ACADÉMIE DE TOULOUSE
Liberté
Égalité
Fraternité

ACADÉMIE DE VERSAILLES
Liberté
Égalité
Fraternité

ACADÉMIE DE MONTPELLIER
Liberté
Égalité
Fraternité

Corrigé des olympiades de NSI

Académies de Toulouse, Versailles et Montpellier - 8 avril 2026

Cette épreuve est individuelle et dure trois heures.

Aucun document n'est autorisé. Aucun matériel électronique n'est autorisé, en particulier **les calculatrices et les ordinateurs ne sont pas autorisés**.

Le seul langage de programmation autorisé dans cette épreuve est Python. Le code proposé doit impérativement être proprement indenté afin d'éviter toute ambiguïté quant à sa validité. De plus, pour les mêmes raisons, on veillera à ne pas écrire le code d'une fonction à cheval sur deux pages. Plus généralement, on veillera à la lisibilité, la simplicité et l'efficacité du code proposé. De plus, l'utilisation d'identifiants significatifs pour les fonctions et les variables, ainsi que l'emploi judicieux de commentaires dans le code seront appréciés.

Ce sujet comporte deux exercices totalement indépendants. Ces deux exercices doivent être **traités sur des copies séparées**.

À tout moment vous pouvez faire appel à une fonction définie plus haut dans l'exercice, même si vous n'avez pas traité la question correspondante. Si vous ne pouvez pas formuler une réponse complète à une question, il vous est néanmoins conseillé d'exposer le bilan de vos pistes de recherche.

Solution

Ce document est un corrigé de l'épreuve. Les réponses sont données après chaque question. Le code de chaque fonction est donné avec des annotations de types décrivant ses arguments et sa sortie, une description, le cas échéant des hypothèses portant sur les arguments, et enfin un jeu de tests. Ces éléments n'étaient pas attendus dans les copies des élèves.

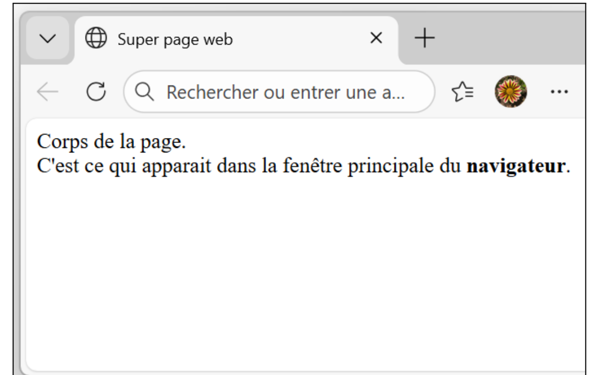
Exercice 1 : Générer et traiter du code HTML

Introduction

Les pages web sont décrites par un code source rédigé en HTML (HyperText Markup Language). Ce code source est ensuite interprété par le navigateur afin d'afficher le contenu de la page. Par exemple, le code HTML de la figure 1(a) s'affiche dans un navigateur comme présenté en figure 1(b).

```
1  <html>
2    <head>
3      <title>Super page web</title>
4    </head>
5    <body>
6      Corps de la page.<br>
7      C'est ce qui apparait dans la fenêtre
8      principale du <strong>navigateur</strong>.
9    </body>
10 </html>
```

(a) Code HTML



(b) Aperçu dans le navigateur

FIGURE 1 – Un exemple de page web simple.

Un code HTML est une chaine de caractères. En langage Python, le code sera ainsi de type string.

Utilisation des chaines de caractères

On rappelle quelques éléments de syntaxe en Python pour manipuler les chaines de caractères :

- chaine[i] vaut le caractère positionné à l'indice i dans chaine ;
- len(chaine) vaut le nombre de caractères dans chaine ;
- chaine[i:j] vaut l'extrait de chaine qui va de l'indice i inclus jusqu'à l'indice j exclu ;
- l'opérateur in permet de tester la présence d'un caractère dans une chaine de caractères ;
- l'opérateur + permet de concaténer deux chaines de caractères ;
- la fonction str permet d'obtenir la chaine correspondant à un entier.

Par exemple, si ch1 vaut "olympiades", ch1[0] vaut 'o', ch1[9] vaut 's', len(ch1) vaut 10 et ch1[4:8] vaut "piad". De plus 'a' in ch1 vaut True tandis que 'z' in ch1 vaut False.
Si on définit ch2 = "NS" + "I", alors ch1 + ch2 + "_" + str(26) vaut "olympiadesNSI_26".

On accède donc à une chaine de caractères comme si c'était une liste de caractères, mais **on ne peut pas la modifier**. Par exemple, **on ne peut pas écrire** chaine[0] = 'a'.

Un code HTML commence par la balise <html> et se termine par la balise </html>. Entre ces deux balises, on distingue les deux parties suivantes.

- L'en-tête, comprise entre les balises <head> et </head>. Cette partie contient des informations qui ne sont pas directement affichées dans la page. Elle contient notamment la balise obligatoire <title>, qui correspond au titre affiché dans l'onglet du navigateur.
- Le corps de la page, compris entre les balises <body> et </body>. Cette partie contient le contenu de la page : texte, titres, listes, etc.

Question 1

Écrire le code HTML d'une page web dont le titre est « Olympiades de NSI » et dont le contenu est la phrase « En HTML, je rends le web exceptionnel ! ».

Solution

```html
<html>
  <head>
    <title>Olympiades de NSI</title>
  </head>
  <body>
    En HTML, je rends le web exceptionnel !
  </body>
</html>
```

Partie 1 : Générer du code HTML

Dans cette partie, on souhaite écrire un programme Python générant le code HTML d'une page web.

Question 2

Recopier et compléter l'instruction `return` de la fonction `creer_debut` donnée ci-dessous.

```python
def creer_debut(titre_page):
    """
    titre_page : str
    retour : str
    Retourne le code HTML, jusqu'à la balise body ouvrante incluse,
    d'une page dont le titre est titre_page
    """
    html_o = "<html>"
    head_o = "<head>"
    contenu_head = "<title>" + titre_page + "</title>"
    head_f = "</head>"
    body_o = "<body>"
    return ...
```

Solution

```python
def creer_debut(titre_page):
    """
    titre_page : str
    retour : str
    Retourne le code HTML, jusqu'à balise body ouvrante incluse,
    d'une page dont le titre est titre_page
    """
    html_o = "<html>"
    head_o = "<head>"
    contenu_head = "<title>" + titre_page + "</title>"
    head_f = "</head>"
    body_o = "<body>"
    return html_o + head_o + contenu_head + head_f + body_o
```

On remarque qu'il n'est pas nécessaire d'écrire les indentations et les retours à la ligne dans le code HTML produit. En effet, ces derniers ne sont pas significatifs en HTML.

Comme le montre la figure 2, HTML permet de définir des titres de niveaux 1 à 6, à l'aide des balises <h1> à <h6> associées respectivement aux balises fermantes </h1> à </h6>.

```html
<html>
  <head>
    <title>Mes titres</title>
  </head>
  <body>
    <h1>Planning année 2026</h1>
    <h2>Mois</h2>
    <h3>Semaine</h3>
    <h4>Jour</h4>
    <h5>Matin ou soir</h5>
    <h6>Heure</h6>
    <h1>Planning année 2027</h1>
  </body>
</html>
```

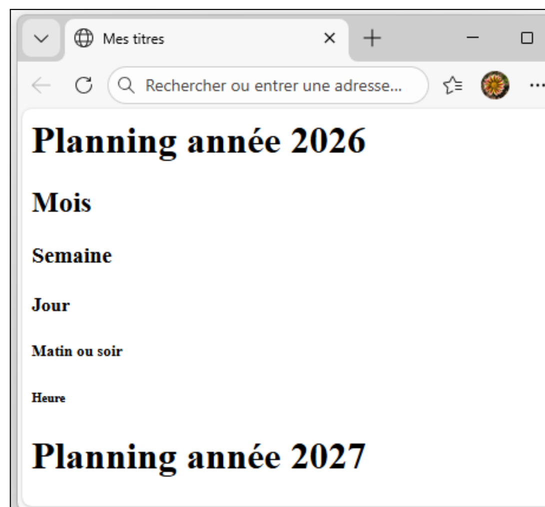


FIGURE 2 – Un exemple de titres de niveau 1 à 6.

Question 3

Écrire une fonction `creer_titre` qui prend en paramètres une chaine de caractères `intitule` ainsi qu'un entier `niveau` compris entre 1 et 6, et qui renvoie un code HTML permettant de créer un titre de niveau `niveau` contenant le texte `intitule`.
Cette fonction, comme toutes celles générant du code HTML, renvoie une chaine de caractères.

Solution

```python
def creer_titre(intitule, niveau):
    """
    intitule : str, niveau : int
    retour : str
    préconditions : 1 <= niveau and niveau <= 6
    Retourne le code HTML créant un titre de contenu intitule et de
     ↪ niveau niveau
    """
    ouverture = "<h" + str(niveau) + ">"
    fin = "</h" + str(niveau) + ">"
    return ouverture + intitule + fin
```

Comme le montre la figure 3, pour créer une énumération en HTML on utilise :

- les balises <ol> et pour délimiter la liste ;
- les balises <li> et pour délimiter chaque élément.

On crée une fonction `creer_enum` qui prend en paramètre une liste de chaines de caractères `liste_items` et qui renvoie un code HTML créant une énumération contenant les éléments de `liste_items`.

