Python

Hadrien Bodin

Mines

9 décembre 2025

Table des matières

Pourquoi Python ?

La lisibilité :

- pas de délimitations begin end ;
- uniquement des indentations
- la lisibilité fait partir de l'ADN du langage
- typage dynamique

Pourquoi Python ?

La puissance du langage :

- types très puissants et flexibles
  - entiers non bornés, nombres complexes
  - listes, strings Unicode
  - tables de hash : dictionnaires et ensembles
  - langage orienté objet : définir ses propres types
- un grand nombre de librairies (qui s'interfacent facilement en C et Cpp)
- gestion automatique de la mémoire

Pourquoi Python ?

La compilation :

- Python est un langage interprété
- Un script s'effectue ligne par ligne
- Cela permet un usage interactif
- Retenir cependant qu'une précompilation existe

Pratiques à adopter

Un code doit être intelligible. Il faut se souvenir qu'un code est plus souvent lu qu'écrit.
Quelques principes à toujours appliquer :

- Keep it short and simple
- Nom des fonctions, variables etc explicite
- Commentaires SSI nécessaires
- Annotations de type

Règles de nommage

Un relecteur doit être capable de comprendre l'intérêt d'un objet à partir de son seul nom.

- On peut écrire des noms "longs"
- Choisir une unique langue pour la nomenclature

Le lecteur a un "contexte d'attention". Si une variable n'est utilisée que sur quelques lignes, on peut se permettre de la nommer de façon moins explicite.

Règles de nommage

Un relecteur doit être capable de comprendre l'intérêt d'un objet à partir de son seul nom.

- On peut écrire des noms "longs"
- Choisir une unique langue pour la nomenclature

Le lecteur a un "contexte d'attention". Si une variable n'est utilisée que sur quelques lignes, on peut se permettre de la nommer de façon moins explicite.

Règles de nommage

- On peut écrire des noms "longs"
- Choisir une unique langue pour la nomenclature

Les variables et les objets

- Variables et fonctions : minuscule + underscore
- Constantes : Tout en majuscule
- Classe : Majuscule + minscule

```
position_x, position_y, masse_1, masse_2 # des variables
NOMBRE_PLANETES # une constante
get_acceleration() # une fonction
Simulation() # une classe
```

Les commentaires

Bien penser à aérer le code (sauter des lignes quand nécessaire).
Commentaire pour ce qui est nécessaire : souvent les noms de variables suffisent.

Annotation de type

Il est de bonne pratique d'annoter les types des variables attendues et renvoyées par une fonction.

- Pour la lisibilité
- Pour le contrôle des erreurs

On déclare le type de chaque paramètre de la fonction, ainsi que le type de ce qui est retourné par la fonction.

Annotation de type

```
def get_item(my_list : list[float], pos : int = 2)
    -> float:
    return my_list[pos]
```

Un bon IDE

Configurer son IDE pour travailler proprement.

Les variables

En Python, une variable peut être vue comme une adresse vers un objet. Ce qui est derrière cette adresse peut changer. Cela n'a aucune importance pour l'interpréteur.

```
a = '1'
type(a)
    -> 'str'
a = int(a)
type(a)
    -> 'int'
```

La mutabilité

Un objet mutable est un objet qui peut-être modifié une fois créé.
Les entiers ne sont pas mutables

```
a = b = 1
a += 1
a is b
    -> False
```

a pointe initialement vers l'objet 1.
Comme cet objet n'est pas mutable, modifier la valeur de a correspond à créer un nouvel objet (2) et à faire pointer a vers ce nouvel objet.

La mutabilité

A contrario, les listes, par exemple, sont mutables

```
a = b = [1]
a.append(2)
a is b
    -> True
print(b)
    -> [1, 2]
```

Argument des fonctions

Il est parfois utile d'utiliser des arguments par fonction dans une fonction.
La valeur d'un tel argument est évalué à la création de la fonction.

