Langages de script (Python)
CMTP n° 6 : Mini-projet

Avec ce sujet nous nous lançons dans un mini projet qui occupera les prochaines 2 séances (*spoiler :* nous allons implémenter un interpréteur textuel de Logo avec graphisme Tortue). L'idée est de compléter un projet de bonne envergure en Python et, en le faisant, de découvrir certains outils, techniques, et abstractions typiques de la programmation en Python d'applications complexes, comme notamment : la gestion de bibliothèques externes et de tests unitaires.

Modalités de travail ce sujet est à réaliser soit seul soit en binôme. C'est à vous de choisir. Si vous choisissez de travailler en binôme, vous serez en charge de vous organiser à la fois pour former un binôme et pour collaborer à distance avec votre coéquipier.

Gestion des bibliothèques externes

Python dispose d'un écosystème des bibliothèques et paquets logiciels considérables. La façon la plus populaire de distribuer ces paquets aux utilisateurs est via PyPI, le *Python Package Index* que vous trouverez à l'adresse https://pypi.org/. Vous pouvez y chercher des paquets Python prêts à être installés sur vos machines.
Une fois identifié le(s) paquet(s) que vous souhaitez installer, vous pouvez l'installer avec l'outil `pip` (normalement pre-installé avec Python) sur la ligne de commande, avec la commande `pip install PAQUET` où `PAQUET` est le nom du paquet choisi.
Pour éviter des "polluer" votre système avec beaucoup des paquets (qui seront à mettre à jour, pourraient être incompatible entre eux, etc.) la meilleure pratique pour gérer les dépendances en Python est de créer un *environnent virtuel* (*virtualenv* en anglais) pour chaque projet que vous souhaitez réaliser. Chaque virtualenv contiendra les dépendances du projet en question et peut être utilisé et entretenu (mise à jour, installation, et suppression des paquets) indépendemment des autres virtualenv dont vous disposez. Vous pouvez aussi créer, supprimer, activer (avant de l'utiliser) et déactiver chaque virtualenv.
Vous trouverez la documentation officiel à propos de environnements virtuels et de pip à l'adresse : https://docs.python.org/3/tutorial/venv.html . Prenez un moment pour la lire avant de continuer.
Voici une session complète qui montre comment, à partir du shell (N.B : pas directement dans l'interpréteur Python !) créer un virtualenv (nommé "testenv"), l'activer, y installer une bibliothèque externe avec `pip`, et le déactiver avant de passer à autre chose :

```
# on commence par creer un virtualenv nomme "testvenv"
$ python3 -m venv testvenv

# cela creera un directory "testvenv", contenant:
# (le contenu pourrait etre different chez vous)
ls testvenv/
bin  include lib  lib64 pyvenv.cfg share

# on "active" le virtualenv
# notez le prompt du shell: il montre le nom du virtualenv actuellement actif
$ source testvenv/bin/activate
(testvenv) $

# essayons d'utiliser la bibliotheque requests, pour effectuer des requetes HTTP,
# qui ne fait pas partie de la bibliotheque standard de Python
$ python
Python 3.9.1+ (default, Feb 5 2021, 13:46:56)
>>> import requests
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'requests'
>>> exit()
```

```
# normal, elle n'est pas installe, donc on l'installe avec pip
(testvenv) $ pip install requests
Collecting requests
  Using cached requests-2.25.1-py2.py3-none-any.whl (61 kB)
Collecting urllib3<1.27,>=1.21.1
  Using cached urllib3-1.26.3-py2.py3-none-any.whl (137 kB)
Collecting certifi>=2017.4.17
  Using cached certifi-2020.12.5-py2.py3-none-any.whl (147 kB)
Collecting idna<3,>=2.5
  Using cached idna-2.10-py2.py3-none-any.whl (58 kB)
Collecting chardet<5,>=3.0.2
  Using cached chardet-4.0.0-py2.py3-none-any.whl (178 kB)
Installing collected packages: urllib3, certifi, idna, chardet, requests
Successfully installed certifi-2020.12.5 chardet-4.0.0 idna-2.10 requests-2.25.1 urllib3-1.26.3
(testvenv) $

# A la fois srequests et ses dependances on ete installees. Essayons a nouveau:
(testvenv) $ python
Python 3.9.1+ (default, Feb 5 2021, 13:46:56)
>>> import requests
>>> r = requests.get("http://lemonde.fr")
>>> r.status_code
200
>>> r.headers["Content-Type"]
'text/html; charset=UTF-8'
>>> exit()

# On sort maintenant du virtualenv de test (notez que le prompt du shell redevient normal).
# On peut aussi le supprimer completement si necessaire.
(testenv) $ deactivate
$ rm -rf testenv/
$
# N.B: votre *code* ne doit pas resider lui meme dans le virtualenv, qui contiendra seulement
# vos dependances logicielles. Votre code peut se trouver n'importe ou, il faut juste
# l'executer *apres* avoir active le bon virtualenv.
```

