CM 5 : Récursivité (1)

Info1.Algo1

2022-2023 Semestre Impair

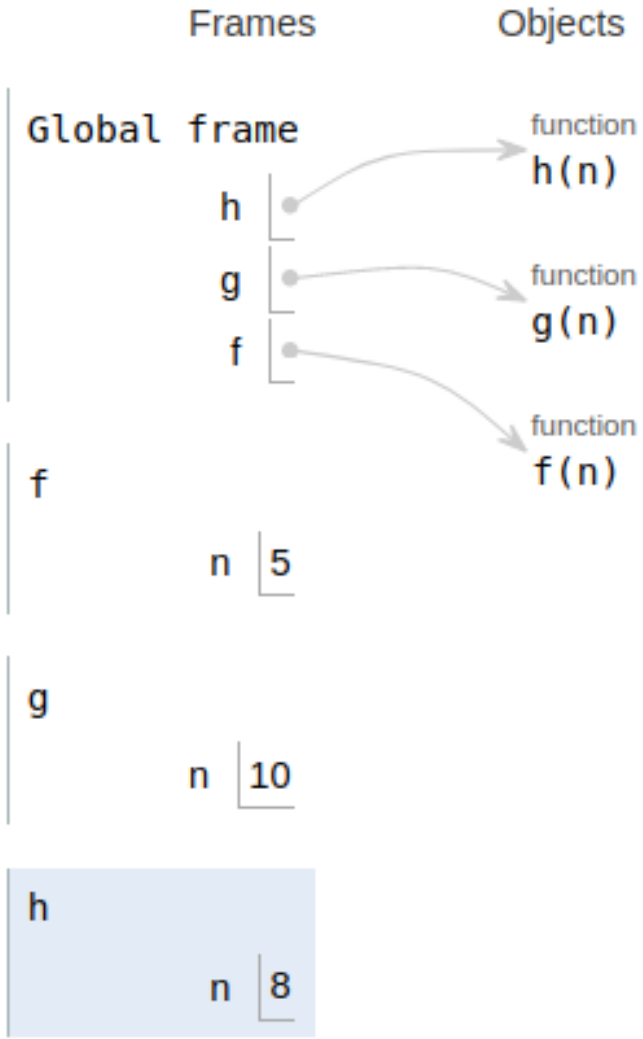
Plan

Première approche

Rappel : Que se passe-t-il quand une fonction en appelle une autre?

```
1  def h(n):
2    print(n)
3
4  def g(n):
5    h(n-2)
6
7  def f(n):
8    g(2*n)
9
10 f(5)
```

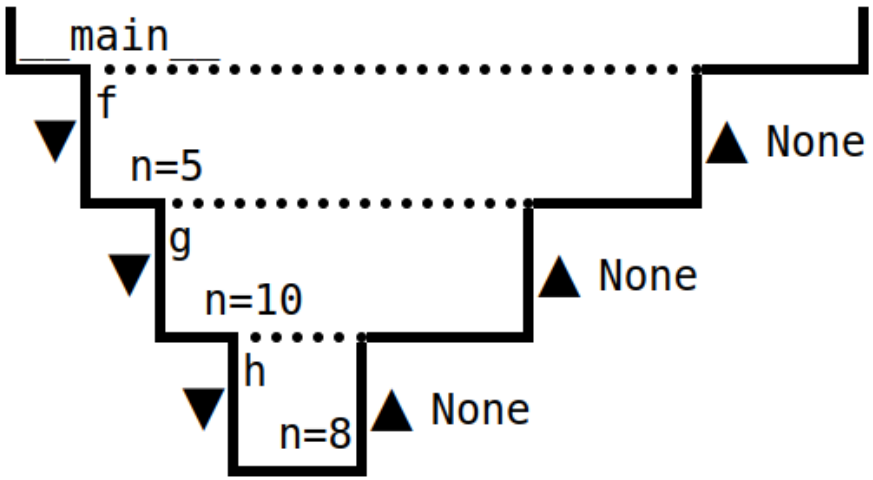
Visualiser avec PythonTutor



Première approche

Schéma temps/mémoire :

- temps : de gauche à droite
- mémoire : de haut en bas



Observations

- **Appel de fonction :** création d'un **nouveau contexte** d'exécution de cette fonction.
- **Une seul contexte actif** à un moment donné.
- **Variables locales :** valeur spécifique à chaque contexte (même si nom identique).
- **Retour de la fonction :** le contexte d'exécution disparaît.

