

Cours d'algorithmique et programmation 2

Cours 2

Saïd Jabbour

UFR des Sciences
Licence Sciences et Technologie
mentions Mathématiques et Informatique
Semestre 2

Section 1

Représenter des données complexes

Représenter des données complexes

Quelques exemples d'objets et des caractéristiques qui les définissent :

▶ le **guerrier** d'un jeu par tour :

- ▶ son nom
- ▶ son nombre de points de vie
- ▶ son force
- ▶ son adresse
- ▶ ...

▶ un **nombre rationnel** :

- ▶ son numérateur
- ▶ son dénominateur

▶ un **étudiant** :

- ▶ son nom
- ▶ son prénom
- ▶ son âge
- ▶ ses notes en algo2
- ▶ ...

▶ une **formation** :

- ▶ son nom
- ▶ son niveau (L, M, D)
- ▶ ses étudiants
- ▶ ...

Représenter des données complexes

- ▶ Chacun des objets précédents nécessite **plusieurs données d'un niveau plus élémentaires**
- ▶ Nécessité de **collecter** toutes ces données ensemble :
 - ▶ les **listes** le permettent, mais ne sont pas naturelles dans ces exemples

```
# pour représenter 1/3
maFraction = [1, 3]
# pour calculer un arrondi de 1/3
maFraction[0]/maFraction[1]

# pour représenter un guerrier
unGuerrier = ['Boris', 78, 8, 90, 10]
# pour savoir s'il est vivant
unGuerrier[1] > 0
```

Il faut numéroté les caractéristiques !

Représenter des données complexes

- ▶ Chacun des objets précédents nécessite plusieurs **données d'un niveau plus élémentaires**
- ▶ Nécessité de **collecter** toutes ces données ensemble :
 - ▶ les **dictionnaires** sont plus adaptés :

```
# pour représenter 1/3
maFraction = {'num': 1, 'den': 3}
# pour calculer un arrondi de 1/3
maFraction['num']/maFraction['den']

# pour représenter un guerrier
unGuerrier = {'nom': 'Boris', 'ptsDeVie': 78,
              'force': 8, 'adresse': 90, 'armure': 10}
# pour savoir s'il est vivant
unGuerrier['ptsDeVie'] > 0
```

- ▶ Mais les dictionnaires sont lourds à manipuler
- ▶ Les noms des caractéristiques sont représentées par des chaînes de caractères
- ▶ Les dictionnaires sont d'un **type mutable** !

Section 2

Les classes

Construire son propre type

Python (comme tous les langages de programmation) permet de **définir son propre type** de données.

Objectif : on veut pouvoir définir et utiliser des fractions ainsi

```
>>> f1 = Frac(1,3) # represente 1/3
>>> print(f1)
1/3
>>> f2 = Frac(1,9) # represente 1/9
>>> f3 = f1.add(f2) # calcule 1/3 + 1/9 et retourne 4/9
>>> print(f3)
4/9
>>> f3.num()
4
>>> f3.den()
9
```

On va associer aux caractéristiques de l'objet complexe toutes les fonctionnalités nécessaires pour l'utiliser

⇒ On trouvera dans un type **des données** et **des fonctions** pour manipuler les données.

Les classes

Un nouveau type de données s'appellera toujours une **classe** en Python.
Première étape : on définit la classe, et comment obtenir une **instance** de la classe. On écrit le **constructeur** de la classe (la méthode `__init__`).

```
1 class Frac:
2     '''La classe Frac permet de modeliser un nombre
3     rationnel, avec son numerateur et son denominateur.
4     '''
5
6     def __init__(self,num,denom):
7         '''Frac,int,int -> Frac
8         cree une fraction a partir de son numerateur
9         et de son denominateur.
10        '''
11        # une fraction va posseder
12        # une caracteristique __num
13        self.__num = num
14        # une fraction va posseder
15        # une caracteristique __den
16        self.__den = denom
```


Classes et instances

- ▶ On appelle **classe** un type en Python.
Cela est vrai pour les types prédéfinis que vous connaissez :

```
>>> help(int)
Help on class int in module builtins:
class int(object)
|   int(x=0) -> integer
|   int(x, base=10) -> integer
|   Convert a number or string to an integer,
|   ...

>>> help(list)
Help on class list in module builtins:
class list(object)
|   list() -> new empty list
|   list(iterable) -> new list initialized
|                   from iterable's items
|   ...
```

Classes et instances

- ▶ On appelle **classe** un type en Python.
Cela est vrai pour les types que vous aurez définis :

```
>>> import exples3
>>> help(Frac)
Help on class Frac in module exples3:
class Frac(builtins.object)
|   La classe Frac permet de modeliser un nombre
|   rationnel, avec son numerateur et son
|   denominateur.
...

```

Classes et instances

- On appelle **instance** un objet construit sur le modèle d'une classe

```
1 >>> l1 = list() # l1 est une instance de list
2 >>> l2 = [1,2]  # l2 est une autre instance de list
3 >>> f = exples3.Frac(1,3)
4           # f est une instance de Frac
5 >>> type(l1)
6 <class 'list'>
7 >>> type(f)
8 <class 'exples3.Frac'>
```

Plus précisément, les variables **l1**, **l2** et **f** contiennent la référence, respectivement, d'une instance de **list**, d'une autre instance de **list**, et d'une instance de **Frac**.