Exercice 1 : Installation et utilisation des bibliothèques externes

- Créez un nouveau virtualenv nommé "langscript" et, comme dans l'exemple ci-dessus, installez à son intérieur la bibliothèque `requests`. Consultez la documentation de la bibliothèque `requests` sur pypi, à partir de la page https://pypi.org/project/requests/.
- En utilisant `requests` écrivez un programme exécutable `check_website.py` qui prend comme argument sur la ligne de commande un URL de site web est affiche sur la sortie standard soit un message "website URL is up and running", soit un message "website URL appears to be down :-(" selon si le site répond correctement ou pas. (Astuce : un site web a répondu correctement si son status code est un entier dont "2xx". Vous pouvez aussi utiliser la méthode `.raise_for_status()` sur les objets retournés par `request.get()` pour lancer une exception en cas de status code qui indique une erreur. Toutes les exceptions qui `requests` peut soulever héritent de `requests.exceptions.RequestException`.)
- Essayez d'exécuter votre programme avec le virtualenv "langscript" actif et non actif. Que se passe-t-il dans les deux cas ?

Tests unitaires

Aucun projet logiciel "sérieux" serait considéré aujourd'hui comme prêt à être utilisé en production et facile à maintenir sans un jeu des tests qui l'accompagne. Avant de se lancer dans l'implémentation d'un vrais projet, voyons donc comment effectuer des *tests unitaires automatisés* en Python.

Plusieurs *framework* de support à différent parties de l'activité de testing logiciel existe un Python. Les plus populaires sont `unittest`,[1] `pytest`,[2] `doctest`,[3] `hypothesis`,[4] et `unittest.mock`.[5]
Pour ce sujet nous allons utiliser `pytest`, donc créez (ou activez) un virtualenv et installez-y ce framework de test (avec `pip install pytest`). Vous pouvez consulter la documentation de pytest à l'adresse https://docs.pytest.org/. Voici un `"Hello, World!"` en test unitaire :

```
# content of the file: add.py

def add(x, y):
    """Add two numbers"""
    return x - y # BUG!


def test_add_integers():
    """Test if addition works"""
    assert add(40, 2) == 42
```

Le fichier `add.py` contient à la fois l'implémentation de votre code métier (`add()`) et un test unitaire associé (`test_add_integers()`). Le test utilise l'instruction `assert` de Python qui vérifie que, pendant l'exécution, une expression booléenne soit vraie. Si cela n'est pas le cas une exception `AssertionError` sera lancée et, dans le cas d'exécution avec pytest, le test sera considéré un échec. Pour exécuter le test on exécutera `pytest add.py` :

```
$ pytest add.py
add.py F       [100%]
======== FAILURES ========
-------- test_add --------

    def test_add():
        "test if the addition works"
>       assert add(40, 2) == 42
E       assert 38 == 42
E        +  where 38 = add(40, 2)

add.py:10: AssertionError
======== short test summary info ========
FAILED add.py::test_add - assert 38 == 42
======== 1 failed in 0.08s ========
```

