Structures de données - introduction

BARBIER J.M. - LGT Dumezil - NSI

24 novembre 2025

Table des matières

1 Exemple : calculatrice RPN

1.1 Principe

1.1.1 Calculatrice RPN

RPN : Reverse Polish Notation (notation polonaise inversée)…

C'est un type de calculatrice où les opérations sont faites après avoir entré les nombres concernés. Même si ce n'est pas très courant aujourd'hui, ce type de calculatrice a été très populaire dans les années 80 et 90, notamment avec les calculatrices HP (Hewlett-Packard), et permettaient (une fois qu'on avait pris le coup de main) de faire des calculs plus rapidement qu'avec une calculatrice classique.

Les nombres rentrés par l'utilisateur sont stockés dans une pile (empilés les uns sur les autres), et lorsqu'on fait une opération, on reprend de la pile les éléments nécessaires à l'opération, et on remet ensuite le résultat dans la pile.

http://www.poleyland.com/hp48/

Exemple de calcul : pour faire $(3 + 4) * 5$, on entre successivement : 3, ENTER, 4, (ENTER facultatif), +, 5, (ENTER facultatif), $\star$

Exercice : calculer en RPN les expressions suivantes (et vérifier avec une calculatrice classique) :

- $(5 + 2) * (8 - 3)$
- $\frac{(3+5)^2}{4}$
- $\cos(\frac{3}{4}) + \sin(\frac{1}{4})$

1.1.2 Opérations

Pour réaliser ce genre d'outil, on a besoin de pouvoir mettre des chiffres dans une **pile** : le dernier nombre rentré est le premier qu'on peut récupérer : **LIFO : Last In First Out**, comme quand on range les assiettes dans un tiroir.

On doit pouvoir réaliser les opérations suivante sur une pile (**spécification**) :

- créer une pile : fonction `newpile()` ou constructeur si objet `Pile()` : renvoie une pile vide. Utilisation : `pile = newpile()` ou `pile = Pile()`.
- rajouter un élément dans la pile : `empile(pile, elt) -> None` ou `pile.empile(elt) -> None`
- récupérer le dernier élément de la pile (et le supprimer de la pile par la même occasion) : `depile(pile) -> nombre` (erreur si la pile est vide), ou bien `pile.depile() -> nombre` (pour une pile objet)
- renvoyer le nombre d'éléments dans la pile `nbelts(pile) -> nombre`, ou `len(pile) -> nombre` (pour une pile objet)

La manière exacte de réaliser ces opérations est réalisée lors de **l'implémentation**.

2 Implémentation d'une pile

2.1 Tests de la spécification

2.1.1 Tests de la spécification fonctionnelle

```
## Test de l implémentation fonctionnelle
pile = newpile()
empile(pile, 1)
```

```
empile(pile, 2)
assert nbelts(pile) == 2, pile
assert depile(pile) == 2, pile
assert nbelts(pile) == 1, pile
assert depile(pile) == 1, pile
assert nbelts(pile) == 0, pile
try:
    depile(pile)
    print("Une erreur aurait dû avoir lieu")
except AssertionError:
    pass
```

2.1.2 Tests de la spécification objet

2.1.3 Tests

```
## Test de cette implémentation
pile3 = Pile()
pile3.empile(1)
pile3.empile(2)
assert str(pile3) == "[1, 2]"
assert pile3.nbelts() == 2
assert len(pile3) == 2
assert pile3.depile() == 2
assert pile3.nbelts() == 1
assert pile3.depile() == 1
assert pile3.nbelts() == 0

try:
    pile3.depile()
    print("Une erreur aurait dû avoir lieu")
except AssertionError:
    pass
```

2.2 Implémentation 1 : fonctionnelle

2.2.1 Programme

```
def newpile():
    return list()

def empile(pile, elt):
    pile.append(elt)

def depile(pile):
    assert len(pile) > 0, "Pile vide"
    return pile.pop()
```

```
def nbelts(pile):
    return len(pile)
```

2.3 Implémentation 2 : objet

2.3.1 Programme

```
class Pile():
    def __init__(self):
        self.__liste = []
    def empile(self, elt):
        self.__liste.append(elt)
    def depile(self):
        assert len(self.__liste) > 0, "Pile vide"
        return self.__liste.pop()
    def nbelts(self):
        return len(self.__liste)
    def __str__(self):
        return str(self.__liste)
    def __len__(self):
        return len(self.__liste)
```

2.4 Implémentation 3 : liste chaînée

2.4.1 Via une liste chaînée

On verra cette implémentation plus tard, lorsqu'on étudiera les listes chaînées.