Question 4

a) Écrire l'appel de la fonction permettant d'obtenir l'énumération de la figure 3.

```html
<html>
  <head>
    <title>Mon énumération</title>
  </head>
  <body>
    <ol>
      <li>Lundi</li>
      <li>Mardi</li>
      <li>Jeudi</li>
      <li>Vendredi</li>
    </ol>
  </body>
</html>
```

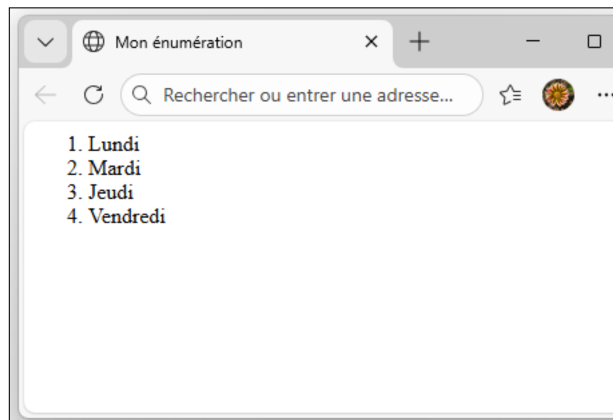


FIGURE 3 – Un exemple d'énumération.

b) Écrire le code de la fonction `creer_enum`.

Solution

a)

```python
creer_enum(["Lundi", "Mardi", "Jeudi", "Vendredi"])
```

b)

```python
def creer_enum(liste_items):
    """
    liste_items : lst de str
    retour : str
    Retourne le code HTML créant une liste ordonnée dont les items
    sont les éléments de liste_items
    """
    contenu = "<ol>"
    for element in liste_items:
        contenu += "<li>" + element + "</li>"
    contenu += "</ol>"
    return contenu
```

Question 5

Écrire le code Python permettant de générer le code HTML de la figure 4 en utilisant les fonctions définies précédemment. *Les indentations et sauts de lignes peuvent être omis dans le code généré.*

Solution

```python
res = creer_debut("Olympiades de NSI")
res = res + creer_titre("Principe",1)
res = res + creer_enum(["Prendre du plaisir à coder",
                        "Résoudre un max de questions",
                        "Découvrir des notions"])
res = res + "</body>" + "</html>"
```

```html
<html>
  <head>
    <title>Olympiades de NSI</title>
  </head>
  <body>
    <h1>Principe</h1>
      <ol>
        <li>Prendre du plaisir à coder</li>
        <li>Résoudre un max de questions</li>
        <li>Découvrir des notions</li>
      </ol>
  </body>
</html>
```

Figure 4 – Code HTML à générer.

Partie 2 : Tester si un texte est un code HTML valide

Dans cette partie, on souhaite écrire un programme Python qui analyse un code source afin de vérifier s'il a été écrit en respectant les règles d'écriture du langage HTML. Le code source à analyser est une chaine de caractères, par exemple :
`codeHTML0 = "<html><body><h1>Titre principal</h1></body></html>"`.

2.1 : Règle de placement des chevrons

En langage HTML, on utilise des chevrons (les caractères `'<'` et `'>'`) pour délimiter les portions de code qui constituent des balises. On suppose que dans le code à analyser, les chevrons ne sont utilisés que pour délimiter les balises (ils n'apparaissent pas dans le texte à afficher sur la page notamment). Chaque chevron ouvrant `'<'` doit être fermé par un chevron fermant `'>'`, et ce avant d'ouvrir un autre chevron. Ainsi les chaines `"aaa"`, `"aa<bbbb>aaa"` et `"aa<bbbb>aaa<b>"` sont bien chevronnées, mais les chaines `"aa<b>bbb>aaa"`, `"aa<b<bbb>aaa"` et `"aa<bb<c>bb>aaa"` ne le sont pas.

On suppose qu'on dispose d'une fonction `bien_chevronnee` qui prend en paramètre une chaine de caractères et qui teste si celle-ci est bien chevronnée. *Comme toutes les fonctions qui testent une condition, cette fonction renvoie* `True` *si la condition est vérifiée et* `False` *sinon.*

Solution

```python
def bien_chevronnee(codeHTML):
    """
    codeHTML : str
    retour : bool
    Renvoie True si les chevrons de codeHTML sont bien placés, False
    ↪ sinon
    """
    est_valide = True
    chevron_ouvert = False
    i = 0
    while (est_valide and i < len(codeHTML) ):
        if codeHTML[i] == "<":
            # rencontrer un chevron ouvrant est valide si on n'en a pas
            ↪ déjà un ouvert
            est_valide = not chevron_ouvert
            chevron_ouvert = True
```

```
15            if codeHTML[i] == ">":
16                # rencontrer un chevron fermant est valide si on a déjà un
                  ↪  chevron ouvert
17                est_valide = chevron_ouvert
18                chevron_ouvert = False
19            i = i+1
20        return est_valide and not chevron_ouvert
```

Question 6

Donner et justifier la valeur renvoyée par chacun des appels suivants :

- `bien_chevronnee("<body>>")` ;
- `bien_chevronnee("</body> <body>")` ;
- `bien_chevronnee("<titre <toto>>")`.

Solution

- `bien_chevronnee("<body>>")` vaut `False` car le deuxième chevron fermant ne correspond à aucun chevron ouvrant.
- `bien_chevronnee("</body> <body>")` vaut `True` car chaque chevron fermant correspond à un fermant, sans imbrication. Le fait que la balise feramante précède la balise ouvrante n'est pas pris en compte par la fonction `bien_chevronnee`.
- `bien_chevronnee("<titre <toto>>")` vaut `False` car le deuxième chevron ouvrant est ouvert avant que le premier ne soit fermé.

2.2 : Règle d'écriture des balises

- On appelle **balise** une chaine qui commence par `'<'` et se termine par `'>'`, par exemple `<html>`, `<h1>`, `</p>` ou encore `<p id="para1">`.
- On appelle alors **contenu d'une balise** la chaine de caractères qui est entre ces deux chevrons.
- Une **balise fermante** est une balise dont le contenu commence par le caractère `'/'`, par exemple `</html>`, `</p>`.
- Une **balise ouvrante** est une balise qui n'est pas une balise fermante, par exemple `<html>`, `<p>`.

Question 7

Écrire une fonction `est_balise` qui prend en paramètre une chaine de caractères `chaine` et qui teste si `chaine` est une balise. Par exemple `est_balise("<html>")` renvoie `True`, tandis que les appels `est_balise("<html")` et `est_balise("html<>")` renvoient `False`.

Solution

```
1   def est_balise(balise):
2       """
3       balise : str
4       retour : bool
5       Renvoie True si balise est de longueur >= 2, commence par un
6       chevron ouvrant et invalidet par un chevron fermant, False sinon
```

```
7       """
8
9       return len(balise) >= 2 and balise[0] == "<" and balise[-1] == ">"
```

Le contenu d'une balise est constitué de son nom suivi éventuellement d'attributs. On appelle donc **nom d'une balise** le premier mot de son contenu, c'est-à-dire la chaine de caractères qui commence immédiatement après le chevron ouvrant et se termine soit juste avant le premier espace, soit juste avant le chevron fermant. Par exemple, le nom de la balise `<h1 id="titre1">` est `"h1"`, tout comme celui de la balise `<h1>`. Le nom de la balise `</h1>` est `"/h1"`.

Question 8

Écrire une fonction `nom_balise` qui prend en paramètre une balise `b` et qui renvoie son nom.

Solution

```
1  def nom_balise(balise):
2      """
3      balise: str
4      retour: str
5      préconditions : est_balise(balise)
6      Renvoie le nom de la balise sans les chevrons
7      """
8      nom_b = ""
9      i = 1
10     while i < len(balise) and balise[i] != " " and balise[i] != ">":
11         nom_b = nom_b + balise[i]
12         i = i + 1
13     return nom_b
```

Question 9

Écrire une fonction `est_fermant` qui prend en paramètre un nom de balise `nom_b` et qui teste si `nom_b` est le nom d'une balise fermante.
Par exemple `est_fermant("html")` renvoie `False` et `est_fermant("/html")` renvoie `True`.

Solution

```
1  def est_fermant(nom_balise):
2      """
3      nom_balise : str
4      retour : bool
5      préconditions : est_balise(balise)
6      Renvoie True si nom_balise commence par '/', False sinon
7      """
8      return nom_balise[0] == "/"
```

Question 10

Écrire une fonction `est_ouvrant` qui prend en paramètre un nom de balise `nom_b` et qui teste si `nom_b` est le nom d'une balise ouvrante.

Par exemple `est_ouvrant("html")` renvoie `True` et `est_ouvrant("/html")` renvoie `False`.

Solution

```python
def est_ouvrant(nom_balise):
    """
    balise : str
    retour : bool
    préconditions :
    Renvoie True si nom_balise commence par un caractère différent de '/'
    False sinon
    """
    return not est_fermant(nom_balise)
```

En HTML, certains noms de balises sont standardisés (comme `body`, `h1`, `/p`, `/html`, etc.), tandis que d'autres (comme `toto` ou `/tata`) n'existent pas et ne sont pas autorisés. Pour savoir quels noms de balise sont autorisés, on dispose d'une liste contenant les noms de toutes les balises ouvrantes autorisées. Les balises fermantes autorisées sont les balises fermantes correspondant aux balises ouvrantes de cette liste. Dans ce sujet par exemple, la liste autorisée est `liste_autorisee = ["html", "head", "title", "body", "h1", "h2", "h3", "h4", "h5", "h6", "p", "ol", "li", "strong", "br"]`.