Attention

Attention aux objets mutables en valeur par défaut

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
    -> [1]
print(f(2))
    -> [1, 2]
print(f(3))
    -> [1, 2, 3]
```

Argument des fonctions

- Les fonctions peuvent avoir des arguments obligatoires ou non.
- Les arguments peuvent être nommés ou positionnels.
- Les arguments non obligatoires ont des valeurs par défaut.
- A la définition, il faut d'abord déclarer les arguments obligatoires puis les autres.
- A l'appel, l'ordre des arguments nommés n'importe pas

Arguments obligatoires

```
def f(a, b):
    print(f"{a}_{b}")
f(1,2)
    -> 1_2
f(a=1, b=2)
    -> 1_2
f(b=2, a=1)
    -> 1_2
f(1)
    -> f() missing 1 required positional argument: 'b'
```

Argument des fonctions

- Dans un appel de fonction, les arguments nommés doivent suivre les arguments positionnés.
- L'ordre des arguments nommés (obligatoires ou non) n'importe pas.

Les *args et **kwds permettent de passer un nombre arbitraire d'arguments (réciproquement non nommés et nommés).

Gestion des erreurs

Vous avez pu le voir, il est commun d'avoir des erreurs dans notre code.
Soit car on les rencontre, soit car on souhaite les causer.
Il faut distinguer les erreurs de syntaxe et les erreurs d'exécution.
ex : indentation manquante vs division par zéro

Catcher une erreur

Parfois on est conscient qu'une erreur peut survenir et on ne souhaite pas faire crasher le programme si on la rencontre.
La syntaxe try, except sert à cela.
On essaie de faire quelquechose, en s'attendant à un échec.

Lever une erreur

A l'inverse, il peut être pertinent de lever une erreur. C'est souvent utile pour prévenir l'utilisateur qu'il essaie d'appeler la fonction d'une façon non prévue par l'implémentation.
On utilise la commande raise avec le nom de l'erreur.

Créer notre propre type d'objet

Une fois ces bases posées, il est possible de s'intéresser à la programmation orientée objet.
Pour décrire un objet complexe, il semble pertinent d'avoir besoin de recourir à une structure de donnée complexe.

Exemples

Quel type python choisir pour décrire un ou une élève ? Une classe ? Une école ?

Les classes

Pour définir de nouveau types, on utilise une classe.
La classe peut être vue comme un générateur d'objets.
Tous les objets issus de la même classe partagent les mêmes propriétés (mais pas forcément les mêmes valeurs).

Les classes

Les objets issus d'une classes contiennent :

- Des attributs
- Des méthodes

```
a = np.array([1])
a.shape
a.mean()
```

Créer une classe

Le décorateur à utiliser est **class**

```
class Eleve:
    age = 25
    def say_hello():
        print("Hello")

e = Eleve
e
    -> <class '__main__.Eleve'>
e = Eleve()
e
    -> <__main__.Eleve object at 0x0000019FE45C5010>
```

Ici on pointait dans un premier temps vers la classe, et ensuite vers un objet instancié par la classe.
Retenir que toutes les méthodes doivent à minima contenir l'argument self.

Accéder aux éléments de la classe

On peut accéder aux différents éléments de la classe.

```
e = Eleve()
e.say_hello()
    -> Hello
e.age
    -> 25
e.age = 30
e.age
    -> 30
e.note = 15
e.note
    -> 15
```

Attention, si changer un attribut ou en rajouter un à la volée est possible, cela est souvent déconseillé.
Il faut souvent considérer les attributs comme privés : Seule la classe elle même devrait avoir le droit de les modifier.

init

L'instanciation en appelant un objet classe crée un objet vide.
Il peut cependant être pertinent de créer des instances personnalisées correspondant à un état initial spécifique.
À cet effet, une classe peut définir une méthode spéciale nommée __init__(). L'instanciation de la classe appelle automatiquement cette méthode.

```
class Eleve:
    def __init__(self):
        self.age = 25
    def say_hello():
        print("Hello")
```

init

On comprend tout de suite qu'il est pertinent de donner des arguments à la méthode init.

```
class Eleve:
    def __init__(self, age, nom):
        self.age = age
        self.nom = nom
    def say_hello():
        print("Hello")

e = Eleve(25, "Hadrien")
```

Variables de classe

En général, les variables d'instance stockent des informations relatives à chaque instance alors que les variables de classe servent à stocker les attributs et méthodes communes à toutes les instances de la classe.

Attention à ne pas abuser des variables de classe.

