Debuter avec les chaines Python

March 29, 2022

Ecrire un programme qui va manipuler les trames capturées n'est pas véritablement compliqué, mais encore faut-il avoir un minimum de bases sur la manipulation des chaînes de caractères en Python. Ici encore un certain nombre de pré-requis simples sont nécessaires. Il vous est largement conseillé de lire ces quelques éducatifs qui vont grandement vous simplifier la vie pour la suite de la SAE24.

1 Chaîne de caractères en Python

Commençons par le début.
On peut initialiser une chaîne en utilisant les simples ou les doubles quotes:

[1]:
```
chaine1="ceci est une première chaine"
chaine2='ceci est une deuxième chaine'
print(chaine1)
print(chaine2)
```

```
ceci est une première chaine
ceci est une deuxième chaine
```

C'est extrêmement pratique lorsque la chaîne contient l'autre caractère !

[2]:
```
chaine1="pour gérer l'apostrophe ..."
chaine2='pour gérer les "guillemets" anglais'
print(chaine1)
print(chaine2)
```

```
pour gérer l'apostrophe …
pour gérer les "guillemets" anglais
```

Une chaîne de caractères est tout simplement un tableau de caractères.
Chaque caractère peut être invoqué séparément par son index (attention, la numérotation commence à 0).
L'exemple ci-dessous devrait en faire la preuve:

[4]:
```
chaine="Il fait beau pour la saison !"
# La fonction "len" retourne la longueur de la chaîne:
print(len(chaine))
print(chaine)

# On saute une ligne avec un print() "vide"
```

```
print()

# On affiche quelques caractères, sachant que le premier caractère a l'index 0:
print(chaine[0])
print(chaine[1])
print(chaine[8])
print()

# Le dernier caractère a le rang correspondant à la (longueur de la chaîne -1)
print(chaine[len(chaine)-1])
```

```
29
Il fait beau pour la saison !

I
l
b

!
```

2 Quelques petits trucs à savoir

Les chaînes se manipulent très facilement en Python. Bien plus facilement que dans beaucoup d'autres langages. Par exemple, la concaténation de deux chaînes est très simple:

[30]:
```
ch1 = "voici le début"
ch2 = "voilà la fin"
ch = ch1 + ch2
print(ch1)
print(ch2)
print(ch)
```

```
voici le début
voilà la fin
voici le débutvoilà la fin
```

Et pour que ce soit plus propre, on rajoute une chaîne entre les deux ...

[16]:
```
ch=ch1 + " et " + ch2
print(ch)
```

```
voici le début et voilà la fin
```

Attaquons maintenant les subtilités ...
Il est **IMPORTANT** voire **ESSENTIEL** de bien comprendre la différence entre un **tableau de caractères** et un **tableau d'octets**.

Mais pour bien comprendre cela, il faut comprendre deux notions préliminaires:
- **Le typage automatique des variables Python (et ses conséquences sur les fonctions)**
- **Le codage interne des caractères en mémoire (ASCII, UTF-8, ...)**

3 Typage automatique des variables Python

Commençons par une démonstration simple sur le typage automatique de Python:

[5]:
```
# a et b sont créées et initialisées comme des ENTIERS
a=3
b=5
print(a+b)

# a et b sont créées et initialisées comme des CARACTÈRES (ou des chaînes de 1␣
 ↪seul caractère, ce qui est pareil)
a='3'
b='5'
print(a+b)

# la fonction int() transforme une caractère en entier ...
print(int(a)+int(b))
```

```
8
35
8
```

En conséquence, la fonction “print” n’a pas le même comportement ...
- Pour deux entiers, le signe “+” signifie “addition” (la somme est faite avant de l’afficher) - Pour deux chaînes, le signe “+” signifie “concaténation” (les deux chaînes sont concaténées avant d’afficher le résultat) - Enfin, en utilisant la fonction int, on peut transformer un caractère en entier, mais c’est juste pour information, ce n’est pas le propos ici.

