RÉPUBLIQUE
FRANÇAISE

Liberté
Égalité
Fraternité

100537 : PYTHON PERFECTIONNEMENT

PARTIE 1/2

Institut de la Gestion publique et du Développement économique

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

IGPDE

Objectifs de la formation

- Après la phase d'initiation (stage Python 8487), ce module a pour but:

- De vous sensibiliser sur les bonnes pratiques Python.
- De vous faire découvrir l'éventail des modules Python.
- De vous permettre de délivrer du code efficace et plus robuste.
- De maitriser les différentes notions structurantes de Python

- En résumé:
- « D'aller un peu plus loin dans votre apprentissage »

Objectifs pédagogiques :

- ❑ Maitriser le langage Python
- ❑ Adopter des bonnes pratiques
- ❑ Avoir large vision des possibilités de Python
- ❑ Etre autonome

Le programme

Jour 1

Réactivation

Les signatures des fonctions

La gestion des exceptions

Les modules internes

Jour 2

Les expressions régulières

Python avancé

Objets et classes en Python

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

IGPDE

Présentation du formateur, des stagiaires et du stage

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

IGPDE

Réactivation

Séquence 1

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

IGPDE

Sommaire

- ❑ Rappels
- ❑ PEP8 : bien présenter son code

Rappels

- Jupyter
- Les types élémentaires
- Les fonctions et méthodes

manip_jupyter.ipynb

manip_type.ipynb

manip_fonction.ipynb

Les normes de codage Python: PEP8

PEP = Python Enhancement Proposals

- Définir des règles de développement communes entre développeurs
- Invitation forte pour toute la communauté Python à écrire un code de la même façon
- Gain d'énergie et de temps pour l'appropriation des codes partagés ou repris
- La PEP8 s'attache principalement aux règles stylistiques (Mise en page, indentation, espacement, docStrings, nommage des variables)

Règles

- Une ligne doit contenir au maximum 79 caractères

```
print("-- Je suis une ligne de 80 caractères maximum, pas plus pas moins –")
```

- L'indentation (en Python) → une importance capitale et incontournable : 4 espaces.

```
class FirstClass:
    def __init__(self, luggages):
        self.luggages = luggages

    def travel(self):
        print('Luxe, calme et volupté')
```

- 2 lignes vides doivent être ajoutées entre 2 éléments de haut niveau (ex : 2 classes)
- 1 ligne vide doit séparer chaque fonction (ou méthode)
- Les noms (variable, fonction, classe, ...) ne doivent pas contenir d'accent (que des lettres ou chiffres)

Règles

Espaces dans les instructions (Syntaxe anglosaxone)

- Pas d'espace avant : mais un après
  Exemple : {orange: 2}
- Opérateurs : un espace avant et un après.
  Exemple : i = 1 + 1
- Aucun espace avant et après un signe = lors d'une assignation de la valeur par défaut d'un paramètre de fonction.
  Exemple : def oiseau(bec=True, ailes=2, pattes=2)
- Une seule instruction par ligne.

Docstring

```
# -*- encoding : utf-8 -*-
# tkPhone.py
# import ----------------------------------------------------------------------
import shutil
import sys
from tkPhone_IHM import Allo_IHM

# définition de classe Allo ---------------------------------------------------

class Allo(Allo_IHM) :
    """
        Répertoire téléphonique avec :
           - Affichage d'une personne
           - Ajout d'une nouvelle personne
           - Modification d'une personne sélectionnée
           - Suppression d'une personne sélectionnée
           - Sauvegarde du fichier en cours (./phones.txt)
           - Restauration du fichier sauvegardé (./phonesav.txt)
    """

    def __init__(self, fic="phones.txt", ficsav="phonesav.txt") :
        """
            Constructeur de l'interface graphique.
        """
        super().__init__(fic, ficsav)

    def afficher(self, event=None) :
        """
            Méthode permettant d'afficher la personne sélectionnée
        """
        self.clear()
        valcur = len(self.select.curselection())
```

DocString pour la class Allo

DocStrings pour les méthodes de la Class Allo

- Toujours 3 guillemets ouvrants + Commentaire + 3 guillemets fermants
- Accessible dans l'interpréteur Python avec la commande HELP(nom_de_l'objet)

Commentaires

- Les commentaires en Python sont délimités par #
- Écrire des phrases complètes, ponctuées et compréhensibles
- Le commentaire doit être cohérent avec le code
- Il doit suivre la même indentation que le code qu'il commente
- Ne pas décrire le code, expliquer plutôt à quoi il sert
- Il est « plus que » préconisé d'être écrit en Anglais

Python coders from non-English speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don’t speak your language.

Nommage

- Classes : lettres majuscules en début de mot (Camel case) : MyFirstClass
- Exceptions : similaire aux classes mais avec un Error à la fin. MyGreatError
- Fonctions : minuscules et underscore : my_function()
- Méthodes : minuscules, underscore et self en premier paramètre : my_method(self)
- Arguments des méthodes et fonctions : identique aux fonctions.
  my_function(param=False)
- Variables : identique aux fonctions.
- Constantes : tout en majuscules avec underscore si nécessaire. PI = 3.14

RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

IGPDE

Imports

- 1 module par ligne

  import os
  import sys

  - *On proscrit* : import os, sys

- Ordonnancement et hiérarchie des modules à importer :

  - import de module
  - import de contenu de module
  - import de la lib standard
  - import de libs tierces parties
  - **import de votre projet**

  Exemple :

```
import os          # import module de la lib standard
import sys         # on groupe car du même type

from itertools import islice                # import de contenu de module
fromcollections import namedtuple           # on groupe car du même type

import requests  # import lib tierce partie
import arrow       # on groupe car du même type

from django.conf import settings          # import de contenu du module tierce partie
from django.shortcuts import redirect    # on groupe car du même type

import mon_projet  # import du module de mon projet
```

Pour aller plus loin

https://peps.python.org/pep-0008/

- L'outil : pycodestyle (en ligne de commande) permet de référencer le non respect de la Pep8 sur chaque ligne de code en précisant les n° de lignes et de colonnes.
- Utilisation : >> pycodestyle monprog.py

Toutes les PEP

https://peps.python.org/pep-0000/

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

IGPDE

Les signatures des fonctions et le passage des paramètres

Séquence 2

RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

IGPDE

Sommaire

- ❑ Les modes de déclaration de paramètres
- ❑ Les modes de passage des paramètres
- ❑ Le scope des variables

Les modes de déclaration de paramètres

- La signature des fonctions/méthodes peut prendre la forme la plus générique possible
  - *def fonction(*args, **kwargs) :* **args et kwargs sont des conventions**
- Ou être très précise :
  - *def fonction(longueur, largeur, hauteur, unite) :*
- Il est possible de spécifier des valeurs par défaut
  - *def fonction(longueur, largeur**, hauteur= 5**) :*

**args dans la signature capture tous les paramètres positionnels dans un tuple*

*** kwargs dans la signature capture tous les paramètres mot-clé dans un dictionnaire*

Les modes de passage des paramètres

- Le passage des paramètres peut prendre deux formes
  - Forme positionnelle : *appel de fonction(premier, second)*
  - Forme de mot-cle : *appel de fonction(nom = valeur)*
  - Forme mixte : *appel de fonction(premier, second, nom = valeur)*

Règle 1: Les arguments positionnels en premier

Dans la définition et dans l'appel

Exemple: signature de fonction simple

Entrée [11]:
```
def volume(longueur, largeur, hauteur):
    print('longeur:', longueur )
    print('largeur:', largeur )
    print('hauteur:', hauteur )

    return longueur*largeur*hauteur

print(volume(4,3,2)) # respect de la position
print(volume(4, hauteur = 2, largeur = 3))  # mix position et mot-cle
```

```
longeur: 4
largeur: 3
hauteur: 2
24
longeur: 4
largeur: 3
hauteur: 2
24
```

