Pô !

Homer Simpson, The Simpson

Diviser chacune des difficultés que j'examinerais, en autant de parcelles qu'il se pourrait, et qu'il serait requis pour les mieux résoudre.

René DESCARTES, Méth. II, 8

2

La Programmation Orientée Objet (POO)

Extrait du programme

Thème : Structures de données

Contenus :

Structures de données, interface et implémentation.

Capacités attendus :

Spécifier une structure de données par son interface.
Distinguer interface et implémentation.
Écrire plusieurs implémentations d'une même structure de données.

Commentaires :

L'abstraction des structures de données est introduite après plusieurs implémentations d'une structure simple comme la file
(avec un tableau ou avec deux piles)

Boule de neige II

Contenus :

Vocabulaire de la programmation objet : classes, attributs, méthodes, objets.

Capacités attendus :

Écrire la définition d'une classe.
Accéder aux attributs et méthodes d'une classe.

Commentaires :

On n'aborde pas ici tous les aspects de la programmation objet comme le polymorphisme et l'héritage.

Thème : Langages et programmation

Contenus :

Modularité.

Capacités attendus :

Utiliser des API (Application Programming Interface) ou des bibliothèques.
Exploiter leur documentation.
Créer des modules simples et les documenter.

Commentaires :

Avertissement ! ! !

Si vous vous souvenez bien de vos cours de première, les ordinateurs gèrent très mal les nombres décimaux (par exemple 0,1) et pas du tout les nombres réels (par exemple π).
Partant de ce constat, le fil rouge de ce cours est de créer un nouveau type qui permettra de gérer les fractions. Les personnes averties savent très bien qu'il existe déjà un module `fractions` qui résout ce problème (voir exemple ci-contre).
Le propos de ce cours **n'est pas** d'utiliser des fractions mais bien de comprendre **comment on peut créer un nouvel objet.**

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(1, 3) + Fraction(1, 2)
Fraction(5, 6)
```

I. Interface et implémentation

Cette partie est très fortement inspirée d'un document proposé par Romain Janvier, professeur NSI à Bourg-Lès-Valence.

a) Exemple d'introduction

Ada et Alan sont deux étudiants travaillant sur un projet qui permettrait de faire des calculs mathématiques très précis. Pour cela, ils veulent passer par des fractions, plutôt que des réels, à cause des erreurs de virgules flottantes. Ils ont besoin de décider comment représenter les fractions.
Ils hésitent entre un couple d'entiers (tuple), un tableau avec deux entiers et un dictionnaire :

Avec un tuple :

```
>>> F = (1, 3)
>>> F[0] #numérateur
1
>>> F[1] #dénominateur
3
```

Avec un tableau :

```
>>> F = [1, 3]
>>> F[0] #numérateur
1
>>> F[1] #dénominateur
3
```

Avec un dictionnaire :

```
>>> F = {"numerateur" : 1,
      "denominateur" : 3)
>>> F["numerateur"]
1
>>> F["denominateur"]
3
```

Le tableau et le dictionnaire sont **mutables** mais pas le tuple. En dehors de cela, il n'y a pas de grosse différence entre le tuple et le tableau.
Le dictionnaire permettrait de rajouter des informations supplémentaires, comme par exemple la valeur approchée.
Les deux étudiants hésitent, alors Ada propose de s'en occuper pendant qu'Alan commencera à implémenter des fonctions pour les opérations de base (addition, soustraction, ...)

Malheureusement Alan a besoin de connaître le numérateur et le dénominateur de chaque fraction s'il veut les additionner. Comment faire s'il ne sait pas si la fraction sera représentée par un couple ou un dictionnaire ?
Ada lui propose alors de faire 3 fonctions permettant de créer une fraction, d'obtenir son numérateur et son dénominateur.

```
>>> F = Fraction(1, 3)
>>> numerateur(F)
1
>>> denominateur(F)
3
```

Ainsi, Alan peut travailler sans que ses fonctions ne dépendent des choix d'Ada. De son côté, elle peut faire une première implémentation très simple, par exemple avec un tuple et si elle décide de passer sur un dictionnaire, elle n'aura que ces 3 fonctions à modifier et cela ne changera rien pour les autres fonctions rajoutées après.

b) Abstraction et interface

Cet exemple montre le principe de **l'abstraction des structures de données** et des **interfaces**. Le module qu'ils vont proposer permettra de manipuler des fractions sans avoir à connaître leur implémentation en interne. C'est l'abstraction des structures de données.
La description des fonctions à la disposition des utilisateurs s'appelle l'**interface**.
L'intérêt pour l'utilisateur, c'est qu'il n'est pas nécessaire de connaître les détails internes, ni le code des fonctions. Lorsque vous utilisez `Image` du module **PIL**, vous n'avez pas besoin de savoir comment elle est représentée en mémoire ou comment est programmée la fonction **getpixel()**. Vous avez juste besoin de savoir qu'elle existe, à quoi elle sert et surtout, comment l'utiliser.

c) Améliorer le module

Pour l'instant l'interface est :

Fonction	Description
`Fraction(n, d)`	Renvoie une fraction correspondant à n/d, avec d non nul.
`numerateur(F)`	Renvoie le numérateur de la fraction F.
`denominateur(F)`	Renvoie le dénominateur de la fraction F.

Alan travaille sur la fonction permettant d'additionner deux fractions :