Question 11

Écrire une fonction `nom_autorise` qui prend en paramètres un nom de balise `nom_balise` et une liste de noms des balises ouvrantes autorisées `liste_nbo` et qui teste si `nom_balise` est autorisé.

Solution

```python
def nom_autorise(nom_balise, liste_nbo):
    """
    nom_balise : str, liste_nbo : lst de str
    retour : bool
    Teste si nom_balise est ouvrant et dans liste_nbo
    ou s'il est fermant et dans liste_nbo sans le / initial
    """
    if est_fermant(nom_balise):
        return nom_balise[1:] in liste_nbo
    else:
        return nom_balise in liste_nbo
```

Dans un code HTML, la plupart des balises fonctionnent par paire : une portion de code qui commence par une balise ouvrante se termine par la balise fermante correspondante. Par exemple, un paragraphe qui commence par une balise ouvrante de nom `p` doit être fermé par une balise fermante de nom `/p`.

Question 12

Écrire une fonction `correspondance` qui prend en paramètres deux noms de balises `nom_b1` et `nom_b2` et qui teste si `nom_b2` est le nom de balise fermante correspondant au nom de balise ouvrante `nom_b1`. Par exemple `correspondance("p","/p")` et `correspondance("h1","/h1")` renvoient `True` tandis que `correspondance("p","/h1")` renvoie `False`.

Solution

```python
def correspondance(nom1, nom2):
    """
    nom1 : str, nom2 : str
    retour : bool
    Renvoie True si nom2 est la balise fermante de nom1, False sinon
    """
    verif1 = est_ouvrant(nom1) and est_fermant(nom2)
    verif2 = ('/' + nom1) == nom2
    return verif1 and verif2
```

2.3 : Règle de placement des balises

On souhaite analyser désormais un code HTML donné dans sa totalité et supposé **bien chevronné**, pour vérifier que les balises sont bien placées les unes par rapport aux autres. Pour cela, on commence par extraire la liste des noms des balises de ce code HTML, les portions de code en dehors des balises n'étant pas utiles pour faire cette vérification.

Question 13

Écrire une fonction `extrait_noms` qui prend en paramètre une chaine de caractères `codeHTML` et qui renvoie la liste des noms de toutes les balises, ouvrantes et fermantes, présentes dans `codeHTML`.
Par exemple pour `codeHTML1 = "<html><body><h1 id="titre1">Titre</h1></body></html>"`, `extrait_noms(codeHTML1)` renvoie `["html","body","h1","/h1","/body","/html"]`.

Solution

```python
def extrait_noms(html):
    """
    html: str
    retour: lst de str
    Renvoie la liste des noms de toutes les balises utilisées dans html
    """
    liste_balises = []
    curseur = 0
    while curseur <= len(html) - 1:
        acc = 1
        if html[curseur] == "<":
            while curseur + acc < len(html) and html[curseur+acc] != ">" :
                acc += 1
            liste_balises.append(nom_balise(html[curseur:curseur + acc +
             1]))
        curseur += acc
    return liste_balises
```

On l'a dit, en HTML certaines balises fonctionnent par paire. Les autres fonctionnent seules : on les appelle balises **orphelines**. Par exemple, comme on a pu le voir en figure 1, la balise `<br>` qui permet un saut de ligne n'appelle pas de balise fermante correspondante : c'est une balise orpheline. De même, la balise `<img src = "mon_image.png">` qui permet d'insérer une image nommée `mon_image.png` ne nécessite pas de balise fermante. Cette balise, ainsi que toutes les balises de nom `img`, sont des balises orphelines.

On admet qu'on dispose d'une fonction est_orpheline qui prend en paramètre un nom de balise nom_b et qui renvoie True s'il s'agit d'un nom d'une balise orpheline, False sinon.

Question 14

Écrire une fonction sans_orphelines qui prend en paramètre une liste de noms de balises liste_noms et qui renvoie une nouvelle liste composée des noms de liste_noms qui ne sont pas le nom d'une balise orpheline. Cette fonction ne doit pas modifier la liste prise en paramètre.
Par exemple sans_orphelines(["html","body","img","h1","/h1","br","/body","/html"]) renvoie ["html","body","h1","/h1","/body","/html"].

Solution

```python
def sans_orphelines(liste_noms):
    """
    liste_noms: str
    retour: list de str
    Renvoie la liste des noms de liste_noms qui ne sont pas le nom
    d'une balise orpheline
    """
    return [nom for nom in liste_noms if not est_orpheline(nom)]
```

Pour qu'un code HTML soit valide, il faut qu'il soit **bien balisé**, c'est-à-dire que les règles suivantes concernant les balises par paires soient respectées :

a) chaque balise ouvrante doit être fermée par une balise fermante correspondante et vice-versa (chaque balise fermante doit correspondre à une balise ouvrante) ;

b) les balises doivent être imbriquées correctement : si une balise b2 est ouverte avant que la balise b1 qui la précède soit fermée, il faut fermer la balise b2 avant de fermer la balise b1.

Par exemple les codes HTML "<h1> Titre </h1> Paragraphe </p>" et "<h1> Titre </h1> <p> Paragraphe" ne respectent pas la condition *a*). Le code HTML "<h1> Titre </h1></h1>" ne respecte pas non plus cette condition, car seule l'une des deux balises </h1> a été ouverte.
Le code HTML "<body> <p> Paragraphe </body> </p>" respecte la condition *a*) mais pas la condition *b*) : la balise <p> devrait être fermée avant de fermer la balise <body>.
Le code HTML "<h1> Titre </h1><p> Voici un paragraphe </p>" quant à lui est bien balisé.

Question 15

a) Le code HTML suivant est-il bien balisé ? Justifier.
"<body><p id="para1"> Voici un paragraphe <h1> et un titre </p></h1></body>"

b) Recopier et compléter le code HTML suivant pour qu'il soit bien balisé.
"<body> <p> Voici un paragraphe important"

Solution

a) Ce code est mal balisé car la balise <h1> est ouverte après <p> mais fermée après <\p>.

b) "<body> <p> Voici un paragraphe important</p></body>"

On souhaite écrire une fonction html_valide qui prend en paramètres un code source codeHTML et la liste des noms des balises ouvrantes autorisées liste_nbo et qui teste si le code est bien balisé.

Pour cela, on extrait tout d'abord du code HTML la liste des noms des balises qui fonctionnent par paire grâce aux fonctions précédentes. On étudie alors les éléments présents dans cette liste pour tester si les balises sont bien organisées. Pour réaliser cette étude on utilise une pile.

Utilisation des piles

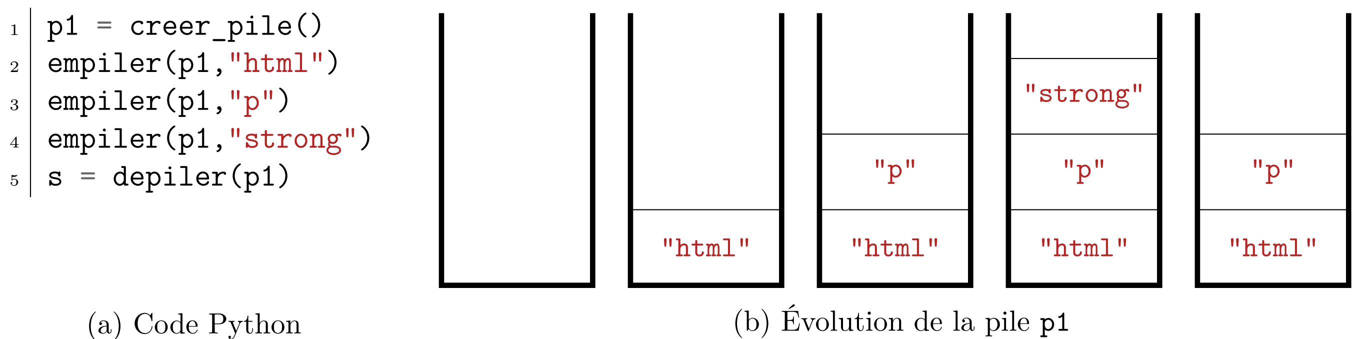
Une **pile** est une structure de données linéaire dont voici les principales caractéristiques.

- Une pile peut contenir aucun, un ou plusieurs éléments de manière ordonnée.
- Lorsqu'un élément est ajouté à une pile, il est placé au sommet de la pile.
  On dit alors qu'on **empile** cet élément.
- Seul l'élément placé au sommet de la pile est accessible et lui seul peut être supprimé de la pile.
  On dit alors qu'on **dépile** la pile.

On suppose ici que l'on dispose des fonctions suivantes pour manipuler des piles en Python :

- `creer_pile` qui renvoie une pile vide ;
- `est_vide` qui prend en paramètre une pile et renvoie `True` si elle est vide, `False` sinon ;
- `empiler` qui prend en paramètres une pile et un élément et qui modifie la pile de façon à empiler cet élément ;
- `depiler` qui prend en paramètre une pile, renvoie le sommet de la pile et modifie la pile de façon à la depiler ;
- `sommet` qui prend en paramètre une pile et renvoie l'élément au sommet de la pile sans la dépiler.