```
class Eleve:
    ecole = 'Mines'          # class variable shared by all instances
    def __init__(self, nom):
        self.nom = nom     # instance variable unique to each instance

>>> d = Eleve('eleve_1')
>>> e = Eleve('eleve_2')
>>> d.ecole                  # shared by all dogs
'Mines'
>>> e.ecole                  # shared by all dogs
'Mines'
>>> d.nom                   # unique to d
'eleve_1'
>>> e.nom                   # unique to e
'eleve_2'
```

Variables de classe

Les objets mutables peuvent présenter des surprises

```
class Eleve:
    ecole = 'Mines'             # class variable shared by all
    notes = []
    def __init__(self, nom, note):
        self.nom = nom      # instance variable unique to eac
        self.notes.append(note)

e = Eleve('h', 15)
e.notes
    -> [15]
d = Eleve('i', 20)
e.notes
    -> [15, 20]
```

Il vaut donc mieux toujours utiliser des variables d'instances

repr

Pour une raison ou une autre, on peut avoir envie de print un objet.

```
print(d)
    -> <__main__.Eleve object at 0x000002721492041O>
```

Par défaut le résultat n'est pas très pertinent... Pour cela il existe la méthode __repr__.
Le retour de cette fonction est affiché par le print.

repr

```
class Eleve:
    def __init__(self, nom):
        self.nom = nom
    def __repr__(self):
        return f"eleve: {self.nom}"

e=Eleve("Hadrien")
print(e)
    -> Eleve: Hadrien
```

call

Comme son nom l'indique cette méthode permet d'appeler directement un objet de la classe. A titre d'exemple, si notre classe sert à décrire une fonction, la méthode call permet d'évaluer cette fonction. Évidemment, la méthode call peut prendre autant d'arguments que souhaité.

Exercice

Créer une classe Polygone sans utiliser numpy :

- Un polygone est constitué d'un ensemble de Points (Classe à écrire)
- A sa création, on lui adjoint un nom. Une liste de coordonnées peut par ailleurs être renseignée (non obligatoire).
- Ecrire une méthode pour ajouter un point
- Ecrire une méthode pour supprimer le i-ème point
- Ecrire une méthode pour récupérer la position du i-ème point
- Ecrire une méthode pour translater le polygone le long d'un vecteur
- Un print de l'objet doit afficher, entre autres, le nombre de points, et le barycentre

ERREURS

Revenir au raise

Exemples

Revenir au raise

Héritage

Si deux classes sont proches, on peut vouloir faire hériter une classe fille d'une classe mère :
On peut par exemple avoir une classe "Personne" qui contient des informations sur quelqu'un.
Une classe "Eleve" aurait donc toutes les raisons de partager des similarités avec la classe "Personne".

Héritage

```
class Personne:
    def __init__(self, age, nom):
        self.age = age
        self.nom = nom

    def say_hello():
        print('hello je suis un humain')

    def say_goodbye():
        print('goodbye')

class Eleve(Personne):
    def __init__(self, age, nom, ecole):
        self.ecole = ecole
        super().__init__(age, nom)

    def say_hello():
        print('hello je suis   tudiant  ')
```

Les méthodes spéciales

Nous avons défini ce qu'est une méthode d'une classe.
Il existe une certaine famille de méthodes dont les noms sont réservés.
On les appelle les dunders. Elles se caractérisent car les noms sont entourés d'underscore.

- __init__()
- __repr__()

Ces méthodes sont normalisées et doivent toujours avoir le même comportement.

Opérateurs

Les opérateurs classiques que l'on utilise en python peuvent être vus comme de simple décorateurs syntaxique.
Derrière leur exécution, se cache l'appel à une fonction.
Par exemple, sommer deux entiers revient à appeler une méthode de la Classe int qui prend un autre int en paramètre.

Opérateurs

On comprend donc qu'il peut être pertinent de redéclarer ces méthodes pour les classes que l'on construit.
Par exemple, il peut être pertinent de pouvoir sommer deux vecteurs.

Déclaration d'un opérateur