```
def plus(F1, F2):
    n1 = numerateur(F1)
    d1 = denominateur(F1)
    n2 = numerateur(F2)
    d2 = denominateur(F2)
    return Fraction(n1*d2 + n2*d1, d1*d2)
```

Il est alors confronté à un problème, aucune simplification de la fraction n'est faite. Il aimerait bien que la fonction `Fraction(n, d)` renvoie une fraction sous sa forme réduite et avec le dénominateur positif. Il demande donc à Ada de rajouter une simplification de la fraction $\frac{n}{d}$ dans la fonction `Fraction(n,d)`.

L'interface devient alors :

Fonction	Description
`Fraction(n, d)`	Renvoie une fraction irréductible correspondant à n/d et dont le dénominateur est positif et non nul.
`numerateur(F)`	Renvoie le numérateur de la fraction F.
`denominateur(F)`	Renvoie le dénominateur de la fraction F.
`plus(F1, F2)`	Renvoie une fraction irréductible égale à F1 + F2 et dont le dénominateur est positif.

d) Intérêt de la modularité

Le découpage en plusieurs modules d'un projet permet de se répartir la tâche à plusieurs. Il permet aussi de s'assurer qu'un changement de structure de données sur une des parties ne nécessite pas de modifier un grand nombre de fonctions.
En séparant l'interface de l'implémentation, on augmente la modularité du programme et on facilitera la recherche et la correction de bugs, ainsi que l'ajout de nouvelles fonctionnalités.
Mais cela nécessite aussi de bien définir à l'avance les interfaces des principales fonctions et de bien documenter le code pour savoir clairement à quoi sert chaque fonction, et idéalement, comment elle fonctionne.

Exercice 1
Proposer un interface le plus précis possible pour notre nouveau type `Fraction`.

Fonction	Description
`Fraction(n, d)`	Renvoie une fraction irréductible correspondant à n/d et dont le dénominateur est positif et non nul.
`numerateur(F)`	Renvoie le numérateur de la fraction F.
`denominateur(F)`	Renvoie le dénominateur de la fraction F.
.....	.....
.....	.....
`plus(F1, F2)`	Renvoie une fraction irréductible égale à F1 + F2 et dont le dénominateur est positif.
.....	.....
.....	.....
.....	.....
.....	.....
.....	.....
.....	.....

II. Paradigmes de programmation

Cette partie est très fortement inspirée d'un document proposée par Frédéric Manson, professeur NSI.

a) Différents paradigmes

Les langages de programmation sont nombreux et variés. On peut les regrouper dans plusieurs classes, correspondantes à des schémas de pensée différents : ce sont les **paradigmes de programmation**. Il est d'ailleurs assez usuel maintenant qu'un langage appartienne à plusieurs de ces classes, c'est par exemple le cas de Python (ainsi que de C++, Ruby, OCaml, ...). Certains de ces paradigmes sont mieux adaptés que d'autres pour traiter des problèmes spécifiques. On verra ultérieurement qu'il est possible d'utiliser plusieurs paradigmes à l'intérieur d'un même programme.
Les principaux paradigmes sont :

- impératif (Fortran, COBOL, BASIC, Pascal, C, PHP, ...),
- fonctionnelle (Lisp, OCaml, Haskell, ...)
- événementielle (JavaScript, Processing, Scratch...)
- orientée objet (C++, PHP, Python, ...)

b) Programmation impérative

Le **paradigme impératif** est le paradigme le plus traditionnel. Les premiers programmes ont été conçus sur ce principe :

- Une suite d'instructions qui s'exécutent séquentiellement, les unes après les autres.
- Ces instructions comportent :
  Des affectations
  Des boucles (pour ..., tant que ..., répéter ... jusqu'à ...)
  Des conditions (si ... alors ... sinon)
  Des Branchements/sauts sans condition
- La programmation impérative actuelle limite autant que possible les sauts sans condition. Ce sous-paradigme est appelé programmation structurée. Les sauts sont utilisés en assembleur (instructions BR adr , « branch vers adresse »).
  Un programme utilisant de nombreux sauts est qualifié de « programmation spaghetti » [1], pour la clarté toute relative avec laquelle on peut le dérouler. Certains langages peuvent donner facilement ce style de code (BASIC, FORTRAN, ...)
- L'usage des fonctions, comme on a pu le voir en première, est aussi une variante de la programmation impérative, appelée programmation procédurale. Elle permet de mieux suivre l'exécution d'un programme, de le rendre plus facile à concevoir et à maintenir, et aussi d'utiliser des bibliothèques.

c) Programmation fonctionnelle

Ce type de programmation sera vu en fin d'année. Tout ce que l'on peut dire pour le moment c'est que dans ce paradigme tout est fonction.

d) Programmation objet

Comme son nom l'indique, le paradigme objet donne une vision du problème à résoudre comme un ensemble d'objets. Ces objets sont définies par :

- leurs propriétés ou caractéristiques. On les appelle les **attributs**.
- leurs comportements ou interactions. On les appelle les **méthodes**.

Les objets interagissent entre eux en respectant leur interface.
L'**encapsulation** introduit une nouvelle manière de gérer des données. Il ne s'agit plus de déclarer des données générales puis un ensemble de sous-programmes destinés à les gérer de manière séparée, mais bien de réunir le tout sous le couvert d'**une seule et même entité**.

1. `https://linuxfr.org/news/encore-un-exemple-de-code-spaghetti-toyota`

III. Création de l'objet `Fraction`

Le « moule » avec lequel on va fabriquer un objet est appelé une **classe**.
La classe `Fraction` comprend par exemple les **attributs** (propriétés) :

- `num` pour le numérateur
- `den` pour le dénominateur
- `keskc` pour le type fraction

Elle comprend les **méthodes** (comportements) :

- `numerateur(F)`
- `denominateur(F)`
- `affiche(F)`
- `approche(F)`
- `plus(F1, F2)`
- `moins(F1, F2)`
- `fois(F1, F2)`
- `divisePar(F1, F2)`
- `puissance(F, exp)`
- `plusGrand(F1, F2)`
- `plusPetit(F1, F2)`

Quand on crée un objet `Fraction`, l'ordinateur crée ce que l'on appelle une **instance** de la classe. C'est-à-dire que tous les objets de la classe auront les mêmes **attributs** et **méthodes**. Autrement dit, deux `fractions`, bien que différentes, sont deux **instances** de la même classe et à ce titre auront toutes les deux les attributs `numerateur`, `denominateur` et `keskc` ainsi que les méthodes `affiche()`, `somme()`, ... **encapsuler** dans l'objet lui même.

On utilisera souvent dans la suite le site Python Tutor [2] qui nous permet de visualiser notre code.
Le but est créer notre « moule », la **classe** `Fraction`.
En utilisant ensuite ce moule avec le code

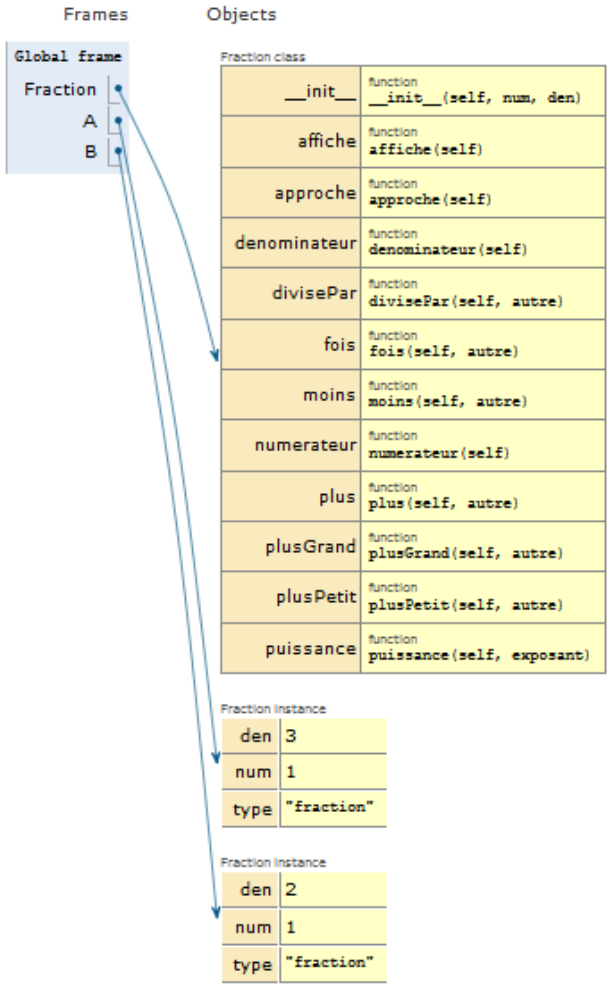
```
A = Fraction(1,3)
```

puis

