TD, TP et projets

lucas.bourneuf@inria.fr

1 TD — Structures de contrôle et erreurs

L'objectif de ce TD est d'une part de bien comprendre les structures de contrôle, et les messages d'erreur de l'interpréteur CPython, de manière à assimiler le principe d'algorithmique, et d'apprendre à lire les messages d'erreurs.
La figure 1 n'utilise que des concepts de base valides dans tous les langages impératifs. La figure 2 utilise des concepts spécifiques à Python. Les tables 3 et 4 s'intéressent aux erreurs.

Abstenez-vous d'utiliser un ordinateur pour répondre ; ces notions sont trop importantes pour être laissées à quelqu'un d'autre que vous. N'utilisez un ordinateur pour tester vos réponses, qu'une fois la totalité du TD traitée.

Les codes sont écris en **Python 3**. Autrement dit, `print` est une fonction, et donc écrite avec des parenthèses, et la fonction `input` retourne une chaîne de caractère correspondant à une ligne lue dans l'entrée standard (le clavier). En dinopython, il faudrait ajouter les lignes `from __future__ import print_function` et `input = raw_input` pour rendre ces codes fonctionnels.

```
a, b, c = 10, 5, 1
a + b
a - c
print(a, b, c)
a += b
b = a - b
a -= b
print(a, b, c)
```

(a)

```
a, b, c = True, False, True
if a and b:
    if c or a:
        print(1)
    elif c:
        print(2)
    elif c or c:
        print(3)
    else:
        print(4)
if (a or b) and (b or c):
    if b:
        print(5)
    elif a:
        print(6)
else:
    print(7)
```

(b)

```
a = 10
while a > 0.5:
    a -= 2
    print(a)
    a -= 1
```

(c)

```
p, q = 10, 20
while p:
    p -= 1
    q -= -1
print(q)
```

(d)

```
a = 10
while a != 0:
    a -= 1
    a == 0
    print(a)
print(a)
```

(e)

```
l = [1, 2, 3, 4, 5]
for e in l:
    print(e, l)
    if e % 2:
        print(e * 2)
    elif not e or e:
        print(e ** 2)
```

(f)

```
a = 0
a += 1
b = a + 1
print(a + b)
```

(g)

Figure 1: Principes de base de programmation : si le code est valide, indiquer quelles sont les valeurs imprimées en sorties standard. Si le code n'est pas valide, indiquer quelle erreur est levée, ce qu'elle signifie, quelle instruction l'a causée, et proposer une correction.

```python
a, b, c = [None] * 3
print(a, b, c)
```

(a)

```python
def ink_spot(n):
    return ("I don't want to set the "
            + str(n) + " on fire")
words = ('banana', 'cthulhu', 'world')
for word in words:
    print(ink_spot(word))
```

(b)

```python
p, q, r = True, False, None
print(any((p and q, p or q, p or r)))
print(all((p or q, p or r, r or q)))
```

(c)

VIDE: A REMPLIR

```python
print(len({'a': 'bb', 'bb':'cc'}['a']))
```

(d)

```python
a = [3, 2, 1]
print(a.pop() + a.pop() * a.pop())
```

```python
ps = [i % 2 for i in range(0, 20, 4)]
print(True and 'all(p == 0 for p in ps)')
```

(e)

```python
if (i for i in range(2)): print(1)
if any(i for i in range(2)): print(2)
if all(i for i in range(2)): print(3)
```

(f)

```python
def f(n):
    print(n)
    return n
print(f(1), f(2), f(3))
```

(g)

```python
a = [1]
b = a
b.append(2)
print(a, b)
```

(h)

```python
def f(l=[]):
    l.append(1)
    return l
print(f(), f(), f())
```

(i)

```python
a, b, c = [1, 2, 3]
print(a, b, c, sep='#', end='4\n')
```

(j)

```python
b = set()
while len(b) < 2:
    b.add(1)
print(len(b))
```

(k)

Figure 2: Principes de Python : si le code est valide, indiquer quelles sont les valeurs imprimées en sorties standard. Si le code n'est pas valide, indiquer quelle erreur est levée, ce qu'elle signifie, quelle instruction l'a causée, et proposer une correction.

