Efficacité des types et structures de données
Efficacité des opérations : *list*
Deque : listes pour *appendleft* efficace
Temps amorti constant
Cas en moyenne vs pire cas
Ajouter des types de données
Relation entre types et structure de données
Structures de données récursives
Structure et comportement
Interaction entre types de données et algorithmes
Conclusion du module de Mise à niveaux

Efficacité des types et structures de données

Lors de la conception de types de données, il existe un certain nombre de principes de conception à suivre pour s'assurer que le code est aussi efficace que possible.

La complexité en temps des opérations détermine l'efficacité de vos programmes : vous devez connaître l'efficacité de ce que vous utilisez dans vos codes. Cette information est extrêmement bien documentée en Python.

Efficacité des opérations

La fonction *len* dans tous les cas (*list*, *tuple*, *set*, *dict*) est dans O(1) en temps et c'est vraiment sympa !

Pourquoi devrions-nous compter tous les éléments d'une structure chaque fois que nous voulons connaître sa longueur ?

Pour *list*, on a :

insert O(n)
append O(1)

Nous verrons que c'est un temps amorti parce que, lorsque nécessaire, l'espace alloué à la list doit être augmenté, ce qui nécessite du temps supplémentaire.

remove O(n)
contains O(n)

Deque

Si vous devez souvent insérer au début de la liste, vous pouvez considérer le conteneur *Deque* qui fait partie des collections de Python (https://docs.python.org/3/library/collections.html#collections.deque). C'est un conteneur semblable à la *list* avec des opérations append et pop rapides.

```
>>> from collections import deque
>>> d = deque('ghi')              # nouvelle deque avec 3 éléments
>>> for elem in d:                # itération sur les éléments
...       print( elem.upper() )
G
H
I
>>> d.append('j')                 # ajout à droite
>>> d.appendleft('f')             # ajout à gauche
>>> d                             # imprime la deque
deque(['f', 'g', 'h', 'i', 'j'])
>>> d.pop()                      # accès et retrait de l'élément à droite
'j'
>>> d.popleft()                   # accès et retrait de l'élément à gauche
'f'
```

Temps amorti constant

Qu'est-ce que l'implémentateur du type *Deque* a fait pour obtenir une telle amélioration sur l'opération *appendleft* nous concerne, nous les futurs constructeurs de types et structure de données !

Le temps constant amorti dans O(1) représente la performance d'une méthode et signifie que si le nombre d'invocation de la méthode est suffisamment grand, le temps moyen de sa performance est constant.

Une opération peut nécessiter un temps dans O(n) dans le pire cas, par exemple pour allouer de la mémoire supplémentaire à une *list*.

Cas en moyenne vs pire cas

La plupart du temps O(1), mais en pire cas peut monter jusqu'à O(n).

	add	remove	contains
dict	O(1)→O(n)	O(1)→O(n)	O(1)→O(n)
set	O(1)→O(n)	O(1)→O(n)	O(1)→O(n)
Deque	O(1)	pop : O(1) remove : O(n)	O(n)

Ajouter des types de données à nos codes

Jusqu'ici, nous avons vu certains des types de données en Python et l'efficacité de leurs opérations principales. Nous avons aussi assez d'expérience avec l'analyse mathématique et la notion asymptotique pour pouvoir expliquer quelles opérations sont efficaces (ou non) et quantifier à quel point elles le sont.

Python est extensible en permettant la définition de nouvelles classes

- On peut définir de nouveaux types à volonté
- On peut utiliser la nature orientée objet du langage pour construire des types de données abstraits (ADT)

Les nouvelles classes représentent de nouveaux types

- Les méthodes sont des fonctions associées à la classe
- Les attributs sont des données associées à la classe

Ils sont faciles à définir ! Maintenant, votre objectif est de savoir comment structurer des données de sorte que l'efficacité des opérations les manipulant soit ce que vous voulez qu'elle soit !