```
B = Fraction(1,2)
```

on crée deux **instances** `A` et `B` de cette classe.

On constate aussi que tous les attributs et toutes les méthodes sont **encapsuler** dans la classe.



2. `http://pythontutor.com/`

a) Le constructeur

Nous créons notre premier moule, notre **classe** `Fraction`.
Ensuite nous allons définir notre première méthode, qui est une méthode particulière, le **constructeur**. Comme son nom l'indique, le constructeur est la méthode qui va permettre de créer une **instance** de la classe.
En Python, le constructeur se nomme toujours `__init__`.

```
1  class Fraction :
2      '''
3      classe  définit par
4      - numérateur
5      - dénominateur
6      '''
7
8      #constructeur
9      def __init__(self, num, den) :
10         self.num = num
11         self.den = den
```

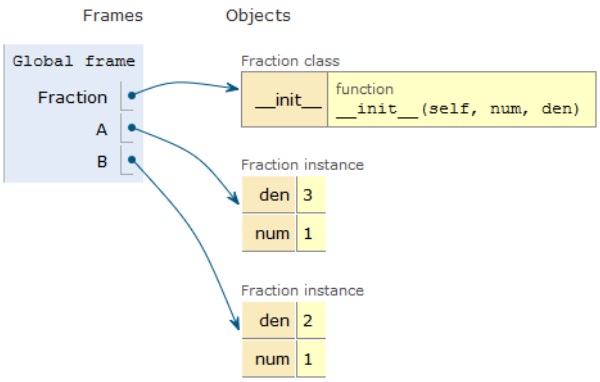
Quelques explications supplémentaires. Le constructeur à toujours besoin, au minimum de l'argument `self`. C'est tout simplement pour dire que le constructeur s'adresse à lui-même. Cela semble évident, mais il faut le préciser.
Ensuite la ligne 10 peut se comprendre ainsi :

$$\underbrace{\texttt{self.num}}_{\text{le numérateur de la fraction créée}} = \underbrace{\texttt{num}}_{\text{la valeur num donnée en argument à la ligne 9}}$$

Une fois le constructeur donné, on peut créer toutes les instances voulues de notre classe :

```
1 >>> A = Fraction(1, 3)
2 >>> B = Fraction(1, 2)
```

Pour y voir un peu plus clair, on peut aussi tester le code sur Python Tutor :
`http://pythontutor.com/`



Nous avons donc réussi à créer un nouvel objet qui nous permet de stocker en même temps deux entiers, le numérateur et le dénominateur. C'est plutôt sympathique, mais c'est encore très limité.
On peut néanmoins avoir facilement accès au numérateur et au dénominateur de chaque fraction en écrivant le nom de la fraction suivi d'un point puis de l'attribut voulu.

```
1 >>> A.num
2 1
3
4 >>> A.den
5 3
```

Alerte ROUGE!!!

Python étant un langage très large d'esprit dans ses pratiques[3], il permet une grande ouverture d'esprit. Malheureusement, il permet aussi d'écrire des choses assez bizarres voire choquantes pour ceux ayant découvert la POO avec un autre langage.
Vous pouvez ainsi modifier, l'air de rien, votre instance comme dans le code ci-contre.
Ce genre de code est totalement impossible à faire dans la plupart des langages orientés objets sans passer par ce qu'on appelle des **accesseurs** et des **mutateurs**.
De manière générale, ne choquez personne et ne faites jamais ça.

