1 Cahier des charges

- ☞ *Les scripts de ce projet doivent être complétés à partir des squelettes de code fournis et rassemblés dans une archive* `zip` *nommée* `Eleve1_Eleve2_projet_brainfuck.zip`.
- ☞ *La date limite de rendu du projet est le samedi 8/01/2022 à 23h59 heure de Paris, archive à déposer dans :*
- ☞ *L'archive fournie contient :*
  - *Trois scripts avec squelette de code* `decouverte.py`, `brainfuck.py` *et* `sandbox.py`.
  - *Un script* `test_brainfuck.py` *pour lancer des tests sur* `brainfuck.py`. *Il doit être dans le même dossier que* `brainfuck.py`.
  - *Des fichiers textes : d'une part* `3mousquetaires.txt` *pour* `decouverte.py` *; d'autre part* `inputN.txt` *et* `judgeN.txt` *avec* `N` *variant entre 1 et 13 contenant d'une part une instance d'entrée de* `brainfuck.py` *et d'autre part une instance de la sortie attendue. Tous ces fichiers doivent être dans le même dossier que les scripts.*
- ☞ *Les réponses aux questions qui ne nécessitent pas de code doivent être fournies dans le code source sous forme de commentaire en les préfixant par le numéro de la question.*
- ☞ *Chaque fonction doit être documentée avec une* `docstring`.
- ☞ *Les parties les moins évidentes du code doivent être commentées de façon pertinente.*
- ☞ *Le code produit doit être confronté aux tests unitaires fournis pour deux fonctions et aux tests généraux en exécutant* `test_brainfuck.py`. *Les test échoués doivent être mentionnés en commentaire au début du fichier* `brainfuck.py`.

2 Introduction aux fichiers textes et aux dictionnaires

Les codes et réponses de cette partie seront complétés dans le fichier `decouverte.py` *qui est fourni. La spécification de chaque fonction est précisée dans sa* `docstring` *et les effets de bord attendus (pour les fonctions d'affichage) ou des tests accompagnent le squelette de code.*

Objectif :

L'objectif de cette partie est de découvrir une nouvelle structure de données, le dictionnaire de type `dict`, *qui représente une collection de valeurs indexées par des clefs qui ne sont pas forcément des entiers comme dans les tableaux de type* `list`. *L'accès en* lecture / écriture *s'effectue avec l'opérateur* `[]` *comme pour les tableaux, le parcours s'effectue par clef ou par couple* (clef, valeur). *On introduit aussi la manipulation de fichiers textes en lecture / écriture.*

Méthode *Dictionnaires*

Les **dictionnaires** permettent de construire des tables de symboles indexées par des clefs non entières, par exemple des chaînes de caractères.

Instruction ou expression	Rôle
`dicoc = dict() ou dico = {}`	crée un dictionnaire vide
`dico['Paul'] = 18`	associe la valeur 18 à la clef 'Paul' dans le dictionnaire
`dico.keys()`	clefs du dictionnaire (itérable avec une boucle `for`)
`dico.items()`	couples (clef, valeur) du dictionnaire (itérable avec une boucle `for`)

À l'instar des éléments d'un tableau ordonné par leur index, les éléments d'un dictionnaire ne sont pas ordonnés et on ne peut pas prévoir l'ordre d'apparition lorsqu'on itère avec une boucle `for` sur un dictionnaire (pour les dernières versions de Python, ils sont classés dans l'ordre d'insertion des couples (clef, valeur)).

```
>>> note = {}
>>> note['Knuth'] = 18
>>> note['Euclide'] = 20
>>> for (clef, val) in note.items(): # parcours par (clef, valeur)
...     print('clef = ', clef, '|', 'val = ', val)
...
clef = Knuth | val = 18
clef = Euclide | val = 20
>>> for clef in note:   # parcours par clef
    print('clef = ', clef, '|', 'val = ', note[clef])
...
clef = Knuth | val = 18
clef = Euclide | val = 20
```

Méthode *Fichiers*

On accède à un fichier texte à travers un descripteur de fichier renvoyé par la fonction builtins `open`, le paramètre `mode` précisant le mode d'accès (*lecture/écriture*).

- En *lecture*, on peut lire une ligne avec la méthode `readline()` ou parcourir le fichier ligne par ligne avec une boucle `for`.
- En *écriture*, le fichier ouvert est écrasé par défaut et on écrit par ajout à la fin avec la méthode `write`.