Pytest a trouvé un seul test à executer dans le fichier `add.py` (l'heuristique utilisé est de considerer toutes les fonctions dont le nom commence par `test_` comme étant des tests unitaires), il a executé 100% de (un seul) tests, et a obtenue 100% d'échecs. Pour chaque échec un diagnostique détaillé est fourni, avec la raison de l'erreur avec un pointeur dans le code vers l'origine de l'échec.
En générale on préfère séparer le code métier du code de tests, dans des fichiers séparés, p.ex. :

```
# content of the file: arith.py

def add(x, y):
    """Add two integers"""
    return x + y

def div(x, y):
    """Divide x by y.
    Will raise ZeroDivisionError if y is 0.
    """
    return x // y
```

1. https://docs.python.org/3/library/unittest.html
2. https://docs.pytest.org/
3. https://docs.python.org/3/library/doctest.html
4. https://hypothesis.readthedocs.io/
5. https://docs.python.org/3/library/unittest.mock.html

```python
# content of the file: test_arith.py

import pytest
from arith import add, div

def test_add_positive():
    assert add(40, 2) == 42

def test_add_negative():
    assert add(40, -2) == 38

def test_div_multiple():
    assert div(10, 5) == 2

def test_div_remainder():
    assert div(10, 3) == 3

def test_div_byzero():
    # use the pytest.raises() context manager to verify that tested code raises an
    # exception, when *it should* do so
    with pytest.raises(ZeroDivisionError):
        div(7, 0)
```

Noter l'utilisation du *context manager* `pytest.raises()` pour vérifier que le code testé lance une exception, quand il est censé le faire, sans mettre le test en échec.
Une fois bien séparé votre code entre code métier et tests unitaires (ces dernier dans des fichiers Python nommés `test_*.py`, et qui contient chaque test case dans des fonctions nommés `def test_*()`), vous pourrez exécuter tous vos tests unitaires d'un coup en exécutant pytest sur *le directory* qui contient votre code, p.ex. :

```
$ pytest solution/unittest/
======== test session starts ========
collected 5 items
solution/unittest/test_arith.py .....    [100%]
======== 5 passed in 0.02s ========
```

Cette fois-ci pytest a trouvé 5 tests unitaires dans le directory `solution/unittest/`, tous dans le fichier `test_arith.py`; 100% des ces tests sont passés.

Exercice 2 : Listes triées... et testées

Un type abstrait (ou ADT, pour *Abstract Data Type* en anglais) est un type de données défini en termes des fonctions qui le manipule, sans se soucier de sa représentation concrète en mémoire.

- Implémentez, dans un module Python `slist.py` le type abstrait *liste triée d'entiers* (ou `slist` pour *sorted list*), en écrivant les fonctions suivantes :
  - `def init()` : crée une slist vide et la retourne
  - `def init(iterable)` : variante de `init()` qui consomme un iterable et initialise une liste triée avec son contenu
  - `def length(slist)` : retourne la longueur de la liste triée slist
  - `def contains(slist, elt)` : vérifie si `elt` est présent dans la liste triée `slist` et retourne un booléen pour l'indiquer
  - `def add(slist, elt)` : ajoute l'entier `elt` à la liste triée `slist` en maintenant l'invariant que la représentation interne de la slist soit trié (dans un ordre quelconque). *Retourne* la nouvelle liste (*attention :* pas de *mutation* de liste, l'interface de slist est complètement fonctionnelle)
  - `def join(slist1, slist2)` : fusionne les listes triées `slist1` et `slist2`, et retourne la nouvelle liste (triée)
  - `def remove(slist, elt)` : supprime `elt` de la liste triée `slist` et retourne la nouvelle liste triée. Si `elt` n'est pas présent dans `slist`, lance une exception `ValueError`
- Implémentez, dans un module Python `test_slist.py` un jeu de test complet pour votre implémentation du type abstrait `slist`. Veillez à bien tester le plus des cas et de situations possibles, y compris les cas exceptionnels

– Vérifiez la couverture de code (*code coverage*) de vos tests sur votre implémentation de slist, c.a.d., le taux des lignes de code de votre implémentation exécutées pendant l'exécution des tous les tests unitaires. Pour le faire installez et utilisez la bibliothèque `pytest-cov`.[6] Avez vous atteint le 100% de code coverage ?