a	`a = b`	`File "<stdin>", line 1, in <module>`
		`  a = b`
		`NameError: name 'b' is not defined`

b	`if 10 = 8:`
	`  10 + 8`
		`File "<stdin>", line 1`
		`  if 10 = 8:`
		`  ^`
		`SyntaxError: invalid syntax`

c	`assert 48 == '0'`	`File "<stdin>", line 1, in <module>` `AssertionError`

d	`2 * (3 - 4`	`File "<stdin>", line 2`
		`  ^`
		`SyntaxError: unexpected EOF while parsing`

e	`msc = {'f': 't'}` `msc['t']`	`File "<stdin>", line 1, in <module>` `KeyError: 't'`
f	`msc = {'f': 't'}` `{msc: 19}`	`File "<stdin>", line 2, in <module>` `TypeError: unhashable type: 'dict'`

g	`if True:`
	`  pass`
	`pass`
	`else:`
	`  pass`
		`File "<stdin>", line 4`
		`  else:`
		`  ^`
		`SyntaxError: invalid syntax`

h	`(3)[0]`	`File "<stdin>", line 1, in <module>`
		`  (3)[0]`
		`TypeError: 'int' object is not subscriptable`

i	`sum(1, 2, 3, 4)`	`File "<stdin>", line 1, in <module>` `TypeError: sum expected at most 2 arguments, got 4`
j	`if 9 in 899:` `  pass`	`File "<stdin>", line 1, in <module>` `TypeError: argument of type 'int' is not iterable`
k	`''.join((1, 2))`	`File "<stdin>", line 1, in <module>` `TypeError: sequence item 0: expected str instance, int found`
l	`10 + '10'`	`File "<stdin>", line 1, in <module>` `TypeError: unsupported operand type(s) for +: 'int' and 'str'`

Figure 3: Ne plus avoir peur des erreurs : indiquer quelle erreur est levée, ce qu'elle signifie, quelle instruction l'a causée et pourquoi, et proposer une correction qui préserve le comportement du programme.

```
if 'hal' = 'ibm':
    print('kbk')
```

(a)

```
scores = {'diphonic': 10, 'yoddle': 8.45}
print(sum(scores.values()), scores['grunt'])
```

(b)

```
nb_user = input("Gimme an integer before midnight: ")
print(10 + nb_user)
```

(c)

```
d = {}
for idx in range(0, 10, 3):
    d[idx] = idx**5
d[6]
d[2]
d[10]
```

(d)

```
elif True:
    pass
```

(e)

```
print(f, u, n, k,y)
```

(f)

```
d = int(input('str'))
str(d) * 4 + d
```

(g)

Figure 4: Détection des erreurs : si le code lève une erreur, indiquer quelle partie est problématique, quel problème est rencontré par l'interpréteur, et éventuellement prédire l'erreur levée.

2 TP — Introduction à la librairie standard de Python

L'objectif de ce TP est de découvrir quelques fonctions built-in, ainsi que trois modules particulièrement utiles en Python, présents dans la librairie standard.

La librairie standard de python est gigantesque et extrêmement riche, donnant au langage la propriété *battery included* qui est une des motivation de l'adoption du langage dans de nombreux milieux, y compris scientifiques. Ici, nous nous concentrerons sur trois modules que les programmeurs initiés utilisent presque tous les jours.

2.1 Built-in

Une partie des built-in de python sont déjà connus, comme par exemple `print`, `input`, `open` ou `range`. Cependant, bien d'autres sont importants, incluant notamment `enumerate`, `zip`, `min`, `max`, `set`, `all` ou `any`.

La documentation officielle : `https://docs.python.org/3/library/functions.html`

Simplifiez les programmes de la figure 5. Si besoin, considérez que `a = [1, 2, 3, 4, 5]` et `b = [6, 7, 8, 9, 10, 11]`. Si vous ne comprenez pas un programme, essayez avec d'autres valeurs, comme `[1, 1, 1, 3, 1, 1]`. Lorsque nécessaire, proposez de meilleurs noms pour les variables.

