de la même façon le fichier `temperatures.csv`, et effectuez les mêmes analyses.

4. Observez la fonction suivante :

```
def mystere(L: list, x: int) -> bool:
    cpt = 0
    for elt in L :
        if elt > x:
            cpt = cpt + 1
            if cpt == 3:
                return True
        else:
            cpt = 0
    return False
```

Que retourne-t-elle lorsqu'on lui passe en paramètres la liste des températures et 30 ? Et lorsque l'on passe 35 au lieu de 30 ? Faites-la tourner dans *Python Tutor* pour bien comprendre ce qui se passe. Quelle est sa complexité ?

3.7 Pour aller plus loin : notion de complexité en moyenne

Dans l'exercice 2.1.3 nous avons effectué des mesures sur des listes, mesures dont l'évaluation nécessite la prise en compte de *tous les éléments de la liste* et, ce, *quel que soit la liste* considérée. Il existe cependant des propriétés qui peuvent être calculées sans nécessairement prendre en compte tous les éléments de la liste considérée. Par exemple, si l'on cherche à évaluer une propriété *"au moins un des éléments de la liste est pair"*, alors on peut arrêter l'évaluation dès qu'un élément pair est rencontré : le résultat est alors déterminé et ne changera pas quelques soient la valeur des autres éléments. Dans le même registre, le code de la fonction suivante est aussi élégant qu'inefficace :

```
def existePairListeInefficace(L: list):        # gaspille de l'energie
    return nbPairsListe(L) != 0
```

Exercice 3.7.1 On considère les fonctions `existePairListe(L)`, qui renvoie `True` si au moins un des nombres de la liste est pair et `False` sinon, et `tousPairsListe(L)` qui renvoie `True` si tous les nombres de la liste sont pairs, et `False` sinon.

Écrire ces deux fonctions en faisant en sorte qu'elles retournent leur résultat dès que celui-ci est déterminé. Testez ces fonctions.

Exercice 3.7.2 On se propose d'approfondir l'exercice 3.7.1

Comparer le nombre de tests réalisés par les fonctions `existePairListe(L)` et `existePairListeInefficace(L)` pour les cas où une liste de 100 éléments est passée en paramètre et que cette liste :

- ne contient aucun nombre pair (appelé complexité dans le pire cas) ;
- ne contient aucun nombre impair (appelé complexité du meilleur cas).

Pour déterminer si une liste contient ou pas un nombre pair, nous avons implémenté deux algorithmes, l'un naïf (exercice 3.7.2) l'autre optimisé (exercice 3.7.1) puis nous avons comparé leurs performances dans les meilleurs et pires cas. Mais en pratique, est-on généralement plus proche du meilleur cas ? du pire cas ? ou bien entre les deux ? En fait, pour comparer les performances de ces deux algorithmes, il est intéressant de comparer le nombre *moyen* de tests utilisés par chaque algorithme pour traiter un ensemble *pertinent* de listes.

Pour définir mathématiquement cette notion d'ensemble pertinent, il est d'usage en informatique de considérer des ensembles regroupant toutes les entrées ayant la même taille. Ici, la

taille de l'entrée de nos deux algorithmes correspond à la longueur de la liste passée en paramètre. Ainsi la question que nous allons résoudre pour chaque algorithme est : « combien de tests nécessite en moyenne cet algorithme pour traiter une liste de n entiers ? ».

Pour l'algorithme naïf (noté $\mathcal{A}_{\text{naïf}}$), qui parcourt systématiquement toute la liste liste_n de n éléments, le coût de traitement de toute liste de n entiers est de n tests. On a donc :

$$\text{Complexité\_Moyenne}\left(\mathcal{A}_{\text{naïf}}(\mathsf{liste}_n)\right) = n$$

Pour analyser l'algorithme optimisé (noté $\mathcal{A}_{\text{optimisé}}$) appliqué à une liste liste_n de n éléments, nous allons simplifier le problème en considérant le cas où les nombres pairs et impairs apparaissent de façon équiprobable dans les listes considérées. Comme on s'intéresse à la parité, on peut de plus restreindre l'analyse aux listes dont les éléments appartiennent à $\{0,1\}$ [2]. On note L_n l'ensemble des listes de n éléments dans $\{0,1\}$. Calculons maintenant le coût de traitement de toutes les listes de L_n en remarquant les faits suivants :

- il y a 2^n listes dans L_n ;
- une liste sur deux de L_n commence par un nombre pair et donc 2^{n-1} listes nécessitent un seul test pour être traitées par l'algorithme optimisé ;
- une liste sur quatre de L_n a son premier nombre pair en deuxième position et donc 2^{n-2} listes nécessitent deux tests ;
- de façon plus générale, il y a 2^{n-i} listes de L_n ayant son premier nombre pair en i-ième position et demandant i tests ;
- le traitement de la seule liste de L_n ne contenant pas de nombre pair nécessite n tests.

Au total il faut $1.2^{n-1} + 2.2^{n-2} + 3.2^{n-3} + \cdots + n.2^{n-n} + n$ tests, le nombre moyen de tests est donc de

$$\frac{\sum_{i=1}^{i=n} i.2^{n-i} + n}{2^n} = \sum_{i=1}^{i=n} \frac{i}{2^i} + \frac{n}{2^n} = 2 - \frac{1}{2^{n-1}}$$

Sous l'hypothèse d'équiprobabilité des nombres pairs et impairs, on a donc :

$$\text{Complexité\_Moyenne}\left(\mathcal{A}_{\text{optimisé}}(\mathsf{liste}_n)\right) = 2 - \frac{1}{2^{n-1}} < 2$$

Exercice 3.7.3 Calculer le nombre *moyen* de tests réalisés par l'algorithme optimisé dans le cas de listes ne contenant qu'un seul nombre pair placé aléatoirement dans la liste.

3.8 L'essentiel du chapitre

3.8.1 Typage

Une bonne pratique est de mettre les types dans les définitions de fonctions :

```
def f(n: int, m: int) -> int:
    if n < m:
        return n
    return m
```

On peut ainsi faire un résumé des fonctions que l'on connait avec leur typage :

2. Cette restriction ne modifie en rien le calcul du nombre de tests à réaliser puisqu'il suffit de transformer les éléments pairs en 0 et les impairs en 1 pour passer d'une liste d'entiers à une liste de 0 et de 1 .

```
len(L: list) -> int
range(fin: int) -> list
range(debut: int, fin: int) -> list
range(debut: int, fin: int, pas: int) -> list
```

Attention, ces indications de typage ne sont à écrire qu'au moment de la *définition* de la fonction, ou quand on documente leur existence comme ci-dessus, il ne faut pas les écrire au moment des *appels* à la fonction.

3.8.2 Complexité

La *complexité* exprime l'ordre de grandeur du nombre d'opérations effectuée par une fonction, en fonction des tailles des paramètres. Typiquement quand une fonction prend en paramètre une liste de taille `n`, la complexité peut être $\mathcal{O}(n)$, $\mathcal{O}(n^2)$, $\mathcal{O}(2^n)$, ... Quand elle prend en paramètre deux listes, l'une de taille n_1 et l'autre de taille n_2, la complexité peut être $\mathcal{O}(n_1+n_2)$, $\mathcal{O}(n_1 \times n_2)$, ...

Chapitre 4. Accès indexé

4.1 Accès indexé dans les listes

Aux chapitres précédents, on accédait aux listes dans l'ordre de leur contenu. On peut cependant aussi utiliser un accès indexé. C'est-à-dire que lorsque l'on écrit :

```
x = L[2]
```

la valeur écrite dans `x` est celle de l'élément d'indice `2` de la liste `L`, on n'a pas besoin de parcourir la liste, on peut directement accéder à l'élément d'indice `2`. Tout comme pour `range`, les indices commencent à partir de `0`, c'est donc sur cet exemple le *troisième* élément de la liste que l'on récupère ainsi, le premier élément étant d'indice 0, le second d'indice 1, etc.

Par exemple la fonction suivante :

```
def sommeListe(L: list) -> int:
    s = 0
    for x in L:
        s = s + x
    return s
```

peut ainsi s'écrire aussi sous la forme :

```
def sommeListe2(L: list) -> int:
    s = 0
    for i in range(len(L)):
        s = s + L[i]
    return s
```

Dans `sommeListe`, on disait à Python de récupérer un à un les éléments de la liste `L`. Dans `sommeListe2` on lui demande seulement que `i` prenne les valeurs `0`, `1`, ..., `len(L)-1`, et l'on récupère alors l'élément d'indice `i` de la liste `L`. Cela revient au final bien au même.

L'intérêt d'utiliser un accès indexé, c'est que l'on a un meilleur contrôle sur le parcours de la liste. Par exemple, on peut calculer la somme de la première moitié de la liste seulement :

```
def sommeListeMoitie(L: list) -> int:
    s = 0
    for i in range(len(L) // 2):
        s = s + L[i]
    return s
```

On peut aussi parcourir deux listes en même temps (supposées de même taille), pour calculer un produit scalaire par exemple :

```
def produitScalaire(L1: list, L2: list) -> int:
    s = 0
    for i in range(len(L1)):
        s = s + L1[i] * L2[i]
    return s
```

On peut également *écrire* dans la liste avec l'affectation :

```
L[2] = 0
```

écrit 0 à l'indice 2 de la liste L. Attention, la liste est *modifiée*, ainsi la fonction :

```
def metAZero(L: list):
    for i in range(len(L)):
        L[i] = 0
```

modifie la liste qui a été passée en paramètre
Enfin, on peut facilement construire une liste ainsi :

```
maListe = [0] * 1000
```

qui "multiplie" donc par 1000 la liste contenant un seul élément 0, produisant ainsi une liste de 1000 éléments tous égaux à 0.

Exercice 4.1.1 Écrire une fonction `maximumListe(L: list) -> int` qui retourne le maximum des nombres contenus dans la liste. Attention, par rapport à l'exercice 2.1.3, cette fois-ci on ne suppose pas que les nombres sont forcément positifs.

Exercice 4.1.2 Écrire une fonction `position(L: list, x: int) -> int` qui retourne l'indice de la première apparition du nombre `x` dans la liste `L`, s'il y apparaît, `-1` sinon.

Exercice 4.1.3 Écrire une fonction `positionDernier(L: list, x: int) -> int` qui retourne l'indice de la *dernière* apparition du nombre `x` dans la liste `L`, s'il y apparaît, `-1` sinon.

Exercice 4.1.4 Écrire une fonction `absListe(L: list)` qui modifie la liste pour remplacer chaque élément par sa valeur absolue (en utilisant la fonction `abs`). Note : puisque la fonction modifie la liste, elle n'a pas besoin de retourner une valeur, il n'y a donc pas d'instruction `return` à mettre, la fonction se termine simplement en retournant rien (`None`).

Exercice 4.1.5 Taper les instructions suivantes dans Python Tutor et prenez le temps d'expliquer précisément l'évolution des variables pendant l'exécution pas à pas de ces instructions.

```
L = [1] * 10
L[1] = 3

for i in range(10):
    L[i] = i*i
x = L[3]
```

Exercice 4.1.6 Écrire une fonction `sommeListes(L1: list, L2: list) -> list` qui calcule la somme de deux listes (supposées de même taille) élément par élément, c'est-à-dire que `sommeListes([34,54],[10,11])` doit retourner la liste `[44,65]`.

4.2 Exercices de révisions et compléments

Exercice 4.2.1 Écrire une fonction `inverseListe(L: list)` qui "inverse" l'ordre de la liste : elle crée une nouvelle liste dont le premier élément est le dernier de `L`, le deuxième est l'avant-dernier de `L`, etc. jusqu'au dernier élément qui est le premier de `L`.

Exercice 4.2.2 Écrire une fonction `alterne(L1: list, L2: list) -> list` qui, étant données deux listes `L1` et `L2` supposées de même taille, construit une liste deux fois plus grande, contenant l'alternance des éléments de `L1` et de `L2`.

Exercice 4.2.3 Écrire une fonction `estTrie(L: list) -> bool` qui retourne `True` si la liste `L` est triée dans l'ordre croissante, et `False` sinon.

Exercice 4.2.4 Longueur de monotonie

- Écrire une fonction `compte(L: list, n: int) -> int` qui compte le nombre d'apparition du nombre `n` dans la liste `L`.

Une monotonie est une succession de valeurs identiques (possiblement une seule valeur).

- Écrire une fonction `tailleMonotonie(L: list, n: int) -> int` qui retourne la longueur de la plus grande monotonie de `n` dans la liste `L`.
- Écrire une fonction `plusGrandMonotonie(L: list) -> int` qui retourne l'entier de la liste ayant la plus grande monotonie. En cas d'égalité, elle doit retourner celui qui apparaît en premier dans la liste.

4.3 Pour aller plus loin : Les chaînes de caractères

En plus des listes, Python permet de manipuler des chaînes de caractères, autrement dit, du texte !

Par exemple :

```
s = "abc"
```

s est alors une chaîne de caractères contenant le caractère `"a"` à l'indice 0, le caractère `"b"` à l'indice 1, et le caractère `"c"` à l'indice 2.

On peut également parcourir la chaîne de caractères comme une liste :

```
for c in s:
    print(c)
```

ou de manière indexée :

```
for i in range(len(s)):
    print(s[i])
```

En anglais les chaînes de caractères s'appellent *string*, Python affichera donc pour elles le type `str`.

4.3.1 Parcours de chaînes de caractères

Exercice 4.3.1 Écrire une fonction `nombreA(s: str) -> int` qui retourne le nombre d'apparitions du caractère `"a"` dans la chaîne de caractères `s`.

Exercice 4.3.2 Écrire une fonction `nombreQU(s: str) -> int` qui retourne le nombre d'apparitions des caractères `"q"` et `"u"` l'un après l'autre consécutivement dans la chaîne de caractères `s`.

4.3.2 Fabrication de nouvelles chaînes de caractères

Contrairement aux listes, les chaînes de caractères ne sont pas modifiables :

```
s[0] = "a"
```

nous répond

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

Du coup pour produire une nouvelle chaîne de caractères, on est obligé d'en fabriquer une nouvelle, morceau par morceau, par exemple ce code qui duplique les lettres de la chaîne `s` :

```
s = "abc"
s2 = ""
for c in s:
    s2 = s2 + c + c
```

Exercice 4.3.3 Écrire une fonction `padIpadO(s: str) -> str` qui retourne une nouvelle chaîne contenant la même chose que `s` sauf que tous les caractères `"i"` et `"o"` sont remplacés par le caractère `"a"`.

Exercice 4.3.4 On peut aussi créer des listes de chaînes de caractères !

Taper les instructions suivantes et observez :

```
L = [ "zéro", "un", "deux", "trois" ]
x = L[1]
L[3] = "quatre!"
```

Écrire une fonction `nomJour(i: int) -> str` qui reçoit en paramètre le numéro d'un jour de la semaine (entre 1 et 7) et qui renvoie une chaîne de caractères contenant le nom de ce jour.

Exercice 4.3.5 Écrire une fonction `inverseChaine(s: str) -> str` qui retourne une nouvelle chaîne contenant le même contenu que `s`, mais dans l'ordre inverse : le premier caractère de `s` se retrouve à la fin de la chaîne retournée, le deuxième de `s` se retrouve en avant-dernière position de la chaîne, etc. et inversement le dernier caractère de `s` se retrouve au début de la chaîne retournée, l'avant-dernier caractère de `s` se retrouve en deuxième position de la chaîne retournée, etc.

Exercice 4.3.6 Chiffre de César

Le principe du chiffre de César est de chiffrer un texte en remplaçant chacun de ses caractères par le caractère qui est 13 positions plus loin dans l'alphabet. Ainsi tous les caractères `"a"` deviennent `"n"`, tous les caractères `"b"` deviennent `"o"`, etc., et inversement tous les caractères `"n"` deviennent `"a"`, tous les caractères `"o"` deviennent `"b"`, etc.

Écrire une fonction `cesar(s: str) -> str` qui retourne une version chiffrée de la chaîne `s` avec le chiffre de César. On supposera pour simplifier que la chaîne ne contient que des lettres minuscules ou des espaces (ces derniers ne seront pas modifiés par le chiffrage). Que se passe-t-il si l'on l'appelle deux fois sur la même chaîne ? Que signifie le message chiffré `obawbhe` ?

Note : plutôt qu'énumérer tous les caractères possibles, on peut utiliser d'une part la fonction prédéfinie `ord(c: str) -> int` qui retourne le numéro du caractère `c`, et d'autre part `chr(i: int) -> str` qui retourne le caractère qui a pour numéro `i`.

Exercice 4.3.7 Écrire une fonction `nbMots(s: str) -> int` qui retourne le nombre de mots contenus dans la chaîne `s`. On supposera pour simplifier qu'ils sont simplement séparés par une espace.

Exercice 4.3.8 Écrire une fonction `niOuiNiNon(s: str) -> bool` qui retourne `True` si la chaîne `s` ne contient ni le mot `oui` ni le mot `non`, et `False` sinon. Pour simplifier on ne cherchera que ces mots en minuscule.

4.4 L'essentiel du chapitre

4.4.1 Listes

On peut accéder à l'élément d'indice `i` d'une liste `L` avec la notation `L[i]`. On a ainsi deux manières de parcourir une liste :

```
for x in L:
    ... x ...
```

```
for i in range(len(L)):
    ... L[i] ...
```

La deuxième forme est nécessaire lorsque l'on a besoin de jouer sur les indices. Sinon, la première forme est plus simple à écrire.

On peut construire une liste en "multipliant" simplement une liste contenant un seul élément, par exemple pour construire une liste contenant 1000 fois l'élément 0 :

```
maListe = [0] * 1000
```

Attention : il y a donc deux significations pour les caractères `[]` :

- `[x]` seul, construit la liste contenant le nombre `x`.
- `L[i]` avec `L` qui est une liste, récupère l'élément d'indice `i` dans la liste `L`.

4.4.2 Types

Il ne faut pas confondre les différents types :

- Les indices au sein des listes et chaînes de caractères (que l'on appelle en général plutôt `i`)
- Les listes (que l'on appelle en général plutôt `L`)
- Les éléments contenus dans les listes (que l'on appelle en général plutôt `x`, `element`, ...)

Chapitre 5. Dessin d'images

Une image, en informatique, est un simple tableau à deux dimensions de points colorés (appelés pixels, *picture elements*).

Les coordonnées (x, y) d'un pixel expriment sa position au sein de l'image : x est son abscisse, en partant de la gauche de l'image, et y est son ordonnée, en partant du haut de l'image (à l'inverse de l'ordonnée mathématique, donc). Elles partent toutes deux de 0. Le schéma ci-dessous montre un exemple d'image en gris sur fond blanc, de 7 pixels de largeur et 4 pixels de hauteur, les abscisses vont donc de 0 à 6 et les ordonnées vont de 0 à 3.

0,0	1,0	2,0	3,0	4,0	5,0	6,0
0,1	1,1	2,1	3,1	4,1	5,1	6,1
0,2	1,2	2,2	3,2	4,2	5,2	6,2
0,3	1,3	2,3	3,3	4,3	5,3	6,3

La couleur RGB (r, g, b) d'un pixel est la quantité de rouge (r pour *red*), vert (g pour *green*) et de bleu (b pour *blue*) composant la couleur affichée à l'écran. Le mélange se fait comme trois lampes colorées rouge, vert et bleue, et les trois valeurs r, g, b expriment les intensités lumineuses de chacune de ces trois lampes, exprimées entre 0 et 255. Par exemple, $(0, 0, 0)$ correspond à trois lampes éteintes, et produit donc du noir. $(255, 255, 255)$ correspond à trois lampes allumées au maximum [1], et produit donc du blanc. $(255, 0, 0)$ correspond à seulement une lampe rouge allumée au maximum, et produit donc du rouge. $(255, 255, 0)$ correspond à une lampe rouge et une lampe verte allumées au maximum, et produit donc du jaune, et ainsi de suite. Cela correspond très précisément à ce qui se passe sur vos écrans ! Si vous prenez une loupe, vous verrez que l'écran est composé de petits points (appelés *sous-pixels*) verts, rouges et bleus, allumés plus ou moins fortement pour former les couleurs.

Dans ce chapitre et le suivant, nous étudierons deux grands types d'opérations disponibles typiquement dans les logiciels de manipulation d'image : nous produirons d'abord des images de zéro (synthèse d'images), puis nous transformerons des images existantes (traitement d'images, ou filtres).

1. Le maximum de ce qu'elles peuvent produire. Même s'ils peuvent ainsi afficher des millions de couleurs différentes, nos écrans restent limités : https://www.youtube.com/watch?v=_NzVmtbPOrM

5.1 Mise en jambes

python permet bien sûr de manipuler des images, à l'aide de la *python Imaging Library* (PIL), qui est préinstallée sur vos machines. Nous vous fournissons un module `bibimages.py` qui permet d'utiliser plus simplement la bibliothèque PIL. Ce module est disponible en téléchargement sur le site https://moodle.u-bordeaux.fr/course/view.php?id=14677. Allez sur ce site, retrouvez dans la section du chapitre 5 le fichier `bibimages.py`, utiliser un clic droit dessus et utilisez "enregistrer sous" pour l'enregistrer à côté de vos autres fichiers python. Ce module comporte aussi une poignée de fonctions qui permettent de manipuler les images. Pour utiliser un module il faut commencer par l'*importer*, et toute session de travail sur les images doit commencer par la phrase magique :

```
from bibimages import *
```

Voici un résumé des fonctions que nous utiliserons dans ce chapitre :

`ouvrirImage(nom:str) -> image`	Ouvre le fichier *nom* et retourne l'image qu'il contient, par exemple : `img = ouvrirImage("teapot.png")`
`nouvelleImage(large:int, haut:int) -> image`	Retourne une image de taille $large \times haut$, initialement noire, par exemple : `img = nouvelleImage(300, 200)`
`afficherImage(img:image)`	Affiche l'image *img*, par exemple : `afficherImage(img)`
`colorierPixel(img:image, x:int, y:int, (r,g,b))`	Peint le pixel (x, y) dans l'image *img* de la couleur (r, g, b), par exemple : `colorierPixel(img, 10, 10, (255, 0, 0))`
`largeurImage(img:image)`	Retourne le nombre de colonnes de pixels contenues dans *img*, par exemple : `l = largeurImage(img)`
`hauteurImage(img:image)`	Retourne le nombre de lignes de pixels contenues dans *img*, par exemple : `h = hauteurImage(img)`

Exercice 5.1.1 TP Après avoir suivi le début de cette page pour mettre en uvre `bibimages`, exécuter les instructions suivantes :

```
monImage = nouvelleImage(300,200)
colorierPixel(monImage, 10, 10, (255,255,255))
afficherImage(monImage)
```

Expliquer ce qu'effectue chaque instruction, et observer l'image produite. Confirmer ainsi le sens dans lequel fonctionnent les coordonnées.

