L'essentiel à savoir en NSI

1. Tableaux et listes

Parcours d'un tableau

Il existe deux façons de parcourir un tableau en Python :

- Par valeur :
  ```
  for elt in tab:
  ```
- Par indice :
  ```
  for i in range(len(tab)):
  ```

Attention : range(8) s'arrête à i = 7 !

Tableaux à deux dimensions

Un tableau 2D est un tableau de tableaux. Exemple :

```
T = [[3, 4, 8, 5],
     [2, 0, 4, 3],
     [1, 4, 6, 4]]
```

Propriétés utiles	Exemples d'accès	Parcours complet d'un tableau 2D :

`len(T) -> nombre de lignes`
`len(T[0])-> nombre de colonnes`
	`T[1][2] = 4`
	`T[0][3] = 5`
		`for i in range(len(T)):`
		`  for j in range(len(T[0])):`
		`  T[i][j] = ...`

2. Récursivité

Un algorithme récursif doit respecter 3 règles :

- Avoir un cas de base (condition d'arrêt).
- S'appeler lui-même.
- Converger vers le cas de base (éviter une infinité d'appels).

Remarque : à chaque appel récursif, le processeur sauvegarde les informations dans la pile d'exécution (adresse de retour, paramètres, variables...). La pile se vide une fois l'exécution terminée.

3. Programmation Orientée Objet (POO)

Le principe de la POO est de regrouper données (attributs) et traitements (méthodes) dans des classes.

- Une classe définit les attributs et les méthodes de ses objets.
- Un objet est une instance d'une classe.
- Le mot-clé self fait référence a l'instance en cours.
- Le constructeur __init__ est appelé a la création d'un objet.

Utilisation

```
# Creer un objet
eleve1 = Eleve('Bob')

# Modifier l'objet
eleve1.ajouter_note(19)
eleve1.get_notes()
```

Définition de la classe

```
class Eleve:
  def __init__(self, nom):
    self.nom = nom
    self.notes = []
    self.moyenne = None

  def get_nom(self):
    return self.nom

  def ajouter_note(self, note):
    self.notes.append(note)

  def calculer_moyenne(self):
    moy = sum(self.notes) # sum à éviter
    self.moyenne = moy / len(self.notes)
```

4. Structures de données séquentielles

Pile (LIFO - Last In, First Out)

La **dernière** donnée insérée est la première a sortir.

Operations de base

- `CREER_PILE() -> crée une pile vide`
- `EMPILER(P, e) -> insère e au sommet`
- `DEPILER(P) -> retire et retourne le sommet`
- `EST_VIDE(P) -> True si la pile est vide`

Attention

Pour conserver l'état initial d'une pile p,

- créer une pile p'
- empiler p dans p'
- puis dépiler p' dans p.

File (FIFO - First In, First Out)

La **première** donnée insérée est la première a sortir.

- `CREER_FILE() -> crée une file vide`
- `ENFILER(f, élément) -> ajoute élément en queue`
- `DEFILER(f) -> retire et retourne la tête`
- `EST_VIDE(f) -> True si la file est vide`

Listes chaînées

Une liste simplement chainée est composée de maillons, chacun contenant une valeur et un pointeur vers le maillon suivant.

5. Structures hiérarchiques : les Arbres

Un arbre est constitué de nœuds relies par des arêtes selon une relation père-fils.

- Profondeur d'un nœud : distance (en arêtes) depuis la racine.
- Hauteur d'un arbre : profondeur maximale d'une feuille.
- Taille : nombre total de nœuds.
- Arité d'un nœud : nombre de fils.

Remarque : cette structure est récursive — chaque nœud est lui-même racine d'un sous-arbre.

Arbres binaires

Chaque nœud possède au maximum deux fils (gauche et droit).

Encadrement de la taille n pour une hauteur h : $h \leq n \leq 2^h - 1$

Implémentations d'un arbre binaire

- Par tableau de tableaux : [racine, arbre_gauche, arbre_droit]

```
arbre = ['A', ['B', ['D', [], []], ['E', [], []]], ['C', ['F', [], []], []]]
```

- Par une classe Python (avec attributs racine, gauche, droit).