Par exemple, lorsqu'on exécute les instructions de la figure 5(a) la pile `p1` évolue comme représenté à la figure 5(b). À la fin de cette exécution la variable `s` contient la chaine de caractères `"strong"`.




```
1 p1 = creer_pile()
2 empiler(p1,"html")
3 empiler(p1,"p")
4 empiler(p1,"strong")
5 s = depiler(p1)
```

(a) Code Python

(b) Évolution de la pile `p1`

FIGURE 5 – Exemple d'utilisation d'une pile.

Pour vérifier le bon balisage d'un code, on utilise une pile initialement vide. On parcourt la liste des noms de balises du code HTML (sans les balises orphelines), et pour chaque nom de balise `nom_b` on applique les règles suivantes :

- si `nom_b` est un nom de balise ouvrante :
  - si `nom_b` fait partie des balises autorisées, il est empilé puis on continue le parcours de la liste ;
  - sinon on peut conclure qu'il s'agit d'un mauvais balisage ;
- si `nom_b` est un nom de balise fermante :
  - si la pile est vide, on peut conclure qu'il s'agit d'un mauvais balisage ;
  - si la pile n'est pas vide, il faut comparer `nom_b` avec le sommet de la pile :
    - · si le sommet de la pile est le nom de balise ouvrante correspondant à `nom_b`, c'est signe d'un bon balisage ponctuel, on dépile puis on continue le parcours de la liste ;
    - · sinon, on peut conclure qu'il s'agit d'un mauvais balisage.

À la fin du parcours, pour conclure que le code est bien balisé, il faut que la pile soit vide.

Les figures 6 et 7 illustrent le déroulement de l'algorithme pour les deux codes HTML suivants.

```
codeHTML2 = "<body><h1> Titre </h1> <p> Premier paragraphe </p></body>"
codeHTML3 = "<body><h1> Titre<p> Premier paragraphe </h1></p></body>"
```

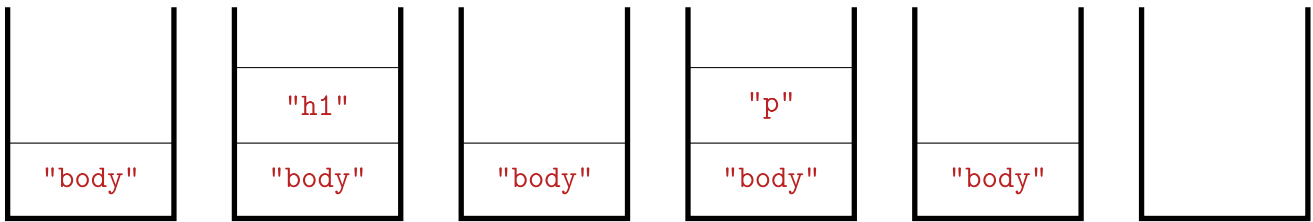


FIGURE 6 – Étapes de l'algorithme sur `codeHTML2`, la valeur renvoyée à la fin est `True`.

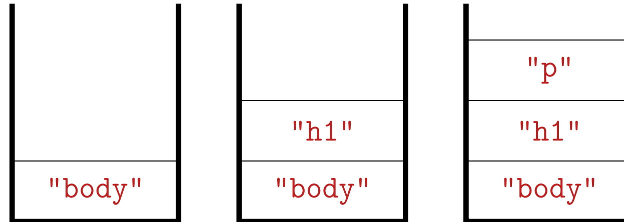


FIGURE 7 – Étapes de l'algorithme sur `codeHTML3`, la valeur renvoyée à la fin est `False`.

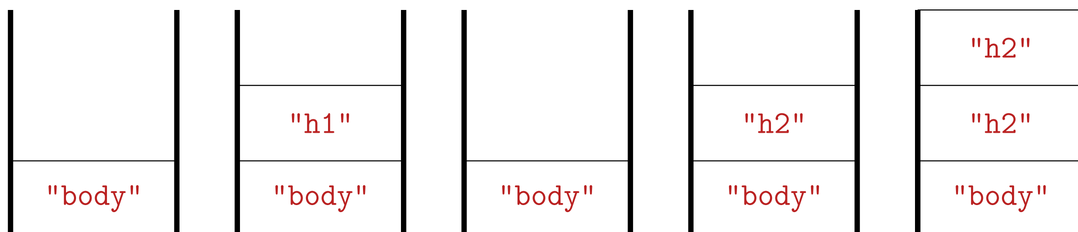
Question 16
À la manière des schémas des figures 6 et 7, indiquer l'évolution de la pile ainsi que la valeur renvoyée lorsqu'on applique l'algorithme précédent sur les deux codes HTML suivants.

a) `codeHTML4 = "<body><h1> Titre </h1> <h2> Sous titre<h2></body>"`

b) `codeHTML5 = "</body><h1> Titre </h1> <h2> Sous titre</h2></body>"`

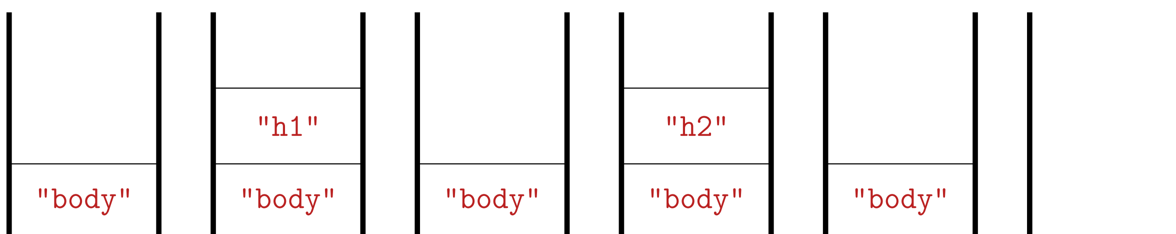
Solution

Étapes de l'algorithme sur `codeHTML4`, la valeur renvoyée à la fin est `False`



Étapes de l'algorithme sur `codeHTML5` La pile est créée vide, mais lorsqu'on traite la première balise, qui est fermante, on s'arrête et on renvoie `False`.

Autre exemple non demandé par l'énoncé Avec `codeHTML6 = "<body><h1> Titre </h1> <h2> Sous titre</h2></body>"` la valeur renvoyée à la fin est `True`.



Question 17
Écrire le code de la fonction `html_valide`.

Solution

```
def html_valide(html, liste_nbos):
    """
    html: str, liste_nbos:lst de str
    retour: bool
    préconditions : bien_chevrone(html_valide)
    Renvoie True si le fichier html est correctement balisé
    """
    pile = creer_pile()
    liste_nom_b = sans_orphelines(extrait_noms(html))
    i = 0
    invalide = False
    while i < len(liste_nom_b) and not invalide:
        nom_b = liste_nom_b[i]
        if nom_autorise(nom_b, liste_nbos):
            if est_ouvrant(nom_b):
                empiler(pile, nom_b)
            elif est_fermant(nom_b):
                if est_vide(pile):
                    invalide = True
                else:
                    if correspondance(sommet(pile), nom_b):
                        depiler(pile)
                    else:
                        invalide = True
        else:
            invalide = True
        i = i + 1
    return est_vide(pile) and not invalide
```

On a profité ici de la fonction `nom_autorise` pour tester si la balise est autorisée avant de tester si elle est ouvrante ou fermante. Inverser ces deux tests n'est pas pénalisé.

Partie 3 : Extraire la structure d'un code HTML valide

Dans cette partie, on souhaite écrire un programme Python qui puisse générer un sommaire de manière automatique à partir du code HTML, **supposé valide**, d'une page web.

3.1 : Notion de sommaire

L'exemple de la figure 8 illustre le résultat que nous souhaitons obtenir : sur la partie gauche de la page est visible le sommaire incluant tous les titres et sous-titres présents dans la page. Les différents niveaux de ceux-ci sont représentés à l'aide d'indentations :

- le titre « Description de HTML », comme les autres titres de niveau 1, n'est pas indenté ;
- le titre « Syntaxe de HTML », comme les autres titres de niveau 2, est indenté par rapport aux titres de niveau 1 ;

- le titre « Origine du côté de SGML », comme les autres titres de niveau 3, est indenté par rapport aux titres de niveau 2.

FIGURE 8 – Extrait de la page web de Wikipédia sur « Hypertext Markup Language »

On fait l'hypothèse que, comme dans cet exemple, les titres de la page ne sautent pas de niveau : un titre de niveau 3 par exemple ne peut suivre directement un titre de niveau 1, il y a nécessairement un titre de niveau 2 entre les deux. De plus, afin de regrouper tous les titres de la page dans un même sommaire, on considère que le premier titre de la page est le titre fictif de niveau `h0` et d'intitulé `"Sommaire"`.

Pour décrire la structure hiérarchique du sommaire d'une page web, nous introduisons une relation de parenté entre ses titres.

- Le titre fictif `"Sommaire"`, de niveau `0`, est le seul titre qui ne possède pas de parent.
- Chaque titre `t` de niveau `n` possède un **unique parent** : c'est le titre de niveau `n-1` qui le précède dans la page. Dans le cas où plusieurs titres de niveau `n-1` le précèdent, c'est le dernier d'entre eux, c'est-à-dire le plus proche de `t` dans la page, qui est son parent.
- Si le titre `t1` est parent du titre `t2`, on dit que `t2` est un **enfant** du titre `t1`.
  Ainsi un titre de niveau `n` peut avoir zéro, un ou plusieurs enfants, mais tous sont de niveau `n+1`.