On donne un exemple de lecture du texte contenu dans `3mousquetaires.txt` qui est recopié à l'envers dans `3mousquetaires-envers.txt`.

```
entree = open("3mousquetaires.txt", mode="r", encoding="utf-8")
entree.readline()
tab_ligne = []
for ligne in entree:
    tab_ligne.append(lignes)
entree.close()
sortie = open("3mousquetaires-envers.txt", mode="w", encoding="utf-8")
for k in range(len(tab_lignes) - 1, -1, -1):
    sortie.write(tab_lignes[k])
sortie.close()
```

```
sortie_lecture = open("3mousquetaires-envers.txt", mode="r", encoding="
    utf-8")
sortie_lecture.close()
```

1. Compléter la fonction `extraire_tab_mot(path)` pour que sa spécification et les tests unitaires fournis dans `decouverte.py` soient satisfaits. Le fichier `'3mousquetaires.txt'` doit se trouver dans le même dossier que `decouverte.py`.

```
def extraire_tab_mot(path):
    """
    Découpe le texte contenu dans le fichier texte de
    chemin path selon le séparateur espace simple

    Paramètre :
        - path : str

    Retour:
        - tableau de mots de type str
    """
    #à  compléter


# test unitaire
tab_mots_mousquetaires = extraire_tab_mot('3mousquetaires.txt')
assert (len(tab_mots_mousquetaires), tab_mots_mousquetaires[:3]) ==
    (239574, ['introduction', 'il', 'y'])
```

Si l'interpréteur Python ne trouve pas le fichier `'3mousquetaires.txt'`, c'est un problème de répertoire de travail. Le code ci-dessous devrait permettre de récupérer le bon chemin relatif pour le fichier :

```
import os
absolute_path = os.path.dirname(os.path.abspath(__file__))
path_mousquetaire = os.path.join(absolute_path,'./3mousquetaires.txt')
tab_mots_mousquetaires = extraire_tab_mot(path_mousquetaire)
```

2. Compléter la fonction `histogramme(tab)` pour que sa spécification et le test unitaire fournis dans `decouverte.py` soient satisfaits.

3. Compléter la fonction `plus_frequent_mot(histo)` pour que sa spécification et les tests unitaires fournis dans `decouverte.py` soient satisfaits.

3 Présentation du langage de programmation *brainfuck*

Objectif :

L'objectif de cette partie est présenter un langage de programmation minimaliste qui porte le nom de brainfuck.

Histoire 1

Pour faire les présentations, citons la notice Wikipedia :

https://fr.wikipedia.org/wiki/Brainfuck

> Brainfuck *est un langage de programmation exotique, inventé par* Urban Müller *en 1993. Il tire son nom de l'union de deux mots anglais,* brain *(« cerveau ») et fuck (« … »), et joue sur les mots, puisque ce langage est volontairement simpliste, et parce que l'expression* Brain Fuck *évoque, en argot, ce qui met le cerveau dans un état de confusion par sa complexité apparente*
>
> *L'objectif de Müller était de créer un langage de programmation simple, destiné à fonctionner sur une* machine de Turing, *et dont le compilateur aurait la taille la plus réduite possible. Le langage se satisfait en effet de seulement huit instructions. La version 2 du compilateur originel de Müller, écrit pour l'Amiga, ne pesait lui-même que 240 octets [...]. Le* brainfuck *est pourtant un langage Turing-complet, ce qui signifie que, malgré les apparences, il est théoriquement possible d'écrire n'importe quel programme informatique en* brainfuck.
>
> *La contrepartie est que les programmes produits sont inefficaces et difficiles à comprendre.*
>
> *Le* brainfuck *utilise un tableau de cases dans lequel il stocke des valeurs. Chacune de ces cases prend une valeur entière codée sur 8 bits, ce qui permet des valeurs entre 0 et 255. Il utilise également un pointeur, c'est-à-dire une variable qui prend des valeurs entières positives indiquant l'indice de la case du tableau que l'on modifie actuellement. Par convention, on attribue en informatique l'indice 0 à la première case du tableau.*
>
> *Au lancement du code* brainfuck *, le pointeur est initialisé à 0 et toutes les cases du tableau sont nulles.*
>
> *Les huit instructions du langage, chacune codée par un seul caractère, sont les suivantes :*