a

```
acc = 0
for val in b:
    acc += val
print(acc)
```

b

```
idx = 0
while idx < len(a):
    print(idx, a[idx])
    idx += 1
```

c

```
idx = 0
while idx < len(a) and idx < len(b):
    print(a[idx], b[idx])
    idx += 1
```

d

```
import math
maximum = -math.inf
for val in a:
    if maximum < val:
        maximum = val
print(maximum)
```

e

```
for idx, elem in list(enumerate(a)):
    for elem2 in list(a[idx+1:]):
        if elem2 == elem:
            a.remove(elem)
print(a)
```

f

```
a = [1, 1, 1, 3, 1, 1]
c = []
for idx, elem in list(enumerate(a)):
    double = False
    for elem2 in list(a[idx+1:]):
        if elem2 == elem:
            double = True
    if not double:
        c.append(elem)
a = c
print(a)
```

g

```
c = [0, 1, 0, 0, 0, 1, 1]
ok = True
for elem in c:
    ok = ok and elem
    if not ok:
        break
print(ok)
```

h

```
c = [0, 1, 0, 0, 0, 1, 1]
ok = False
for elem in c:
    ok = ok or elem
    if ok:
        break
print(ok)
```

i

```
size = 0
for elem in a:
    size += 1
print(size)
```

j

```
c = []
for elem in a:
    c.insert(0, elem)
print(c)
```

Figure 5

2.2 *itertools*

Itertools, comme son nom l'indique, regroupe des fonctions travaillant sur le parcours de conteneurs, et propose, en plus de fonction au comportement très particuliers, des fonction qui étendent/généralisent les fonctions déjà existantes, ou des recettes permettant de reproduire des comportement complexes.

Parcourez la doc, et utilisez les fonctions qui y sont présentes pour simplifier les codes de la figure 6. Si besoin, considérez que `a = [1, 2, 3, 4, 5]` et `b = [6, 7, 8, 9, 10, 11]`. Si vous ne comprenez pas un programme, essayez avec d'autres valeurs, comme `[1, 1, 1, 3, 1, 1]`. Lorsque nécessaire, proposez de meilleurs noms pour les variables.

a
```
for p in a:
    for q in b:
        print(p, q)
```

b
```
for idx, e in enumerate(b):
    for f in b[idx+1:]:
        print(e, f)
```

c
```
while True:
    for e in a:
        print(e)
```

d
```
c = [1, 0, 1, 1, 0, 0]
for e, f in zip(c, b):
    if e: print(f)
```

e
```
curr_val, curr_rep = a[0], 1
for e in a[1:]:
    if e == curr_val:
        curr_rep += 1
    else:
        print([curr_val] * curr_rep)
        curr_val, curr_rep = e, 1
if curr_rep > 0:
    print([curr_val] * curr_rep)
```

f
```
for p, q in zip(a, b):
    print(p, q)
if len(a) > len(b):
    for e in a[len(b):]:
        print(e, 'nope')
elif len(a) < len(b):
    for e in b[len(a):]:
        print('nope', e)
```

g
```
start, step = 0, 2
while start <= (len(b) - step):
    print(b[start:start+step])
    start += step
```

h
```
start, width = 0, 3
while start <= (len(b) - width):
    print(b[start:start+width])
    start += 1
```

i
```
for idx, e in enumerate(b):
    for f in b[:idx]:
        print(e, f)
    for f in b[idx+1:]:
        print(e, f)
```

j
```
c = []
for e in a:
    c.append(e)
for e in b:
    c.append(e)
```

Figure 6

2.3 *collections & functools*

Ces deux modules ont des usages très avancés, nous allons nous contenter de quelques unes de leurs fonctionnalités. Simplifiez les codes dans la figure 7 en utilisant des fonctions des modules collections et functools. Si besoin, considérez que `a = [1, 2, 2, 1, 3, 2]`.

a

```
counts = {}
for val in a:
    if val in counts:
        counts[val] += 1
    else:
        counts[val] = 1
best = max(counts.values())
for val, count in counts.items():
    if count == best:
        print(val)
```

b

```
config = {'log': True, 'level': 'debug'}
cli = {'level': 'info', 'data': a}
to_find = 'level'  # could be any key
if to_find in cli:
    print(cli[to_find])
elif to_find in config:
    print(config[to_find])
else:
    print(to_find, 'not given')
```

c

```
assoc = {}
for idx, val in enumerate(a):
    if val in assoc:
        assoc[val].append(idx)
    else:
        assoc[val] = [idx]
```

d

```
acc = 0
for e in a:
    acc += e
print(acc)
```

Figure 7

3 TP — Manipulation d'images, interlacement texte/image

L'objectif de ce TP est de créer un programme avec interface CLI permettant de convertir une image en version ASCII, avec couleurs, en utilisant un texte d'entrée. Ce sujet s'inspire du post reddit suivant : `https://huit.re/texted-pale-blue-dot`

Autre exemple : `https://huit.re/texted-hadoken`

Le principe est le suivant : le programme a en entrée un texte et une image. La sortie est une nouvelle image, où chaque pixel de l'image d'entrée est remplacé par une lettre du texte, de telle manière à recréer l'image, mais avec le texte. L'image de sortie sera donc plus massive, car pour afficher chaque lettre il faudra plusieurs pixels.

