PYTHON 3

SOMMAIRE

Edition : juillet 2020 – janvier 2022

Ce poly est une reprise du FUN MOOC : Python 3 : des fondamentaux aux concepts avancés du langage – INRIA – UCA - Arnaud Legout, Thierry Parmentelat, Marie-Hélène Comte. Qu'ils et elle en soient remercié-e-s.
https://www.fun-mooc.fr/courses/course-v1:UCA+107001+session02/courseware/a7116af790134f2fb7244260fa53f695/e571aec07f434b418812579414da2e67/

PYTHON 3

0 – Introduction

0.1 - Principes du langage

- Langage pragmatique !
  - Tout est objet mais utilisation procédurale de base.
  - Syntaxe propre et intuitive

Les types de programmation

Type de programmation	Langage de référence	Mot clé - Structure	Autres langages
Impérative (=procédurale)	C	main	**Python**, JavaScript, etc.
Objet	Java	classe	C++, **Python**, etc.
Fonctionnelle	Lisp	lambda expression	Java, JavaScript, **Python**, etc.
Evénementielle	JavaScript	event, callback	Java, **Python**, C#, JavaScript
Web-Serveur	PHP	echo, le serveur web	Java, C#, **Python**, JavaScript

0.2 - Références

Site Python.org

- https://www.python.org

Documentation Python : à regarder !!!

- https://www.python.org/doc/
- Accueil de la doc : https://docs.python.org/fr/3/
- Tutoriel : https://docs.python.org/fr/3.10/tutorial/index.html : très bien !
- Bibliothèque standard : https://docs.python.org/fr/3.10/library/index.html
- Guide de style : https://peps.python.org/pep-0008/ : l'indentation, c'est 4 espaces, point barre !
    - https://python.sdv.univ-paris-diderot.fr/15_bonnes_pratiques/
- Etc.

Langue

- Une partie de la documentation est en français, pas forcément tout.
- Le code et la documentation du code sont en anglais
- Rappel : **Google Chrome** gère une traduction des pages de façon pratique.

Autres ressources

Il y en a plein, plein, plein !

- **W3S** : https://www.w3schools.com/python/
- **OCR** : https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python
- **FUN-MOOC** : https://www.fun-mooc.fr/courses/course-v1:UCA+107001+session02/courseware/a7116af790134f2fb7244260fa53f695/e571aec07f434b418812579414da2e67/

0.3 - Bref historique

Les différentes versions

- Python 1.0 : janvier 1994 !!!
- Python 2.0 : octobre 2000
- Python 2.7 jusqu'à décembre 2016. Maintenu jusqu'en 2020.
- Python 3 à partir de décembre 2008.
- La cohabitation des v2 et v3 versions a été une source de problèmes. Il y a eu une rupture de compatibilité entre la v2 et la v3. Douloureux mais nécessaire ! Beaucoup à cause de l'encodage des chaines de caractères.
- Aujourd'hui, Python 3 uniquement : le Python c'est le Python 3, comme le HTML c'est le HTML 5 et le CSS le CSS 3.

Pourquoi Python ?

- Créateur du langage : **Guido Van Rossum**, fan de la série télévisé « **Monty Python**'s Flying Circus » (69-74). Films notables des Monty Python : Sacré Graal-75, LaVie de Brian-79. Film notable de Terry Gilliam : Brazil-85, L'Armée de 12 singes-95, L'Homme qui tua Don Quichotte-2018.
- Le spam chez les Monty Python : https://www.youtube.com/watch?v=zLih-WQwBSc. C'est ce l'humour anglais… Il y a des **spam**, des **eggs**, du **ham** et du **bacon**. (Littéralement SPAM se traduit en anglais : Shoulder of Pork and Ham, autrement dit épaule et jambon de porc).

Évolutions

- **Guido Van Rossum**, dictateur bénévole à vie », BDFL, jusqu'à la 572è proposition d'amélioration du langage validée, en 2018.
- Les évolutions sont discutées collectivement et c'était le BDFL qui tranchait !

0.4 - Installation de base

Latest version

- 09/2022 : Python 3.10.7 (version stable).
- https://www.python.org/downloads/

Interpréteur de base

- L'installation installe un interpréteur de base qui est accessible en ligne de commandes dans un terminal.
- C'est un « **REPL** » Read Evaluate Print Loop. Autrement dit un programme **CLI** (Command Line Interface) qui lit les commandes qu'on saisit, les évalue, affiche le résultat et recommence (loop).
- C'est une « calculette ».

Distributions

- En plus de l'implémentation traditionnelle (**CPython)** on trouve de nombreuses distributions (implémentations alternatives).
- A noter : **Anaconda** Python, orientée gestion de données (du Python avec des modules dédiés à la gestion de données).
- https://www.python.org/download/alternatives/

Vocabulaire

- **« Un terminal » :** c'est un **REPL** (Read Evaluate Print Loop), un programme **CLI** qui lit les commandes qu'on saisit, les évalue, affiche le résultat et recommence) qui comprend les commandes du système d'exploitation. On dit aussi :
  - Interpréteur (de commande)
  - Terminal
  - Console
  - Shell
- **« L'invite de commande » :** c'est le symbole (les symboles) qui précéde(nt) l'endroit où on saisit dans le REPL. On dit aussi : « **prompt** ». Selon le REPL (l'interpréteur), le **prompt** change :
  - Windows : C :>
  - Mac : blabla %
  - Python : >>>
  - MySQL : mysql>
  - node : >

Sur MAC

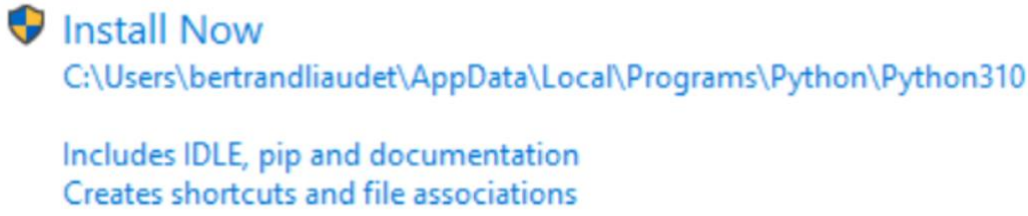
- Par défaut : python 2 installé.
  - La commande python --version affiche la version.
  - La commande python dans un terminal ouvre une console python 2.
  - La commande exit() permet de sortir de la console

- On installe la « latest feature release » de python : https://www.python.org/downloads/
  - La commande python3 --version affiche la version.
  - La commande python3 ouvre une console python 3.
  - Les commande >>>import sys puis >>>print(sys.version) affichent la version
  - La commande >>>exit() permet de sortir de la console.

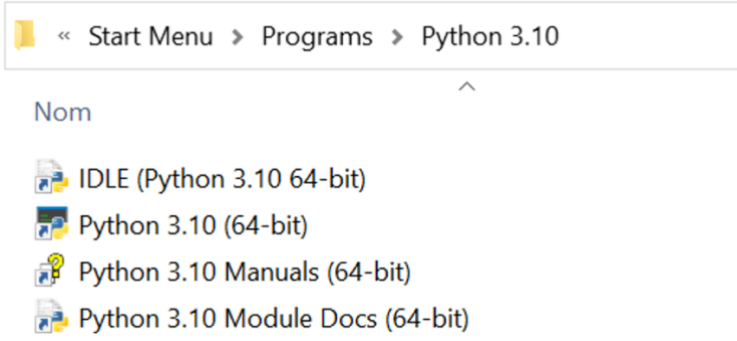
- L'installation permet d'accéder à l'IDE « IDLE » qui se trouve dans Applications/Python 3.8 :
  - La commande idle ouvre une fenêtre correspondant à l'IDE.

Sur Windows

- Sur Windows 10 : si on tape « python » dans un terminal et que Python n'est pas installer, il propose une fenêtre d'installation avec la version 3.9
- On peut aussi installer la « latest feature release » de python : https://www.python.org/downloads/



- Installation par défaut :



- On peut customiser (ou pas) l'installation :
  - Install for all users
  - Add python to environnement variables
  - Choix du dossier d'installation
- En console, la commande python --version affiche la version
- La commande python dans un terminal ouvre une console python 3.8.3
  - La commande exit() permet de sortir de la console

0.5 - IDE

PyCharm

- PyCharm est un IDE très utilisé dans l'écosystème Python.
- Il est très bien, mais on ne l'utilisera pas.
- https://www.jetbrains.com/fr-fr/pycharm/

IDLE

- L'installation de Python installe un IDE de base : IDLE
- IDLE est un interpréteur de base avec des fonctionnalités pratiques : indentation automatique, débuggeur, …
- On peut utiliser n'importe quel autre IDE, plus pratique : Visual Studio Code, PyCharm, …
- Utilisation de IDLE :
  - Création d'un fichier de code : File / New file. On peut créer le fichier ou ouvrir un fichier existant. Ca ouvre une fenêtre d'édition. Quand on « run », ça revient sur l'interpréteur IDLE en le réinitialisant.
  - Pour rappeler le code, en haut.
  - Réinitialisation de l'interprétateur python : F5

Ipython : un peu compliqué à installer

➢ ***C'est quoi***

- Ipython est une version améliorée du CLI et de l'IDLE.

➢ ***Sur MAC***

- Pour créer un environnement de travail : home % python3 -m venv learn-ipython. Ca crée un répertoire learn-ipython avec du contenu.
- Pour activier l'environnement de travail : home % source learn-ipython/bin/activate
- De là : (learn-ipython) home % python → on démarre la version 3.8.3 (plus besoin de python3, car on a créé l'environnement avec python3).
- (learn-ipython) home % pip install --upgrade pip
- (learn-ipython) home % pip install ipython
- (learn-ipython) home % ipython

➢ ***Sur PC***

- Pour créer un environnement de travail : C :>python -m venv learn-ipython
- Pour activier l'environnement de travail : C :>learn-ipython\Scripts\activate.bat
- (learn-ipython) C :> pip install --upgrade pip (il faut peut-être le faire avant d'être dans l'environnement. Pour desactiver : deactivate.bat).
- (learn-ipython) C :> pip install ipython
- (learn-ipython) C :>ipython

➢ ***Usages dans ipython pour améliorer la productivité***

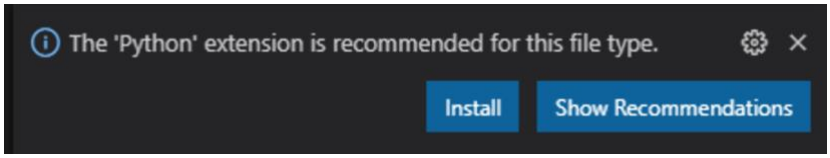
https://www.youtube.com/watch?v=ASyAQa9XLpw

Visual Studio Code

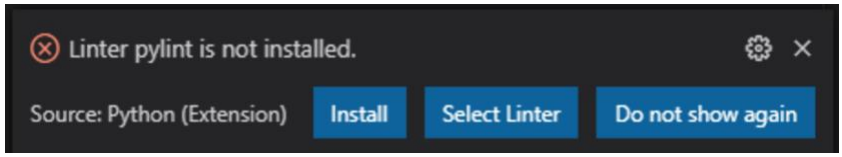
- https://code.visualstudio.com/

➢ *Première utilisation*

- On se crée un dossier de travail et on crée un fichier test.py : il faut installer une extension pour python :



- On va aussi installer PyLint, outil utilisé pour les erreurs dans le code Python. Ça ouvre automatiquement le terminal du VSC.