```
class My_int:
    def __init__(self, val):
        self.val = val
    def __add__(self, my_other_int):
        return self.val + my_other_int.val
```

Les opérateurs

Il existe un grand nombre d'opérateurs qu'il est possible de définir. Les slides suivantes mettent en avant une liste non exhaustive.

Les opérateurs de conversion

Pour changer le type d'un objet vers un objet de type courant :

- __str__(self)
- __int__(self)
- __float__(self)
- __bool__(self)
- __dict__(self)

Note : La conversion vers un booléen est notamment utile si on veut appeler une condition sur l'objet (if object :)

Les opérateurs unaires

Pour faire une opération sur l'objet seul :

- __pos__(self)
- __neg__(self)
- __abs__(self)

Les opérateurs arithmétiques

Pour faire une opération entre deux objets :

- __add__(self, other) # self + other
- __sub__(self, other) # self - other
- __mul__(self, other) # self * other
- __truediv__(self, other) # self / other
- __floordiv__(self, other) # self // other
- __mod__(self, other) # self % other

Bien sûr, il est souvent nécessaire de faire un test sur le type de la variable other pour pouvoir faire le calcul.
Auquel cas, il peut être pertinent de renvoyer la valeur NotImplemented pour les cas non pris en charge.

Les opérateurs arithmétiques

```
class My_int:
    def __init__(self, val):
        self.val = val
    def __add__(self, my_other_int):
        if isinstance(my_other_int, int):
            return self.val + my_other_int.val
        return NotImplemented
```

Les opérateurs arithmétiques

Ces méthodes existent aussi avec le prévixe 'r' pour right.
Ces méthodes seront appelées si l'objet est présent à droite d'un opérateur et l'objet à gauche ne supporte pas l'opération.

```
a = 3 #int
b = My_int(2)

b+a # ok car definit pour la classe b
a+b # not ok car non d finit pour les int
    # -> il faut impl menter __radd__
```

Les opérateurs arithmétiques

Ces méthodes existent aussi avec le prévixe ’i’ pour inplace.
Elles permettent d’implémenter toute la famille des +=, *= etc etc

```
b = My_int(2)
b+=1 # besoin d'implémenter __iadd__
```

Les opérateurs de comparaison

Enfin, certaines méthodes sont implémentables pour définir des comparaisons entre deux objets, bien que cela n'ait pas toujours de sens. Pour qu'une séquence d'objets soit triable, il faut à minima définir l'opérateur d'égalité et d'infériorité stricte.

- __eq__(self, other) # self == other
- __ne__(self, other) # self != other
- __lt__(self, other) # self < other
- __le__(self, other) # self <= other
- __gt__(self, other) # self > other
- __ge__(self, other) # self >= other

Les opérateurs d'accession

Un objet peut se comporter comme un conteneur. Cela signifie que l'objet contient plusieurs autre objet (comme une liste contiendrait des int).
Auquel cas, certaines méthodes sont à définir pour revenir au comportement classique d'un tel conteneur.

- __len__(self) # self == other
- __getitem__(self, key) # objet[key]
- __setitem__(self, key, value) # objet[key] = value
- __delitem__(self, other) # del objet[key]
- __contains__(self, key) # kei in objet

Itérateur

Si on a un objet de type Ecole, qui contient une liste d'Eleves, on peut, avoir envie de boucler sur les élèves.
La conteneurisation précedemment décrite permet de boucler sur les élèves de cette façon :

```
ecole = Ecole()
for i in range(len(ecole)):
    eleve = ecole[i]
```

Au demeurant, la méthode de boucle sur un index n'est pas la meilleure.
On préfère en général se ramener à une expression de la forme :

```
ecole = Ecole()
for eleve in ecole :
```

Cela n'est possible que si la classe Ecole() définit un iterateur.

Définir un itérateur

Pour définir un itérateur, il faut implémenter les méthodes suivantes :

- __iter__
- __next__

Iter

Cette fonction doit renvoyer un objet qui est itterable. Cet objet peut être un des attributs de la classe mère

Next

Cette fonction doit renvoyer le prochain objet que l'on souhaite récupérer.
Une fois tous les objets récupérés, cette fonction doit raise un StopIteration

Les modules

Nous avons maintenant vu comment créer des classes et des fonction puissantes, modulables.
Pour mettre à disposition ces fonctions aux utilisateurs, on utilise des modules.
Par exemple, numpy et pandas sont des modules.
Le but est ainsi de pouvoir faire un :

```
from pyecole import Ecole
```

Importer un module

Il existe plusieurs façon d'importer des fonctions depuis un module

```
import module
module.fonction()
```

ou

```
import module as md
md.fonction()
```

Importer un module

Il existe plusieurs façon d'importer des fonctions depuis un module

```
from module import fonction
fonction()
```

ou

```
from module import fonction as func
func()
```

ou

```
from module import * # a ne pas faire
```

Importer un module

La fonction dir() permet de lister tous les noms définis par un module.