Peindre le pixel de coordonnées $(250, 150)$ en vert (n'oubliez pas d'appeler de nouveau `afficherImage(monImage)` pour afficher le résultat). Peindre le pixel de coordonnées $(50, 150)$ en violet.

Que contiennent les variables `l` et de `h` après avoir exécuté les instructions suivantes ?

```
l = largeurImage(monImage)
h = hauteurImage(monImage)
```

Récupérez l'image de la célèbre théière de l'Utah (voir Wikipedia) sur la page des supports du site

https://moodle.u-bordeaux.fr/course/view.php?id=14677
et sauvegardez-la à côté de vos fichiers `.py`. Exécuter les instructions suivantes :

```
theiere = ouvrirImage("teapot.png")
afficherImage(theiere)
```

Télécharger au moins une autre image à partir de la même page, et vérifiez que vous pouvez également l'ouvrir.

Vous pourrez bien sûr, pour tous les exercices, utiliser d'autres images venant d'Internet ou d'ailleurs (pour autant que les licences sous lesquelles lesdites images sont placées vous le permettent !).

Rappel :

Une chaîne de caractères est une suite de valeurs numériques élémentaires, dont chacune représente un caractère. En python, on définit une chaîne de caractères comme un ensemble de caractères placés entre deux caractères "guillemets" ou "apostrophe" : `"Bonjour"`, `'tout le monde'`. De nombreuses fonctions permettent de manipuler des chaînes de caractères : la fonction `print` permet d'afficher une chaîne de caractères, l'opérateur "`+`" permet de concaténer deux chaînes pour en former une nouvelle, etc.

Ici, `"teapot.png"` est une chaîne de caractères contenant le nom du fichier à ouvrir. python ne cherche pas à comprendre ce qu'il y a entre les guillemets. Si par contre on oublie les guillemets, python va chercher une variable appelée `teapot`, ce qui n'est pas du tout ce que l'on veut.

5.2 Tracés de segments

Exercice 5.2.1 Quelles sont les coordonnées des pixels d'une ligne horizontale au milieu d'une image de hauteur `hauteur` et de largeur `largeur` ? Bien spécifier le premier et le dernier pixel. Laquelle parmi les deux coordonnées reste constante et laquelle est variable ?

Écrire une fonction `ligneHorizontaleBlancheAuMilieu(img)` qui utilise une boucle `for` et `range` pour dessiner une ligne horizontale blanche au milieu de l'image `img` (séparant donc l'image donnée en deux parties). Testez-la sur la théière, sur une image noire (générée par un appel à `nouvelleImage`), ainsi que sur au moins une autre image. Notez que notre fonctionne ne *retourne rien* : en effet, elle *modifie* l'image passée en paramètre, elle n'a donc pas besoin de la retourner.

Exercice 5.2.2 Écrire une fonction `ligneHorizontaleAuMilieu(img,c)` qui a donc un paramètre en plus, `c`, qui permet ainsi de désormais choisir la couleur de la ligne lors de l'appel. En vous inspirant de la fonction précédente, il suffit de remplacer, dans le corps de la fonction, la couleur blanche `(255,255,255)` par la couleur `c`. Tester :

```
ligneHorizontaleAuMilieu(monImage, (255,0,0))
afficherImage(monImage)
ligneHorizontaleAuMilieu(monImage, (0,255,0))
afficherImage(monImage)
```

La fonction récupère donc l'ensemble des trois valeurs r, g, et b dans le seul paramètre `c`, qu'elle peut passer tel quel à `colorierPixel`.

Exercice 5.2.3 Écrire une fonction `ligneHorizontale(img,c,y)` qui dessine, dans une image `img` donnée, une ligne horizontale de couleur `c` à la distance `y` du haut de l'image.

Vous pouvez l'appeler plusieurs fois sur une même image avec des paramètres différents pour constater le résultat. C'est tout l'intérêt d'avoir ajouté des paramètres : on n'a pas besoin de modifier la fonction pour obtenir un dessin évolué.

Que se produit-il si l'on appelle la fonction `ligneHorizontale(img,c,y)` avec une valeur de `y` qui dépasse la hauteur de l'image `img` ?

Améliorer le code de votre fonction afin de tester d'abord si la valeur de `y` est valide. Si ce n'est pas le cas, votre fonction devra ne rien dessiner.

Donner une nouvelle version de la fonction `ligneHorizontaleAuMilieu2(img,c)` qui dessine une ligne horizontale de couleur `c` au milieu de l'image `img` en ne faisant qu'appeler la fonction `ligneHorizontale(img,c,y)`.

Exercice 5.2.4 En faisant appel à la fonction `ligneHorizontale`, écrire une fonction `grilleHorizontale(img, c, d)` qui dessine une grille de lignes horizontales espacées par `d` pixels.

Note : profitez de la fonction `range(debut,fin,pas)`, qui est toute prête à vous fournir la liste exacte des ordonnées concernées.

Par exemple, `grilleHorizontale(monImage,(255,255,0),10)` dessinerait dans `monImage` des lignes jaunes avec pour ordonnées 0, 10, 20, 30, etc.

Exercice 5.2.5 Écrire une fonction `rectangleCreux(img,x1,x2,y1,y2,c)` qui dessine les côtés d'un rectangle de couleur `c`, les coins du rectangles étant les pixels de coordonnées $(x1, y1)$, $(x2, y1)$, $(x1, y2)$, $(x2, y2)$. Il s'agit bien sûr de dessiner quatre lignes formant les côtés du rectangle. Testez-la plusieurs fois avec des coordonnées différentes et des couleurs différentes, sur une image noire et sur la photo de théière.

Que se passe-t-il si l'un des paramètres dépasse de la taille de l'image ? Est-il facile de corriger le problème ?

Exercice 5.2.6 Qu'effectue la fonction suivante ? N'hésitez pas à la copier depuis le PDF disponible sur le site du cours et à la coller dans Python Tutor.

```
def remplir(img, c):
    l = largeurImage(img)
    h = hauteurImage(img)
    for x in range(l):
        for y in range(h):
            colorierPixel(img, x, y, c)
```

Décrire précisément ce qu'effectue chaque ligne, testez-la. Est-ce que les pixels sont coloriés ligne par ligne ou colonne par colonne ? Si le premier pixel colorié est celui de coordonnées $(0, 0)$, quelles sont les coordonnées du pixel colorié en deuxième, $(0, 1)$ ou $(1, 0)$?

Mêmes questions pour la version suivante :

```
def remplirBis(img, c):
    l = largeurImage(img)
    h = hauteurImage(img)
    for y in range(h):
        for x in range(l):
            colorierPixel(img, x, y, c)
```

Définir une fonction `remplir2(img,c)` produisant le même résultat, en utilisant la fonction `ligneHorizontale`.

Serait-il possible possible d'obtenir le même effet avec la fonction `grilleHorizontale` ?

Exercice 5.2.7 En vous inspirant très largement de la fonction `remplir`, écrivez une fonction `rectanglePlein(img,x1,x2,y1,y2,c)` qui dessine un rectangle plein de couleur `c` dont les coins sont les pixels de coordonnées $(x1, y1)$, $(x2, y1)$, $(x1, y2)$, $(x2, y2)$.

Exercice 5.2.8 On veut colorier toute une image `img` en gris. On a commencé par taper ceci :

```
#code a completer
c = (127,127,127)
l = largeurImage(img)
h = hauteurImage(img)
```

et il s'agit maintenant de parcourir toute l'image pour la colorier. Parmi les codes suivants, lesquels sont corrects ? Rayer les codes erronés.

```
for x in range(h):
  for y in range(l):
    colorierPixel(img, x, y, c)
```

```
for x in range(1,l,1):
  for y in range(1,h,1):
    colorierPixel(img, x, y, c)
```

```
for y in range(h):
  for x in range(l):
    colorierPixel(img,x, y, c)
```

```
for x in range(1,l+1):
  for y in range(1,h+1):
    colorierPixel(img, x, y, c)
```

```
for x in range(l) and y in range(h):
  colorierPixel(img, x, y, c)
```

Exercice 5.2.9 Écrire une fonction `remplirDegradeGris(img)` qui remplit l'image `img` avec un dégradé de gris depuis du noir en haut à du blanc en bas : tous les pixels de la ligne tout en haut de l'image doivent être noirs, et tous les pixels de la ligne tout en bas de l'image doivent être blancs, et les pixels des lignes intermédiaires, d'un niveau de gris intermédiaire, en dégradé.

Que suffit-il de changer pour que le dégradé soit de la gauche vers la droite ?

Que suffit-il de changer pour que le dégradé soit du bas vers le haut ?

5.3 Exercices de révisions et compléments

Exercice 5.3.1 Écrire une fonction `ligneVerticale(img,c,x)` qui dessine, dans une image `img` donnée, une ligne verticale de couleur `c` à la distance `x` du bord gauche de l'image.

En faisant appel à celle-ci, écrire une fonction `grilleVerticale(img,c,d)` qui dessine une grille de lignes verticales espacées par `d` pixels.

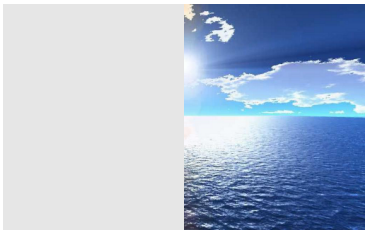
Exercice 5.3.2 Écrire une fonction `remplirRougeVertJaune(img)` qui remplit l'image `img` avec un dégradé de couleurs : le coin en haut à gauche doit être noir, le coin en bas à gauche doit être vert, le coin en haut à droite doit être rouge, et le coin en bas à droite doit être jaune (le jaune est le mélange à part égale de rouge et de vert).

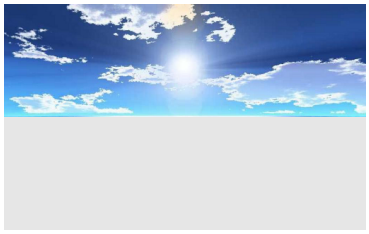
De la même façon, écrivez `remplirRougeBleuViolet(img)` et `remplirVertBleuCyan(img)`

Exercice 5.3.3 En faisant appel aux fonctions `grilleHorizontale(img,c,d)` et `grilleVerticale(img,c,d)`, écrire une fonction `grilleCarree(img,c,d)` qui dessine dans une image une grille carrée d'espacement `d` pixels. Tester avec plusieurs images et plusieurs valeurs de `d`.

Exercice 5.3.4 (extrait DS 2015-16) On souhaite masquer la moitié d'une image à l'aide d'une couleur. Voici un exemple :


image initiale


partie gauche masquée


partie inférieure masquée

1. Écrire la fonction `masquerGauche(img,c)` qui remplit de la couleur `c` la moitié gauche de l'image `img` et laisse inchangé le reste de l'image.

2. Écrire la fonction `masquerBas(img,c)` qui remplit de la couleur `c` la moitié inférieure de l'image `img` et laisse inchangé le reste de l'image.

Exercice 5.3.5 Écrire une fonction `diagonale(img)` qui dessine en blanc la ligne diagonale entre le coin en haut à gauche et le coin en bas à droite de l'image. Tester avec différentes tailles d'image, et corriger l'erreur que vous avez commise.

Exercice 5.3.6 On souhaite ajouter des lignes verticales noires à une image.

Écrire une fonction `emprisonner(img, nbBarreaux, epaisseur)` qui ajoute `nbBarreaux` lignes verticales d'`epaisseur` pixels et réparties de façon homogène dans l'image.

Exercice 5.3.7 En vous souvenant du schéma du cercle trigonométrique pour écrire l'équation paramétrique de la courbe du cercle, écrire une fonction `cercle(img,x,y,r)` qui dessine un cercle de rayon r dont le centre a pour coordonnées (x, y).

Note : Les outils trigonométriques (`cos`, `sin`, `pi`, ...) sont disponibles en tapant :

```
from math import *
```

Exercice 5.3.8 On souhaite masquer une image en diagonale à 45 degré à l'aide d'une couleur. Voici un exemple :


image initiale


partie diagonale masquée

1. Écrire la fonction `masquerDiagonale(img, c)` qui remplit de la couleur `c` le triangle inférieur gauche de l'image `img` et laisse inchangé le reste de l'image.

2. Est-ce que votre fonction est correcte pour une image qui serait plus haute que large ? Sinon, corrigez.

Exercice 5.3.9 Écrire une fonction `remplirDegradeDiagonaleBleu(img)` qui remplit l'image `img` avec un dégradé de bleu en diagonale : le coin en haut à gauche doit être noir, le coin en bas à droite doit être bleu.

Écrire une fonction `remplirDegradeDiagonaleBleuInverse(img)` qui remplit l'image `img` avec un dégradé de bleu en diagonale dans l'autre sens : le coin en haut à gauche doit être bleu, le coin en bas à droite doit être noir.

En combinant les formules utilisées pour `remplirRougeVertJaune` et pour `remplirDegradeDiagonaleBleu`, écrire une fonction `remplirCouleurs(img)` qui remplit l'image `img` avec un dégradé entre les différentes couleurs : le coin en haut à gauche doit être bleu, le coin en bas à gauche doit être vert, le coin en haut à droite doit être rouge, et le coin en bas à droite doit être jaune.

5.4 L'essentiel du chapitre

On peut ou bien ouvrir une image existante dans un fichier :
`monImage = ouvrirImage("fichier.png")`
ou bien créer une image :
`monImage = nouvelleImage(300, 200)`

Dans les deux cas on peut ensuite colorier des pixels :
`colorierPixel(monImage, 50, 50, (100, 0, 0))`

Pour connaitre la taille de l'image, on utilise les fonctions `largeurImage` et `hauteurImage`.

Ainsi, une fonction typique qui trace un dessin dans une image est :

```
def f(img, n):
    l = largeurImage(img)
    h = hauterImage(img)
    for x in range(l):
        y = [...]
        c = [...]
        colorierPixel(img, x, y, c)
```

Et on l'utilise par exemple ainsi :

```
monImage = nouvelleImage(300, 200)
f(monImage)
afficherImage(monImage)
```

Note : la fonction `f` prend une image, et non le nom d'un fichier. Cela permet ainsi d'appeler `f` plusieurs fois, possiblement avec la même image et différents autres arguments, les effets des appels s'accumulant dans l'image, ou possiblement avec d'autres images, etc. à volonté.

Chapitre 6. Manipulations d'images

6.1 Manipulations d'images

Jusqu'ici on a seulement peint des pixels, on peut également récupérer la couleur d'un pixel :

`couleurPixel(img:image, x:int, y:int)`	Retourne la couleur du pixel (x, y) dans l'image *img*, par exemple : `(r,g,b) = couleurPixel(img, 10, 10)`

Il est très courant d'appliquer un *filtre* sur une image, c'est-à-dire un calcul sur chacun des pixels de l'image. L'instruction

```
(r,g,b) = couleurPixel(img, 10, 10)
```

récupère les valeurs r, g et b de la couleur du pixel de coordonnées $(10, 10)$. Par exemple, pour ne conserver que la composante rouge du pixel $(10, 10)$, on peut utiliser :

```
colorierPixel(img, 10, 10, (r,0,0))
```

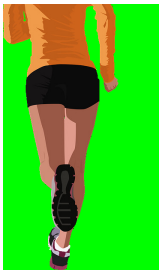
Exercice 6.1.1 Écrire une fonction `filtreRouge(img:image)` qui, pour chaque pixel de l'image, ne conserve que la composante rouge comme expliqué ci-dessus. Testez-la sur la photo de théière. Faites de même pour le vert et le bleu et affichez les trois résultats ainsi que l'image d'origine côte à côte. Remarquez notamment que pour le bleu il n'y a pas d'ombre en bas à droite. En effet, la source lumineuse en haut à gauche ne contient pas de bleu.

Exercice 6.1.2 Écrire une fonction `modifierLuminosite(img:image,facteur:float)` qui pour chaque pixel multiplie par *facteur* les valeurs des trois composantes r, g, et b. Remarquez que la fonction `colorierPixel` n'apprécie pas que l'on donne des valeurs non entières. Utilisez donc la fonction `int(truc)` qui arrondit `truc` à l'entier inférieur.

Testez les facteurs `2`, `1.2`, `0.8`, `0.5` sur la photo de théière (n'oubliez pas de recharger la photo à chaque fois pour éviter de cumuler les effets).

Exercice 6.1.3 On dispose d'une photographie d'un personnage prise sur un fond vert (sur la photocopie le vert apparaît en gris clair, l'image est disponible sur le site du cours). On souhaite incruster ce personnage sur un autre fond comme dans l'exemple suivant. On va procéder étape par étape, en commençant par une version simpliste qui ne réalisera pas complètement l'objectif, puis on va la corriger petit à petit pour obtenir le résultat voulu.

Les images `personnage.png` et `fond.jpg` sont disponibles en téléchargement sur le site du cours.



`personnage.png`

`fond.jpg`

incrustation

1. Écrire et tester une fonction `copieCoinImage(avantPlan, fond)` qui copie l'image `avantPlan` dans l'image `fond`. On place l'image en avant plan en haut à gauche de l'image de fond (on fait correspondre leurs coins supérieurs gauches). On suppose également que l'image de fond est assez grande pour contenir toute l'image à copier.

2. Écrire et tester une fonction `copieImage(avantPlan, fond, dx, dy)` qui copie l'image `avantPlan` dans l'image `fond` en faisant correspondre le pixel `(0,0)` de l'image d'avant plan avec le pixel `(dx,dy)` de l'image de fond, i.e. on place l'image d'avant-plan à un endroit voulu dans l'image de fond.

3. On suppose que le fond vert, qui ne doit pas apparaître dans l'image résultat, est composé uniquement de pixels de couleur (0,255,0). Écrire et tester une fonction `incrusterImage(avantPlan, fond, dx, dy)` qui incruste sans copier les pixels verts l'image `avantPlan` dans l'image `fond` tout en faisant correspondre le pixel `(0,0)` de l'image d'avant plan avec le pixel `(dx,dy)` de l'image de fond.

4. On souhaite fabriquer l'image donnée en exemple. Il s'agit maintenant de centrer horizontalement le coureur qui incruste l'image `imagePersonnage` dans l'image `fond`.

5. On souhaite maintenant ajouter un cadre en pointillé autour de l'incrustation. Écrire une fonction `rectanglePointille(image,x1,y1,x2,y2,c)` qui dessine dans l'image le rectangle de couleur `c` défini par `(x1,y1,x2,y2)` en n'écrivant qu'un pixel sur deux.

Exercice 6.1.4 Écrire une fonction `monochrome(img)` qui pour chaque pixel, calcule la moyenne $lum = \frac{r+g+b}{3}$ des composantes r, g, b, et peint le pixel de la couleur (lum, lum, lum).

En observant bien, les éléments verts semblent cependant plus foncés que sur la photo d'origine. L'œil humain est effectivement peu sensible au bleu et beaucoup plus au vert. Une conversion plus ressemblante est donc d'utiliser plutôt $l = 0.3 * r + 0.59 * g + 0.11 * b$. Essayez, et constatez que les éléments verts ont une luminosité plus fidèle à la version couleur.

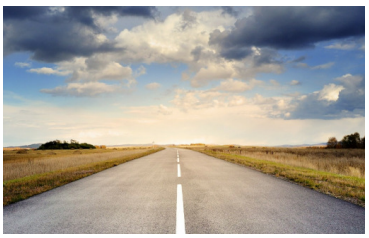
Exercice 6.1.5 Écrire une fonction `noirEtBlanc(img)` qui convertit une image monochrome, telle que produite par la fonction `monochrome` de l'exercice 6.1.4, en une image noir et blanc : chaque pixel peut valoir (0,0,0) ou (255,255,255) selon que la luminosité est plus petite ou plus grande que 127.

Cette conversion ne permet néanmoins pas de représenter les variations douces de luminosité. L'exercice difficile 6.2.5 présente une méthode plus avancée résolvant cette limitation.

6.2 Exercices de révisions et compléments

Exercice 6.2.1 On cherche à réaliser la fusion de deux images supposées de même taille, comme dans l'exemple suivant.

`img1`



`img2`

`img3`

Écrire une fonction `fusion(img1,img2,img3)` qui place dans `img3` l'image correspondant à la moyenne, pixel à pixel, des images `img1` et `img2` (supposées de même taille).

On pourra tester les deux images `ocean.png` et `road2.jpg` disponibles sur le site du cours.

Exercice 6.2.2 L'effet de postérisation est obtenu en diminuant le nombre de couleurs d'une image. Une manière simple de l'obtenir est d'arrondir les composantes r, g, b des pixels de l'image à des multiples d'un entier, par exemple des multiples de 64. Écrivez donc une fonction `posteriser(img,n)` qui arrondit les composantes des pixels de l'image à un multiple de n. Essayez sur une photo avec $n = 64, 128, 150, 200$.

Exercice 6.2.3 Une manière simple de rendre une image floue est d'utiliser l'opération moyenne. Écrire une fonction `flou(img)` qui pour chaque pixel n'étant pas sur les bords, le peint de la couleur moyenne des pixels voisins [1]. Comparer le résultat à l'original. Pourquoi faut-il plutôt passer une deuxième image en paramètre à la fonction, et ne pas toucher à la première image ?

Exercice 6.2.4 Écrire une fonction `emprisonnerJoliment(img:image, nbBarreaux:int, epaisseur:int)` qui améliore le rendu de l'exercice 5.3.6 en faisant apparaître un dégradé de gris sur chaque barreau. Pour calculer la couleur d'un pixel d'un barreau, on pourra utiliser une formule telle que

$$\text{couleur} = \frac{\text{distance au centre du barreau} * 255}{\text{épaisseur}} \ .$$

Exercice 6.2.5 Comme vous l'avez constaté dans l'exercice 6.1.5, la qualité des images produites par la fonction `noirEtBlanc` laisse à désirer. L'algorithme proposé par Floyd et Steinberg, permet de limiter la perte d'information due à la quantification des pixels en blanc ou noir. Écrire une fonction `floydSteinberg(img:image)` qui convertit une image monochrome en noir et blanc à l'aide de l'algorithme de Floyd et Steinberg (cf. page Wikipedia : http://fr.wikipedia.org/wiki/Algorithme_de_Floyd-Steinberg). Note : concernant la fonction `couleur_la_plus_proche()`, dans cet exercice on la fait retourner seulement soit du noir, soit du blanc.