Un interpréteur Logo

Nous souhaitons réaliser un interpréteur batch pour le langage LsLogo, défini dans la suite. LsLogo est un langage de programmation graphique qui permet de faire déplacer une tortue dans un canevas rectangulaire, en dessinant.

Canevas Le taille du canevas est déterminée au démarrage de l'interpréteur LsLogo, et normalement consiste en LINES lignes × COLS colonnes, où LINES et COLS correspondent aux dimensions du terminal dans lequel le programme tourne. Pour identifier la position dans le canevas on utilisera les coordonnées (l, c) où $l \in \{0, \ldots, \text{LINES} - 1\}$ et $c \in \{0, \ldots, \text{COLS} - 1\}$. La position (0,0) se trouve en haut à gauche du terminal ; les numéros de ligne augmentent vers le bas de l'écran, les numéros de colonnes vers la droite.

Tortue L'état de la tortue est un tuple $\langle pos, \text{dir}, \text{pen}, \text{color} \rangle$ ou *pos* est la position de la tortue dans le canevas, *dir* la direction vers laquelle elle est orientée (un angle entre 0 et 360, prenant comme valuer que des multiples de 45 dégrées, où la direction 0 pointe vers droite et les valeurs augmentent en sens antihoraire), et *pen* est un booléen (le stylo de la tortue touche par terre—et donc la tortue écrit quand elle se déplace—si et seulement si `pen == True`), et *color* un caractère Unicode. Au départ, la tortue se trouve en position (0,0), elle regarde vers droite (angle=0), son stylo est levé, et son "couleur" est le caractère espace.

Instructions Un programme en LsLogo est une séquence d'instructions localisées, une par ligne, dans la forme : `POS STATEMENT`, où `POS` est un entier (p.ex., 40) qui dénote la position de l'instruction (et permet de la référencer à l'intérieur du programme) et `STATEMENT` une instruction ; `POS` et `STATEMENT` sont séparés par un ou plusieurs espaces. Les commentaires sont aussi autorisés en LsLogo, soit dans des lignes qui commences par "//", soit *inline* après des vrais instructions qui terminent avec une séquence d'espace et un commentaire.
Un programme LsLogo valide doit toujours avoir une instruction avec position 10, qui est la première instruction exécutée au lancement du programme.
Les instructions possibles sont les suivantes :

NAME* = *EXPR (affectation de variable) où NAME est un identifiant formé seulement par des lettres minuscules, et EXPR une *expression*. Une expression en LsLogo est soit un entier, soit un nom de variable (= un identifiant), dont la valeur actuelle est récupérée et affectée à la variable NAME. Certains variables sont prédéfinies en LsLogo :

width largeur du canevas, en colonnes

height hauteur du canevas, en lignes

xpos cordonnée X de la position courante de la tortue $\in \{0, \ldots \text{width} - 1\}$

ypos cordonnée Y de la position courante de la tortue $\in \{0, \ldots \text{height} - 1\}$

Essayer d'affecter les variables prédéfinies cause une erreur et l'arrêt de l'exécution du programme.

FORWARD *N* avance de N cases dans la direction courante de la tortue, où N est une expression. Si la tortue arrive aux bords du canevas en se déplaçant, elle continuera en rentrant du coté opposé du canevas (*wrap around*). À noter : si N est négatif la tortue *reculera* à la place d'avancer, mais ne changera pas sa direction.

ROTATE *D* tourne la tortue de D dégrées, sans la déplacer. D est une expression qui doit correspondre à un entier multiple de 45. Si cela n'est pas le cas, le programme s'arrête avec une erreur.

PEN DOWN pose le stylo de la tortue

PEN UP lève le stylo de la tortue

6. https://pypi.org/project/pytest-cov/

COLOR *C* où C est un caractère Unicode (on n'utilisera pas des vrais couleur, mais il y a plein des caractères sympa[7] à utiliser à leur place !).

Quand la tortue se déplace, si le stylo est posé, elle laisse *derrière soi*, i.e., dans toutes les cases sur son chemin, y compris la case de départ mais *à l'exception de la case d'arrivée* une trace de case remplies avec le caractère correspondant à la "couleur" actuelle.