Entrée [12]:
```
print(volume(hauteur = 2, largeur = 3, 4))  # mix position et mot-cle MAIS ERREUR de preséance
```

```
  File "<ipython-input-12-3fd0451c5503>", line 1
    print(volume(hauteur = 2, largeur = 3, 4))  # mix position et mot-cle
                                           ^
SyntaxError: positional argument follows keyword argument
```

Signatures gloutonnes

manip_argument_1.ipynb

Regroupement total

```
def espace_occupe(*args):
    print('arg:', *args)
    total = 1
    for cote in args:
        total *= cote
    return total
```

```
print('espace:', espace_occupe(2,3))
print('espace:', espace_occupe(5, 7, 8))
print('espace:', espace_occupe(5, 7, 8, 2) )
```

```
arg: 2 3
espace: 6
arg: 5 7 8
espace: 280
arg: 5 7 8 2
espace: 560
```

*args peut se situer à n'importe quel emplacement dans la signature MAIS toujours avant les arguments clé/valeur

Avec kwargs

manip_argument_2.ipynb

```
[28]: def volume3(largeur = 1, matiere = 'vide', **kwargs):
          print(kwargs)
          longueur = kwargs['longueur']
          hauteur= kwargs['hauteur']

          print('longeur:', longueur )
          print('largeur:', largeur )
          print('hauteur:', hauteur )
          print('matiere', matiere)

          return f"{longueur*largeur*hauteur} de {matiere}"
      print(volume3(longueur = 3,hauteur = 2 ))
```

```
{'longueur': 3, 'hauteur': 2}
longeur: 3
largeur: 1
hauteur: 2
matiere vide
6 de vide
```

Règle 2 :**kwargs ne peut se trouver qu'à la fin des parametres

Sinon : syntaxError

Bon à savoir :
L'opérateur * devant une séquence réalise un 'déballage'
de la séquence (tuple, liste)

l'opérateur ** joue ce rôle pour les dictionnaires

```
Entrée [36]: a =(1, 2, 3)
             print(a)
             print(*a,sep=' puis ')

          (1, 2, 3)
          1 puis 2 puis 3
```

Contraintes sur la nature des arguments

Pour n'accepter que des parametres cle/valeur

def fonction(,long, larg):*

Pour n'accepter que des paramètres par position:

def fonction(p,r,s,/,) : ➔ *(caractère '/' Python >3.8)*

Le premier paramètre obligatoirement par position

def fonction(p,/,m,g): etc..

Utilisation: permet de faire évoluer les noms de variables sans perte de compatibilité

Les annotations

L’objectif est de documenter la fonction (appel et résultat) SANS IMPACT sur son fonctionnement

```
Entrée [56]: def volume(longueur:int, largeur:int, hauteur:int) -> "un volume de format int" :
                 print('longeur:', longueur )
                 print('largeur:', largeur )
                 print('hauteur:', hauteur )

                 return longueur*largeur*hauteur
```

```
Entrée [57]: print(help(volume))
             print(volume.__annotations__)
             volume(2,3,4)
```

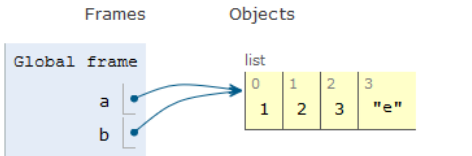
```
Help on function volume in module __main__:

volume(longueur: int, largeur: int, hauteur: int) -> 'un volume de format int'

None
{'longueur': <class 'int'>, 'largeur': <class 'int'>, 'hauteur': <class 'int'>, 'return': 'un volume de format
int'}
longeur: 2
largeur: 3
hauteur: 4
Out[57]: 24
```

Syntaxe param : < expression>
→ <expression> :

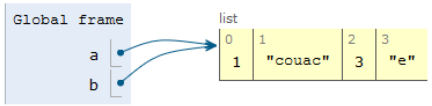
Les pièges du passage de paramètre

```
1  a = [1, 2, 3, 'e']
2  b = a
3  print(id(a),id(b))
4  print(a)
5  print(b)
```

```
140681079950920 140681079950920
[1, 2, 3, 'e']
[1, 2, 3, 'e']
```




```
a[1] = 'couac'
```



Se rendre sur :http://www.pythontutor.com

Préférer des structures non muables

```
def fonction(conteneur, b):
    conteneur+= b
    return len(conteneur)
```

Avec une liste

```
a = ['e','r', 'i']
aj = [3,3]
print(fonction(a, aj))
```

```
5
```

```
print(a)
```

```
['e', 'r', 'i', 3, 3]
```

Avec un tuple

```
a = ('e','r', 'i')
aj = (4,5)
print(fonction(a, aj))
```

```
5
```

```
print(a)
```

```
('e', 'r', 'i')
```

A commenter

```python
def fonction2(b, liste =[]):
    liste.append(b)
    return liste
```

```python
a= 4
print(fonction2(a))
```

```
[4]
```

```python
print(fonction2(a, [1,2]))
```

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

```python
c= 5
print(fonction2(c))
```

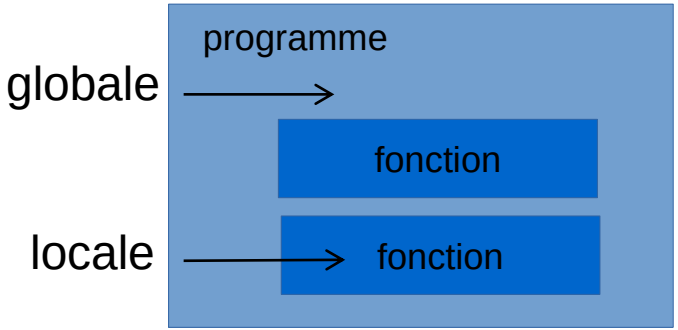
```
[4, 5]
```

Solution ?

Le scope

- Une variable est dans le scope (portée) du bloc qui porte sa déclaration (initialisation).
  - Une variable initialisée dans un programme est globale
  - Une variable initialisée dans une fonction/méthode est locale à ce bloc

- **MAIS** Python introduit ‘une intelligence’ dans la détermination du scope (portée d’une variable)
  …qu’il faut parfois tempérer...

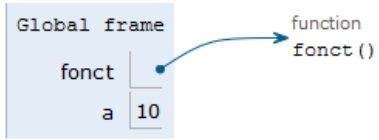


Exemples de scope

```
def fonct():
    print(a)

a = 10
fonct()
```

Print output

```
10
```




```
def fonct():
    a = 5
    print('var a dans fonct',a)

a= 10
print('var a en début',a)
fonct()
print(a)
print('var a en fin',a)
```

```
var a en début 10
var a dans fonct 5
10
var a en fin 10
```

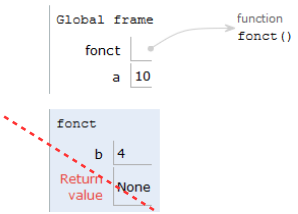
Variable locale : détruite en sortie de fonction

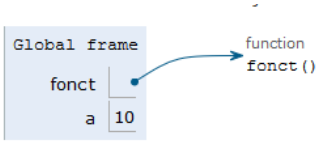
```
def fonct():
    b = 4
    print(a)
    print(b)

a = 10
fonct()
print(b)
```

Print output

```
10
4
```

NameError: name 'b' is not defined





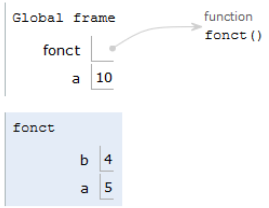
Quand tout se passe bien

Python détecte que ‘a’ est une variable locale et la supprime en sortant de la fonction

```
def fonct():
    b = 4
    a = 5
    print(a)
    print(b)

a = 10

fonct()

print(a)
```

Print output

```
5
4
10
```



Pour aller plus loin

```
def fonct():
    print(a)
    a = 20

a = 10
fonct()
print(a)
```