- Attention aux warnings : il y a peut-être des variables d'environnements à mettre à jour.

➢ *Terminal*

- On peut ouvrir un terminal pour exécuter un programme en ligne de commande. Le terminal s'ouvre dans le dossier du dossier d'ouverture de Visual Studio Code.

➢ *Exécuter sans débogage*

- On peut exécuter le code par le menu : exécuter sans débogage

➢ *Exécuter avec débogage*

- On peut déboguer le code par le menu : exécuter avec débogage.
- On peut mettre des points d'arrêt dans le code.
- La console de débogage permet d'afficher le contenu des variables et aussi de les changer.
- On peut exécuter le code ligne par ligne.

0.6 - pip : package installer for Python

- pip est l'installeur de package python : il est installé par défaut avec l'installation de python.
- Les packages sont sur https://pypi.org/
- pip est un package accessible ici : https://pypi.org/project/pip/
- En ligne de commande on utilise pip (windows) ou pip3 (mac et windows)
- Par exemple :

```
pip install numpy pandas scikit-learn matplotlib seaborn
```

  - Ca permet d'installer numpy, pandas et scikit-learn qui sont des bibliothèques de traitement de données pour la data-science et le machine learning
  - et matplolib et seaborn qui sont des bibliothèques pour afficher des données en mode graphique.

0.7 - Généralités sur le langage

Syntaxe de base

- Des types
- Des structures de contrôle
- Des fonctions « built in »
- La modularité et les espaces de nommage
- Compréhension de listes
- La programmation objet
- Programmation avancée : métaclasses (classe qui instancie des… classes), décorateurs, etc.

Fonctionnement du langage

- Lisible
- Tout est objet
- Liaison lexicale et typage dynamique
- Itération
- Gestion automatique de la mémoire

Objectifs

- Écrire du code propre et commentée (# et ‘’’)
- Lire du code
- Bien utiliser les librairies : on peut tout faire avec Python.

Écriture compacte lisible

- Langage pragmatique : procédural ou objet au choix

```
# hello world procedural classique
print('Hello World')
```

- Langage très lisible : orienté présentation : indentation obligatoire

```
# parcourir une liste
eleves=['toto', 'tata', 'titi']
for eleve in eleves: # pour chaque eleve de eleves
   print(eleve)
```

- Possibilité de compacter le code tout en restant lisible : compréhension de liste

```
maListe=[1,2,3]

# version avec compréhension de liste
def squares(liste):
   return [x*x for x in liste]

print(maListe)
print(squares(maListe))

# version Classique
def squares2(liste):
   listeOut=[]
   for x in liste:
      listeOut.append(x*x)
   return listeOut

print(squares2(maListe))
```

- Compréhension de liste et lamba expression :

```
# lambda expression : la variable square sera comme une fonction
square = lambda x: x**2

maListe=[1,2,3]

# compréhension de liste et lambda expression
def squares(liste):
   return [square(x) for x in liste]

print(maListe)
print(squares(maListe))
```

- Autre exemple compacte : opérateur conditionnel

```
# version compacte
def fibo(n):
   return 1 if n <=1 else fibo(n-1)+fibo(n-2)

# version C ou Java :  return n <=1 ? 1 : fibo(n-1)+fibo(n-2)

fibo(10)

# version Classique
def fibo2(n):
   if n <=1:
      return 1
   else :
      return fibo2(n-1)+fibo2(n-2)

fibo2(10)
```

Les fonction built-in = fonctions natives

- Ce sont les fonctions présentent directement dans le langage, sans avoir à les importer à partir d'une librairie :
- https://docs.python.org/3/library/functions.html
- https://www.w3schools.com/python/python_ref_functions.asp

Librairie standard (modules)

- Maintenue comme le langage.
- Pour accéder aux fonctions d'un module, par exemple : >>> import math
- Pour utiliser une fonction d'un module importé, par exemple : >>> math.sqrt(x)
- Pour importer une seule fonction et l'utiliser directement :

```
from math import sqrt
sqrt(9)
```

- https://docs.python.org/3/library/numeric.html
- https://www.w3schools.com/python/python_math.asp

Librairies non standards

- **Calcul scientifique :** scipy, sympy, …
- **Traitement de données :** numpy, pandas, scikit-learn, …
- **Affichage de données :** matplotlib, seaburn, …
- **Web** : django, flask, …
- **Tout existe !** Pour parler à sa porte de garage ou autre chose.
- Les bibliothèques peuvent intégrer du code C et une interface python. Ainsi, c'est toujours très rapide (par exemple numpy).

Les notebook : éditeur commun de la communauté datascience

notebooks jupyter, google colaboratory (sur google drive), …

Langage auto-documenté

- Une fois un module importé : >>> import math
- On peut lister les fonctions du module, par exemple : >>> dir(math)
- Pour avoir de l'aide sur une fonction d'un module importé, par exemple : >>> help(math.ceil)

docstring

- On peut créer la documentation de nos propres modules et fonctions pour qu'elles soient accessibles via l'interpréteur python avec les « triples apostrophes » : docstring.

```
def testDocstring(a, b):
    '''return a list [a, b].'''
    return [a, b]

help(testDocstring)
```

Portabilité

- Mac, Windows, Linux, Raspberry Pi, etc.

Licence

- Python Software Foundation
- On peut faire ce qu'on veut de python, y compris à des fins commerciales.

Nouveautés et avantages de python 3 :

- Unicode
- f string : à partir de la 3.6, les f-string : pratique ! Mettez à jour vos versions !
- Écriture en compréhension
- Etc.

Conclusion : programmation rapide !

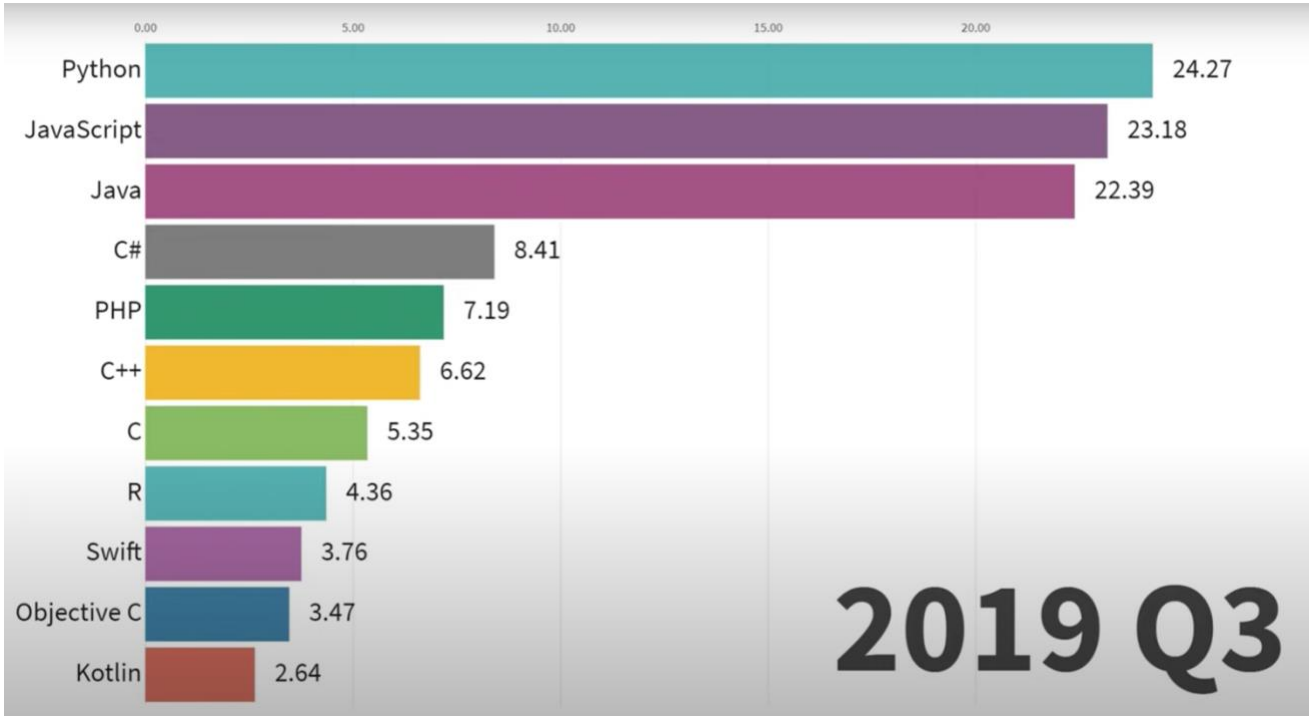
- Qualités du Python :
  - Types de base très puissants.
  - Code souple et puissant.
  - Graphisme
  - Web
  - Programmation système
  - Analyse de données facilitée et rapide.
  - Documentation facilitée avec notebooks Jupyter (texte, image, code et résultats).

⇒ Langage qui permet de développer des prototypes et de faire des tests très facilement.
⇒ Langage qui permet de développer des applications complètes.