```
import module
dir(module)
```

Importer un module

A l'import d'un module, l'interpréteur python va cherche un module du bon nom dans cet ordre :

- Les modules par défaut dans sys.builtin_module_names
- Puis dans sys.path qui comprend :
  - Le dossier courant
  - Ce qui est pointé par PYTHONPATH
  - site-packages

Créer un module

Si on a deux fichiers dans le même répertoire (par exemple utils.py et main.py), on peut avoir besoin des fonctions déclarées dans utils dans le main.
On peut alors directement, depuis main, exécuter :

```
import utils
```

Le chemin utilisé est le chemin relatif par rapport au dossier courant.

Créer un module

Pypi est le gestionnaire public de module. Il sert à exposer ses modules au public, qui sont instalables via pip install. Lorsque l'on crée un module / un package, il faut veiller à ce que son nom soit unique et ne corresponde à aucun autre module présent sur pypi.

Créer un module

Si votre module contient plusieurs scripts (ou un seul), il faut les mettre dans un même dossier, qui constituera votre package.
Le nom du dossier correspond au nom du package.

```
import utils #avant
import packageutils.utils
```

Créer un module

Admettons que notre package comporte deux fichiers utils.py et helpers.py Si utils nécessite les fonctions déclarées dans helpers, on peut les importer de la sorte :

```
from .helpers import myfunction
```

Ici le .helpers signifie "va chercher le submodule helpers qui est dans le même module que moi".

Créer un module

Il faut déclarer un fichier __init__.py pour déclarer à Python que ce dossier est un module.
Dans le init, on peut rajouter des attributs supplémentaires. Souvent, on l'utilise pour exposer des constantes propres à ce module.
Le init, permet par ailleurs d'exposer certaines fonctions au niveau du module, et non d'un submodule.

Créer un module

Le init sert notamment à remonter des fonctions au niveau module.

```
from packageutils.utils import myfunction
```

On peut plutôt rajouter dans le init cet import :

```
from .utils import myfunction
```

Ainsi, depuis l'extérieur du module on peut simplement faire :

```
from packageutils import myfunction
```

TODO

- __main__
- argparse
- fin module
- pip
- venv
- uv

Exécution d'un programme

Le monde ne se limite pas à jupyter notebooks.
Un programme est un ensemble de script s'exécutant. Un programme a un début et une fin.

Python est un langage qui peut être déployé en production. Google, Meta, Netflix, Spotify et d'autres ont des backend qui tournent en python.

Différents langages

Différents langages correspondent à différents kernels. Dans chaque kernel il faut savoir identifier :

- Le langage qui lui correspond
- Le scope atteignable par ce kernel

Bash

- A quoi correspond l'invit de commmande bash ?
- Quel langage est interprété ?
- A quoi ça sert ?
- Quel est le scope ?
- Que ce passe-t-il quand on exécute la commande "python" ?

Bash

- Bash est un langage proche de l'OS. Cela permet de "dialoguer avec l'ordinateur".
- Un environnement bash a accès aux fonctions standard, à celles qui sont définies dans un bashrc ou bash_aliases, et à celles potentiellement définies dans le script courant.
- Un environnement bash a aussi accès aux exécutables pointés par les variables d'environnement.
- Lancer un script dans un fichier .sh revient à instancier un nouveau contexte.

- Un environnement python a uniquement accès aux fonctions et variables définies dans cet environnement.
- Lancer un script .py revient à instancier un nouveau contexte.
- Un script .py doit être exécuté avec le programme python.
- Une fois le programme terminé, le contexte est perdu.

Exécution

Depuis un terminal bash :

```
>>> python utils.py
>>> somme(3,4)
    >>> bash: syntax error near unexpected token `3,2'
>>> python make_sum_3_4.py
    >>> 7
```

Exemple

Pourquoi cette erreur ?

A quoi ressemblent les deux scripts utils et make_sum_3_4 ?

Retour sur l'exécutable python

Le programme python :

Extrait de la doc

python
[-bBdEhiIOPqRsSuvVWx ?]
[-c command | -m module-name | script | -]
[args]

Ce qui correspond à :

- Utiliser python
- Avec des options (bBdEhiIOP etc...)
- Pour éxécuter une commande un module un script ou rien
- Avec des arguments

Retour sur l'exécutable python

L'exécuté :

- Laissé vide ouvre un kernel python
- Un nom de fichier va exécuter ce fichier
- -m module-name va exécuter le main d'un module qui se trouve dans le syspath
- -c command va interpréter directement la commande