En reprenant l'exemple de la figure 8,

- les titres « Début », « Dénomination », « Évolution du langage », « Description de HTML », « Interopérabilité de HTML », « Notes et références » et « Bibliographie » sont les sept enfants du titre fictif « Sommaire » ;
- le titre « Description de HTML » a cinq enfants, le premier étant « Syntaxe de HTML » ;
- le titre « Syntaxe de HTML » a deux enfants, et ceux-ci n'ont pas d'enfants.

Question 18

Donner la liste des titres ancêtres du titre « Technique d'échappement » de la figure 8. Autrement dit remonter de parent en parent depuis ce titre tant que cela est possible. Préciser pour chaque titre son niveau.

Solution

- « Jeux de caractères » de niveau `2`
- « Description de HTML » de niveau `1`
- « Sommaire » de niveau `0`

On ne pénalisera pas si « Technique d'échappement » de niveau `3` figure dans la liste des ancêtres (on n'a pas définit la relation ancêtre, ce qui peut laisser penser que c'est la cloture **réflexive** transitive de la relation parent).

On suppose qu'on a déjà extrait du code HTML la liste de ses titres sous la forme d'une liste de couples `(nom_balise: str, intitule: str)` comme en figure 9(a). On cherche alors à produire un sommaire comme en figure 9(b). La première étape consiste à extraire de la liste de titres la structure hiérarchique du sommaire, la seconde consiste à afficher le sommaire connaissant sa structure.

```
l_titres_w = [
  ("h1", "Description de HTML"),
  ("h2", "Syntaxe de HTML"),
  ("h3", "Origines du côté de SGML"),
  ("h3", "Syntaxe HTML décortiquée"),
  ("h2", "Structure des doccuments HTML")
]
```

(a) Liste de titres HTML

```
Sommaire
  Description de HTML
    Syntaxe de HTML
      Origines du côté de SGML
      Syntaxe HTML décortiquée
    Structure des documents HTML
```

(b) Sommaire correspondant

FIGURE 9 – Entrée et sortie de notre production de sommaire.

3.2 : Représentation de la structure hiérarchique des titres en Python

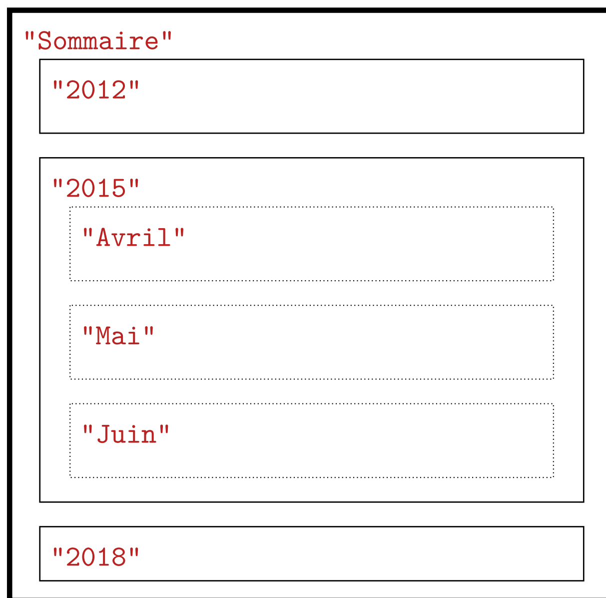
Afin de représenter la structure hiérarchique du sommaire en Python, nous introduisons une nouvelle représentation des titres, dans laquelle chaque titre stocke la liste de ses enfants. Ainsi un **titre hiérarchique** est un triplet (c'est-à-dire un `tuple` à trois éléments) formé comme suit :

- le premier élément, de type `int`, donne le niveau du titre (entre 0 et 6) ;
- le deuxième élément, de type `str`, donne l'intitulé du titre ;
- le troisième élément, de type `list`, donne la liste de tous les titres enfants, représentés eux aussi par un tel triplet.

La figure 10(b) illustre la structure hiérarchique de la page web de la figure 10(a), la figure 10(c) donne le titre hiérarchique qui représente cette structure en Python.
En Python, les indentations et retours à la ligne ne sont pas pris en compte dans une définition de liste en extension. Cela permet d'écrire du code plus lisible comme en figure 10(c).

On souhaite écrire une fonction `creer_hierarchie` qui prend en paramètre une liste de titres HTML et en extrait la structure sous la forme d'un titre hiérarchique. Par exemple le titre hiérarchique de la figure 10(c) serait le résultat de l'appel de `creer_hierarchie` sur la liste `lt_meteo` suivante.

(a) Page web



(b) Structure hiérarchique

```
(0, "Sommaire", [
    (1, "2012", []),
    (1, "2015", [
        (2, "Avril", []),
        (2, "Mai", []),
        (2, "Juin", [])
    ]),
    (1, "2018", [])
])
```

(c) Titre hiérarchique

FIGURE 10 – Structure d'une page web induite par ses titres et représentation en Python.

```
[("h1","2012"), ("h1","2015"), ("h2","Avril"), ("h2","Mai"), ("h2","Juin"), ("h1","2018")]
```

Question 19

a) Donner le titre hiérarchique que renvoie `creer_hierarchie` pour la liste de titres ci-contre (à gauche).

b) Donner la liste de titres HTML pour laquelle `creer_hierarchie` renvoie le titre hiérarchique ci-contre (à droite).

```
[("h1", "Impératif"),
 ("h1", "Déclaratif"),
 ("h2", "Fonctionnel"),
 ("h3", "OCaml"),
 ("h2", "Logique"),
 ("h1", "Objet")
]
```

```
(0, "Sommaire", [
  (1, "Générer", []),
  (1, "Valider", []),
  (1, "Structure", [
    (2, "Sommaire", []),
    (2, "Hiérarchie", [
      (3, "Q21", [])
    ])
  ])
])
```

Solution

a)

```
(0, "Sommaire", [
    (1, "Impératif", []),
    (1, "Déclaratif", [
        (2, "Fonctionnel", [
            (3, "OCaml", [])
        ]),
        (2, "Logique", [])
    ]),
    (1, "Objet", [])
])
```

b)

```
[("h1", "Générer"), ("h1", "Valider"), ("h1", "Structure"), ("h2",
↪   "Sommaire"),("h2", "Hiérarchie"), ("h3", "Q21")]
```

Afin d'extraire la structure hiérarchique d'une liste de titres HTML, nous proposons l'algorithme suivant qui utilise la structure de `Pile` introduite page 12.

- On crée un triplet `sommaire` représentant le titre hiérarchique de niveau 0, d'intitulé `"Sommaire"` et initialement sans enfant.
- On crée une pile vide et on y empile `sommaire`.

- Pour chaque titre `t` de la liste de titres :
  - on récupère l'entier `nt` donnant le niveau du titre `t` ;
  - on crée un triplet `th` représentant le titre de niveau `nt`, d'intitulé celui de `t` et sans enfant ;
  - on compare le niveau du titre `nt` au niveau du sommet de la pile :
    - tant que le sommet de la pile est de niveau `ns > nt-1`, on dépile la pile ;
    - dès que `ns = nt-1`, on a trouvé le parent du titre `t` dans la hiérarchie, on ajoute donc `th` aux enfants du sommet de la pile, puis on empile `th` ;
    - le cas `ns < nt-1` ne devrait pas se produire car les titres ne sautent pas de niveau.

La figure 11 illustre l'état de la pile lors de l'exécution de l'algorithme sur la liste `lt_meteo`. Chaque élément de la pile stocke une référence vers un titre hiérarchique en cours de remplissage, c'est-à-dire un titre auquel on peut encore ajouter des enfants. Afin de ne pas alourdir la présentation, les enfants de chaque titre ne sont pas explicités et `"Sommaire"` a été abrégé en `"Somm."`.

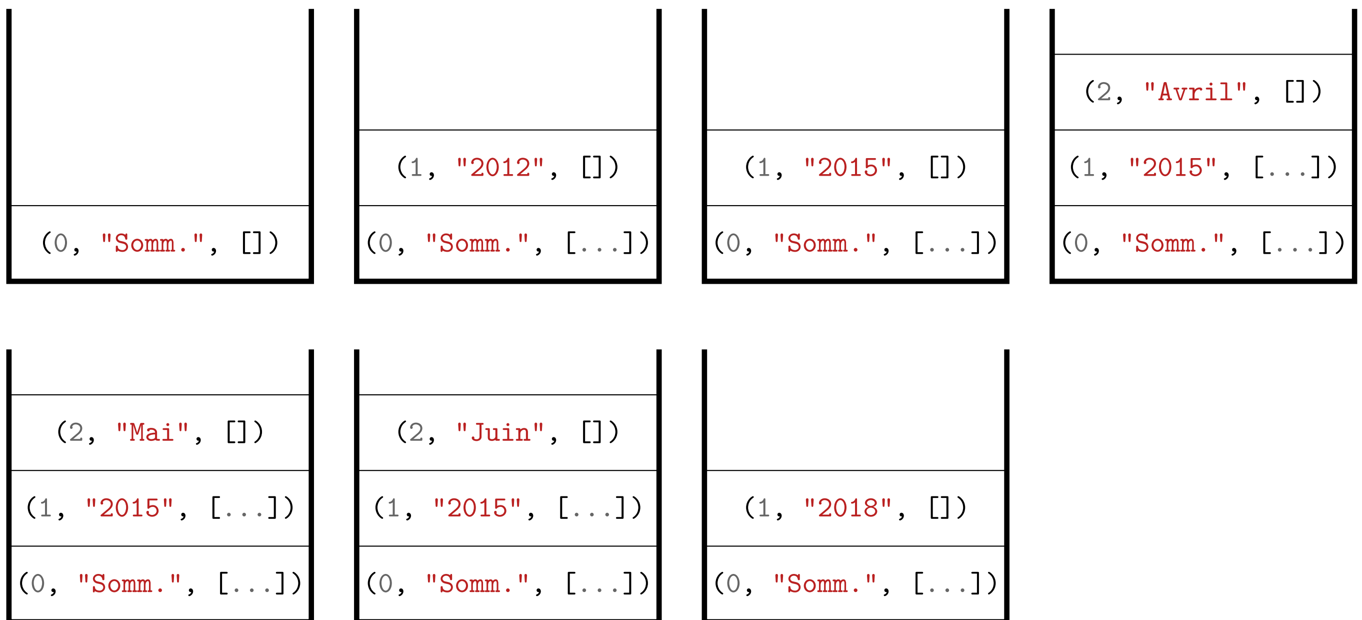


FIGURE 11 – Évolution de la pile lors de l'appel `creer_hierarchie(lt_meteo)`.

