PAPY_TP2_eleves

September 11, 2023

Ecole Centrale Nantes Option InfoIA PAPY: Programmation avancée python TP2 Année 2023-2024 Auteur: Lucas Lestandi lucas.lestandi@ec-nantes.fr V1.1

1 TP2 : Structuration et types de données

Durée 2+4h

Objectifs: - Mise en application des concepts basiques : listes, dictionnaires, fonctions - Programmation Orientée Objet (POO) - héritage - composition - Mise en œuvre sur problèmes plus complexes

1.1 Exercices d'application

1.1.1 Listes et dictionnaires

 Exercice : Ecrire un script qui transforme une liste de dictionnaires en dictionnaire

Exemple de comportement:

```
data =  [{'Lea': 'java', 'Bobby': 'python'},
         {'Lea': 'php', 'Bobby': 'java'},
         {'Lea': 'cloud', 'Bobby': 'big-data'}]

#résultat
{'Lea': ['java', 'php', 'cloud'], 'Bobby': ['python', 'java', 'big-data']}
```

 Exercice : Ecrire un script qui renvoie les 10 mots les plus fréquents dans un texte a

Zen of python

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
```

```
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Résultat attendu

```
Percentage share of each word :
is                  : 7.30%
better              : 5.84%
than                : 5.84%
to                  : 3.65%
the                 : 3.65%
Although            : 2.19%
be                  : 2.19%
should              : 1.46%
never               : 1.46%
of                  : 1.46%
```

[2]:
```
zen="""Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!"""
```

```
<b> Code Golf Challenge : </b> le but de cet exercice est de proposer la
    solution au même problème qui utilise un nombre minimal de charactères. Les <code>import</
    Pour mesurer la longueur du script, on il suffit de le copier coller dans une chaine et d'
```

1.1.2 Fonctions

```
<b> Exercice : </b>Ecrire une fonction qui calcule la moyenne d'un nombre inconnu d'arguments.
```

```
>>> result1 = avgfun(1, 2, 3)
>>> result2 = avgfun(2, 6, 4, 8)

>>> print(round(result1, 2))
>>> print(round(result2, 2))
2.0
5.0
```

 Exercice : Ecrire une fonction <code>crier</code> qui transforme un str en majuscules

Exemple

```
>>> saluations(crier)
BONJOUR, JE SUIS CRÉÉ PAR UNE FONCTION PASSÉ EN ARGUMENT.
>>> saluations(chuchoter)
bonjour, je suis créé par une fonction passé en argument.
```

 Exercice : Ecrire une fonction décorateur <code>log</code> qui prend en argument une

Exemple d'utilisation du décorateur

```
>>> @log
>>> def add(x, y):
>>>     return x + y

>>> add(3.5, 7)
Fonction add appelée avec les arguments positionnels (3.5, 7) et les arguments par mot clé {}.

10.5
```

1.1.3 Programmation orientée objet

Quelques mots sur les protocoles et le *duck typing* En python, un grand nombre de comportements sont attendus d'un objet (et donc d'une classe), comme la possibilité de l'afficher, de le sauvegarder ou même des opérations plus abstraites. Souvent le paradigme de programmation python est exprimé par la notion de ducktyping.

> *If it walks like a duck and it quacks like a duck, then it must be a duck*

En d'autres termes, le typage est dynamique, il dépend des méthodes et attributs disponibles et non de la classe effectivement utilisée.

[129]:
```
def calcule(a, b, c): return (a+b)*c

a = calcule (1, 2, 3)
b = calcule ('pommes ', 'et oranges, ', 3)

print(a)
print(b)
```

```
9
pommes et oranges, pommes et oranges, pommes et oranges,
```

Tant que les objects savent gérer les opérations `+` et `*`, on peut les appeler sans soucis. Le typage des entrées/sorties de la fonction est purement dynamique.