Caractère	Signification
>	incrémente (augmente de 1) le pointeur
<	décrémente (diminue de 1) le pointeur
+	incrémente l'octet du tableau sur lequel est positionné le pointeur (l'octet pointé)
−	décrémente l'octet pointé
.	sortie de l'octet pointé (valeur ASCII)
,	entrée d'un octet dans le tableau à l'endroit où est positionné le pointeur (valeur ASCII)
[	saute à l'instruction après le] correspondant si l'octet pointé est à 0
]	retourne à l'instruction après le [si l'octet pointé est différent de 0

Exemple 1

On déroule ci-dessous l'exécution du programme `++>,<[>.+<-]` en surlignant en gris la position dans le code et le pointeur sur la case mémoire.
Par défaut, on lit le caractère courant du code, on exécute l'instruction puis on se déplace d'un caractère dans le code sauf pour les caractères `[` ou `]` qui provoquent des sauts dans le code et permettent de réaliser des instructions conditionnelles ou des boucles. Une instruction peut provoquer un changement d'état de la mémoire (caractères + ou −, entrée `,`), une sortie (`.`) ou un déplacement vers une autre case mémoire (< ou >).

1. On incrémente de 1 la case 0 initialisée à 0.

Code

+	+	>	,	<	[	>	.	+	<	−	]

Mémoire

Case	0	1
Valeur	1	0

2. On incrémente de 1 la valeur de la case pointée.

Code

+	+	>	,	<	[	>	.	+	<	−	]

Mémoire

Case	0	1
Valeur	2	0

3. On incrémente de 1 le pointeur de case mémoire.

Code

+	+	>	,	<	[	>	.	+	<	−	]

Mémoire

Case	0	1
Valeur	2	0

4. *Entrée* On affecte à la case mémoire pointée, une entrée (valeur d'un octet entre 0 et $2^8 - 1 = 255$) interprétée comme un code ASCII . On suppose que l'entrée est 48, code ASCII du caractère `'0'` représentant l'entier 0.

La fonction `ord` permet d'obtenir le code ASCII d'un caractère.

```
>>> ord('0')
48
```

Code

+	+	>	,	<	[	>	.	+	<	−	]

Mémoire

Case	0	1
Valeur	2	48

5. On décrémente le pointeur de case mémoire, on revient sur la case précédente.

Code

+	+	>	,	<	[	>	.	+	<	−	]

Mémoire

Case	0	1
Valeur	2	48

6. *Test / Saut* La case mémoire est non nulle, on saute dans le code vers le caractère suivant le `[` correspondant. Le contenu de la case mémoire est non nul, on se déplace d'un caractère dans le code.

Code

+	+	>	,	<	[	>	.	+	<	−	]

Mémoire

Case	0	1
Valeur	2	48

7. On incrémente de 1 le pointeur de case mémoire.

Code

+	+	>	,	<	[	>	.	+	<	−	]

Mémoire

Case	0	1
Valeur	2	48

8. *Sortie* On affiche sur la sortie le caractère dont le code ASCII est contenu dans la case mémoire, le caractère `'0'` est donc affiché sur la sortie.

Code

+	+	>	,	<	[	>	.	+	<	-	]

Mémoire

Case	0	1
Valeur	2	48

9. On incrémente de 1 la valeur de la case pointée.

Code

+	+	>	,	<	[	>	.	+	<	-	]

Mémoire

Case	0	1
Valeur	2	49

10. On décrémente le pointeur de case mémoire, on revient sur la case précédente.

Code

+	+	>	,	<	[	>	.	+	<	-	]

Mémoire

Case	0	1
Valeur	2	49

11. On décrémente la valeur de la case mémoire.

Code

+	+	>	,	<	[	>	.	+	<	-	]

Mémoire

Case	0	1
Valeur	1	49

12. *Test / Saut* La case mémoire est non nulle, on saute dans le code vers le caractère suivant le `[` correspondant.

Code

+	+	>	,	<	[	>	.	+	<	-	]

Mémoire

Case	0	1
Valeur	1	49

13. Deuxième tour de boucle, on incrémente de 1 le pointeur de case mémoire.

Code

+	+	>	,	<	[	>	.	+	<	-	]

Mémoire

Case	0	1
Valeur	1	49

14. *Sortie* On affiche sur la sortie le caractère dont le code ASCII est contenu dans la case mémoire, le caractère `'1'` est donc affiché sur la sortie.

