Université de Lille 1 SCIENCES ET TECHNOLOGIES

ufr d'IEEA
Formations en
Informatique de
Lille 1

L2 Informatique - 2016–2017

Algorithmes et Programmation 2

DS1 - 15/10/2016

durée 2h - documents non autorisés (sauf memento)

Ce devoir contient trois exercices indépendants. Sauf mention du contraire, on ne vous demande pas la documentation des fonctions que vous réaliserez.

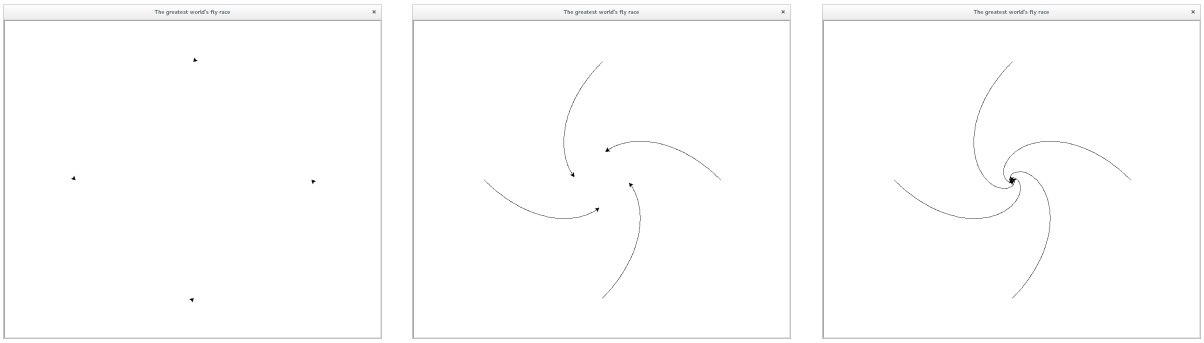
Dans les exercices 2 et 3 vous n'utiliserez pas de boucles `while` et l'usage de l'itération `for` n'est autorisé que pour la construction de listes par compréhension.

Exercice 1 *Les mouches*

Voici un exercice classique de cinématique que, rassurez-vous, on ne vous demande pas de résoudre, mais de programmer.

> Quatre mouches sont initialement situées aux quatre sommets d'un carré. On peut supposer que les mouches sont numérotées dans le sens trigonométrique de 1 à 4. Les quatre mouches volent toutes à la même vitesse en se dirigeant toujours vers la mouche voisine : la mouche numéro 1 suit la mouche numéro 2, la numéro 2 suit la numéro 3, la numéro 3 suit la numéro 4 qui suit la numéro 1. Quelle est la trajectoire de ces quatre mouches ?

Une réponse graphique est donnée sur la figure 1.



(a) Au départ (b) Intermédiaire (c) Vers la fin

FIGURE 1 – Quatre mouches se poursuivant

Un module nommé `fly` (mouche) a été conçu pour représenter des mouches. Ce module offre des fonctionnalités proches de celles du module `turtle` : déplacement (`forward`), détermination de la position courante (`pos`), détermination de la direction courante (`heading`, en degrés), détermination de la direction vers une autre mouche (`towards`), modification de la direction courante (`setheading`). Mais ces fonctions n'ont pas d'effet graphique. Voici une session utilisant ce module :

```
>>> fly1 = fly.make()
>>> fly.pos(fly1)
(0.0, 0.0)
>>> fly.heading(fly1)
0.0
```

```
>>> fly.forward(fly1, 100)
>>> fly.pos(fly1)
(100.0, 0.0)
>>> fly.setheading(fly1, 90)
>>> fly.heading(fly1)
90.0
>>> fly.forward(fly1, 100)
>>> fly.pos(fly1)
(100.0, 100.0)
>>> fly2 = fly.make()
>>> fly.towards(fly2, fly1)
45.0
>>> fly.setheading(fly1, 180)
>>> fly.forward(fly1, 100)
>>> fly.towards(fly2, fly1)
90.0
>>> fly.forward(fly1, 100)
>>> fly.towards(fly2, fly1)
135.0
>>> fly.setheading(fly1, 270)
>>> fly.forward(fly1, 100)
>>> fly.towards(fly2, fly1)
179.99999999999997
```

L'angle donné par `towards(fly2, fly1)` est celui qu'il faut à `setheading` pour que la mouche `fly2` soit dirigée vers `fly1`.

Question 1 À la lecture de cette session, quels sont d'après vous les constructeurs, sélecteurs, modificateurs sur les mouches ?

Question 2 Choisissez une représentation pour les mouches, et programmez en PYTHON les constructeurs et sélecteurs du module.

Question 3 Programmez maintenant la fonction `forward`.

Revenons maintenant au problème initial. Vous allez désormais utiliser le module `fly`.