Première approche

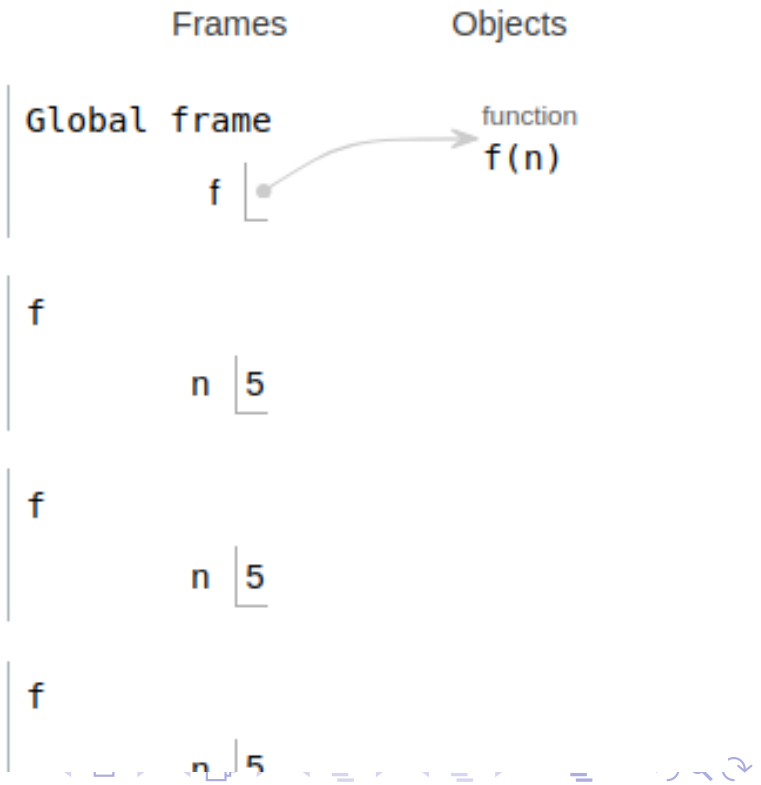
Que se passe-t-il quand une fonction s'appelle elle-même?

Définition

On appelle **fonction récursive** une fonction qui s'appelle *(directement ou indirectement)* elle-même.

```
1 def f(n):
2   f(n)
3
4 f(5)
```

Visualiser avec PythonTutor

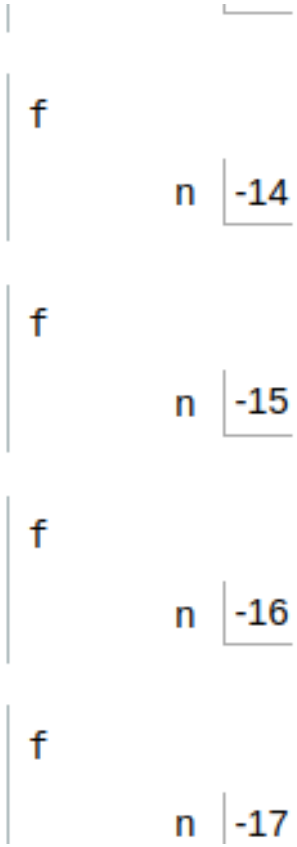


Première approche

Observation : On doit faire varier le paramètre au fur et à mesure des appels...

```
1 def f(n):
2   f(n-1)
3
4 f(5)
```

Visualiser avec PythonTutor

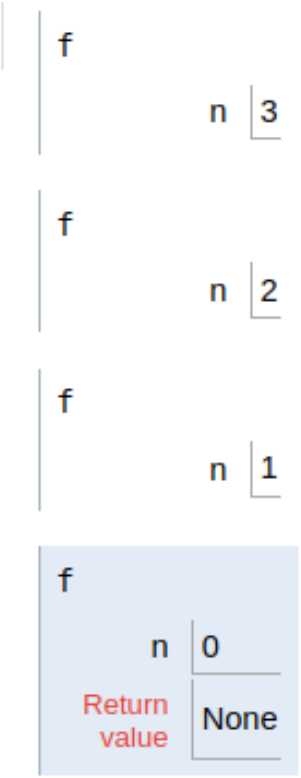


Première approche

Observation : Dans un cas au moins, la fonction ne doit pas s'appeler elle même...