Code

+	+	>	,	<	[	>	.	+	<	-	]

Mémoire

Case	0	1
Valeur	1	49

15. On incrémente de 1 la valeur de la case pointée.

Code

+	+	>	,	<	[	>	.	+	<	-	]

Mémoire

Case	0	1
Valeur	1	50

16. On décrémente le pointeur de case mémoire, on revient sur la case précédente.

Code

+	+	>	,	<	[	>	.	+	<	-	]

Mémoire

Case	0	1
Valeur	1	50

17. On décrémente la valeur de la case mémoire.

Code

+	+	>	,	<	[	>	.	+	<	-	]

Mémoire

Case	0	1
Valeur	0	50

18. *Test / Saut* La case mémoire est nulle, pas de saut, on avance d'un caractère dans le code mais il n'y en a plus donc le programme se termine..

Code

+	+	>	,	<	[	>	.	+	<	-	]

Mémoire

Case	0	1
Valeur	0	50

Et maintenant quelques questions, auxquelles vous répondrez en insérant des commentaires au début du fichier `brainfuck.py`.

1. Écrire un programme *brainfuck* qui affiche les trois caractères `NSI`.
2. Écrire un programme *brainfuck* qui ne se termine pas (boucle infinie).
3. Le langage de programmation *brainfuck* peut s'exécuter sur un ordinateur d'architecture Von Neumann. Décrire brièvement le rôle de chacune de ses grandes composantes : **la mémoire** et l'unité centrale de traitement dont les deux parties sont **l'unité de contrôle** et **l'unité de calcul**.

4 Commenter un programme *brainfuck*

Objectif :

*L'objectif de ce projet est d'écrire un **interpréteur** de programme écrit en brainfuck, c'est-à-dire un programme qui prend en entrée un autre programme et l'exécute en affichant éventuellement des erreurs de syntaxe ou d'exécution.*
Un programme brainfuck est constitué des huit caractères correspondant à des instructions mais peut contenir d'autres caractères qui peuvent correspondre à des commentaires et qui ne sont pas pris en compte par l'interpréteur.

1. Le fichier `input11.txt` contient un programme *brainfuck* avec commentaires, dont l'exécution affiche `Hello World!`. La sortie est dans le fichier `judge11.txt`.
2. Copier le fichier `input1.txt` sous le nom `input1-commentaire.txt` et commenter dans ce dernier le programme *brainfuck* contenu sur la deuxième ligne.
3. Copier le fichier `input12.txt` sous le nom `input12-commentaire.txt` et commenter dans ce dernier le programme *brainfuck* contenu sur la deuxième ligne.

5 Écrire un interpréteur *brainfuck*

Objectif :

Dans cette partie, on complète le squelette de code d'un interpréteur brainfuck, on le soumet à un jeu de tests puis on le confronte à un juge en ligne sur :

`https://www.codingame.com/ide/puzzle/what-the-brainfuck`

Méthode

Le fichier `brainfuck.py` contient un squelette d'interpréteur *brainfuck* conçu pour lire un fichier comme `input3.txt` :

```
1 4 2
,>,><[<[>>+>+<<<-]>>>[<<<+>>>-]<<-]>. Commentaire
4
9
```

- La première ligne contient trois entiers `L`, `S` et `N` séparés par un espace.
- `S` est la taille du tableau de cases mémoires nécessaire pour l'exécution
- Les `L` lignes suivantes contiennent le programme *brainfuck* avec d'éventuels commentaires.
- Les `N` lignes suivantes contiennent les éventuelles entrées du programme (une entrée par ligne).

L'archive réunit 13 fichiers `inputN.txt` avec $1 \leqslant N \leqslant 13$ et 13 fichiers `judgeN.txt` contenant les sorties que doit afficher l'interpréteur en exécutant les programmes.
Le fichier `test_brainfuck.py` exécute `brainfuck.py` sur ces 13 instances. On peut rajouter des fichiers `inputN.txt` et `judgeN.txt` correctement formatés, il faut alors adapter la variable `NB_TESTS`. Certains codes de tests contiennent des erreurs de syntaxe ou d'exécution (pointeur mémoire hors de la plage licite, valeur hors de la plage $[\![0;255]\!]$).
Dans ce cas l'interpréteur va affiche des messages d'erreur sur la sortie :

- `SYNTAX ERROR` si les crochets sont mal fermés ou imbriqués ;
- `POINTER OUT OF BOUNDS` si le pointeur de case mémoire sort des limites (entre 0 et `S`-1) ;
- `INCORRECT VALUE` si une valeur lue sur l'entrée est en dehors de la plage licite (entre 0 et 255, codage sur 1 octet).