UnboundLocalError: local variable 'a' referenced before assignment

Python considère à tort que ‘a’ est une variable locale
parce qu’elle est affectée dans la fonction
(même après l’instruction qui l’utilise)

```
def fonct():
    global a
    print(a)
    a = 20


a = 10
fonct()
print(a)
```

Print output

```
10
20
```

Le mot clé **global** détrompe Python

Exercices

exo_1.ipynb

exo_2.ipynb

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

IGPDE

Les exceptions

Séquence 3

RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

IGPDE

Sommaire

- ❑ Utilisation de base
- ❑ Utilisation avancée
- ❑ Créer ses exceptions

Utilisation de base

L’extrême souplesse de Python peut aboutir rapidement à des erreurs difficiles à debugger…

- Une exception se définit simplement comme une erreur détectée lors de l'exécution d'un code (quel qu'il soit)

Exemple :

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Utilisation de base

●On entend donc par « gestion d'exception » la mise en œuvre d'un dispositif de détection d'erreur (*prévisible ou inattendue*) permettant de mener une action suite à sa levée

Une approche simple par l'exemple : utilisation du bloc **try :** et de la clause **except :**

→ soit une fonction de calcul de rapport entre 2 floats. La fonction enverra une erreur lorsque le dénominateur sera égal à 0.

```
def rapport(x, y) :
    try:
        print(x/y)    # Unique action de la fonction rapport dans un bloc TRY
    except :          # interception d'une éventuelle erreur
        print("Erreur détectée")
        return None
```

Résultat après exécution :

>>> rapport(10, 2) → Résultat : 5.0

>>> rapport(10, **0**) → **Erreur détectée**

Principe et anatomie du bloc

un bloc **try :**

code principal à exécuter

une clause **except** suivi de **:**

code à exécuter en cas d'interception d'erreur (s)

Utilisation de base

- Utilisation d'un argument d'exception

except *argument* :

•Nouvelle approche simple par l'exemple :

→ soit une fonction de calcul de rapport entre 2 floats. La fonction enverra une erreur lorsque le dénominateur sera égal à 0.

```
def rapport(x, y) :
    """
        Calcul le rapport entre 2 floats. Une exception sera levée en cas de division par 0
    """
    try:
        print("Resultat : ", x/y)  # Unique action de la fonction rapport dans un bloc TRY
    except ZeroDivisionError :     # interception de l'erreur ZeroDivisionError si le dénomiteur est = 0
        print("Division par zéro")
        return None
```

Résultat après exécution :

>>> rapport(10, 2) → Résultat : 5.0

>>> rapport(10, **0**) → **Division par zéro**

Anatomie du bloc

un bloc **try :**
code principal à exécuter
une clause **except** suivi de ***l'argument d'exception*** à tester et de **:**
code à exécuter en cas d'erreur (interception)

Plusieurs motifs

- Il est possible de gérer plusieurs clauses except avec arguments dans une instruction try :

•Complément sur l'exemple précédent :

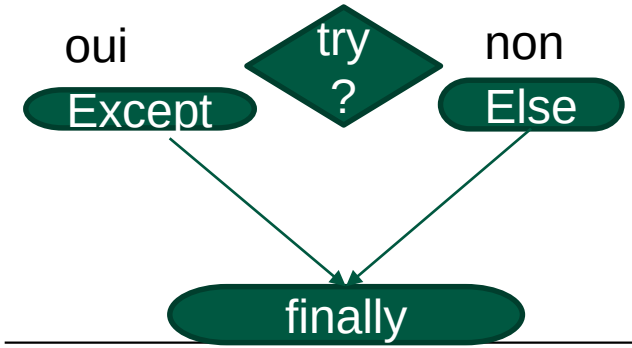
→ soit une fonction de calcul de rapport entre 2 floats.
La fonction enverra une erreur lorsque le dénominateur sera égal à 0 ou lorsqu'un des deux paramètres ne sera pas un nombre.

```
def rapport(x, y) :
    """
        Calcul le rapport entre 2 floats.
        Une exception sera levée en cas de division par 0 ou si l'un des deux paramètres n'est pas un nombre.
    """
    try:
        print("Resultat : ", x/y)  # Unique action de la fonction rapport dans un bloc TRY
    except ZeroDivisionError :     # interception de l'erreur ZeroDivisionError si le dénomiteur est = 0
        print("Division par zéro")
        return None
    except TypeError : # interception de l'erreur TypeError si une valeur est d'un type incoorect pour l'opération
        print("Le type entré ne correspond pas")
        return None
```

Résultat après exécution :
>>> rapport(10, 2) → Résultat : 5.0
>>> rapport(10, **0**) → **Division par zéro**
>>> rapport(10, **'a'**) → **Le type entré ne correspond pas**

Synthèse

Try:
Except Error1
Except Error2
Finally:
bloc de rebranchement toujours exécuté



OU

Try:
Except Error1 **as err**
Except Error2
Else:
curiosité de Python
Exécuté si aucune exception
Finally:
bloc de rebranchement toujours exécuté

as err permet d'instancier un objet exception

Créer ses exceptions

L’instruction raise <exception> permet de lever une exception.
Exemple: raise Exception(‘ca ne passe pas’)

On peut utiliser les exceptions standards (NameError, TypeError etc..)
Ou surcharger la classe Exception avec ses propres motifs.

Liste des exceptions standards:
https://www.tutorialsteacher.com/python/error-types-in-python

Exercice

exo_exception_enonce.ipynb

a) Créer un fichier « fictest.txt » dans lequel vous saisirez une ligne de texte de votre choix.

b) Modifier le code du notebook afin qu'une exception soit levée (interceptée) lorsque le fichier passé en paramètre n'existe pas.

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

IGPDE

Les modules internes

Séquence 4

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

IGPDE

Sommaire

- ❑ Module random
- ❑ Modules OS et SYS
- ❑ Module collections

Import

- La liste standard comprend plus de 200 modules
- Voir la documentation officielle :

https://docs.python.org/3/py-modindex.html

Un module interne ou externe s'utilise de la même façon :

import <module>

from <module> import <elements> as <alias>

help('<nom module>') pour connaître le détail

Module random

Manip en groupe

1. Trouver l’aide du module
2. A quoi sert-il ?
3. Comment l’utilise-t-on ? (import etc.)
4. Construire une fonction simulant un lanceur de dé

Le module OS

- os
  - Objets et méthodes en relation avec l'operating system.
  - Des méthodes pour explorer les systèmes de fichier :

Comme os.scandir()

```
import os

dirList = os.scandir("/root") # current directory
for dir in dirList:
  if dir.is_dir():
    print(dir.name)
  else:
    print('fichier',dir.name)
```

```
fichier .bashrc
fichier .profile
.ipython
.conda
.local
```

Méthode walk

os.walk(‘répertoire’) : exploration récursive d’un répertoire.

```
import os
for racine, dirs, fichiers in os.walk("/opt/conda/share/jupyter/lab/themes"):
    print(dirs)
    for nom in dirs:
        print("repertoire:", nom )
    for nom in fichiers:
        print("fichier complet", f"{racine}/{nom}")
```

```
['@jupyterlab']
repertoire: @jupyterlab
['theme-light-extension', 'theme-dark-extension']
repertoire: theme-light-extension
repertoire: theme-dark-extension
[]
fichier complet /opt/conda/share/jupyter/lab/themes/@jupyterlab/theme-light-extension/af7ae50
5a9eed503f8b8e6982036873e.woff2
fichier complet /opt/conda/share/jupyter/lab/themes/@jupyterlab/theme-light-extension/packag
e.json
```

Autre manière : module glob

```
import glob
for name in glob.glob('/root/.*'):
    print(name)
```

```
/root/.bashrc
/root/.profile
/root/.ipython
/root/.conda
/root/.local
```