Question 20

Écrire le code de la fonction `creer_hierarchie`.

Solution

```python
def creer_hierarchie(liste_titres_html):
    """
    liste_titres_html :
    retour :
    Renvoie ...
    """
    sommaire = (0, "Sommaire", [])
    pile = creer_pile()
    empiler(pile, sommaire)
    for i in range(0, len(liste_titres_html)):
        (str_niv, str_intitule) = liste_titres_html[i]
```

```python
        nt = int(str_niv[1]) #on récupère le niveau sous forme d'entier
        th = (nt, str_intitule, [])
        (ns,_,_) = sommet(pile)
        while ns > nt - 1:
          depiler(pile)
          (ns,_,_) = sommet(pile)
        #ici sommet(pile) est le titre parent de th
        sommet(pile)[2].append(th) #on ajoute th à la liste des enfants
        empiler(pile, th) #on empile th
    return sommaire
```

3.3 : Génération du sommaire d'une page web

Dans cette dernière partie, nous allons générer une chaine de caractères permettant d'afficher le sommaire d'une page HTML à partir d'un titre hiérarchique.

Rappel

Les chaines de caractères Python peuvent utiliser des caractères spéciaux :

- `"\n"` permet d'obtenir un saut de ligne ;
- `"\t"` permet d'obtenir une indentation.

On donne ci-contre un exemple d'une chaine utilisant ces caractères et son affichage dans la console.

```
>>> texte = "I.\nII.\n\t1.\n\t2.\nIII."
>>> print(texte)
I.
II.
  1.
  2.
III.
```

Question 21

Écrire une fonction `creer_sommaire` qui prend en paramètre un titre hiérarchique `titre_h` et qui renvoie la chaine de caractères permettant d'afficher le sommaire correspondant comme illustré à la figure 9(b).

Solution

Cette fonction revient à faire un parcours en profondeur de l'arborescence qu'est `titre_h`. On propose une version à l'aide d'une pile, et une version à l'aide d'une fonction auxiliaire récursive. Dans la version à l'aide d'une pile, on remarque l'importance de parcourir la listes des enfants d'un titre à l'envers, afin d'empiler en dernier le premier des enfants, ainsi c'est le premier enfant de la liste qui sera traité en premier.

Version avec une pile

```python
def creer_sommaire(titre_h):
    """
    titre_h: titre hiérarchique
    retour: str
    Renvoie une chaine permettant d'imprimer le sommaire qu'encode
    titre_h avec des indentations et retours à la ligne
    """
    res = ""
    pile = creer_pile()
```

```python
    empiler(pile, titre_h)
    while not est_vide(pile) :
        titre_courant = depiler(pile)
        res += "\t" * titre_courant[0] + titre_courant[1] + "\n"
        liste_enfants = titre_courant[2]
        #on parcourt les enfants de droite à gauche pour les empiler
        for i in range(len(liste_enfants)-1, -1, -1):
            empiler(pile, liste_enfants[i])
    return res
```

Version récursive (non récursive terminale)

```python
def creer_sommaire_ter(titre_h):
    """
    titre_h: titre hiérarchique
    retour : str
    Renvoie une chaine permettant d'imprimer le sommaire qu'encode
    titre_h avec des indentations et retours à la ligne
    """
    (niv, intitule, enfants) = titre_h
    res = "\t"*niv + intitule + "\n"
    for enfant in enfants :
        res += creer_sommaire(enfant)
    return res
```

Version avec une fonction auxiliaire récursive

```python
def creer_sommaire_bis(titre_h):
    """
    titre_h: titre hiérarchique
    retour : str
    Renvoie une chaine permettant d'imprimer le sommaire qu'encode
    titre_h avec des indentations et retours à la ligne
    """
    (niv, intitule, enfants) = titre_h
    assert (niv == 0)
    creer_sommaire_aux (titre_h, "", "")
```

```python
def creer_sommaire_aux(titre_h, decalage, debut):
    """
    titre_h: titre hiérarchique, decalage : str, debut : str
    retour : str
    Renvoie la chaine constituée de debut puis d'une chaine
    correspondant au sommaire qu'encode titre_h
    où toutes les lignes sont préfixées par decalage
    """
    (niv, intitule, enfants) = titre_h
    res = debut
    res = res + decalage + intitule + "\n"
    for enfant in enfants :
        res = creer_sommaire_aux(enfant, decalage + "\t", res)
    return res
```

Exercice 2 : Gestion d'un tournoi

Cet exercice comporte trois parties qui sont indépendantes.

Partie 1 : Simulation du tournoi

On s'intéresse à l'organisation de tournois à élimination directe entre n joueurs. On choisit de numéroter les joueurs de 0 à $n-1$ et on organise $n-1$ matchs numérotés de n à $2n-2$ qui sont effectués dans cet ordre. Chaque match conduit à l'élimination d'un joueur, si bien qu'à l'issue du dernier match ne subsiste que le vainqueur du tournoi. On peut représenter un tel tournoi à l'aide d'une structure arborescente.

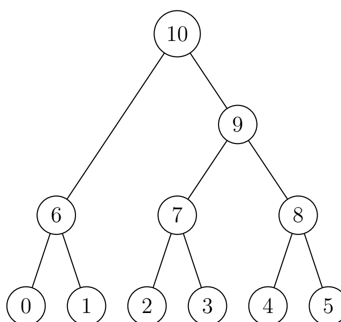


FIGURE 12 – Premier exemple de tournoi avec $n = 6$ joueurs.

Dans le tournoi représenté en figure 12 le premier match est le match 6 qui oppose les joueurs 0 et 1. Le match suivant, numéroté 7, oppose les joueurs 2 et 3 puis le match 8 oppose 4 et 5. Pour le match 9 ce sont les gagnants des matchs 7 et 8 qui se rencontrent. Le vainqueur du match 9 affronte alors le vainqueur du match 6 lors du dernier match 10.

On représente en Python un tournoi à n joueurs par une `list` **de** `lists` `L`, respectant les trois conditions suivantes.

a) Les n premiers éléments de `L` sont des `lists` vides, ils correspondent aux joueurs qui entrent dans le tournoi. Ainsi pour $0 \leqslant$ `k` $\leqslant n-1$, `L[k]` vaut `[]`.

b) Les $n-1$ éléments suivants correspondent aux différents matchs. Un match est représenté par une `list` `[k_1, k_2]` permettant de déterminer les deux participants du match. Ainsi `k1` et `k2` sont soit directement des indices de joueurs, soit des indices de matchs **antérieurs** (c'est-à-dire des matchs déjà joués). En d'autres termes, pour $n \leqslant$ `k` $\leqslant 2n-2$, `L[k]` est de la forme `[k1, k2]` avec `k1` $\neq$ `k2` (un joueur ne peut jouer contre lui-même) et `k` doit être strictement plus grand que `k1` et `k2` (`k1` et `k2` sont des matchs antérieurs ou des joueurs).

c) Pour tout entier $0 \leqslant$ `k` $\leqslant 2n-3$, il existe un unique entier `j` $>$ `k` tel que `k` figure dans la `list` `L[j]`. En d'autres termes, le gagnant du match `k` d'indice strictement inférieur à $2n-2$ (c'est-à-dire sauf pour la finale) est qualifié pour participer à un unique match d'indice strictement supérieur à `k` (c'est-à-dire qui se déroule ultérieurement).

Le tournoi de la figure 12 est donc représenté par la `list` de `lists` suivante.

```
L1 = [ [], [], [], [], [], [], [0,1], [2,3], [4,5], [7,8], [6,9] ]
```

Deuxième exemple de tournoi. Dans le tournoi représenté en figure 13 , 6 joueurs s'affrontent. Le premier match est le match 6 qui oppose les joueurs 0 et 5. Le match suivant, numéroté 7, oppose le joueur vainqueur du match 6 et le joueur 2. Le match 8 oppose les joueurs 4 et 1. Pour le match 9 ce sont les gagnants des matchs 7 et 8 qui se rencontrent. Enfin, le vainqueur du match 9 affronte le joueur 3 lors du dernier match 10.

