**Conception Orientée Objet : COO**

**Classe** : Modèle d’objet regroupant des propriétés(attributs) et des méthodes(comportement).

**Objet** : C’est une entité autonome ayant ses propres propriétés et méthodes. C’est l’instanciation d’une classe.

**Abstraction** : se focaliser sur l’essentiel et ignorer les détails.

**Encapsulation** : Protection des données via une accessibilité (public, privé, protected,..)

**Comprendre l'analyse orientée objet et son processus**

1. Rassembler les **besoins : Fonctionnels et non fonctionnels**
2. **Décrire** l’application
3. **Identifier** les principaux objets
4. Décrire les **interactions**
5. Créer un **diagramme** de classe

**Cas d'utilisation** : Pour décrire un use case il faut réunir :

* Un Objectif : Quel est le but ?
* Un acteur : Qui en a besoin ?
* Le traitement : Comment c’est traité ?

**Scénario**: Un enchainement logique de use case

**Scénario alternatif**: Une déviation par rapport au scénario principal

**Les structures de données**

Une structure de données est seulement un arrangement intentionnel des données. Il s’agit ici donc d’un l'arrangement intentionnel relative à des données conservées en mémoire. C’est-à-dire à l'intérieur d'une application en cours d'exécution

Les collections :

* **Tableaux** : structure de donnée nommée, de taille fixe, contenant des données même type dont chaque valeur est référencée par un index.
  + Jagged array(tableau en escalier) : c’est un tableaux à deux dimension dont la deuxième dimension est variable ou dynamique.
  + Désavantage : Ne convient pas lorsqu’on à bcp d’opérations de trie sur de grandes quantité de donnée car couteux en performance.

**Recherche binaire ou dichotomique (diviser et conquérir) :** on sépare puis on regarde si avant ou après ça correspond, puis on affine la recherche. Ne fonctionne que sur des tableaux triés.