Python sur le marché des langages

https://www.youtube.com/watch?v=Og847HVwRSI

- L'ère du Fortran : 1965-1979 – la programmation procédurale
- L'ère du Pascal : 1980-1984 – la programmation procédurale
- L'ère du C : 1985-2000 – la programmation procédurale
- L'ère du Java : 2001-2017 – la programmation objet
- Triarchie : Java, JavaScript, Python : depuis 2018 – la programmation mixte
- L'ère du Python : depuis 2019 – la programmation mixte



1 - Type et typage dynamique. Espace des variables et espace des objets (10 min)

Tout est objet

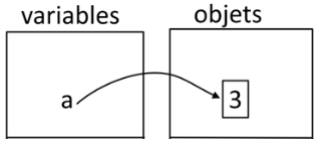
- **Toute information manipulée par python est un objet**.
- Une information peut être un **nombre**, une **chaine de caractère**, une **liste d'informations**, etc : elle peut contenir plusieurs informations et avoir plusieurs formes différentes.
- **Un objet contient des informations** (ou données) mais aussi des mécanismes qui permettent de les manipuler. Ces mécanismes sont appelés « méthodes ».
- **Un objet a un type**. Le type caractérise le format de l'information portée par l'objet (un nombre, une chaine de caractères, une liste, etc.). Le type caractérise aussi les méthodes qui pourront s'appliquer à l'objet.

L'espace des objets

- Les objets sont créés dans la mémoire de l'ordinateur, dans un espace particulier appelé « **espace des objets** ».
- Quand on écrit : >>> 'toto' dans l'interpréteur, on crée un objet dans l'espace des objets qui contient comme information : 'toto'. Cet objet est **créé dynamiquement** et **automatiquement typé** comme une chaine de caractère. Il **peut donc accéder à toutes les méthodes associées au type** chaine de caractères, par exemple : upper() pour mettre en majuscule. Pour cela on écrit : >>> 'toto'.upper(). Dans ce cas, dans l'espace des objets, on a toujours 'toto', mais on affiche 'TOTO'.

L'espace des variables = espace de nommage

- Pour **nommer les objets**, on dit aussi **les référencer**, on associe un **objet** à une **variable** avec une affectation. Par exemple : >>> note=1. Ce faisant, on crée l'objet 1 dans l'espace des objets, la variable note dans l'**espace des variables** aussi appelé **espace de nommage**. Et on fait le lien entre les deux. Il faut bien nommer ses variables : ça participe à la documentation automatique du code.



- Si on fait >>> a=3 , python crée l'objet 3, la variable a et le lien entre les 2. Si on fait ensuite >>> a='toto' , python crée l'objet 'toto', casse le lien entre a et l'objet 3 (on dit qu'il déréférence l'objet), puis fait un lien entre a et 'toto'. La variable sera désormais de type chaine de caractères.
- L'objet 3 n'est plus référencé : il sera supprimé automatiquement par le **garbage collector** qui est un mécanisme de l'interpréteur.
- Le type n'est pas lié à la variable. Le **type est lié à l'objet** : Python est un langage à **typage fort** : le type d'un objet ne peut pas changer.
- La variable peut changer d'objet en cours d'exécution. C'est à éviter en général pour plus de lisibilité.
- On peut supprimer la variable a en faisant : >>> del a . Ca supprime la variable de l'espace des variables. Si l'objet référencé par a n'est plus référencé par aucune autre variable, alors il sera supprimé automatiquement par le garbage collector.

2 – Bases syntaxiques - 1

2.0 - Les variables et les mots-clés

Les variables

- Caractères possibles dans le nom : AB..Y, ab..yz, 01..89, _
- Le seul caractère spécial utilisable est le « _ ». On ne peut pas utiliser « - ».
- On peut aussi mettre des caractères accentués, des caractères d'autres langues : β est un nom de variable possible : à éviter !
- On ne peut pas utiliser les mots-clés du langage. True ne peut pas être une variable, mais true peut l'être !
- Un nom de variable ne peut pas commencer par un chiffre. Il peut commencer par un _
- Le nom d'une variable doit être le plus explicite possible. Le python utilise le « snake_case » en plus du « camelCase ». Le snake case pour les variables et le constantes du programme age_du_capitaine, MA_CONSTANTE. Le came case pour les classe : class Capitaine, dateNaissance, ageDuCapitaine() (https://python.sdv.univ-paris-diderot.fr/15_bonnes_pratiques/)

Les 35 mots-clés du langage

- **None** **False** **True** not and or in is
- if else elif
- pass for while break continue
- import from
- class def return global **nonlocal**
- try except as finally raise
- del with
- lambda yield
- assert
- **await** **async**

https://www.programiz.com/python-programming/keyword-list#is

2.1 - Les 4 types numériques : int, float, complex, bool

- Les entiers : int. >>>n=5 Les entiers ont une précision illimité.
- Les flottants : float. >>>x=4.3 Les float ont une précision limité. >>>0.3-0.1
- Les nombres complexes : complex. >>>c=(4.3+5j)
- Les booléens : bool. >>>flag=True

  ⇒ 0 et 0+0j valent False
  ⇒ Toutes les autre nombres, -5, 5, 0.5, 5.5, 1/3 , -1/3, 4+0j, etc. valent True

type()

```
i=1
type(i)             # <class 'int'>
type(5.3)           # <class 'float'>
int(4.3)            # 4
float(4)            # 4.0
complex(4) # 4+0j
5e2                 # 500 ⇔ 5*10**2
5e-2                # 0,05 ⇔ 5*10**-2
```

usage

- 0 vaut False, 1 vaut True (que 0 ou 1 soient entier, float ou complexe).
- / : division réelle // : division entière % : modulo ** : puissance

```
10/3                 # 3.3333333333333335
10//3                # 3
10%3                 # 1
10**3                # 1000
0.0==False           # True
1==True              # True
```

isinstance(value, type)

```
isinstance(23, int)             # True
x=4.0
isinstance (x, float)           # True
isinstance (False, bool)        # True
```

L'interpréteur comme calculette

```
(2 * 30) + (10 * 5)   # 110

(2 + 3j) * 2.5j       # complexe : (-7.5+5j)

pi = 3.14159          # le . pour la virgule
2 * pi * 10

from math import e, pi, sqrt
e, pi                 # (2.718281828459045, 3.141592653589793)

sqrt(9)               # 3.0
int(sqrt(9))          # 3 – int() pour changer de type

x= 23_456_789         # 23456789

x=1                   # 1
x=x+1                 # 2
x+=1                  # 3 : idem avec *  /  %  ** etc.
x**=2                 # 9

10**-12               # 1e-12
```

Calcul flottant

Pour préciser le calcul avec des float, on peut utiliser les modules Decimal ou Fraction

```
# le problème :
print(0.3-0.1)
# 0.19999999999999998
0.3-0.1==0.2
# False


from decimal import Decimal
print(Decimal(0.3)- Decimal(0.1))
# 0.19999999999999999833466546306
Decimal(0.3)- Decimal(0.1) == Decimal(0.2)
# False
# Conclusion : plus de précision, mais ça ne change rien.



from fractions import Fraction
print(Fraction(3,10)-Fraction(1,10))
# 1/5
Fraction(3,10)-Fraction(1,10) == Fraction(2, 10)
# True



# limites du calcul flottant
# https://docs.python.org/fr/3/library/sys.html
import sys
print(sys.float_info)

sys.float_info.max=1.7976931348623157e+308
sys.float_info.min=2.2250738585072014e-308
sys.float_info.dig=15                           # nb décimales max
sys.float_info.epsilon=2.220446049250313e-16 # écart minimum
```

Binaire, octale, hexadécimal, toute base et décalage de bits

```
deux_cents = 0b11001000      # binaire
print(deux_cents)            # 200
bin(200)                     # 0b11001000

deux_cents = 0o310           # octal
print(deux_cents)
oct(200)                     # 0o310

deux_cents = 0xc8            # hexa
print(deux_cents)
hex(200)                     # 0xc8

deux_cents = int('3020', 4)  # base 4
print(deux_cents)

10>>1     #5 : 10 c'est 0b1010 (binaire)
          #  >>1 c'est un décalage des bit de 1 à droite
          #  0b1010>>1 c'est 0b101 soit 5
10<<1     #20 : Ob1010 passe à  Ob10100 soit 20
10<<2     #40 : Ob1010 passe à Ob101000 soit 40
x=10
x<<=2     # x vaut 40

10 & 20   #0 : c'est : 0b1010 & Ob10100
          # & c'est 'et' : on fait un et bit à bit
          # ici, le résultat c'est Ob00000
10 & 23   #2 : c'est : 0b1010 & Ob10111 = Ob00010
10 | 20   #30: c'est : 0b1010 | Ob10100 = Ob11110
```

2.2 - Les chaines de caractères

Codage et décodage

- Le problème : coder et décoder les caractères qui entrent dans une chaine de caractères.
- **1ère génération de codage :** ASCII (7 bits), ASCII étendu (8 bits=1 octet=1 byte), ISO (8 bit)
- **2ème génération de codage :** projet Unicode : UTF-8, UTF-16, UTF-32 (plusieurs octets). UTF-8 est compatible avec ASCII.
- ASCII et UTF-8 sont des jeux de caractères. La police est un affichage particulier pour un jeu de caractères.
- **Python est compatible avec Unicode. Il faut toujours utiliser UTF-8, sauf contre-indications explicite.**
- Les fonctions ord() et chr() permettent de passer d'un caractère à son numéro de code et réciproquement.

```
ord('a')     # retourne 97
ord('β')     # retourne 946
chr(65)      # retourne 'A'
chr(945)     # retourne 'α'
```

type str

- Type chaine de caractère : str
- Apostrophe ou guillemets. Python met des apostrophes par défaut, des guillemets si
- Les chaines de caractères sont **immuables** : une fois définie, on ne peut pas les changer : l'objet ne peut pas changer ses informations.

```
a='bacon'              # a vaut 'bacon'
b=a                    # b vaut 'bacon'
b="c'est un 'spam'"    # b vaut "c'est un 'spam'", a vaut 'bacon'
```

- >>>dir(str) : toutes les méthodes. Les méthodes __ : méthodes spéciales qui s'utilisent dans certain contexte.
- >>>help(str.title) : pour avoir la doc de la fonction title (str ? sous ipython - :q pour sortir)

Manipulation de base

- string[3] : 4ème caractère de la chaine (on commence à 0) >>> 'bonjour'[3] # 'j'
- len(str) :longeur de la chaine >>> len('coucou') # 6
- + : concaténation >>> 'a'+'b' # 'ab'
- [:] : sous chaine >>> 'abcde'[1:3] # 'bc'

Autres manipulations pratiques

- find(str) ou index(str) : position de str dans la chaine.
  - Usage : >>> 'abcde'.find('c') # 2
  - find retourne -1 si absent - index retourne une erreur
  - index servira pour toutes les séquences
  - rfind(str) : cherche à partir de la fin
- replace(mot1, mot2) : remplace un mot par un autre (mot1 par mot2)
  - Usage : >>> 'coucou'.replace('ou', 'her') # retourne 'chercher'

Chaine remplie et chaine vide

```
chaine = ''

if chaine :
   print('chaine remplie')
else :
   print('chaine vide') # c'est ce qui s'affiche avec chaine = ''
```

Plein de fonctions :

- upper(), lower() = casefold() : met en majuscules, en minuscules
- capitalize() : met la première lettre majuscule, le reste en minuscules.
- title() : met la 1ère lettre de chaque mot en majuscule, la suite en minuscules.
- swapcase() : inverse la casse (minuscule devient majuscules et réciproquement)

- startswith(str), endswith() : vérifie si la chaine commence, se termine, par str
- isupper(), islower(), isalpha(), isalnum, etc : vérifie le type de chaine

- strip() : supprime les espaces de début et de fin

- count(str) : retourne le nombre d'occurrences de str

- center(taille) : retourne centré dans la taille proposée.
- rjust(taille), ljust : retourne justifié à droite, à gauche, dans la taille proposée.