Module sys

- On connaît déjà la liste des arguments sys.argv
- Sys embarque d'autres composants :
  - sys.path : fourni les répertoires de recherche des modules
  - sys.stdout, sys.stdin et sys.stderr

```
import sys
print(sys.path)
fe = sys.stdout
fe.write('essai\n')
for line in sys.stdin:
    fe.write(line)
```

```
root@e10f632c3716:/mytmp# cat essai.py |python essai.py
['/mytmp', '/usr/local/lib/python37.zip', '/usr/local/lib/python3.7', '/usr/local/lib/python3.7/lib-dynload', '/usr/local/lib/python3.7/site-packag
es']
essai
import sys
print(sys.path)
fe = sys.stdout
fe.write('essai\n')
for line in sys.stdin:
    fe.write(line)
```

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

Le module collections

`namedtuple()`	fonction permettant de créer des sous-classes de `tuple` avec des champs nommés
`deque`	conteneur se comportant comme une liste avec des ajouts et retraits rapides à chaque extrémité
`ChainMap`	classe semblable aux dictionnaires qui crée une unique vue à partir de plusieurs dictionnaires
`Counter`	sous-classe de `dict` pour compter des objets hachables
`OrderedDict`	sous-classe de `dict` qui garde en mémoire l'ordre dans lequel les entrées ont été ajoutées
`defaultdict`	sous-classe de `dict` qui appelle une fonction de fabrication en cas de valeur manquante
`UserDict`	surcouche autour des objets dictionnaires pour faciliter l'héritage de `dict`
`UserList`	surcouche autour des objets listes pour faciliter l'héritage de `list`
`UserString`	surcouche autour des objets chaînes de caractères pour faciliter l'héritage de `str`

Les tuples nommés

Objectif : Attribuer un nom à chacun des éléments d'un tuple (champ)
et permettre d'accéder aux éléments à partir de ce nom comme on pourrait le faire avec un attribut d'un objet.

- Création d'un tuple nommé en deux phases :
  - Définition d'un nouveau type de tuple nommé
  - Création du tuple nommé

namedtuple.py

```
1  from collections import namedtuple
2
3  produit = namedtuple('produit', ['code', 'marque','modele','prix'])
4  harley = produit(5549000000995, 'Harley Davidson', 'Road King Classic', 26000.00)
5  print(harley, type(harley))
```

Shell

```
>>> %Run namedtuple.py
  produit(code=5549000000995, marque='Harley Davidson', modele='Road King Classic', prix=26000.0) <class '__main__.produit'>
>>>
```

Utilisation

Comme dans un tuple normal, on peut accéder à un champ par son numéro **MAIS aussi** par son nom de colonne

```
harley= produit(5549000000995, 'Harley Davidson', 'Road King Classic', 26000.00)
m2 =produit(55505568, 'Honda', 'Goldwing', 25999.00 )
```

```
print(harley.marque)
print(m2.marque)
print(m2[1])
```

```
Harley Davidson
Honda
Honda
```

Ce sont des classe légères : objet sans méthode avec des attributs non modifiables
Ils sont utiles pour manipuler des fichiers structurés de volume important en lecture seule (ex: référentiel)

Les compteurs automatiques: counter

La classe Counter est une sous-classe de dict qui permet le dénombrement d'objets hachables.

Permet de compter automatiquement des éléments.

```
1  from collections import Counter as Counter
2  a = Counter('pourquoi python est bien')
3  print(a)
```

```
Counter({'o': 3, ' ': 3, 'p': 2, 'u': 2, 'i': 2, 't': 2, 'n': 2, 'e': 2, 'r': 1, 'q': 1, 'y': 1, 'h': 1, 's': 1, 'b': 1})
```

Ajouter des éléments

```
4  a.update('encore')
5  print(a)
```

```
Counter({'o': 3, ' ': 3, 'p': 2, 'u': 2, 'i': 2, 't': 2, 'n': 2, 'e': 2, 'r': 1, 'q': 1, 'y': 1, 'h': 1, 's': 1, 'b': 1})
Counter({'o': 4, 'e': 4, ' ': 3, 'n': 3, 'p': 2, 'u': 2, 'r': 2, 'i': 2, 't': 2, 'q': 1, 'y': 1, 'h': 1, 's': 1, 'b': 1, 'c': 1})
```

Defaultdic

Soit un dictionnaire de liste: cle => ['x','y' ..]

Entrée [2]:

```
b = dict()
if 2 in b:
    b[2].append('two')
else:
    b[2] = ['two']
print(b)
print(b[0])
```

```
{2: ['two']}

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-2-dd76f082d011> in <module>
      5     b[2] = ['two']
      6 print(b)
----> 7 print(b[0])

KeyError: 0
```

Comment éviter le test ?
Et l'erreur de clé

Sans defaultdict

Avec defaultdict

manip_dict.ipynb

```
Entrée [1]: from collections import defaultdict
            c = defaultdict(list)
            c[2].append('two')
            print(c)
            print(c[0])
```

```
defaultdict(<class 'list'>, {2: ['two']})
[]
```

La classe dict propose la méthode setdefault(cle , valeur) mais son usage est très limité par rapport à defaultdict.

Exercice

exo_lipsum_corr.ipynb

Lire les fichiers lipsum1 à 3

Quelles sont les 3 lettres les plus utilisées dans chacun des textes ?

python decompte.py lipsum1.txt
[('i', 213), ('e', 211), ('u', 183)]

python decompte.py lipsum2.txt
[('i', 268), ('e', 267), ('u', 239)]

python decompte.py lipsum3.txt
[('e', 304), ('i', 296), ('a', 243)]

python decompte.py lipsum4.txt
fichier inconnu

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

IGPDE

Les expressions régulières (regex)

Module RE

Séquence 5

RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

IGPDE

Sommaire

- ❑ Points d attention
- ❑ Les méthodes de base
- ❑ Exemples de motif

Avant d’utiliser le module RE

Des méthodes de la classe str (String) remplacent avantageusement les fonctions de REGEX
Exemples:

- Index : 'abecdef'.index('c') #3
- Replace: 'abecdef'.replace('e', 'x') #'abxcdxf'
- Count: 'abecdef'.count('e') #2
- Split: 'abecdef'.split('e') #['ab', 'cd', 'f']
- Endswith: 'abecdef.avi'.endswith('.avi') #True
- Startswith: '2019ecdef.txt'.startswith('19',2) #True

Suite

- Décomposer une URL : urllib.parse

  Ce module propose des méthodes pour isoler le protocole (http), le nom du serveur , la ressource, les paramètres etc.

- Décomposer une chemin de fichier : os.path()

  Méthodes split() ou splitext()

Le module RE: méthodes de base

- Import re
- Fonctionne comme en Perl
- 4 méthodes principales
  - Search() : recherche du motif dans la chaine
  - Match() : recherche du motif **au début** de la chaine
  - Findall() : recherche toutes les occurrences du motif
  - Sub() : recherche et substitution

Et un iterateur: finditer

Bon à savoir : il est parfois nécessaire de protéger les caractères spéciaux ('\')
dans les motifs ou les chaines de caractère. Le prefixe '**r**' avant une chaine rempli
ce rôle en créant une raw-string : exemple **r**'ma chaine\n' au lieu de 'ma chaine\\n'

Findall et Search, match

Findall

```
import re
recherche_chat = re.findall('chat\w+', "le chaton est sur le toit du chateau",)
if recherche_chat:
    print("j ai trouve le chat", len(recherche_chat), 'fois')
    print(recherche_chat)
```

```
j ai trouve le chat 2 fois
['chaton', 'chateau']
```

liste

Search, match

```
recherche_http = re.match('http://(.+)/', 'http://exemple.com/page1')
if recherche_http:
    print('site',recherche_http[1])
    print(recherche_http.group(1))
    print('position_deb,position_fin', recherche_http.span(1))
    print(recherche_http[0][7:18])
    print(recherche_http.start(1))
```

```
site exemple.com
exemple.com
position_deb,position_fin (7, 18)
exemple.com
7
```