Parcours d'un arbre binaire

Type de parcours

`Préfixe : Racine - Gauche - Droite`
`Infixe : Gauche - Racine - Droite`
`Postfixe : Gauche - Droite - Racine`
`En largeur : niveau par niveau (utilise une file)`

Astuce memoire

R = Racine, G = Gauche, D = Droite
préfixe -> par-gauche
postfixe -> par-droite
infixe -> par-dessous

Arbres Binaires de Recherche (ABR)

Dans un ABR, pour chaque nœud :

- Toutes les valeurs du sous-arbre gauche sont inférieures à la valeur du nœud.
- Toutes les valeurs du sous-arbre droit sont supérieures ou égales a la valeur du nœud.

Le parcours infixe d'un ABR donne les clés dans l'ordre croissant.

Recherche d'une clé

Similaire à la recherche dichotomique.
Complexité : O(log n) si l'arbre est équilibré.

```
fonction recherche(a, cle):
  Si a vide:
    retourner Faux
  e = etiquette de a
  Si e == cle:
    retourner Vrai
  Sinon si e < cle:
    recherche(droite(a), cle)
  Sinon:
    recherche(gauche(a), cle)
```

Insertion d'une clé

On cherche la position par recherche, puis on insère comme fils d'une feuille.

```
fonction insertion(a, clé):
  Si a vide:
    retourner ABR(cle, vide, vide)
  e = etiquette de a
  Si e < cle:
    ABR(e, insertion(gauche(a), clé),
        droite(a))
  Sinon:
    ABR(e, gauche(a),
        insertion(droite(a), cle))
```

6. Structures associatives : les Dictionnaires

Un dictionnaire associe des valeurs a des cles. A partir d'une cle, on accede directement a la valeur associee.

```
D = {cle_1: valeur_1, cle_2: valeur_2, ...}
```

- Les dictionnaires sont mutables : D['cle1'] = nouvelle valeur
- On ne peut pas accéder à leur contenu par indice numérique.

Iterations

```
# Sur les cles
for c in D.keys():

# Sur les valeurs
for v in D.values():

# Sur les paires cle/valeur
for c, v in D.items():

# Test d'appartenance
'a' in D.keys()
7 in D.values()
```

Tableaux de dictionnaires

Tres courant en NSI :

```
Tab = [
  {'a': 1, ...},
  {'b': 6, ...},
  {'c': 4, ...}
]

# Acces :
Tab[2]['c'] = 4
```

7. Structures relationnelles : les Graphes

Un graphe est une structure composée de :

- Sommets (nœuds) : représentent les entités.
- Arêtes (arcs) : représentent les relations entre sommets.

Un graphe peut être :

- Non oriente : les arêtes n'ont pas de direction (relation symétrique).
- Oriente : les arêtes ont une direction (d'un sommet source vers un sommet destination).

Implémentation

Matrice d'adjacence

Matrice 2D ou mat[i][j] indique la presence d'une arete entre i et j. Adapte aux graphes denses.

Liste d'adjacence

Chaque sommet a la liste de ses voisins. Adapte aux graphes creux.

```
graphe = {
  'A': ['B', 'C'],
  'B': ['A', 'C', 'D'],
  'C': ['A', 'D', 'E'],
  'D': ['B', 'C', 'E'],
  'E': ['C', 'D']
}
```

Parcours de graphes

BFS - Parcours en largeur

Explore tous les voisins d'un sommet avant de passer aux suivants.

- Utilise une FILE (FIFO).
- Utile pour trouver le plus court chemin dans un graphe non pondéré.

DFS - Parcours en profondeur

Explore un chemin aussi loin que possible avant de revenir en arrière.

- Utilise une PILE (LIFO).
- Utile pour détecter des cycles ou résoudre des labyrinthes.