- Etc. Il faut toutes les regarder !

Manipulation en rapport avec les listes : détaillée avec les listes.

- split() : retourne une liste de mots.
  - Usage : >>> 'hello world'.split() # retourne ['hello', 'world']
  - Variante : rsplit() : à creuser !
- separateur.join(liste de mots) : retourne une chaine de mots séparés par le séparateur.
  - Usage : >>> '-'.join(['hello', 'world']) # retourne 'hello-world'

sys.argv[…])

- Pour récupérer les arguments d'un programme en ligne de commande

```
# fichier test.py

import sys
print(sys.argv[0], sys.argv[1°)

# dans un terminal :
# C:>python test.py 5
# test.py 5
```

String avec triples apostrophes (ou guillemets)

- Pour mettre un texte avec des passages à la ligne dans une string

```
texte='''voici du texte
qu'on peut passer
à la ligne'''

# texte vaut : 'voici du texte\nqu'on peut passer\nà la ligne'

# \n : c'est un passage à la ligne
```

Synthèse sur les string

```
s='bacon'
len(s)                  # longueur vaut 5
print(s + ' and eggs') # concatenation : résultat "bacon and
eggs"
print(s[1:4])           # slice : résultat "aco"
print(s[2])             # accès direct : vaut "c"
# Immuable = non modifiable => s[2] = "a" : Erreur
# Liste de caractères
for c in s: # c pour caractère ou char, s pour string
  print(c) # affiche chaque caractère

for i in range(len(s)): # i pour index dans la string s
  print(s[i]) # afffiche chaque caractère

print(a.replace('c', 'll')) # affiche 'ballon'
t="voici du texte qu'on peut spliter"
t2=t.split()  # ['voici', 'du', 'texte', "qu'on", 'peut',
'spliter']
s='-'.join(t2)# "voici-du-texte-qu'on-peut-spliter"
print(t2)
print(s)
```

Axiomatique des string :

- ***3 méthodes pour tout faire :***
  - longeur : len
  - concaténation : +
  - sous-chaine (substr) : slice : [:]

  ⇒ On peut se débrouiller avec les 3 méthodes pour tout faire !
  ⇒ Avec en plus ord('A') et chr(65) ou upper() et lower() pour les traitements de base des caractères.

- ***Il existe de nombreuses autres méthodes de string qui facilitent les calculs :***
  - split, join, replace,
  - find, replace,
  - strip,
  - upper, lower, capitalize, title (capitalize chaque mot)

2.3 – écrire : print(), lire : input(), commenter

écrire : print() : pour afficher du texte et/ou des valeurs à l'écran

- ***print() de base***

```
print("age:", age)  # age: 20 # un espace entre les 2 paramètres
```

- ***end : pour éviter que ça passe à la ligne***

```
print("toto a ", end='') # le end remplace le passage à la ligne
print("20 ans")          # toto a 20 ans
print("toto a", end=': ')
print("20 ans")          # toto a: 20 ans
```

- ***f-string de base : pour gérer un affichage formaté***

```
# avec une f-string :
print(f"age:{age}") # age:20 # il n'y a que ce que l'on écrit
                    # la variable est entre accolade
```

- ***f-string et end:***

```
print(f"age:{age}", end="")
```

- ***Avec plus de 2 paramètres :***

```
nom="toto"
age=20
print("nom:", nom, "age:", age)  # nom: toto age:
print(f"nom:{nom} age:{age}")    # nom:toto age:20
```

lire : input() : pour lire une valeur au clavier

- ***input() : fonction pour saisir quelque chose au clavier : retourne toujours une string***

```
age=input('Entrez votre age : ')

type(age)   # <class 'str'>

age=int(input('Entrez votre age : '))

type(age)   # <class 'int'>
```

- ***Connaitre le type à un autre***
  - Retourner le type d'une variable : type(var)
- ***Changer le type d'une variable***
  - Changer le type d'une variable : int(var), float(var), bool(var), str(var)
- ***Tester le type d'une variable : isinstance(variable, type)***
  - isinstance(age, int), isinstance(age, float), isinstance(age, bool), isinstance(age, str),
- ***Vérifier qu'une string est un entier :***
  - age.isnumeric()

    ⇒ ça ne marche que pour les entiers positifs. Pour le reste, il faut écrire ses propres fonctions.

commenter le code : # et ‘’’

- Commentaire de fin de ligne : #
- Commentaires de bloc : ‘’’ ‘’’

- C’est utilisé pour les commentaires de fonctions (docstring).
- Exemple avec une version améliorée de la documentation

```
def test(a, b):
    '''
        quid : retourne les 2 paramètres in dans une liste
        in : 2 variables a et b au choix
        out : rien
        return : a list [a, b]
        méthode de résolution: RAS
    '''
    return [a, b] # on return dans une liste
```

« f string » détaillé : format pratique depuis le python 3.5

- f-string : string formatée
- ça s'utilise pour formater un affichage de string :

  ⇒ f-string
  ⇒ https://he-arc.github.io/livre-python/fstrings/index.html

➢ ***Exemple basique***

```
age=20
nom='toto'
print(age*5)
print(nom+' a '+str(age)+' ans')

# depuis python 3.5 : les « f string »
print(f"{nom} a {age} ans")
```

➢ ***Exemple subtil :***

```
l=[i*5 for i in range(10)]
for i in range(len(l)):
  print(f"{'key':>10s}: {i:2d} {'-':^6} value: {l[i]:2d} {'-':^6}
tiers: {l[i]/3:6.2f}")

# il y a 11 morceaux dans la f-string (entre les " "):

#  1: {'key':>10s} le mot 'key' sur 10 caractères, callé à droite
#  2: ": " est affiché tel quel
#  3: {i:2d} la variable i est un entier sur 2 caractères
#  4: " " est affiché tel quel
#  5: {'-':^6} le mot '-' est centré sur 6 caractères.
#  6: " value: " est affiché tel quel.
#  7: # {l[i]:2d} la variable l[i] est un entier sur 2 caractères
#  8: " " est affiché tel quel
#  9: {'-':^6} le mot '-' est centré sur 6 caractères.
# 10: " tiers: " est affiché tel quel
# 11: {i/3:6.2f} le float i/3 est sur 6 caractères dont 2 après
la virgule

'''
       key:  0   -    value:  0   -    tiers:   0.00
       key:  1   -    value:  5   -    tiers:   1.67
       key:  2   -    value: 10   -    tiers:   3.33
       key:  3   -    value: 15   -    tiers:   5.00
       key:  4   -    value: 20   -    tiers:   6.67
       key:  5   -    value: 25   -    tiers:   8.33
       key:  6   -    value: 30   -    tiers:  10.00
       key:  7   -    value: 35   -    tiers:  11.67
       key:  8   -    value: 40   -    tiers:  13.33
       key:  9   -    value: 45   -    tiers:  15.00
'''
```

- ***# présentation formaté du code :***

```
l=[i*5 for i in range(10)]
for i in range(len(l)):
       print(
              f"{'key':>10s}: {i:2d} "
              f"{'-':^6} "
              f"value: {l[i]:2d} "
              f"{'-':^6} "
              f"tiers: {l[i]/3:6.2f}"
       )
```

format : ancienne technique (on peut oublier)

```
age=20
nom='toto'
print(age*5)
print(nom+' a '+str(age)+' ans')

print("{} a {} ans".format(nom, age))
```

2.4 - Test, bloc d'instruction, indentation

bases

- Un test (if) permet d'évaluer une expression booléenne qui sera vraie (True) ou (Fausse)
- if a==5 : # on teste si l'expression a==5 est vraie ou fause
- Plusieurs opérateurs permettent de construire des expressions booléennes :
  - Opérateurs de comparaison : ==, !=, >, >=, <, <=
  - Opérateurs d'appartenance : in, not in (a in liste)
  - Opérateur de négation : ! ou not

```
if note > 10:
    print('reçu')
    print('bravo !')
else:
    print('recalé')
```

- Derrière le : on trouve un bloc d'instructions qui est décalé d'un nombre de caractères fixes : de base, 4 caractères. Le bloc se termine avec la fin du décalage.
- Avec cette gestion des blocs, la convention de codage fait partie de la syntaxe. Il n'y a qu'une seule façon de coder, facile à lire et facile à écrire.
- Le : sert à rendre les choses plus lisibles (dans l'absolu, on pourrait s'en passer).
- Un problème : ça supporte mal le copier-coller : il faut bien vérifier que la convention de codage (qui est de la syntaxe) suit !
- Bons usages :
  - Eviter d'avoir trop d'imbrications de blocs de code
  - Eviter de dépasser des lignes de 79 caractères
  - Tout ce qui est entre (), [], {} supporte le passage à la ligne en respectant la syntaxe des blocs d'instructions.

Indentation, guide de style, convention de codage

- Avec le test, apparaît la première notion d'indentation et de style d'écriture.
- La PEP : Python Enhancement Proposals est un ensemble de Proposition d'Amélioration de Python.
- La PEP-008 définit les conventions de codage qui s'appliquent à toute la librairie standard, et particulièrement :
  - La question de l'indentation : doublez l'indentation dans une liste sur plusieurs lignes si on poursuit avec une indentation normale.
  - La question des espaces : espace après les virgules, espace autour des opérateurs mais pas des doubles opérateurs spéciaux (**, <<, etc.).
  - La question des commentaires.
- L'outil pep8 permet de vérifier le style du code.
- L'outil autopep8 modifie le code pour le rendre conforme.
- Ces outils se trouvent sur la plateforme de de modules Python Package Index (PyPI).
- Ils s'installent avec l'outil pip ou pip3 selon le système d'exploitation.