```
>>> A.num = 2
#maintenant la fraction A est 2/3

>>> A.truc = "bizarre"
#on a rajouté un attribut chelou à notre fraction

>>> A.num = "deux"
#maintenant la fraction A n'a plus de sens
```

Exercice 2

Améliorer le constructeur de la classe `Fraction` de la manière suivante :

- En utilisant la commande `assert` on va s'assurer que la fraction créée est composée de deux nombres entiers, le dénominateur étant non nul.
- Quelques soient les nombres entiers proposés par l'utilisateur, la fraction stockée ne devra pas avoir de signe négatif au dénominateur. Exemples : $\frac{1}{-2} = \frac{-1}{2}$ ou $\frac{-7}{-4} = \frac{7}{4}$.
- Quelques soient les nombres entiers proposés par l'utilisateur, la fraction stockée sera sous sa forme irréductible.
- On rajoute l'attribut `keskc` qui dans notre cas sera toujours `"fraction"`.

3. Le fondateur de Python, Guido van Rossum, est fréquemment citée pour cette phrase : « We are all consenting adults here ». Elle signifie : « Nous sommes tous des adultes consentants ». Sous entendu : si vous voulez vous tirer une balle dans le pied, allez-y, vous êtes adulte après tout. D'après le cours Python d'OpenClassrooms :
`https://openclassrooms.com/fr/courses/4302126-decouvrez-la-programmation-orientee-objet-avec-python/4313211-comprenez-lencapsulation`

b) Méthodes

Si par exemple, on teste l'addition ou l'affichage de la manière la plus intuitive, ça plante... (voir le code ci-dessous)
Il est intéressant de remarque que néanmoins la commande `print(A)`, ligne 7, fournit énormément d'information :

- `__main__.Fraction object` nous avons à faire à une instance de la classe `Fraction`
- `at 0x000000DB5BD5A940` nous connaissons l'adresse mémoire dans lequel cette instance est stockée. Cette adresse est écrite en hexadécimale.

Enfin, on aurait pu simplement écrire `A`, ligne 9, pour obtenir à peu près les mêmes informations.

```
1 >>> A + B
2 Traceback (most recent call last):
3   File "<ipython-input-8-151064de832d>", line 1, in <module>
4     A + B
5 TypeError: unsupported operand type(s) for +: 'Fraction' and 'Fraction'
6
7 >>> print(A)
8 <__main__.Fraction object at 0x000000DB5BD5A940>
9 >>> A
10 <__main__.Fraction at 0xdb5bd5a940>
```

On va commencer par créer une fonction `affiche()` qui, comme son nom l'indique, va gérer l'affichage de la fraction. Quand une fonction est directement liée à la classe `Fraction`, on la nomme **méthode**.

```
1 class Fraction :
2     '''
3     classe  définit par
4     - num le numérateur, un entier
5     - den le dénominateur, un entier non nul
6     '''
7
8     #constructeur
9     def __init__(self, num , den):
10         self.num = num
11         self.den = den
12                 #j'ai pas tout mis !
13
14
15     def affiche(self) :
16         '''
17         affiche la fraction sous la forme num/den
18         '''
19         print(str(self.num) + '/' + str(self.den))
```

Pour appliquer la **méthode** `affiche()` à l'**instance** `A`, on écrira `A.affiche()` et non pas `affiche(A)` pour bien préciser que l'on applique la méthode à `A`.

```
1 >>> A.affiche()
2 1/3
```

Exercice 3
Rajouter les trois méthodes `numerateur()`, `denominateur()` et `approche()`.

c) Méthodes utilisant 2 instances

Pour effectuer la somme de 2 fractions, ont doit avoir accès aux **attributs** de 2 **instances** de la **classe** `Fraction`.
Notre méthode **plus()** devra donc s'appliquer sur la première fraction et prendre en argument la deuxième fraction.

```
class Fraction :
    '''
    classe  définit par
    - num le numérateur, un entier
    - den le dénominateur, un entier non nul
    '''

    #constructeur
    def __init__(self, num , den):
        self.num = num
        self.den = den
                #j'ai pas tout mis

    def plus(self, F) :
        '''
        effectue la somme avec la fraction F
        '''
        #on utilise le constructeur
        calcul = Fraction(self.num * F.den + F.num * self.den , self.den * F.den )
        return calcul
```

On peut maintenant effectuer nos sommes de fractions sans soucis.

```
>>> A = Fraction(1, 3)
>>> B = Fraction(1, 2)
>>> C = A.plus(B)
>>> C.affichage()
5/6
```

Exercice 4
Rajouter les méthodes `moins()`, `fois()`, `divisePar()`, `puissance`, `plusGrand()` et `plusPetit()`.

IV. Hors programme mais diaboliquement sympathique : les fonction spéciales

– Mais Monsieur, c'est sympa votre truc, et je dois même vous avouer je trouve ça génial, mais en même temps, ça me gène de vous le dire, mais écrire régulièrement `A.plus(B)` ou `A.affiche()` je trouve ça assez pénible et même très lourd à la longue. On ne pourrait pas imaginer une solution pour écrire plus simplement `A + B` ou `print(A)` ? ? ? ?
– Et bien oui mon petit scarabée[4]*. Nous pouvons faire ça. Mais c'est hors-programme. Mais ce n'est pas très compliqué.*

Nous allons procéder à ce qu'on appelle une **surcharge d'opérateur**. Autrement dit, l'opérateur `+` fait déjà des choses formidables et variées (comme l'addition de deux entiers, l'addition d'un entier et d'un flottant, l'addition de deux flottants et enfin la concaténation de deux chaînes de caractères). Nous pouvons surcharger cet opérateur avec notre addition de deux fractions.
Pour ce faire, il suffit de remplacer le nom de notre méthode `plus` par la **fonction spéciale**ment[5] associée à l'opérateur `+` : `__add__`.