Objet Match

Objet match et méthode group()

- Accès par méthodes ou par attribut
  - Méthode principale : group() : Utilisée dans le cadre de capture avec parenthèse.

```
]: import re
   chaine = "python est puissant"
   mots= re.search('(\w+) (\w+) (\w+)' , chaine)
   if mots:
       print("chaine de départ:", chaine)
       print("l'index 0 est toujours la chaine complete:", mots[0])
       print(mots.group())
       print("par la methode group():", mots.group(2))
       print("par un attribut de  l'objet match:", mots[2])
       print(mots.groups())
       print(mots.start(2),mots.end(2))
       print(mots.span(2))
```

Retourne 'None' sinon

```
chaine de départ: python est puissant
l'index 0 est toujours la chaine complete: python est puissant
python est puissant
par la methode group(): est
par un attribut de  l'objet match: est
('python', 'est', 'puissant')
7 10
(7, 10)
```

Les modificateurs

manip_regexp.ipynb

- re.IGNORECASE ou re.I
- re.MULTILINE ou re.M

```
import re
chaine = "Pourquoi\npourqui"
recherche_apos = re.findall('^pourqu', chaine, re.I)
print(recherche_apos)

# combiner 2 modificateurs
print("avec multiligne")
recherche_apos = re.findall('^pourqu', chaine, re.I|re.M)

print(recherche_apos)
print('Attention les REGEXP sont puissantes')
print("sans multiligne")
recherche_apos = re.findall('pourqu', chaine, re.I)

print(recherche_apos)
```

Ajoute un '^' et '$' à chaque ligne

```
import re
chaine = 'Pourquoi\npourqui'
print(chaine)
```

```
Pourquoi
pourqui
```

```
print(re.findall('^pour', chaine))
```

```
[]
```

Avec modificateur

```
print(re.findall('^pour', chaine,re.I|re.M))
```

```
['Pour', 'pour']
```

```
print(re.findall('^pour', chaine,re.M))
```

```
['pour']
```

Exemples de motif

- Recherche de mot : \w+
- Recherche de bord de mot : \b *(exemple page suivante)*
- Recherche de nombre : \d \d\d \d{2}
- Recherche de mail (.+)@(.+)
- Glouton ou non glouton : .+?
- Recherche dans un ensemble [a-z] ou l'inverse de l'ensemble [^a -z]
- Motif : '.' , '*', ' ?'

Divers : ‘raw’ string - glouton

Bordure de mot \b

```
: import re
chaine = "python est puissant"
mots= re.search(r'\b(.+) \b(.+) \b(.+)' , chaine)
if mots:
    print(mots.groups())
```

```
('python', 'est', 'puissant')
```

A noter : l’utilisation d’une raw string

```
ch1 = r'eric\n'
ch2 = 'eric\n'
print(len(ch1))
print(len(ch2))
```

```
6
5
```

```
: chaine = '123\b4'
print(chaine)
```

```
124
```

```
chaine ='python est puissant'
print(re.findall(r'([^ ]+)', chaine))
```

```
['python', 'est', 'puissant']
```

Mode glouton/non glouton

```
: import re
chaine = "12:25:2019 suite"
chaine2 ="2:25:2019 suite"
heure= re.match("(\d\d?):", chaine)
if heure:
    print(heure[1])
heure= re.match("(\d.*):", chaine)
if heure:
    print('mode glouton:', heure[1])
heure= re.match("(\d.*?):", chaine)
if heure:
    print(heure[1])
heure= re.match("(\d\d?):", chaine2)
if heure:
    print(heure[1])
heure= re.match("(\d.*):", chaine2)
if heure:
    print("mode glouton:",heure[1])
heure= re.match("(\d.*?):", chaine2)
if heure:
    print(heure[1])
```

```
12
mode glouton: 12:25
12
2
mode glouton: 2:25
2
```

Exercice

Exo-3.ipynb

manip_regexp_exo-3

- Soit une chaine d'un sous-titre :

  « 2'5'': Mon toit est à toi, Toinon »

  Compter le nombre de 'toi' (3 caractères)
- Même question avec 'Toi' ou 'toi'
- Compter le nombre de mot se terminant par 'n'
- Isoler les minutes et les secondes

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

IGPDE

Python avancé

Séquence 6

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

IGPDE

Sommaire

- ❑ Listes ‘compréhension’
- ❑ Les tris

Compréhension de liste

- Augmenter sa productivité avec le temps en optimisant son code.
- Une idée simple : simplifier le code pour le rendre plus lisible et donc plus rapide à écrire et plus simple à maintenir.
- Réduire les boucles FOR en 1 seule ligne (en intégrant si besoin un filtrage conditionnel)
- Dénommées « listcomps » dans le jargon Pythonnien

Exemples

- **Avec une boucle For … in**

```
var = '12345678910'
liste_car = [ ]
for car in var :
    liste.append(car)
```

- **Avec une Listecomp**
- Sans condition

```
new_liste = [item for item in liste]   → liste_car = [car for car in var]
ou
new_liste = [function(item) for item in liste]
```

- Avec condition :

```
new_liste = [item for item in liste if condition(item)] →
ou
new_liste = [function(item) for item in liste if condition(item)]
```

Selection, filtre

- Exemple :

Objectif : Filtrer les valeurs de liste_a et conserver dans liste_b celles qui sont supérieures à 4.

Écriture standard avec une boucle for :

```
>>> liste_a = [1,4,2,7,1,9,0,3,4,6,6,6,8,3]
>>> liste_b = []
>>> liste_a = [1,4,2,7,1,9,0,3,4,6,6,6,8,3]

    liste_b = []

    for x in liste_a:
        if x > 4:
            liste_b.append(x)
    print(liste_b)
  [7, 9, 6, 6, 6, 8]
```

Écriture avec compréhensions de listes :

```
>>> liste_a = [1,4,2,7,1,9,0,3,4,6,6,6,8,3]
>>> liste_b = [x for x in liste_a if x > 4]
>>> print(liste_b)
  [7, 9, 6, 6, 6, 8]
>>>
```

Exemple

- Utilisation d'une fonction sur chaque item d'une liste :

  Objectif : Convertir une liste contenant plusieurs items texte (String) en entier (Integer)

```
>>> items = ["5","10","15","20","25","30","35","40"]
>>> items = [int(x) for x in items]
>>> print(items)
  [5, 10, 15, 20, 25, 30, 35, 40]
>>>
```

Exercice

Exo_listcomp

1 - Réaliser le même script en 3 lignes seulement afin d'afficher le même résultat

list_comprehension.py

```
liste_a = [1,4,2,6,7,1,9,0,3,4,6,6,6,8,3,6]
liste_b = []
for x in liste_a:
    if x == 6:
        liste_b.append(x)
print("Longueur de liste_b : {}".format(len(liste_b)))
```

Shell

```
>>> %Run list_comprehension.py

  Longueur de liste_b : 5

>>>
```

2 - En une ligne convertir en texte les valeurs de liste_b
(puis vérifier la bonne conversion avec un print de liste_b)

3 – A partir de liste_a, créer en une ligne une nouvelle liste nommée « liste_c » qui contiendra seulement le carré des valeurs comprises entre 2 et 6 :

[16, 4, 36, 9, 16, 36, 36, 36, 9, 36]

Correction

```
# Correction exercice : compréhensions de listes

# 1 - Réaliser le même script en 3 lignes seulement afin d'afficher le même résultat

liste_a = [1,4,2,6,7,1,9,0,3,4,6,6,6,8,3,6]
liste_b = [x for x in liste_a if x == 6]
print("Longueur de liste_b : {}".format(len(liste_b)))

# 2 - En une ligne convertir en texte les valeurs de liste_b

liste_b = [str(x) for x in liste_b]
# (puis vérifier la bonne conversion avec un print de liste_b)

print(liste_b)

# 3 - A partir de liste_a, créer en une ligne une nouvelle liste nommée « liste_c »
# qui contiendra les valeurs suivantes : [16, 4, 36, 9, 16, 36, 36, 36, 9, 36]

liste_c = [x**2  for x in liste_a if x >= 2 and x < 7]
(ou liste_c = [x*x  for x in liste_a if x >= 2 and x < 7])
print(liste_c)
```