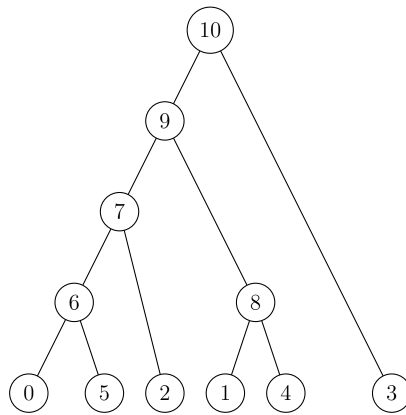


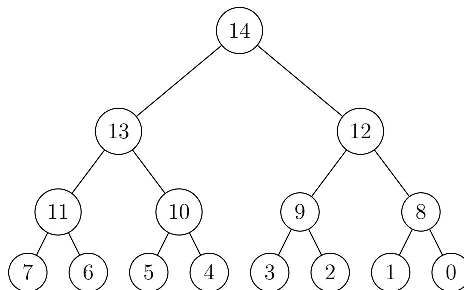
FIGURE 13 – Deuxième exemple de tournoi avec $n = 6$ joueurs.

Le tournoi de figure 13 est donc représenté par la `list` de `lists` suivante.

`L2 = [ [], [], [], [], [], [], [0,5], [2,6], [1,4], [7,8], [3,9] ]`

Question 1

Donner la `list` de `lists` représentant le tournoi suivant.



Solution

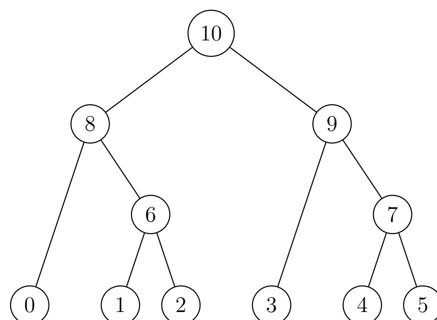
`[[],[],[],[],[],[],[],[],[0, 1], [2, 3], [4, 5], [6, 7], [8, 9], [10, 11], [12, 13]]`

Question 2

Dessiner le tournoi représenté par la `list` de `lists` suivante.

`[[], [], [], [], [], [], [1, 2], [4, 5], [0, 6], [3, 7], [8, 9]]`

Solution



Question 3

Écrire une fonction `condition_a` qui prend en paramètre une `list` de `lists` `L` ainsi qu'un entier `n` et qui renvoie `True` si `L` respecte bien la condition a), `False` sinon.

On rappelle la condition a) : **les n premiers éléments de la `list` sont des `lists` vides.**

Solution

```
1  def condition_a(L, n):
2      """
3      list[int] * int -> bool
4      Entrée : Une liste L et un entier n
5      Sortie : Un booléen testant si L satisfait a)
6      """
7      for k in range(n):
8          if L[k] != []:
9              return False
10     return True
```

Question 4

Écrire une fonction `condition_b` qui prend en paramètre une `list` de `lists` `L` ainsi qu'un entier `n` et qui renvoie `True` si `L` respecte bien la condition b), `False` sinon.

On rappelle la condition b) : **pour tout entier `k` avec $n \leqslant$ `k` $\leqslant 2n - 2$, `L[k]` est de la forme `[k_1, k_2]` avec `k1` $\neq$ `k2` et `k` est strictement plus grand que `k1` et `k2` .**

Solution

```
1  def condition_b(L, n):
2      """
3      list[int] * int -> bool
4      Entrée : Une liste L et un entier n
5      Sortie : Un booléen testant si L satisfait b)
6      """
7      for k in range(n, 2 * n - 1):
8          if len(L[k]) != 2:
9            return False
10         k1, k2 = L[k][0], L[k][1]
11         if not (k1 != k2 and k > k1 and k > k2):
12             return False
13     return True
```

Question 5

Recopier et compléter les lignes 6 à 9 du code de la fonction `condition_c` ci-après. Cette fonction prend en paramètre une `list` de `lists` `L` ainsi qu'un entier `n` et renvoie `True` si `L` respecte bien la condition c), `False` sinon.

On rappelle la condition c) : **pour tout entier `k` avec $0 \leqslant$ `k` $\leqslant 2n - 3$, il existe un unique entier `j` $>$ `k` tel que `k` figure dans la `list` `L[j]`.**

```
1  def condition_c(L, n):
2    deja_vu = [False] * (2 * n - 2)
```

```
3     for k in range(n, 2 * n - 1):
4         k1, k2 = L[k][0], L[k][1]
5         if deja_vu[k1] or deja_vu[k2]:
6             return ...
7         deja_vu[k1] = ...
8         deja_vu[k2] = ...
9     return ...
```

Solution

```
1 def condition_c(L, n):
2     """
3     list[int] * int -> bool
4     Entrée : Une liste L et un entier n
5     Sortie : Un booléen testant si L satisfait c)
6     """
7     deja_vu = [False] * (2 * n - 1)
8     for k in range(n, 2 * n - 1):
9         k1, k2 = L[k][0], L[k][1]
10        if deja_vu[k1] or deja_vu[k2]:
11            return False
12        deja_vu[k1] = True
13        deja_vu[k2] = True
14    return True
```

Il est ainsi possible de déterminer si une `list` de `lists` `L` représente bien un tournoi entre n joueurs grâce à la fonction suivante.

```
1 def est_valide(L, n):
2     return condition_a(L, n) and condition_b(L, n) and condition_c(L, n)
```

Un **oracle** est un tableau carré de taille $n \times n$ dont la case de ligne i et de colonne j prévoit le résultat de la rencontre entre le joueur i et le joueur j. En Python on utilisera une `list` composée de n `lists` de taille n. Ainsi `oracle[i][j]` vaut `True` si le joueur i va gagner contre le joueur j et `oracle[i][j]` vaut `False` sinon. Il est impossible qu'il y ait match nul. Les cases de la diagonale peuvent contenir des booléens quelconques (elles ne sont pas utilisées car un joueur ne se rencontre jamais lui-même).

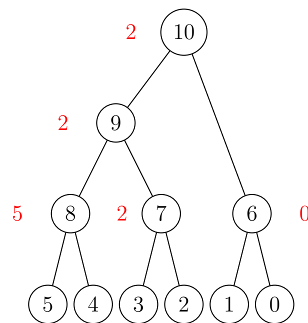
La figure 14 donne un exemple d'oracle pour 6 joueurs. Cet oracle est représenté en Python par la `list` de `lists` `[[True, True, False, True, True, False], [False, True, ...], ...]`

i \ j	0	1	2	3	4	5
0	True	True	False	True	True	False
1	False	True	True	True	False	True
2	True	False	True	True	False	True
3	False	False	False	True	False	False
4	False	True	True	True	False	False
5	True	False	False	True	True	True

FIGURE 14 – Oracle pour 6 joueurs.

Question 6
Dessiner le tournoi de la figure 12 en mettant en évidence les vainqueurs des différents matchs selon l'oracle de la figure 14.

Solution



Une `list` composée de n `lists` de booléens de taille n est **antisymétrique** quand pour tout `i` $\neq$ `j`, si `oracle[i][j]` vaut `True` alors `oracle[j][i]` vaut `False` et si `oracle[i][j]` vaut `False` alors `oracle[j][i]` vaut `True`.

On remarque qu'un oracle est **antisymétrique**.

Question 7
Écrire une fonction `est_antisymetrique` qui prend en paramètre `oracle` une `list` composée de n `lists` de booléens de taille n et qui renvoie `True` si `oracle` est bien antisymétrique et `False` sinon.

Solution

```python
def est_antisymetrique(oracle):
    """
    list[list[int]] -> bool
    Entrée : Une liste de listes oracle
    Sortie : Un booléen vérifiant si oracle est bien un oracle
    """
    n = len(oracle)
    for i in range(n):
        for j in range(i):
            if oracle[i][j] != (not oracle[j][i]):
                return False
    return True
```

Question 8
Écrire une fonction `vainqueur` qui prend en paramètre un tournoi `L`, un entier n donnant le nombre de joueurs ainsi qu'un oracle `oracle` et qui renvoie le numéro du joueur vainqueur du tournoi.
La fonction complétera une `list` `vainqueurs` de taille $2n-1$ telle que la case `vainqueurs[k]` soit égale au vainqueur du match `k`. On note que pour $0 \leqslant$ `k` $\leqslant n-1$, `vainqueurs[k]` vaut `k`. En effet, si $0 \leqslant$ `k` $\leqslant n-1$, `k` ne désigne pas un match, mais uniquement un numéro de joueur et ce joueur est directement qualifié pour le match suivant.

Solution

```python
def vainqueur(L, n, oracle):
    """
    list[int] * int * list[list[int]] -> int
    Entrée : Une liste de tournoi L , un entier n et un oracle
    Sortie : L'indice du vainqueur du match L[n]
    """
    vainqueurs = [-1] * (2 * n - 1)
    for k in range(n):
        vainqueurs[k] = k
    for k in range(n, 2*n-1):
        k1 = L[k][0]
        k2 = L[k][1]
        i = vainqueurs[k1]
        j = vainqueurs[k2]
        if oracle[i][j]:
            vainqueurs[k] = i
        else:
            vainqueurs[k] = j
    return vainqueurs[2*n-2]
```

Partie 2 : Analyse des performances

Un responsable d'un club souhaite analyser la performance des joueurs au cours des différentes saisons. Il dispose d'un fichier `scores.csv` contenant les scores des joueurs pour la saison.
Le format CSV (pour Comma Separated Values, soit en français valeurs séparées par des virgules) est un format texte représentant des données tabulaires sous forme de valeurs séparées par des points virgules.