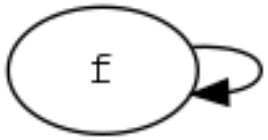
```
1 def f(n):
2   if n==0:
3     print('Fini!')
4   else:
5     f(n-1)
6
7 f(5)
```

Visualiser avec PythonTutor



Premières définitions

```
1 def f(n):
2   if n==0: # Cas d'arret
3     print('Fini!')
4   else: # Cas récursif
5     f(n-1) # Diminution n -> n-1
```

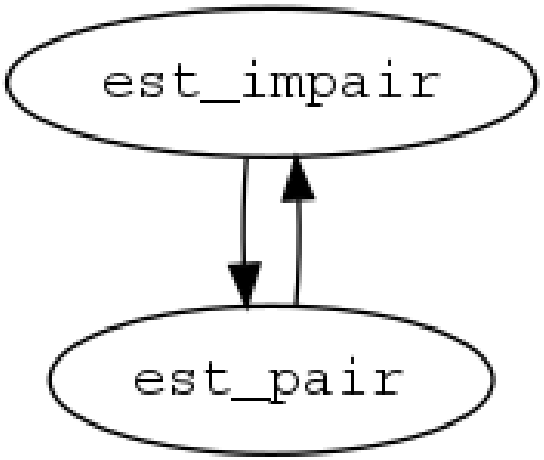


Définitions

- **Cas d'arrêt :** valeur du(des) paramètre(s) de la fonction telle que la fonction ne s'appelle pas elle-même.
- **Cas récursif :** valeur du(des) paramètre(s) de la fonction telle que la fonction s'appelle elle-même.
- **Taille du problème :** paramètre qui décroît lors de l'appel récursif.

Premières définitions

```
1  def est_pair(n):
2      if n==0:
3          return True
4      else :
5          return est_impair(n-1)
6
7  def est_impair(n):
8      if n==0:
9          return False
10     else :
11         return est_pair(n-1)
```



Définition

L'appel récursif peut-être indirect : une fonction en appelle une 2ème, qui en appelle une 3ème, ... , qui appelle la première. On dit alors que ces fonctions sont **mutuellement récursives**.

Motivations

Motivation 1

Certaines **situations que l'on souhaite modéliser** s'expriment naturellement sous forme récursive. La modélisation par une fonction récursive devient une traduction directe de la situation à modéliser.

Exemple : poupées russes (matriochkas)

Une **matriochka** :

- Soit ne s'ouvre pas.
- Soit s'ouvre en deux et contient une autre **matriochka** plus petite.

Motivations

Exemple : entiers naturels

Un **entier naturel** est :

- Soit l'entier 0.
- Soit le successeur $S(n)$ d'un **entier naturel** n.

Cette façon de définir les entiers fait partie des axiomes pour l'arithmétique proposés à la fin du XIXe siècle par Giuseppe Peano.

Motivations

Exemple : généalogie

Une personne A est un **descendant** d'une autre personne B si et seulement si :

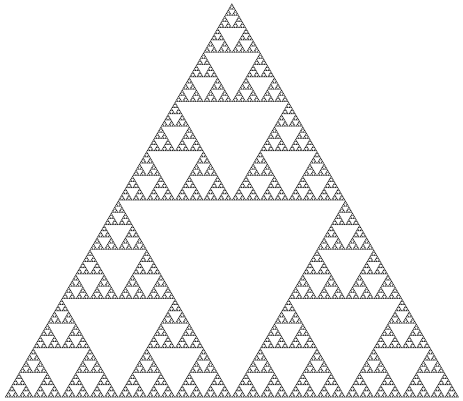
- Soit A est un enfant de B.
- Soit A est un enfant d'un **descendant** de B.