NAME* += *EXPR incrément (ou décrément, si la valeur de EXPR est négative) le contenu d'une variable. Si NAME n'a jamais été affectée avant, le programme s'arrête avec une erreur.

GOTO *POS* saute à l'exécution de la ligne du programme étiquetée par la valeur correspondante à POS, ou POS est une expression. Si aucune ligne du programme a comme étiquette la valeur correspondante à POS, le programme s'arrête avec une erreur.

GOTONZ *POS EXPR* comme GOTO, mais saute à la ligne dénotée par POS seulement si l'expression EXPR ne vaut pas 0. Si EXPR vaut 0, l'exécution passe à la ligne suivante GOTONZ

Exemples Voici un premier programme en LsLogo, qui affiche un carré de $ autour de l'écran :

```
// let's draw a $ border around the screen !
10 xsteps = width
11 xsteps += -1
12 ysteps = height
13 ysteps += -1
20 PEN DOWN
21 COLOR $
30 FORWARD xsteps  // top border
40 ROTATE -90
50 FORWARD ysteps  // right border
60 ROTATE -90
70 FORWARD xsteps  // bottom border
80 ROTATE -90
90 FORWARD ysteps  // left border
// all done
91 PEN UP
```

Voici un programme qui dessine une grande X de hauteur 10 (sans se soucier de la taille du canevas...) :

```
10  size = 10
11  PEN DOWN
12  COLOR \
// we start at the top-left of the "X", facing right
// draw the "\" part of the "X"
20  ROTATE -45
30  FORWARD size
// we are now at bottom-right, go to bottom-left
60  ROTATE -135
61 PEN UP
70  FORWARD size
// we are now at bottom-left, facing left
// go up 1 row, because we draw behind the turtle
71 ROTATE -90
72 FORWARD 1
// draw the "/" part of the "X"
80  ROTATE -45
91 PEN DOWN
92 COLOR /
100 FORWARD size
// all done
130 PEN UP
```

7. https://en.wikipedia.org/wiki/List_of_Unicode_characters

Voici un programme un brin plus complexe, qui dessine un tourbillon de @ :

```
//  draw a (squared) swirl
10  size = 10
11  length = 1
//  go far enough from the borders (assuming canvas is at least 15x15)
20  ROTATE 45
30  FORWARD 7
40  ROTATE -45
50  COLOR @
60  PEN DOWN
//  main loop, draw a side of the "swirl", then rotate
100 FORWARD length
110 ROTATE 90
120 size += -1
130 length += 1
140 GOTONZ 100 size
// all done
200 PEN UP
```

. . . ou encore sa version en boucle infinie, avec *wrap around* les bords du canevas :

```
//  Draw a swirl. Forever.
10  length = 1
20  COLOR *
30  PEN DOWN
//  main loop, draw a side of the "swirl", then rotate
100 FORWARD length
110 ROTATE 90
120 length += 1
130 GOTO 100
```

Exercice 3 : Interpréteur pour LsLogo

Implémentez un interpréteur pour LsLogo. Il doit être possible de lancer l'interpréteur sur la ligne de commande avec la syntaxe `lslogo.py LINES COLS PROGRAM` où LINES et COLS indiquent la taille du canevas et PROGRAM le nom d'un fichier textuel contenant un programme en LsLogo. (Astuce : vous pouvez passer la taille courante du terminal à votre interpréteur via les variables shell $LINES et $COLUMNS.)

Votre interpréteur aura un "clock" qui affiche à chaque "tick" l'état du canevas après l'exécution d'une instruction. Pour éviter que l'exécution se passe trop vite pour pouvoir voir quelque chose, il sera probablement nécessaire soit de attendre un retour à la ligne après chaque instruction (p.ex., avec `input()`) soit d'insérer des délais entre instructions (p.ex., avec `time.sleep()`).

