Cours d'algorithmique et programmation 2

Anne Parrain

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

Section 1

Les assertions

Les assertions

Question : Comment faire pour vérifier des conditions sur des variables imposées par les algorithmes ?

```
assert expression_booleenne
```

- Les assertions sont des conditions qui doivent absolument être vérifiées lors de l'exécution du programme.
- Elles ne peuvent pas être *réparées* par le programme.
- On peut les voir comme une manière forte d'exprimer une contrainte de spécification ;
- Ou pour exprimer et garantir des propriétés clés à des endroits pertinents dans le code.

Exemple d'utilisation des assertions (2)

```
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!=0 -> Frac
8          cree une fraction a partir de son numerateur
9          et de son denominateur.
10         '''
11         assert denom != 0
12         # une fraction va posseder
13         # une caracteristique __num
14         self.__num = num
15         # une fraction va posseder
16         # une caracteristique __den
17         self.__den = denom
```

Exemple d'utilisation des assertions (2 - suite)

```
>>> f = Frac(3,2)
>>> f = Frac(5,0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in __init__
AssertionError
```

Section 2

Quelques méthodes particulières

Nous avons encore des petits soucis. . .

```
>>> from fractions import *
>>> f = Frac(5,6)
>>> print(f)
<fractions.Frac object at 0x7f4a8e110400>
>>> f2 = Frac(50,60)
>>> f == f2
False
>>> f3 = Frac(5,6)
>>> f == f3
False
>>> id(f)
139958187787264
>>> id(f3)
139958187789168
>>> id(f2)
139958187787376
```

Des méthodes un peu particulières

Toutes les classes écrites en Python possèdent de base un certain nombre de méthodes. Ces méthodes ont un nom de la forme `__nom_methode__`.

Exemples :

- `__init__`
- `__str__`
- `__eq__`
- `...`

Des méthodes un peu particulières

Ainsi, on peut écrire une classe `A` vide, construire des instances, les afficher, les comparer.

```
1 class A:
2     def f(self):
3         print("beuh")
```

ce qui peut donner :

```
>>> a = A()
>>> print(a)
<__main__.A object at 0x7f4a8e11d048>
>>> b = A()
>>> a == b
False
```

Redéfinir la méthode `__str__`

- La redéfinition de `__init__` est obligatoire pour définir correctement ses propres instances
- De la même façon, on peut redéfinir la méthode `__str__` : lors d'une instruction `print(a)` où a est une variable quelconque, python exécute en fait `print(a.__str__())`.

Redéfinir certaines méthodes particulières

On remplace la méthode `affiche` de la classe `Frac` (ou `FracMut` par la méthode `__str__` :

```
1 # dans la classe Frac
2     def __str__(self):
3         '''Frac -> str'''
4         return self.str()
```

(On peut maintenant supprimer `affiche`)

On obtient maintenant :

```
>>> from fractions import *
>>> f = Frac(5,6)
>>> print(f)
5/6
```

Tester l'égalité de deux fractions

Deux fractions sont équivalentes si elles représentent la même fraction...
On veut pouvoir exprimer que $\frac{5}{2} = \frac{10}{4}$.

- Pour cela, on redéfinit dans la classe `Frac` (ou `FracMut`) la méthode `__eq__` :

```
1 # dans la classe Frac
2     def __eq__(self,f):
3         '''Frac, Frac -> boolean'''
4         return self.__num == f.num()
5                and self.__den  == f.den()
```

- Maintenant, il faut conserver les fractions sous forme irréductible ...

Le plus grand commun diviseur

```
1  # hors de la class Frac (ou FracMut)
2  def pgcd(a,b) :
3      '''int, int -> int
4      retourne le plus grand commun diviseur de a et de b
5      '''
6      n1 = a
7      n2 = b
8      while n2 != 0 :
9          temp = n2
10         n2 = n1%n2
11         n1 = temp
12     return n1
```

Un nouveau constructeur

```
1  class Frac :
2      def __init__(self,num,denom):
3          '''Frac,int,int -> Frac
4          cree une fraction a partir de son numerateur
5          et de son denominateur.
6          '''
7          assert denom !=0
8          pg = pgcd(num,denom)
9          self.__num = num//pg
10         self.__den = denom//pg
11         if self.__den < 0 and self.__num > 0 :
12             self.__den = -self.__den
13             self.__num = -self.__num
```