Relation entre ADT et structure de données

Structure de données

- Organisez les données efficacement en utilisant les types existants (par exemple, vous pouvez utiliser la *list* comme vous le souhaitez, vous pouvez utiliser la *list* pour implémenter tous les types de données: pile, file d'attente, etc (et nous le ferons dans le cours).
- Il faut cependant s'assurer que votre ADT fournit une utilisation des plus efficaces de sa structure de données.
- Comment utiliser la bonne abstraction ? Par exemple, vous pouvez utiliser des *str* pour garder toutes vos informations. Vous pourriez aussi définir un type plus riche pour mieux organiser leurs nombreux éléments, par exemple un code de pays, un code régional, un numéro et peut-être une extension dans des numéros de téléphone.

ADT

- Approche de modélisation pour définir un nouveau type de données par son comportement du point de vue des utilisateurs
- Analyse des propriétés de ses opérations telle que les complexités en espace et en temps.

Structures de données récursives

Principes de conception simples :

Améliorer les performances en augmentant le stockage supplémentaire

- Un extra dans O(1) est généralement une bonne idée
- Pour un stockage supplémentaire dans O(n), ça doit être convaincant !

Structures de données récursives :

Vous pourriez vous demander pourquoi mon code est si lent ? Serait-ce votre compétence de codage ? En fait, cela est souvent lié à un mauvais choix dans la structuration des données. Or, l'un des objectifs de ce cours est de vous assurer que vous êtes au courant de toutes les façons mêmes très complexes dont vous pourriez structurer les données pour rendre vos codes plus efficaces, beaucoup plus efficace !

Il est souvent facile d'écrire du code en utilisant des *list* mais la performance peut en souffrir.

Structure de données récursives

Au cœur de cette optimisation se trouve la notion de structures arborescentes. Les arbres binaires, par exemple, que vous avez déjà vus. Cela requiert cependant un développement de code plus complexe. Cet ajout de complexité est nécessaire et en vaut la peine, car ils sont beaucoup plus efficaces et rendent plus efficaces toutes les autre parties de votre code qui dépendent de l'accès aux données.

L'une des raisons pour lesquelles ils sont plus efficaces provient du principe de diviser-pour-régner (à approfondir en IFT2125), dont la récursivité en est inhérente. Par exemple, lorsque vous effectuez une recherche dans un dictionnaire ou un annuaire téléphonique, vous ne cherchez généralement pas une page à la fois ! Vous divisez le livre en parties plus faciles à explorer qui vous rapproche peu à peu de ce que vous cherchez.

Structure et comportement

Un dernier concept de génie logiciel consiste à vous assurer de bien séparer la structure et le comportement de vos types de données.

La raison principale est que la structure peut changer alors que le comportement, que vous avez déclarer aux programmeurs d'applications, doit rester la même. Le code des programmeurs d'applications dépend du comportement. Si tout à coup ils se plaignent de la lenteur de votre type de données, en ayant bien séparé la structure et le comportement, vous pouvez retravailler la structure sans affecter le comportement et, du coup, leur code.

Vous pouvez utiliser des *list* et des *dict* à l'interne de votre structure, mais assurez-vous d'entourer (wrapper) les opérations dans un comportement déclaré qui n'affectera pas les autres codes lorsque vous déciderez de les changer.

Ce sont des principes qui consistent à cacher les représentations internes de vos codes (information hiding).

Interaction entre type de données et algorithmes

Les nouveaux types de données sont conçus pour un usage spécifique. Le plus souvent pour des algorithmes spécifiques.

Ce cours ne concerne pas les algorithmes !

L'algorithme est un ensemble d'opérations, étape par étape, pour résoudre un problème (IFT2125). Certains algorithmes efficaces deviennent possibles uniquement si un certain type de données est disponible. Nous verrons des exemples pendant le cours.

Conclusions du module

- Nous avons vu juste assez du langage Python pour être sur le même niveau et poursuivre son apprentissage au fur et à mesure des besoins du cours
- Nous avons vu un sous-ensemble des types de données de Python pour pouvoir commencer à creuser et à absorber plus de détails à leur sujet
- Nous avons appris la notion asymptotique pour pouvoir apprécier les efficacités des types de données de Python et pour éventuellement documenter les types que nous créerons
- Nous allons donc par la suite sauter dans les merveilles du monde des types et structures de données
- Nous verrons de nouveaux concepts au besoin
- Nous utiliserons les principes orientés objet