```
    def __add__(self, autre) :
        '''
        effectue la somme avec la fraction autre
        '''
        #on utilise le constructeur
        calcul = Fraction(self.num * autre.den + autre.num * self.den , self.den *
    autre.den )
        return calcul
```

Ensuite on pourra allégrement faire nos additions avec le `+` :

```
>>> A = Fraction(1,3)
>>> B = Fraction(1,2)
>>> C = A + B
>>> C.affiche()
5/6
```

– A ce propos, petit scarabée, on pourra aussi surcharger l'opérateur pour éventuellement additionner une fraction avec un entier et ... additionner un entier avec une fraction ... ce qui n'est pas tout à fait la même chose, n'est-ce pas ?
– Euh ... oui ... bien sûr ... (j'ai rien compris).

Nous avons aussi déjà vu que lorsque l'on appelle `A` la réponse n'est pas forcément ce qu'il y a de plus intéressant : `<__main__.Fraction at 0xdb5bd66198>`. On pourrait peut-être modifier la représentation de l'objet. Nous avons pour cela la **fonction spéciale** `__repr__` :

```
        def __repr__(self) :
        '''
        affiche la représentation de la
    fraction
        '''
        return "L'objet est du type "+
    self.type+" est vaut "+str(self.num)
    + '/' + str(self.den)
```

Le résultat est le suivant :

```
>>> A = Fraction(1,3)
>>> A
L'objet est du type fraction est vaut 1/3
```

4. C'est son grand retour !
5. On parle aussi de **méthode spéciale**.

De la même manière la **fonction spéciale** associé à la fonction `print()` est `__str__`. Nous avons juste à remplacer le nom de la méthode `affiche` par `__str__` :

```
    def __str__ :
        '''
        affiche la fraction sous la forme num/den si den différent de 1
                sinon affiche l'entier num
        '''
        if self.den==1:
            print(str(self.num))
        else :
            print(str(self.num) + '/' + str(self.den))
```

Ainsi on a :

```
>>> A = Fraction(1,3)
>>> B = Fraction(1,2)
>>> C = A + B
>>> print(C)
5/6
```

Voici la liste des **fonctions spéciales** qui permettent de surcharger les opérateurs de base :

Opérateur	Notation	Méthode à définir
Signe positif	+	`__pos__`
Signe négatif	-	`__neg__`
Addition	+	`__add__`
Soustraction	-	`__sub__`
Multiplication	*	`__mul__`
Division	/	`__truediv__`
Exponentiation	**	`__pow__`
Division entière	//	`__floordiv__`
Reste de la division entière	%	`__mod__`
Égal	==	`__eq__`
Différent	!=	`__ne__`
Strictement plus petit	<	`__lt__`
Plus petit ou égal	<=	`__le__`
Strictement plus grand	>	`__gt__`
Plus grand ou égal	>=	`__ge__`
« non » logique	not	`__not__`
« et » logique	and	`__and__`
« ou » logique	or	`__or__`

Exercice 5
Reprendre, améliorer et finaliser la classe `Fraction` en rajoutant un maximum de fonctions spéciales.

V. Module

Maintenant que nous avons terminé de programmer notre classe, nous avons un objet `Fraction` que l'on va pouvoir utiliser dans n'importe quel autre programme qui le nécessiterait.

- Sauvegarder votre code sous le nom `maFraction.py`. On a ainsi un module `maFraction` disponible.
- Ouvrer un nouvel onglet.
- Commencer par importer la classe `Fraction` provenant de votre module.
- Utiliser les méthodes.

```
1 from maFraction import Fraction
2
3 A = Fraction(1,3)
4 B = Fraction(1,2)
5 print(A+B)
```

VI. Hors programme : notion de public privé

Désolé, c'est hors programme et nous n'avons pas le temps d'aborder ce point ... Donc je n'en parlerai pas.

VII. Annexe : la PEP 8

Une PEP (ou *Python Enhancement Proposals*) est comme son nom l'indique une proposition d'amélioration de Python. Une fois une PEP validée, elle est publiée avec son numéro. La PEP 8 est celle auquel on fait le plus référence car elle conseille sur les bons usages d'écriture du code. Respecter la PEP 8 n'est pas une obligation, mais si on la respecte du mieux possible, son code est plus rapidement accessible à une autre personne.
D'une manière générale, on doit utiliser la convention dite **CamelCase**, où la « casse de chameau ». Cette convention consiste à écrire sans espace, sans _, en mettant en lettre capital le début de chaque mot, ce qui provoque des ondulations comme un dos de chameau : DesOndulationsCommeUnDosDeChameau.

- On n'utilise jamais de caractères spéciaux (accents, espace, ..)
- Le nom d'une variable doit commencer par une minuscule : `maVariable`
- Le nom d'une classe doit commencer par une majuscule : `MaClasse`
- Le nom d'une fonction doit commencer par une minuscule : `maFonction()`

`https://www.python.org/dev/peps/pep-0008/`

VIII. Exercices

Exercice 6 (Boules de neige)

Boule de neige II

Comme on peut le lire dans l'article wikipédia suivant `https://fr.wikipedia.org/wiki/Liste_des_chats_des_Simpson`, la plupart des chats des Simpson s'appelle *Boule de Neige*.
Créer une interface décrivant les interactions éventuelles entre les chats des Simpson, Lisa et Bart (mais vous pouvez rajouter d'autres personnages).
Implémenter cette interface.

Exercice 7

Écrire la classe `Eleve` qui permettrait pour un élève de stocker dans un élève les données suivantes : son nom, son prénom, ses notes obtenues à trois DS, ainsi qu'une méthode qui calculerait la moyenne des trois notes.

1. Compléter :

```
class Eleve:
    def __init__(self, Nom, Prenom, Note1, Note2, Note3):
        ....
        ....
        ....
        ....
        ....
        ....