Afin de ne pas réinventer la roue, il est conseillé d'utiliser le module *pillow*, qui propose une interface pour l'ouverture, la modification et l'écriture d'images. Documentation : `https://pillow.readthedocs.io`

3.1 I/O

I Chemin vers une image, chemin vers un fichier texte, chemin vers la cible

O pas de spécification

side-effect écriture d'une image dans le fichier cible

4 Projet — Brainfuck

L'idée est d'implémenter un compilateur de brainfuck. Ce projet fait travailler les structures de données de base, et est un bon support de réflexion sur ce qu'est un langage, un compilateur, et la norme. Un parallèle avec Python est fait en dernière partie.
Si plusieurs groupes travaillent sur ce sujet, c'est l'occasion de mettre en place des benchmarks pour comparer les performances de chaque compilateurs.

4.1 Définition

Le brainfuck est un langage bien plus bas niveau que Python et même C. Il n'y a pas d'identifiants (les noms de variables par exemple), et uniquement 8 caractères acceptés, et sont tous associé à une instruction extrêmement simple. Ces caractères sont décris en table 1. Par défaut, tout autre caractère est ignoré.
En brainfuck, la mémoire est considérée comme une séquence de cellule chacune portant une valeur codée sur un octet (par défaut 0), et une tête de lecture, appelée pointeur, pointe vers une cellule (initialement, la première).
Les instructions permettent le déplacement du pointeur, ainsi que la manipulation de la valeur pointée. Ainsi, le code source `+>++>-` est une séquence de 6 instructions, dont l'effet sera, dans l'ordre: mettre la case mémoire pointée à 1, décaler le pointeur mémoire d'1 case à droite, ajouter 2 à la case mémoire pointée, décaler le pointeur mémoire d'une case à droite, retirer 1 à la case mémoire pointée.
Appliquer ce code sur une mémoire définie par $[0, 0, 0]$ avec la position initiale du pointeur sur la case de gauche fixe l'état final de la mémoire à $[1, 2, -1]$.

Instruction	interprétation	Équivalent en Python
+	incrémente la valeur pointée	variable += 1
-	décrémente la valeur pointée	variable -= 1
>	déplace le pointeur à droite	sélection d'une variable
<	déplace le pointeur à gauche	sélection d'une variable
.	affiche la valeur pointée dans stdout	print
,	la valeur pointée est modifiée par stdin	input
[	saute au crochet fermant associé si la valeur pointée est zéro	boucles et conditions
]	saute au crochet ouvrant associé si la valeur pointée n'est pas zéro	boucles et conditions

Table 1: Les huit instructions définies par la norme du langage brainfuck. Les quatre premières sont relatives à la gestion de la mémoire, les deux suivantes aux entrées/sorties, et les 2 dernières au contrôle de flux.

4.1.1 Objectifs

Implémenter un interpréteur de brainfuck. L'idée est principalement d'allouer une mémoire, de l'initialiser à zéro, lire le code source caractère par caractère en appliquant les modification, puis de terminer lorsque le code source est épuisé.
En parallèle, vous devrez benchmarquer votre interpréteur, c'est-à-dire mesurer sa performance en temps sur des problèmes définis. Vous pourrez alors comparer vos implémentations avec les celles des autres groupes.
Il deviendra alors nécessaire d'ajouter une étape de compilation ou de l'analyse statique de code pour améliorer les performances.

4.2 Feuille de route

1. implémenter les 8 instructions en python ; bien déterminer les entrées et les sorties de chaque instruction.
2. produire des tests unitaires pour au moins 6 des 8 instructions

3. implémenter l'interpréteur simplement[1], le tester sur des codes sources glanés sur Wikipédia, Rosetta Code ou encore copy.sh

4. implémenter les tests fonctionnels associés

5. proposer et implémenter des méthodes pour passer de l'interprétation à la compilation (cf section 4.3). Benchmarquer ces méthodes pour vérifier qu'elles permettent bien un gain de performance.

