Mémo d'assistance au cours de cartographie avec Python

Éric Guichard

Avril 2023

Note Les documentations pour `python` ne se comptent plus et leur utilité est incontestable. Par exemple la documentation officielle de python : `https://www.python.org`. J'apprécie `https://fr.scribd.com/document/356740978/courspython3-pdf`.

Ce document informel est un « pense-bête » adapté à un cours de cartographie tenu à l'Enssib.

Ceci n'est pas une documentation pour `python`.

1 Apostrophes et guillemets

En anglais : *single and double quotes* : ' et ".

Attention Il m'arrivera peut-être d'écrire ’ au lieu de '. En ce cas, ne cherchez pas à reproduire l'apostrophe typographique dans vos scripts, utilisez l'apostrophe simple.

1.1 Absence de différences

Une variable texte peut indifféremment être détaillée entre guillemets ou entre apostrophes :

`a="bonjour"` ou `a='bonjour'`

Remarque J'aurais dû écrire `a='bonjour'` mais cela me prend beaucoup plus de temps. D'où l'« erreur » évoquée au paragraphe précédent...

Dans la pratique, on essaie d'utiliser le délimiteur qui est le moins utilisé.

Si j'écris `a='aujourd'hui'`, j'induis une erreur : il y a 3 apostrophes. Je préfèrerai alors écrire `a="aujourd'hui"`.

Une autre solution consiste à *protéger* l'apostrophe du mot :

`a='aujourd\'hui'`

Par exemple, si j'ai besoin d'insérer dans une variable l'expression `<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"` j'écrirai `a='<svg xmlns="http...xlink"` 'car il n'y a pas d'apostrophes dans cette expression pleine de guillemets.

1.2 Cas des variables avec plusieurs lignes

Deux solutions :

— avec des sauts de ligne encodés : `a='première ligne\nseconde ligne'` ou `a="première ligne\nseconde ligne"`.

— en utilisant 3 guillemets :
```
a="""un paragraphe séparé
par de banals
retours chariots"""
```
produit le résultat escompté. Cette solution fonctionne aussi avec 3 apostrophes.

Note Perl n'aurait pas eu le même comportement : ce dernier langage interprète ce qui est entre guillemets, mais pas ce qui est entre apostrophes : dans le premier cas `\n` produit un saut de ligne, dans le second, il produit un `\` suivi d'un `n`.

1.3 Longs commentaires

Comme python ne tient pas compte des formes textuelles non assignées à une variable, tout paragraphe entre 3 apastrophes ou guillemets sera pris comme un commentaire (détournement fréquent) :
```
''' Ici
un long
commentaire'''
```

2 Surprises

2.1 Concaténation

Si j'écris `a="bonjour"` et `b=" à tous"` un `print (a+b)` donnera `bonjour à tous`, ce qui est intuitif.

Mais si j'écris `c=2`, un `print (a+b+" "+c)` me renverra un message d'erreur, alors que si j'avais écrit `c="2"`, j'aurais obtenu comme imaginé `bonjour à tous 2`.

Évidemment, si j'écris `d=4` et `print (c+d)`, j'obtiendrai 6.

Il faut convertir `c` en chaîne de caractères pour l'intégrer en une autre (`string`) : `print (a+b+" "+str(c))`

2.2 D'une chaîne à un nombre

L'inverse peut se produire quand on lit une série dans un fichier : ce qu'on croit être un nombre `n` sera pris comme un caractère et on ne pourra pas l'utiliser pour des calculs. Cf. `Ex3diable.py`.

En ce cas, la fonction (pour un entier) est `int()`. Ex. : `int(n)`.

Pour s'y retrouver, la fonction `type`, qui précise le type de la chose manipulée, est très utile :

`print (type(a))` donne `<class 'str'>`

`print (type(c))` donne `<class 'int'>`

`print (type(2.5))` donne `<class 'float'>`

`l=[2,5]` suivi d'un `print (type(l))` donne `<class 'list'>`.

2.3 Extrapolation

Soit la série d'instructions suivantes, séparées par des lignes.

```
e="2"  f=5  g="."  h=e+g+str(f)  print (h)  i=float(h)
```

Un `print (type(i/2))` donnera `<class 'float'>`. Ouf!

2.4 Aparté sur les flottants

Il est très difficle d'écrire tous les nombres possibles avec un ordinateur. Nos machines connaissent en général des nombres entiers (*integer : int*) et des nombres avec un certain « nombre » fixé de chiffres après la virgule.

Par exemple, un `print (2/3)` donnera `0.6666666666666666`.

Le nombre de chiffres après la virgule peut être fixé :
`print ("{:.2f}".format(2/3))` donnera `0.67`.

Mais ce nombre dépend aussi de divers paramètres (processeur, puissance de la machine, etc.). Et on a parfois des surprises. Donnons deux exemples, avec la factorielle :

`fact(n)`, aussi écrit `n!` est le produit de tous les nombres entre 1 et n :
`4!` = 1 * 2 * 3 * 4 = 24.
`5!` = 1 * 2 * 3 * 4 * 5 = 120.

On remarque vite que `n!`=`n * (n-1)!`. Par exemple, `5! = 5 * 4!`. Et donc que $\frac{n!}{(n-1)!} = n$.

Cette évidence peut mener à des résultats **croustillants** : des erreurs de calcul quand les nombres sont très grands ou très petits. Donnons un exemple.

```
def facto(n):
    resu=1
    for i in range (2,n+1):
        resu=resu*i
    return resu

for a in range (30,50):
b=a+1
c=1/facto(a)
d=facto(b)
print ("le résultat suivant doit valoir ",b)
print(d*c)
```

Dès que `a` vaut 30, on a des surprises : `le résultat suivant doit valoir  31`
`30.999999999999996`

Et pour `a > 170`, on obtient la réponse suivante :
`OverflowError: int too large to convert to float`.