Une référence :

https://python.sdv.univ-paris-diderot.fr/15_bonnes_pratiques/

2.5 - Boucle de base : for i in range(5) et while(condition) :

Boucle for i in range(5)

- Une boucle permet de répéter une instruction (ou plusieurs).
- La boucle de base est la boucle for i in range(5) : ça permet de récupérer des i de 0 à 4 et de faire 5 fois des actions
- On utilise une boucle for quand on sait combien de fois on va boucler.

```
for i in range(5):
    print(I, i*2)
```

Les différents range()

- range(5) : de 0 à 5 non compris, 1 par 1 par défaut
- range(2, 20) : de 2 à 20 non compris, 1 par 1 par défaut
- range(2, 20, 2) : de 2 à 20 non compris, par pas de 2 (2 par 2).

Boucle for c in chaine

```
for c in "bonjour": # tous les caractères 1 par 1
  print(c)
```

- affiche les lettres 1 par une

Boucle for i in len(chaine)

```
mot = "bonjour"
for in range(len(mot)):
  print(mot[i])
```

- comme le précédent : affiche les lettres 1 par une

Boucle while (condition)

```
import random
valeur_au_hasard=0
while (valeur_au_hasard <=5)
  valeur_au_hasard = random.randint(1, 10)
  print(valeur_au_hasard)
```

- On utilise une boucle while quand on ne sait combien de fois on va boucler.

break : pour sortir de la boucle

- Dans le code précédent, on peut écrire :

```
import random
valeur_au_hasard=0
while (True)
  valeur_au_hasard = random.randint(1, 10)
  if valeur_au_hasard <=5 :
     break
  print(valeur_au_hasard)
```

- Dans ce cas, on n'affiche les valeurs tant qu'elles sont > 5

2-6 - Séquences

- Séquence = ensemble fini et indicé de n éléments ordonnées de 0 à n-1.
- Les types « séquence » sont principalement :

  ⇒ les **chaines de caractères** : 'toto à 20 ans'
  ⇒ les **listes** : [1, 5, 3]
  ⇒ les **tuples** : (1,5, 3).
- range(5), range(1,5) et range(1, 5, 2) sont aussi des séquences.

Usages communs à toutes les séquences :

```
phrase='toto a 20 ans'    # séquence string

type(phrase)        # <class 'str'>
phrase[0]           # t
len(phrase)         # 13
'20' in phrase      # True
'20' not in phrase # False
phrase + '. Il est étudiant' # concatenation
phrase.index('o')   # 1 : position du 1er 'o', erreur si pas
phrase.count('o')   # 2 : il y a 2 'o'
min(phrase)         # espace : code ASCII(' ')=ord(' ')=32
max(phrase)         # é :      code ASCII('é')=ord('é')=233
'cou_'*3            # cou_cou_cou
```

```
liste=[2, 20, 1, 20]    # séquence liste

type(liste)         # <class 'list'>
liste [0]           # 2
len(liste)          # 4
20 in liste         # True
20 not in liste     # False
liste + [6, 20]     # concatenation :[2, 20, 1, 20, 6, 20]
liste.index(20)     # 1 : position du 1er 20, erreur si pas
liste.count(20)     # 2 : il y a 2 20
liste.count(100)    # 0 : il n'y a pas de 20
min(liste)          # 1
max(liste)          # 20
[1, 2]*3            # [1, 2, 1, 2, 1, 2]

# Pareil avec des tuple
```

Slicing

Le slicing permet d'extraire des morceaux, d'insérer et de supprimer.
Il y a 3 paramètres : début, fin, pas (comme dans un range).
C'est très pratique !

- sequence [debut :fin :pas]
- Par défaut : début vaut 0
- Par défaut : slice vaut len(sequence)
- Par défaut : len(sequence) peut être supprimé
- Par défaut le pas c'est 1

```
phrase='toto a 20 ans'

# slice positifs :
a='bonjour'
a[:]           # bonjour
a[0:len(a)]    # bonjour
a[0:50]        # bonjour

a[1:]          # onjour
a[2:]          # njour

a[:]           # bonjour
a[0:len(a):1] # bonjour
a[0:len(a):2] # bnor : pas de 2 : 1 sur 2

a[::1]         # bonjour
a[::2]         # bnor

a[2::2]        # nor : on démarre à n, par pas de 2 jusqu'à la fin
```

- ***Slicing, suite***

```
#-------------------------------------
# slice avec debut et ou fin négatifs :
# len(a) peut être supprimé

a[len(a)-1:]   # r : on démarre au dernier
a[-1:]         # r : on démarre au dernier
a[-2:]         # ur : on démarre à l'avant-dernier

a[:len(a)-1]  # bonjou : on va jusqu'à l'avant-dernier
a[:-1]        # bonjou : on va jusqu'à l'avant-dernier
a[:-2]        # bonjo
a[2:-2]       # njo

#-------------------------------------
# slice avec pas négatif : on avance de la fin au début
# attention on part de la fin, on va vers le début
a='bonjour'
a[::-1]       # ruojnob
a[::-2]       # ronb
a[1:5:-1]     # vide car le début est avant la fin
a[-1:-3:-1]   # ru : de len-1 à len-3 non compris

# avec un pas négatif, le début par défaut c'est -1, la fin par
défaut c'est -len-1
```

Séquence remplie et séquence vide

```
sequence = '' # ou =[] ou )() ou range(0)

if sequence :
  print('sequence remplie')
else :
  print('sequence vide') # c'est ce qui s'affiche
```

2.7 - Les listes

- **Type souple, puissant et central en python** : au cœur des programmes python.
- **Une liste est une séquence** : toutes les opérations applicables aux séquences sont applicables aux listes (paragraphe précédent : 2.3).
- C'est un **objet mutable** : on peut modifier l'objet sans faire de copie.
- On peut ajouter des éléments, en supprimer, au début, au milieu, à la fin.
- Une liste est une « Séquence de références d'objets hétérogènes » : la taille d'une liste c'est la taille des références, pas la taille des objets qu'elle contient. La fonction sys.getsizeof(var) permet de connaître la taille d'une liste.

Usages de base

```
li=[10, 20, 30]
print(li)             # [10, 20, 30]

for i in range(len(li)):
  print(li[i])        # 10 20 30 l'un en dessous de l'autre

for val in l:
  print(val)          # 10 20 30 l'un en dessous de l'autre

prenoms=['toto', 'titi', 'tata']
print(prenoms)        #['toto', 'titi', 'tata']

for i in range(len(prenoms)):
  print(prenoms[i])   # toto titi tata l'un en dessous de l'autre

for prenom in prenoms:
  print(prenom)       # toto titi tata l'un en dessous de l'autre
```

Création

```
l=[]                    # initialisation d'une liste vide
print(type(l))          # <class 'list'>
print(l)                # []

l=[10, 20, 30]
print(l)                # [10, 20, 30]

l=list(range(5))
print(l)                # [0, 1, 2, 3, 4]

l=[i for i in range(5)]
print(l)                # [0, 1, 2, 3, 4]

# on peut aussi mettre tout ce qu'on veut dans une liste :
a=5
l=[a,'coucou', 2.3, True]
print(l)                # [5, 'coucou', 2.3, True]

l=['Pierre', 'Paul', 'Jacques']
```

ajouter : append()

```
l=[10, 20, 30]          # initialisation d'une liste
l.append(40)            # ajoute 40 à la fin : [10, 20, 30, 40]
```

supprimer : pop()

```
val = l.pop()           # retire le dernier : l vaut [10, 20, 30]
print(val, l)           # 40 [10, 20, 30]
```

modifier

```
l[0]=l[1]+5
print(l)                # [10, 25, 30]
```

accès au dernier éléments

```
print(l[len(l)-1])     # 30
print(l[-1])           # 30 : dans les crochets len(l) équivaut à 0
```

Usages communs à toutes les séquences :

```
li=[2,3,4,5,3]
type(li)            # <class 'list'>

li[0]               # 2
len(li)             # 5
4 in li             # True
5 not in li         # False
li.index(3)         # 1 : c'est la position du 1er <3>
li.index(3, 2)      # 4 : c'est la position du 2ème <3>
                    #     à partir de l'élément n° 2
li.count(3)         # 2 : il y a 2 valeurs 3
min(li)             # 2
max(li)             # 5

5*[2]               # [2,2,2,2,2]
1*li                # [2,3,4,5,3,2,3,4,5,3]
li + [6,7]          # concatenation : [2,3,4,5,3,6,7]

del li[2]          # [2,3,5,3]
```

Slicing (cf p.39)

Le slicing permet d'extraire des morceaux, d'insérer et de supprimer.
C'est très pratique !
Il y a 3 paramètres : début, fin, pas (comme dans un range).