Fonctionne pour d'autres structures

- L'utilisation du principe de compréhension s'applique est également aux dictionnaires et aux ensembles:
  - dictionnaires (dict)

Exemple : création d'un dictionnaire à partir d'une liste des codes et noms des départements

```
code_departement = [("001","AIN"),("002","AISNE"),("003","ALLIER"),("004","ALPES HTE PROVENCE"),("005","HAUTES ALPES"),("00
      ("007","ARDECHE"),("008","ARDENNES"),("009","ARIEGE"),("010","AUBE"),("011","AUDE"),("012","AVEYRON"),("013"," BOUCHE
      ("015","CANTAL"),("016","CHARENTE"),("017","CHARENTE MARITIME"),("018","CHER"),("019","CORREZE"),("02A","CORSE DU SUD
dept_code = {code : departement for code, departement in code_departement}
print(dept_code, type(dept_code))
```

```
{'001': 'AIN', '002': 'AISNE', '003': 'ALLIER', '004': 'ALPES HTE PROVENCE', '005': 'HAUTES ALPES', '006': 'ALPES MARITIMES
', '007': 'ARDECHE', '008': 'ARDENNES', '009': 'ARIEGE', '010': 'AUBE', '011': 'AUDE', '012': 'AVEYRON', '013': ' BOUCHES D
U RHONE', '014': 'CALVADOS', '015': 'CANTAL', '016': 'CHARENTE', '017': 'CHARENTE MARITIME', '018': 'CHER', '019': 'CORREZE
', '02A': 'CORSE DU SUD', '02B': 'HAUTE CORSE'} <class 'dict'>
```

  - ensembles (set)

Exemple : création d'un ensemble représentant des carrés des entiers comprise entre -2 et 5

```
>>> set(k*k for k in range(-2,6))
{0, 1, 4, 9, 16, 25}
```

Les tris

Sujet central en programmation.

Le cas le plus simple: les tris de liste

- Utilisation de la fonction sorted (pas de modification de la liste - ducktyping)

- Utilisation la méthode ‘sort’ de la classe List (modification de la liste et retour de la valeur None)

Cas simple des listes homogènes

```
a = [ 2,  3, 1, 8, 7, 4, 0]
b = sorted(a)
print('sorted a  et b')
print(a,b)

c =[ 2,  3, 1, 8, 7, 4, 0]
print('methode sort c avant ')
print(c)
c.sort()
print('methode sort c après')
print(c)
print(c.sort())
```

```
sorted a  et b
[2, 3, 1, 8, 7, 4, 0] [0, 1, 2, 3, 4, 7, 8]
methode sort c avant
[2, 3, 1, 8, 7, 4, 0]
methode sort c après
[0, 1, 2, 3, 4, 7, 8]
None
```

conséquence

```
: b = sorted(a)[0:5]
  print(b)

  [0, 1, 2, 3, 4]

: print(c.sort()[0:5])

  ---------------------------------------------------------------------------
  TypeError                                 Traceback
  <ipython-input-20-b3eefe99eed3> in <module>
  ----> 1 print(c.sort()[0:5])

  TypeError: 'NoneType' object is not subscriptable
```

Cas des listes hétérogènes

```
a = [ 2, 3, 'b', 1, 'c', 'a' ,8, 7, 4, 0]
a.sort()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-22-bbe05cf733ca> in <module>
      1 a = [ 2, 3, 'b', 1, 'c', 'a' ,8, 7, 4, 0]
----> 2 a.sort()

TypeError: '<' not supported between instances of 'str' and 'int'
```

Python ne sait pas comparer un integer et une chaine

Solution : transformer à la volée un integer en string

On applique la fonction str()
Sur chaque élément

```
a = [ 2, 3, 'b', 1, 'c', 'a' ,8, 7, 4, 0]
a.sort(key=str)
print(a)
```

```
[0, 1, 2, 3, 4, 7, 8, 'a', 'b', 'c']
```

Le mot clé : key

```
a = [ 2, 3, 'b', 1, 'c', 'a' ,8, 7, 4, 0]
a.sort(key=lambda x: str(x))
print(a)
```

```
[0, 1, 2, 3, 4, 7, 8, 'a', 'b', 'c']
```

Fonction anonyme

```
def monfiltre(item):
    return(str(item))
a = [ 2, 3, 'b', 1, 'c', 'a' ,8, 7, 4, 0]
a.sort(key=monfiltre)
print(a)
```

```
[0, 1, 2, 3, 4, 7, 8, 'a', 'b', 'c']
```

Fonction

Tri sur des éléments de listes ou tuples

```
a = [('b',3) , ('b', 1) , ('a', 5) , ('c',1)]
```

```
sorted(a)
```

```
[('a', 5), ('b', 1), ('b', 3), ('c', 1)]
```

```
sorted(a, key= lambda x:(x[0], x[1]))
```

```
[('a', 5), ('b', 1), ('b', 3), ('c', 1)]
```

```
sorted(a,key=lambda x: x[1] )
```

```
[('b', 1), ('c', 1), ('b', 3), ('a', 5)]
```

```
sorted(a, key= lambda x : (x[1],x[0]))
```

```
[('b', 1), ('c', 1), ('b', 3), ('a', 5)]
```

```
sorted(a, key= lambda x : (x[1],x[0]), reverse = 1)
```

```
[('a', 5), ('b', 3), ('c', 1), ('b', 1)]
```

```
sorted(a, key= lambda x : (x[0],x[1]), reverse = 1)
```

```
[('c', 1), ('b', 3), ('b', 1), ('a', 5)]
```

Le tri Python est dit ‘stable’

Tri sur la 2eme occurrence du tuple

Idem avec le module operator

```
from operator import itemgetter
sorted(a, key = itemgetter(1))
```

```
[('b', 1), ('c', 1), ('b', 3), ('a', 5)]
```

Tri d’objet

```
class Joueur:
    def __init__(self, nom, score, age):
        self.nom = nom
        self.score = score
        self.age = age
    def __str__(self):
        return(f"{self.nom}, {self.age}, {self.score}")
    def __repl__(self):
        return(f"{self.nom}, {self.age}, {self.score}")
j1 =Joueur('GEGE', 120, 50)
j2 = Joueur('MARTY', 110, 40)
j3 = Joueur('DODO', 105, 45)
a = list((j1, j2, j3))
print("non trié")
for obj in a:
    print(obj)
```

```
non trié
GEGE, 50, 120
MARTY, 40, 110
DODO, 45, 105
```

Echec des méthodes standards

```
print("trié")
#a.sort()      #ne fonctionne pas
b= sorted(a)   #ne fonctionne pas
```

```
trié

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-63-c399afaf89b8> in <module>
     17 print("trié")
     18 #a.sort()     #ne fonctionne pas
---> 19 b= sorted(a)  #ne fonctionne pas

TypeError: '<' not supported between instances of 'Joueur' and 'Joueur'
```

Utilisation d'une fonction

```
for obj in sorted(a,key=lambda x: x.score):
    print(obj)
print("ou descendant")
for obj in sorted(a,key=lambda x: x.score, reverse =True):
    print(obj)
```

attribut

```
DODO, 45, 105
MARTY, 40, 110
GEGE, 50, 120
ou descendant
GEGE, 50, 120
MARTY, 40, 110
DODO, 45, 105
```

OU

```
from operator import attrgetter
for obj in sorted(a,key=attrgetter('age')):
    print(obj)
```

```
MARTY, 40, 110
DODO, 45, 105
GEGE, 50, 120
```

Avec une méthode de l’objet

Il est possible de définir un critère de tri natif.