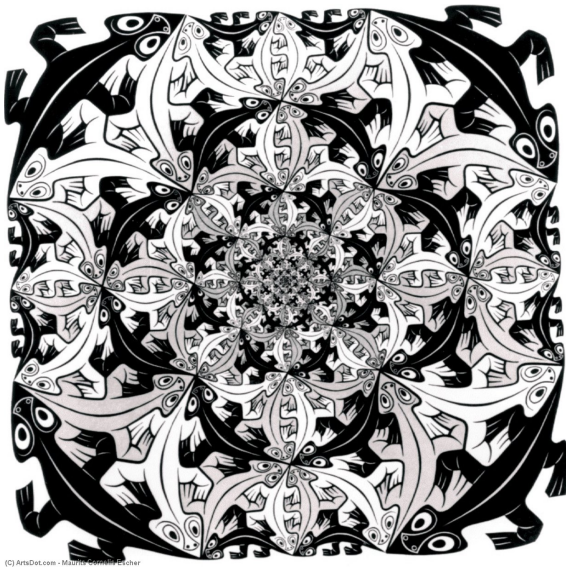
Exemple : dictionnaire

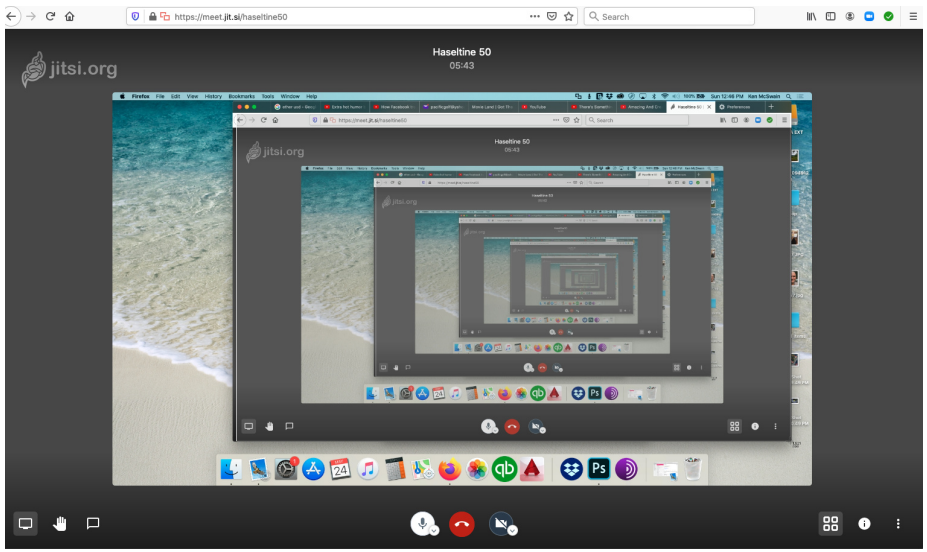
Chaque mot du dictionnaire est défini par d'autres mots eux-mêmes définis par d'autres mots dans ce même dictionnaire...

(exemple de récursivité mutuelle)

Motivations







Motivations

Motivation 2

Certaines **structures de données** informatiques (listes chaînées, arbres, ...) sont définies de manière récursive et il est naturel d'utiliser des fonctions récursives pour les parcourir et y effectuer des traitements.

Motivations

Motivation 3

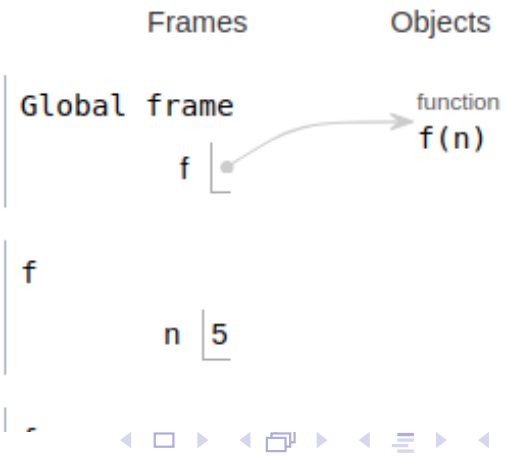
La récursivité constitue une **autre façon d'aborder la notion d'itération** en algorithmique.

```
1 def f(n):
2   if n==0:
3     print('Fini!')
4   else:
5     print(n)
6     f(n-1)
7
8 f(5)
```