Dans l'exercice suivant, on va passer en revue quelques unes des méthodes spéciales `__dunder__` pour les appliquer à une classe de vecteurs ainsi que quelques décorateurs de méthode.

Pour la liste complete des méthodes spéciales : https://docs.python.org/3/reference/datamodel.html#special-method-names

`<b> Exercice "Objet Pythonique" (1): </b> A partir du squelette de classe <code>Vecteur</code> Completer les méthodes ci-dessous pour vérifier les tests proposés dans le main.`

> Remarque: on peut définir la classe au choix directement dans le notebook ou dans un fichier séparé. Pour les fichiers séparés, on pourra tester directement "à la main" à l'aide du `if __name__=="__main__:"` ou en important dans ce notebook (recommandé) `python %load_ext autoreload %autoreload 2` et `python from vecteur import Vecteur # puis exécuter les exemples ci dessous`

```
from array import array # On évite ainsi de recourrir à numpy en utilisant un built-in
import math # bien utile pour tout un tas d'opérations

class Vecteur:
    def __init__(self, x, y):
        pass

    def __repr__(self):
        """Représentation textuelle de l'objet"""
        pass

    def __iter__(self):
        """Iterateur sur les élements du vecteur, utilise yield"""

    def __str__(self):
        """Appelé par print()"""
        pass

    def __eq__(self, other):
        """Surcharge de l'opérateur =="""
        pass

    def __abs__(self):
        """surcharge du built-in abs()"""
        pass
```

```
>>> v1 = Vecteur(3, 4)
>>> print(v1.x, v1.y)
3.0 4.0
>>> x, y = v1
>>> x, y
(3.0, 4.0)
>>> v1
```

```
Vecteur(3.0, 4.0)
>>> v1_clone = eval(repr(v1))  # la bonne façon de cloner c'est avec copy. Ici on montre l'uti
>>> v1 == v1_clone
True
>>> print(v1)
(3.0, 4.0)
>>> abs(v1)
5.0
```

[73]:
```
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[74]:
```
from vecteur import Vecteur
```

[75]:
```
v1 = Vecteur(3, 4)
print(v1.x, v1.y)  #3.0, 4.0
x, y = v1
print(x,y)
print(Vecteur(3.0, 4.0)) # (3.0, 4.0)
v1_clone = eval(repr(v1))  # la bonne façon de cloner c'est avec copy. Ici on␣
 ↪montre l'utilité d'avoir un __repr__ qui écrit l'appel au constructeur de␣
 ↪classe combiné avec eval qui evalue un str en python.
v1 == v1_clone #True
print(v1)   #(3.0, 4.0)
print(abs(v1)) #5.0
v1 #(3.0, 4.0)
```

```
3.0 4.0
3.0 4.0
(3.0, 4.0)
(3.0, 4.0)
5.0
```

[75]:
```
Vecteur(3.0, 4.0)
```

```
<b> Exercice "Objet Pythonique" (2): </b> Ajouter les méthodes méthode d'initialisation qui ac
```

```
from array import array # On évite ainsi de recourrir à numpy en utilisant un built-in
import math # bien utile pour tout un tas d'opérations

class Vecteur:
    def __init__(self, x, y):
        ...
    ...

    @classmethod
```

```
    def from_iterable(cls, it):
        pass


    @classmethod
    def from_polar(cls, r: float,theta: float):
        pass
from math import pi
v2=Vecteur.from_iterable([1,5])
print(v2)
v2=Vecteur.from_polar(1,pi/4)
print(v2)
```

[76]:
```
from math import pi
v2=Vecteur.from_iterable([1,5])
print(v2)
v2=Vecteur.from_polar(1,pi/4)
print(v2)
```

```
(1.0, 5.0)
(0.7071067811865476, 0.7071067811865475)
```

 Exercice "Objet Pythonique" (4): protéger les variables x et y dérrière des <code>_x</