Manipulations géométriques

Exercice 6.2.6 (extrait DS 2014-15) Un reporter a pris plusieurs photographies d'un paysage. Elles sont de même hauteur mais montrent des points de vue décalés. Il souhaite maintenant les mettre côte à côte pour obtenir une image panoramique de ce paysage (sans se soucier des problèmes de perspective).

1. Écrire une fonction `assembler(imgGauche, imgDroite, imgPano)` qui place dans `imgPano` l'image correspondant à l'assemblage horizontal des images `imgGauche` et `imgDroite`. Quelle doit être la largeur de l'image `imgPano` passée en paramètre ?

2. Écrire une fonction `assemblerTrois(imgGauche, imgCentre, imgDroite, imgPano)` qui assemble trois images horizontalement dans `imgPano`, en utilisant la fonction précédente.

Exercice 6.2.7 (extrait DST 2015-16) L'objectif de cet exercice est d'écrire le code de deux fonctions Python permettant d'effectuer la transformation miroir (symétrie axiale) d'une image. Voici un exemple :

1. Idéalement, pour un résultat réellement correct, il faudrait en fait pour chaque composante prendre la racine carrée de la moyenne des carrés des pixels voisins, voir https://www.youtube.com/watch?v=LKnqECcg6Gw, mais contentez-vous d'une simple moyenne, cela sera déjà bien.

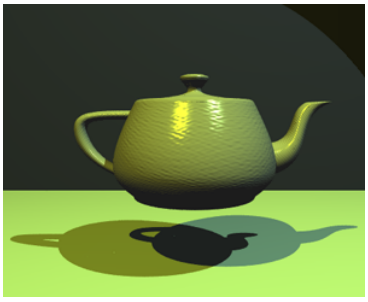
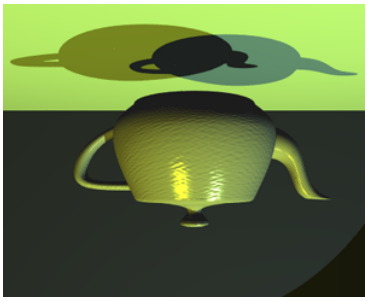
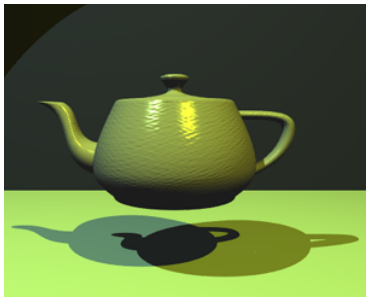


image initiale


miroir vertical


miroir horizontal

1. Écrire une fonction `miroirVertical(img1:image, img2:image)` qui place dans `img2` l'image correspondant au miroir vertical de `img1` (`img1` et `img2` sont supposées de même taille).

2. Écrire une fonction `miroirHorizontal(img1:image, img2:image)` qui place dans `img2` l'image correspondant au miroir horizontal de `img1`.

Exercice 6.2.8 Écrire une fonction `rotation90(img1:image,img2:image)` qui place dans `img2` l'image `img1` tournée de 90 degrés vers la droite, c'est-à-dire dans le sens horaire.

Pourquoi est-on obligé d'utiliser une deuxième image, plutôt que faire la rotation « en place », c'est-à-dire en n'utilisant qu'une image ?

Exercice 6.2.9

1. Écrire une fonction `rotationCoincoin(img1:image,img2:image,angle:float)` qui stocke dans `img2` l'image `img1` tournée de `angle` radians vers la droite par rapport au coin supérieur gauche de l'image. Quelques conseils :

   - Parcourir les pixels de l'image `img2` et calculer la position du pixel source correspondant dans l'image `img1` ; s'il est en-dehors de l'image, stocker un pixel noir.

   - Dans un premier temps, on pourra calculer la distance du pixel au coin supérieur gauche et l'angle par rapport à l'horizontale puis calculer les coordonnées du pixel ayant la même distance au coin supérieur gauche mais l'angle diminué.
   - Dans un deuxième temps, on pourra utiliser la transformation mathématique :

$$\begin{aligned} x' &= x \cdot \cos(\varphi) + y \cdot \sin(\varphi) \\ y' &= -x \cdot \sin(\varphi) + y \cdot \cos(\varphi) \end{aligned}$$

2. Écrire une fonction `rotationCentre(img,angle)` qui retourne une image correspondant à l'image `img1` tournée de `angle` radians par rapport au centre de l'image.

Complément : retourner une nouvelle image

Jusqu'ici les fonctions que vous avez écrites ont systématiquement modifié une image leur étant passée en paramètre, par exemple l'image `img` passée à la fonction `filtre(img)`. Vous avez également placé le résultat d'une fonction dans une seconde image également passée en paramètre, par exemple dans l'image `img2` passée à la fonction `filtre2(img1,img2)`.

Une alternative est de créer l'image de résultat directement à l'intérieur de la fonction (avec la fonction `nouvelleImage`) et de la retourner après y avoir placé le résultat. Par exemple :

```
def filtre3(img:image) -> image:
  imgRes = nouvelleImage(largeurImage(img),hauteurImage(img))
  ... # calcul des pixels de imgRes a partir de img
  return imgRes
```

L'avantage d'une telle approche est de pouvoir facilement appeler une fonction sur le résultat de la précédente, par exemple `filtre3(filtre3(img))`.

Image floue

Exercice 6.2.10 Modifiez la fonction `flou(img:image)->image` de l'exercice 6.2.3 pour qu'elle retourne une version floutée de l'image `img`. Créez des images de plus en plus floues en appelant de façon répétée cette fonction.

Ajoutez à la fonction des paramètres `x1,x2,y1,y2` qui désignent les coordonnées d'une portion rectangulaire de l'image qui doit être floutée plutôt que toute l'image. Floutez ainsi seulement un visage apparaissant sur une photo de votre choix (on pourra par exemple utiliser le logiciel *gimp* pour déterminer les coordonnées à utiliser).

Agrandissement d'une image

Exercice 6.2.11 Écrire une fonction `agrandirFacteur2(img:image)->image` qui retourne une image correspondant à l'agrandissement d'un facteur deux de l'image `img`. Pour réaliser cet agrandissement, chaque pixel de l'image source donnera un carré 2×2 pixels dans l'image destination.

Tester cette fonction sur l'image `teapot.png`. Appeler plusieurs fois la fonction pour produire un agrandissement d'un facteur 8 de cette même image. L'image apparait quelque peu *pixelisée*, comment atténuer facilement ce phénomène ?

Une image peut en cacher une autre – stéganographie

On peut utiliser le fait que l'œil ne distingue pas les faibles différences de couleur pour dissimuler une image dans une autre. Prenons deux images de mêmes dimensions. Pour chacune des coordonnées (x, y), on va mélanger une partie du pixel (x, y) de l'image A avec une partie du pixel (x, y) de l'image B en faisant en sorte que l'image obtenue A' apparaisse presque comme l'image A.

Voyons comment traiter une seule composante de couleur, disons le rouge. Pour simplifier plus encore supposons que l'intensité de la composante rouge soit codée par un entier compris entre 0 et 99 999. Imaginons que l'on veuille mélanger un pixel de l'image A dont la composante rouge vaut 17 234 avec un pixel de l'image B dont la composante rouge vaut 45 678. La technique à mettre en œuvre consiste à recombiner les trois premiers chiffres de la composante issue de A avec les deux premiers chiffres de celle issue de B en plaçant les chiffres issus de A en tête. Ce mélange des deux composantes produit le nombre 17 245. On remarque que la valeur de la composante du pixel de l'image A' est proche de celle de A car on a gardé ses trois chiffres les plus significatifs. Sur notre exemple, la différence est seulement 11 : la valeur obtenue est 17 245 contre 17 234 à l'origine. Au pire, cette technique de camouflage transforme les deux derniers chiffres de 00 à 99, ou vice-versa, ce qui fait une différence de 99, au plus.

Maintenant pour décoder une image cachée dans une autre, on peut retrouver une couleur proche de chaque pixel de la seconde image grâce aux 2 derniers chiffres de chaque composante. La composante de valeur 17 245 indique que la valeur de la composante cachée se situe entre 45 000 et 45 999 soit une erreur au maximum de 999. En ajoutant 500 à cette valeur, et donc en estimant la valeur de la composante cachée à 45 500, on réduit l'erreur maximale à 500.

Exercice 6.2.12 Écrire une fonction `decoder_composante(n:int)->int` qui, à partir d'une valeur `n` comprise entre 0 et 99 999 retourne la valeur de la composante cachée.

Appliquons maintenant cette technique à une composante d'un pixel dont la valeur est comprise en 0 et 255. Pour ce faire on exploite l'écriture en base 2 d'une composante : une telle valeur est codée sur un octet en binaire, soit 8 chiffres binaires. Pour camoufler une image on conserve les 5 premiers chiffres de la composante issue de A et les trois premiers de celle issue de B. Ainsi à partir de deux octets $a_7a_6a_5a_4a_3a_2a_1a_0$ et $b_7b_6b_5b_4b_3b_2b_1b_0$ on obtient le nombre binaire $a_7a_6a_5a_4a_3b_7b_6b_5$. Réciproquement pour découvrir la valeur cachée dans une composante, il s'agit d'isoler les 3 derniers chiffres binaires de celle-ci puis de les *décaler* de 5 chiffres. À partir du nombre binaire $a_7a_6a_5a_4a_3b_7b_6b_5$, on isole $b_7b_6b_5$ pour obtenir $b_7b_6b_500000$. Maintenant on sait que la composante originale doit avoir une valeur comprise entre $b_7b_6b_500000$ et $b_7b_6b_511111$: l'erreur maximale possible vaut donc 11111 (soit 31 en base 10). Aussi, en ajoutant 10000 (soit 16) à la composante décodée on réduit l'erreur maximale à 16.

Exercice 6.2.13 L'objectif est d'écrire une fonction permettant de découvrir une image cachée à partir d'une image source.

1. Écrire une fonction `decoder_composante(n:int)->int` qui, à partir d'une valeur `n` comprise entre 0 et 255 retourne la valeur de la composante cachée.
2. Écrire une fonction `devoiler_image(img:image)->image` qui retourne l'image cachée.
3. Utiliser cette fonction pour décoder l'image cachée dans `stegano.png` disponible sur le site du cours.

Exercice 6.2.14 Écrire une fonction `dissimuler_image(image1:image,image2:image)->image` qui retourne une nouvelle image où l'`image1` cache l'`image2`. On supposera que les images sont de même taille (on peut facilement utiliser un outil externe pour redimensionner une des deux images au besoin).

6.3 L'essentiel du chapitre

On peut récupérer la couleur d'un pixel ainsi :
`(r,g,b) = couleurPixel(img, 10, 10)`
On peut alors recalculer les valeurs de `r`, `g`, `b`, les coordonnées, etc. avant d'appeler `colorierPixel` de nouveau.

Ainsi une fonction typique qui retravaille une image est :

```
def f(img:image, n:int):
    l = largeurImage(img)
    h = hauterImage(img)
    for x in range(l):
        for y in range(h):
            (r,g,b) = couleurPixel(img, x, y)
            r = [...]
            x2 = [...]
            [...]
            colorierPixel(img, x2, y2, (r, g, b))
```

Chapitre 7. Boucle conditionnelle, représentation des nombres

7.1 Boucle conditionnelle

La boucle `while` (tant que) permet de répéter une partie de programme tant qu'une condition est vraie. Attention, cela signifie que si vous vous trompez, éventuellement votre programme va *planter*, c'est-à-dire répéter indéfiniment la boucle `while` sans jamais s'arrêter, si la condition ne devient jamais fausse. Vous pouvez alors appuyer sur Control-c pour interrompre python.

Par exemple,

```
s = 0
for x in range(10):
  s = s + x
```

peut s'écrire

```
s = 0
x = 0
while x < 10:
  s = s + x
  x = x + 1
```

Certains problèmes ne peuvent cependant pas se résoudre à l'aide d'une boucle `for` mais uniquement à l'aide d'une boucle `while`, le nombre d'itérations n'étant pas connu a priori. Considérons par exemple le problème suivant.

Exercice 7.1.1 On place 100 euros sur un compte bancaire avec 1% d'intérêts par an. Écrire une suite d'instructions calculant le nombre d'années nécessaires pour que ce placement soit au minimum doublé.

Essayez de résoudre ce problème en utilisant une boucle `for`. On constate que l'on ne sait pas à l'avance combien de tours il faut effectuer (c'est justement l'objectif du problème). Utilisez donc à la place une boucle `while`.

Exercice 7.1.2 On considère la fonction suivante :

```
def mystere(n:int):
    s = 0
    while n > 0 :
        s = s + (n % 10)
        n = n // 10
    return s
```

1. Quelle est la valeur de `mystere(2501)`. De façon générale, que calcule la fonction `mystere` ?

2. Pourquoi est-il plus pratique d'utiliser un `while` qu'un `for` pour coder la boucle de la fonction `mystere` ?

3. Écrire une fonction `nombreDeChiffres(n: int) -> int` qui retourne le nombre de chiffres contenus dans l'écriture décimale du nombre entier `n` (supposé non nul).

   Par exemple, la valeur de `nombreDeChiffres(2501)` est 4.

4. Écrire une fonction `plusGrandChiffre(n: int) -> int` qui retourne le plus grand chiffre contenu dans l'écriture décimale du nombre entier `n`.

   Par exemple, la valeur de `plusGrandChiffre(2501)` est 5.

Exercice 7.1.3 (extrait épreuve de mathématiques Bac S, 2019) Soit A un nombre réel strictement positif. On considère l'algorithme suivant :

```
i = 0
while 2**i <= A:
  i = i + 1
```

où "`**`" représente l'élévation à la puissance. On observe que la variable `i` contient la valeur 15 en fin d'exécution de cet algorithme. L'affirmation suivante est elle vraie ou fausse (justifier) :

$$2^{15} \leq A < 2^{16}$$

Exercice 7.1.4 [extrait DST 2015-16]

Dans la fonction `mystere` suivante, `a` et `n` sont deux entiers positifs supérieurs ou égaux à 0.

```
def mystere(a:int, n:int):
    i = 1
    x = a
    y = n
    while y > 0:
        if y % 2 == 1:
            i = i * x
        x = x * x
        y = y // 2
    return i
```

1. Simuler l'exécution de `mystere(a,n)` lorsque `a` est égale à `2` et `n` est égale à `8` en donnant les valeurs successives des variables `i`, `x` et `y` dans le tableau suivant :

   `mystere(2,8)`

i	
x	
y	

   Quelle est la valeur retournée par la fonction dans ce cas ?

2. Que retourne la fonction lorsque `a` et `n` prennent les valeurs suivantes ?

a	2	3
n	6	4
valeur retournée		

3. Comment interpréter la valeur retournée par la fonction `mystere` en général, pour deux valeurs `a` et `n` quelconques ?

7.2 Recherche à l'aide d'une boucle conditionnelle

Exercice 7.2.1 Pour simuler le lancer d'un dé à 6 faces marquées avec les valeurs de 1 à 6, on peut utiliser l'appel `randrange(1,7)`, après avoir ajouté à son code la ligne :

```
from random import randrange
```

En utilisant cette fonction, écrire le code des fonctions suivantes :

1. `obtenirUn6() -> int`, qui ne prend aucun paramètre, et retourne le nombre de lancés effectués avant d'obtenir le nombre 6.

2. `obtenirUnDouble() -> int`, qui retourne le nombre de lancés effectués pour obtenir deux fois consécutivement le même nombre.

7.3 Représentation des nombres

7.3.1 Entiers

Alors que les humains représentent les nombres en *décimal*, c'est-à-dire en base 10 (notamment parce que l'on a 10 doigts), les ordinateurs représentent les nombres en binaire, c'est-à-dire en base 2. Ils écrivent ainsi les nombres avec seulement les chiffres 0 ou 1, chaque chiffre est appelé bit.

Avec 1 chiffre décimal on peut écrire les nombres de 0 à 9, avec 2 chiffres décimaux on peut écrire les nombres de 0 à 99, plus généralement avec n chiffres décimaux, on peut écrire les nombres de 0 à $10^n - 1$.

De même, avec 1 bit on peut écrire les nombres de 0 à 1, avec 8 bits (un octet) on peut écrire les nombres de 0 à $2^8 - 1 = 255$, avec 32 bits on peut écrire les nombres de 0 à $2^{32} - 1 = 4\,294\,967\,295$.

python sait en fait représenter des nombres entiers de taille arbitraire, il utilise autant de bits que nécessaires pour cela [1].

Par ailleurs, pour les chaînes de caractères, python utilise un octet par lettre [2]. Pour les images, il utilise 3 octets par pixel (un pour R, un pour G, un pour B). Une image de 1000 pixels sur 1000 occupe donc a priori 3 méga-octets [3].

7.3.2 Non entiers

Lorsqu'un nombre n'est pas entier, au lycée vous avez vu la notation scientifique :

$$0,25 = 2,5 \times 10^{-1}$$
$$0,033 = 3,3 \times 10^{-2}$$
$$etc.$$

Ici, on a gardé 2 chiffres significatifs. Les ordinateurs utilisent également ce genre de notation, appelée à virgule flottante (car la position de la virgule est variable, en anglais `float`). Python utilise la précision fournie par le processeur de l'ordinateur (64 bits) ce qui permet d'obtenir environ 16 chiffres décimaux significatifs.

C'est *environ* car le processeur utilise des puissances de 2 plutôt que des puissances de 10 (car avec ses transistors il compte en base 2), et est alors obligé d'arrondir à sa propre manière.

Par exemple, voici ce qui se passe avec python lorsque l'on essaie de mettre de plus en plus de chiffres dans un nombre non entier :

1. arrondi à un multiple de 32
2. sans compter les questions de lettres accentuées, les idéogrammes chinois, etc.
3. On utilise ensuite typiquement une compression JPEG ou PNG pour réduire la taille, mais c'est toute une autre histoire !

```
>>> 0.123456789
0.123456789
>>> 0.1234567890123456
0.1234567890123456
>>> 0.12345678901234567
0.12345678901234566
>>> 0.123456789012345678
0.12345678901234568
>>> 0.1234567890123456789
0.12345678901234568
```

Le processeur est coincé, il n'a pas la place pour stocker plus d'une quinzaine de chiffres, il est obligé d'arrondir.

De toutes façons le calcul en virgule flottante ne peut pas être de précision infinie, il suffit d'essayer de calculer `1/3` :

```
>>> 1/3
0.3333333333333333
```

De nouveau le processeur arrondit.

Si l'on essaie de calculer une racine carrée :

```
>>> from math import sqrt
>>> x = sqrt(2)
>>> x
1.4142135623730951
```

$\sqrt{2}$ n'est même pas rationnel, le processeur est obligé d'arrondir. Si l'on essaie de retrouver 2 :

```
>>> x*x
2.0000000000000004
```

On ne retombe pas juste...

Un autre point surprenant est que puisque le processeur calcule en binaire, ce qui ne tombe pas sur une puissance de deux ne tombe pas "juste". Par exemple :

```
>>> 0.1+0.2
0.30000000000000004
```

Ni `0.1` ni `0.2` ne sont des puissances de deux ou des sommes de puissances de deux, et le résultat `0.3` non plus. Le processeur a dû arrondir, mais il ne pouvait pas savoir quel sens était le plus approprié, il en a choisi un dont le résultat est certes surprenant pour nous :)

Pire, si l'on compare les résultats :

```
>>> 0.1 + 0.2 == 0.3
False
```

En effet, à cause des arrondis choisis par le processeur, la partie gauche de la comparaison vaut `0.30000000000000004`, ce qui n'est effectivement pas égal à `0.3`...

Pour comparer des nombres à virgule flottante, il faut donc plutôt vérifier si la différence est considérée comme suffisamment petite :

```
>>> 0.1 + 0.2 - 0.3 < 0.00000000001
True
```

Exercice 7.3.1 TP Note : étudier la section 7.3 depuis son début jusqu'ici !

On a vu que $\frac{1}{3}$ ne peut pas être représenté exactement. Mais apparement on parvient à retomber juste quand on remultiplie par 3 [4] :

```
>>> (1/3)*3
1.0
```

Mais c'est juste de la chance ! Avec 998 cela ne passe pas :

```
>>> (1/998)*998
0.9999999999999999
```

Écrire une fonction `fraction() -> int` qui calcule le plus petit i pour lequel le calcul $(1/i) * i$ ne retombe pas juste à `1.0`.

Exercice 7.3.2 TP Écrire une fonction `chiffres() -> int` qui estime le nombre de chiffres décimaux que l'ordinateur peut effectivement stocker. Il suffit pour cela de calculer $1+(1/(10^i))$ pour i de plus en plus grand, et s'arrêter lorsque python considère en fait que c'est égal à `1.0` (à cause de l'arrondi).

Utilisez maintenant plutôt $1+(1/(2^i))$, vous obtenez 53, il se trouve que c'est effectivement le nombre de chiffres significatifs utilisés par le processeur en binaire !

7.4 Exercices de révisions et compléments

Exercice 7.4.1 (extrait DST 2014-2015) Pour tout entier $n \geq 0$, le *nombre de Cullen d'indice* n est le nombre $n \times 2^n + 1$.

Les nombres de Cullen grandissent de manière très rapide, on souhaite étudier pour quel n on atteint une certaine borne x. On va faire cela de manière assez aggressive avec une boucle `while`.

1. Écrire une fonction `cullen(n:int) -> int` prenant en paramètre un entier `n` et retournant le nombre de Cullen d'indice `n`. On rappelle que, en python, le calcul de la valeur puissance a^k s'écrit `a**k`.

2. Écrire une fonction `indiceCullenSup(x:int) -> int` prenant en paramètre un entier `x`, et retournant le plus petit entier n tel que le nombre de Cullen d'indice n soit strictement supérieur à `x`.

3. Écrire une fonction `indiceCullenInf(x:int) -> int` prenant en paramètre un entier `x`, et retournant le plus grand entier n tel que `x` soit strictement supérieur au nombre de Cullen d'indice n.

Exercice 7.4.2 Écrire une fonction `logarithme(a:int, n:int) -> int` 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*) ?

4. Ce qui parait surprenant quand nous effectuons le calcul en décimal, cf https://www.pinterest.fr/pin/619456123731421333/

Primalité

Exercice 7.4.3 L'objectif est d'écrire un *test de primalité*, c'est à dire une fonction `premier(n:int) -> bool` qui retourne `True` si l'entier naturel $n > 1$ est premier et `False` sinon. Pour cela on parcourt les nombres entre 2 et $n-1$ pour chercher un diviseur d de n : dès que l'on en trouve un on arrête les calculs, n n'est pas premier.

