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))
```