Le fichier `scores.csv` contient les colonnes suivantes :

- `Nom` : le nom du joueur ;
- `Saison_1, Saison_2, ... , Saison_n` : les scores obtenus par le joueur au cours des n dernières saisons.

Par exemple, avec le fichier de la figure 15, on sait qu'Alfred a obtenu le score de 130 pour la saison 1, 50 pour la saison 2, ...

```
Nom;Saison_1;Saison_2;Saison_3;Saison_4;Saison_5
Alfred;130;50;20;100;60
Erwan;10;20;10;20;0
Carole;70;110;10;60;50
Diego;10;30;30;130;70
Bernadette;80;110;70;70;40
Farid;20;80;30;40;0
```

FIGURE 15 – Exemple d'un fichier `scores.csv`.

Afin de traiter les données du fichier `scores.csv`, on donne le code suivant.

```python
import csv

```

```python
def lire_scores(fichier):
  scores = []
  with open(fichier, newline='') as csvfile:
    reader = csv.reader(csvfile, delimiter = ';')
    next(reader) # Ignorer l'en-tête
    for ligne in reader:
      nom = ligne[0]
      score_manches = [int(ligne[1]), int(ligne[2]), int(ligne[3]),
                       int(ligne[4]), int(ligne[5])]
      scores.append((nom, score_manches))
  return scores

scores = lire_scores("scores.csv")
```

Remarques :

- La méthode `append()` via `liste.append(element)` ajoute l'élément `element` à la fin de la liste `liste`.
- Dans ce code on a choisi de traiter le cas avec $n = 5$ saisons dans un souci de lisibilité, mais pour la suite on suppose que la fonction gère le cas général avec n saisons.

Avec le fichier `scores.csv` de la figure 15, la variable `scores` a alors la valeur suivante.

```python
[ ('Alfred', [130, 50, 20, 100, 60]), ('Erwan', [10, 20, 10, 20, 0]),
  ('Carole', [70, 110, 10, 60, 50]), ('Diego', [10, 30, 30, 130, 70]),
  ('Bernadette', [80, 110, 70, 70, 40]), ('Farid', [20, 80, 30, 40, 0])]
```

Question 9

a) Donner le type Python de la variable `scores` (on précisera tous les types imbriqués).

b) Que vaut `scores[2][1][0]` ?

c) Quelle instruction permet de récupérer le score de Farid à la saison 4 ?

Solution

a) Au sein de la fonction, la variable locale `scores` est définie comme une `list`. Chaque élément de la `list` est un tuple constitué d'une chaine de caractères (le nom du joueur) et d'une `list` d'entiers (les scores des différentes saisons).

b) `scores[2][1][0]` vaut 70.

c) L'instruction `scores[5][1][3]` permet de récupérer le score de Farid à la saison 4.

Question 10

Écrire une fonction `moyenne` qui prend en paramètre une `list` d'entiers `L` et qui renvoie un flottant représentant la moyenne des entiers de `L`. On supposera que `L` contient au moins une valeur.

Solution

```python
def moyenne(L):
    """
    list[int] -> float
    Précondition : len(L) > 0
    Entrée : Une liste d'entiers
```

```python
    Sortie : La moyenne des éléments de L
    """
    s = 0
    for x in L:
        s = s + x
    return s / len(L)
```

À partir de cette question, on rappelle qu'il y a n saisons.

Question 11

En déduire une fonction `champion_du_club` qui prend en paramètre une variable `scores` récupérée par un appel à la fonction `lire_scores` et qui renvoie le nom du joueur ayant le meilleur score moyen sur l'ensemble des saisons. On supposera que chaque joueur possède une moyenne distincte.

Solution

```python
def champion_du_club(scores):
    """
    list[(str, list[int])] -> str
    Entrée : Une liste de type score
    Sortie : Le nom du champion ayant la meilleure moyenne.
    """
    n = len(scores)
    nom_meilleur = ""
    moy_meilleur = -1
    for i in range(n):
        nom = scores[i][0]
        moy = moyenne(scores[i][1])
        if moy > moy_meilleur:
            moy_meilleur = moy
            nom_meilleur = nom
    return nom_meilleur
```

Question 12

Écrire une fonction `champions_par_saison` qui prend en paramètre une variable `scores` récupérée par un appel à la fonction `lire_scores` ainsi qu'un entier `i` et qui renvoie la `list` du ou des indices des champions de la saison `i + 1`. Par exemple :

- `champions_par_saison(scores, 0)` renvoie `[0]` car Alfred qui est d'indice 0 a le meilleur score de la saison 1 (0 + 1).
- `champions_par_saison(scores, 1)` renvoie `[2, 4]` car Carole (indice 2) et Bernadette (indice 4) ont toutes les deux le meilleur score de la saison 2 (1 + 1).

Solution

```python
def champions_par_saison(scores, i):
    """
    list[(str, list[int])] * int -> str
    Entrée : Une liste de type score et un entier
```

```
 5        Sortie : La liste des indices des champions ayant le meilleur
 6                 score à l'indice i
 7        """
 8        champions = []
 9        score_max = 0
10        for j in range(len(scores)):
11            if scores[j][1][i] == score_max:
12                champions.append(j)
13            elif scores[j][1][i] > score_max:
14                champions = [j]
15                score_max = scores[j][1][i]
16        return champions
```

Question 13

En déduire une fonction `champion_des_champions` qui prend en paramètre une variable `scores` récupérée par un appel à la fonction `lire_scores` et qui renvoie le nom du joueur ayant gagné le plus de saisons. En cas d'égalité, le joueur ayant la meilleure moyenne parmi ceux qui ont gagné le plus de saisons est sélectionné. On supposera encore que chaque joueur possède une moyenne distincte. Dans notre exemple Bernadette est la championne des champions, car elle a gagné deux saisons comme Diego, mais elle a une meilleure moyenne que lui.

Solution

```
 1    def champion_des_champions(scores):
 2        """
 3        list[(str, list[int])] * int -> str
 4        Entrée : Une liste de type score et un entier
 5        Sortie : Le nom du champion ayant gagné le maximum de saison
 6        """
 7        champions = [0] * len(scores)
 8        nombre_de_saisons = len(scores[0][1])
 9        for i in range(nombre_de_saisons):
10            champions_saisons = champions_par_saison(scores, i)
11            for champ in champions_saisons:
12                champions[champ] += 1
13        le_champion = -1
14        max_par_saison = -1
15        for i in range(len(champions)):
16            if champions[i] == max_par_saison and\
17               moyenne(scores[i][1]) > moyenne(scores[le_champion][1]):
18              le_champion = i
19            elif champions[i] > max_par_saison:
20                le_champion = i
21                max_par_saison = champions[i]
22        return scores[le_champion][0]
```

Partie 3 : Création d'équipes

Un club de programmation compétitive de p adhérents souhaite organiser un tournoi interne. Pour accroître la cohésion entre les adhérents et limiter le nombre de rencontres au cours du tournoi, la responsable du club décide de répartir les p adhérents en plusieurs équipes en respectant les contraintes suivantes :

- chaque adhérent fait partie d'une et une seule équipe ;
- toutes les équipes ont le même nombre de joueurs à un près ;
- il y a au moins deux équipes ;
- chaque équipe a au moins q joueurs et au plus k joueurs avec $2 \leqslant q < k < p$. De plus, afin de former au moins deux équipes, on suppose que q est inférieur ou égal au quotient de la division euclidienne de p par 2 (ce quotient est obtenu en Python via `p//2`).

On considère une fonction `comp_equipes` qui prend en paramètres trois entiers p, q et k vérifiant les contraintes ci-dessus et qui renvoie, dans une `list`, toutes les compositions d'équipes possibles. Chaque composition possible est donnée sous forme d'une `list` dont les éléments sont les nombres de joueurs de chaque équipe.

Par exemple l'appel `comp_equipes(8, 2, 5)` renvoie la `list` suivante.
`[[2, 2, 2, 2], [2, 3, 3], [4, 4]]`

Autre exemple, l'appel `comp_equipes(15, 2, 5)` renvoie la `list` suivante.
`[[2, 2, 2, 2, 2, 2, 3], [2, 2, 2, 3, 3, 3], [3, 3, 3, 3, 3], [3, 4, 4, 4], [5, 5, 5]]`

Question 14

Donner la `list` que renvoie l'appel `comp_equipes(17, 3, 6)`.

Solution

```
>>> comp_equipes(17, 3, 6)
[[3, 3, 3, 4, 4], [4, 4, 4, 5], [5, 6, 6]]
```

Question 15

Écrire la fonction `comp_equipes`.

Solution

```python
def comp_equipes(p, q, k):
    """
    int * int * int -> list[int]
    Renvoie toutes les compositions d'équipes possibles de taille
    comprise entre q et k et dont le nombre de joueurs total est p.
    """
    liste_equipes = []
    # Les longueurs sont comprises entre la partie entière supérieure
    # de p / k et la partie entière inférieure de p / q
    for long in range((p + k - 1) // k, p // q + 1):
        # liste_equipes contient toutes les combinaisons possibles
        # de tailles croissantes comprises entre la partie entière
        # supérieure de p / k et long - 1

        # On ajoute d'abord les équipes qui ont p // long joueurs
```

```
        comp_equipe = [p // long] * (long - p % long)
        # Puis les équipes qui ont un joueur de plus
        comp_equipe += [p // long + 1] * (p % long)
        liste_equipes.append(comp_equipe)
    # Remarque : La liste est renversée pour faire comme dans l'exemple.
    return list(reversed(liste_equipes))

```