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


l[0:3]      # du 0ème au 3ème non compris: ['a','b','c']

l[2:]       # du 2ème à la fin : ['c','d,'e']

l[0::2]     # 1 sur 2 : ['a','c,','e']

l[::-1]     # tout en sens inverse : ['e,','d,','c','b','a']

l[3:1:-1]   # en sens inverse, du 3 au 1 non compris: ['d,','c']



l[1:3]=[6,7,8]     # on remplace 1er et 2ème par [6,7,8]
l                  # ['a', 6, 7, 8, 'd', 'e']

l[5:5]=[1,2]       # on insère ensuite avant le 5ème
l                  # ['a', 6, 7, 8, 'd', 1, 2, 'e']

l[1:4]=[]          # on supprime de 1 à 3, on n'insère rien
l                  # ['a', 'd', 1, 2, 'e']
del l[1:3]         # on supprime de 1 et 2
l                  # ['a', 2, 'e']
```

Rappel du slicing de toutes les séquences :

```
phrase='toto a 20 ans'

print(phrase[7:11])        # '20 a': slicing, 7 à 11 exclu
print(phrase[:4])          # 'toto': 0 à 4 exclu

print(phrase[10:])         # 'ans': de 10 à la fin
print(phrase[10:100])      # 'ans': de 10 à la fin
print(phrase[100:200])     # de 100 à 200 : chaine vide

print(phrase[:])           # renvoie une shallow copie de phrase

print(phrase[0:14:2])      # 'tt  0as': slicing par pas de 2

# print(phrase[100])       # IndexError
print(phrase[-11:-5])      # 'to a 2': de -11 à -6: dernier en -1
print(phrase[:-7])         # 'toto a': du début à -6
print(phrase[3:-3])        # 'o a 20': de 3 à -3: dernier en -1

print(phrase[::-1])        # 'sna 02 a otot':
                           # de fin à début, inversé, pas <0
print(phrase[8:6:-1])      # '02' : du 8 à 7 inversé, pas <0
```

Toutes les méthodes sur les listes

```
dir(list) # append, pop, insert, remove, extend, clear,
          # ll=l.copy() : shallow copy
          # reverse, sort
          # index, count

help(list.append) : pour avoir la doc de la fonction title (str ?
sous ipython - :q  pour sortir)
```

Exemples - 1

```
l=[1,2,3]

l.extend([4,5,6])      # ajoute un objet à la fin de la liste
print(l)               # [1, 2, 3, 4, 5, 6]

l.sort(reverse=True)
print(l)               # [6, 5, 4, 3, 2, 1]
l.sort()               # sort fonctionne « en place » : sur l
                       # jamais d'affectation avec sort()
print(l)               # [1, 2, 3, 4, 5, 6]

x=l.pop()
print(x)               # 6
print(l)               # [1, 2, 3, 4, 5]

x=l.pop(0)
print(x)               # 1
print(l)               # [2, 3, 4, 5]


l.append(6)
l.insert(0, 1)
l.insert(3,'x')
print(l)               # [1, 2, 3, 'x', 4, 5, 6]

l.remove('x')
print(l)               # [1, 2, 3, 4, 5, 6]
del l[2]               # [1, 2, 4, 5, 6]
```

Exemples - 2

```
ll=l.copy()             # shallow copy : duplication des éléments
ll.pop()
print(ll)               # [1, 2, 4, 5]
print(l)                # [1, 2, 4, 5, 6]

ll.clear()
print(ll)               # []
print(l)                # [1, 2, 4, 5, 6]
```

Exemples - 3

```
s='bonjour tout le monde'
l=s.split()
print(l)                # ['bonjour', 'tout', 'le', 'monde']
l[0]=l[0].upper()
print(l)                # ['BONJOUR', 'tout', 'le', 'monde']
print(" ".join(l))      # 'BONJOUR tout le monde'
print(" - ".join(l))    # 'BONJOUR - tout - le - monde'
```

Technique spéciale : suppression de tous les doublons

On passe par une liste spéciale : un ensemble = set, défini entre {}

```
l=[1, 5, 2, 3, 2, 1, 3 , 4]
s=set(l)
type(s)       # set
l=list(s)
l             # [1, 2, 3, 4, 5]

# version coute :
l=[1, 5, 2, 3, 2, 1, 3 , 4]
l = list(set(l))
```

2.8 - Boucle et fonction : factorisation du code

Boucle for, range(), enumerate()

- Une boucle permet de répéter une instruction (ou plusieurs).
- La boucle de base est la boucle for.

  ⇒ La boucle for du python est l'équivalent des foreach d'autres langages : on accède à la valeur, pas à l'indice.
- range(5) est un objet équivalent à une liste de 5 éléments de 0 à 5 : [0,1,2,3,4]

```
for val in range(5): # range(5) est un objet <=> [0,1,2,3,4]
    print(val**2)
print('------------------------')

for val in [5, 2, 4, 3, 1]:
    print(i**2)
print('------------------------')

for val in ['coucou', 2, True]:
    print(i)
```

for + enumerate() : accès à l'indice

- La fonction enumerate() permet d'avoir un compteur de 0 à n-1 dans la liste :

```
villes = ["Paris", "Nice", "Lyon"]
for i, ville in enumerate(villes):
    print(i, ville)
```

```
for i, val in enumerate(range(5, 10)):
    print(i, val)
```

Fonction

- Une fonction permet de réutiliser le même code sans avoir à le réécrire.

```
def printListeCarres(liste):
    print('------------------------')
    for i in liste :
        print(i**2)

liste1=[0, 1, 2, 3, 4]
printListeCarres(liste1)

printListeCarres([10, 4.3, -5, 2.3e3])
```

- liste1 et [10, 4.3, -5, 2.3e3] sont passées par référence à printListeCarres : autrement dit, ce n'est pas toute la liste qu'on passe en paramètre, mais uniquement la référence de l'espace des variables qui permet d'accéder à la liste dans l'espace des objets.
- Version avec une fonction qui retourne la liste des carrés

```
def returnListeCarres(liste):
    listeCarres=[]
    for i in liste :
        listeCarres.append(i**2)
    return listeCarres

liste1=[0, 1, 2, 3, 4]
listeCarres= returnListeCarres(liste1)
print('------------------------')
print(listeCarres)

print('------------------------')
print(returnListeCarres([10, 4.3, -5, 2.3e3]))
```

Subtilités d'écriture elif et return

- Fonction de vérification si un nombre est premier, version 1 :

```
def premier(n):
    if n <= 0:                  # cas 1
        return                  # return None pour 1 entrée non valide
    elif n == 1:                # cas 2
        return False
    else:                       # cas 3
        for i in range(2, n):
            if n % i == 0:
                return False
    return True                 # cas 4
```

- **Version sans elif** : comme on return dans chaque alternative, on peut remplacer le elif par un if et supprimer le else

```
def premier2(n):
    if n <= 0:                  # cas 1
        return                  # return None pour 1 entrée non valide
    if n == 1:                  # cas 2
        return False
    for i in range(2, n):       # cas 3
        if n % i == 0:
            return False
    return True                 # cas 4
```

- **Version compacte** et optimisée

```
def premier3(n):
    if n <= 0:  return                  # cas <=0 - entrée non valide
    if n == 1:  return False #          # cas =1
    if n % 2 == 0: return False #       # cas pair
    for i in range(3, n, 2):
        if n % i == 0: return False     # cas general pas premier
    return True                         # cas ou n est premier
```

Subtilités d'écriture formatée

- Test avec formatage des entiers :

```
for test in [-2, 1, 2, 4, 19, 35]:
    print(f"premier({test:2d}) = {premier(test)}")
```

  - A noter : **{test:2d}** qui permet de formater l'entier sur 2 caractères.
  - On peut aussi faire {reel :5.2f} qui formate le réel sur 5 caractères avec 2 chiffres après la virgule :

```
import math
print(f"PI: ({math.pi:5.2f})")
```

  - On peut aussi faire {chaine : >20s} qui formate la chaine sur 20 caractères en alignant à droite (>) :

```
spam="spam, bacon, eggs"
print(f"Monty Python: ({spam:>20s})")
```

  - https://pyformat.info

2.9 - Écriture en compréhension

- En mathématiques, pures !, on peut écrire des définition en compréhension. Par exemple :
  - E = {log(k) / k ∈ {1, 2, …, 9, 10} }. C'est une définition mathématique en compréhension. On la lit ainsi : E égale l'ensemble des logarithmes de k tel que k appartient à l'ensemble {1,2,3,4,5,6,7,8,9,10} qui est défini en semi-extension.
  - Ce type d'écriture se retrouve directement en Python. On peut écrire :

```
E = [ math.log(k) for k in range(1,11) ]
```

- En mathématiques, si on veut définir les entiers multiples de 3 inférieurs à 100 peut écrire :
  - E= {k / k ∈ {1, 2, …, 99, 100} et k%3==0}.
  - On pourrait l'écrire en python :

```
E= [ k for k in range(1, 101) if k%3==0]
```

- Autre exemple : on veut calculer les log de valeurs contenues dans une liste listeDepart si ces valeurs sont positives.

```
listeDepart=[5, 4.3, -2, 10, 0]
listeLog= [math.log(k) for k in listeDepart if k>0]
```

- On peut appliquer ça à des chaines de caractères. Si on veut reformater une liste de noms tout en minuscules avec la première lettre en majuscule (fonction capitalize()) et remettre en ordre alphabétique, on écrit :

```
lesNoms=['MAcron', 'LePen', 'Pecresse', 'MELENCHON', 'hidalgo',
'JAdot', 'taubIRA', 'monteBourg']

lesNoms=[nom.capitalize() for nom in lesNoms]
lesNoms.sort()
```

  - Le tri fonctionne « sur place » c'est-à-dire qu'il modifie la liste à laquelle il s'applique et ne retourne rien. On ne peut donc pas l'appliquer sur les [ ] de l'écriture en compréhension. Il faut passer par une variable.
- Avec un peu de pratique, ce type d'écriture est plus facile à maintenir que des boucles for et des if.

2.10 - Notion de module

Présentation

- Un module est un fichier .py.
- Quand on importe ce fichier avec l'instruction import, on accède aux fonctions, aux variables et aux classes qu'il définit.
- Ce fichier peut être :
  - un fichier qu'on a écrit nous-même
  - un fichier de la bibliothèque standard : https://docs.python.org/fr/3/library/index.html
  - un fichier d'un module non-standard
- Exemple d'utilisation : le module random

```
import random
print(random)
dir(random)
help(random.randint)
```

- Exemple d'utilisation : la fonction randint()

```
import random
print(random.randint(1,100))  # un entier entre 1 et 100
```

La librairie standard

- Il y a une centaine de modules standards : c'est la bibliothèque standard.
- On distingue la bibliothèque standard des fonctions natives ou builtin comme abs(), float() ou input() qui n'ont pas besoin d'import pour être accessibles.
- Possibilités de la librairie standard :
  - Calcul mathématique : https://docs.python.org/fr/3/library/numeric.html : math, cmath, decimal, fractions, random, statistique…
  - OS : https://docs.python.org/fr/3/library/allos.html : os, io, time, etc.
    - os.getcwd()
    - os.listdir()
  - Persistance des données : https://docs.python.org/fr/3/library/persistence.html
  - Compression de données et archivage : https://docs.python.org/fr/3/library/archiving.html
  - Formats de fichiers : https://docs.python.org/fr/3/library/csv.html : csv, configparser, etc. :
  - Cryptographie : https://docs.python.org/fr/3/library/crypto.html
  - Programmation fonctionnelle : https://docs.python.org/fr/3.6/library/functional.html
  - Programmation parallèle : https://docs.python.org/fr/3/library/concurrency.html : threading, multiprocessing…
  - Réseau : https://docs.python.org/fr/3.6/library/ipc.html
  - Internet : https://docs.python.org/fr/3.6/library/internet.html
  - Graphisme : tkinter, etc. : https://docs.python.org/fr/3/library/tk.html
  - Écrire des expressions régulières : regex : https://docs.python.org/fr/3/howto/regex.html
  - Gérer des dates : datetime : https://docs.python.org/fr/3.6/library/datetime.html
  - API, entrées/sortie asynchrone : https://docs.python.org/fr/3/library/asyncio.html
  - Les outils de développement : https://docs.python.org/fr/3/library/unittest.html
  - Etc.

Les modules non standards

- Principes
  - Il y en a des centaines de milliers !
  - On peut les charger et les importer dans nos programmes.
  - Par exemple : numpy et pandas pour du datasciences.
- Python Package Index : PyPI.
  - Les modules non standards se trouvent sur la plateforme de de modules Python Package Index : PyPI.
  - A partir du PyPI, on peut trouver les modules qui nous intéressent : par exemple, on cherche numpy, on trouve : numpy 1.19.0. Pour l'installer on fait un « pip install numpy »
  - Dans le cas de numpy, on trouve aussi un site dédié : https://numpy.org
- pip, pip3
  - Les modules non standards s'installent avec l'outil pip ou pip3 selon le système d'exploitation.
  - https://docs.python.org/fr/3/installing/index.html

Techniques d'importation

- Importation de modules standards ou non standards installés avec pip :