Visualiser avec PythonTutor

Print output (drag lower right corner to resize)

```
5
4
3
2
1
Fini!
```



Motivations

Écriture quivalente sous forme de boucle :

```
1 def f(n):
2   while n!=0:
3     print(n)
4     n -= 1
5   print('Fini!')
6
7 f(5)
```

Motivations

Motivation 4

Certains **langages de programmation**, dits fonctionnels (Lisp, OCaml, ...), font largement appel à la récursivité.

Plan

Analyser une fonction récursive

Pour comprendre le fonctionnement d'une fonction récursive :

Méthode 1

Étudier le comportement :

- pour le cas d'arrêt
- pour le cas où on appelle le cas d'arrêt
- pour le cas où on appelle le cas qui appelle le cas d'arrêt
- etc.

Analyser une fonction récursive

Exemple

```
1 def factorielle(n):
2   if n==0:
3     return 1
4   else:
5     fact = factorielle(n-1)
6     return n*fact
```

Que retourne la fonction factorielle lors des appels suivants :

1. `factorielle(0)`
2. `factorielle(1)`
3. `factorielle(2)`
4. `factorielle(3)`
5. `factorielle(4)`

Analyser une fonction récursive

Pour comprendre le fonctionnement d'une fonction récursive :

Méthode 2

Représenter tous les appels sur un **schéma temps/mémoire**, avec à chaque fois :

- le nom de la **fonction**.
- la valeur des **paramètres**.
- la valeur du **retour**.

Axes :

- temps : de gauche à droite
- mémoire : de haut en bas

Conseil : on peut s'aider avec Python Tutor.

Analyser une fonction récursive

Exemple

```
1 def factorielle(n):
2   if n==0:
3     return 1
4   else:
5     fact = factorielle(n-1)
6     return n*fact
7
8 factorielle(4)
```

Faire la représentation temps/mémoire de ce programme

(Visualiser avec PythonTutor)

Écrire une fonction récursive

On souhaite **écrire** une fonction récursive répondant à un problème donné :

Problème posé de manière récursive

Lorsque le problème est **posé de manière récursive**, l'écriture de la fonction récursive permettant de le résoudre consiste en une **traduction directe**.
Cette méthode est applicable entre autre dans le cas de **fonctions mathématiques définies par récurrence** :

- cas d'arrêt $\leftrightarrow$ initialisation
- cas récursif $\leftrightarrow$ hérédité

Écrire une fonction récursive

Exemple

Soit a un entier relatif et n un entier naturel. On définit la puissance a^n par récurrence de la façon suivante :

$$\begin{cases} a^n = 1 & \text{si } n = 0 \\ a^n = a.a^{n-1} & \text{sinon.} \end{cases}$$

Écrire la fonction récursive `puissance` qui accepte en paramètres un entier relatif `a` et un entier naturel `n` et retourne le résultat du calcul de `a**n`.

Écrire une fonction récursive

Pour écrire une fonction récursive répondant à un problème donné :

Méthode générale

- Identifier le **cas d'arrêt**.
- Le cas d'arrêt est l'occasion de s'interroger sur le **type de la valeur retournée**.
- Identifier le **sous-problème récursif** :
  - Quelle est la taille du problème?
  - Comment diminue-t-elle?
- **Écrire l'appel récursif** et récupérer sa valeur de retour.
- Transformer cette valeur avant de la retourner.
- *Eventuellement : raffiner le code pour le simplifier.*

Écrire une fonction récursive

Exemple

Écrire la fonction récursive `somme_chiffres` qui accepte en paramètre un entier naturel `n` et retourne la somme des chiffres *(en base 10)* de `n`.