6. utiliser la section 4.5 pour transformer votre programme en un réel compilateur

4.3 De l'interprétation à la compilation

Au départ, notre implémentation se rapproche d'un **interpréteur**. En d'autres mots, le programme lit directement le code source, et exécute les instruction sans grande intelligence.
C'est, d'une certaine manière, ce que fait l'interpréteur Python : il lit le code source (le fichier .py) et exécute son contenu[2].

L'autre manière de faire est de compiler le programme avant exécution, c'est-à-dire d'utiliser un **compilateur** pour transformer le code source en un *code objet*. Le code objet peut être n'importe quoi, mais en pratique, on vise généralement un langage très bas niveau, très proche de l'ordinateur, de manière à minimiser la complexité du langage que l'ordinateur doit utiliser pour exécuter le programme. L'exemple historique de la compilation est certainement le langage C, qui est traduit directement en *assembleur*, un langage extrêmement bas niveau qui est presque équivalent au binaire, le langage de l'ordinateur. L'exemple encore plus historique est l'assembleur lui-même, qui est compilé (on dit aussi *assemblé* dans ce cas précis) vers du binaire par un compilateur appelé dans ce cas précis un *assembleur*.

Dans notre cas, nous avons un interpréteur, et dans l'optique de pouvoir aller plus vite dans l'exécution d'un programme brainfuck, il nous faut trouver un moyen de le compiler.
Nous n'allons pas compiler vers de l'assembleur, car cela serait un peu compliqué à mettre en place, mais plutôt vers... du python !

L'intuition est la suivante : nous allons reprendre très exactement le code de notre interpréteur, mais plutôt qu'exécuter les instructions, nous allons les écrire dans un fichier avec le langage python. Ce fichier contiendra donc le *code objet* (écrit en python), généré par notre *compilateur* à partir d'un *code source* (écrit en brainfuck). Notre compilateur est donc un compilateur brainfuck-python.
Le fichier écrit en sortie (dont le nom est donné comme entrée au programme) contient donc, à la fin de la compilation, du code Python prêt à l'exécution : il suffit de lancer l'interpréteur python dessus, et le programme s'exécute.

Pour avoir un code objet (code python généré) encore plus optimisé, il serais de bon ton de réaliser une *analyse statique du code*, c'est-à-dire une analyse qui ne nécessite pas d'exécuter le code source. Une analyse simple et pouvant potentiellement améliorer significativement le code objet de notre compilateur est celle des caractères consécutifs. En effet, lorsque nous compilons `++++`, il serait malin de convertir l'ensemble de ces instructions, non pas en ajoutant quatre fois `memoire[p_memoire] += 1`, mais en ajoutant une seule fois `memoire[p_memoire] += 4`.

De la même manière, d'autres combinaisons d'instructions pourraient être converties en une seule, pour optimiser la compilation, l'exécution, voir même les deux. Pensez aux expressions régulières.

4.4 Aide et conseils

- votre premier interpréteur n'a pas besoin de tout gérer. Laissez les boucles de côté tant que le reste n'est pas fonctionnel. (autrement dit: écrivez des tests fonctionnels qui ne font pas intervenir de boucle)

[1] commencez avec des exemples simples, avec juste quelques instructions, et gardez les boucles pour la fin
[2] IRL, tous les interpréteurs et compilateurs empruntent un peu des deux mondes, car les deux approches sont nécessaires

- le code d'addition de deux nombres positifs en mémoire est particulièrement simple: `[->+<]`. Il est donc adapté au test de votre interpréteur.
- les entrées et sorties du programme (`,` et `.`) doivent être étudiées avec attention. La méthode la plus simple est certainement de convertir l'input en entier, et de toujours conserver la mémoire comme une valeur entre 0 et 255 (un octet, *cf* `latableASCII`). Notez que dans la norme du brainfuck, la valeur de fin de fichier est 10 (EOF en ASCII). Aussi, faites bien attention au codes trouvé sur le net. Par exemple le sleep sort sur rosetta code ne permet pas de trier des nombres, mais des caractères ascii. (donc, les 10 chiffres, mais aussi les lettres, les caractères spéciaux,...)
- certains programmes trouvés sur le net seront longs, très longs à l'exécution, du moins avec une implémentation naïve. Il est probable que vous ne puissiez faire tourner le sleep sort et la suite de Hailstone. Utilisez des codes plus simples dans un premier temps (additionneurs, afficheurs de texte). À terme, une bonne compilation et des optimisations pourraient permettre de faire tourner ces codes.
- dés que possible, implémentez le *point d'arrêt* : une nouvelle instruction qui affiche l'état de la mémoire, et attend que l'utilisateur presse entrée avant de continuer l'exécution. C'est très utile pour chasser les bugs.
- le benchmarking, dans l'idéal, devrait se faire sur une seule machine, sur un jeu de test complet et complexe, permettant une comparaison en temps entre différents compilateurs.
- le programme *99 bottles of beer* est un défi : une implémentation naïve, même écrite en C, est très lente. Il suffit d'un peu d'analyse statique pour résoudre ce problème, même si le compilateur est en python.
- un compilateur avec une belle interface d'utilisation, c'est un gros plus : il ne faudrait pas avoir à trifouiller dans votre code pour changer le nom des fichiers contenant les codes source et objet (par exemple, l'exécutable python reçois le fichier à exécuter par ligne de commande, vous n'avez pas à modifier son code source et à le recompiler pour définir quel fichier doit être lu, ni quel niveau d'optimisation doit être utilisé). C'est le moment d'utiliser les arguments de ligne de commande ou des fichiers de configuration. (tip: s'inspirer de ce qui existe... et suivre les standards. Mots-clef: docopt, argparse)