    def moyenne(self):
        ....
        ....
```

2. (a) Définir une instance pour Ada Lovelace qui a obtenu les notes suivantes : 18 au DS1, 19 au DS2 et 20 au DS3.
   (b) Écrire l'instruction pour faire afficher sa moyenne.

Exercice 8 (*)

Créer une classe `Horloge`, puis la tester.
Les attributs sont :

- `heures` ;
- `minutes` ;
- `secondes`.

Les méthodes sont :

- `ticTac` : cette méthode augmente l'horloge d'une seconde ;
- `reveil(h , mn , s)` : sonne le réveil à une heure donnée par l'utilisateur ;
- `__repr__` : représente l'objet.

Exercice 9 (**)

Exercice proposé par Frédéric Manson, professeur NSI.

1. Créer une classe `Boite`. Cette classe a pour attributs :
   - `longueur`
   - `largeur`
   - `hauteur`
   - Ces trois attributs sont dans un ordre décroissant longueur $\geq$ largeur $\geq$ hauteur (on peut choisir des dimensions entre 1 et 50)

   Elle a pour méthodes :
   - `volume`, qui comme son nom l'indique donne le volume d'une boite
   - `rentreDans(autreBoite)`, qui renvoie vrai si l'objet `Boite` rentre dans `autreBoite`.
2. Créer aléatoirement une liste d'une vingtaine de boîtes.
3. A l'aide d'un algorithme glouton, donner une suite de boîtes aussi grande que possible qui rentrent les unes dans les autres.
   Rappel : pour trier les boîtes, on fera appel à la fonction `sorted()` ou la méthode `sort()` avec en argument une fonction qui donne la clef pour effectuer le tri. Cette fonction à passer en paramètre prend un élément de la liste et retourne ce sur quoi doit s'effectuer le tri.

```
def laClef(objet):
    """
    Renvoie la valeur qui sera examinée pour le tri
    """
    return valeur de l'objet
liste_tries = sorted(liste, key = laClef)
```

   On peut rajouter `reverse = True` pour avoir l'ordre décroissant.

IX. Corrections

Correction de l'exercice 1

Proposer un interface le plus précis possible pour notre nouveau type `Fraction`.

Fonction	Description
`fraction(n, d)`	Renvoie une fraction irréductible correspondant à n/d et dont le dénominateur est positif et non nul.
`numerateur(F)`	Renvoie le numérateur de la fraction F.
`denominateur(F)`	Renvoie le dénominateur de la fraction F.
`affiche(F)`	Affiche la fraction F dans la console sous la forme numérateur/dénominateur.
`approche(F)`	Renvoie une valeur approchée de la fraction.
`plus(F1, F2)`	Renvoie une fraction irréductible égale à F1 + F2 et dont le dénominateur est positif.
`moins(F1, F2)`	Renvoie une fraction irréductible égale à F1 - F2 et dont le dénominateur est positif.
`fois(F1, F2)`	Renvoie une fraction irréductible égale à F1 $\times$ F2 et dont le dénominateur est positif.
`divisePar(F1, F2)`	Renvoie une fraction irréductible égale à F1 $\div$ F2 et dont le dénominateur est positif.
`puissance(F, exp)`	Renvoie une fraction irréductible égale à F^{exp}.
`plusGrand(F1, F2)`	Renvoie Vrai si F1 est plus grand que F2.
`plusPetit(F1, F2)`	Renvoie Vrai si F1 est plus petit que F2.

Correction de l'exercice 2

Améliorer le constructeur de la classe `Fraction` de la manière suivante :

- En utilisant la commande `assert` on va s'assurer que la fraction créée est composée de deux nombres entiers, le dénominateur étant non nul.
- Quelques soient les nombres entiers proposés par l'utilisateur, la fraction stockée ne devra pas avoir de signe négatif au dénominateur. Exemples : $\frac{1}{-2} = \frac{-1}{2}$ ou $\frac{-7}{-4} = \frac{7}{4}$.
- Quelques soient les nombres entiers proposés par l'utilisateur, la fraction stockée sera sous sa forme irréductible.
- On rajoute l'attribut `type` qui dans notre cas sera toujours `"fraction"`.

```
class Fraction :
    '''
    classe  définit par
    - num le numérateur, un entier
    - den le dénominateur, un entier non nul
    '''
    #constructeur
    def __init__(self, num, den):
        #vérifications d'usage
        assert den!=0, "on ne peut diviser par zéro"
        assert type(num)==int and type(den)==int, "le numérateur et le dénominateur
     doivent être des entiers"

        #signe
        if den < 0 :
            num, den = -num, -den

        #simplification en calculant le PGCD
        #autrement dit le plus grand diviseur commun
        m = min(num, den)
        #d est le diviseur commun
        #au minimum 1 au maximum m, le minimum de num et den
        d = 1
        for i in range(2,m+1):
            if (num%i==0 and den%i==0):
                #c'est un diviseur commun
                d = i
        self.num = num//d
        self.den = den//d
        self.keskc = "fraction"
```

Correction de l'exercice 3

Rajouter les trois méthodes `numerateur()`, `denominateur()` et `approche()`.

```
class Fraction :
    '''
    classe  définit par
    - num le numérateur, un entier
    - den le dénominateur, un entier non nul
    '''

    #constructeur
    def __init__(self, num, den):
        #vérifications d'usage
        assert den!=0, "on ne peut diviser par zéro"
        assert type(num)==int and type(den)==int, "le numérateur et le dénominateur doivent être des entiers"

        #signe
        if den < 0 :
            num, den = -num, -den

        #simplification en calculant le PGCD
        #autrement dit le plus grand diviseur commun
        m = min(num, den)
        #d est le diviseur commun
        #au minimum 1 au maximum m, le minimum de num et den
        d = 1
        for i in range(2,m+1):
            if (num%i==0 and den%i==0):
                #c'est un diviseur commun
                d = i
        self.num = num//d
        self.den = den//d
        self.keskc = "fraction"

    def numerateur(self):
        '''
        renvoie le numérateur de la fraction
        '''
        return self.num

    def denominateur(self):
        '''
        renvoie le dénominateur de la fraction
        '''
        return self.den

    def affiche(self) :
        '''
        affiche la fraction sous la forme num/den si den différent de 1
                sinon affiche l'entier num
        '''
        if self.den==1:
            print(str(self.num))
        else :
            print(str(self.num) + '/' + str(self.den))

    def approche(self) :
        '''
       renvoie une valeur approchée de la fraction
        '''
        return self.num / self.den
```

Correction de l'exercice 4

Rajouter les méthodes `moins()`, `fois()`, `divisePar()`, `puissance`, `plusGrand()` et `plusPetit()`.