Pour exécuter dans `test_brainfuck.py` l'interpréteur sur différents fichiers `inputN.txt`, on a défini toutes les fonctions utiles à l'intérieur d'une seule fonction `main(input_path, output_path)` qui est le point d'accès pour le script de test. Par exemple `main('input1.txt', 'output1.txt')` lit son entrée dans `'input1.txt'` puis écrit sa sortie dans `'output1.txt'` et `test_brainfuck.py` va comparer celle-ci avec celle attendue dans `'judge1.txt'`.

Les fonctions définies dans `main` ne sont visibles qu'à l'intérieur de `main` et toutes les variables définies dans le corps principal de `main` sont accessibles depuis les fonctions définies dans `main`. Ceci permet d'éviter l'usage de variables globales.

On donne ci-dessous la structure simplifiée du squelette fourni dans `brainfuck.py`.

```
import sys

def main(input_path, output_path):
    """
    Exécute l'interpréteur brainfuck sur un fichier d'entrée
    imprime sa sortie sur un fichier de sortie
    """
```

```
    def read_input():
        """Lit le flux d'octets fourni par input_file"""
        return input_file.readline()

    def load(input_file):
        """
        Lit un fichier d'entrée pour l'interpréteur
        Initialise et renvoie les variables utiles
        """

    def syntax_parser(code):
        """Parcourt le code, détecte les erreurs de crochets mal formés
            """

    def test_unitaire_syntax_parser():
        """Tests unitaires pour syntax_parser"""

    def check_error(memory, pointer):
        """
        Vérifie une erreur de dépassement de pointeur mémoire
        ou de dépassement de capacité (entre 0 et 255) pour une valeur
        """

    def test_unitaire_check_error():
        """Tests unitaires pour check_error"""

    def print_memory(memory, pointer, output_file):
        """Imprime la case mémoire courante sur la sortie"""

    def read_data(memory, pointer, data):
        """Lit une entrée du programme brainfuck"""

    def decode(token):
        """Décode le caractère token du programme brainfuck"""

    def interpreter(memory, code, pointer):
        """Interprète le code brainfuck"""

    # début du corps principal de main(input_path, output_path)
    # tests unitaires pour syntax_parser et check_error
    test_unitaire_syntax_parser()
    test_unitaire_check_error()
    # Ouverture des fichiers d'entrée/sortie pour l'interpréteur
    # si chemin vide c'est l'entrée/sortie standard (clavier / écran)
    if input_path != "":
        input_file = open(input_path, "r")
    else:
        input_file = sys.stdin
    if output_path != "":
```

```
        output_file = open(output_path, "w")
    else:
        output_file = sys.stdout
    # chargement du fichier d'entrée
    # initialisation de la mémoire du code, du dictionnaire de pointeurs
    memory, code, data, pointer = load(input_file)
    # parcours du code, détection d'erreur de syntaxe, association des
        crochets ouvrant / fermant
    syntax_error, bracket_opening, bracket_ending = syntax_parser(code)
    if syntax_error:
        print("SYNTAX ERROR", file = output_file)
    else:
        # on interpètre le code s'il n'y a pas d'erreur de syntaxe
        interpreter(memory, code, data, pointer)
    # on ferme les fichiers d'entrée / sortie
    input_file .close()
    output_file.close()

# exécution de la fonction main si le module est appelé directement
if __name__ == "__main__":
    main("input1.txt","output1.txt")
```

Les variables principales définies dans `main` sont :

Nom	Type	Exemple	Sens
`pointer`	`dict`	`{'memory' : 0, 'data' : 0, 'code' : 0}`	Pointeurs de position pour le code, la mémoire, les données
`code`	`str`	`'+++>,<[>.+<-]'`	code source *brainfuck*
`memory`	`list`	`[0, 0]`	tableau de cases mémoire
`data`	`list`	`[8]`	tableau des entrées du programme (entiers entre 0 et 255)
`bracket_opening`	`dict`	`{4:2}`	`'['` en position 4 ferme `']'` en position 2
`bracket_ending`	`dict`	`{2:4}`	`'['` en position 2 fermé par `']'` en position 4
`token`	`str`	`'+'`	Code d'instruction *brainfuck* parmi `'+-<>,.[]'`