Le résultat n'est pas si mauvais : `170!` est un nombre composé de 307 chiffres...

3 Lire, écrire (dans) un fichier

3.1 Écrire simplement

`fichier=open("cercles.html", "w")` (`"w"` pour *write*) crée le fichier `cercles.html` sur le disque dur.

Pour le programme, son nom sera `fichier`. Pour y inscrire la variable `a`, il suffira d'écrire `fichier.write(a)`.

3.2 Lire simplement

Reste à savoir comment on va traiter le fichier lu. La solution la plus simple est d'utiliser une *library* (`csv.reader`, via un `import csv`).

Si le fichier s'appelle `monfic`, un
`nomlocalnDuFichiers=csv.reader(open("monfic",'r',encoding='UTF-8'))`
suffira.

Souvent, on précise le délimiteur :
`nomlocalnDuFichiers=csv.reader(open("monfic",'r',`
`encoding='UTF-8')), delimiter='\t')`.

3.3 Exemple de lecture avec `csv.reader`

On gagne la suppression des sauts de ligne et une lecture « automatisée ».

```
nomInterneDuFichier = csv.reader(
open("cercles", 'r', encoding='UTF-8'), delimiter='\t')

totaldeslignes = list(nomInterneDuFichier)
#lignes du type [2,3,5]
for lignecourante in totaldeslignes:
    print (lignecourante)
    coord1,coord2,rien=lignecourante #grâce au délimiteur \t
    #la variable inutile (rien) est indispensable...
    print (coord2)
#rappel print (coord2+4) ne fonctionnera pas
```

3.4 Exemple de lecture sans `csv.reader`

```
nomInterneDuFichier = open("cercles", 'r', encoding='UTF-8')
sommelignes = list(nomInterneDuFichier)
for lignecourante in sommelignes:
    print (lignecourante)
    coord1,coord2,rien=lignecourante.split('\t')
    print (coord2,rien)
#un saut de ligne en trop
    print (coord2, "<-coord2")
```

```
rien=rien.replace('\n',"")
print (rien," <-rien")
```

Il faut donc être plus prudent. D'autant que python (contrairement à perl) a besoin du nombre exact de variables créées par le séparateur. D'où la sollicitation de la variable `rien`, au nom explicite.

Solution Mettre tout ce qui ne nous intéresse pas dans une liste, qui absorbera en vrac tous les objets restants

```
listerien=[] #on définit cette liste
for... :
    x,y,listerien=lignecourante.split('\t')
```

4 Formatage

Et usages de la commande `format`

4.1 Premier exemple

Tiré du web.

```
stock = ['papier', 'enveloppe', 'chemise', 'encre']
print("Nous avons de l'{} et du {} en stock\n".format(stock[3],stock[0]))
#autre solution, moins lisible:
print("Nous avons de l'{0[3]} et du {0[0]} en stock\n".format(stock))
```

4.2 Exemple simplissime

```
print ("x vaut {}, y vaut {} et z vaut {}".format('2','3','4'));
```

Donne comme résultat
`x vaut 2, y vaut 3 et z vaut 4`.

Très utile pour le svg...

5 Nettoyages

On a déjà vu le `replace` (`r=r.replace('\n',"")`).

`strip` est utile quand il s'agit de travailler sur des objets comme des noms de fichiers qu'on veut débarrasser des espaces (␣) qui parfois les enveloppent. Ex. :
`a="␣fichier␣␣"`
`a=a.strip()`
`print ("T"+a+"T")` renvoie `TfichierT`.

Paramètres complémentaires :

`strip('liste de caractères bordants à enlever')` Ces caractères peuvent être dans le désordre mais tous doivent être présents (y compris l'espace au besoin).

Attention `strip` enlève les *whitespaces*, souvent traduits par « espaces » en français. En fait, un *whitespace* est un caractère blanc, donc : une ou plusieurs espaces ␣, une tabulation ou un saut de ligne.

En bref, pour un `a="Essai␣␣ \n"` (des espaces, des tabulations et un saut de ligne à droite), `a.strip()` renverra un `Essai` tout court.

Seconde remarque La gestion des espaces (et des sauts de ligne) par `python` peut s'avérer désarçonnante. Par exemple, `print ("XXX","YYY")` affiche un `XXX YYY` (introduction d'une espace).

5.1 Boucles et conditions

Car des `for`, `if`, `while`, etc.

5.1.1 Listes ordonnées

Si j'écris `liste=range(0,10)` et si je demande l'affichage des éléments de la liste, **10 ne sera pas affiché** ! En fait, cette `liste` contient dix nombres, de **0 à 9** !

Preuve-exemple :

```
liste=range(0,10)
for i in liste:
    print (i)
print ("Attention "+str(len(liste))+" n'est pas affiché")
```

5.2 Sorties de boucles ou conditions

On usera de 3 types d'instructions : `pass` (peu utile), `continue` (sortie temporaire) et `break` (sortie définitive).

Exemples

```
for i in liste:
    if i==3:
            break
    print (i)
```

Seront affichés : 0,1,2.

*

```
for i in liste:
    if i==3:
        continue
    print (i)
```

Seront affichés : 0,1,2, puis 4, 5... 9.

*

```
for i in liste:
    if i %2==0: #si i divisé par 2 donne un reste nul
        print (str(i)+ " est pair")
    else:
        pass
```

Les lignes contenant `else` et `pass` sont en fait inutiles. Mais elles peuvent être confortables

— quand on n'est pas sûr de soi (pas de `if` sans `else`),

— si on veut remettre à plus tard les instructions du `else`.