1. Écrire la fonction `premier(n:int) -> bool` à l'aide d'une boucle `for` implémentant cet algorithme.

2. Tester cette fonction en affichant tous les nombres premiers inférieurs à 100.

3. Modifier la fonction `premier` pour transformer la boucle `for` en boucle `while`.

4. 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 » ? (qui est très coûteuse à appeler) Modifier la fonction `premier` en conséquence.

5. Améliorer [5] `premier` 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 7.4.4 On suppose dans cet exercice que l'on dispose de la fonction `premier` décrite dans l'exercice 7.4.3, que son temps de calcul est proportionnel à $\sqrt{n}$ lorsque n est premier, et qu'il 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)`). [6]

1. Écrire une fonction `premierSuivant(n:int)->int` 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}.

Exercice 7.4.5 (extrait DST 2015-16)

1. Écrire une fonction `facteurImpair(n:int) -> int`, qui, étant donné un entier naturel `n` non-nul, renvoie le plus grand diviseur impair de `n`. Le résultat sera obtenu par une succession de divisions de `n` par 2 tant que `n` est pair. Par exemple, `facteurImpair(504)` retournera 63, puisque 504 est pair, 252 (504 divisé par 2) et 126 (252//2) sont pairs, et 63 (126//2) est impair.

2. Écrire une fonction `puissanceDiviseur(p:int,n:int) -> int`, qui, étant donné un entier naturel `n` non-nul, renvoie la plus grande puissance de `p` qui divise `n`.

   Par exemple, `puissanceDiviseur(2,504)` retournera `8`, puisque 504 est divisible par $8 = 2^3$, mais pas par $16 = 2^4$.

5. il existe des tests de primalité sophistiqués *radicalement* plus efficaces que le test naïf décrit dans l'exercice 7.4.3 ; ils permettent de tester en quelques secondes la primalité d'un nombre dont l'écriture décimale comporte plusieurs centaines de chiffres.

6. Par ailleurs, l'écart moyen entre deux premiers consécutifs est environ $\ln(n)$, il y aura donc un nombre de tels appels qui est négligeable devant $\sqrt{n}$.

3. On suppose avoir la fonction `premierSuivant(n:int) -> int` qui renvoie le premier nombre premier strictement supérieur à `n`. Par exemple, `premierSuivant(2)` vaut 3, `premierSuivant(3)` et `premierSuivant(4)` valent 5.

   En utilisant la fonction `premierSuivant(n)`, écrire une fonction `decompositionFacteursPremiers(n:int) -> list` qui renvoie la liste des nombres constituant la décomposition de `n` en un produit de facteurs premiers.

   Cette décomposition est obtenue en divisant `n` par 2 autant de fois que possible, en continuant de la même façon avec 3, puis avec 5, avec 7, et ainsi de suite jusqu'à ce que `n` ait la valeur 1.

   Par exemple, `decompositionFacteursPremier(504)` retournera la liste `[2,2,2,3,3,7]`, puisque $504 = 2 * 252$, $252 = 2 * 126$, $126 = 2 * 63$, et 63 est impair. Ensuite, $63 = 3 * 21$, $21 = 3 * 7$ et 7 n'est plus divisible par 3. Comme 7 n'est pas divisible par 5, 5 n'apparaît pas dans la liste. Enfin, $7 = 7*1$, il n'y a plus rien à décomposer, donc la liste est complète.

   Pour ajouter des valuers à la liste, vous pouvez utiliser `L = L + [x]`

   Attention, inutile de chercher à utiliser les fonctions des questions précédentes.

Suites

Exercice 7.4.6 Une suite de Syracuse est définie par récurrence de la façon suivante :

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

où a est un entier naturel strictement positif.

1. Calculer à la main les premiers termes de la suite pour $a = 5$ puis pour $a = 7$.

2. Écrire une fonction `syracuse(a:int, n:int) -> int` qui calcule le terme de rang `n` de cette suite lorsque le premier terme u_0 est égal à `a`. Tester cette fonction en donnant pour `a` la valeur 5, puis la valeur 7 et plusieurs valeurs pour `n`.

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:int) -> int` qui calcule et retourne 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 $a < 100$ (utilisez une boucle bien sûr, en utilisant `print` pour montrer les résultats !) 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 :

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

   pour calculer longueur(27). Améliorer le code de la fonction `longueur(a)` pour éviter les calculs inutiles.

7. Écrire la fonction `listeSyracuse(a)` qui calcule et retourne la *liste*

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

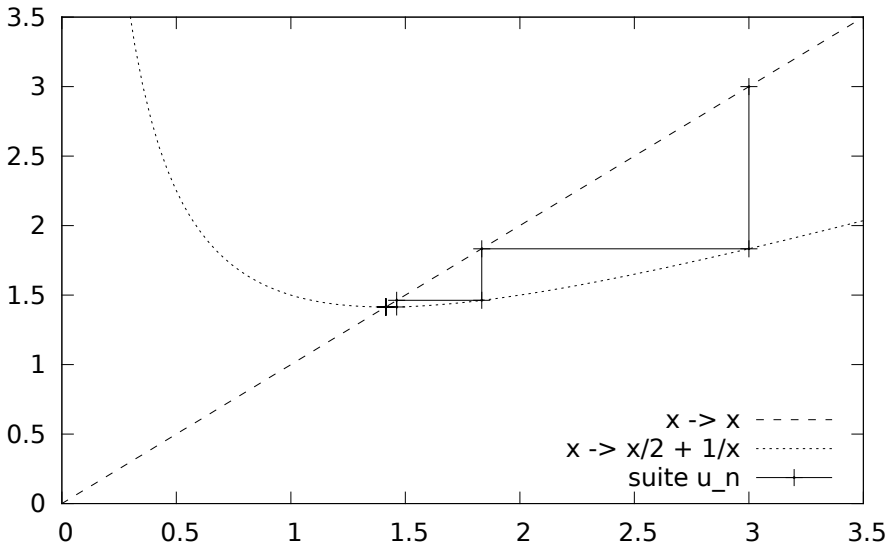
où u_n désigne le premier terme égal à 1. Par exemple, avec $a = 7$ on obtient la liste [7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1].

8. Écrire une fonction `hauteur(a:int)` qui calcule et retourne la valeur maximale de u_n lorsque le premier terme u_0 vaut a ; par exemple `hauteur(7)` vaut 52.

Exercice 7.4.7 Comment calculer $\sqrt{2}$ avec k chiffres après la virgule ? Une méthode mathématique simple est d'utiliser la suite u_n suivante :

$$\begin{cases} u_0 & = 3 \\ u_{n+1} & = \frac{u_n}{2} + \frac{1}{u_n} \end{cases}$$

On peut notamment afficher le graphe suivant pour observer la convergence.



1. Écrire une fonction `suite(n:int) -> float` qui calcule et retourne le terme de rang n de cette suite. Notes : ne vous embêtez pas à stocker toutes les valeurs ! une seule variable suffit pour garder le résultat intermédiaire.

2. Utilisez une boucle pour afficher les 10 premiers termes de la suite. Une valeur approchée de $\sqrt{2}$ est 1.414213562373095048[...]. On constate effectivement que la suite converge vers cette valeur, mais sans pouvoir toutefois l'atteindre : le nombre de chiffres des valeurs de l'ordinateur est limité !

3. Plus généralement, pour calculer $\sqrt{x}$, on peut utiliser la méthode de Newton : la suite

$$\begin{cases} u_0 & = x \\ u_{n+1} & = \frac{1}{2}(u_n + \frac{x}{u_n}) \end{cases}$$

où u_0 est une première estimation grossière de $\sqrt{x}$, on a utilisé x lui-même pour simplifier. Écrire une fonction `sqrt(x:float) -> float` qui retourne une approximation de $\sqrt{x}$ en calculant u_{10} et imprimant les valeurs intermédiaires u_n au passage. Tester avec 2, 3, 10.

4. Tester avec 1000. On constate que la convergence est lente, on n'est pas sûr que 10 itérations suffisent. Remplacer la boucle `for` par une boucle `while` pour continuer le calcul tant que la différence entre deux termes consécutifs est plus grande que 0.0000001. Tester avec 1000000.

Dessin d'une fractale

Exercice 7.4.8

$$\begin{cases} z_0 = c \\ z_{n+1} = z_n \times z_n + c \end{cases}$$

Le langage python permet de manipuler facilement les nombres complexes :

- Pour initialiser une variable complexe il suffit d'écrire : `c = complex(re,im)` où `re` désigne la partie réeelle de `c` et et `im` la partie imaginaire de `c`.
- Les opérateurs arithmétiques classiques (+, -, *, /) fonctionnent naturellement sur les complexes.
- La fonction `abs(c:complex)` permet de récupérer le module de `c`.

1. Définir une fonction `suiteComplexe(x:float, y:float, size:int, n:int) -> complex` qui retourne la valeur de n-ième terme de la suite (z_n) avec $c = (x+iy)*3/size-(2+1.5i)$.
2. Cette suite est souvent divergente, mais d'une manière irrégulière. On s'intéresse au cas où le module de z_n dépasse 2, et surtout au nombre d'itérations n nécessaire pour cela. On se limitera cependant à 256 itérations.

   En s'inspirant de la fonction `suiteComplexe`, définir avec une boucle `while` une fonction `suiteComplexeDiverge(x:float, y:float, size:int) -> int` qui retourne le plus petit entier n tel que l'on n'a plus la condition « $n < 256$ et le module de z_n est inférieur à 2 ».
3. Définir la fonction `mandelbrot(size:int)` qui retourne une image de taille $(size, size)$ telle que le niveau de gris du pixel de coordonnées (x, y) est déterminé par la valeur calculée par `suiteComplexeDiverge(x,y, size)`.

   Dessiner l'image retournée par `mandelbrot(256)`.
4. Définir la fonction `mandelbrotCouleur(size:int)` sur le même schéma que la fonction `mandelbrot(size)` mais cette fois-ci la couleur de chaque pixel est :

   $$((n*8)\,\%\,256,\ (n*32)\,\%\,256,\ (n*64)\,\%\,256)$$

   où n est la valeur retournée par la fonction `suiteComplexeDiverge(x, y, size)`.

7.5 L'essentiel du chapitre

7.5.1 Boucle conditionnelle

```
while (condition):
    instructions exécutées tant que
    condition est vraie
```

Exemple :

```
i = 1
while i < n:
    i = i * 2
```

7.5.2 Précision de calcul

En python, les nombres entiers sont représentés de manière exacte quelle que soit leur taille.

Les nombres non entiers utilisent par contre une virgule flottante, ce qui limite leur précision, en pratique à une bonne quinzaine de chiffres.

Chapitre 8. Sommets d'un graphe

Un graphe est une modélisation d'un ensemble (non vide) d'objets reliés entre eux ; les objets sont appelés sommets, et les liens entre les objets sont appelés arêtes.

On peut utiliser les graphes pour représenter diverses situations courantes : les liens d'amitié entre étudiants, les matches entre équipes de sport, les liaisons moléculaires entre des atomes, les routes entre les villes, ...

Nous vous fournissons une définition python de la *classe* des graphes, ainsi que des autres classes (sommets et arêtes) nécessaires. Ces définitions se trouvent dans le module `bibgraphes.py` écrit par des enseignants, disponible en téléchargement sur le site https://moodle.u-bordeaux.fr/course/view.php?id=14677. Allez sur ce site, retrouvez dans la section du chapitre 8 le fichier `bibgraphes.py`, utiliser un clic droit dessus et utilisez "enregistrer sous" pour l'enregistrer à côté de vos autres fichiers python. Ce module comporte aussi une vingtaine de fonctions qui permettent de manipuler graphes, sommets et arêtes sans connaître les détails des classes correspondantes. 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 bibgraphes import *
```

La figure 8.1 montre un exemple de graphe : les sommets sont quelques grandes villes françaises, et une arête entre deux villes indique que des rames TGV circulent entre elles [a] Le détail du dessin, notamment la position géographique étrange de Nantes, est sans importance, la seule chose qui compte est que ce graphe comporte les 9 sommets et les 20 arêtes de la figure ci-contre.

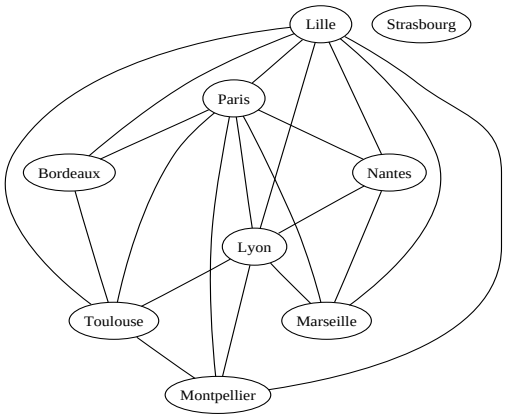


FIGURE 8.1 – Le graphe des lignes de TGV en 2005

a. Ce graphe correspond à la situation de l'année 2005 : Strasbourg est un sommet isolé, car la ligne TGV Est n'était pas encore en service à l'époque.

On peut stocker toutes les informations d'un graphe dans un fichier de format `.dot`. Pour manipuler un graphe dans un programme python, on utilisera la fonction suivante :

`ouvrirGraphe(nom:str) -> graphe`	Ouvre le fichier *nom* et retourne le graphe contenu dedans, par exemple : `tgv = ouvrirGraphe("tgv2005.dot")`
`afficherGraphe(G:graphe)`	dessine le *graphe* `G`, par exemple : `afficherGraphe(tgv)`

Pour travailler sur le graphe du TGV (fichier `tgv2005.dot` disponible sur le site du cours), on commencera donc par l'instruction suivante :

```
tgv = ouvrirGraphe("tgv2005.dot")
```

Avec Python Tutor, le graphe apparait à l'écran. Si l'on utilise Spyder, il faut utiliser l'instruction suivante :

```
afficherGraphe(tgv)
```

Si dans une session python, à la suite de cette instruction, on tape simplement `tgv`, ou `print(tgv)`, l'interprète (en jargon : *Python Shell*) répond : `<graphe: 'tgv2005'>` pour indiquer que c'est un graphe dont le nom est `'tgv2005'`. 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 il n'y a pas de variable appelée `Bordeaux` qui contiendrait ce sommet. On peut obtenir la liste complète des sommets d'un graphe G avec la fonction `listeSommets(G:graphe)`. En reprenant le même exemple, si l'on tape `listeSommets(tgv)` on obtient :

```
[<sommet: 'Lille', 'white', False>, <sommet: 'Paris', 'white', False>, ...
 <sommet: 'Bordeaux', 'white', False>, ... <sommet: 'Strasbourg', 'white', False>]
```

Chaque sommet est affiché avec confirmation de sa classe, suivie du nom du sommet et de sa couleur — pour l'instant `"white"`, donc coloré en blanc ; la dernière composante est booléenne et indique si le sommet est marqué ou non : ces marques seront utilisées section 10.3, 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 lorsque l'on affiche le sommet ou lorsque l'on dessine le graphe. Comme on l'a expliqué ci-dessus, ce n'est pas le nom d'une variable, c'est juste une étiquette appliquée au sommet. Il faut utiliser la fonction `sommetNom(G:graphe,nom:str)` pour accéder au sommet par son nom : `sommetNom (tgv, "Bordeaux")` est une expression correcte pour désigner ce sommet, et on peut ensuite le stocker dans une variable en utilisant une affectation :

```
bx = sommetNom (tgv, "Bordeaux")
```

Pour bien distinguer les deux, nous avons choisi ici un nom de variable `bx` distinct de l'étiquette *Bordeaux*.

`listeSommets(G:graphe) -> list`	retourne la *liste* des *sommets* de `G`, par exemple : `l = listeSommets(tgv)`
`nbSommets(G:graphe) -> int`	retourne le *nombre* de sommets de `G`, c'est-à-dire la *taille* de la liste précédente, par exemple : `n = nbSommets(tgv)`
`sommetNom(G:graphe, etiquette:str) -> sommet`	retourne le *sommet* de `G` désigné par son *nom* (`etiquette`), par exemple : `bdx = sommetNom(tgv, "Bordeaux")`
`nomSommet(s:sommet) -> str`	retourne le *nom* du sommet `s`, par exemple : `etiquette = nomSommet(bdx)`

Quelques graphes dont celui du TGV sont disponibles sur le site du cours https://moodle.u-bordeaux.fr/course/view.php?id=14677. Il est aussi possible de récupérer des graphes sur Internet. On pourra par exemple consulter la bibliothèque https://networkdata.ics.uci.edu/index.php qui contient d'autres exemples. Les formats supportés par le module `bibgraphes.py` sont .dot, .gml et .paj. Attention à la taille des graphes, certains contiennent de millions de sommets (*nodes*) ou d'arêtes (*edges*) et seront *très* longs à traiter !

8.1 Échauffement : coloriages

Un sommet s peut être colorié avec une couleur c par la fonction `colorierSommet(s:sommet, c:couleur)`, où c est une chaîne de caractères. La fonction `afficherGraphe(G:graphe)` 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 `"orange"`, `"chocolate"` et `"tomato"` !

`colorierSommet(s:sommet, c:str)`	colorie le *sommet* `s` avec la *couleur* `c`, par exemple : `colorierSommet(bdx, "red")`
`couleurSommet(s:sommet) -> str`	retourne la *couleur* du *sommet* `s`, par exemple : `c = couleurSommet(bdx)`
`afficherGraphe(G:graphe)`	dessine le *graphe* `G`, par exemple : `afficherGraphe(tgv)`

Dans les deux exercices suivants on suppose que la variable `tgv` contient le graphe du TGV (Figure 8.1).

Exercice 8.1.1 TP

Note : étudier le chapitre le 8 depuis le début pour avoir les explications sur le fonctionnement des graphes en python.

1. Stocker le sommet nommé *Paris* du graphe `tgv` dans une variable `p`.
2. Appeler la fonction `colorierSommet` en lui passant en paramètre la variable `p` et la couleur `"green"` (vert). Appeler la fonction `afficherGraphe` sur le graphe. Constater la coloration.
3. Colorier le sommet *Bordeaux* du graphe `tgv` en bleu, en une seule ligne, en combinant les appels de fonctions plutôt que passer par une variable. Relancer `afficherGraphe` pour constater la coloration.

Exercice 8.1.2 Puisque la fonction `listeSommets` retourne une liste des sommets du graphe, on peut l'utiliser au sein d'une boucle `for` pour effectuer une opération sur chacun des sommets du graphe.

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

a) en affichant la liste des sommets du graphe,

b) en dessinant le graphe.

Écrire l'instruction qui permet d'annuler l'opération précédente, c'est-à-dire de recolorier les sommets en blanc ; vérifier que les sommets du graphe `tgv` sont bien decoloriés.

Exercice 8.1.3

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

2. Écrire une fonction `nbSommetsColores(G:graphe)->int` qui compte les sommets colorés de G (c'est-à-dire les sommets qui ont une couleur différente de `"white"`) en *appelant* la fonction précédente et la fonction `nbSommets(G)`.

8.2 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.

Une boucle est une arête dont les deux extrémités sont confondues et correspondent au même sommet.

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, on parle alors d'arête multiple. Sur le graphe ci-contre, il y a une boucle autour du sommet B et deux boucles autour du sommet D.

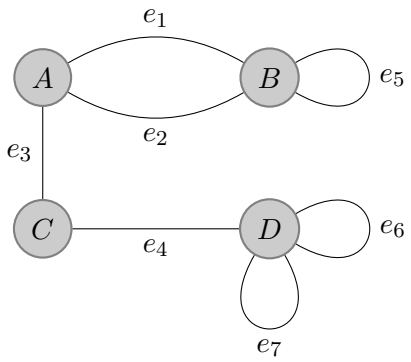


FIGURE 8.2 – un graphe avec une arête double et trois boucles

La fonction `listeVoisins(s:sommet)->list` 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 dans le graphe figure 8.2, 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 (que l'on peut considérer dans un sens ou dans l'autre).

Le degré d'un sommet s est le nombre d'incidences d'arêtes, et la fonction `degre(s:sommet)` retourne 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* : il ne faut jamais de lettre accentuée dans le nom d'une *fonction* python.

`listeVoisins(s:sommet) -> list`	retourne la *liste* des *voisins* du sommet `s`, par exemple : `L = listeVoisins(bdx)`
`degre(s:sommet) -> int`	retourne le *degré* du sommet `s`, qui est aussi la *taille* de la liste des voisins, par exemple : `n = degre(bdx)`

Exercice 8.2.1 Pour chaque sommet du graphe de la figure 8.2, noter sur papier son nom, son degré et écrire la liste de ses voisins (leur ordre dans la liste est sans importance). Écrire l'instruction python qui permet d'imprimer la même chose en TP (le fichier correspondant à ce graphe est `fig32.dot`).

Exercice 8.2.2 Appeler la fonction `listeVoisins` pour afficher la liste des villes voisines de *Nantes* dans le graphe du TGV.

Exercice 8.2.3 Écrire une fonction `colorierVoisins(s:sommet,c:str)` 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 du TGV. Vérifier le résultat de deux façons : afficher la liste des sommets du graphe, et l'afficher.

8.3 Calculs sur les 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 8.2 (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).

Exercice 8.3.1 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 ?

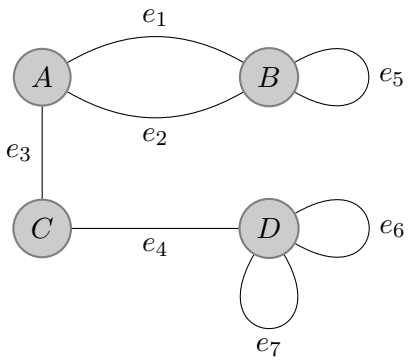


Figure 8.2 idem au graphe page 62.

Exercice 8.3.2
1. Écrire une fonction `degreMax(G:graphe)->int` qui calcule et retourne le maximum des degrés des sommets d'un graphe G.
2. Écrire une fonction `degreMin(G:graphe)->int` qui calcule et retourne le minimum des degrés des sommets d'un graphe G. Pourquoi cette question est-elle un peu plus difficile que la précédente ?

Note : On pourra utiliser `listeSommets(G)[0]` pour accéder au premier sommet du graphe.

Exercice 8.3.3 Écrire une fonction `nbSommetsDegre(G:graphe,d:int)->int` qui calcule et retourne le nombre de sommets du graphe G ayant pour degré d.

Exercice 8.3.4 TP Ouvrez le graphe *Power Grid* qui représente le réseau électrique américain, à télécharger depuis le site du cours (fichier `power.gml`). N'essayez pas de le faire afficher, il est très gros, cela prendrait beaucoup de temps !

En utilisant les fonctions de l'exercice précédents et la fonction `nbSommets` :

- Vérifier qu'il n'y a pas de sommet isolé (c'est-à-dire de degré 0).
- Compter le nombre de sommets de degré 1 et de sommets de degré 2.
- Calculer le degré maximum des sommets.
- Calculer la moyenne des degrés des sommets.

On peut ainsi avoir une idée de la robustesse du réseau électrique américain.

8.4 Exercices de révisions et compléments

Calculs sur les degrés

Exercice 8.4.1 TP Ouvrez le graphe *Les Miserables* qui représente la co-apparition des personnages du roman *Les Misérables*, à télécharger depuis le site du cours (fichier `lesmis.gml`).

Écrivez une fonction pour déterminer quel personnage interagit le plus avec d'autres personnages (la fonction `nomSommet(s:sommet)->str` retourne l'étiquette du sommet s). Il s'agira très probablement du personnage principal de l'œuvre. A-t-il pour autant croisé *Eponine* durant le récit ?

Exercice 8.4.2 TP Écrivez une fonction `moyenneDegresVoisins(s:sommet)->float` qui calcule la moyenne des degrés des voisins du sommet `s`. Écrivez une fonction `nbDegreMoindreVoisins(G:graphe)->int` qui compte le nombre de sommets pour lesquels leur degré est inférieur à la moyenne des degrés de leurs voisins. Appelez-la sur différents graphes et comparez à chaque fois au nombre de sommets. Écrivez une fonction `nbDegreMoindre(G:graphe)->int` qui compte le nombre de sommets dont le degré est inférieur à la moyenne des degrés des sommets du graphe (réutilisez la fonction de l'exercice 8.3.1). Que remarquez-vous ?

C'est un effet bien connu : sur les réseaux sociaux, on a l'impression d'avoir moins d'amis que nos amis ont d'amis, alors qu'en fait globalement non. C'est simplement parce que les personnes ayant beaucoup d'amis sont sur-représentées dans la moyenne ; c'est expliqué dans la vidéo Micmaths https://www.youtube.com/watch?v=MySkCFFgiRQ

Boucle conditionnelle

Exercice 8.4.3 TP Une marche aléatoire dans un graphe est un chemin construit aléatoirement : partant d'un sommet initial, on tire au hasard le sommet suivant parmi ses voisins et, en répétant $(k-1)$ autres fois ce processus, on obtient un chemin de longueur k.

Pour choisir au hasard un sommet parmi la liste des voisins d'un sommet `s` vous pourrez utiliser l'instruction : `elementAleatoireListe(listeVoisin(s))`

Dans cet exercice, on cherche à mesurer expérimentalement, pour un graphe et un sommet donnés, la longueur moyenne des marches aléatoires qui reviennent au sommet de départ.

1. Écrire une fonction `longueurCycleAleatoire(G:graphe,s:sommet)->int` qui retourne la longueur d'une marche aléatoire partant du sommet `s` et s'arrêtant dès que l'on revient sur ce même sommet.

2. Tester votre fonction sur un chemin `"s0"` – `"s1"` – ... – `"s9"` de longueur 10 créé à l'aide de l'appel `construireGrille(1,10)`. On pourra afficher le nom des sommets visités successivement.

3. Écrire une fonction `moyenneCycleAleatoire(G:graphe,etiquette:str,k:int)->int` qui retourne la longueur moyenne de `k` marches aléatoires formant un cycle et partant du sommet initial dont le nom est `etiquette`.

4. Tester ainsi votre programme sur un chemin de longueur 10 :

   a) `moyenneCycleAleatoire(construireGrille(1,10), "s0", 1000)`

   b) `moyenneCycleAleatoire(construireGrille(1,10), "s2", 1000)`

c) `moyenneCycleAleatoire(construireGrille(1,10), "s4", 1000)`

Commenter et expliquer les résultats obtenus.

8.5 L'essentiel du chapitre

Un graphe est une modélisation d'un ensemble (non vide) d'objets reliés entre eux ; les objets sont appelés sommets, et les liens entre les objets sont appelés arêtes.

On peut ouvrir un graphe et l'afficher :