On a choisi de traiter les entrées / sorties comme des fichiers car c'est un principe héritée du système d'exploitation UNIX, partagé par de nombreux interpréteurs de commandes comme Bash et des langages de script comme Python.
Par défaut `input` lit sur **l'entrée standard** (le clavier) et `print` imprime sur **la sortie standard** (une fenêtre sur l'écran) à travers des fichiers accessibles depuis le module `sys` par `sys.stdin` et `sys.stdout`. Il est possible de rediriger l'entrée ou la sortie standard vers un autre fichier comme on le fait avec `inputN.txt` pour l'entrée et `outputN.txt` pour la sortie. Si on passe des chemins vides en paramètres (`main("", "")`), l'entrée et la sortie sont positionnées sur les valeurs standards (clavier / écran).

Répondre aux questions sans code Python sous la forme de commentaires insérés au début du fichier `brainfuck.py` sous forme de commentaires.

1. Ouvrir le fichier `sandbox.py` pour mettre au point les fonctions ci-dessous en s'aidant de leur spécification et des jeux de tests unitaires fournis avant de les recopier dans `brainfuck.py`.

   - `syntax_parser(code)`
   - `check_error(memory, pointer)`

   La fonction `syntax_parser(code)` est la plus complexe. Pour vous aider à la compléter, on donne ci-dessous une trace d'exécution de `syntax_parser('++[>++[-]<-]')` qui renvoie `(False, {8:6, 11:2}, {6:8, 2:11})`. On a commenté les différentes étapes.

```
Itération k : 0      # premier tour de boucle
token : +            # lecture du caractère stocké dans token
bracket_stack : []
bracket_opening : {}
bracket_ending : {}


Itération k : 1       # second tour de boucle
token : +             # lecture du caractère stocké dans token
bracket_stack : []
bracket_opening : {}
bracket_ending : {}


Itération k : 2       # troisième tour de boucle
token : [             # lecture du caractère stocké dans token
bracket_stack : [2]   # token == '[' on empile la position sur la pile
    bracket_stack
bracket_opening : {}
bracket_ending : {}


Itération k : 3       # quatrième tour de boucle
token : >             # lecture du caractère stocké dans token
bracket_stack : [2]
bracket_opening : {}
bracket_ending : {}


Itération k : 4      # cinquième tour de boucle
token : +            # lecture du caractère stocké dans token
bracket_stack : [2]
bracket_opening : {}
bracket_ending : {}


Itération k : 5      # sixième tour de boucle
token : +            # lecture du caractère stocké dans token
```

```
bracket_stack : [2]
bracket_opening : {}
bracket_ending : {}


Itération k : 6     # septième tour de boucle
token : [           # lecture du caractère stocké dans token
bracket_stack : [2, 6] # token == '[' on empile la position sur la pile
    bracket_stack
bracket_opening : {}
bracket_ending : {}


Itération k : 7     # huitième tour de boucle
token : -            # lecture du caractère stocké dans token
bracket_stack : [2, 6]
bracket_opening : {}
bracket_ending : {}


Itération k : 8     # neuvième tour de boucle
token : ]           # lecture du caractère stocké dans token
bracket_stack : [2] # caractère == ']' on dépile le sommet de la pile
    bracket_stack (6)
bracket_opening : {8: 6} # crochet ']' en pos 8 ouvert par crochet '['
    en pos 6
bracket_ending : {6: 8} # crochet '[' en pos 6 fermé par crochet ']' en
    pos 8


Itération k : 9     # dixième tour de boucle
token : <           # lecture du caractère stocké dans token
bracket_stack : [2]
bracket_opening : {8: 6}
bracket_ending : {6: 8}


Itération k : 10    # onzième tour de boucle
token : -          # lecture du caractère stocké dans token
bracket_stack : [2]
bracket_opening : {8: 6}
bracket_ending : {6: 8}


Itération k : 11  # douzième tour de boucle
token : ]         # lecture du caractère stocké dans token
bracket_stack : [] # caractère == ']' on dépile le sommet de la pile
    bracket_stack (2)
bracket_opening : {8: 6, 11: 2} # crochet ']' en pos 11 ouvert par
    crochet '[' en pos 2
```

```
bracket_ending : {6: 8, 2: 11} # crochet '[' en pos 2 fermé par crochet
    ']' en pos 11
```

2. D'après le squelette de code fourni dans `brainfuck.py`, quelle fonction Python possède la *docstring* ci-dessous, obtenue à l'aide de la fonction `help(fonction_mystere)` de Python ?

```
Return a Unicode string of one character with ordinal i;
0 <= i <= 0x10ffff
```

