Python : mes notes

DATE

Ce fichier, outre mon usage personnel, me permet de comparer l'utilisation de QUARTO avec celle de TEXMACS pour carnet de code style JUPYTER.

TABLE DES MATIÈRES

1. Affichages

`print` sur une seule ligne

```
>>> L=list(range(10))
    print(*L)
    for x in L:print(x,end="␣")
    print("")
    print('␣'.join([str(x) for x in L]))
    print('␣'.join(map(str, L)))
    import string
    alphabet = string.ascii_lowercase
    print(alphabet)
    print(*list(alphabet))
```

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
abcdefghijklmnopqrstuvwxyz
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

```
>>>
```

affichage nombres et textes mélangés

Méthode la plus récente et « moderne » avec les `f-strings`, super pratiques, introduites avec 3.6. On met les variables entre `{}` :

```
>>> h2g2,onze = 42,11
    print(f"voici␣{h2g2}␣&␣{onze}")#
    print(f"2^(-{onze})␣=␣{2**(-onze):.6f}")#6 décimales
```

```
voici 42 & 11
2^(-11) = 0.000488
```

```
>>>
```

◦ Méthodes plus anciennes : (cliquer sur ◦)

formatage des nombres

```
>>> #{:09.4f}
    #0 signifie : on rajoute des zéros
    #9 signifie : 9 caractères en tout y compris la virgule (sauf grand nombre)
    #.4 signifie : 4 chiffres après la virgule
    from random import *
    L=[1.2,1.987654321,3,0.222,999888777.111222333]#liste à arrondir
    for x in [1.2,1.987654321,3,0.222,999888777.111222333]:print(f"{x:09.4f}", end="␣")
    print("")
    print(*list(map(lambda x: round(x, 2), L)))#utilisation de map
    print(*[round(x, 2) for x in L])#compréhension de liste
    print(*[f"{x:.3f}" for x in L])#noter les 0 forcés pour faire 3 décimales
    print(*[f"{10**randint(1,5):09.0f}" for _ in range(10)])#entiers avec 0 à gauche pour 9 digits
  0001.2000 0001.9877 0003.0000 0000.2220 999888777.1112
  1.2 1.99 3 0.22 999888777.11
  1.2 1.99 3 0.22 999888777.11
  1.200 1.988 3.000 0.222 999888777.111
  000000100 000010000 000000010 000000100 000100000 000000010 000100000 000100000 000000010 000000010
>>>
```

2. `input` multiples

Ce code fonctionne très bien, mais TEXMACS (tout comme QUARTO) gère mal les `input` c'est pourquoi le chunk suivant, bien que théoriquement fonctionnel, ne peut pas être exécuté ici dans l'environnement TEXMACS :

```
>>> coeff=input("3␣coeff␣séparés␣par␣des␣virgules␣:␣")
    print(coeff)
    (a,b,c)=coeff.split(',')
    print(a,b,c)
```

3. itérateurs

Certaines fonctions produisent des *itérateurs* : `zip`, `range`, `enumerate`, `map`, `filter`, `reversed`

Pour exploiter le résultat il faut « casser » l'itérateur selon l'une des trois manières suivante :

- `list`(*iterateur*)
- `for` *x* `in` *iterateur*
- **itérateur*

¡¡ un *itérateur* n'est « cassable » qu'une seule fois (sauf `range`) (comme les vieilles tirelires en porcelaine…) !!

```
>>> def truc(x):return(100*x)#fonction numérique pour map
    def machin(x):return(x%2==0)#fonction booléenne pour filter
    [a,b,c,d,e,f]=[
          range(3),zip([1,2,3],["u","v","w"]),
          enumerate(["t","o","t","o"]),map(truc,[1,2,3]),
          filter(machin,[31,32,33,34,35]),reversed([5,6,7])
                ]
    print(a,b,c,d,e,f)#objets bizarres
    print("--")
    print(*a,*b,*c,*d,*e,*f)
    print("==")
    print(*a,*b,*c,*d,*e,*f)#seul range est "cassable" plusieurs fois
  range(0, 3) <zip object at 0x118ad7980> <enumerate object at 0x10d890b30> <map object at
  0x118a9da20> <filter object at 0x118accb50> <list_reverseiterator object at 0x118acfb80>
  --
  0 1 2 (1, 'u') (2, 'v') (3, 'w') (0, 't') (1, 'o') (2, 't') (3, 'o') 100 200 300 32 34 7 6 5
  ==
  0 1 2
>>>
```