Tester l'égalité de deux fractions

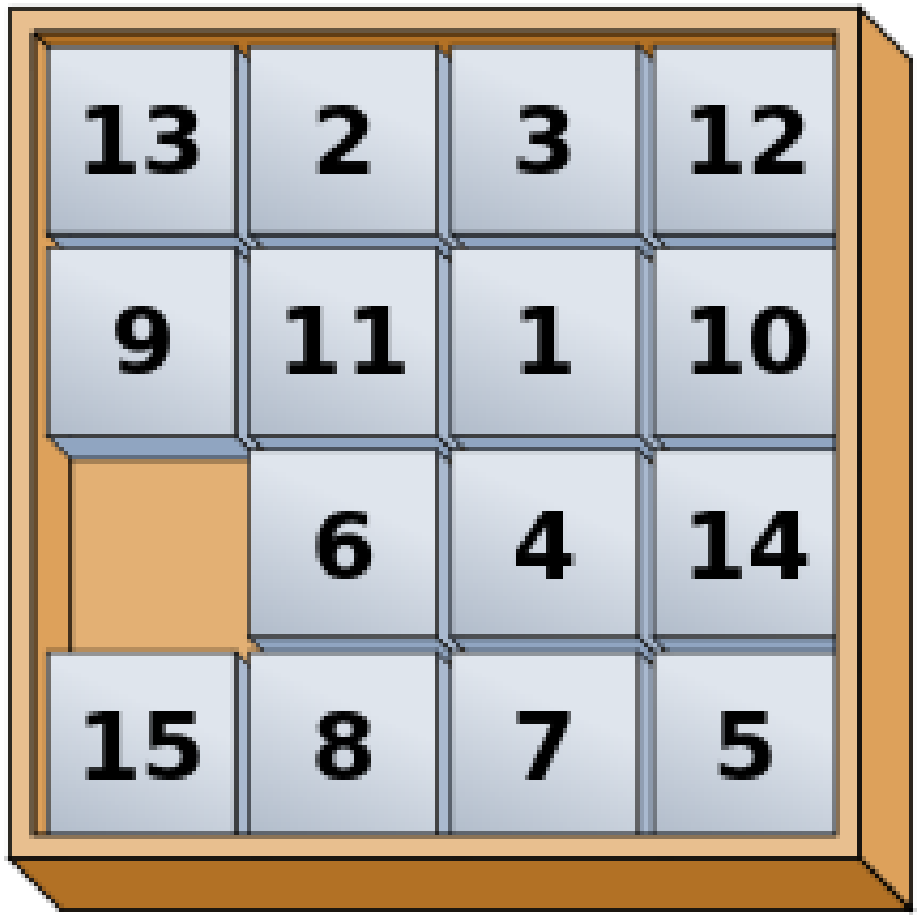
Et maintenant :

```
>>> from fractions import *
>>> f = Frac(5,6)
c>>> f2 = Frac(50,60)
>>> f == f2
True
>>> id(f)
139958187787264
>>> id(f2)
139958187787376
```

Section 3

Un nouveau jeu

Le jeu du taquin



Modéliser le jeu de taquin

Représenter le jeu

- le plateau de jeu est une matrice 4×4
- dont chaque cellule a une valeur entre 1 et 15
- sauf la cellule vide qui contient -1

Fonctionnalités

- afficher le plateau de jeu
- bouger un palet
- initialiser une partie
- détecter une fin de partie

Application principale

```
if __name__ == '__main__' :
    taq = Taquin(7)          # par defaut dimension 4 !
    print(taq)
    taq.reinit()
    while not taq.partie_finie():
        print(taq)
        choix = int(input("Quelle valeur voulez-vous\
                           bouger ? "))
        res = taq.bouge_case(choix)
        if not res :
            print("Non, ce n'est pas possible avec\
                   cette valeur.")
    print(taq)
    print("Partie finie. Bravo")
```

Construire le taquin – À vous !