4.5 Pistes pour amélioration/enrichissement

- permettre l'usage des saut de ligne et des espaces pour l'indentation
- ajouter une instruction du même genre que `.` pour imprimer la valeur ASCII plutôt que la valeur de la mémoire elle-même (ou l'inverse, selon ce que vous avez implémenté)
- ajouter une instruction pour les commentaires, pour quitter le programme, et tout autre idée qui vous plaira. N'oubliez pas de garder la documentation de votre implémentation à jour !
- lever une erreur lorsqu'un caractère non valide apparaît dans le code source, ou l'ignorer silencieusement ?
- proposer une nouvelle instruction permettant d'initialiser les cases suivantes avec une string
- gérer gracieusement les cas limites, comme le dépassement de mémoire ou de bande.
- rendre votre compilateur configurable : taille de la bande mémoire, verbosité, alphabet d'instruction...
- analyse statique du code source : détection de boucle non fermée
- exemple d'option: obtenir une mémoire virtuellement illimitée en utilisant un dictionnaire plutôt qu'une liste, ou en augmentant la taille de la liste lorsque c'est nécessaire
- exemple d'option: indiquer s'il faut ou non optimiser le code objet
- faire un compilateur encore plus rapide en compilant vers du C

- gérer les extensions du brainfuck : `http://esolangs.org/wiki/Extended_Brainfuck`
- proposer une extension pour gérer du parallélisme
- une mémoire multi-dimensionnelle est-elle utile ?
- appliquer des algos génétiques sur des codes brainfuck : `https://huit.re/sNxCk7SB`

4.6 Parallèle avec Python

Par ce travail, nous avons étudié la norme d'un langage nommé brainfuck, que nous avons utilisée pour implémenter un programme comprenant ce langage, et réagissant à sa lecture comme indiqué dans la norme, ou, dans le cas du compilateur, produisant un code objet produisant les mêmes effets de bord que le code source écrit en brainfuck. Le résultat : un compilateur/interpréteur de brainfuck.

Ce schéma est loin d'être anodin : c'est celui utilisé partout pour tous les langages. Pour Python par exemple, la norme est définie par la *Python Software Foundation*, qui héberge sur python.org la documentation et la référence du langage. En plus de cela, ce même site met en avant l'implémentation *de référence* du langage : CPython.
Il existe d'autresimplémentations (`https://www.python.org/download/alternatives`), qui diffèrent généralement par le langage utilisé pour écrire le compilateur (C# pour ironpython en C#, java pour jython) ou par un code objet fondamentalement différent (par exemple pypy, qui compile vers du C plutôt que du bytecode).
Lorsque vous *lancez python*, vous utilisez en fait un compilateur/interpréteur qui implémente la norme du langage, et qui va lire votre code source. Si votre code n'utilise pas la même norme que le compilateur, des erreurs vont apparaître.
Dans ce cas, souvent, c'est l'humain qui ne s'est pas bien exprimé dans le langage et qui va, en face de l'erreur levée par le compilateur, modifier son code afin de l'adapter à la norme. Néanmoins, beaucoup plus rarement, c'est un *bug* du compilateur : il ne suit pas exactement la norme, car les personnes qui l'ont implémenté ont fait une erreur, ou n'ont pas prévu un cas limite particulier.

Maintenant, vous savez faire la différence entre Python[3], Python[4] et Python[5] !

