Exercices de Programmation & Algorithmique 1

Série 2 – Fonctions et instructions conditionnelles — Le module Droite
(29 septembre 2022)

Département d'Informatique – Faculté des Sciences – UMONS

Pré-requis : expressions booléennes; instructions conditionnelles; fonctions et fonctions booléennes (cours jusqu'au **Chapitre 4**).

Objectifs : être capable d'appeler une fonction; de créer sa propre fonction et son propre module; apprendre à décrire les étapes d'un algorithme (en français ou en pseudo-code); utiliser Python en mode script.

---

1 Le contrat

1.1 A réaliser sur papier

Lisez l'énoncé suivant et expliquez **sur papier** une solution pour le résoudre. Pour ce faire, vous pouvez décrire les différentes étapes en français ou en donner un pseudo-code.

1 Considérons les billets et pièces de valeurs suivantes : 20 euros, 10 euros, 5 euros, 2 euros, 1 euro. Écrivez une fonction `rendreMonnaie` qui prend en paramètre un entier `prix` et un 5-uple $(x_1, x_2, x_3, x_4, x_5)$ d'entiers représentant le nombre de billets et de pièces de chaque sorte que donne un client pour payer l'objet dont le prix est mentionné. x_1 correspond au nombre de billets de 20 euro, x_2 correspond au nombre de billets de 10 euro, etc. La fonction doit retourner un 5-uple représentant la somme qu'il faut rendre au client, décomposée en billets et pièces (dans le même ordre que précédemment). La décomposition doit être faite en utilisant le plus possible de billets et pièces de grosses valeurs.

1.2 A réaliser sur machine

2 Implémentez la fonction `rendreMonnaie` décrite dans la section précédente en suivant votre solution décrite sur papier.

1.2.1 Module 'Droite'

Pour cet exercice, nous allons créer une série de fonctions concernant le travail sur les droites. Nous allons placer ces fonctions dans un *module* et tester ce module grâce à des fonctions placées dans un autre fichier. Pour ce faire, vous allez donc créer deux fichiers :

- un fichier nommé `droite.py` et y placer l'implémentation des fonctions décrites ci-dessous : ce sera votre module `droite`. Pour rappel, vos fonctions peuvent se servir des autres fonctions déjà implémentées ;
- un fichier nommé `droite_test.py` qui importera votre module ("`import droite`" au début du fichier) et dont le but est de tester *chacune* des fonctions du module `droite`. Vérifiez que les tests proposés fournissent la bonne réponse. Libre à vous d'implémenter autant de tests que vous le voulez mais chaque fonction du module `droite` doit être testée au moins d'une façon.

Notations :
Nous représentons une droite d dans $\mathbb{R}^2$ par le triplet $(a, b, c) \in \mathbb{R}^3$ de telle manière à ce que :

$$d \equiv ax + by = c$$

Rappelons que (a, b) est un vecteur normal de d, et que les droites décrites par les triplets $(a, b, c) \in \mathbb{R}^3$ et $\alpha(a, b, c)$, avec $\alpha \in \mathbb{R}_0$, sont équivalentes.
De plus, nous représentons un point de $\mathbb{R}^2$ par un couple de flottants.

Implémentez les fonctions suivantes :

3 `droite(p1, p2)`
Entrée : Deux points `p1` et `p2`.
Sortie : Un triplet représentant la droite passant par `p1` et `p2`.
Tests :
```
droite((-2, 0), (1, 1.5)) → (-0.5, 1, 1.0)
droite((0, -3), (0, 5)) → (1, 0, 0)
droite((0, -1), (0, -1)) → None
```

4 `appartient(d, p)`
Entrée : Une droite `d` et un point `p`.
Sortie : *True* si `p` $\in$ `d`, *False* sinon.
Tests :
```
appartient((-0.5, 1, 1.0), (-2, 0)) → True
appartient((-0.5, 1, 1.0), (1, 1.5)) → True
appartient((-0.5, 1, 1.0), (0, 0)) → False
```

5 `paralleles(d1, d2)`
Entrée : Deux droites `d1` et `d2`.
Sortie : *True* si `d1` et `d2` sont parallèles, *False* sinon.
Tests :
```
paralleles((0, 1, 1), (0, 2, 3)) → True
paralleles((-0.5, 1, 1.0), (0, 2, 3)) → False
```

6 `intersection(d1, d2)`
Entrée : Deux droites `d1` et `d2`
Sortie : Le point d'intersection de `d1` et `d2` s'il existe, `None` sinon.
Tests :
```
intersection((-0.5, 1, 1.0), (0, 2, 3)) → (1.0, 1.5)
intersection((0, 1, 1), (0, 2, 3)) → None
```

7 `droite_normale(d, p)`
Entrée : Une droite `d` et un point `p`.
Sortie : La droite perpendiculaire à `d` passant par `p`.
Tests :
```
droite_normale((-0.5, 1, 1.0), (-2, 0)) → (2.0, 1, -4.0)
droite_normale((-0.5, 1, 1.0), (3, 4)) → (2.0, 1, 10.0)
```

8 `symetrie_orthogonale(d, p)`
Entrée : Une droite `d` et un point `p`.
Sortie : Le point qui est l'image de `p` par la symétrie orthogonale de droite `d`.
Tests :
```
symetrie_orthogonale((-0.5, 1, 1.0), (-2, 0)) → (-2.0, 0.0)
symetrie_orthogonale((-0.5, 1, 1.0), (3, 4)) → (4.200... 002, 1.5999...
996)
```

9 `distance_droite_point(d, p)`
Entrée : Une droite `d` et un point `p`.
Sortie : La distance entre `d` et `p`.
Tests :
```
distance_droite_point((-0.5, 1, 1.0), (3, 4)) → 1.3416407864998741
distance_droite_point((-0.5, 1, 1.0), (-2, 0)) → 0.0
```

1.2.2 Visualisation (facultative) de vos droites

Un module est à votre disposition pour visualiser les droites que vous manipulerez. Vous **ne devez pas modifier** ce module mais il peut être **utilisé** pour afficher à l'écran les droites créées par vos fonctions. Pour ce faire, vous devez télécharger le fichier `plot.py` disponible sur la plateforme moodle. Pour initialiser cet outil, il suffit d'entrer les lignes de commandes python suivantes :

```
from plot import *
p = Plot()
```

Le principe d'utilisation de cet outil est d'ajouter des droites, et ensuite de lui demander de les dessiner. Pour ajouter une droite à afficher il suffit d'utiliser la méthode `prepare`. Par exemple si `dro` est un triplet représentant une droite, utilisez la commande suivante : `p.prepare(dro)`
La méthode `draw` permet d'afficher les droites qui ont été ajoutées, tandis que la méthode `reset` permet de réinitialiser l'affichage graphique (effacer les droites précédemment ajoutées).
Voici un exemple complet de code pour afficher deux droites :

```
from plot import *
p = Plot()
p.prepare((1,1,3))
p.prepare((-2,1,-1))
p.show()
```

Notez bien que l'écriture de l'équation d'une droite n'est pas unique et que les valeurs proposées dans les tests ci-dessous peuvent être différentes de celles que vous obtiendrez.

2 Exercices complémentaires

★☆☆ 10 Effectuez sur papier l'analyse Top-Down de l'énoncé suivant : Écrivez une fonction qui, étant donnés deux points (x_1, y_1) et (x_2, y_2), calcule et retourne la distance euclidienne entre ces deux points.

Implémentez votre solution en suivant votre analyse Top-Down.

Modifiez ensuite l'implémentation pour que la fonction prenne en paramètre deux couples de réels au lieu de 4 réels.

★☆☆ 11 Effectuez sur papier l'analyse Top-Down de l'énoncé suivant : Étant donnés 4 points entrés par l'utilisateur, calculez et affichez le périmètre du parallélogramme correspondant. Les points $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ et (x_4, y_4) correspondent au coin supérieur gauche, au coin supérieur droit, au coin inférieur droit et au coin inférieur gauche. Vérifiez que les cotés soient bien égaux deux à deux.

★★☆ 12 Soit l'équation du second degré $\alpha x^2 + \beta x + \gamma = 0$ où α, β et γ sont entrés par l'utilisateur, écrivez un programme qui calcule et affiche les solutions (s'il y en a).

A partir de l'analyse réalisée lors de la série précédente, découpez le problème en sous-parties qui pourraient être mises sous forme de fonctions. La fonction calculant les racines retourne soit `None`, soit un couple représentant les deux racines (ou deux fois la même racine si elle est unique).

★☆☆ 13 Dans le module `random`, la fonction `randint(a,b)` retourne un nombre aléatoire compris entre a et b (inclus). Ecrivez une fonction qui génère 3 nombres aléatoires représentant 3 dés à jouer et qui retourne `True` si les dés forment un 421, `False` sinon.

★★☆ 14 Implémentez une fonction `multiple9` qui prend un paramètre entier $n \in 1, \ldots, 9$ et qui retourne le produit de n par 9 sous forme de chaîne de caractères, sans utiliser l'opérateur de multiplication ni de division.

★★☆ 15 Une fonction `duree` qui prend deux paramètres `debut` et `fin`. Ces deux paramètres sont des couples dont la première composante représente une heure et la seconde composante représente les minutes. Cette fonction doit calculer le nombre d'heures et de minutes qu'il faut pour passer de `debut` à `fin`.

Tests : `duree((14, 39), (18, 45))` → `(4, 6)`

```
duree((6, 0), (5, 15))  →  (23, 15)      ⚠ ///(0, 45)//
```

★★★ 16 Même exercice que le précédent mais sans utiliser les instructions conditionnelles.

Veuillez compléter le module `Droite` par les fonctions suivantes :

★★☆ 17 `confondues(d1, d2)`
Entrée : Deux droites `d1` et `d2`.
Sortie : *True* si `d1` et `d2` sont confondues, *False* sinon.
Tests :
```
confondues((0, 1, 1), (0, 2, 3))  →  False
confondues((0, 1, 1), (0, 2, 2))  →  True
confondues((1, 2, 1), (4, 5, 6))  →  False
```

★★☆ 18 `droite_parallele(d, p)`
Entrée : Une droite `d` et un point `p`.
Sortie : La droite parallèle à `d` passant par `p`.
Tests :
```
droite_parallele((-0.5, 1, 1.0), (3, 4))  →  (-0.5, 1, 2.5)
droite_parallele((-0.5, 1, 1.0), (-2, 0))  →  (-0.5, 1, 1.0)
```

★★☆ 19 `coefficient_angulaire(d)`
Entrée : Une droite `d`.
Sortie : Le cexercicesoefficient angulaire de la droite `d` s'il existe, `None` sinon.
Tests :
```
coefficient_angulaire((-0.5, 1, 1.0))  →  0.5
coefficient_angulaire((1, 0, 0))  →  None
```

★★☆ 20 `intersection_abscisses(d)`
Entrée : Une droite `d`.
Sortie : Le point intersection de `d` avec l'axe des abscisses s'il existe, `None` sinon.
Tests :
```
intersection_abscisses((-0.5, 1, 1.0))  →  (-2.0, 0.0)
intersection_abscisses((0, 1, 1.0))  →  None
```

★☆☆ 21 Documentez (à l'aide des commentaires multilignes) votre module `Droite` et testez l'affichage de la documentation à l'aide de `help()`.