Tant qu’on y est, voici une proriété intéressante de la fonction printf: l’utilisation de la lettre “f” en première position de print(f’ ... ‘) permet d’insérer des variables ou instructions entre accolades dans le texte. On peut utiliser l’apostrophe (’) ou les guillements anglais (“) comme vu précédemment pour délimiter le texte. On refait l’exercice précédent en plus présentable:

[8]:
```
# a et b sont créées et initialisées comme des ENTIERS
a=3
b=5
print(f'La somme des deux entiers est: {a+b}')

# a et b sont créées et initialisées comme des CARACTÈRES (ou des chaînes de 1␣
 ↪seul caractère, ce qui est pareil)
a='3'
b='5'
print(f"La somme des deux chaînes est: {a+b}")

# la fonction int() transforme une caractère en entier ...
print(f'La somme des deux chaînes converties en entiers est: {int(a)+int(b)}')
```

```
La somme des deux entiers est: 8
La somme des deux chaînes est: 35
La somme des deux chaînes converties en entiers est: 8
```

4 Représentation interne des caractères

Un octet ne pouvant stocker que des valeurs numériques (de 0 à 255, ce n'est pas une nouvelle…), chaque caractère affichable est représenté en mémoire par une valeur numérique correspondante.
La convention la plus connue (et la plus longtemps utilisée) est le code ASCII (American Standard Code for Information Interchange).
Vous pouvez, par exemple, jeter un coup d'oeil sur la table de correspondance ASCII ici:
https://fr.wikibooks.org/wiki/Les_ASCII_de_0_%C3%A0_127/La_table_ASCII
Le code ASCII d'un caractère peut être affiché grâce à la fonction ord():

[75]:
```
# Voici le code ASCII affiché en décimal et en hexadécimal de plusieurs␣
↪caractères:

print(f'le code ASCII de A est {ord("A")}, soit {hex(ord("A"))} en hexa')
print(f"le code ASCII de 5 est {ord('5')}, soit {hex(ord('5'))} en hexa")

a='3'
b='Z'
print(f'le code ASCII de {a} est {ord(a)}, soit {hex(ord(a))} en hexa')
print(f'le code ASCII de {b} est {ord(b)}, soit {hex(ord(b))} en hexa')
```

```
le code ASCII de A est 65, soit 0x41 en hexa
le code ASCII de 5 est 53, soit 0x35 en hexa
le code ASCII de 3 est 51, soit 0x33 en hexa
le code ASCII de Z est 90, soit 0x5a en hexa
```

Le problème de ce code ASCII est le nombre limité de valeurs … ce qui ne convient pas au codage de tous les caractères de tous les alphabets de tous les langages du monde (sans compter les alphabets arabe, cyrillique, … qui ont des caractères totalement différents).

La solution adoptée par le code ASCII a consisté :
- à réserver la première moitié des valeurs (de 0 à 127) pour des caractères "relativement" standards (lettre minuscules, majuscules, chiffres, et quelques autres)
- a laisser la deuxième moitié (de 128 à 255) à la discrétion de chaque pays, pour coder ses caractères spécifiques comme pour l'alphabet français qui comporte des caractères accentués (é,è,ê,à,â,î,ç,…) qui n'existent pas dans d'autres langages.

Ainsi sont nées les différentes normes ISO 8859-n ou le "n" définit globalement l'utilisation de la deuxième partie en fonction des pays.

Mais 256 caractères sont vraiment insuffisants, même pour un seul pays, qui peut avoir des besoins supérieurs.

Dans le but de créer un codage international unique, des chercheurs se sont creusés la tête pout trouver un système permettant de coder tous les caractères spécifiques internationaux qui sont au nombre de plusieurs centaines de milliers.

Et c'est ainsi qu'est né l'UNICODE …

Sans rentrer dans les détails, l'UNICODE n'est pas un seul codage mais une nouvelle famille de codages, dont le plus utilisé aujourd'hui en europe (pour nos caractères) est l'UTF-8.

Les UNICODES permettent de coder plus d'un million de caractères et répondent ainsi à tous les besoins. Du coup, comme ça ne rentrera jamais dans un seul octet … le nombre d'octets utilisés pour coder un caractère est devenu variable ! ET C'EST LÀ QUE LES ENNUIS COMMENCENT !!!