- on peut donner en paramètre la dimension du taquin
- par défaut, elle vaut 4
- il faut construire le plateau de jeu
- et mémoriser les coordonnées du palet manquant (*le vide*)

Construire le taquin

```
1  class Taquin:
2      '''Classe qui modelise le jeu du taquin.
3      Par defaut, le taquin est construit en dim. 4 x 4.
4      '''
5
6      def __init__(self,dim=4):
7          '''Taquin, int -> Taquin
8          construit un taquin de dim lignes x dim colonnes
9          '''
10         assert dim > 1
11         self.__plateau = []
12         self.__dim = dim
13         for i in range(dim):
14             ligne = []
15             for j in range(dim):
16                 ligne.append((i*dim)+j+1)
17             self.__plateau.append(ligne)
18         self.__plateau[-1][-1] = -1
19         self.__coords_vide = (self.__dim-1,self.__dim-1)
```

Afficher le plateau de jeu – À vous !

```
+--+--+--+--+
|  | 7| 4|12|
+--+--+--+--+
|15| 2|14| 9|
+--+--+--+--+
| 6| 3| 5|13|
+--+--+--+--+
| 8|11|10| 1|
+--+--+--+--+
```

- construire une ligne de traits `+--+--` ...
- construire une ligne du plateau `|  | 7| 4` ...
- représenter tout le plateau dans une seule chaîne de caractères

Afficher le plateau (1)

```
1     def __str__(self):
2         '''Taquin -> str
3         retourne le plateau sous forme texte.
4         '''
5         mess=''
6         for i in range(self.__dim):
7             mess += self.ligne_traits()
8             mess += self.ligne_plateau(i)
9         mess += self.ligne_traits()
10        return mess
11
12    def ligne_traits(self) :
13        '''Taquin -> str
14        retourne une ligne de traits.
15        '''
16        res = ''
17        for i in range(self.__dim) :
18            res += '+--'
19        res += '+\n'
20        return res
```

Afficher le plateau (2)

```
1     def ligne_plateau(self,lig) :
2         '''Taquin, int -> str
3         retourne la lig-ieme ligne du plateau
4         sous forme textuelle.
5         '''
6         res = ''
7         for i in range(self.__dim) :
8             res += '|'
9             if self.__plateau[lig][i] == -1 :
10                res += "  "
11            elif self.__plateau[lig][i] < 10 :
12                res += " "+str(self.__plateau[lig][i])
13            else :
14                res += str(self.__plateau[lig][i])
15        res +="|\n"
16        return res
```

Bouger une case – À vous !

```
+--+--+--+--+
| 4| 7|13| 2|      taq.bouge_case(10)
+--+--+--+--+
|11|  |10|12|
+--+--+--+--+
| 1|14|15| 5|
+--+--+--+--+
| 6| 8| 9| 3|
+--+--+--+--+
```

Quelles sont les étapes pour `bouge_case(val)` ?

Bouger une case – À vous !

```
+--+--+--+--+
| 4| 7|13| 2|      taq.bouge_case(10)
+--+--+--+--+
|11|  |10|12|
+--+--+--+--+
| 1|14|15| 5|
+--+--+--+--+
| 6| 8| 9| 3|
+--+--+--+--+
```

Quelles sont les étapes pour `bouge_case(val)` ?

- récupérer les coordonnées de `val` si `val` est voisin du vide
  - récupérer les coordonnées des voisins du vide
  - regarder si `val` en fait partie, et retourner ses coordonnées
- si possible, permuter dans le plateau la valeur du vide et de `val`

Bouger une case

```
1     def bouge_case(self,val):
2         '''Taquin (modif), int -> boolean
3         bouge le palet qui contient la valeur val
4         si c'est possible et retourne Vrai.
5         si le mouvement n'a pas pu avoir lieu,
6         retourne Faux. '''
7         lig,col = self.coords(val)
8         if (lig,col) == (-1,-1) :
9             return False
10        self.permute_case_vide(lig,col)
11        return True
```

Les coordonnées des voisins du vide

```
1     def voisins_vide(self):
2         '''Taquin -> list((int,int))
3         retourne la liste des coordonnees des voisins
4         du vide.
5         '''
6         res = []
7         l_vide, c_vide = self.__coords_vide
8         if l_vide > 0 :
9             res.append((l_vide-1,c_vide))
10        if l_vide < self.__dim - 1 :
11            res.append((l_vide+1,c_vide))
12        if c_vide > 0 :
13            res.append((l_vide,c_vide-1))
14        if c_vide < self.__dim - 1 :
15            res.append((l_vide,c_vide+1))
16        return res
```