Appels inutiles

Exemple

On considère le programme suivant :

```
1 def puissance_de_2(n):
2   if n==0:
3     return 1
4   else:
5     return puissance_de_2(n-1)+puissance_de_2(n-1)
6
7 n = int(input())
8 print(puissance_de_2(n))
```

Appels inutiles

1. Déterminer l'affichage de ce programme en fonction de l'entier (positif) entré par l'utilisateur.
2. Réaliser une représentation temps/mémoire de ce programme lorsque l'utilisateur entre `n=3`.
3. Que se passera-t-il si l'utilisateur entre `n=10` ?
4. Quelle amélioration **doit-on** apporter à la fonction `puissance_de_2` ?

Pré-condition

Exemple

On considère la fonctions `factorielle` définie par :

```
1 def  factorielle (n):
2   if n==0:
3     return 1
4   else :
5     return n* factorielle (n-1)
```

1. Vérifier sa pré-condition avec une assertion.
2. Une fois que la pré-condition a été vérifiée, est-il possible de lever une erreur d'assertion lors d'un appel récursif?
3. Renommer la fonction en `factorielle_rec` et réserver la vérification de la pré-condition à une nouvelle fonction `factorielle` appelant la fonction `factorielle_rec`.

Plan

Rappels

Abstraction de données

On appelle **abstraction de données** la définition d'un type de données avec la description des **opérations qu'il est possible de lui appliquer**, indépendamment de la manière dont cela sera programmé.

La **description de chaque opération** précise:

- Ses entrées.
- Les préconditions qui doivent être vérifiées pour que l'opération s'exécute correctement.
- Ses sorties.
- Les post conditions vraies après exécution de l'opération.

Rappels

Structure de données

Une **structure de données** est décrite en présentant :

- L'**abstraction de données** qui la définit.
- Son **implémentation**, c'est-à-dire sa mise en oeuvre matérielle.

Rappels

Exemple : Tableau

- description du type
  - Le **nombre d'éléments** du tableau est **fixe**.
  - Tous les éléments du tableau sont de **même type**.
  - Chaque élément est repéré par sa position dans le tableau, son **indice**.
- opérations
  - **création** en précisant le type des éléments et la taille du tableau.
  - **accès** à un élément `tableau[i]` en lecture et modification.
- implémentation en python avec les listes
  - interdiction d'utiliser les opérations qui modifient la taille (`del`, `append`, `insert`...)
  - interdiction d'utiliser les opérations qui ne maintiennent pas le type unique

Description du type

Principe

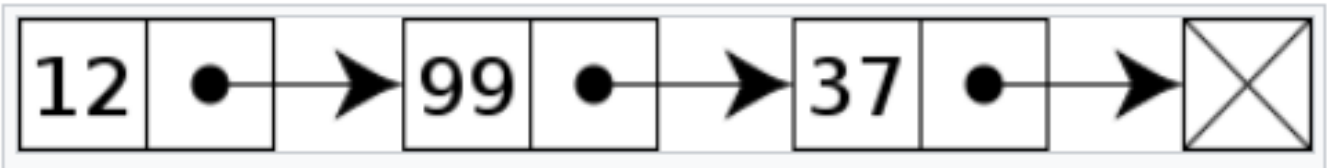
L'abstraction **liste chaînée** que nous allons utiliser dans ce cours permet de représenter une **séquence de taille arbitraire** d'éléments de **même type**, sous la forme d'une **succession de cellules** constituées chacune :

- d'une valeur associée à l'élément (la **tête** de la liste)
- d'un moyen d'accéder à la cellule suivante (la **queue** de la liste)

Description du type

Exemple

La séquence des trois entiers 12, 99 et 37 est représentée par la liste chaînée suivante :



- la **tête** de la liste est la valeur 12.
- la **queue** de la liste est la liste constituée des valeurs 99 et 37.

Remarques

- La queue de la liste est elle-même une liste.
- Toute liste aboutit, à un moment donné, à la liste vide (représentée ici par une croix).