Encore un peu de vocabulaire

- ▶ On appelle **classe** un type.
- ▶ On appelle **instance** un objet de ce type.
- ▶ On appelle **attributs** les données qui caractérisent une instance. Par exemple, `__num` et `__den` sont des attributs de la classe `Frac`. Soient `f1` et `f2` les deux variables définies par :

```
>>> f1 = Frac(1,3)
>>> f2 = Frac(2,9)
```

On a `f1.__num` et `f1.__den` qui valent respectivement 1 et 3 ;
tandis que `f2.__num` et `f2.__den` qui valent respectivement 2 et 9.

- ▶ On appelle **méthodes** les fonctions définies dans une classe. Elles sont associées à une syntaxe particulière.

Que contient une classe ?

- ▶ Chaque classe possède un **constructeur** : c'est la méthode qui permet de créer une instance. Elle s'appelle `__init__`
 - ▶ Comme toutes les méthodes, son premier paramètre est `self` : c'est la variable qui contient la référence de l'instance courante.
 - ▶ Elle définit les informations qui doivent être fournies pour construire une instance (ses autres paramètres).
 - ▶ C'est dans le constructeur qu'on définit et qu'on initialise les **attributs** des instances. Leur nom commence toujours par `__`

```
1 class Frac:
2
3     def __init__(self, num, denom=1):
4         '''Frac,int,int -> Frac
5         '''
6         self.__num = num
7         self.__den = denom
```

permet :

```
>>> f = Frac(2,9) # represente 2/9
>>> f2 = Frac(5)  # represente 5/1 = 5
```

Que contient une classe ?

- ▶ Chaque classe possède **des méthodes** : ce sont les fonctionnalités que l'objet doit fournir :
 - ▶ pour les fractions, on veut pouvoir additionner deux fractions, multiplier deux fractions, soustraire deux fractions, ...
 - ▶ pour le cas des guerriers, on veut qu'une instance puisse dire si elles en vie, si elle a atteint sa cible, ...
- ▶ La classe est **responsable** de ses attributs, elle doit contrôler leur cohérence
- ▶ Ainsi on accède aux attributs d'une instance **uniquement** depuis les méthodes de la classe de l'instance.
- ▶ Le premier paramètre d'une méthode est toujours **self**
- ▶ Spécificité syntaxique : on appelle une **méthode** en utilisant la **notation pointée** : **comme pour les listes !**.

```
>>> l = list()
>>> l.append(3) # j'appelle la methode append
                # de la classe list sur l'instance l
```

Additionner deux fractions - À vous !

```
1 class Frac :
2     # suite de la classe ...
3
4     def add(self,f) :
5         '''Frac,Frac -> Frac
6         retourne une nouvelle fraction qui est
7         le resultat de l'addition de self et f.
8         '''
9         numerateur = self.__num*f.__den
10                    + f.__num*self.__den
11         denominateur = self.__den*f.__den
12         return Frac(numerateur,denominateur)
```

On peut maintenant faire :

```
>>> f1 = Frac(1,3)  # represente 1/3
>>> f2 = Frac(1,9)  # represente 1/9
>>> f3 = f1.add(f2) # calcule 1/3 + 1/9 et retourne ???
```

Que contient **f3** ?

Les fractions - bilan d'étape

Il nous manque :

- ▶ la conservation sous forme irréductible des fractions
- ▶ un moyen pour afficher une fraction
- ▶ soustraire deux fractions
- ▶ multiplier deux fractions
- ▶ inverser une fraction
- ▶ proposer un arrondi de la fraction : en entier, en flottant,
- ▶ ...

Afficher une fraction

```
1 class Frac:
2     # suite ...
3     def str(self) :
4         '''Frac -> str
5         retourne une representation
6         de la fraction en str.
7         '''
8         return str(self.__num)+'/'+str(self.__den)
9
10    def affiche(self) :
11        '''Frac -> rien
12        affiche la fraction sur la sortie standard.
13        '''
14        print(self.str())
```

Afficher une fraction

Maintenant on peut faire :

```
>>> f1 = Frac(1,3) # represente 1/3
>>> f2 = Frac(1,9) # represente 1/9
>>> f3 = f1.add(f2) # calcule 1/3 + 1/9 et retourne ???
>>> f3.affiche()
12/27
```

Les appels de méthode

La notation pointée est utilisée couramment, mais ce n'est que du *sucre syntaxique*. Les deux instructions suivantes sont équivalentes :

```
>>> f3 = f1.add(f2)
# j'appelle la methode
# add de f1 et je passe
# f2 en parametre
# (self c'est f1)
```

```
>>> f3 = Frac.add(f1,f2)
# j'appelle la fonction
# add de la classe Frac
# avec en parametres
# f1 (pour self) et f2
```

Comme on pouvait déjà le faire avec les listes :

```
>>> l = list()
>>> l.append(1)
# j'appelle la methode
# append de l et
# je passe 1 en parametre
#
```

```
>>> l = list()
>>> list.append(l,1)
# j'appelle la fonction
# append de la classe list
# avec en parametres
# l (pour self) et 1
```

Retour aux fractions...- À vous !

- ▶ On reporte à plus tard la question de la réduction de fraction
- ▶ Proposez les méthodes pour
 - ▶ retourner le numérateur d'une fraction
 - ▶ retourner le dénominateur d'une fraction
 - ▶ retourner le produit d'une fraction par un entier
 - ▶ retourner le produit de deux fractions
 - ▶ retourner la soustraction de deux fractions
 - ▶ retourner l'inverse d'une fraction
 - ▶ retourner l'arrondi à un entier d'une fraction
 - ▶ retourner l'arrondi à un flottant d'une fraction