```
tgv = ouvrirGraphe("tgv2005.dot")
afficherGraphe(tgv)
```

Il y a une différence entre un sommet et som nom, il existe deux fonctions pour passer de l'un à l'autre :

```
bdx = sommetNom(tgv,"Bordeaux")
n = nomSommet(bdx)
```

On peut colorier les sommets et récupérer leur couleur (écrite sous forme d'une chaîne de caractères) :

```
colorierSommet(bdx, "red")
c = couleurSommet(bdx)
```

On peut parcourir tous les sommets d'un graphe :

```
for s in listeSommets(tgv):
    colorierSommet(s, "red")
```

Les sommets ont un degré qui correspont à l'incidence des arêtes qui sont autour :

```
print(degre(bdx))
```

On peut parcourir les voisins d'un sommet :

```
for s in listeVoisins(bdx):
    colorierSommet(s, "blue")
```

Chapitre 9. Logique de base

Depuis le premier chapitre, nous utilisons des expressions logiques pour exprimer les conditions des blocs `if`. L'exactitude de ces expressions est cruciale pour que les programmes que l'on écrit se comportent correctement. La plupart du temps, on construit *intuitivement* ces expressions et l'on arrive à se convaincre qu'elles sont correctes. Mais dès qu'elles sont un peu compliquées, il vaut mieux utiliser un *raisonnement*, c'est-à-dire en fait écrire une *preuve* pour s'assurer que le résultat obtenu est bien celui voulu.

La section 9.1 fixe le vocabulaire et les notations principales qui vont être utilisées pour effectuer des démonstrations, elle doit être considérée comme un vade-mecum auquel on se réfère lorsque le besoin se fait sentir de préciser une notion. Vous pouvez vous reporter aux livres [1, 2] pour une approche plus exhaustive.

9.1 Notions plus ou moins connues . . .

Puisque nous allons aborder les notions de preuves et de logique, il nous faut un « terrain de jeu ». En fait nous en aurons plusieurs, les nombres, les ensembles, les graphes. Pour pouvoir pratiquer, il faut cependant connaître les règles en vigueur dans ces différents terrains. À priori, les nombres sont maitrisés depuis le primaire et nous ne nous attarderons pas à rappeler les règles connues, pour les ensembles il en va autrement, quant aux graphes, plusieurs chapitres 8, 10, 11, leurs sont dévolus, nous en verrons quelques-uns.

9.1.1 Vocabulaire

Un énoncé sera pour nous un texte, qui pourra être **vrai** ou **faux**. Il sera éventuellement placé entre guillemets « . », lorsqu'on souhaitera le mettre en exergue. Par exemple « Il fait beau », « les Martiens sont plus verts que les Vénusiens », « $1 + 1 = 0$ ». Une assertion est un énoncé tenu pour **vrai** dans un certain contexte. Ainsi l'énoncé « La somme des angles d'un triangle vaut 180 degrés » est une assertion dans la géométrie d'Euclide, mais pas dans d'autres géométries, par exemple sur Terre un triangle isocèle de 10 000 km de côté a trois angles droits ! De même l'énoncé « $1 + 1 = 0$ » est une assertion fausse dans les entiers naturels, mais vraie dans les entiers modulo 2.

9.1.2 Quantificateurs

Nous utiliserons les deux quantificateurs $\forall$ *pour tout* – on trouve aussi les expressions « pour chaque » « quelque soit », et $\exists$ *il existe* – « il y a un » « on peut trouver un ». L'ordre des quantificateurs de même nature (quantificateur universel vs quantificateur existentiel) n'a pas d'importance, l'ordre des quantificateurs de nature différente a une importance. Ci-après, p désigne un énoncé quelconque.

$\forall x, \forall y, p(x, y)$	a la même signification que	$\forall y, \forall x, p(x, y)$
$\forall x, \exists y, p(x, y)$	n'a pas le même sens que	$\exists y, \forall x, p(x, y)$

Par exemple, l'énoncé :
(1) « Pour tout entier relatif x, pour tout entier relatif y, $x \times y = y \times x$ »
$(\forall x \in \mathbb{Z}, \forall y \in \mathbb{Z}, x \times y = y \times x)$

est identique à l'énoncé :
(2) « Pour tout entier relatif y, pour tout entier relatif x, $x \times y = y \times x$ »
$(\forall y \in \mathbb{Z}, \forall x \in \mathbb{Z}, x \times y = y \times x)$.

Alors que l'énoncé :
(3) « Pour tout entier relatif x, il existe un entier relatif y tel que $y > x$ »
$(\forall x \in \mathbb{Z}, \exists y \in \mathbb{Z}, y > x)$,

n'a pas le même sens que :
(4) « Il existe un entier relatif y, tel que pour tout entier relatif x, $y > x$ »
$(\exists y \in \mathbb{Z}, \forall x \in \mathbb{Z}, y > x)$.

En effet, dans l'énoncé (3), y peut être choisi librement après le choix de x (et l'énoncé est ainsi **vrai**, par exemple avec $y = x+1$). Alors que dans l'énoncé (4), y est choisi avant de choisir x, et donc doit être le même pour tous les x (et dans $\mathbb{Z}$, un nombre plus grand que tous les autres n'existe pas, donc l'énoncé (4) est **faux**).

Un autre exemple plus subtil :
« Pour tout entier naturel x strictement positif, il existe un entier naturel y tel que $x > y$ »
$(\forall x \in \mathbb{N}^+, \exists y \in \mathbb{N}, x > y)$,

n'a pas le même sens que :
« Il existe un entier naturel y, tel que pour tout entier naturel x strictement positif, $x > y$ »
$(\exists y \in \mathbb{N}, \forall x \in \mathbb{N}^+, x > y)$.

Les deux énoncés sont **vrai**s, mais ne sont pas équivalents : dans le premier cas plusieurs y permettent de conclure, par exemple en prenant $y = x - 1$ mais ce n'est pas la seule solution ; dans le second cas, le seul entier naturel y que l'on peut utiliser pour conclure que l'énoncé est **vrai** est $y = 0$.

Exercice 9.1.1
On considère 3 entiers différents v_1, v_2, v_3

1. Comment écrire à l'aide d'une expression booléenne
$$\forall i, v_i \text{ est pair}$$
2. Comment écrire à l'aide d'une expression booléenne
$$\exists i, v_i \text{ est impair}$$
3. Pourquoi ne fait-on pas comme cela, lorsque l'on a affaire à des énoncés concernant pas seulement 3 entiers, mais $\mathbb{N}$, $\mathbb{Z}$, ou $\mathbb{R}$ tout entier ?

9.1.3 Négation

Il est parfois nécessaire de pouvoir exprimer la négation d'un énoncé. La négation sera utile dans les preuves, en particulier lorsqu'il faudra faire une preuve par l'absurde.

Dans un énoncé en langue naturelle, il suffit d'utiliser les termes exprimant une négation tels que **non, ne . . . pas**, voire de faire précéder l'énoncé de la locution **il est faux de dire que**. Nier l'énoncé « il pleut » peut se faire avec « il ne pleut pas ». Dans les énoncés mathématiques avec des symboles, on utilisera le *symbole barré* quand il existe, par exemple la négation de « $x = y$ » s'écrira « $x \neq y$ ». Ou, s'il existe, le symbole adapté, ainsi la négation de « $x < y$ » s'exprimera par « $x \geq y$ ». Lorsqu'une expression est quantifiée, on appliquera les règles suivantes :

1. la négation d'un énoncé quantifié universellement se fera en remplaçant le quantificateur universel par le quantificateur existentiel, et en prenant la **négation** de la suite de l'expression.

   Par exemple, la négation de $\forall x, f(x) = 1$ est $\exists x, f(x) \neq 1$.

2. la négation d'un énoncé quantifié existentiellement se fera

   a) Soit en barrant le quantificateur existentiel et en laissant **inchangée** la suite de l'expression.

   Par exemple, la négation de $\exists x, f(x) = 1$ est $\nexists x, f(x) = 1$.

   b) Soit, de manière équivalente, en utilisant le quantificateur universel suivi de la **négation** de l'expression.

   Par exemple, la négation de $\exists x, f(x) = 1$ est $\forall x, f(x) \neq 1$.

Exemple 9.1 (négation)
Appliquons les principes de négation sur des énoncés en langue naturelle

1. La négation de « Quelque soit l'oiseau que l'on considère, il vole », s'exprimera par « Il existe un oiseau qui ne vole pas ».

2. La négation de « Il existe un jour férié en février »

   a) En utilisant la première approche « Il n'existe pas de jour férié en février ».

   b) En utilisant le quantificateur universel donnera l'énoncé « Quelque soit le jour férié que l'on considère, il n'est pas en février »

Considérons les deux énoncés mathématiques « Tout entier naturel est strictement supérieur à 2 » ($\forall x \in \mathbb{N}, x > 2$) et « Il existe un entier naturel égal à 2 » ($\exists x \in \mathbb{N}, x = 2$)

1. La négation de « $\forall x \in \mathbb{N}, x > 2$ » donnera « $\exists x \in \mathbb{N}, x \leq 2$ »

2. La négation de « $\exists x \in \mathbb{N}, x = 2$ » donnera

   a) $\nexists x \in \mathbb{N}, x = 2$

   b) $\forall x \in \mathbb{N}, x \neq 2$

†

Exercice 9.1.2
On considère une liste d'entiers naturels et les deux propriétés suivantes

$$\forall x \in L, x \text{ est pair} \tag{9.1}$$

$$\exists x \in L, x \text{ est impair} \tag{9.2}$$

1. Parmi les 8 codes, le(s)quel(s) choisir pour résoudre la première propriété (Equation 9.1), et le(s)quel(s) choisir pour résoudre la seconde propriété (Equation 9.2) ?

```python
def solveA(L:list) -> bool:
    i = 0
    while i < len(L) and L[i]%2 == 0:
        i = i + 1
    return (i == len(L))
```

```python
def solveB(L:list) -> bool:
    i = 0
    while i < len(L) and L[i]%2 != 0:
        i = i + 1
    return (i == len(L))
```

```python
def solveC(L:list) -> bool:
    i = 0
    while i < len(L) or L[i]%2 == 0:
        i = i + 1
    return (i == len(L))
```

```python
def solveD(L:list) -> bool:
    i = 0
    while i < len(L) or L[i]%2 != 0:
        i = i + 1
    return (i == len(L))
```

```python
def solveA2(L:list) -> bool:
    i = 0
    while i < len(L) and L[i]%2 == 0:
        i = i + 1
    return (i < len(L))
```

```python
def solveB2(L:list) -> bool:
    i = 0
    while i < len(L) and L[i]%2 != 0:
        i = i + 1
    return (i < len(L))
```

```python
def solveC2(L:list) -> bool:
    i = 0
    while i < len(L) or L[i]%2 == 0:
        i = i + 1
    return (i < len(L))
```

```python
def solveD2(L:list) -> bool:
    i = 0
    while i < len(L) or L[i]%2 != 0:
        i = i + 1
    return (i < len(L))
```

2. Expliquez pourquoi, dans ces codes l'expression `(i < len(L))` est considérée comme la **négation** de `(i == len(L))` ?

3. Quelle autre expression aurait-on pu écrire en python ?

9.1.4 Si . . . Alors

Certains énoncés en langage naturel, sont formulés à l'aide de la locution **Si . . . Alors**, comme « S'il fait beau, [alors] je vais à la plage ». Que l'on peut aussi exprimer sous la forme « il fait beau *implique* je vais à la plage » ou encore « il fait beau *entraîne* je vais à la plage » ou « je vais à la plage, *dès qu'* il fait beau ». Mais pour autant, je peux aller à la plage même s'il ne fait pas beau.

L'énoncé « si A alors B » exprime que dès que A est vrai, B est forcément vrai – on dit aussi que A est une condition suffisante pour B (mais pas nécessaire : B pourrait être vrai sans que A soit vrai). Il exprime aussi que B **doit** être vrai pour que A **puisse** être vrai – on dit donc aussi que B est une condition nécessaire pour A (mais pas suffisante : B pourrait être vrai sans que A soit vrai).

Soit l'énoncé (1) « si **p** alors **q** » où **p** et **q** sont des expressions quelconques.

- La négation de l'énoncé (1) est « **p** et non **q** ».
  Quand l'énoncé (1) est vrai, sa négation est fausse. Quand l'énoncé (1) est faux, sa négation est vraie.

- La contraposée de l'énoncé (1) est « si non **q** alors non **p** ».
  L'énoncé (1) et sa contraposée sont **équivalents**.

- La réciproque de l'énoncé (1) est « si **q** alors **p** ».
  Il n'y a aucun lien entre la véracité de l'énoncé (1) et celle de sa réciproque.

Par exemple, pour l'énoncé (2) « si je suis au deuxième étage alors je ne suis pas au rez-de-chaussé »,

- Sa négation est : « je suis au deuxième étage et je suis au rez-de-chaussé ». C'est effectivement en contradiction avec l'énoncé (2).

- Sa réciproque est « si je ne suis pas au rez-de-chaussé alors je suis au deuxième étage ». Ce n'est effectivement pas du tout lié à l'énoncé (2). On pourrait être au premier étage par exemple.
- Sa contraposée est « si je suis au rez-de-chaussé alors je ne suis pas au deuxième étage ». C'est effectivement équivalent à l'énoncé (2).

Exercice 9.1.3
Pour chaque énoncé, donnez sa négation, sa réciproque et sa contraposée

1. Si T est un triangle rectangle, alors le carré de la longueur de l'hypothénuse est égal à la somme des carrés des longueurs des deux autres côtés.
2. S'il pleut, alors je prends un parapluie.
3. Si tout nombre pair supérieur à 3 se décompose comme une somme de 2 nombres premiers, alors tout nombre impair supérieur à 6 se décompose comme une somme de 3 nombres premiers.
4. Si tout nombre impair assez grand se décompose comme somme de 3 nombres premiers, alors tout nombre pair assez grand se décompose comme somme de 4 nombres premiers.

Exercice 9.1.4 (Tâche de Wason)
On a disposé devant vous 4 cartes, sur la face exposée vous voyez respectivement

- un « A »,
- un « C »,
- un « 10 »,
- et un « 11 ».

L'expérimentateur vous explique qu'en fait un symbole est inscrit sur chacune des faces des cartes (une lettre d'un côté et un nombre de l'autre), et que la règle des écritures est « Si sur une face il y a une voyelle, alors sur l'autre face il y a un nombre pair ».
Votre tâche est de vérifier si la règle est bien valide pour toutes les cartes, en retournant certaines des cartes présentées.
Quelle(s) carte(s) retournez-vous ?

Exercice 9.1.5 (Police)
Vous avez pour obligation de contrôler un débit de boissons et de dresser un procès-verbal en cas d'infraction à la loi qui stipule qu'« il est interdit de servir de l'alcool à un mineur ».
4 tables sont occupées par des personnes (vous masquant leurs consommations) et des verres (dont les consommateurs vont revenir) :

- à la première table une personne de 30 ans,
- à une autre une personne de 16 ans,
- à la troisième table un verre de boisson alcoolisée,
- et à la quatrième un verre de sirop à l'eau.

Qui contrôlez-vous ?

Les deux exercices (9.1.4, 9.1.5) sont identiques et pourtant les psychologues ont constaté que les réponses différaient. Voir l'article wikipédia sur la tâche de sélection de Wason.

Exercice 9.1.6 (Blague de bar)

4 informaticiens rentrent dans un bar.
Le barman leur demande « Vous prenez tous une bière ? »

- Le premier informaticien répond « Je ne sais pas. ».
- Le deuxième informaticien répond « Je ne sais pas. ».
- Le troisième informaticien répond « Je ne sais pas. ».
- Le quatrième informaticien répond « Ah ben du coup, oui ! ».

Que s'est-il passé ?

9.2 Exercices de révision et compléments

Quantificateurs, Négation

Exercice 9.2.1
On se place dans $\mathbb{Z}$, pour chacun des énoncés, indiquez s'il est vrai, et donnez sa négation

1. $\forall a, \forall b, a + b = 5$
2. $\exists a, \forall b, a + b = 5$
3. $\forall a, \exists b, a + b = 5$
4. $\exists a, \exists b, a + b = 5$

Exercice 9.2.2
Pour chacun des énoncés suivants, indiquez s'il est possible d'établir sa véracité par une preuve directe (ou un contre-exemple), donnez l'énoncé contraire et indiquez si la réfutation est possible par une preuve directe (ou un contre-exemple).

1. $\forall x \in \mathbb{N}, x + 1 > x$
2. $\exists x \in \mathbb{N}, 2 \times x \leq x$
3. $\forall x \in \mathbb{N} - \{0\}, \forall y \in \mathbb{N} - \{0\}, x \times y \neq x + y$
4. $\exists x \in \mathbb{N}, \exists y \in \mathbb{N}, x \times y = x + y$
5. $\exists x \in \mathbb{N}, \forall y \in \mathbb{N}, x \times y \leq x + y$
6. $\forall x \in \mathbb{N} - \{0\}, \exists y \in \mathbb{N}, x \times y > x + y$

Exercice 9.2.3
Y-a-t-il une différence entre les énoncés suivants ?

1. $\forall x \in \mathbb{R}^*, \exists y \in \mathbb{R}^*, x \times y = 1$

2. $\forall y \in \mathbb{R}^*, \exists x \in \mathbb{R}^*, x \times y = 1$

3. $\exists x \in \mathbb{R}^*, \forall y \in \mathbb{R}^*, x \times y = 1$

4. $\exists y \in \mathbb{R}^*, \forall x \in \mathbb{R}^*, x \times y = 1$

On s'intéressera plus précisément aux énoncés 1 et 2, 3 et 4, 1 et 3, 2 et 4. $\mathbb{R}^* = \mathbb{R} \setminus \{0\}$.

Exercice 9.2.4
Pour chacun des énoncés de l'exercice 9.2.3, écrire l'énoncé contraire (sa négation).

9.3 Applications pratiques

9.3.1 $\exists, \forall$

Si l'on a une liste de nombres L et que l'on veut vérifier :

$$\exists x \in L, x \text{ est pair}$$

On peut parcourir la liste L, et noter quand on rencontre un nombre pair :

```
def existeUnPair(L:list) -> bool:
    resultat = False
    for x in L:
        if x % 2 == 0:
            resultat = True
    return resultat
```

C'est-à-dire : on part de l'a-priori que la liste ne contient pas de nombre pair, et l'on parcourt la liste. Si l'on rencontre un nombre pair, on peut corriger notre a-priori. Après avoir parcouru la liste, on peut retourner le résultat.

Si l'on veut maintenant plutôt vérifier :

$$\forall x \in L, x \text{ est pair}$$

Il faut maintenant vérifier que la propriété est bien vraie pour *tous* les éléments de la liste. Autrement dit on inverse l'a-priori : on suppose a priori que c'est bien vrai, et l'on vérifie si la liste ne contient pas de contre-exemple à ce que l'on cherche à vérifier :