L'UTF-8 fonctionne selon ce principe:
- il utilise les mêmes 128 premières valeurs que l'ASCII, ce qui le rend compatible, - il utilise les 128 autres valeurs comme des préfixes qui indiquent le nombre d'octets utilisés pour les autres caractères …

Un bon exemple valant mieux qu'un long discours …
- Le caractère "e" est codé par la valeur 65 comme en ASCII (soit \x41 en hexa. Le préfixe \x est équivalent à 0x, il indique une valeur hexa. Vous trouverez les deux notations selon les fonctions, elles sont équivalentes) - Le caractère "ê" est codé par les deux valeurs \xC3 \xAA (C3 indique que le caractère est codé sur deux octets) - Le caractère "û" est codé par les deux valeurs \xC3 \BB - Le caractère "é" est codé par les deux valeurs \xC3 \A9

Pour l'alphabet français, vous trouverez rarement des caractères codés sur plus de deux octets. Cependant, le premier caractère n'est pas toujours \xC3 (même si c'est très souvent le cas).

Retenez donc ceci: **en UTF8, un caractère accentué est codé sur DEUX octets**

4.1 Tableau de CARACTERES vs tableau d'OCTETS

Avez tout ce que nous venons de voir, vous devriez être en mesure de comprendre les subtilités suivantes (qui vont vous être d'une aide inestimable pour mener à bien la SAE24 …)

[13]:
```
#La chaîne A est initialisée en tant que "tableau de caractères"
chaineA = 'Tête brûlée'
print(chaineA)
print(f'La longueur de chaineA est: {len(chaineA)}')

print("")

# La chaîne B est initialisée en tant que "tableau d'octets" grâce à la méthode␣
 ↪encode() qui permet
# tout simplement de convertir le TYPE de chaineA de "tableau de caractères" en␣
 ↪"tableau d'octets",
# sans rien changer au contenu
# A SAVOIR: Comme Python utilise UTF8 par défaut, decode() est équivalent à␣
 ↪decode('utf8')
chaineB = chaineA.encode()
print(chaineB)
print(f'La longueur de chaineB est: {len(chaineB)}')
```

```
Tête brûlée
La longueur de chaineA est: 11

b'T\xc3\xaate br\xc3\xbbl\xc3\xa9e'
La longueur de chaineB est: 14
```

La première chose à remarquer est la suivante:
Python affiche 11 comme longueur de la chaîneA. Pourtant, vous savez maintenant que 14 octets sont utilisés pour la coder, mais il vous épargne (heureusement) ce genre de détails. La fonction "len()" (comme print() dans l'exemple précédent) est prévue pour se comporter différemment en fonction du type de l'opérande.
- Pour une chaîne de caractères, la fonction renvoie le nombre de "caractères" de la chaîne, indépendamment de la façon dont chaque caractère est stocké en mémoire.
- En revanche, pour un tableau d'octets, le fonction "len()" retourne le nombre réel d'octets que la chaîne occupe en mémoire.

La deuxième chose à remarquer concerne la fonction "print()". On le savait déjà, le comportement de cette fonction diffère selon le type de l'opérande (souvenez vous de l'exemple plus haut avec le "+" sur des entiers et des caractères).
Ici encore, c'est vérifié: - Pour un tableau de caractères (une chaîne), la fonction print interprète l'UTF-8 et affiche les caractères à l'écran (et donc les caractères accentués).
- Pour un tableau d'octets, aucune interprétation n'est due … cependant (et ça peut dérouter …) "print()" affiche quand même des caractères si la valeur de l'octet est comprise entre 0 et 128 (caractère ASCII standard). Sinon, la valeur de l'octet est affichée en hexadécimal.

AFIN D'EVITER TOUTE AMBIGUITE, l'affichage d'un tableau d'octets commence **TOUJOURS** par la lettre "b" suivie d'une apostrophe.
Ceci devrait vous expliquer le résultat que vous obtiendrez régulièrement en affichant les champs des trames capturées sur le réseau, les valeurs des champs ne correspondant pas toujours à des caractères affichables, c'est pour cela que les champs correspondent à des tableaux d'octets... il vous faudra donc les gérer en tang que tel et penser à les "décoder" pour pouvoir afficher les chaînes affichables.