```
class Fraction :
    '''
    classe  définit par
    - num le numérateur, un entier
    - den le dénominateur, un entier non nul
    '''

    #constructeur
    def __init__(self, num, den):
        #vérifications d'usage
        assert den!=0, "on ne peut diviser par zéro"
        assert type(num)==int and type(den)==int, "le numérateur et le dénominateur doivent être des entiers"

        #signe
        if den < 0 :
            num, den = -num, -den

        #simplification en calculant le PGCD
        #autrement dit le plus grand diviseur commun
        m = min(num, den)
        #d est le diviseur commun
        #au minimum 1 au maximum m, le minimum de num et den
        d = 1
        for i in range(2,m+1):
            if (num%i==0 and den%i==0):
                #c'est un diviseur commun
                d = i
        self.num = num//d
        self.den = den//d
        self.keskc = "fraction"

    def numerateur(self):
        '''
        renvoie le numérateur de la fraction
        '''
        return self.num

    def denominateur(self):
        '''
        renvoie le dénominateur de la fraction
        '''
        return self.den

    def affiche(self) :
        '''
        affiche la fraction sous la forme num/den si den différent de 1
                sinon affiche l'entier num
        '''
        if self.den==1:
            print(str(self.num))
        else :
            print(str(self.num) + '/' + str(self.den))

    def approche(self) :
        '''
       renvoie une valeur approchée de la fraction
        '''
        return self.num / self.den

    def plus(self, autre) :
```

```
61          '''
62          effectue la somme avec la fraction autre
63          '''
64          #on utilise le constructeur
65          calcul = Fraction(self.num * autre.den + autre.num * self.den , self.den *
        autre.den )
66          return calcul
67
68      def moins(self, autre) :
69          '''
70          effectue la différence avec la fraction autre
71          '''
72          #on utilise le constructeur
73          calcul = Fraction(self.num * autre.den - autre.num * self.den , self.den *
        autre.den )
74          return calcul
75
76      def fois(self, autre) :
77          '''
78          effectue le produit avec la fraction autre
79          '''
80          #on utilise le constructeur donc pas besoin de simplifier
81          calcul = Fraction(self.num * autre.num, self.den * autre.den )
82          return calcul
83
84      def divisePar(self, autre) :
85          '''
86          effectue le quotient avec la fraction autre
87          '''
88          #on utilise le constructeur
89          calcul = Fraction(self.num * autre.den, self.den * autre.num )
90          return calcul
91
92      def puissance(self, exposant):
93          '''
94          calcule la fraction à la puissance exposant
95          '''
96          calcul = Fraction(self.num**exposant, self.den**exposant)
97          return calcul
98
99      def plusGrand(self, autre) :
100         '''
101         renvoie Vrai si la fraction est plus grande que autre
102         '''
103         #on passe par une mise au même dénominateur
104         return (self.num * autre.den >= autre.num * self.den)
105
106     def plusPetit(self, autre) :
107         '''
108         renvoie Vrai si la fraction est plus petite que autre
109         '''
110         #on passe par une mise au même dénominateur
111         return (self.num * autre.den <= autre.num * self.den)
```

Correction de l'exercice 5

Reprendre, améliorer et finaliser la classe `Fraction` en rajoutant un maximum de fonctions spéciales.

```
def PGCD(a,b):
    '''
    renvoie le PGCD de a et de b
    autrement le plus grand diviseur commun
    '''
    m = min(a,b)
    #d est le diviseur commun
    #au minimum 1 au maximum m, le minimum de num et den
    d = 1
    for i in range(2,m+1):
        if (a%i==0 and b%i==0):
            #c'est un diviseur commun
            d = i
    return d


class Fraction :
    '''
    classe  définit par
    - num le numérateur, un entier
    - den le dénominateur, un entier non nul
    '''

    #constructeur
    def __init__(self, num, den):
        #vérifications d'usage
        assert den!=0, "on ne peut diviser par zéro"
        assert type(num)==int and type(den)==int, "le numérateur et le dénominateur doivent être des entiers"
        #signe
        if den < 0 :
            num, den = -num, -den
        #simplification
        d = PGCD(num, den)
        self.num = num//d
        self.den = den//d
        self.kskc = "fraction"

    def numerateur(self):
        '''
        renvoie le numérateur de la fraction
        '''
        return self.num

    def denominateur(self):
        '''
        renvoie le dénominateur de la fraction
        '''
        return self.den

    def __repr__(self) :
        '''
        affiche la représentation de la fraction
        '''
        return "L'objet est du type "+self.kskc+" est vaut "+str(self.num) + '/' + str(self.den)

    def __str__(self) :
        '''
        affiche la fraction sous la forme num/den si den différent de 1
                sinon affiche l'entier num
```

```
60          '''
61          if self.den==1:
62              return str(self.num)
63          else :
64              return str(self.num) + '/' + str(self.den)
65
66      def __add__(self, autre) :
67          '''
68          effectue la somme avec la fraction autre
69          '''
70          #on utilise le constructeur
71          calcul = Fraction(self.num * autre.den + autre.num * self.den , self.den *
        autre.den )
72          return calcul
73
74      def __sub__(self, autre) :
75          '''
76          effectue la différence avec la fraction autre
77          '''
78          #on utilise le constructeur
79          calcul = Fraction(self.num * autre.den - autre.num * self.den , self.den *
        autre.den )
80          return calcul
81
82      def __mul__(self, autre) :
83          '''
84          effectue le produit avec la fraction autre
85          '''
86          #on utilise le constructeur donc pas besoin de simplifier
87          calcul = Fraction(self.num * autre.num, self.den * autre.den )
88          return calcul
89
90      def __truediv__(self, autre) :
91          '''
92          effectue le quotient avec la fraction autre
93          '''
94          #on utilise le constructeur
95          calcul = Fraction(self.num * autre.den, self.den * autre.num )
96          return calcul
97
98      def __pow__(self, exposant):
99          '''
100         calcule la fraction à la puissance exposant
101         '''
102         calcul = Fraction(self.num**exposant, self.den**exposant)
103         return calcul
104
105     def __ge__(self, autre) :
106         '''
107         renvoie Vrai si la fraction est plus grande que autre
108         '''
109         #on passe par une mise au même dénominateur
110         return (self.num * autre.den >= autre.num * self.den)
111
112     def __gt__(self, autre) :
113         '''
114         renvoie Vrai si la fraction est strictement plus grande que autre
115         '''
116         #on passe par une mise au même dénominateur
117         return (self.num * autre.den > autre.num * self.den)
118
119     def __le__(self, autre) :
120         '''
121         renvoie Vrai si la fraction est plus petite que autre
122         '''
```