4. Listes

utilisation de *

```
>>> #concaténation
    A=[1,2]
    B=[3,4]
    print([A,B])#liste de listes
    print([*A,*B])#liste unique
```

```
[[1, 2], [3, 4]]
[1, 2, 3, 4]
```

```
>>> print(A)
    print(*A)#affichage sans les crochets
```

```
[1, 2]
1 2
```

```
>>>
```

Cela marche aussi très bien pour les tuples ou les dictionnaires.

Trier une liste

`L.sort()` modifie la liste `L` tandis que `sorted(L)` non :

```
>>> L = [4,2,1,3]
    print(f"L={L}")
    L.sort()#c'est une méthode : cela modifie la liste
    print(f"L.sort() a modifié la liste : L={L}")
    shuffle(L)
    print(f"shuffle(L) a modifié la liste aussi : L={L}")#modifie la liste aussi
    print(f"sorted ne modifie pas L :sorted(L)={sorted(L)} mais L={L}, ")
```

```
L=[4, 2, 1, 3]
L.sort() a modifié la liste : L=[1, 2, 3, 4]
shuffle(L) a modifié la liste aussi : L=[2, 4, 3, 1]
sorted ne modifie pas L :sorted(L)=[1, 2, 3, 4] mais L=[2, 4, 3, 1],
```

```
>>>
```

5. Boucles

mot clé `continue`

saute une étape de la boucle en cours (la plus ``intérieure")

```
>>> for j in range(2):
        for i in range(20):
            if(i<=18):continue
            print(i,j)
```

```
19 0
19 1
```

```
>>>
```

6. Maths

exponentielle complexe

La fonction `exp` du module `math` ne traite pas les complexes et produira une erreur, il faut faire attention, donc, aux `from math import *` si l'on veut par la suite faire de l'exponentielle complexe :

```
>>> import cmath
    import math
    print(cmath.exp(1j))
```

```
(0.5403023058681398+0.8414709848078965j)
```

```
>>> from math import *
    from cmath import exp as expC
    print(expC(1j))
```

```
(0.5403023058681398+0.8414709848078965j)
```

```
>>>
```

7. manipulation des chaînes

supprimer au début et à la fin

la méthode `.strip` ne modifie pas la variable. son effet est de supprimer les caractères demandés, à la fin et au début

attention, si l'on applique `strip` à une ligne extraite d'un `open` il se peut que Python considère que la ligne se termine par un `\n` donc la ligne que vous voyez comme, par exemple, `altitude,1500;` ne donnera rien après un `strip(";")` parce que, pour Python, il est possible que `;` soit l'avant-dernier caractère seulement (le dernier étant l'invisible retour chariot `\n`) ce qui veut dire que Python voit `altitude,1500;\n` .

Alors il faut faire `strip(";\n")` pour enlever le `;` (et le `\n` du coup)

```
>>> line="\n␣␣␣␣;;;\n␣␣␣␣␣␣bon;jour;;;;;;␣␣␣␣␣;;;;;\n;;;␣␣␣␣"
    line = line.strip('\n;␣')#enlève récursivement \n, espaces, et point-virgules en début et fin
    print(line)
    texte="abdjsqbdjksq123dsklqm"
    alphabet = "abcdefghijklmnopqrstuvwxyz"
    texte=texte.strip(alphabet)
    print(texte)
  bon;jour
  123
>>>
```