```
import random
print(random.randint(1,100))  # un entier entre 1 et 100

from random import randint
print(randint(1,100))  # un entier entre 1 et 100

from random import *
print(randint(1,100))  # un entier entre 1 et 100
```

- Importation de modules-fichiers de notre application :

```
# Fichier FifoTU.py : pour faire les tests unitaires de Fifo.py
# Fifo.py déclare une classe
```

⇒ Option 1 : Fifo.py est dans le même dossier que FifoTU.py

```
from Fifo import Fifo
```

⇒ Option 2 : Fifo.py est dans le dossier « mesClasses » qui se trouve dans le dossier de FifoTU.py

```
import sys
sys.path.append("mesClasses")
from Fifo import Fifo
```

⇒ Option 3 : Fifo.py est dans le dossier parent du dossier de FifoTU.py

```
import sys
sys.path.append("..")
from Fifo import Fifo
```

- Option 4 : Fifo.py est dans le chemin en dur fournit à sys.path

```
import sys
sys.path.append("/var/www/usager/django")
from Fifo import Fifo
```

- Il y en a des centaines de milliers !
- On peut les charger et les importer dans nos programmes.
- Par exemple : numpy et pandas pour du datasciences.

2.11 - Première approche des exceptions

Principe de l'exception

- Quand un code bogue, il génère une exception.
- Exemple de base : 1/0

```
>>> 1/0
ZeroDivisionError: division by zero
```

- L'erreur provient d'une exception : on ne peut pas diviser par 0.
- L'erreur arrête le programme.
- Le but va être de capturer les exceptions pour en faire ce qu'on veut et éviter que le programme s'arrête.

try … except

- Le **try** permet d'essayer. Le **except** permet de réagir en cas d'exception dans l'essai.
- On peut faire le try …. except où on veut dans la remontée de l'exception : **capturer l'exception où on veut** pour la traiter.

Exemple

- Ici une fonction qui peut générer une exception : diviser(a, b)
- On capture l'exception à l'usage.

```
def diviser(a, b):
    print(a/b)

n=0
try:
    diviser(1,n)
except ZeroDivisionError:
    print('Division par 0 impossible')

print('continuons notre programme ...')
```

  - Le programme va afficher 'Division par 0 impossible'
  - Puis il va continuer.
  - Sans le try except, il aurait « planté » et se serait arrêté.

Type d'exception

- Dans le code précédent, on a précisé : « ZeroDivisionError »
- Il faut préciser la bonne exception pour que ça traite l'exception. Si on met un « FloatingPointError», ça ne traitera pas l'exception de la division par 0.
- On peut aussi ne pas préciser le type : dans ce cas, ça traite tous les cas.
- Il y a une hiérarchie des exceptions : https://docs.python.org/3/library/exceptions.html#exception-hierarchy

- Exemple sans type d'exception :

```
def diviser(a, b):
    print(a/b)

n=0
try: diviser(1,n)
except: print('Division par 0 impossible')

print('continuons notre programme ...')
```

  - Le programme va afficher 'Division par 0 impossible'
  - Puis il va continuer.
  - Sans le try except, il aurait « planté » et se serait arrêté.

Exemple : vérifier le type saisi :

- On doit saisir un entier : comment gérer l'erreur ?

  ⇒ On boucle sur la saisie tant que ça n'a pas marché :

```
while True:
  try:
     a=int(input("a:"))
     break
  except ValueError:
     print("entier !!!")
print("a:", a)
```

- Avec une fonction :

```
def inputInt(texte):
  while True:
     try:
        a=int(input(texte))
        return a
     except ValueError:
        print("entier attendu")

a=inputInt("a: ")
print("a saisi:", a)
```

3 – Bases syntaxiques - 2

3.1 - Les fichiers – JSON - MySQL

Création d'un fichier texte

```
f=open("./tmp/testFichier.txt", "w", encoding='utf-8')
for i in range(10):
    f.write(f"numéro {i}\n")

f.close()
```

- open est une fonction builtin de Python.
- Mode d'ouverture : w (write), r (read)
- En mode « w », le fichier va être créé s'il n'existe pas déjà, écrasé s'il existait déjà. Le dossier « tmp » doit exister, sinon, on a une erreur.
- open() retourne un objet fichier. C'est un fichier texte.
- f.write permet d'écrire du texte en format string.
- \n pour passer à la ligne.
- Sans le f.close, le fichier n'est pas mis à jour.

Parcours d'un fichier

- Un fichier est un **itérateur** : on peut le mettre dans une boucle for.

```
f1=open("./tmp/testFichier.txt", "r", encoding='utf-8')
f2=open("./tmp/testFichier2.txt", "w", encoding='utf-8')
for ligne in f1:
    ligne=ligne.split() # la chaine devient un tableau de mots
    ligne[0]=ligne[0].upper()
    f2.write(" : ".join(ligne)+ "\n") # join : inverse de split

f1.close()
f2.close()
```

- On réouvre le fichier créé en mode read : « r ».
- On crée un 2ème fichier en mode wrire : « w »
- On récupère chaque ligne du fichier f1 avec un for
- On transforme la ligne en tableau de mots
- On modifie le 1er mot pour montrer que c'est possible
- On refabrique une chaîne à partir du tableau de mots avec un « : » en séparateur.
- Et on rajoute le « \n »

Nom des fichier : attention au /

- Attention au / dans le nom des fichiers
- Sur MAC, le \ (anti-slash ou back-slash) ne marche pas
- Sur PC, si on utilise un \ , il faut ajouter un « r » (pour row string) devant la chaine de caractère du fichier :

```
f=open(r"file\testFichier.txt", "w", encoding='utf-8')
```

Fichier Binaire

- Un fichier binaire est un objet fichier particulier.
- Un fichier binaire permet de lire et d'écrire du binaire et non plus du texte.
- Un fichier binaire s'ouvre en mode binaire : 'rb', 'wb', ou 'rb+' ('rb+' c'est read et write).
- Il permet de stocker des int, des float, les listes, des objets, etc.

Les fichiers avec context manager et « with … as »

- Un « context manager » c'est du code qui s'exécute tout seul en fonction du contexte.
- Dans le cas des fichiers, on aimerait que la fermeture du fichier soit géré automatiquement (ce qui sécurise le code en cas d'erreur).
- L'instruction « with … as » permet ça.

```
with open("tmpp/testFichierWith.txt", "w", encoding='utf-8') as
f:
  for i in range(10):
    f.write(f"numéro {i}\n")
```

- Le with gère la fermeture du fichier.

```
with open("testFichierWith.txt", "r", encoding='utf-8') as f1:
  with open("testFichierWith2.txt", "w", encoding='utf-8') as f2:
    for ligne in f1:
      ligne=ligne.split() # la chaine devient un tableau de mots
      ligne[0]=ligne[0].upper()
      f2.write(",".join(ligne)+ "\n") # join : inverse de split
    print("travail terminé")
```

  - Si le fichier testFichierWith.txt n'existe pas, on a une erreur : FileNotFoundError
  - Si le fichier testFichierWith2.txt n'est pas modifiable (après un chmod 444 par exemple), on a une erreur : PermissionError
  - On peut encadrer le premier with avec un « try » et gérer toutes les exceptions.

Fichier JSON, XML, CSV

- On utilise le module json
- Il va nous permettre :

  ⇒ de stocker nos données en format JSON : json.dump(data, fileJSON)
  ⇒ de charger (parser) des données en format JSON dans un objet Python : data=json.load(fileJSON)

  ⇒ C'est très facile !

```
import json

data = [
    # des types qui ne posent pas de problème
    [1, 2, 'a', [3.23, 4.32], {'eric': 32, 'jean': 43}],
    # un tuple : deviendra une liste
    (1, 2, 3),
]
'''
data = [
  { "id":1, "nom": "toto", "notes": [15, 12, 13] },
  { "id":2, "nom": "tata", "notes": [16, 14, 9] }
]
'''
# écrire data dans un fichier JSON
with open("./tmp/s1.json","w", encoding='utf-8') as fileJSON:
    json.dump(data, fileJSON)

# lire un fichier JSON
with open("./tmp/s1.json", encoding='utf-8') as fileJSON:
    data2 = json.load(fileJSON)

data
data2
```

- On peut faire la même chose en XML -> doc ici.
- Et la même chose en CSV (format excel) -> doc ici

API JSON et fichier JSON sur le WEB

- On utilise le module requests :
  - PyPI : https://pypi.org/project/requests/
  - Documentation officielle : https://docs.python-requests.org/en/latest/
  - Version française : https://fr.python-requests.org/en/latest/

- Le module « requests » va nous permettre d'accéder à des API ou à des fichiers sur le WEB. Ça peut être en GET ou en POST. Ici, on ne regarde que le GET de données JSON.

- Exemples de tutos :
  - https://www.nylas.com/blog/use-python-requests-module-rest-apis/#roles-of-http-apis-rest
  - https://rtavenar.github.io/poly_python/content/api.html

- Installation :

```
C:>pip3 install requests
```

- Code python pour accéder à un fichier : http://bliaudet.free.fr/json/eleves.json