Dans la classe (Joueur) ajouter une méthode pour ‘less than’

```
def __lt__(self, other):
    if self.age < other.age:
        return True
    else:
        return False
setattr(Joueur,'__lt__', __lt__ )
```

Mais cela n’est plus recommandé car trop rigide

Tri multicritères

```
a = list((j1, j2, j3))
for obj in sorted(a,key=attrgetter('score', 'age')):
    print(obj)
```

```
DODO, 45, 105
MARTY, 40, 120
GEGE, 50, 120
```

2 Joueurs avec le même score

Trier un dictionnaire

Tri sur la clé

La fonction sorted appliquée sur un dictionnaire retourne une liste de clé triée.

```
b = {'e' : 2, 'a': 1 , 'c': 3 , 'd' : 4, 'b' : 5 }
c = sorted(b)
print( c)
```

```
['a', 'b', 'c', 'd', 'e']
```

Tri sur les valeurs

```
b = {'e' : 2, 'a': 1 , 'c': 3 , 'd' : 4, 'b' : 5 }
d = sorted(b.items(), key= lambda x: x[1] )
print( d)
```

```
[('a', 1), ('e', 2), ('c', 3), ('d', 4), ('b', 5)]
```

Dict.items() retourne une liste de tuple (clé, valeur)

Un dictionnaire restitue les données dans l'ordre d'insertion.

Exercice

Corriger le code:

Exo-tri

```
tu = ([1,2], (5,6), {2:3})
sorted(tu)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-32-51538caab5e1> in <module>
      1 tu = ([1,2], (5,6), {2:3})
----> 2 sorted(tu)

TypeError: '<' not supported between instances of 'tuple' and 'list'
```

Correction

```
sorted(tu, key=lambda x: list(x)[0])
```

```
[[1, 2], {2: 3}, (5, 6)]
```

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

IGPDE

Les classes

Séquence 7

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

IGPDE

Sommaire

- ❑ Les méthodes internes
- ❑ Les décorateurs
- ❑ Les classes et leurs décorateurs

Méthodes internes

- Méthodes préfixées par 2 ‘_’
- La compréhension de leur comportement est essentiel à la maîtrise de Python
- Souvent à l’origine de message d’erreur

```
: 'essai'()
```

```
---------------------------------------------
TypeError
<ipython-input-1-5223455f306c> in <module>
----> 1 'essai'()

TypeError: 'str' object is not callable
```

Utilisation

Ces méthodes permettent de modifier ou sur surcharger le comportement standard d'un objet.

Les plus courantes:
__init__
__str__
__repr__

Elles sont exploitées dans les sections suivantes.

Les décorateurs

- Ils servent à encapsuler une fonction/méthode
- Ils s'utilisent avec leur nom préfixé de '@'

exemple :

```
@deco
def ma_fonction():
```

Ce sont des fonctions qui appellent une fonction en ajoutant une valeur ajoutée

Utilisations

manip_decorateur.ipynb

- Encapsuler des attributs (lecture seule) et les protéger.
- Poser des timers pour mesurer les temps d'exécution
- Sécuriser des accès
- Formater du texte (vers du html)
- On n'est pas limité à un seul niveau d'imbrication

Classes et objets

Rappels sur le modèle de classe de Python:

- Attributs de classe
- Méthodes de classe (@classmethod)
- Attributs d'un objet
- Méthodes d'un objet

Un objet est une structure de la forme d'un dictionnaire: l'accès à un élément se fait par un accesseur (clé) . On fait suivre l'instruction par '()' pour exécuter une méthode ou sans '()' pour manipuler un attribut.

L'accès se fait par *monObjet.accesseur*

Pas de protection forte des attributs ou des méthodes privées.

Exemple

```
class Demo1():
    def __init__(self, nom):
        self.nom = nom
```

```
personne1 = Demo1('Martin')
```

```
print(vars(personne1))
```

```
{'nom': 'Martin'}
```

Ajout d'un attribut à la volée

```
personne1.prenom = 'eric'
print(vars(personne1))
print(dir(personne1))
```

```
{'nom': 'Martin', 'prenom': 'eric'}
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex_
_', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'nom', 'prenom']
```

Attributs privés

```
class Demo2():
    def __init__(self, name, lastname):
        self.nom = name
        self.prenom = lastname
        self.__id = self.nom + '.' + lastname
    def retourne(self):
        return (self.__id)
```

```
personne2 = Demo2('german', 'eric' )
```

```
print(personne2.__id)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-7-e502043003d9> in <module>
----> 1 print(personne2.__id)

AttributeError: 'Demo2' object has no attribute '__id'
```

```
print(vars(personne2))
```

```
{'nom': 'german', 'prenom': 'eric', '_Demo2__id': 'german.eric'}
```

```
print(personne2._Demo2__id)
```

```
german.eric
```

Attention aux erreurs

```
personne2.__id = 'essai__'
print(vars(personne2))
```

```
{'nom': 'german', 'prenom': 'eric', '_Demo2__id': 'german.eric', '__id': 'essai__'}
```

```
print(personne2.__id)
print(personne2.retourne())
```

```
essai
german.eric
```

Comment trouver le cheminement de recherche d'un attribut ?

Déclaration

```
class Demo3():
    def __init__(self, name, lastname):
        print('dans init')
        self.nom = name
        self.prenom = lastname
        self.id   = self.nom + '.' + lastname
        self.mail = self.id + '@demo.fr'

    def retourne(self):
        print('dans retourne')
        return (self.id)

    def __setattr__(self,attr, val):
        print(f'set sur {attr}')
        super().__setattr__(attr, val)

    def __getattr__(self,attr):
        print(f'get sur {attr}')
        return super().__getattribute__(attr)

    def __getattribute__(self, nom):
        print('GET general sur ', nom)
        return super().__getattribute__(nom)
```

Exécution

```
personne3 = Demo3('Mercet', 'martin')
```

```
dans init
set sur nom
set sur prenom
GET general sur  nom
set sur id
GET general sur  id
set sur mail
```

```
print(personne3.retourne())
```