Question 4 Programmez une fonction `distance` qui donne la distance entre deux mouches. Vous prendrez comme distance la plus grande des deux valeurs absolues des différences des coordonnées.

```
>>> distance(fly1, fly2)
100.00000000000001
```

Question 5 En supposant donnée une liste nommée `FOUR_FLIES` de quatre mouches disposées au quatre sommets d'un carré, programmez le parcours de ces mouches qui se poursuivent. Vous pourrez supposer qu'à chaque étape les mouches font un pas de longueur définie par une constante `STEP` en direction de la mouche qu'elles poursuivent. À la fin de chaque étape de l'itération, les coordonnées des quatre mouches sont imprimées. L'itération s'arrête dès que les mouches sont à distance inférieure à un seuil défini lui aussi pas une constante `THRESHOLD` strictement supérieure à `STEP`.

Exercice 2 *Tri par insertion*

Dans cet exercice on s'intéresse au tri de listes selon l'ordre défini par l'opérateur PYTHON <=.

Question 1 Réalisez une fonction récursive `insert` qui renvoie une nouvelle liste dans laquelle un élément x a été inséré parmi les éléments d'une liste triée ℓ. Accompagnez cette fonction d'une documentation avec exemples.

```
>>> insert(3, [1, 2, 4, 5])
[1, 2, 3, 4, 5]
```

Question 2 Réalisez une fonction récursive `insert_sort` qui renvoie une nouvelle liste triée contenant tous les éléments de la liste qu'on lui passe en paramètre, en suivant l'algorithme du tri par insertion. Accompagnez cette fonction d'une documentation avec exemples.

```
>>> insert_sort([3, 1, 4, 1, 5, 9, 2])
[1, 1, 2, 3, 4, 5, 9]
```

Question 3 Indiquez si vos deux fonctions sont récursives terminales ou non.

Question 4 Indiquez quelles sont les listes d'une longueur fixée n pour lesquelles le nombre de comparaisons d'éléments de ces listes pour les trier avec votre fonction `insert_sort` est maximal (pire des cas).

En notant $c(n)$ ce nombre de comparaisons pour une liste de longueur n, établissez une relation de récurrence définissant la valeur de $c(n)$ en fonction de $c(p)$ avec $p < n$.

Puis donnez une expression de $c(n)$ en fonction de n.

Comparez le pire des cas du tri insertion au pire des cas du tri rapide vu en cours.

Exercice 3 *Autour des préfixes*

Commençons par une fonction préliminaire qui pourra servir.

Question 1 Réalisez sous forme récursive un prédicat nommé `all` qui renvoie la valeur `True` si tous les éléments de la liste passée en paramètre valent `True`, et renvoie `False` dans le cas contraire.

```
>>> all([])
True
>>> all([True, True])
True
>>> all([True, False, True])
False
```

Dans tout ce qui suit on appelle *mot* toute chaîne de caractères (`str` en PYTHON). Un mot **u** est un préfixe d'un mot **v** si le mot **u** est le début du mot **v**. Ainsi les mots `a`, `ab` et `abc` sont des préfixes du mot **v** = `abc`. De plus le mot vide est un préfixe de tout mot.

Question 2 Réalisez sous forme récursive un prédicat nommé `is_prefix_of` qui renvoie `True` si le mot passé en premier paramètre est un préfixe du mot passé en second paramètre.

```
>>> is_prefix_of('', 'abc')
True
>>> is_prefix_of('a', 'abc')
True
>>> is_prefix_of('abc', 'abc')
True
>>> is_prefix_of('ac', 'abc')
False
```

Dans la suite on appelle *langage* tout ensemble de mots. Nous n'envisagerons que des langages finis qui seront représentés en PYTHON par des listes de chaînes de caractères, ces chaînes étant toutes différentes (pas de répétition de deux fois la même chaîne dans la liste).

Question 3 Réalisez un prédicat nommé `is_prefix_in` qui renvoie `True` si le mot passé en premier paramètre est préfixe d'au moins un mot du langage passé en second paramètre.

```
>>> lang = ['ab', 'abc']
>>> is_prefix_in('', lang)
True
>>> is_prefix_in('a', lang)
True
>>> is_prefix_in('abc', lang)
True
>>> is_prefix_in('ac', lang)
False
```

Un langage est dit *préfixe* si aucun mot de ce langage n'est un préfixe d'un **autre** mot de ce langage. Ainsi le langage $\{\texttt{ac}, \texttt{abc}\}$ est préfixe car aucun des deux mots n'est préfixe de l'autre. En revanche le langage $\{\texttt{ab}, \texttt{abc}\}$ n'est pas préfixe car le premier mot est un préfixe du second.

Question 4 Réalisez un prédicat nommé `is_prefix_language` qui renvoie `True` si le langage passé en paramètre est préfixe, et `False` dans le cas contraire.

```
>>> lang1 = ['ac', 'abc']
>>> lang2 = ['ab', 'abc']
>>> is_prefix_language(lang1)
True
>>> is_prefix_language(lang2)
False
```