3 Spécification d'une structure de données

3.1 A retenir

3.1.1 Spécification

Lorsqu'on veut développer un programme, il faut le plus vite possible *identifier* et *spécifier* les **structures de données** : quelles données doit-on stocker, et comment doit-on pouvoir les manipuler ?

Une **spécification** est une définition formelle du comportement d'une certaine structure de données.

- c'est un contrat entre l'utilisateur et la personne qui va développer (implémenter) la structure de données
- elle dit ce que doit faire la structure de données
- elle ne dit pas *comment* le faire (implémentation)
- elle doit être précise et rigoureuse
- elle doit éviter de poser des contraintes d'implémentation

Une fois la spécification écrite, on peut écrire des tests pour vérifier que l'implémentation respecte bien la spécification, et ensuite on peut implémenter la structure de données (**TDD** : Test Driven Development).

4 File (ou queue)

4.1 Principe

4.1.1 File ou queue (FIFO)

Une file est une structure de données qui permet de stocker des éléments dans un ordre précis : le premier élément entré est le premier élément sorti (FIFO : First In First Out).

C'est le principe de la file d'attente au magasin : la première personne arrivée à la caisse est la première à partir…

4.2 Spécification et implémentation(s)

4.2.1 Spécification

En vous inspirant de la spécification de la pile, écrivez la spécification d'une file.

4.2.2 Tests

En vous inspirant des tests de la pile, écrivez des tests pour vérifier la spécification d'une file.

4.2.3 Implémentation

- en vous inspirant de l'implémentation 1 de la pile, écrivez une implémentation d'une file de manière fonctionnelle (info utile : `pop()` prend aussi un argument `index` pour sortir un élément à un index donné, en particulier 0 pour sortir le premier élément).
- en vous inspirant de l'implémentation 2 de la pile, écrivez une implémentation d'une file de manière objet.

5 Mini projet

5.1 Calculatrice RPN

5.1.1 A faire

Ecrire une fonction `rpn(pile, actions)` prenant comme arguments une pile `pile` et une chaîne de caractères `actions`, qui réalise les opérations contenues dans `action` sur la pile `pile`. La fonction renvoie la pile résultante.

La `pile` n'est accessible que par les fonctions/méthodes définie dans sa spécification, on utilisera la version OBJET de la pile.

```
def rpn(pile: Pile, actions: str) -> Pile:
    # votre code ici
    return pile
```

5.1.2 Exemples

— pile vide, actions = “2 2 + 3 *” : réalise les actions “empiler 2”, “empiler 2”, “depiler deux valeurs, les additionner et empiler le résultat”, “empiler 3”, “dépiler deux valeurs, les multiplier et empiler le résultat”. La pile contient donc à la fin un seul nombre : 12
— pile contenant 2 (bas) et 3 (haut), actions = “/ 1 +” : réalise les action “dépiler deux valeurs, les diviser (3/2) et remettre le résultat dans la pile”, “empiler 1”, “dépiler deux valeurs, les additionner et remettre le résultat dans la pile”. La pile contient donc à la fin un seul nombre : 2,5.

Actions possibles : `+`,`-`,`*`,`/`,`^` ou `**`, `cos`, `sin`, `tan`, `dup` (dupliquer le dernier élément), `swap` (échanger les deux derniers éléments de la pile) et `drop` (supprimer le dernier élément de la pile)

5.1.3 Indications

Commencez par écrire des assertions permettant de vérifier le bon fonctionnement de la calculatrice, du genre

```
# Test basique
pile = Pile()
actions = "2 2 + 3 *"
pile = rpn(pile, actions)
assert pile.nbelts() == 1, f"Pas le bon nombre d'éléments : {pile}"
assert pile.depile() == 12, f"Mauvais résultat : {pile}"
```

Fonctions utiles :

— `actions.split(" ")` permet de couper une chaîne de caractères à chaque espace pour la transformer en liste
— `float(str)` permet de transformer une chaîne de caractères en nombre, en levant une exception `ValueError` si le nombre est invalide
— depuis 2021, le python a une structure de contrôle match/case qui peut remplacer des suites de `elif` (ce qui ne veut pas dire que c’est à utiliser forcément ici) :

```
command = 'Hello, World!'
match command:
case 'Hello, World!':
    print('Hello to you too!')
case 'Goodbye, World!':
    print('See you later')
case other:
    print('No match found')
```