[3] le langage
[4] la norme
[5] l'implémentation

5 Projet — Automate cellulaire

Ce projet consiste en la réalisation d'un automate cellulaire un peu plus élaboré que le jeu de la vie, où les cellules se déplacent dans un monde à deux dimensions, et où s'organise une sélection qui favorise certains comportements.

5.1 Définition

Une cellule est positionnée quelque part dans l'univers, et possède une certaine quantité d'énergie. De plus, elle possède une *orientation* dans l'espace, décrite comme la direction vers l'un de ses huit voisins de Moore.

En outre, et c'est là tout l'intérêt, elle possède trois probabilités complémentaires, ou *probabilités de mouvement* (PM), associées à son orientation, à la droite de son orientation, et à la gauche de son orientation. Les PM décrivent la prochaine case qu'atteindra la cellule ainsi que sa prochaine orientation.

Par exemple, une cellule orientée vers *haut-droit* ayant des PM de 10%, 50% et 40% aurait, au prochain tour, 10% de chance d'aller vers le haut, 50% de chances d'aller en haut à droite, et 40% de chances d'aller vers la droite. Une fois ce déplacement effectué, elle aura une nouvelle orientation, une nouvelle position, mais toujours les même PM.

La sélection s'opère par l'énergie : une cellule entrant dans une case récupère l'énergie qui s'y trouve, et perd une (petite) quantité d'énergie fixé. Une cellule n'ayant plus d'énergie disparaît.

La variabilité est introduite par un mécanisme de clonage simple: lorsqu'une cellule atteint un certains seuil d'énergie (à définir), elle se scinde en deux cellules dont les probabilités seront légèrement modifiées.

5.2 Feuille de route

1. implémenter l'univers avec un dictionnaire ou une liste de liste
2. implémenter le concept de cellule comme une quantité d'énergie associée à des probabilités de déplacement
3. proposer une fonction qui calcule la prochaine action d'une cellule
4. faire des tests unitaires
5. implémenter la routine qui fait passer un tour à la simulation
6. implémenter une vue de l'univers
7. implémenter la boucle principale
8. implémenter l'énergie et sa collecte par les cellules
9. implémenter une fonction qui donne une version légèrement modifiée des PM en entrée
10. implémenter le clonage de cellule
11. montrer l'évolution de divers facteurs avec le temps
12. améliorer la simulation ; voir section 5.3

5.3 Pistes pour amélioration/enrichissement

La liste suivantes propose des idées d'amélioration de la simulation. Prenez ce qui vous semble important, une fois que la simulation de base est fonctionnelle.

Améliorations de la simulation :

- vitesse: aller plus vite consomme plus d'énergie
- comment réagissent deux cellules lorsqu'elles se touchent ?
- l'univers n'est pas nécessairement uniforme ; plus coûteux en énergie, nourriture plus rare, ralentissements...
- les cellules jouent-elles en même temps, ou chacune à leur tour ?
- la nourriture dispersée dans l'univers peut être regénérée aléatoirement, ou en utilisant un automate cellulaire de type *jeu de la vie*, par exemple *seeds* (`https://huit.re/life-seeds-automata`).[6]
- plutôt que 5 directions, il pourrait y avoir 3 ou 7 directions associées à des PM.

Améliorations de l'analyse :

- appliquer des méthodes de clustering sur les PM, voir si différentes *espèces* (au sens très large) co-existent.
- rendre la simulation interactive : possibilité pour l'utilisateur de consulter les PM d'une cellule, de modifier l'univers,...
- statistiques sur les probabilités de déplacement, sur leur évolution
- affichage de la généalogies des cellules
- quid de la reproductibilité des simulations ?

Augmentation de la variabilité :

- soumission de plus de comportements à la variabilité (tels que la valeur seuil d'énergie nécessaire au clonage)
- le clonage, c'est bien, mais la reproduction sexuée, c'est mieux
- plutôt que 5 directions, il pourrait y avoir 3 ou 7 directions associées à des PM.
- chaque individu pourrait avoir non pas juste une série de PM, mais un automate à état fini où chaque nœud est une PM. Tous les N tours, l'état suivant de l'automate est tiré au hasard ou en fonction de l'environnement de la cellule et définit de facto sur quelles PM les prochains déplacement seront calculés.

[6] Ou une autre règle. Voir cette liste presque exhaustive : `https://huit.re/life-like-automata`