Description du type

Opérations autorisées

- Opérations de création
  - Créer une liste vide
  - Créer une liste à partir d'une tête et d'une queue déjà existante
- Opérations d'accès
  - Accéder à la tête de la liste
  - Accéder à la queue de la liste
  - Déterminer si la liste est vide

Les insertions et suppressions en milieu, en fin ne sont pas autorisées.
Ces opérations plus évoluées devront être écrites dans des fonctions appropriées.

Description du type

Spécification des fonctions d'interface

Les opérations autorisées seront accessibles via les seules fonctions suivantes :

- Opérations de création
  - `creer_liste_vide()` : retourne une liste vide.
  - `creer_liste(t,q)` : retourne la liste constituée de la tête `t` et de la tête `q` passées en paramètres.
- Opérations d'accès
  - `tete(liste)` : retourne la tête de la liste passée en paramètre.
  - `queue(liste)` : retourne la queue de la liste passée en paramètre.
  - `est_vide(liste)` : retourne `True` si la liste passée en paramètre est vide, `False` sinon.

Description du type

Détails de l'implémentation

L'implémentation particulère choisie ici est la suivante :

- Une liste vide est représentée par `None`.
- Une liste non vide est un tuple `(t,q)`.

```
1  def creer_liste_vide():
2      return None
3
4  def creer_liste(t,q):
5      return t,q
6
7  def tete( liste ):
8      return liste[0]
9
10 def queue( liste ):
11     return liste[1]
12
13 def est_vide( liste ):
14     return liste ==None
```

Description du type

Exemple 1

On donne le programme suivant :

```
1 liste0  = creer_liste_vide()
2 liste1  = creer_liste(13, liste0 )
3 liste2  = creer_liste(21, liste1 )
4 liste3  = creer_liste(8, liste2 )
```

1. Représenter par une succession de cellule la liste chaînée `liste3`.
2. Écrire son implémentation sous-jacente sous forme de tuples.
3. Déterminer la valeur des expressions suivantes :
   - `est_vide(queue(liste1))`
   - `est_vide(queue(liste2))`
   - `tete(queue(queue(liste3)))`

Bilan

L'implémentation qui vient d'être présentée résulte d'un **choix** spécifique :

- Les listes sont **non-mutables** : les éléments d'un tuple ne peuvent pas être modifiés. On doit constituer une nouvelle liste dès que l'on souhaite "modifier" une valeur de la séquence.
- Ce choix (inspiré des langages fonctionnels) permet toutefois d'éviter de nombreux soucis liés aux **effets de bords**.

Autres implémentations

D'autres implémentations possible :

- **Python** : avec le type `deque`, le type `list` natif, une classe...
- **C** : avec un `struct`, un tableau de taille très supérieure pour décaler les éléments ajoutés...
- **Caml** : avec les listes natives `(h::t)`

Écriture de fonctions

Une structure récursive

La liste chaînée est une **structure de données récursive**. En effet, on peut dire d'une liste qu'elle est :

- soit vide
- soit construite à partir d'un élément (la tête) et d'une **autre liste** (la queue).

Remarque : d'autres structures de données récursives (arbres, ...) seront vues ultérieurement.

Écriture de fonctions

Il est naturel (mais pas obligatoire) de manipuler les listes chaînées à l'aide de **fonctions récursives**.

Rappel

Lors de l'écriture d'une fonction récursive il est nécessaire :

- D'écrire un ou plusieurs **cas d'arrêt**.
- De **diminuer la taille du problème** lors de l'**appel récursif**.
- D'éviter les doubles appels récursifs.

Écriture de fonctions

Méthodologie

- **Cas d'arrêt :** "doit-on toujours parcourir la liste en entier?"
  - **Oui** (somme des éléments, maximum, etc.) : il y a alors un seul cas d'arrêt (typiquement : liste vide ou réduite à un élément)
  - **Non** (rechercher un élément, vérifier une condition, etc.) : On obtient d'autres cas d'arrêt (élément trouvé, condition invalidée...)
- **Cas récursif :** l'appel récursif se fait très souvent (il peut y avoir des exceptions) sur la **queue de la liste passée en paramètre**.

Écriture de fonctions

Exemple 2

Écrire la fonction récursive `longueur` qui retourne le nombre d'éléments d'une liste chaînée.