TP: chaînes de caractères et programmes de dessin

Il s'agit du TP noté de l'année 2021.

1. Substitutions dans des chaînes de caractères

Pour rappel, les chaînes de caractères en Python s'utilisent comme ceci:

- une chaîne de caractère peut être écrite explicitement entre guillements doubles `"comme ceci"` ou simples `'comme cela'`
- si `s` et `t` sont des chaînes de caractères, `s+t` est la chaîne obtenue par concaténation: `"abcd" + "ef" == "abcdef"`
- si `s` est une chaîne de caractères, `len(s)` donne le nombre de caractères qu'elle contient: `len("abd") == 3`
- si `s` est une chaîne de caractères, `s[i]` est le caractère en position `i`, représenté comme chaîne de caractères de longueur 1 (le premier caractère a la position `0` et le dernier a la position `len(s) - 1`): `"abcde"[3] == "d"`
- si `s` est une chaîne de caractères et `n` est un entier, `s * n` est la chaîne obtenue en répétant `n` fois `s`: `"bla" * 3 == "blablabla"`
- on peut faire une boucle qui traite chaque caractère d'une chaîne, dans l'ordre, en écrivant `for c in s`, par exemple

  ```
  for c in "paf":
      print("un caractère:", c)
  ```

  affichera

  ```
  un caractère: p
  un caractère: a
  un caractère: f
  ```

À la différence des listes, les chaînes de caractères ne sont pas modifiables: si `s` est une chaîne de caractères, l'instruction `s[i] = "a"` est interdite et provoque une erreur.

Question 1. Écrire une fonction `double(s)` qui attend une chaîne de caractères `s` et renvoie la chaîne obtenue en doublant chacun de ses caractères. Par exemple, `double("bonjour")` doit envoyer `"bboonnjjoouurr"`.

```
def double(s):
    "Écrivez votre code ici"
```

Quelques exemples pour vérifier que la fonction marche. Si une erreur se produit, c'est que votre fonction ne fonctionne pas.

```
assert double("salut") == "ssaalluutt"
assert double("quoi ?") == "qquuooii  ??"
assert double("") == ""
print("Ça a l'air bon")
```

Question 2. Écrire une fonction `substitution_lettre(s, l, t)` qui attend une chaîne de caractères `s`, une lettre `l` et une chaîne de caractère `t` et qui renvoie la chaîne obtenue en remplaçant chaque occurence de la lettre `l` par la chaîne `t`. Le test qui suit donne des exemples.

```
def substitution_lettre(s, l, t):
    "Écrivez votre code ici"
```

Quelques exemples pour vérifier que la fonction marche.

```
assert substitution_lettre("Oui. Ça fonctionne.", ".", "!") == "Oui! Ça fonctionne!"
assert substitution_lettre("Effacer les espaces", " ", "") == "Effacerlesespaces"
assert substitution_lettre("Bonjour!", "o", "ogodo") == "Bogodonjogodour!"
print("Ça a l'air bon!")
```

Question 3. On veut généraliser cette fonction pour substituer plusieurs lettres en même temps. On va représenter une substitution par un dictionnaire qui associe à chaque lettre le texte à remplacer. Quelques rappels sur les dictionnaires:

- un dictionnaire est une structure qui associe des *clés* (qui peuvent être des nombres, des chaînes de caractères, ou d'autres types) et des *valeurs* (qui peuvent être n'importe quoi)
- on écrit un dictionnaire avec la notation suivante: `{"x": 32, "paf": "toto"}` associe la clé `"x"` à la valeur `32` et la clé `"paf"` à la valeur `"toto"`
- si `d` est un dictionnaire, `d[x]` permet d'obtenir la valeur associée à la clé `x`, si cette clé est présente, cela fait une erreur si la clé n'est pas présente dans `d`
- l'opération `x in d` permet de tester si la clé `x` est présente ou pas dans `d`

Écrire une fonction `substitution(s, d)` qui attend une chaîne de caractères `s` et un dictionnaire `d` et qui effectue une substitution. Les clés de `d` sont supposées être des lettres, les valeurs associées sont les chaînes à substituer. Voir les tests plus bas pour des exemples.

```
def substitution(s, d):
    "Écrivez votre code ici"
```

```
assert substitution("abracadabra", {"a": "ou", "b": "d", "d": "b"}) == "oudroucouboudrou"
assert substitution("essai",  {"s": "x", "a": ""}) == "exxi"
print("Ça a l'air bon!")
```

Question 4. Écrire une fonction `substitution_iteree(s, d, n)` qui applique `n` fois successives la substitution `d` à la chaîne de caractères `s`.

```
def substitution_iteree(s, d, n):
    "Écrivez votre code ici"
```

```
assert substitution_iteree("abc", {"a": "bb", "b": "ac", "c": "ba"}, 2) == "acacbbbaacbb"
assert substitution_iteree("a+b", {"a": "(b+b)", "b": "a*a"}, 3) == "((b+b)*(b+b)+(b+b)*(b
print("Ça a l'air bon!")
```

2. Dessiner en suivant un programme

On va maintenant interpréter des chaîne de caractères comme des suites d'instructions de dessin, de différentes façons.