Aucune bibliothèque graphique devrait être nécessaire pour l'affichage, vous pouvez afficher chaque ligne du canevas avec `print()`. Pour afficher la tortue et ça direction, vous pouvez utiliser des caractères Unicode, p.ex., des flèches orientées dans les 8 directions possibles. [8] (Pensez donc à empêcher l'utilisation des mêmes caractères comme argument pour l'instruction COLOR.)

Veillez à bien découper votre code en module Python séparés. P.ex., vous pourriez avoir les fichiers suivants, dans un directory `lslogo/` :

lslogo.py pilote de l'interpréteur, script exécutable

parser.py parsing du langage

state.py état du programme

interpreter.py évaluation des instructions

ui.py affichage

8. https://en.wikipedia.org/wiki/List_of_Unicode_characters#Arrows

. . . il s'agit juste d'une proposition, c'est à vous de implémenter *et concevoir* votre code comme bon vous semble. Une proposition de `state.py` est disponible sur Moodle, comme inspiration ou à réutiliser (à vous de choisir !).

Exercice 4 : Tests unitaires pour l'interpréteur LsLogo

Votre implémentation de LsLogo doit être accompagnée par un jeu de tests unitaires qui offrent une couverture de code de votre implémentation d'au moins 80% sur les lignes de code métier. Implémentez un tel jeu de test avec pytest, en stockant les tests dans des fichiers `test_*.py`. Vérifiez avec pytest-cov que la couverture de code soit au rendez-vous.
À noter : vous êtes libre de choisir le processus de développement que vous préférez, vous pouvez donc implémenter vos tests unitaires avant, pendant, ou après l'implémentation du code métier de l'exercice précèdent.

Exercice 5 : Analyseur statique pour LsLogo

Certains classes d'erreurs dans un programme LsLogo peuvent facilement être détectées statiquement, sans exécuter le programme. Écrivez un *analyseur statique* des programmes en LsLogo, capable de détecter au moins les erreurs suivants :

- erreurs de syntaxe
- manque de l'instruction initiale avec position 10
- réutilisation des positions des instructions
- affectation des variables prédéfinie
- rotation d'angles qui ne sont pas multiples de 45
- incrément de variables non affectées
- usage des variables non affectées dans des expressions
- GOTO/GOTONZ vers instructions non présent dans le programme

Attention : pour certaines classes d'erreurs vous pourrez en empêcher toutes les instances, pour d'autre non (ou pas facilement. . .).

Exercice 6 : Embellisseur de code pour LsLogo

Écrivez des programmes intéressant en LsLogo et exécutez les avec votre interpréteur (et utilisez-les aussi comme cas pour votre jeu de tests !). En écrivant en LsLogo, vous vous apercevrez que la maintenance des numéros des instructions peut devenir assez frustrant.
Implémentez donc un *embellisseur de code* qui lit un programme LsLogo et écrit un programme sémantiquement équivalent, mais syntactiquement plus propre, dans lequel notamment :

- les numéros des instructions commence de 10 et sont espacés de 10 (*Attention :* cela nécessite de renuméroter les instructions, en le faisant il faut préserver la sémantique de GOTO/GOTONZ !)
- la colonne de début de chaque instruction (après sa position) est la même dans tout le programme

Pour les plus motivés, voici des extensions possibles (et optionnelles) pour LsLogo et votre interpréteur du langage :

Exercice 7 : Expression arithmétiques (optionnel)

Les valeurs en LsLogo sont assez limités, ils peuvent être soit des entiers littéraux soit des noms de variables. Ajoutez au langage le support pour les expressions arithmétiques : addition, soustraction, multiplication et division.
Est-ce que cette extension vous permet d'écrire des programmes LsLogo plus expressifs ? De quoi d'autre auriez vous besoin pour pouvoir dessiner des cercles ?

Exercice 8 : Interface graphique (optionnel)

La bibliothèque standard de Python dispose d'un module pour afficher sur un canevas graphique du *turtle graphics* : https://docs.python.org/3/library/turtle.html Utilisez le pour ajouter une interface utilisateur graphique. Vous pourrez ensuite retirer du langage la restriction que les "couleurs" sont des caractères et que les rotations soient de multiple de 45 dégrées.