3. Dans `brainfuck.py` compléter la fonction `read_data(memory, pointer, data)` en respectant sa spécification.

4. Dans `brainfuck.py` compléter la fonction `decode(token)` en respectant sa spécification.

   Est-il pertinent d'insérer une précondition `assert token in '+-<>,.[]', "Token invalide"` au début de la fonction ? Pourquoi

5. Dans `brainfuck.py` compléter la fonction `interpreter(memory, code, pointer)` en respectant sa spécification.

   Premier test : exécuter `brainfuck.py` qui va appeler `main("input1.txt","output1.txt")`.

6. Si le premier test a réussi, ouvrir le script `test_brainfuck.py` et l'exécuter. Pour les 13 fichiers `inputN.txt` avec $1 \leqslant N \leqslant 13$, il va comparer les sorties `outputN.txt` avec les 13 sorties attendues dans les fichiers `judgeN.txt`.

```
Test input1.txt réussi
Test input2.txt réussi
Test input3.txt réussi
Test input4.txt réussi
Test input5.txt réussi
Test input6.txt réussi
Test input7.txt réussi
Test input8.txt réussi
Test input9.txt réussi
Test input10.txt réussi
Test input11.txt réussi
Test input12.txt réussi
Test input13.txt réussi
13/13 tests réussis, Bravo !
```

7. Si les 13 tests ont été passés avec succès confronter le code de `brainfuck.py` au juge en ligne :

   https://www.codingame.com/ide/puzzle/what-the-brainfuck

   ⚠ Pour le juge en ligne, il faut lire et écrire sur l'entrée et la sortie standard, relisez le dernier point de méthode pour trouver comment paramétrer l'appel de la fonction `main`.

❄ ❄

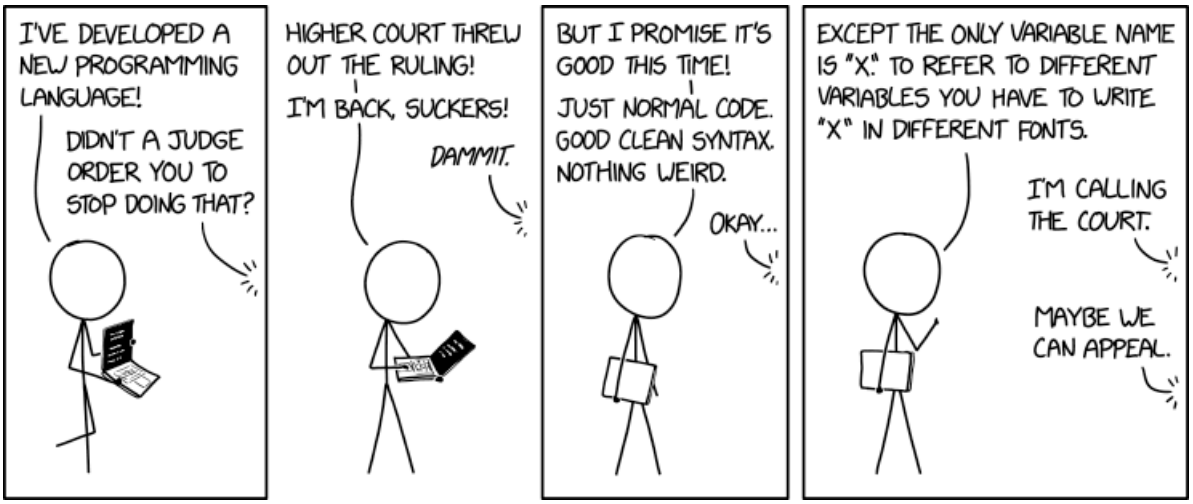
Q'affiche ce programme `brainfuck`*? Maintenant, vous pouvez tricher avec votre interpréteur* `brainfuck.py` *(fichier* `'input14.txt'`*)!*

```
++++++++++[>+>+++>+++++++>++++++++++<<<<-]>>>++++.
>++++++++++.++++++++++.--------------------.
++++++++++++++++.+++.<<++.>++++.
>---------.----------.+++++++.
```

❄ ❄

Un petit dessin du célèbre blog XKCD pour terminer avec une explication ici :

https://www.explainxkcd.com/wiki/index.php/2309:_X



Source : XKCD 2309