* **Listes** : la grande différence entre un tableau et un Liste c’est la façon dont on accède aux données, notamment l’accès direct ou aléatoire, dans le cas des tableaux. Dans une liste chaque objet est stocké avec la référence de l’objet suivant, donc un accès séquentiel Les listes sont plus souple quand il s’agit d’ajouter de nouveaux objets. On retrouve par ailleurs les liste dites doublement liés où chaque objet contient la référence des d’éléments next et previous. En Python les listes sont des tableaux dynamiques et non pas des listes liées.
* **Les piles** : LIFO, les méthodes en python sont append (x) et pop (), extend(L), insert(i,x).
* **ADT ( Abstact Data Type) :** Tous les types données peuvent être considérés comme étant des ADT selon langage utilisé.
* **Queues : FIFO,** les méthodes en python sont put(), get(). Utilisé surtout dans la communication entre les threads.
* **Priority Queue** : Donc le principe est simple, on a une "Queue", on a des éléments, mais ces éléments ont un ordre de priorité.  Elle est non définie en python.
* **Deque (Double-**Ended-Queue) : c'est une file d'attente, qu'on peut dévider dans les deux sens. On peut donc choisir d’enlever en LIFO ou en FIFO. En Python on la retrouve dans collections.deque
* **Les tableaux associatifs :** c’est un tableau ayant une clé à la place de l’index. Ce qui donne une collection de pair clé-valeur indissociable. Les valeurs peuvent être des objets, tout comme les clés (mais très rare). En python cela correspond aux dictionnaires dict.
* **Hachage** : C'est simplement un algorithme particulier qui permet de générer un résumé d'un objet ou d'une chaîne. Les fonctions de hachage sont non-réversibles 🡺 perte d’information. Il est implémenté en python par la méthode hash()
  + **Regles de hachage**
    - Le hachage est déterministe dans le même contexte : même source🡺 même résultat
    - Deux objets égaux retournent le même hachage
    - Deux objets différents pourraient retourner le même hachage : la notion de collision
* **Set** (ensemble) : C’est une collection de valeurs non triées. Pas d’index, pas de séquence, pas de clé. **Avantage** : Pas de doublons, recherche rapide pour tester la présence ou l’absence d’un élément. Il est implémenté sous forme de tableau de hachage. Il existe sous deux formes en python : set et frozenset
* **Les arbres :** c’est une structure de donnée à plusieurs niveaux(nœuds). Le premier élément applé le root node qui donnera des références sur des nœuds intermédiaires qu’on appelle childs, ainsi de suite. Les nœuds de mêmes niveaux sont les siblings et les derniers nœuds des leaf node. On parle aussi d’arbre binaire dans le sens ou chaque nœud ne peut avoir que 0 ou 2 enfants.
  + **BST(Binary Search Tree) :** Le Principe , Dépuis la racine ( root node) on a deux enfants, un à gauche et l’autre à droite. Pourquoi est-ce qu'on n'en fait que deux, d’ailleurs ? Et bien, parce que chaque fois que je vais accéder à un noeud, ça va être très simple en termes d'algorithmique, je vais regarder quels sont les deux enfants. Je vais avoir deux références sur l'enfant de gauche et l'enfant de droite. Si le noeud n'a pas d'enfant, les deux références seront nulles. La règle qui fait que le BST est trié, c'est que l'enfant de gauche a toujours une valeur qui est en dessous du parent. Moins que le parent. Et l'enfant de droite a toujours une valeur supérieure au parent. Si par exemple, je stocke des entiers. Je vais mettre 50 comme première valeur, et admettons que, tout à coup, je dois stocker 25. Et bien je vais venir sur le parent, je vais regarder si 25 est supérieur ou inférieur au parent, et s'il est inférieur, je le pose à gauche s'il est supérieur, je le pose à droite. Avantages : Insertion rapide et recherche rapide, reste trie même après insertion. Non définie nativement en python
  + **Heaps (tas) :** c’est un arbre binaire équilibré à la différence du BTS. A partir du premier niveau(root node), on va toujours remplir à gauche, puis à droite. Et ensuite, au descend au niveau suivant, et en fait, horizontalement on va toujours remplir, et quand on arrive à la fin, on passe au niveau suivant. Ce qui fait que ça va nous faire un arbre complètement équilibré. On distingue deux type de heap en fonction de la valeur au sommet (root node).
    - **Le min heap :** Un enfant est toujours plus grand que son parent 🡺 le root nod est la plus petite valeur
    - **Le max heap :** Un enfant est toujours plus petit que son parent 🡺 le root node est la plus grande valeur.
    - **Dans les deux types on doit toujours s’assurer que cette règle est respectée à chaque ajout d’une nouvelle valeur :** Donc on ajoute la valeur selon l’ordre d’ajout horizontale, puis on vérifie, si oui on laisse la valeur, sinon on inverse avec le parent. On fait la vérification pour chaque niveau.
    - En python c’est la **heapq** qui implémente **les heaps.**
* **Les graphes** : C’est une collection d’éléments lié entre eux de façon plurilatérale. Donc un graphe c'est un ensemble de sommets, et puis un ensemble d'arêtes, qui vont lier ces différents sommets. Le graphe peut être dirigé ou non dirigé. on peut avoir des graphes qui sont pondérés, ou on dit aussi valués, éventuellement, parce que, entre les différents sommets, et bien on peut donner un poids aux différentes arêtes. En fait, le graphe est un concept assez général, et on peut en faire des implémentations spécifiques. On peut faire des graphes spécifiquement pour un calcul d'itinéraire, et on a vu quelques structures qui sont des graphes simplifiés sur les listes liées, ou sur les arbres.

Python Training

Python est un langage interprété et dynamiquement typé : cela suppose qu’on pas besoins d’indiquer les types données que l’on va utiliser, l’instar des langages comme le C++,Java, c# etc.

Fonctions de base :

* Print() : afficher à l’écran
* Input() : Lire au clavier
* Types() : retourn le type de donnée
* Str.format() : formater une chaine

Une fonction lambda est fonction qui contient une seul instruction :

variable = lambda parametre1,2 : expresssion

Pour importer une bibliotheque en python :

* Import <non\_module>
* Form <non\_module> import <non\_module> : si on ne veut pas point- preceder le nom du module de la fonction
* Form <non\_module> import\* : pour tous les modules

La gestion des erreurs : try /except(+ else, finaly) . le mot clé raise permet de lever une exception donnée. On peut aussi utiliser le assert pour verifier d’une condition donnée est valide.