```
def tousPairs(L:list) -> bool:
    resultat = True
    for x in L:
        if x % 2 != 0:
            resultat = False
    return resultat
```

On a inversé la condition dans le `if` : pour vérifier que c'est bien vrai que tous les nombres sont pairs, ce qu'on vérifie, c'est plutôt s'il existe peut-être un contre-exemple !

9.3.2 Conclure plus rapidement

Pour contredire un énoncé, il suffit donc de trouver un contre-exemple : une fois que l'on en a trouvé un on est sûr que l'énoncé est faux, il n'y a pas besoin de trouver tous les contre-exemples. En python on peut profiter de cela :

```
def tousPairs(L:list) -> bool:
    for x in L:
        if x % 2 != 0:
            return False
    return True
```

Dès que l'on a trouvé un élément qui n'est pas pair, on peut conclure immédiatement en utilisant `return False`, ce qui arrête complètement le calcul de la fonction `f` et retourne immédiatement le résultat `False`. En effet, on n'a pas besoin de continuer à laisser tourner la boucle `for`, car on connait déjà la réponse finale, puisque l'on vient de trouver un contre-exemple.

Après la boucle for, on trouve `return True`, sans aucun test devant. En effet, si l'exécution de `f` est arrivée jusque là, c'est que la boucle `for` s'est terminée normalement. Cela signifie donc que jamais `return False` n'a été exécuté (sinon on n'aurait pas terminé la boucle for). Cela signifie donc que jamais on n'a trouvé d'élément qui n'était pas pair. Cela signifie donc que tous les éléments sont pairs. Et donc effectivement *enfin* on peut conclure en retournant `True`.

Une erreur courante, lorsque l'on voit un énoncé "tester si tous les éléments de la liste sont X" est d'utiliser un `if` pour vérifier si X est bien vérifié, et effectuer `return True` dans ce cas. Cela ne peut pas fonctionner, puisque `return` interrompt la boucle `for`, et empêche donc de vérifier si X est bien vérifié pour les autres éléments de la liste. Il *faut* inverser la condition X, c'est-à-dire transformer l'énoncé en "tester s'il n'existe pas d'élément de la liste qui ne vérifie pas X".

9.3.3 Coloriages

Exercice 9.3.1 Écrire une fonction `existeCouleur(G:graphe,c:str)->bool` qui renvoie `True` s'il existe au moins un sommet de couleur c dans le graphe G et `False` sinon.

Exercice 9.3.2 Écrire une fonction `toutCouleur(G:graphe,c:str)->bool` qui renvoie `True` si tous les sommets du graphe G sont de couleur c et `False` sinon.

9.3.4 Images

Exercice 9.3.3 Écrire une fonction `contientDuBlanc(img)->bool` qui renvoie `True` si l'image contient au moins un pixel qui est blanc, et `False` sinon.

Exercice 9.3.4 Écrire une fonction `contientDuVert(img)->bool` qui renvoie `True` si l'image contient au moins un pixel qui est plutôt vert, c'est-à-dire que ses composantes (r, g, b) sont telles que $g \geq 1.2 \times r$ et $g \geq 1.2 \times b$ et $g > 0$, et `False` sinon. Vérifiez notamment que pour l'image `ocean.png` disponible sur le site du cours ce n'est pas le cas.

Exercice 9.3.5 Écrire une fonction `pasDeCouleur(img)->bool` qui renvoie `True` si l'image ne contient pas de pixel coloré, seulement des pixels ayant différents niveaux de gris entre le blanc pur et le noir pur, c'est-à-dire que pour chacun des pixels les composantes r, g, b sont égales. Vérifiez notamment que c'est vrai pour l'image `texte.png` disponible sur le site du cours, mais que c'est faux pour l'image `texte2.png`

9.4 Exercices de révisions et compléments

9.4.1 Calculs sur les degrés, sommets isolés

Exercice 9.4.1 Écrire une fonction `verifieMolecule(G:graphe)` qui vérifie que tous les sommets blancs sont de degré 1, tous les sommets noirs sont de degré 4, et tous les sommets rouges

sont de degré 2. Elle renverra soit un sommet qui ne vérifie pas ces conditions, soit `None` pour indiquer que tous les sommets les vérifient. Ouvrez les graphes représentant des molécules disponibles sur le site du cours https://moodle.u-bordeaux.fr/course/view.php?id=14677 et testez `verifieMolecule` dessus.

Quelle fonction a-t-on envie d'écrire pour simplifier l'écriture de `verifieMolecule` ?

Exercice 9.4.2 On dit qu'un sommet est isolé s'il n'a aucune arête incidente, c'est-à-dire que son degré est 0. Écrire une fonction `existeIsole(G:graphe)->bool` qui teste si un graphe G a au moins un sommet isolé.

Exercice 9.4.3 Un graphe est dit cubique si tous ses sommets sont de degré 3.

Écrire une fonction `estCubique(G:graphe)->bool` qui teste si le graphe G est cubique. Testez-la sur les graphes tgv2005, Cube, Dodecaèdre, Isocaèdre.

9.4.2 Canicule [extrait DST 2017-18]

Considérons la fonction `mystere` suivante prenant en paramètre une liste de nombres `L` et un nombre `x` :

```
def mystere(L:list, x:int) -> bool:
    cpt = 0
    for i in L :
        if i > x:
            cpt = cpt + 1
            if cpt == 3:
                return True
        else:
            cpt = 0
    return False

juilletTemperaturesMax = [19,23,25,31,34,32,35,29,26,24,26,22,
   23,26,28,31,34,37,26,24,22,23,24,23,23,21,23,26,30,26,25]

temperatureCaniculeCalvados = 30
temperatureCaniculeGironde = 35
```

1. Simulez l'exécution de l'appel
   `mystere(juilletTemperaturesMax, temperatureCaniculeCalvados)`
   en complétant le tableau suivant montrant l'évolution des variables `i` et `cpt` :

i		
cpt		

2. Que retourne l'appel
   `mystere(juilletTemperaturesMax, temperatureCaniculeCalvados)` ?
3. Que retourne l'appel
   `mystere(juilletTemperaturesMax, temperatureCaniculeGironde)` ?
4. Quelle condition (nécessaire et suffisante) doivent vérifier la liste `L` et la valeur `x` pour que l'appel `mystere(L, x)` retourne la valeur `True`.

9.4.3 Voisins

Exercice 9.4.4 (extrait DST 2015-16)

1. Écrire une fonction `estDansLaListe(x,L:list)->bool` qui renvoie `True` si l'élément `x` appartient à la liste `L`, et qui renvoie `False` sinon.

2. Écrire une fonction `estVoisinDAuMoinsUn(x:sommet,L:list)->bool` qui renvoie `True` si le sommet `x` n'est pas dans la liste de sommets `L` et est voisin d'au moins un des sommets de `L`, et qui renvoie `False` sinon. Pour écrire cette fonction, vous pourrez tirer avantage à utiliser la fonction précédente.

On définit le *voisinage commun* de deux ensembles de sommets U et V d'un même graphe G, comme étant l'ensemble des sommets qui ne sont ni dans U ni dans V mais qui sont à la fois voisins d'au moins un des sommets de U et d'au moins un des sommets de V.

3. Écrire une fonction `nbVoisinageCommun(G:graphe,U:list,V:list)->int` qui renvoie le nombre de sommets qui sont dans le voisinage commun des listes de sommets `U` et `V`. Vous pourrez tirer avantage à utiliser la fonction précédente.

9.4.4 Boucles

Dans la suite, on testera les fonctions de cette section sur le graphe de la figure 8.2 (le nom du fichier correspondant est `fig32.dot`).

Exercice 9.4.5 Écrire une fonction `existeBoucle(s:sommet)->bool` qui teste si le sommet s possède une boucle incidente, c'est-à-dire une arête qui relie le sommet s à lui-même , autrement dit que le sommet s apparaît dans sa propre liste de voisins — on dit aussi dans ce cas qu'il existe « une boucle autour de s ».

Par exemple sur le graphe de la figure 8.2, la fonction doit retourner `True` pour les sommets B et D, et `False` pour les sommets A et C.

Exercice 9.4.6 Écrire une fonction `nbBoucles(s:sommet)->int` qui compte le nombre de boucles autour d'un sommet s.

Exercice 9.4.7 Écrire une fonction `sansBoucle(G:graphe)->bool` qui teste si un graphe G est sans boucle. Appliquer cette fonction sur quelques graphes disponibles sur le site du cours.

9.4.5 Voisinage

Exercice 9.4.8 Écrire une fonction `monoCouleur(G:graphe)->bool` qui teste si les arêtes du graphe G sont monocolores (et non bicolores), c'est-à-dire qui renvoie `True` si pour chaque sommet tous ses voisins sont de la même couleur que lui, et `False` sinon.

Exercice 9.4.9 Écrire une fonction `sontVoisins(s1:sommet,s2:sommet)->bool` 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 de la figure 8.2.

Écrire l'instruction qui permet de tester si en 2005 il y avait une ligne directe de TGV entre Bordeaux et Nantes.

9.5 L'essentiel du chapitre

Un énoncé peut être vrai ou faux.
Une assertion est un énoncé tenu pour vrai dans le contexte considéré.

Le quantificateur universel $\forall$ formalise l'expression « pour chaque ».
Le quantificateur existenciel $\exists$ formalise l'expression « il y a un ».

La négation inverse la valeur de vérité de l'énoncé. Notamment par exemple : la négation de $\forall x, f(x) = 1$ est $\exists x, f(x) \neq 1$; la négation de $\exists x, f(x) = 1$ est $\nexists x, f(x) = 1$, ou encore $\forall x, f(x) \neq 1$.

L'énoncé « si A alors B », exprime que B **doit** être vrai pour que A **puisse** être vrai – on dit aussi que B est une condition nécessaire pour A, et A est une condition suffisante pour B.
Soit l'énoncé (1) « si **p** alors **q** » où **p** et **q** sont des expressions quelconques.
La négation de l'énoncé (1) est « **p** et non **q** ». Elle exprime l'inverse de l'énoncé (1).
La réciproque de l'énoncé (1) est « si **q** alors **p** ».
La contraposée de l'énoncé (1) est « si non **q** alors non **p** ». Elle est équivalente à l'énoncé (1).

Lorsque l'on veut vérifier si un élément d'une liste vérifie une condition, on peut utiliser :

```
def existePair(L:list) -> bool:
    for x in L:
        if x%2 == 0:
            return True
    return False
```

Il faut bien attendre la fin de la boucle `for` avant de pouvoir conclure `False`.

Ou on peut aussi utiliser une boucle conditionnelle `while` :

```
def existePair(L:list) -> bool:
    # on cherche le premier élément pair
    i = 0
    while i < len(L) and L[i]%2 != 0:
        i = i+1
    return (i < len(L))
```

Inversement lorsque l'on veut vérifier si *tous* les éléments d'une liste vérifient une condition, on peut utiliser :

```
def tousPairs(L:list) -> bool:
    for x in L:
        if x%2 != 0:
            return False
    return True
```

Il faut bien attendre la fin de la boucle `for` avant de pouvoir conclure `True`.

Alternativement on pourra utiliser la boucle conditionnelle `while` :

```
def tousPairs(L:list) -> bool:
    # on recherche le premier contre-exemple
    i = 0
    while i < len(L) and L[i]%2 == 0:
        i = i+1
    return (i == len(L))
```

Chapitre 10. Formule des poignées de mains, graphes simples

10.1 Formalisation pour les graphes

En informatique, on est souvent amené à faire des démonstrations de propriétés. L'objectif est d'établir sa véracité (ou sa fausseté). Lorsque celle-ci est quantifiée de manière existentielle ($\exists$) il suffirait d'exhiber le cas favorable, mais ce n'est pas toujours aisé. Lorsqu'elle est quantifiée de manière universelle ($\forall$), et que l'ensemble de référence est grand, il faut mettre en œuvre une démonstration puisqu'énumérer toutes les situations serait contre-productif.

On peut distinguer deux grands types de preuves :

1. La preuve directe : ce que l'on doit prouver est une conséquence de propriétés connues. On peut progresser dans le raisonnement petit à petit, en utilisant les propriétés et définitions connues. Pour progresser on peut également utiliser un thèorème : on vérifie que ses conditions sont bien remplies, on peut alors utiliser son résultat pour continuer.

2. La preuve par l'absurde : pour établir une propriété, on fait l'**hypothèse** que la négation de la propriété à démontrer est vraie. On effectue à partir de cette hypothèse un raisonnement qui conduit à une contradiction, on dit aussi incohérence, ou absurdité. Cela conduit donc à invalider l'hypothèse qui avait été faite, et donc que sa négation est vraie c'est-à-dire que la propriété qui nous intéressait est vraie.

Pour pouvoir produire des démonstrations sur les graphes, nous posons ici un formalisme.

On note $S(G)$ l'ensemble des sommets du graphe G et $A(G)$ l'ensemble des arêtes du graphe G.

Ainsi on pourra écrire des énoncés sur les graphes, par exemple :

$$\forall s \in S(G), \text{degré}(s) \geq 1$$

Par ailleurs, le cardinal d'un ensemble E est le nombre d'éléments que contient cet ensemble. On le note couramment "$|E|$". Ainsi, $|S(G)|$ dénote le cardinal de l'ensemble des sommets du graphe G, c'est-à-dire le nombre de sommets de G, et $|A(G)|$ est le nombre d'arêtes de G.

10.2 Formule des poignées de mains

Exercice 10.2.1

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 10.2.2 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.

On peut généraliser ce raisonnement sous la forme d'une relation liant le nombre d'arêtes d'un graphe à la somme des degrés de ses sommets. Cette relation, appelée *formule générale des poignées de mains*, fait partie des résultats à mémoriser :

$$\sum_{s \in S(G)} \text{degré}(s) = 2|A(G)|$$

Exercice 10.2.3 TP

En utilisant la formule de poignées de mains, écrire une fonction `nbAretes(G:graphe)->int` qui calcule et retourne le nombre d'arêtes d'un graphe G.

Appliquer votre fonction `nbAretes` au graphe `fig32.dot` de la figure 8.2 page 62. Vérifier que votre fonction calcule correctement le nombre d'arêtes.

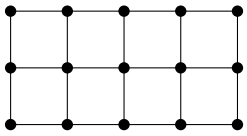
10.2.1 Exercices de révisions et compléments

Exercice 10.2.4 (extrait DS 2016-2017)

1. Un graphe peut-il avoir un seul sommet de degré impair ? Justifier.

2. Quelle propriété mathématique (simple) possède le nombre de sommets de degré impair d'un graphe ? Justifier.

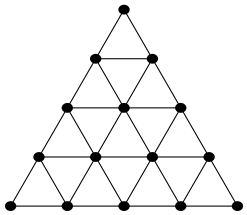
Exercice 10.2.5 Une grille (rectangulaire) $m \times n$ est constituée de m lignes horizontales et n lignes verticales, dont les croisements forment les sommets du graphe ; une grille 3×5 est représentée ci-contre.



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



Exercice 10.2.7 On dispose de 235 machines, que l'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.

10.3 Graphes simples

Dans cette section, 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 du TGV (Figure 8.1) est simple, et celui de la figure 8.2 ne l'est pas ; les graphes simples sont de loin les plus fréquents en pratique.

Exercice 10.3.1

1. Essayer de construire un graphe simple ayant 4 sommets, et tel que les degrés des sommets soient tous distincts.

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.

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

Dans un graphe simple la liste des voisins d'un sommet s ne comporte jamais de répétition, c'est même une propriété caractéristique des graphes simples.

Exercice 10.3.2 Écrire une fonction `nbOccurrences(L:list, x)->int` qui renvoie le nombre d'occurrences de l'élément `x` dans la liste `L`, c'est-à-dire le nombre de fois où il apparaît dans la liste.

Écrire une fonction `estSimple(G:graphe)->bool` qui teste si le graphe G est simple, en s'aidant de la fonction `nbOccurrences` pour vérifier qu'un sommet n'apparaît pas deux fois dans une liste de voisins.

Exercice 10.3.3 TP Expérimentez dans Python Tutor :

```
L1 = [1,2,3]
L2 = [5,6,7]
L = L1 + L2
LL = []
LL = LL + [0]
LL = LL + [1]
```

L'opérateur + sait donc "ajouter" des listes : il fabrique une nouvelle liste qui contient d'abord les éléments de la liste de gauche, puis ceux de la liste de droite. On peut ainsi par exemple construire la liste des entiers de 1 à n :

```
L=[]
for i in range(1,n+1):
  L = L + [i]
```

Exercice 10.3.4 On raffine ici l'exercice 10.3.2 pour le cas des graphes non simples. On souhaite désormais déterminer quels sommets sont adjacents à des boucles ou des arêtes multiples.

Écrire une fonction `listeSommetsVoisinsRepetes(G:graphe)->list` qui retourne la liste des sommets du graphe G pour lesquels un sommet apparaît au moins deux fois dans les listes des voisins de ces sommets.
Commencer par écrire une fonction `existeVoisinsRepetes(s:sommet) -> bool` qui teste si le sommet `s` a au moins deux fois le même sommet dans la liste de ses voisins. Utiliser cette fonction pour écrire `listeSommetsVoisinsRepetes`.

10.3.1 Exercices de révisions et compléments

Exercice 10.3.5 Un graphe *complet* est un graphe où tout sommet est relié à chacun des autres par une arête. Évaluer le nombre de comparaisons nécessaires à la fonction `estSimple(G)` lorsque `G` désigne un graphe complet à n sommets.

Une méthode plus efficace pour tester si tous les sommets d'une liste sont distincts est de marquer chaque sommet en parcourant la liste, et 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 ! Si un utilisateur applique deux fois la fonction 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:sommet)`
 	
`demarquerSommet(s:sommet)`
	marque le sommet `s`, par exemple :
	`marquerSommet(bdx)`
	démarque le sommet `s`, par exemple :
	`demarquerSommet(bdx)`

`estMarqueSommet(s:sommet) -> bool`	retourne `True` si `s` est marqué, `False` sinon, par exemple : `b = estMarqueSommet(bdx)`

Note : le marquage d'un sommet est indépendant de la coloration d'un sommet.

Exercice 10.3.6

1. Marquez le sommet `Bordeaux` du graphe du TGV. Dessinez le graphe pour observer comment c'est illustré.
2. Écrire une fonction `demarquerVoisins(s:sommet)` qui démarque tous les voisins du sommet s.
3. Écrire une fonction `voisinsDistincts(s:sommet)->bool` qui teste si tous les voisins du sommet s sont distincts en utilisant l'algorithme expliqué plus haut — le résultat est `True` ou `False`. Tester la fonction sur chaque sommet s du graphe de la figure 8.2, et afficher la liste des voisins de s après chaque test pour repérer les sommets marqués (`True` ou `False` apparait à droite) et vérifier que ce sont bien les sommets prévus.
4. Utiliser les fonctions précédentes pour écrire une nouvelle version de la fonction `estSimple(G:graphe)->bool` qui teste si un graphe G est simple.

Tester la fonction `estSimple` sur quelques-uns des graphes disponibles sur le site du cours. Après exécution du test sur un graphe G, afficher G et/ou afficher la liste de ses sommets pour repérer ceux qui sont marqués ; interpréter le résultat, en particulier pour les graphes simples.

10.4 Exercices de révision et compléments

Exercice 10.4.1 Écrire une fonction `listeVoisinsCommuns (s1:sommet,s2:sommet)->list` qui calcule et retourne la liste des voisins communs à deux sommets $s1$ et $s2$.

En utilisant un marquage, écrivez-en une version qui a une bonne complexité.

Écrire une fonction `trajet1Correspondance (s1:sommet,s2:sommet)->bool` qui renvoie `True` ou `False` selon qu'il était possible en 2005 d'aller en TGV d'une ville de sommet `s1` à une ville de sommet `s2` en effectuant au plus une correspondance.

10.5 L'essentiel du chapitre

On note $S(G)$ l'ensemble des sommets du graphe et $A(G)$ l'ensemble des arêtes du graphe. Le cardinal d'un ensemble est noté avec "|...|".

La formule générale des poignées de mains relie les degrés des sommets au nombre d'arêtes du graphe :

$$\sum_{s \in S(G)} \text{degré}(s) = 2|A(G)|$$

On peut prouver des énoncés

- soit directement, en utilisant des propriétés et définitions connues et en appliquant éventuellement des théorèmes.
- soit par l'absurde, en supposant que la négation de ce que l'on souhaite démontrer est vrai, et en montrant alors que cela nous conduit à une contradiction.

On peut "ajouter" des listes avec l'opérateur +. On peut ainsi construire des listes progressivement, par exemple :

