université
PARIS-SACLAY

Programmation en Python
Master 2 Réseaux Télécoms
Examen du 5 janvier 2022

Durée 3 heures 30 minutes, notes manuscrites autorisées

Organisation de l'espace de travail

1. Dans le répertoire du sujet, vous allez retrouver l'archive `exam20220105.zip` avec le code Python à compléter. Décompressez cette archive dans votre profil (par exemple dans le dossier `Documents` ou `MesDocuments`).
2. Lancez VSCode et ouvrez le dossier décompressé (e.g. `Documents\exam20220105`).
3. Complétez le code dans les fichiers `leap.py`, `roman.py` et `cryptarithme.py`.
4. Testez le code avec les tests unitaires fournis.
5. A la fin de l'examen, sauvegardez vos fichiers et fermez la session sans éteindre l'ordinateur.

1 Année bissextile

Une année est dite bissextile (c'est-à-dire, comporte 366 jours) si une des deux conditions est vérifiée :

1. l'année est divisible par 4 et non divisible par 100
2. l'année est divisible par 400

Complétez le code de la fonction suivante :

```
def leap_year(year):
    pass
```

Cette fonction retournera une valeur booléenne qui est vraie si et seulement si son argument est une année bissextile.

2 Numération romaine

n	Symbole pour 10^n	Symbole pour $5 \cdot 10^n$
0	I	V
1	X	L
2	C	D
3	M	$\overline{\text{V}}$
4	$\overline{\text{X}}$	$\overline{\text{L}}$
5	$\overline{\text{C}}$	$\overline{\text{D}}$
6	$\overline{\text{M}}$	

Table 1 – Numération romaine.

Le système de numération romain utilise des symboles spéciaux pour les puissances de 10 et pour les nombres de type $5 \cdot 10^n$ (voir Table 1). Les autres multiples de 10^n sont représentés par ces deux symboles suivant toujours le même schéma. Par exemple, pour $n = 0$ on a :

1=I
2=II
3=III
4=IV
5=V
6=VI
7=VII
8=VIII
9=IX

Pour obtenir la représentation d'un nombre dans ce système on remplace un par un les chiffres de sa représentation décimale par les symboles romains correspondants, en tenant compte de la position du chiffre et en sautant les zéros. On peut alors encoder les nombres compris entre 1 et 3999999.

Complétez le code de la fonction suivante :

```
def to_roman(number):
    pass
```

Cette fonction retournera une chaîne de caractères correspondant à la représentation romaine de son argument. Les symboles de la Table 1 sont répertoriés dans la variable globale `R`. Vous pouvez également reproduire le trait horizontal audessus des lettres (un *vinculum*) en faisant suivre la lettre par le caractère Unicode `\u0305`, par exemple, $\overline{\text{M}}$ correspond à `"M\u0305"`.

3 Cryptarithmes

Dans cet exercice on se propose de créer un programme pour résoudre des *cryptarithmes*, c'est-à-dire des casse-têtes arithmétiques consistant à résoudre une équation où les chiffres dans la représentation décimale d'un nombre sont remplacés par des lettres. Par exemple, le cryptarithme suivant :

$$\text{DONALD+GERALD=ROBERT}$$

admet une seule solution :

$$526485+197485=723970$$

L'expression peut utiliser les quatre opérations arithmétiques, ainsi que l'opérateur de puissance (`**`) et les parenthèses. Les chiffres seront codés par des lettres majuscules sans accent. On a le droit d'utiliser des littéraux entiers, mais pas de mélanger des chiffres et des lettres à l'intérieur du même nombre.

3.1 Répertorier les lettres distinctes

Complétez le code de la fonction suivante :

```
def set_uppercase(s):
    pass
```

Cette fonction doit retourner l'ensemble des lettres majuscules *distinctes* présentes dans le texte du cryptarithme `s`.

3.2 Répertorier les lexèmes

Complétez le code de la fonction suivante :

```
def set_tokens(s):
    pass
```

Cette fonction doit retourner l'ensemble des lexèmes représentant des nombres cachés dans le cryptarithme `s`, c'est à dire des blocs de texte composés uniquement des lettres majuscules. *Indication :* pensez à utliser des expressions rationnelles.

3.3 Une expression λ représentant un cryptarithme

Complétez le code de la fonction suivante :

```
def parse_equation(s):
    pass
```

Cette fonction retournera un tuple `(tokens, eq)`. Le premier élément du tuple est une liste de lexèmes du cryptarithme, et le second est une expression λ *évaluée*, représentant le cryptarithme. Par exemple, pour le cryptarithme

```
s = "HUIT + HUIT = SEIZE"
```

on aura

```
tokens = ['HUIT', 'SEIZE']
```

et

```
eq = lambda HUIT, SEIZE : HUIT + HUIT == SEIZE
```

Notez que l'ordre de lexèmes dans la liste `tokens` doit correspondre à l'ordre des arguments de l'expression `eq`. Dans le cas où le cryptarithme est mal formé, la fonction levera une `ValueError`. *Indications :* Pour transformer le symbole `'='` en opérateur `==` pensez à utiliser la méthode `replace` de la classe `str`. N'oubliez pas de compiler une chaîne de caractères en une expression λ en appelant la fonction `eval`. Si le cryptarithme est mal formé, cet appel lèvera une `SyntaxError`, qu'il conviendra à intercepter.

3.4 Solution d'un cryptarithme

Compléter le code de la fonction suivante :

```
def solutions(s):
    pass
```

Cette fonction retournera un générateur qui produit toutes les solutions du cryptarithme `s`. Les solutions seront donnés sous forme d'une chaine de caractères obtenue à partir de `s` en remplaçant les lettres par les chiffres correspondants. Par exemple, l'expression

```
set(solutions("HUIT + HUIT = SEIZE"))
```

donnera l'ensemble suivant :

```
{'9254 + 9254 = 18508', '8253 + 8253 = 16506'}
```

Notez que le premier chiffre d'un lexème ne peut être zéro. Si le nombre de lettres distinctes dans le cryptarithme est supérieur à 10, la fonction levera une `ValueError`. *Indications :* On peut utiliser la fonction `permutations` du module `itertools` pour énumérer les substitutions possibles. On calculera ensuite les valeurs numériques de chaque lexème et on évaluera l'expression λ donnée par `parse_equation` pour vérifier si on a trouvé une solution.