```
123         #on passe par une mise au même dénominateur
124         return (self.num * autre.den <= autre.num * self.den)
125 
126     def __lt__(self, autre) :
127         '''
128         renvoie Vrai si la fraction est strictement plus petite que autre
129         '''
130         #on passe par une mise au même dénominateur
131         return (self.num * autre.den < autre.num * self.den)
```

Correction de l'exercice 6

Correction de l'exercice 7

Correction de l'exercice 8 (*)

Créer une classe `Horloge`, puis la tester.
Les attributs sont :

- `heures` ;
- `minutes` ;
- `secondes`.

Les méthodes sont :

- `ticTac` : cette méthode augmente l'horloge d'une seconde ;
- `reveil(h , mn , s)` : sonne le réveil à une heure donnée par l'utilisateur ;
- `__repr__` : représente l'objet.

```
def rajouterZero(nb):
    '''
    permet de rajouter un 0 si le nombre est plus petit que 10
    '''
    if nb < 10 :
        return "0"+str(nb)
    else :
        return str(nb)

class Horloge :
    '''
    simule une horloge avec la mesure du temps en heure, minute et seconde
    '''

    def __init__(self):
        self.heure = 0
        self.minute = 0
        self.seconde = 0

    def ticTac(self):
        '''
        rajoute 1 seconde
        '''
        if self.seconde < 59:
            self.seconde += 1
        elif self.minute < 59:
            self.seconde = 0
            self.minute += 1
        else :
            self.seconde = 0
            self.minute = 0
            self.heure += 1

    def reveil(self, h, m, s):
        '''
        sonne le réveil
        '''
        if self.heure == h and self.minute == m and self.seconde == s :
            print("Dring ! Dring !")
            return True
        else :
            return False

    def __repr__(self):
        h = rajouterZero(self.heure)
```

```
        m = rajouterZero(self.minute)
        s = rajouterZero(self.seconde)
        return str(h)+"/"+str(m)+"/"+str(s)


################################################
# On teste la classe
################################################
Hor = Horloge()

#et si on se réveillait à 18h14min05s ???
while not Hor.reveil(18,14,5) :
    print(Hor)
    Hor.ticTac()
```

Correction de l'exercice 9 (**)

Exercice proposé par Frédéric Manson, professeur NSI.

1. Créer une classe `Boite`. Cette classe a pour attributs :
   - `longueur`
   - `largeur`
   - `hauteur`
   - Ces trois attributs sont dans un ordre décroissant longueur $\geq$ largeur $\geq$ hauteur (on peut choisir des dimensions entre 1 et 50)

   Elle a pour méthodes :
   - `volume`, qui comme son nom l'indique donne le volume d'une boite
   - `rentreDans(autreBoite)`, qui renvoie vrai si l'objet `Boite` rentre dans `autreBoite`.
2. Créer aléatoirement une liste d'une vingtaine de boîtes.
3. A l'aide d'un algorithme glouton, donner une suite de boîtes aussi grande que possible qui rentrent les unes dans les autres.
   Rappel : pour trier les boîtes, on fera appel à la fonction `sorted()` ou la méthode `sort()` avec en argument une fonction qui donne la clef pour effectuer le tri. Cette fonction à passer en paramètre prend un élément de la liste et retourne ce sur quoi doit s'effectuer le tri.

```
def laClef(objet):
    """
    Renvoie la valeur qui sera examinée pour le tri
    """
    return valeur de l'objet
liste_tries = sorted(liste, key = laClef)
```

   On peut rajouter `reverse = True` pour avoir l'ordre décroissant.

```
from random import randint

class Boite :
    '''
    boite définie par longueur, largeur, hauteur
    avec longueur >= largeur >= hauteur
    '''

    def __init__(self) :
        self.longueur = randint(30,50)
        self.largeur = randint(10,30)
        self.hauteur = randint(1,10)

    def volume(self) :
```

```
        return self.longueur * self.largeur * self.hauteur

    def rentreDans(self, autre) :
        testLongueur = self.longueur < autre.longueur
        testLargeur = self.largeur < autre.largeur
        testHauteur = self.hauteur < autre.hauteur
        return testLongueur and testLargeur and testHauteur

    ########################################################
    # non demandé
    ########################################################
    def __str__(self) :
        return str(self.longueur)+" "+str(self.largeur)+" "+str(self.hauteur)


#création d'une liste de 20 boîtes
liste = [Boite() for i in range(20)]

#vérification
print("liste non classée")
for i in range(20):
    print(liste[i])

#classement des boîtes du plus grand au plus petit
def laClef(objet) :
    return objet.longueur, objet.largeur, objet.hauteur
liste.sort(key = laClef, reverse = True)

#vérification
print("Liste classée")
for i in range(20):
    print(liste[i])


##############################################
# Algorithme glouton
##############################################
#liste qui recevra les boîtes emboîtées
listeEmboitee = [liste[0]]
#compteur
i = 1
while i<len(liste) :
    nouvBoite = liste[i]
    if liste[i].rentreDans(listeEmboitee[len(listeEmboitee)-1]) :
        #la nouvelle boite rentre dans la dernière boite choisie
        listeEmboitee.append(liste[i])
    i = i + 1

#vérification
print("liste des boites enboitées")
for i in range(len(listeEmboitee)) :
    print(listeEmboitee[i])
```