Pour rappel et information: - La méthode "encode()" permet de changer un tableau de caractères en tableau d'octets.
- La méthode "decode()" permet de faire la conversion inverse

[14]:
```
#Les chaînes A1 et A2 sont initialisées en tant que "tableau de caractères"
chaineA1 = 'ceci est un message simple'
chaineA2 = 'éàêçèîâûïö'

#Les chaînes B1 et B2 sont initialisées en tant que "tableau d'octets"
chaineB1 = chaineA1.encode()
chaineB2 = chaineA2.encode()


print(f'Chaine A1: {chaineA1} de longueur: {len(chaineA1)}')
print(f'Chaine B1: {chaineB1} de longueur: {len(chaineB1)}')
print()
print(f'Chaine A2: {chaineA2} de longueur: {len(chaineA2)}')
print(f'Chaine B2: {chaineB2} de longueur: {len(chaineB2)}')
```

```
Chaine A1: ceci est un message simple de longueur: 26
Chaine B1: b'ceci est un message simple' de longueur: 26

Chaine A2: éàêçèîâûïö de longueur: 10
```

```
Chaine B2: b'\xc3\xa9\xc3\xa0\xc3\xaa\xc3\xa7\xc3\xa8\xc3\xae\xc3\xa2\xc3\xbb\xc
3\xaf\xc3\xb6' de longueur: 20
```

Comme vous pouvez le voir, pour une chaîne ne comportant QUE des caractères standards, chaque caractère étant codé sur un seul octet, la différence ne peut se faire que grâce à la lettre “b” et aux apostrophes qui trahissent le type du tableau.
En revanche, pour une chaîne ne comportant QUE des caractères spécifiques, chaque caractère étant ici codé sur 2 octets … la différence de longueur est plus flagrante !!!

Bon … si vous n’avez pas compris ça, relisez bien. Sinon, demandez à ceux qui ont compris … mais tant que vous n’aurez pas compris, la SAE24 risque d’être compliquée !
Si vous avez compris, voici une dernière précison pour être complet:

Comme dit plus haut, sauf configuration contraire, Python utilise UTF8 par défaut pour coder ses chaines de caractères.
Pour cette raison, la méthode decode() ne fait que changer le type de la variable qui contient la chaîne …
MAIS, decode() accepte un paramètre permettant de préciser un autre système de codage. Dans ce cas, la chaîne sera codée dans ce nouveau système avant de changer le type de la variable. :

[22]:
```
chaineA1 = 'ceci est un message simple'
chaineA2 = 'éàêçèîâûïö'

# Les chaînes B1 et B2 contiendront des codages UTF-8, ... mais c'est déjà le␣
 ↪cas en Python !
# Le résultat est donc le même qu'avec decode() sans paramètre
chaineB1 = chaineA1.encode('utf8')
chaineB2 = chaineA2.encode('utf8')
print(f'Chaine A1: {chaineA1} de longueur: {len(chaineA1)}')
print(f'Chaine B1: {chaineB1} de longueur: {len(chaineB1)}')
print(f'Chaine A2: {chaineA2} de longueur: {len(chaineA2)}')
print(f'Chaine B2: {chaineB2} de longueur: {len(chaineB2)}')
print()

# Les chaînes B1 et B2 contiendront des codages ISO-8859 (ASCII Latin)
# Un caractère = un octet
chaineB1 = chaineA1.encode('8859')
chaineB2 = chaineA2.encode('8859')
print(f'Chaine A1: {chaineA1} de longueur: {len(chaineA1)}')
print(f'Chaine B1: {chaineB1} de longueur: {len(chaineB1)}')
print(f'Chaine A2: {chaineA2} de longueur: {len(chaineA2)}')
print(f'Chaine B2: {chaineB2} de longueur: {len(chaineB2)}')
print()
```

```
Chaine A1: ceci est un message simple de longueur: 26
Chaine B1: b'ceci est un message simple' de longueur: 26
Chaine A2: éàêçèîâûïö de longueur: 10
Chaine B2: b'\xc3\xa9\xc3\xa0\xc3\xaa\xc3\xa7\xc3\xa8\xc3\xae\xc3\xa2\xc3\xbb\xc
3\xaf\xc3\xb6' de longueur: 20
```

```
Chaine A1: ceci est un message simple de longueur: 26
Chaine B1: b'ceci est un message simple' de longueur: 26
Chaine A2: éàêçèîâûïö de longueur: 10
Chaine B2: b'\xe9\xe0\xea\xe7\xe8\xee\xe2\xfb\xef\xf6' de longueur: 10
```