```
>>> v3=Vecteur(1,2)
>>> print(v3.y)
>>> v3.y=1
>>> try:
...     v3.y="1"
... except Exception as e:
...     print("catching exception:", e)
2.0
catching exception: val='1' must be a real
```

[77]:
```
v3=Vecteur(1,2)
print(v3.y)
v3.y=1
try:
    v3.y="1"
except Exception as e:
    print("catching exception:", e)
```

```
2.0
catching exception: val='1' must be a real
```

 Exercice "Objet Pythonique" (5): Surcharge des opérateurs <code>==</code>, <code>+</c

```
>>> print(f"{(v1==v2)=}")
(v1==v2)=False
>>> print(f"{(v1==[3,4])=}")
```

```
(v1==[3,4])=False
>>> print(f"{(v1==Vecteur(3,4))=}")
(v1==Vecteur(3,4))=True

>>> print("\nTest de l'addition de vecteurs:\n----------------------")
>>> print(f"Attendu: (3.0, 4.0)+(1.0, 1)=(4.0, 5.0) \nCalculé: {v1}+{v3}={v1+v3}\n")

Test de l'addition de vecteurs:
----------------------
Attendu: (3.0, 4.0)+(1.0, 1)=(4.0, 5.0)
Calculé: (3.0, 4.0)+(1.0, 1)=(4.0, 5.0)

>>> print([1,3]+v1)
(4.0, 7.0)
>>> "v1"+v1
...
unsupported operand type(s) for +: 'Vecteur' and 'str'
>>> v1+=v2
>>> print(v1)
(3.7071067811865475, 4.707106781186548)
>>> print(-v1)
(-3.7071067811865475, -4.707106781186548)

>>> -v1
(-3.7071067811865475, -4.707106781186548)
>>> Vecteur(1,-1)*v2 #environ 0
1e-16
>>> math.sqrt(2)*v2 #environ (1,1)
Vecteur(1.0000000000000002, 1.0)
```

[120]:
```python
import math
v1 = Vecteur(3, 4)
print(f"{(v1==v2)=}")
print(f"{(v1==[3,4])=}")
print(f"{(v1==Vecteur(3,4))=}")

print("\nTest de l'addition de vecteurs:\n----------------------")
print(f"Attendu: (3.0, 4.0)+(1.0, 1)=(4.0, 5.0) \nCalculé: {v1}+{v3}={v1+v3}\n")
print([1,3]+v1)
try:
    "v1"+v1
except Exception as e:
    print(e)

v1+=v2
print(v1)
print(-v1)
```

```
print(v2.norme)

## produit (scalaire)
print(f"{-v1=} (-3.7071067811865475, -4.707106781186548)")
print(f"{Vecteur(1,-1)*v2=} (0)")
print(f"{math.sqrt(2)*v2=} (1,1)")
```

```
(v1==v2)=False
(v1==[3,4])=False
(v1==Vecteur(3,4))=False

Test de l'addition de vecteurs:
----------------------
Attendu: (3.0, 4.0)+(1.0, 1)=(4.0, 5.0)
Calculé: (3.0, 4.0)+(1.0, 1)=(4.0, 5.0)

(4.0, 7.0)
unsupported operand type(s) for +: 'Vecteur' and 'str'
(3.7071067811865475, 4.707106781186548)
(-3.7071067811865475, -4.707106781186548)
1.0
-v1=Vecteur(-3.7071067811865475, -4.707106781186548) (-3.7071067811865475,
-4.707106781186548)
Vecteur(1,-1)*v2=1.1102230246251565e-16 (0)
math.sqrt(2)*v2=Vecteur(1.0000000000000002, 1.0) (1,1)
```

```
<b> Exercice "Objet Pythonique" (6): </b> pour un vecteur plus complet, ajouter des méthodes <
```