obtenir l'alphabet

```
>>> alphabet = [chr(i) for i in range(ord('a'), ord('z') + 1)]#liste
    print(*alphabet)#méthode manuelle

    import string
    alphabet = string.ascii_lowercase# c'est␣une␣chaîne
    print(alphabet)
    print(*alphabet)#la chaîne a été convertie en liste

  a b c d e f g h i j k l m n o p q r s t u v w x y z
  abcdefghijklmnopqrstuvwxyz
  a b c d e f g h i j k l m n o p q r s t u v w x y z
>>>
```

convertir chaîne en liste

```
>>> a="girafe"
    print(*a)
    print(list(a))
    print(*list(a))
  g i r a f e
  ['g', 'i', 'r', 'a', 'f', 'e']
  g i r a f e
>>> a="ezea;ez␣a;aze;xd␣w;c;t;f\nd"
    print(f"split␣avec␣;␣␣␣->␣␣␣{a.split(";")}")
    print(f"split␣sans␣paramètre␣->␣␣␣{a.split()}")
    print(f"=split␣avec␣retour␣chariot␣␣->␣␣␣{a.split("\n")}")
    print(f"split␣avec␣espace␣␣␣->␣␣␣{a.split(" ")}")
  split avec ;   ->   ['ezea', 'ez a', 'aze', 'xd w', 'c', 't', 'f\nd']
  split sans paramètre ->   ['ezea;ez', 'a;aze;xd', 'w;c;t;f', 'd']
  =split avec retour chariot  ->   ['ezea;ez a;aze;xd w;c;t;f', 'd']
  split avec espace   ->   ['ezea;ez', 'a;aze;xd', 'w;c;t;f\nd']
>>>
```

fichiers `.csv`

convertir fichier `.csv` en dictionnaire

```
>>> def cvs2dict(filename):#filename est le path/nom du fichier
        d = {}
        with open(filename) as file:#file va être un ensemble de lignes de texte
            for line in file:
                line = line.strip('\n␣')#on supprime les \n et espaces au début/fin des lignes
                if not line or line[0] == '#':#si ligne est vide ou commence par # on saute
                    continue
                (a,b,c) = line.split(';') #on suppose que le .csv contient trois colonnes séparées par
    des point-virgules, ça il faut le vérifier avant en ouvrant le .csv
                d[a] = (b, c)
        return d
```

```
>>>
```

Calcul matriciel

voici un exemple par compréhension de liste (aucune boucle `for`)

```
>>> #--------- produit scalaire et produit matrice par vecteur --------
    def dot(X, Y):
        #X,Y deux listes de même taille
        assert (len(X) == len(Y)), f'Dimensions incompatibles len(X)={len(X)}≠len(Y)={len(Y)}'
        return sum([x * y for (x, y) in zip(X, Y)])  # Ou sum(x*y for x,y in zip(X,Y))

    def matVect(A, X):
      # A = matrice (liste de listes) de dimensions mxn
      # X = liste de dimension n
        assert (len(A[0]) == len(X)), f"Dimensions incompatibles {len(A[0])}≠{len(X)}"
        return [dot(L, X) for L in A]
>>> dot([1,2,3],[10,10,0])
  30
>>>
```

Ici on a représenté la matrice A comme une liste de lignes, i.e. pour $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ on aurait `A=[[1,2],[3,4]]`.

***args et **kwargs**

Prendre en argument une variable nommée `*truc` permet de gérer autant d'arguments qu'on veut. Toute entrée est alors considérée comme un tuple :

```
>>> def product(*L):
        x=1
        for y in L:
            x=x*y
        return(x,type(L))
    print(product(11))
    print(product(1,2,3,4))
    print(product([1,2,3,4]))
    print(product(*[1,2,3,4]))
  (11, <class 'tuple'>)
  (24, <class 'tuple'>)
  ([1, 2, 3, 4], <class 'tuple'>)
  (24, <class 'tuple'>)
>>>
```

Prendre une variable `**truc` permet de gérer un dictionnaire

```
>>> def somme(x,y,*args,**kwargs):
        r = x + y
        for i in args: r += i
        for i in kwargs.values(): r += i
        return(r)

    print(somme(1,1,10,10,a=100,b=100,c=100,d=100))
  422
>>>
```

Attention à l'ordre : arguments réguliers < arguments * < arguments **