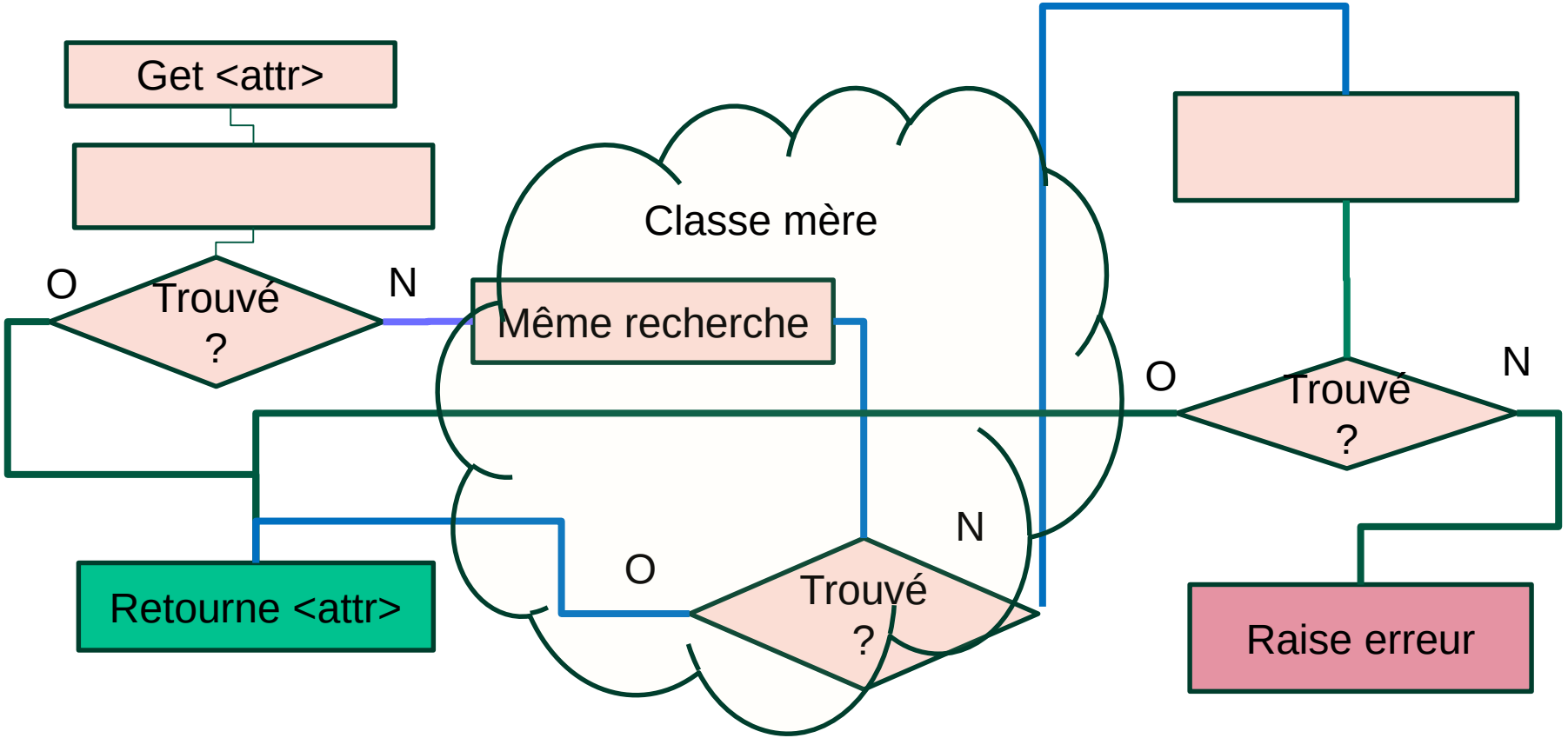
```
GET general sur  retourne
dans retourne
GET general sur  id
Mercet.martin
```

```
print(personne3.adresse)
```

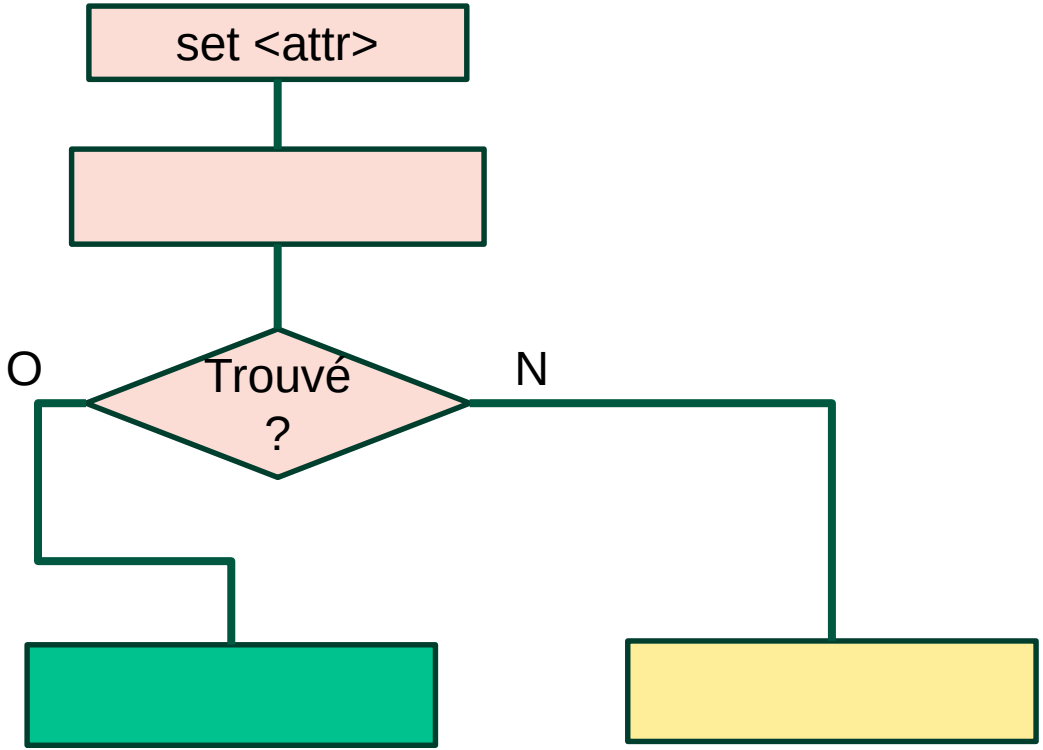
```
GET general sur  adresse
get sur adresse

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent
<ipython-input-12-0fe89408a5b1> in <module>
----> 1 print(personne3.adresse)

<ipython-input-9-91fe0ae820de> in __getattr__(self, attr)
     17     def __getattr__(self,attr):
     18         print(f'get sur {attr}')
---> 19         return super().__getattribute__(attr)
     20
     21     def __getattribute__(self, nom):

AttributeError: 'Demo3' object has no attribute 'adresse'
```

Tracer les flèches D'exécution

Complétez


Get <attr>
Trouvé ?
O
N
Retourne <attr>
Classe mère
Même recherche
Trouvé ?
O
N
Trouvé ?
O
N
Raise erreur

Expliquez les différences pour __setattr__



Le décorateur @classmethod

- Méthodes utilisables sans instanciation d'un objet

```
class Repertoire:

    compteur = 0

    @classmethod
    def ajuste_compteur(cls, nb):
        print(cls)
        cls.compteur = nb
```

Décorateur

Nom de la classe (convention)

```
print(Repertoire.compteur)
Repertoire.ajuste_compteur(23)
print(Repertoire.compteur)
```

Variable de classe

Méthode de classe

```
2
<class '__main__.Repertoire'>
23
```

Le décorateur @staticmethod

- Appelées méthodes statiques
  - Utilisables sans instanciation

```
class Repertoire:

    compteur = 0

    @staticmethod
    def ajuste_compteur(nb):
        Repertoire.compteur = nb
```

décorateur

```
Repertoire.ajuste_compteur(2)
```

Evocation

@staticmethod ne passe pas le nom de la classe à la méthode.
Il sert à regrouper des fonctions disparates au sein d'une classe (package)

Utilisation ?

Exemple utilisation de methode de classe

```
class Dataconteneur():
    def __init__(self, data):
        self.nom = data[0]
        self.prenom = data[1]
    def __str__(self):
        return str((self.nom, self.prenom))

    @classmethod
    def  from_tuple(cls,tuple):
        data = tuple
        objet =cls(data)
        return objet

    @classmethod
    def from_csv(cls, file):
        ## lecture fichier dans data
        ligne = data.split(';')
        data = (ligne[0], ligne[1])
        objet =cls(data)
        return objet
    @classmethod
    def from_xml(cls, file):
        pass
```

```
obj = Dataconteneur.from_tuple(('german', 'eric'))
print(obj, type(obj))
```

```
('german', 'eric') <class '__main__.Dataconteneur'>
```

- Appel à un constructeur avec des paramètres différents
- Parser: *une fonction qui utilise deux classes*.

Décorateurs @property

Il permet de transformer une méthode en attribut

Objectif : Simplifier l'interface de la classe

-Utile pour les attributs calculés ou déduits et privés

Exemple : calcul de la somme des objets d'une classe, l'adresse mail construite à partir du nom et prenom

Exemple

```
class Demo4():
    def __init__(self, name, lastname):
        print('dans init')
        self.nom = name
        self.prenom = lastname
        self.id   = self.nom + '.' + lastname
        self.__mail = self.id + '@demo.fr'


    @property
    def mail(self):
        return self.__mail
    @mail.setter
    def mail(self,_):
        raise Exception("pas de modif sur l 'attribut mail")
```

Fin de la 1ere partie