```
>>> v1.angle(v2)
0.11829166559831755
>>> v1.norme
5.9916187242356855
```

[121]:
```
v1.angle(v2)
```

[121]:
```
0.11829166559831755
```

[122]:
```
v1.norme
```

[122]:
```
5.9916187242356855
```

Bonus: La classe vecteur développée ici est en fait assez générale. Modifier cette classe pour qu'elle accepte des vecteurs de taille quelconques. On pourra utiliser une instance composantes la classe array.array pour stocker les données du vecteur. On remarque au passage que cette nouvelle écriture va alléger le code en supprimant les références explicites à x et y en tirant profit de la redéfinition de **iter** et des formes "dépackées" en *composantes.

1.2 Héritage de classe

 Exercice : Créer une classe <code>Printer</code> dont les instances singent le comport

<u>Politique de création d'une imprimante:</u> on entre les formats disponibles et <code>__init

Définir une classe fille :<code>ColorPrinter</code> une imprimante "couleur" qu

```
>>> generic_printer=Printer(avail_paper=["B1","A4","A0"])
Some formats are incorrect :{'B1'}
>>>   Printer.show_printers_system()
"Formats: ['A0', 'A1', 'A2', 'A3', 'A4', 'A5'],  cls.base_width=200, \n cls.devices=["Printer
>>> generic_printer.send(zen)
##################################################
#  Beautiful is better than ugly. Explicit is    #
#  better than implicit. Simple is better than   #
#  complex. Complex is better than complicated.  #
#  Flat is better than nested. Sparse is better  #
#  than dense. Readability counts. Special cases #
#  aren't special enough to break the rules.     #
#  Although practicality beats purity. Errors    #
#  should never pass silently. Unless explicitly #
#  silenced. In the face of ambiguity, refuse    #
#  the temptation to guess. There should be      #
#  one-- and preferably only one --obvious way   #
#  to do it. Although that way may not be        #
#  obvious at first unless you're Dutch. Now is  #
#  better than never. Although never is often    #
#  better than *right* now. If the               #
#  implementation is hard to explain, it's a bad #
#  idea. If the implementation is easy to        #
#  explain, it may be a good idea. Namespaces    #
#  are one honking great idea -- let's do more   #
#  of those!                                     #
##################################################

>>> generic_printer.send(zen,"A0")
###############################################################################################
#  Beautiful is better than ugly. Explicit is better than implicit. Simple is better than comp
#  Readability counts. Special cases aren't special enough to break the rules. Although practi
#  ambiguity, refuse the temptation to guess. There should be one-- and preferably only one --
#  than never. Although never is often better than *right* now. If the implementation is hard
#  are one honking great idea -- let's do more of those!
###############################################################################################
```

On attend de l'intance de ColorPrinter le même comportement mais avec le texte coloré en bleu dans le terminal.

```
>>> fancy_printer=ColorPrinter("Blue printer", color="blue")
```

```
#####################################################
#  Beautiful is better than ugly. Explicit          #
#  is better than implicit. Simple is better        #
#  than complex. Complex is better than             #
#  complicated. Flat is better than nested.         #
#  Sparse is better than dense. Readability         #
#  counts. Special cases aren't special enough      #
#  to break the rules. Although practicality        #
#  beats purity. Errors should never pass           #
#  silently. Unless explicitly silenced. In the     #
#  face of ambiguity, refuse the temptation to      #
#  guess. There should be one-- and preferably      #
#  only one --obvious way to do it. Although        #
#  that way may not be obvious at first unless      #
#  you're Dutch. Now is better than never.          #
#  Although never is often better than *right*      #
#  now. If the implementation is hard to            #
#  explain, it's a bad idea. If the                 #
#  implementation is easy to explain, it may be     #
#  a good idea. Namespaces are one honking great    #
#  idea -- let's do more of those!                  #
#####################################################
```

On peut voir qu'il y aurait un certain nombre d'actions à effectuer pour que le "driver" d'impression soit correct.

1.3 Problème : Affichage du contenu d'un dossier par approche fonctionnelle