```
python -c "a=12; print(a)"
    -> 12
python main_printer_12.py
    -> 12
python
-> Python 3.13 etc etc etc
>>> print(12)
    -> 12
```

Retour sur les options

Sont très rarement utilisées. On peut tout de même citer :

- -h pour help
- -V pour version
- -i pour interactive (permet de laisser le kernel ouvert après l'exécution d'un programme)
- -v pour verbose

Compréhension de ce que fait le programme

Exécutez un script quelconque avec l'option -v

```
python -i -c "a=12"
>>> a
12
```

Les arguments

Un programme peut avoir besoin d'arguments :
Ils sont placés après le nom du programme et sont captés par le module argparse (que nous verrons juste après).

```
python print_sum.py 3 4
>>> 7
python print_sum.py 3 2 1
>>> 6
python print_sum.py 2 4 --mean
>>> 3
```

Et c'est la même idée pour beaucoup de programmes !

- git clone "nom_du_repo"
- git commit -m "message"

Le main

Souvent, on essaie d'exécuter un main, souvent nommé main.py

Structure d'un main

Les imports

```
from utils import make_sum
```

Les déclarations de fonctions et de la fonction main

```
def f1(x, y, z):
    ........
def main(arg1, arg2, arg3):
    ........
```

L'appel

```
if __name__ == '__main__':
    #parser d'arguments (d crit juste apr s) qui d finit
    main(arg1, arg2, arg3)
```

name == main

```
if __name__ == '__main__':
    main(arg1, arg2, arg3)
```

La variable __name__ vaut :

- __main__ si le fichier est appelé directement par l'éxécutable
- le_nom_du_fichier sinon (dans le cas où le fichier est appelé par un autre fichier, dans le cas d'un import).

Attention

Veiller à ne jamais mettre de code qui serait exécuté à l'extérieur du name==main

Pourquoi

Selon vous, pourquoi ?

name == main

Si le main.py ressemble juste à ca :

```
argument_global_a = 12
print('Je vais executer une tres longue fonction')
main(argument_global_a)
```

Et que dans un autre fichier main_bis.py on veut récupérer argument_global_a, on va faire :

```
from main import argument_global_a
if __name__=='__main__':
    print(argument_global_a)
```

L'exécution va donner :

```
python main_bis.py
>>> Je vais executer une tres longue fonction
>>> 12
```

Module argparse

Par défaut les arguments sont récupérés par l'argument sys.argv

```
python -c "import sys; print(sys.argv)" a b c --test
    -> ['-c', 'a', 'b', 'c', '--test']
```

Il existe cependant un module qui permet de gérer les arguments de manière plus agréable

```
import argparse
```

Module argparse

Création de l'objet parser qui est une instance de la classe argparse.ArgumentParser

```
parser = argparse.ArgumentParser(
                        prog='Mon programme',
                        description='Qui teste le parser',
                        epilog='Bonne chance !')
parser.parse_args()
```

Exercice

Commencez par écrire ce parser. Attention, veillez bien à mettre cette portion de code dans le main !
Appelez ensuite le script avec l'argument -h (pour help).

Ajouter des arguments

Tout se fait via la méthode parser.add_argument

```
parser = argparse.ArgumentParser(
                        prog='Mon programme',
                        description='Qui teste le parser',
                        epilog='Bonne chance !')
parser.add_argument('a')
args = parser.parse_args()
print(args.a)
```

Un argument qui commence par '-' est optionnel et s'écrit en une lettre.
Un argument qui commence par '–' est optionnel et s'écrit en plusieurs lettres.
La fonction add_argument prend énormément d'options.

Exercice

Ecrire un programme qui prend en entrée 3 mots et qui les concatène

- Le paramètre –repeat -r doit permettre de doubler chaque mot. Par défaut les mots ne sont pas doublés.
- Le paramètre –sep -s doit permettre de spécifier le séparateur utilisé entre les mots qui est un espace par défaut.
- Le paramètre –etoile -e doit permettre de rajouter des étoiles en début de ligne.
  - –etoile (ou -e) rajoute une étoile en début et fin de ligne
  - -ee en rajoute 2
  - -eee en rajoute 3, etc...
- Bien sûr tout doit être documenté et un -h doit afficher une aide explicite.