Pour faire effectivement les dessins dans cette feuille Jupyter, exécuter l'incantation suivante:

```
%matplotlib notebook
from matplotlib import pyplot as plt
```

Désormais, vous pouvez commencer une nouvelle figure avec l'instruction `plt.figure()` et ajouter des lignes brisées dans la dernière figure avec l'instruction `plt.plot(lx, ly)` où `lx` et `ly` sont deux listes de nombres, qui doivent être de la même longueur et représentent les coordonnées d'une suite de points. Par exemple:

```
plt.figure()
plt.plot([0,1,2,3,2], [0,3,1,2,2])
```

Remarquez que les axes sont placés automatiquement pour s'adapter aux lignes effectivement tracées, sans imposer que l'échelle soit la même pour les abscisses et les ordonnées. De plus vous pouvez redimensionner la figure et modifier le cadrage en utilisant les boutons. Si vous ajoutez des lignes, la figure s'adaptera si vous n'avez pas encore changé le cadrage.

```
plt.plot([0,10], [0,10])
```

Question 5. Écrire une fonction `trace_NSEO(p,x,y)` qui interprète la chaîne de caractères `p` comme un programme où les lettres `N`, `S`, `E` et `O` signifient respectivement *avancer d'un pas vers le Nord*, *Sud*, *Est*, *Ouest* (en utilisant la convention usuelle que le Nord est orienté vers le haut sur le dessin). Tous les autres caractères seront ignorés. La fonction devra tracer une ligne brisée en utilisant `plt.plot`, en partant des coordonnées `(x,y)`, et renvoyer les coordonnées de la dernière position atteinte comme un couple `(x,y)`. La fonction ne devra **pas** appeler `plt.figure()`, uniquement `plt.plot(...)`.

```
def trace_NSEO(p, x, y):
    "Écrivez votre code ici"
```

Par exemple, le code suivant trace un angle, avec un segment vers le haut puis deux vers la droite, en partant de `(1,1)` et renvoie `(3,2)`:

```
plt.figure()
assert trace_NSEO("NEE", 1, 1) == (3, 2)
```

Le code suivant doit tracer une figure en forme de L avec le coin en (0,0) et terminer en position (0,0):

```
plt.figure()
assert trace_NSEO("NNESESOO", 0, 0) == (0, 0)
```

La figure suivante doit être un escalier de pente $\frac{1+\sqrt{5}}{2}$ (je ne demande pas de justification de cette affirmation, mais n'hésitez pas à y réfléchir):

```
plt.figure()
escalier = substitution_iteree("N", {"N": "E", "E": "EN"}, 10)
(x, y) = trace_NSEO(escalier, 0, 0)
print(y/x)
```

3. Transformer des programmes

Dans les questions suivantes, on demande de manipuler des chaînes de caractères représentant des programmes de dessin. La fonction `trace_NSEO` pourra être utiliser pour vérifier la validité des résultats.

Question 6. Écrire une fonction `rotation_gauche(p)` qui prend un programme de dessin pour `trace_NSEO` et renvoie le programme correspondant au même dessin tourné d'un quart de tour vers la gauche.

```
def rotation_gauche(p):
    "Écrivez votre code ici"
```

Le code suivant doit tracer quatre fois l'escalier pour former un genre de carré penché. Le code vérifie que la position finale est bien (0,0) et qu'après quatre rotations on retrouve bien l'escalier initial. Vous ne devez pas modifier ce code, il vous sert de test.

```
plt.figure()
initial = substitution_iteree("N", {"N": "E", "E": "EN"}, 10)
escalier = initial
(x, y) = (0, 0)
for i in range(4):
    (x, y) = trace_NSEO(escalier, x, y)
    escalier = rotation_gauche(escalier)
print(x == 0, y == 0, escalier == initial)
```

Question 7. Dans le test précédent, les quatre escaliers sont de couleurs différentes parce qu'ils correspondent à quatre tracés différents, parce qu'il y a un appel à `trace_NSEO` par côté. Modifier le code pour qu'il y ait un seul appel à `trace_NSEO`.

```
"Écrivez votre code ici"
```

4. Ajouter des instructions

On veut maintenant étendre notre langage avec de nouvelles instructions: `A` pour *Avancer*, `G` pour *tourner à Gauche* et `D` pour *tourner à Droite*. Ces instructions feront donc référence à une orientation courante, comme dans le dessin à la tortue, `G` et `D` permettant de changer d'orientation s'un quart de tour dans un sens ou dans l'autre, et `A` permettant d'avancer dans la direction courante.

Question 8. Écrire une fonction `interprete_AGD(p, d)` qui remplace dans le programme `p` les lettres `A`, `G` et `D` par des lettres parmi `N`, `S`, `E`, `O` correspondant à des directions absolues. Le paramètre `d` donne la direction initiale, en utilisant ces lettres. Toute autre lettre doit rester inchangée. Notez que `G` et `D` n'apparaissent plus dans la sortie et que les `A` ne sont pas toujours remplacés par les mêmes lettres.

```
def interprete_AGD(p, d):
    "Écrivez votre code ici"
```

Quelques exemples pour tester:

```
assert interprete_AGD("AAGADADAGA", "E") == "EENESE"
assert interprete_AGD("NADADAEDA", "N") == "NNESEO"
assert interprete_AGD("NADA", "O") == "NON"
```

Un exemple qui interprète un programma avec `AGD` et le trace:

```
plt.figure()
p = substitution_iteree("A", {"A": "AGADADAGA"}, 4)
trace_NSEO(interprete_AGD(p, "E"), 0, 0)
```

Question 9. On veut maintenant ajouter deux instructions: `L` pour *Lever le crayon* et `P` pour *Poser le crayon*. Si on peut lever le crayon, le dessin peut consister en plusieurs lignes brisées.

Définir une fonction `trace_NSEOLP(p, x, y)` qui se comporte comme `trace_NSEO` mais fait autant de tracés que nécessaire. On considère qu'initialement le crayon est posé.

```
def trace_NSEOLP(p, x, y):
    "Écrivez votre code ici"
```

Par exemple, le code suivant doit dessiner trois carrés.

```
plt.figure()
(x, y) = trace_NSEOLP("NESOLEEPENOSLSPESON", 0, 0)
print(x == 2, y == -1)
```