```
<b> Exercice : </b> Ecrire une ou des fonction <code>print_dirtree</code> qui prend en argumen
```

cahier des charges:

traiter différement les fichiers et dossier dir_only: bool

profondeur max level: int

nombre max de ligne (pour évider de lister tout le système) length_limit: bool=1000

Exemple

```
>>> print_dirtree("foo/")
foo/
   a.txt
   b.txt
   bar/
      p.txt
      q.txt
   c.txt
```

On utilisera les caractères spéciaux suivant pour l'affichage:

```
space =  '    '
```

```
branch = '    '
tee =    '   '
last =   '   '
```

Bonus: Ajouter du parsing au block `main` pour en faire un vrai programme.

1.4 Problème : Problème de géométrie

```
<b> Exercice : </b> Créer une structure de classes qui permette de manipuler des objets "linéa
<br>
<br>
<u>Remarque:</u> Il est nécessarie de créer un classe <code>Point</code>, une liste de coordonn
<br>
<u>Pour chacun des ses objets (quand c'est possible)</u>:
<ul>
    <li> on souhaite pouvoir les initialiser par leur équation et des coordonnées de point, on
    <li> on souhaite pouvoir savoir si un point appartient à une instance et </li>
    <li> si deux de ces objets s'intersectent en utilisant une méthode/fonction commune, par ex
    <li> on définira une méthode/fonction qui renvoie l'angle entre deux objets linéaires <code
    <li> on définira un opérateur de rotation sur nos objets linéaires <code>rotate(P: List(flo
</ul>
Pour les droites, quelque soit leur mode de définition, on pourra récupérer les représentations
```

Exemple d'utilisation

```
>>> L1=line.vec_init([1,1],[1,0])
>>> L2=line.points_init([0,0],[0,1])
>>> print(L1)
Line object with equation : 0.71x+-0.71y+-0.71=0
with vector:(-0.7071067811865475, -0.7071067811865475) and origin: [1, 0]
>>> print(L2)
Line object with equation : 1.00x+-0.00y+0.00=0
with vector:(-0.0, -1.0) and origin: [0, 0]
>>> np.degrees(lines_angle(L1,L2))
45.0
>>> lines_intersect_point(L1,L2)
[-0.0, -1.0]
>>> rotate2d(np.pi,[-1,0])
array([ 1.0000000e+00, -1.2246468e-16])
```

1.5 Problème : Une approche orienté objet pour afficher l'arborescence

```
<b> Exercice : </b> Ecrire un programme orienté objet qui possède la méthode <code>generate</co
```

Cahier des charges:

traiter différement les fichiers et dossier dir_only: bool

profondeur max level: int

nombre max de ligne (pour évider de lister tout le système) length_limit: bool=1000

Exemple

```
>>> print_dirtree("foo/")
foo/
   a.txt
   b.txt
   bar/
      p.txt
      q.txt
   c.txt
```

On utilisera les caractères spéciaux suivant pour l'affichage:

```
space =  '    '
branch = '   '
tee =    '   '
last =   '   '
```

Suggestion d'approches :

Utiliser une structure d'arbre : cf. cours d'ALGPR EI1 pour représenter l'arborescence et d'autres méthodes/classes pour l'affichage.

Encapsulation de méthodes proche de la fonction précédente avec traitement spécialisé fichier/dossier. On pensera a développer une classe frontale pour gérer les options et l'affichage.

1.6 Afficher l'arborescence dans le terminal

```
<b> Exercice : </b> à l'aide de l'implémentation fonctionnelle ou OOP, créer un programme pytho
```

On pourra utiliser les librairies `argparse` ou bien `getopt/sys.argv` pour la gestions des options/paramètres d'appel et `pathlib` pour gérer le path.

```
$ python tree.py foo
foo/
   a.txt
   b.txt
   bar/
      p.txt
      q.txt
   c.txt
```

```
$ python tree.py foo -d
foo/
   bar/
```

[ ]: