ICC: Programmation — Miniprojet 2

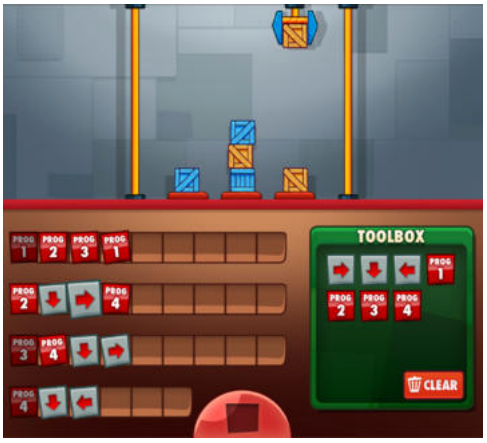
GC/MX, 22 novembre 2019

Pendant cette séance, nous allons modifier notre code pour permettre à la grue de gérer, comme dans le jeu Cargo-Bot, des listes d'instructions plutôt que simplement une instruction à la fois.

Exercice 1. Modélisation des instructions et d'un programme

(a) Pour faire simple, nous allons utiliser la convention que chaque instruction pour la grue sera modélisée dans le code par un string de 3 caractères. Dans le fichier **`cargobot.by`**, ajoutez la déclaration de 3 «constantes» (en fait, des variables normales, mais dont nous ne changerons jamais la valeur) de type **`str`**: **`GO_LEFT`**, **`GO_RIGHT`**, **`GO_DOWN_AND_UP`** (en Python, on écrit souvent les constantes tout en majuscule). Choisissez pour chacune comme valeur une représentation sous forme de string de 3 caractères.

Comme dans Cargo-Bot, nous pourrons remplir quatre sous-programmes avec des instructions. Nous aurons donc aussi des instructions spéciales qui nous permettront de sauter à un sous-programme donné (en termes Python: faire une sorte d'«appel de fonction»). Ajoutez les quatre instructions **`GO_TO_P1`**, **`GO_TO_P2`**, **`GO_TO_P3`** et **`GO_TO_P4`** et choisissez-leur également une représentation sous forme de string de 3 caractères.



Nous avons ainsi les instructions qui correspondent à la «Toolbox» représentée ci-dessus à droite.

(b) Pour modéliser un programme complet avec 4 sous-programmes P1 à P4, créez une nouvelle classe **`Program`** stockera dans 4 propriétés nommées **`P1`** à **`P4`** quatre listes d'instructions, initialement vides. En déclarant la méthode standard **`__init__`**, faites en sorte que l'argument **`P1`** doive être spécifié, mais que **`P2`** à **`P4`** soient des arguments optionnels avec la liste vide comme valeur par défaut.

(c) Ajoutez à **`Program`** le code disponible sur la page Moodle du cours, qui fournit une représentation textuelle d'un programme via **`__repr__`**. Ajoutez également en haut du fichier **`from typing import Optional`**. De retour dans votre ficher **`gui.py`**, créez une nouvelle instance de **`Program`** et insérez des instructions dans **`P1`** et **`P2`** pour que le programme, une fois affiché sur le terminal, vale ceci (ou ceci à la différence près de la représentation textuelle précise que vous aurez choisie pour chaque opération):

```
P1:   -->   -->   >P2
P2:    V    -->    V    <--   >P2
P3:  (vide)
P4:  (vide)
```

Attention à ne pas utiliser directement les strings dans la définition des sous-programmes, mais bien les constantes définies dans **`cargobot.py`**.

(d) Lisez et comprenez dans les grandes lignes ce que fait la méthode **`repr_with_highlight`** et comment.

Exercice 2. Modifications graphiques et exécution automatique d'un programme

(a) Nous allons ajouter à la fenêtre une deuxième zone de texte de type **`Text`**, qui contiendra une représentation textuelle du programme lui-même, à l'instar de ce que fait le **`Text`** que nous avons déjà pour la grue. Pour ce faire, créez un nouveau widget de ce type, stockez-le dans une variable nommée **`program_view`** et insérez-le dans la fenêtre avec un appel à la méthode **`grid`**, de façon à ce qu'il s'insère entre la représentation de la grue et la rangée de boutons en bas. Il doit se redimensionner en largeur mais pas en hauteur. Initialement, il doit afficher 4 lignes de texte.

(b) Faites en sorte que **`program_view`** contienne la représentation du programme déclaré à l'exercice précédent. Pour cela, complétez le code de la fonction **`update_view`**.

(c) L'idée est maintenant de faire exécuter automatiquement par la grue les instructions préprogrammées dans **`program`**. Pour voir ce qui se passe et ne pas exécuter les instructions à toute vitesse, nous allons faire une pause de 500 ms entre chaque instruction. Recopiez dans **`gui.py`** cette fonction, qui effectue la pause (vous aurez besoin d'importer **`sleep`** du module **`time`**):

```
def pause() -> None:
    # fait une pause de 500 ms
    sleep(0.5)
```

(d) Pour exécuter une série d'instructions, ajoutez une nouvelle fonction (pour l'instant vide: **`pass`** est une instruction qui ne fait rien):

```
def execute_program(program: Program, subprogram: List[str]) -> None:
    pass # à implémenter
```

Elle accepte deux paramètres: le premier est le programme complet, que l'on utilisera pour aller chercher les listes d'instructions **`P1`** à **`P4`** si l'on rencontre une des instructions spéciales qui nous demandent de sauter à un sous-programme; le deuxième est la liste d'instructions que l'on va exécuter maintenant. Au début, ce sera la liste **`P1`**.

Pour tester cette fonction, insérez, tout en bas de la fenêtre, un nouveau bouton sous les 3 autres appelé Run. Normalement, on ferait en sorte qu'un clic sur ce bouton cause directement l'exécution de **`P1`**, comme ceci:

```
execute_program(program, program.P1)
```

Mais, pour des raisons dont nous reparlerons dans un prochain cours, ici, nous allons faire ceci à la place, après avoir ajouté **`from threading import Thread`** en haut du fichier:

```
thread = Thread(target=lambda: execute_program(program, program.P1))
thread.start()
```

Vous pouvez considérer que ça a le même effet; nous y reviendrons plus tard.

Discussion: Pourquoi passe-t-on aussi **`program`** comme paramètre si on a déjà la liste d'instructions à exécuter comme argument de toute façon? — La liste à exécuter pourrait contenir une instruction de saut à un autre sous-programme, par exemple **`P2`**. Si l'on n'a pas le programme complet, on n'aura pas accès à la liste d'instructions contenues dans **`P2`**, et donc il nous faut de toute façon véhiculer notre **`Program`**.

(e) Implémentez la fonction **`execute_program`**. Elle doit exécuter chaque instruction contenue dans la liste passée avec le paramètre **`subprogram`**. Faites donc une boucle pour traiter toutes les instructions, et dans la boucle, vérifiez la nature de chaque instruction. Traitez-les comme suit:

— Pour les trois instructions de déplacement de la grue: (i) appelez la méthode correspondante sur **`crane`**; (ii) mettez à jour l'interface graphique en appelant la méthode **`update_view`**; (iii) faites une pause avec la méthode **`pause`**.

— Pour les quatre instructions pour sauter à un sous-programme donné: (i) faites d'abord une pause avec **`pause`**, puis (ii) appelez la méthode **`execute_program`** elle-même en passant le bon sous-programme comme argument.

(f) Finalement, nous souhaitons montrer, dans **`program_view`**, l'instruction qui est en train d'être exécutée. Il se trouve que la classe **`Program`** nous fournit la méthode **`repr_with_highlight`**, qui retourne une représentation textuelle en mettant en évidence l'instruction qui est en train d'être exécutée. Cette méthode prend deux arguments: le premier est le sous-programme qui est en train d'être exécuté, le deuxième est l'index (qui commence à 0) de l'instruction qui est en train d'être exécutée dans le sous-programme.

Essayez d'utiliser cette méthode en lui fournissant les bons arguments de façon à ce que l'instruction en cours d'exécution soit mise en évidence.

Après ces modifications, la grue lancée avec le programme de l'exercice 1 (d) est censée déplacer toutes les caisses de la troisième colonnes à la quatrième, puis de la quatrième à la troisième, et ainsi de suite, sans jamais terminer. Notez que vous pouvez toujours utiliser les boutons pour faire en sorte que la grue exécute des instructions en plus pendant l'exécution du programme.

Amusez-vous à changer les instructions de **`program`** et à observer le comportement de la grue.

Exercice 3. Fonctions d'ordre supérieur

(a) Dans un fichier à part hors dossier miniprojet, copiez-collez le code de départ depuis Moodle pour cet exercice. Ensuite, déclarez et implémentez une fonction **`filter`**, qui accepte, comme premier paramètre: une liste d'un type générique **`T`**; et comme second paramètre: une fonction de filtre (un prédicat) qui retourne un **`bool`** pour un paramètre de type **`T`**. La fonction **`filter`** doit renvoyer, dans une nouvelle liste, tous les éléments de la liste initiale pour laquelle le prédicat est vérifié.

Par exemple, ceci doit afficher **`[1, 3, 3, 5]`**:

```
def is_odd(x: int) -> bool:
    return x % 2 == 1

print(filter([0, 1, 3, 3, 4, 5], is_odd))
```

(b) Utilisez votre fonction **`filter`** pour filter la liste **`pushkin_verse`** et n'en retenir que les mots qui contiennent la lettre russe **`'е'`**.

(c) Si vous avez utilisé une déclaration de fonction traditionnelle pour (b), utilisez une fonction lambda, et vice-versa.

(d) Écrivez une fonction **`map`**, qui accepte, comme premier paramètre: une liste d'un type générique **`T`**; et comme second paramètre, une fonction qui accepte un argument de type **`T`** et retourne une valeur dérivée de type **`U`**. La fonction **`map`** retourne une liste d'éléments de type **`U`** obtenus par l'appel de la fonction passé en second paramètre sur chacune des valeurs de la liste passée en premier paramètre.

Par exemple, ceci doit afficher **`['1', '4', '9', '16']`**:

```
def square_as_string(x: int) -> str:
    return f"{x * x}"

print(map([1, 2, 3, 4], square_as_string))
```

(e) Utilisez votre fonction **`map`** sur le résultat de votre appel de (b) pour obtenir, dans une nouvelle liste, la longueur de chaque mot de la liste avec une fonction lambda.

(f) Essayez de vous passer de la fonction lambda, ceci sans devoir déclarer de fonction supplémentaire.