5 Découpage d'une chaîne

Pour terminer, nous allons utiliser une méthode très puissante dont dispose tout tableau de caractères ou d'octets: la méthode "split()"

[23]:
```
# On initialise une chaine de caractère:
chaineA1 = 'ceci est un message simple'
print(chaineA1)

# On découpe la chaîne selon le caractère " " (espace). Ce caractère est appelé␣
 ↪"motif de césure"
mots = chaineA1.split(" ")
print(mots)

# La chaîne découpée est stockée dans une liste, que l'on peut manipuler comme␣
 ↪un tableau ...
# on peut par exemple afficher chaque élément de la liste, un par un:
print()
print(mots[0])
print(mots[1])
print(mots[2])
print(mots[3])
print(mots[4])

# ou en faisant une boucle
print()
for mot in mots:
    print(mot)
```

```
ceci est un message simple
['ceci', 'est', 'un', 'message', 'simple']

ceci
est
un
message
simple

ceci
est
un
```

```
message
simple
```

[24]:
```
# Dans cet exemple, on découpe la chaîne selon le motif "e" ...
# Ça ne sert probablement à rien mais ça montre que c'est possible !
# Remarquez quand même que le caractère "e" a disparu de la chaine, ce qui␣
 ↪toujours le cas du motif de césure,
# en revanche, le caractère "espace" (qui n'est plus le motif de césure) reste␣
 ↪présent dans les sous-chaînes.

chaineA1 = 'ceci est un message simple'
stupide = chaineA1.split("e")
print(stupide)

# On redécoupe le 3ème élément selon le motif " " (espace)
print(stupide[2].split(" "))
```

```
['c', 'ci ', 'st un m', 'ssag', ' simpl', '']
['st', 'un', 'm']
```

La méthode split() accepte un deuxième paramètre: le nombre de motif de césure à prendre en compte. Par exemple on peut lui dire de ne prendre en compte que le, ou les deux, ou les trois premiers caractères, … :

[25]:
```
chaineA1 = 'ceci est un message simple'
print(chaineA1.split(" ",1))
print(chaineA1.split(" ",2))
print(chaineA1.split(" ",3))
print(chaineA1.split(" ",4))
```

```
['ceci', 'est un message simple']
['ceci', 'est', 'un message simple']
['ceci', 'est', 'un', 'message simple']
['ceci', 'est', 'un', 'message', 'simple']
```

6 Indexation d'une partie d'une chaîne

Un autre moyen de sélectionner une partie de chaîne peut se faire en indexant les caractères sélectionnés, selon la syntaxe [x:y]. Ce moyen simple peut très souvent être utile, mais il faut connaître la position des caractères concernés. Voici quelques exemples :

[100]:
```
chaineA1 = 'ceci est un message simple'
# On affiche les caractères du 10ème au 15ème
# ATTENTION: le premier porte l'index 0 et le dernier n'est jamais affiché ...
print(chaineA1[9:15])

# Ne pas mettre le premier index revient à commencer au début
print(chaineA1[:15])
```

```
# Ne pas mettre le deuxième index revient à finir à la fin
print(chaineA1[15:])
```

```
un mes
ceci est un mes
sage simple
```

Les valeurs des index peuvent également être négatives. Dans ce cas, on numérote depuis la fin de la chaîne:

[103]:
```
chaineA1 = 'ceci est un message simple'
# On affiche les caractères du 5ème au 2ème, mais en comptant depuis la fin.
# ATTENTION: comme avec des valeurs positives, le caractère correspondant au␣
 ↪deuxième index n'est pas affiché
print(chaineA1[-5:-2])

# Ne pas mettre le premier index revient à commencer au début
print(chaineA1[:-3])

# Ne pas mettre le deuxième index revient à finir à la fin
print(chaineA1[-8:])
```

```
imp
ceci est un message sim
e simple
```

Voilà … il y a sans doute encore beaucoup de choses à apprendre sur les chaînes de caractères, mais l'essentiel est dit. En tout cas, suffisamment pour pouvoir travailler sur les trames !