Les coordonnées de val

. . . si `val` est un voisin du vide, `(-1,-1)` sinon . . .

```
1     def coords(self,val):
2         '''Taquin, int -> (int, int)
3         '''
4         voisins_valides = self.voisins_vide()
5         for (lig,col) in voisins_valides :
6             if self.__plateau[lig][col] == val :
7                 return lig,col
8         return -1,-1
```

Permuter une case et le vide

```
1     def permute_case_vide(self,lig,col):
2         '''Taquin (modif), int -> boolean
3         permute le palet (lig,col) avec le vide
4         '''
5         l_vide, c_vide = self.__coords_vide
6         self.__plateau[l_vide][c_vide] =
7                                 self.__plateau[lig][col]
8         self.__plateau[lig][col] = -1
9         self.__coords_vide = lig,col
```

(Ré-)Initialiser une partie – À vous !

- si on mélange les palets et qu'on les pose aléatoirement sur le plateau $\Rightarrow$ on n'est pas sûr de pouvoir arriver à la position de fin de partie

(Ré-)Initialiser une partie – À vous !

- si on mélange les palets et qu'on les pose aléatoirement sur le plateau $\Rightarrow$ on n'est pas sûr de pouvoir arriver à la position de fin de partie
- toutes les situations possibles de départ ne permettent pas d'arriver à la situation de fin de partie !

(Ré-)Initialiser une partie – À vous !

- si on mélange les palets et qu'on les pose aléatoirement sur le plateau $\Rightarrow$ on n'est pas sûr de pouvoir arriver à la position de fin de partie
- toutes les situations possibles de départ ne permettent pas d'arriver à la situation de fin de partie !
- Comment faire pour avoir une situation de départ qui permette d'atteindre la situation finale ?

(Ré-)Initialiser une partie – À vous !

- si on mélange les palets et qu'on les pose aléatoirement sur le plateau ⇒ on n'est pas sûr de pouvoir arriver à la position de fin de partie
- toutes les situations possibles de départ ne permettent pas d'arriver à la situation de fin de partie !
- Comment faire pour avoir une situation de départ qui permette d'atteindre la situation finale ?
- Idée : partir de la situation finale, et faire un grand nombre de mouvements aléatoires

```
Repeter un grand nombre de fois :
    - choisir aleatoirement une case voisine du vide
    - permuter la case et le vide
```

(Ré-)Initialiser une partie (2)

```
1 def reinit(self):
2     '''Taquin (modif) -> None
3     initialise une partie pour jouer.
4     '''
5     for k in range(10000):
6         candidats = self.voisins_vide()
7         ind = random.randint(0,len(candidats)-1)
8         self.permute_case_vide(candidats[ind][0],
9                                candidats[ind][1])
```

Détecter la fin de partie – À vous !

Idée :

- Sachant que le plateau contient `dim` lignes et `dim` colonnes, quelle est la valeur que devrait contenir la case aux coordonnées `(lig,col)` ?

```
     0   1   2   3
   +---+---+---+---+
0  |  1|  2|  3|  4|      dim = 4
   +---+---+---+---+
1  |  5|  6|  7|  8|
   +---+---+---+---+
2  |  9|10|11|12|
   +---+---+---+---+
3  |13|14|15|   |
   +---+---+---+---+
```

Détecter la fin de partie – À vous !

Idée :

- Sachant que le plateau contient `dim` lignes et `dim` colonnes, quelle est la valeur que devrait contenir la case aux coordonnées `(lig,col)` ?
- Réponse : `lig×dim + col + 1`

```
     0   1   2   3
   +---+---+---+---+
0  |  1|  2|  3|  4|      dim = 4
   +---+---+---+---+
1  |  5|  6|  7|  8|
   +---+---+---+---+
2  |  9|10|11|12|
   +---+---+---+---+
3  |13|14|15|   |
   +---+---+---+---+
```

Détecter la fin de partie (2)

```
1  def partie_finie(self) :
2      '''Taquin -> boolean
3      detecte si la partie est finie.
4      '''
5      for i in range(self.__dim):
6          for j in range(self.__dim):
7              if self.__plateau[i][j]
8                                  != (i*self.__dim)+j+1
9                 and not self.__est_vide(i,j):
10                 return False
11     return True
```