```
L=[]
for i in range(1,n+1):
  L = L + [i]
```

Chapitre 11. Chaînes et connexité

11.1 Chaînes

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

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

où l'arête a_i est adjacente au sommet s_{i-1} qui la précède et au 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.

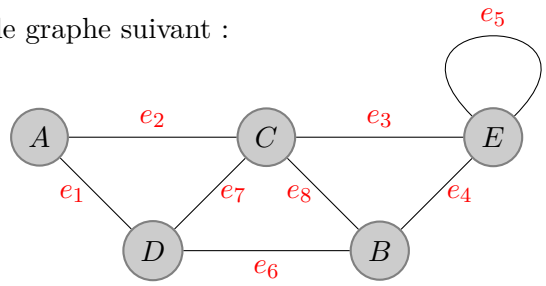
Le nombre d'arêtes n est la longueur de la chaîne (ou du cycle quand la chaîne se trouve être un cycle). Une chaîne peut être réduite à un seul sommet, elle est alors de longueur nulle. C'est d'ailleurs même un cycle.

Une chaîne est élémentaire si ses sommets et arêtes sont *distincts* deux à deux, sauf les sommets extrêmités qui peuvent être égaux (c'est dans ce cas un cycle élémentaire). Plus formellement :

$$\begin{aligned}&\Gamma = [s_0, a_1, s_1, a_2, s_2 \ldots a_n, s_n] \text{ est élémentaire si et seulement si}\\ &\quad \forall\ 0 \leq i < j \leq n, s_i \neq s_j \text{ ou } \{i, j\} = \{0, n\}\\ &\quad \text{et } \forall\ 1 \leq i < j \leq n, a_i \neq a_j\end{aligned}$$

Il ne peut donc y avoir d'allers-retours de la forme $[s, a, t, a, s]$ (où a désigne une arête qui relie les sommets s et t).

Exercice 11.1.1 Soit le graphe suivant :



1. Donner plusieurs exemples de chaînes élémentaires entre A et E.
2. Donner un exemple de chaîne non élémentaire entre A et E.
3. Donner les cycles élémentaires de longueur 4 de ce graphe.

11.2 Démonstration par récurrence

Au chapitre précédent nous avons vu la distinction entre preuve directe et preuve par l'absurde. Il y a une sous-catégorie des preuves directes qui est particulièrement puissante : les preuves par récurrence (aussi appelé induction). Elles utilisent deux étapes :

- On vérifie la propriété sur un cas de base. En général c'est plutôt trivial.

- On suppose que la propriété est vraie pour un ensemble de cas donné. On montre alors que la propriété est vraie pour un nouveau cas.

La situation d'utilisation la plus simple est une récurrence sur les entiers : on montre la propriété pour $n = 0$. On suppose que la propriété est vraie pour un n donné, et on montre qu'elle est alors vraie pour $n + 1$. On a alors une preuve directe pour tout $n \geq 0$, puisqu'il suffit de partir du cas de base 0 et d'appliquer la récurrence autant de fois que nécessaire pour parvenir au n souhaité.

Pour simplifier les démonstrations sur les chaînes, on définit deux opérateurs sur les chaînes.

Si $\Gamma_1 = [s_0, a_1, s_1, \ldots, s_n]$ et $\Gamma_2 = [t_0, b_1, t_1, \ldots, t_m]$ avec $s_n = t_0$, on pose la concaténation : $\Gamma_1 + \Gamma_2 = [s_0, a_1, s_1, \ldots, s_n, b_1, t_1, \ldots, t_m]$ dont la longueur $(n+m)$ se trouve être la somme des longueurs de Γ_1 (n) et Γ_2 (m).

Si $\Gamma = [s_0, a_1, s_1, \ldots, a_n, s_n]$, on pose l'inversion : $\overline{\Gamma} = [s_n, a_n, s_{n-1}, \ldots, a_1, s_0]$.

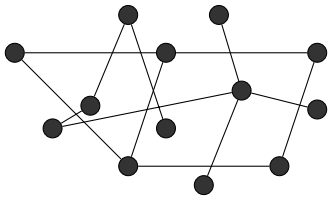
Exercice 11.2.1 Soit Γ une chaîne d'extrémités s et t. On suppose que Γ *n'est pas* élémentaire.

1. Montrer que Γ contient un cycle de longueur non nulle.
2. En déduire qu'il existe une chaîne de longueur strictement plus petite que celle de Γ, qui relie s et t.
3. En déduire par récurrence sur le nombre de cycles inclus de longueur non nulle que pour toute chaîne Γ formant un chemin de s à t, il existe une chaîne *élémentaire* formant un chemin de s à t.

11.3 Connexité

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

Exercice 11.3.1 Le graphe suivant est-il connexe ?



Le graphe du TGV (page 59) est-il connexe ?

Exercice 11.3.2 Indiquer si les propositions suivantes sont vraies ou fausses en justifiant votre réponse (rappel : un sommet isolé est un sommet de degré zéro) :

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 à plusieurs sommets soit connexe il est nécessaire que tous ses sommets soient de degré supérieur ou égal à 1.

Exercice 11.3.3

1. La proposition suivante :

   « un graphe est connexe si et seulement si 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. Question de révision : 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 ».

11.4 Exercices de révisions et compléments

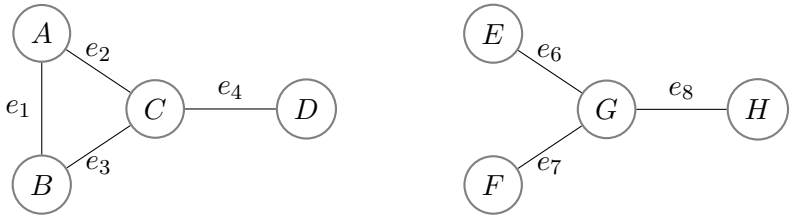
Dans les exercices suivants, une composante connexe d'un graphe $G = (S, A)$ est un sous-ensemble maximal de sommets tels que deux quelconques d'entre eux soient reliés par une chaîne. Pour tout sommet d'un graphe, la composante connexe à laquelle appartient ce sommet est donc l'ensemble des sommets auquel il est relié par au moins une chaîne.

Exercice 11.4.1 Montrer qu'un graphe est connexe si et seulement s'il ne contient qu'une composante connexe.

Exercice 11.4.2 On suppose qu'un graphe G comporte exactement deux sommets de degré impair. En utilisant la formule des poignées de main et en raisonnant par l'absurde, démontrer que ces deux sommets sont dans la même composante connexe.

Exercice 11.4.3 (extrait DST 2014-15 et 2016-17) Dans un graphe connexe, un *isthme* est une arête dont la suppression détruit la connexité du graphe. Autrement dit, une arête e est un isthme d'un graphe connexe G si et seulement si le graphe $G \setminus e$ (le graphe G privé de l'arête e) se décompose en deux composantes connexes.

1. Pour chacun des graphes suivants expliciter tous les isthmes. (Il y en a 4 sur l'ensemble des deux graphes).



2. Soit G un graphe cubique (c'est-à-dire un graphe dont tous les sommets sont de degré 3), connexe, contenant un isthme e. En appliquant la formule de poignées de mains à chacune des composantes connexes, montrer que les deux composantes connexes de $G \setminus e$ ont un nombre impair de sommets.

3. Montrer qu'un graphe connexe dont tous les sommets sont de degré pair ne possède pas d'isthme.

11.5 Algorithmes

Accessibilité

On dit que le sommet t est accessible à partir du sommet s s'il existe une chaîne reliant s et t. En théorie, il est facile de construire progressivement l'ensemble A des sommets accessibles depuis s en utilisant l'algorithme suivant :

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.

Lorsque cet algorithme se termine, l'ensemble A contient tous les sommets accessibles depuis s.

Pour savoir si le sommet t est accessible depuis s, il suffit donc de tester si le sommet t appartient à l'ensemble A, alors t est accessible depuis s, sinon il ne l'est pas.

Exercice 11.5.1 On travaille sur le graphe G de l'exercice 11.1.1. Appliquer à la main sur papier (pas de python) l'algorithme ci-dessus de construction de l'ensemble A des sommets accessibles à partir de E en choisissant toujours à l'étape 2 le sommet ayant le nom le plus petit dans l'ordre alphabétique. Recommencer en faisant varier le sommet de départ.

Expérimentation

Pour rappel, les fonctions qui permettent de manipuler le marquage d'un sommet sont les suivantes :

`marquerSommet(s:sommet)`
 	
`demarquerSommet(s:sommet)`
	marque le sommet `s`, par exemple :
	`marquerSommet(bdx)`
	démarque le sommet `s`, par exemple :
	`demarquerSommet(bdx)`

`estMarqueSommet(s:sommet) -> bool`	retourne `True` si `s` est marqué, `False` sinon, par exemple : `b = estMarqueSommet(bdx)`

Note : le marquage d'un sommet est indépendant de la coloration d'un sommet.

Exercice 11.5.2 TP Écrire les fonctions suivantes :

1. `toutDemarquer(G:graphe)` démarque tous les sommets du graphe G ;
2. `sommetAccessible(G:graphe)->sommet` retourne un sommet non marqué ayant au moins un voisin marqué, ou bien retourne `None` s'il n'existe pas de tel sommet ;

Dans l'interprète python,

- appeler `toutDemarquer` sur le graphe du TGV,
- appeler `marquerSommet` pour marquer l'un des sommets (ne pas choisir Strasbourg),
- afficher le graphe pour vérifier le résultat,
- appeler `sommetAccessible` et stocker le sommet retourné dans une variable,

- marquer ce sommet,
- afficher le graphe pour vérifier le résultat,
- appeler `sommetAccessible` de nouveau, un autre sommet est retourné,
- continuer ainsi à marquer quelques sommets. Quand cela s'arrêtera-t-il ?

Plutôt que de faire tout cela à la main, écrire une fonction `marquerAccessibles(G:graphe,s:sommet)` qui effectue les étapes suivantes :

1. elle marque d'abord le sommet `s`,
2. elle utilise ensuite une boucle `while` : tant que `sommetAccessible` retourne un sommet (et non `None`), elle le marque.

Pour que cela soit moins coûteux, arrangez votre code de façon à n'appeler `sommetAccessible` qu'une seule fois par tour de boucle `while`.

Exercice 11.5.3 TP Définir les fonctions suivantes :

1. `sommetsTousMarques(G:graphe)->bool` teste si tous les sommets du graphe G sont marqués ;
2. `estConnexe(G:graphe)->bool` teste si le graphe G est connexe. Il suffit de commencer par démarquer tout le graphe, puis piocher un sommet au hasard (utiliser `elementAleatoireListe`), appeler `marquerAccessibles`, et tester enfin si tous les sommets se retrouvent marqués.

Tester avec le graphe du TGV et celui de la figure 8.2, page 62.

Ouvrir le graphe *Power Grid* qui représente le réseau électrique américain, à télécharger via http://dept-info.labri.fr/ENSEIGNEMENT/INITINFO/ressources-initinfo/power.gml. Vérifier que ce réseau est connexe.

Exercice 11.5.4 TP En s'inspirant très largement de la fonction `marquerAccessibles` de l'exercice 11.5.2, écrire une fonction :
`estAccessibleDepuis(G:graphe, s:sommet, t:sommet) -> bool` qui retourne `True` si le sommet t est accessible depuis le sommet s dans le graphe G, et `False` sinon.

Attention : il faut commencer par démarquer tout le graphe, car les sommets que l'on marque restent marqués !

Tester avec différents sommets du graphe de l'Europe.

Il est possible d'améliorer l'algorithme afin qu'il se termine parfois plus rapidement. Pour cela, il suffit de stopper la construction de A dès qu'à l'étape 2 on a $x = t$, car cela signifie alors que t est accessible depuis s. Corriger la boucle de `estAccessibleDepuis` pour effectuer cette optimisation.

Une composante connexe d'un graphe $G = (S, A)$ est un sous-ensemble maximal de sommets tels que deux quelconques d'entre eux soient reliés par une chaîne. Pour tout sommet d'un graphe, la composante connexe à laquelle appartient ce sommet est donc l'ensemble des sommets auquel il est relié par au moins une chaîne. Un graphe connexe est donc un graphe qui possède une seule composante connexe.

Exercice 11.5.5 Montrer que l'ensemble A calculé par l'algorithme d'accessibilité correspond à l'ensemble des sommets appartenant à la composante connexe du sommet s choisi initialement.

Connexité

L'algorithme mis en œuvre dans l'exercice 11.5.3 teste si le graphe G est connexe avec le même genre d'algorithme : on choisit un sommet s, on construit l'ensemble A des sommets accessibles depuis s, puis l'on teste si tous les sommets de G appartiennent à A.

Exercice 11.5.6 Montrer que quand G est connexe le résultat de cet algorithme ne dépend pas du choix du sommet initial s.

Exercice 11.5.7 Dans le cas où l'algorithme termine sans marquer tous les sommets (le graphe G n'est pas connexe), l'ensemble A obtenu est seulement l'une des composantes connexes du graphe G.

Écrire une fonction `sommetNonMarque(G:graphe)->sommet` qui retourne un sommet non marqué.

Un sommet retourné par cette fonction appartient donc à une composante connexe dont les sommets ne sont pas encore marqués. Écrire une fonction `nbComposantesConnexes(G:graphe)->int` qui calcule le nombre de composantes connexes du graphe G.

Exercice 11.5.8 Combien le graphe stocké dans le fichier `europe.dot` possède-t-il de composantes connexes ? Comptez manuellement les composantes connexes en dessinant le graphe. Comparez le résultat obtenu avec celui renvoyé par la fonction `nbComposantesConnexes`. *Note* : la Russie est absente du graphe, d'où une composante connexe surprenante — laquelle ?

Exercice 11.5.9 Améliorer la fonction précédente afin qu'elle colore les composantes connexes de différentes couleurs. Pour ce faire, vous pourrez utiliser une palette préfédinie de couleurs (*cf.* Annexe : palettes page 95).

Arêtes (hors-programme)

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

`listeAretesIncidentes(s:sommet)`	retourne la liste des arêtes issues du sommet `s`, par exemple : `L = listeAretesIncidentes(bdx)`
`sommetVoisin(s:sommet, a:arete)`	retourne le voisin du sommet `s` en suivant l'arête `a`, par exemple : `s = sommetVoisin(bdx, a)`

`marquerArete(a:arete)`
 	
`demarquerArete(a:arete)`
	marque l'arête `a`, par exemple :
	`marquerArete(a)`
	démarque l'arête `a`, par exemple :
	`demarquerArete(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 d'accéder à 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 11.5.10 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 `estAccessibleDepuis` comme dans l'exercice 11.5.4, puis afficher le graphe de l'Europe. Les arêtes marquées apparaissent avec une épaisseur et une couleur différentes des arêtes non marquées : que remarque-t-on ?

11.6 Exercices et notions complémentaires

Parcours aléatoire

Le module de manipulation de graphes contient la fonction :

`melange(u:list) -> list`	retourne une copie de la liste `u` dont les éléments ont été permutés de façon aléatoire, par exemple : `L = melange(L)`

Par exemple l'instruction :

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

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

Exercice 11.6.1 TP 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 `estAccessibleDepuis` comme dans l'exercice 11.5.4, puis afficher le graphe de l'Europe : le chemin entre la Belgique et la Hongrie (par exemple) devrait changer à chaque exécution de l'algorithme.

Complexité de l'algorithme de connexité

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 11.6.2

1. Lorsque l'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 ? (pensez à utiliser la formule de poignée de main)

2. En déduire que la complexité *au pire* de la fonction `sommetAccessible` de l'exercice 11.5.2 est *proportionnelle* à $|A| + |S|$.

   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 `estConnexe` (exercice 11.5.3) et `estAccessibleDepuis` (exercice 11.5.4).

4. (bonus++) Il existe des méthodes plus sophistiquées pour programmer ces algorithmes, qui fournissent des fonctions de complexité (au pire) proportionnelle à $(|A| + |S|)$. Dans `sommetAccessible` l'inefficacité vient du fait que l'on reparcourt entièrement le graphe pour trouver un sommet non encore marqué avec voisin marqué. Pour être plus efficace, il suffit de se *souvenir* de ce que l'on vient de marquer, en utilisant l'algorithme suivant :
   a) Marquer `s`
   b) Mettre dans `L` la liste contenant seulement `s`
   c) Tant que `L` n'est pas vide :
      - Mettre dans `L2` la liste vide
      - Pour chaque élément `a` de la liste `L` :
        - Pour chaque voisin `b` de `a` :
          - Si `b` n'est pas marqué :
            - Marquer `b`
            - Ajouter `b` à la liste `L2`
      - Mettre la liste `L2` dans `L`

   Écrivez une nouvelle version de `sommetAccessible` utilisant cet algorithme, essayez-le sur le graphe *Power Grid*, constatez la différence de temps d'exécution par rapport à la version précédente.
5. Montrez que chaque sommet n'est traité qu'une seule fois par la première boucle "Pour".
6. Montrez que chaque arête n'est traitée que deux fois par la deuxième boucle "Pour".
7. En déduire que la complexité est $\mathcal{O}(|A| + |S|)$.

Chemin

Exercice 11.6.3 On voudrait maintenant obtenir un chemin. Il existe des algorithmes efficaces pour obtenir un chemin optimal, mais pour simplifier nous allons écrire un algorithme relativement peu efficace, et qui ne trouvera pas forcément un chemin optimal.

L'idée est d'utiliser une fonction *récursive* : pour savoir si l'on peut aller de `s` à `t`, il suffit de tester pour chaque voisin `v` de `s` si l'on peut aller de `v` à `t`, et le cas échéant d'ajouter `s` au chemin obtenu entre `v` et `t`. Puisqu'on cherche un seul chemin, on peut le retourner dès que l'on en a trouvé un.

i. Pourquoi ce n'est pas aussi simple ?

Pour éviter ce problème, il suffit de *marquer* le sommet `s` avant de parcourir ses voisins, et en tête de fonction, vérifier si le sommet est déjà marqué, auquel cas on retourne tout de suite `None`.

ii. Écrire donc la fonction `chemin(G:graphe,s:sommet,t:sommet)->list` qui retourne un chemin de `s` à `t` s'il en existe un (sous forme de la liste des sommets à parcourir), et sinon `None`. N'oubliez pas de nettoyer le graphe au début, il vous faudra donc écrire deux fonctions : l'une qui nettoie le graphe et appelle simplement l'autre, cette dernière s'appelant elle-même récursivement. N'oubliez pas non plus le cas de base où `t == s`.

iii. Déterminer un chemin entre `Espagne` et `Allemagne` dans le graphe de l'Europe.

11.7 L'essentiel du chapitre

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 est adjacente au sommet s_{i-1} qui la précède et au sommet s_i qui la suit.

Un cycle est une chaîne dont les deux extrémités coïncident ($s_0 = s_n$).

Le nombre d'arêtes n est la longueur de la chaîne.

Une chaîne est élémentaire si ses sommets et arêtes sont *distincts* deux à deux, sauf les sommets extrêmités qui peuvent être égaux (c'est dans ce cas un cycle élémentaire).

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

On dit que le sommet t est accessible à partir du sommet s s'il existe une chaîne reliant s et t.

Une preuve par récurrence se démontre en montrant une hypothèse :

- sur un cas de base.
- sur un cas en supposant qu'elle a déjà été montrée sur des cas plus petits.

Chapitre 12. Dictionnaires (hors-programme)

12.1 Introduction

Nous avons étudié les listes et les chaînes de caractères, qui permettent d'accéder à leurs éléments à l'aide d'indices :

```
>>> L = [34,56,43,4]
>>> print(L[2])
43
>>> s = "abcd"
>>> print(s[1])
"b"
```

Les indices apportent ainsi une numérotation des éléments, qui sont donc dans un certain ordre, numérotés à partir de zéro.

Les dictionnaires (`dict`) vont plus loin que cela : leurs éléments peuvent être numérotés de manière complètement arbitraire. `{}` est le dictionnaire vide, et l'on peut alors y ajouter des éléments de manière semblable à l'accès dans une liste, sauf que l'on peut utiliser n'importe quel indice :

```
>>> d = {}
>>> d[23] = 1
>>> d[42] = 12
>>> d[-12] = 123
>>> print(d[-12])
123
```

Si l'on regarde le contenu du dictionnaire :

```
>>> d
{23: 1, 42: 12, -12: 123}
```

On constate que l'on retrouve dedans à la fois les éléments et leur indice (que l'on appelera plutôt clé). On peut aussi directement écrire le dictionnaire de la même façon :

```
>>> d = {
  23: 1,
  42: 12,
  -12: 123
}
>>> print(d[-12])
123
```

Les indices n'ont en fait pas besoin d'être entiers :

```
>>> d[1.2] = 1234
```

Ni même des nombres en fait !

```
>>> d["abc"] = 12345
```

Ils sont ainsi très pratiques pour exprimer des relations.

On peut parcourir un dictionnaire avec une boucle `for`, c'est par contre la clé que l'on obtient, pour accéder à l'élément lui-même il faut utiliser le dictionnaire :

```
>>> for x in d:
      print(x, d[x])
23 1
42 12
-12 123
1.2 1234
abc 12345
```

12.2 Exercices

Exercice 12.2.1 On souhaite écrire une fonction `contientDoublon(L:list) -> bool` qui teste si la liste `L` contient deux fois la même valeur.

- Écrire une première version qui utilise deux boucles `for`. Quelle est sa complexité ?
- Voici une version qui utilise un dictionnaire :

```
def contientDoublon(L:list) -> bool:
  d = {}
  for x in L:
    d[x] = False
  for x in L:
    if d[x] == True:
      return True
    d[x] = True
```

- Remarquez comme elle ressemble à la fonction `voisinsDistincts` de l'exercice 10.3.6.
- Sachant que l'accès dans un dictionnaire contenant n éléments coûte $log(n)$, quelle est la complexité de `contientDoublon` ?

Exercice 12.2.2 On souhaite écrire une fonction `apparitions(text:str) -> dict` calculant le nombre d'apparition des différentes lettres dans un texte. L'idée est de partir du dictionnaire vide, puis pour chaque lettre du texte mettre la valeur 0 dans le dictionnaire en utilisant la lettre comme indice, puis pour chaque lettre du texte ajouter 1 à la valeur dans le dictionnaire en utilisant la lettre comme indice.

Quelle est la complexité de `apparitions` ?

12.3 L'essentiel du chapitre

Le dictionnaire vide est `{}`

On peut ajouter à un dictionnaire des éléments avec n'importe quelle valeur d'indice :

```
>>> d = {}
>>> d[23] = 1
>>> d[42] = 12
>>> d[-1] = 123
>>> d[1.2] = 1234
>>> d["abc"] = 12345
```

On peut le parcourir avec une boucle `for`, c'est par contre la clé que l'on obtient :

```
>>> for x in d:
      print(x, d[x])
23 1
42 12
-12 123
1.2 1234
abc 12345
```

Annexe A. Palettes

Du point de vue informatique, une *palette* de couleurs n'est rien d'autre qu'une *liste* de couleurs, que l'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).

Annexe B. Aide-mémoire

B.1 Environnement de TP

Deux environnements de travail sont utilisés. L'un, *Python Tutor*, est disponible depuis le site web du cours. Il est utilisable depuis n'importe quel navigateur web supportant javascript. L'intérêt de Python Tutor est qu'il permet d'observer l'exécution du programme pas à pas. Le défaut est qu'il est difficile de sauvegarder ses programmes, et il limite le nombre de pas à 10000 seulement.

L'autre, *spyder*, contient à la fois un éditeur de texte et un interpréteur interactif, ce qui permet à la fois de facilement enregistrer ses programmes, et d'expérimenter avec. Il est disponible sur les postes du CREMI, mais vous pouvez également l'installer sur votre propre ordinateur, cf les instructions sur moodle.

Selon les situations, on préfèrera donc utiliser l'un ou l'autre.

Environnement Python Tutor

L'environnement Python Tutor est disponible depuis le site web du cours :
`https://services.emi.u-bordeaux.fr/pythontutor/visualize-initinfo.html`
Il vous demande votre *login* et *mot de passe* CREMI.
Il suffit alors de taper le code et lancer l'exécution.

On obtient alors une page avec à gauche le code en cours d'exécution ainsi qu'une barre de progression et quelques boutons permettant d'avancer et reculer dans l'exécution, et à droite l'état des variables du programme. On peut ainsi observer l'évolution des variables pendant l'exécution pas à pas.

Le lien *Éditer le code* permet de retourner modifier le programme, avant de l'exécuter de nouveau.

Environnement de travail Linux

Deux systèmes d'exploitation peuvent être installés sur les ordinateurs personnels auquels vous avez accès : Linux et Windows. Les travaux pratiques seront entièrement réalisés dans le cadre du système d'exploitation Linux.

Connexion/Déconnexion

Tapez votre nom de *login*, puis votre *mot de passe* dans la fenêtre de connexion[1]. À la fin du TP, il ne faut **pas oublier** de se déconnecter (se "déloguer" du système).

1. Si le système Linux n'est pas déjà chargé, il faut redémarrer l'ordinateur et choisir l'option Linux au moment opportun.

B.2 Comprendre les messages d'erreur

Nous reprenons ici la liste des messages d'erreur que vous verrez souvent, et dans les pages suivantes des explications du problème et ses solutions potentielles. Cette liste est également disponible, de manière plus pratique, sur la page moodle du cours, "Comprendre les messages d'erreur".

- B.2.1 : NameError : name 'blablaBla' is not defined
- B.2.2 : SyntaxError : invalid syntax
- B.2.3 : SyntaxError : expected an indented block
- B.2.4 : SyntaxError : unindent does not match any outer indentation
- B.2.5 : SyntaxError : unexpected indent
- B.2.6 : SyntaxError : can't assign to operator
- B.2.7 : SyntaxError : can't assign to function call
- B.2.8 : SyntaxError : unexpected EOF while parsing
- B.2.9 : ValueError : math domain error
- B.2.10 : TypeError : 'list' object cannot be interpreted as an integer
- B.2.11 : TypeError : f() takes 2 positional arguments but 3 were given
- B.2.12 : TypeError : f() missing 1 required positional argument : 'y'
- B.2.13 : TypeError : 'int' object is not iterable
- B.2.14 : TypeError : 'int' object is not callable
- B.2.15 : TypeError : unorderable types : ___c_node() <= int()
- B.2.16 : IndexError : image index out of range
- B.2.17 : ImportError : No module named 'bibimages'
- B.2.18 : FileNotFoundError : [Errno 2] No such file or directory : 'tgv2005.dot'

B.2.1 NameError : name 'blablaBla' is not defined

Erreur : le nom 'blablabla' n'est pas défini. Soit vous utilisez la variable (ou la fonction) blablabla avant de la définir soit vous vous êtes trompé en tapant le nom de la variable (ou de la fonction) blablabla. Attention : python est sensible à l'utilisation des majuscules. Il peut aussi arriver que vous ayez oublié des guillemets autour d'une chaîne de caractères (voir exemple 3).

Exemple 1 :

```
def blablaBla(x):
    return x * x
y = blablabla(2)

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

Origine du problème : nous avons mal orthographié “blablabla” : il y un B majuscule dans le 3ième “bla” de la définition de la fonction et pas de majuscule dans l'appel de la fonction.

Exemple 2 :

```
j = 5
x = 2 * i
i = 4

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

Origine du problème : on utilise le nom ’`i`’ dans cette expression alors qu'il n'a jamais été défini avant. Soit on aurait dû initialiser la variable ’`i`’ avant (déplacer l'instruction `i = 4` avant l'instruction `x = 2 * i`), soit on a mal orthographié la variable (on voulait taper ’`j`’ au lieu de ’`i`’).

Exemple 3 : (un peu plus subtil)

```
def f(x):
    m = x*x
    return m

f(5)
y = m

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

Origine du problème : la variable “`m`” est une variable locale à la fonction “`f`”, elle ne “vit” qu'à l'intérieur de la fonction “`f`”.

Exemple 4 :

```
monImage = ouvrirImage(teapot.png)
```

Origine du problème : comme on a oublié les guillemets, python pense que teapot est le nom d'une variable... qui n'est donc pas définie.

Solution :

```
monImage = ouvrirImage("teapot.png")
```

B.2.2 SyntaxError : invalid syntax

Erreur : erreur de syntaxe. Cela veut dire que l'interpréteur est perdu car il n'arrive pas à déchiffrer votre programme. L'origine du problème n'est pas forcément à l'endroit pointé par l'interpréteur, mais peut prendre sa source plus en amont dans le code, comme nous allons le voir dans les exemples ci-dessous.

Exemple 1 :

```
def f(x)
        ^
SyntaxError: invalid syntax
```

Origine du problème : on a oublié les ' :' à la fin de la ligne.
Solution :

```
def f(x):
```

Exemple 2 :

```
if i = 6:
     ^
```

Origine du problème : après le "`if`", `python` s'attend à une expression booléenne, Ici, il se retrouve avec une affectation, et est donc perdu.
Solution : remplacer l'opérateur d'affectation "`=`" par l'opérateur de test d'égalité "`==`"

```
if i == 6:
```

Exemple 3 :

```
 if i == 9
          ^
SyntaxError: invalid syntax
```

Origine du problème : on a oublié les "`:`" à la fin de la ligne.
Solution :

```
if i==9 :
```

Exemple 4 :

```
def f(x):
    return sqrt((x*x - x)
print(f(5))
^
```

Origine du problème : il manque une parenthèse fermante à la ligne précédente.
Solution :

```
def f(x):
    return sqrt((x*x - x))
print(f(5))
```

Exemple 5 :

```
def f(x):
    return sqrt((x*x - x)

print f(5)
     ^
```

Origine du problème : en python 3 (contrairement à python 2), `print` est une fonction comme les autres. Pour appeler cette fonction, il faut donc utiliser des parenthèses.

Solution :

```
def f(x):
    return sqrt((x*x - x)

print (f(5))
```

B.2.3 SyntaxError : expected an indented block

Erreur : problème d'indentation. Le plus fréquemment, il manque une ou plusieurs tabulations à l'endroit pointé par le message d'erreur (voir exemple 1). Parfois, l'origine du problème vient de plus haut dans votre programme. Par exemple, lorsqu'on a commencé une fonction, un `if` ou un `for`, que l'on n'a pas fait suivre par une instruction (voir exemple 2) ou qu'on a commenté l'instruction en question (exemple 3).

Exemple 1 :

```
if i == 9:
toto = 9
```

Solution :

```
if i == 9 :
    toto = 9
```

Exemple 2 :

```
def f(x):

y = 5*2
```

Origine du problème : vous avez déclaré une fonction sans définir son code.

Solution : rajouter un `return` dans votre fonction.

```
def f(x):
    return

y = 5*2
```

Exemple 3 :

```
for i in range(5):
##   print(i)

y = 5*2
```

Origine du problème : vous avez commenté une partie du code, ce qui perturbe son interprétation.

Solution : commenter tout le bloc.

```
##for i in range(5):
##    print(i)

y = 5*2
```

B.2.4 SyntaxError : unindent does not match any outer indentation level

```
if i == 9:
    titi = 5
  toto = 4
  ^
SyntaxError: unindent does not match any outer indentation level
```

Origine du problème : le niveau d'indentation (nombre d'espaces) ne correspond à aucun niveau d'indentation externe. Dans l'exemple ci-dessus, l'instruction "`toto = 9`" est soit trop à gauche, soit trop à droite. Cela arrive souvent lorsqu'on mélange les caractères de tabulation et d'espace.

Solution :

```
if i == 9:
    titi = 5
    toto = 4
```

ou (selon le contexte) :

```
if i == 9:
    titi = 5
toto = 4
```

B.2.5 SyntaxError : unexpected indent

Exemple :

```
def f(x):
    y = x*x
        return y
```

Origine du problème : il y a un niveau d'indentation de trop (il y a trop de tabulations) sur cette ligne.

Solution : supprimer les tabulations superflues.

```
def f(x):
    y = x*x
    return y
```

B.2.6 SyntaxError : can't assign to operator

```
i+1 = 5
^
```

Origine du problème : on ne peut pas affecter une valeur à une expression qui ne représente pas une variable. Ici, à gauche du symbole d'affectation "`=`", on trouve l'expression "`i + 1`, qui ne définit pas une variable dans laquelle stocker la valeur qui est à droite du symbole d'affectation. Ici, on voulait peut-être écrire "`i = 5 - 1`".

B.2.7 SyntaxError : can't assign to function call

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

f(5) = y
^
SyntaxError: can't assign to function call
```

Origine du problème : on ne peut pas affecter une valeur au résultat d'un appel de fonction. En effet, le résultat d'un appel de fonction est une valeur constante (par exemple, "12"), et non une variable permettant de stocker une valeur.

Solution : vous vouliez probablement écrire :

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

y = f(5)
```

B.2.8 SyntaxError : unexpected EOF while parsing

Erreur : on arrive à la fin du fichier (EOF = End Of File) alors qu'on n'a pas terminé l'instruction ou le bloc d'instruction en cours.

Exemple :

```
def f(n):
    s = 0
    for i in range(n):

### fin du fichier ####
```

Solution : compléter la fonction :

```
def f(n):
    s = 0
    for i in range(n):
        s = s + i
    return s
### fin du fichier ####
```

B.2.9 ValueError : math domain error

Exemple :

```
y = -5
x = sqrt (y)
```

Origine du problème : la fonction `sqrt` ne prend en paramètre que des nombres positifs.

B.2.10 TypeError : 'list' object cannot be interpreted as an integer

Erreur : un objet de type "`list`" ne peut pas être interprété comme un entier.

Exemple 1 :

```
L = [5,2,3]
for i in range(L):
    ...
```

Ici on passe une liste en paramètre à la fonction range() alors qu'elle s'attend à avoir un entier.

Solution : ne pas utiliser la fonction `range`, mais directement la liste :

```
L = [5,2,3]
for elt in L:
    ...
```

B.2.11 TypeError : f() takes 2 positional arguments but 3 were given

Erreur : la fonction "`f()`" prend normalement 2 arguments et on l'appelle en lui en passant 3.

Exemple :

```
def f(x,y):
    return x + y

print(f(5,6,7))
```

Solution : appeler la fonction avec le bon nombre d'arguments

B.2.12 TypeError : f() missing 1 required positional argument : 'y'

Erreur : on appelle la fonction 'f()' en ne lui passant pas assez d'arguments.

Exemple :

```
def f(x,y):
    return x + y

print(f(5))
```

Solution : appeler la fonction avec le bon nombre d'arguments

B.2.13 TypeError : 'int' object is not iterable

Erreur : l'interpréteur s'attend à avoir une liste, mais se retrouve avec un entier.

Exemple :

```
n = 5
for i in n:
    print(i)
```

Solution :

```
n = 5
for i in range (n):
    print(i)
```

B.2.14 TypeError : 'int' object is not callable

Erreur : vous utilisez une variable (ici de type entier) comme si c'était une fonctionn c'est-à-dire en lui passant des arguments entre parenthèses.

Exemple 1 :

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

y = 2
z = y(5)
```

Solution : appeler la fonction `f` et non la variable `y` :

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

y = 2
z = f(5)
```

Exemple 2 :

```
monImage = nouvelleImage(300,200)
for y in range(largeurImage(monImage)):
    colorierPixel(monImage,0,y(255,255,255))
```

Origine du problème : ici, on a oublié une virgule.
Solution :

```
monImage = nouvelleImage(300,200)
for y in range(largeurImage(monImage)):
    colorierPixel(monImage,0,y,(255,255,255))
```

B.2.15 TypeError : unorderable types : ___c_node() <= int()

Erreur : on essaye de comparer des choses incomparables, comme par exemple un sommet avec un nombre.

paragraphExemple :

```
c = 5
for s in listeSommets(G):
    if s <= c:
```

Solution : ne pas utiliser le sommet `s` mais un entier (comme par exemple le degré de `s`) :

```
c = 5
for s in listeSommets(G):
    if degre(s) <= c:
```

B.2.16 IndexError : image index out of range

Erreur : vous essayez d'accéder à un pixel qui n'est pas dans l'image.

paragraphExemple :

```
monImage = nouvelleImage(300,200)
colorierPixel(monImage,200,200,(255,255,255))
```

Origine du problème : ici, l'image est de largeur 200 et le pixel le plus à droite a pour abscisse 199 (car la numérotation commence à 0).

B.2.17 ImportError : No module named 'bibimages'

```
from bibimages import *
```

Origine du problème : python ne trouve pas le fichier `bibimages.py`. Il y a deux causes principales à ce problème :

1. soit vous n'avez pas encore récupéré le fichier `bibimages.py` sur la page https://moodle.u-bordeaux.fr/course/view.php?id=14677 ;
2. soit vous l'avez bien récupérée, mais elle est dans un autre répertoire.

B.2.18 FileNotFoundError : [Errno 2] No such file or directory : 'tgv2005.dot'

Erreur : le fichier `'tgv2005.dot'` n'est pas présent dans le répertoire courant. Soit vous n'avez pas téléchargé ce fichier, soit vous l'avez téléchargé dans un autre répertoire.

Solution : vérifiez le contenu du répertoire courant.

B.3 Utilisation de la bibliothèque de graphes

Il faut commencer par importer le module `bibgraphes` :

```
from bibgraphes import *
```

Attention à respecter la distinction entre minuscules et majuscules :

La fonction `ouvrirGraphe` permet d'ouvrir un graphe existant au format `.dot`. Le format `.dot` est très standard et utilisé très largement pour sauvegarder des graphes, en fait c'est un simple fichier texte, vous pouvez l'ouvrir pour voir à quoi il ressemble ! On peut également écrire des graphes, notamment pour sauvegarder un coloriage.

`ouvrirGraphe(nom:str) -> graphe`	Ouvre le fichier *nom* et retourne le graphe contenu dedans, par exemple : `tgv = ouvrirGraphe("tgv2005.dot")`
`ecrireGraphe(G:graphe, nom:str)`	Sauvegarde le graphe `G` dans le fichier *nom*, par exemple : `ecrireGraphe(tgv, "fichier.dot")`
`afficherGraphe(G:graphe)`	dessine le *graphe* `G`, par exemple : `afficherGraphe(tgv)`

L'argument `G` est un graphe	
`listeSommets(G:graphe) -> list`	retourne la *liste* des *sommets* de `G`, par exemple : `l = listeSommets(tgv)`
`nbSommets(G:graphe) -> int`	retourne le *nombre* de sommets de `G`, c'est-à-dire la *taille* de la liste précédente, par exemple : `n = nbSommets(tgv)`
`sommetNom(G:graphe, etiquette:str) -> sommet`	retourne le *sommet* de `G` désigné par son *nom* (`etiquette`), par exemple : `bdx = sommetNom(tgv, "Bordeaux")`

La fonction `afficherGraphe` (aussi appelée `dessiner`) comporte des paramètres facultatifs :

`afficherGraphe(G:graphe,algo='neato')`	dessine le graphe `G` en utilisant un algorithme où les arêtes sont traitées comme des ressorts, par exemple : `afficherGraphe(tgv, algo='neato')`
`afficherGraphe(G:graphe,algo='circo')`	dessine le graphe `G` en utilisant un algorithme de placement des sommets sur un cercle, par exemple : `afficherGraphe(tgv, algo='circo')`

L'argument `s` est un sommet	
`listeVoisins(s:sommet) -> list`	retourne la *liste* des *voisins* du sommet `s`, par exemple : `L = listeVoisins(bdx)`
`degre(s:sommet) -> int`	retourne le *degré* du sommet `s`, qui est aussi la *taille* de la liste des voisins, par exemple : `n = degre(bdx)`
`nomSommet(s:sommet) -> str`	retourne le *nom* du sommet `s`, par exemple : `etiquette = nomSommet(bdx)`
`colorierSommet(s:sommet, c:str)`	colorie le *sommet* `s` avec la *couleur* `c`, par exemple : `colorierSommet(bdx, "red")` (ou `"green"`, `"blue"`, `"white"`, `"cyan"`, `"yellow"`, ...)
`couleurSommet(s:sommet) -> str`	retourne la *couleur* du *sommet* `s`, par exemple : `c = couleurSommet(bdx)`

`marquerSommet(s:sommet)`
 	
`demarquerSommet(s:sommet)`
	marque le sommet `s`, par exemple :
	`marquerSommet(bdx)`
	démarque le sommet `s`, par exemple :
	`demarquerSommet(bdx)`

`estMarqueSommet(s:sommet) -> bool`	retourne `True` si `s` est marqué, `False` sinon, par exemple : `b = estMarqueSommet(bdx)`
`listeAretesIncidentes(s)`	retourne la *liste* des arêtes *incidentes* à `s`, par exemple : `L = listeAretesIncidentes(bdx)`

L'argument `a` est une arête	
`nomArete(a:arete) -> str`	retourne le *nom* (étiquette) de `a`, par exemple : `nom = nomArete(a)`

`marquerArete(a:arete)`
 	
`demarquerArete(a:arete)`
	marque l'arête `a`, par exemple :
	`marquerArete(a)`
	démarque l'arête `a`, par exemple :
	`demarquerArete(a)`

`estMarqueeArete(a:arete) -> bool`	retourne `True` si `a` est marquée, `False` sinon, par exemple : `b = estMarqueeArete(a)`

Arguments : un sommet `s` et une arête `a`	
`sommetVoisin(s:sommet,a:arete)`	retourne le voisin du sommet `s` en suivant l'arête `a`, par exemple : `s = sommetVoisin(bdx, a)`

L'argument `u` est une liste	
`melange(u:list) -> list`	retourne une copie de la liste `u` dont les éléments ont été permutés de façon aléatoire, par exemple : `L = melange(L)`
`elementAleatoireListe(u:list)`	retourne un élément choisi aléatoirement dans la liste `u` si celle-ci est non vide. Si la liste `u` est vide la fonction retourne une erreur `IndexError`. Par exemple : `s = elementAleatoireListe(listeSommets(G))` où `G` contient un graphe

Les fonctions suivantes permettent de construire des graphes de taille variable :

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

B.4 Utilisation de la bibliothèque d'images

Il faut commencer par importer le module `bibimages.py` :

```
from bibimages import *
```

Les fonctions suivantes deviennent alors disponibles :

`ouvrirImage(nom:str) -> image`	Ouvre le fichier *nom* et retourne l'image qu'il contient, par exemple : `img = ouvrirImage("teapot.png")`
`nouvelleImage(large:int, haut:int) -> image`	Retourne une image de taille $large \times haut$, initialement noire, par exemple : `img = nouvelleImage(300, 200)`
`ecrireImage(img:image, nom:str)`	Sauvegarde l'image *img* dans le fichier *nom*, par exemple : `ecrireImage(img, "toto.png")`
`afficherImage(img:image)`	Affiche l'image *img*, par exemple : `afficherImage(img)`

`largeurImage(img:image)`
 	
`hauteurImage(img:image)`
	Retourne le nombre de colonnes de pixels contenues dans *img*, par exemple : `l = largeurImage(img)`
	Retourne le nombre de lignes de pixels contenues dans *img*, par exemple : `h = hauteurImage(img)`

`colorierPixel(img:image, x:int, y:int, (r,g,b))`	Peint le pixel (x, y) dans l'image *img* de la couleur (r, g, b), par exemple : `colorierPixel(img, 10, 10, (255, 0, 0))`
`couleurPixel(img:image, x:int, y:int)`	Retourne la couleur du pixel (x, y) dans l'image *img*, par exemple : `(r,g,b) = couleurPixel(img, 10, 10)`

B.5 Rappel de la syntaxe

Les caractères "␣␣␣␣␣" représentent l'indentation obligatoire.

Affectation : `variable = expression`

Opérateurs mathématiques : opérateurs usuels `+,-,*,/`, division entière `//`, reste de la division entière `%` .

Opérateurs booléens : comparaison `<,>,<=,>=`, égalité `==,!=`, combinaison `and,or,not`

```
if (condition):
␣␣␣␣␣instructions exécutées quand
␣␣␣␣␣condition est vraie
```

Exemple :

```
if age <= age_reduction:
␣␣␣␣prix = prix / 2
```

```
if (condition):
␣␣␣␣␣instructions exécutées quand
␣␣␣␣␣condition est vraie
else:
␣␣␣␣␣instructions exécutées quand
␣␣␣␣␣condition est fausse
```

Exemple :

```
if age <= age_reduction:
␣␣␣␣prix = 5
else:
␣␣␣␣prix = 10
```

```
if (condition1):
␣␣␣␣␣instructions exécutées quand
␣␣␣␣␣condition1 est vraie
elif (condition2):
␣␣␣␣␣instructions exécutées quand
␣␣␣␣␣condition2 est vraie
else:
␣␣␣␣␣instructions exécutées quand
␣␣␣␣␣condition1 et condition2 sont fausses
```

Exemple :

```
if age <= age_reduction1:
␣␣␣␣prix = 5
elif age >= age_reduction2:
␣␣␣␣prix = 7
else:
␣␣␣␣prix = 10
```

Définition d'une fonction :

```
def fonction(parametres, avec, virgules):
␣␣␣␣␣instructions exécutées quand
␣␣␣␣␣fonction est appelée
␣␣␣␣␣return valeur
```

Exemple :

```
def f(n,m):
␣␣␣␣if n < m:
␣␣␣␣␣␣␣␣return n
␣␣␣␣return m
```

Appel d'une fonction :

```
fonction(arguments,a,fournir)
```

Exemple :

```
f(1,3)
```

```
for variable in une_liste:
␣␣␣␣␣instructions exécutées avec variable
␣␣␣␣␣contenant successivement les valeurs
␣␣␣␣␣de une_liste
```

Exemple :

```
s = 0
for a in range(1,10):
␣␣␣␣s = s + a
```

```
while (condition):
    instructions exécutées tant que
    condition est vraie
```

Exemple :

```
i = 1
while i < n:
    i = i * 2
```

La fonction `range` permet de générer des listes d'entiers utilisables par la primitive `for` :

```
list(range(10))                    [0,1,2,3,4,5,6,7,8,9]
list(range(3,7))                              [3,4,5,6]
list(range(1,20,4))                        [1,5,9,13,17]
```

Note : Lorsque l'on a une fonction qui prend en paramètre une liste `L`, on n'utilise pas `range` puisque l'on a déjà une liste pour `for` :

```
def f(L):
  for x in L:
    ...
```

Inversement, si la fonction prend en paramètre un entier `n`, on a besoin d'utiliser `range` pour fabriquer une liste pour `for` :

```
def f(n):
  for i in range(n):
    ...
```

Bibliographie

[1] A. Arnold and I. Guessarian. *Mathématiques pour l'informatique*. Masson, Paris, 1993. ISBN 2-225-84011-3.

[2] Jacques Vélu. *Méthodes mathématiques pour l'informatique*. Sciences Sup. Dunod, Paris, 2005. ISBN 2-10-049149-0.