```
import requests

url = 'http://bliaudet.free.fr/json/eleves.json'
response = requests.get(url)
responseJSON = response.json()

print(responseJSON)


url = 'http://bliaudet.free.fr/json/frequentation-dans-les-
salles-de-cinema.json'
response = requests.get(url)
responseJSON = response.json()

print(responseJSON)
```

- Code python pour accéder à une API JSON :

  https://opendata.paris.fr/api/records/1.0/search/?dataset=velib-disponibilite-en-temps-reel&rows=2000

```
import requests
url =
'https://opendata.paris.fr/api/records/1.0/search/?dataset=velib-
disponibilite-en-temps-reel&rows=2000'
response = requests.get(url)
responseJSON=response.json()
records=responseJSON['records']

print(len(records)) # 1436
print(records[0]['fields']['name']) # Benjamin Godard - Victor
Hugo
```

  - C'est la même structure de code que pour un fichier

- Gestion des erreurs :

```
# il y a une erreur dans l'url : il manque un s à « eleve »
# on aura une HTTPError

url = 'http://bliaudet.free.fr/json/eleve.json'
try:
    response = requests.get(url, timeout=5)
    response.raise_for_status() # pour gérer les erreurs
    # si tout va bien, on traite la response
    responseJSON = response.json()
    print(responseJSON)
except requests.exceptions.HTTPError as errh:
    print(errh)
except requests.exceptions.ConnectionError as errc:
    print(errc)
except requests.exceptions.Timeout as errt:
    print(errt)
except requests.exceptions as err:
    print(err)
```

Connection à un serveur MySQL

➢ ***On utilise les connecteurs MySQL :***

- Doc : https://dev.mysql.com/doc/connector-python/en/connector-python-example-connecting.html
  - Autre doc : https://info.blaisepascal.fr/gerer-une-base-de-donnees-avec-python
- Installation :

```
C:>pip3 install mysql-connector-python
```

➢ ***Récupération de données avec Python sur un serveur MySQL local***

- Ici, on a un utilisateur « root » avec password « root » et une BD « biblio » avec une table « emprunter » :

```
import mysql.connector

cnx = mysql.connector.connect(user='root', password='root',
                              host='127.0.0.1',
                              database='biblio')

cursor = cnx.cursor()

query = (
   "SELECT NA, datEmp, NL "
   "FROM emprunter "
   "WHERE dateRet is null "
   "ORDER BY NA"
)

cursor.execute(query)

# version 1
emps = cursor.fetchall()
for (NL, datEmp, NA) in emps:
  print(f"{NL:3d} - {datEmp} - {NA:3d}")

""" # version 2
cursor.execute(query)
for (NL, datEmp, NA) in cursor:
  print(f"{NL:3d} - {datEmp} - {NA:3d}") """

cursor.close()
cnx.close()
```

- ***Récupération de données avec Python sur un serveur MySQL distant (par exemple, celui du prof !) :***
  - Il faut commencer par récupérer l'IP de la machine distante (ipconfig sur PC, préférence système-réseau sur MAC)
  - Ensuite il suffit de se connecter sur le bon host : si les machines partagent le même réseau (comme celui de l'école), ça va fonctionner :

```
cnx = mysql.connector.connect(user='root', password='root',
                              host='192.168.0.41',
                              database='biblio')
```

- ***Connexion à distance via un client mysql :***

```
mysql -uroot -p -h192.168.0.41
enter password: root
```

- ***Création d'un utilisateur MySQL pour permettre une connexion à distance***

```
CREATE USER root@'%' IDENTIFIED BY 'root';
GRANT ALL PRIVILEGES ON *.* TO root@'%' IDENTIFIED BY 'root';
```

3.2 - Les tuples

Principes

- **C'est une séquence, comme une liste** : on peut lui appliquer les méthodes de séquence : **appartenance**, **crochets**, **slicing**, etc.
- Un tuple est très proche d'une liste : il **peut contenir des données hétérogènes**.
- **A la différence d'une liste**, un tuple est **immuable** : une fois créé, on ne peut pas changer son contenu.
- Un tuple s'écrit avec des **parenthèses**, en séparant les éléments avec une virgule. Il doit forcément y avoir au moins une virgule. On peut se passer des parenthèses.
- Les tuples seront utilisés comme clé de dictionnaire. On y reviendra.

Exemples de code

```
t=()     # objet de type tuple, vide : sans intérêt car immuable !
type(t)  # <class 'tuple'>

t=(3,)   # avec un seul élément il faut une virgule
         # sinon 3 est un entier, les parenthèses un regroupement
t=(3, 2.5, 'tuple', True)
t=3, 2.5, 'tuple', True    # les parenthèses sont facultatives

2.5 in t # True
t[2]     # 'tuple'
t[:2]    # (3, 2.5) le slicing produit un tuple

# conversion list - tuple
a=list(t)  # a vaut [3, 2.5, 'tuple', True]
a[1]=0.5   # on modifie a
t=tuple(a) # t vaut (3, 0.5, 'tuple', True)
           # c'est une technique pour modifier un tuple

# tuple unpacking
(a,b) = [3,4]  # donne les valeurs 3 et 4 à a et b
a, b = 3, 4    # idem sans parenthèses et tuple à droite
a, b = b, a    # inversion en 1 ligne !

# extended tuple unpacking  -> doc
a=list(range(10)) # a est une liste de 0 à 9
x, *y = a         # x vaut 0, y est une liste de 1 à 9
*x, y = a         # x est une liste de 0 à 8, y vaut 9
```

Subtilités

- Écriture sur plusieurs lignes

```
mon_tuple = ([1, 2, 3],
             [4, 5, 6],
             [7, 8, 9],
            )
```

- 2 variables dans un for in

```
entrees = [(1, 2), (3, 4), (5, 6)]
for a, b in entrees:
    print(f"a={a} b={b}")
```

- La fonction zip fusionne 2 listes en 1 liste de tuple. Mais ce n'est pas vraiment une liste : c'est un itérateur, qui évite la charge en mémoire. On y reviendra

```
l1 = [1, 2 ,3, 4]
l2 = [11, 12, 13, 14]
l3 = [21, 22, 23, 24]
l=zip(l1, l2, l3)   #[(1,11,21), (2,12,22), (3,13,23), (4,14,24)]
for t in l:
   print(t)
```

3.3 - Les tables de hash

Principes théoriques

- Les tables de hash sont une structure de données qui permet de pallier à certaines limitation des types séquence.
- Les **séquences** sont **optimisées** pour **l'accès, la modification et la suppression.**
- Les **séquences** ne sont **pas optimisées** pour tester **l'appartenance**.
  - Dans une séquence, le test d'appartenance est un parcours séquentiel. Sa durée (ou complexité) est directement fonction du nombre d'éléments.
- Les **tables de hash** sont **optimisées** en plus pour tester **l'appartenance**.
- Les **séquences** permettent un **accès par un indice entier**, pas par un indice string
  - On peut écrire : a[0]
  - On peut peut pas écrire a['tuple']
  - Par exemple, pour gérer un dictionnaire, il faudrait pouvoir écrire dico['python'] et qu'on récupère la définition du python ('serpent constricteur…')
- Une **table de hash** fonctionne avec des **couples clé-valeur** (key-value).
  - La clé peut être une chaine de caractères (ou un int, un float, un booléen).
  - Une **clé** est forcément **unique** (pensez clé = id) : il n'y a pas de doublon dans une table de hash.
  - Une table de hash est associée à une **fonction de hash** qui fournit un **indice dans une table pour la clé proposé**. Par exemple, pour la clé 'python' du dictionnaire, on récupère l'indice 512.
  - La taille de la table de hash est fixée au départ. Elle ne changera pas. Si 2 clés on le même indice par la fonction de hash, on stocke les deux clé-valeur dans la même case du tableau et le système se débrouille pour récupérer le bon élément quand il le recherche. On parle de **collisions**.
  - Le bilan est que **le « in » est un accès direct et pas séquentiel**. Le temps d'accès est n'est pas fonction du nombre d'éléments : c'est celui de l'exécution de la fonction de hash. Tant que les collisions ne sont pas trop nombreuses sur une même case, cela n'impacte pas les performances. Les tables de hash permettent de : **accéder, modifier, supprimer ou vérifier l'appartenance** avec une **complexité = O(1).**
- Le python gère tout ça de façon efficace et transparente pour l'utilisateur.
- Il y a **2 types de tables de hash** en python : les **dictionnaires** et les **sets**.

Petit test de vitesse d'accès dans une liste et dans un dictionnaire

```
import time

def newListe(n):
   return [x for x in range(n)]

def newHash(n):
   rep={}
   for x in range(n):
      rep[x]=x
   return rep

l1=newListe(100_000)
debut = time.time()
98_765 in l1
print(time.time() - debut) # donne le temps en secondes

l2=newListe(500_000)
debut = time.time()
498_456 in l2
print(time.time() - debut) # donne le temps en secondes

l3=newListe(1_000_000)
debut = time.time()
987_654 in l3
print(time.time() - debut) # donne le temps en secondes

h1=newHash(100_000)
debut = time.time()
98_765 in h1
print(time.time() - debut) # donne le temps en secondes

h2=newHash(500_000)
debut = time.time()
498_456 in h2
print(time.time() - debut) # donne le temps en secondes

h3=newHash(1_000_000)
debut = time.time()
987_654 in h3
print(time.time() - debut) # donne le temps en secondes

'''
RESULTATS :
0.001422882080078125
0.008266925811767578
0.016958951950073242
3.0994415283203125e-06
3.0994415283203125e-06
3.0994415283203125e-06
'''
```

3.4 - Les dictionnaires

Principes

- Les dictionnaires sont des tables de hash : c'est une **collection de couples clé-valeur.**
- Les dictionnaires sont des **objets mutables** : on peut les modifier.
- Dans un dictionnaire, comme dans toute table de hash, le temps d'accès, de modification, de suppression et de vérification d'appartenance est indépendant du nombre d'éléments.
- La clé d'accès peut être n'importe quel objet « hashable » c'est-à-dire sur lequel on peut appliquer une fonction de hash. Tous les objets immuables sont hashables (classiquement, les chaines de caractères, mais aussi les int, les float, les bool). Certains objets mutables sont hashables, pas tous.
- Pour créer un dictionnaire on écrit : **d={}**
- C'est un objet de type <class 'dict'>
- Un **élément** de dictionnaire se présente sous la **forme « key : value »**
- Il n'y a **pas d'ordre dans un dictionnaire** : la key n'est pas triée.

Exemples de code - 1

```
# creation, remplissage, suppression d'un repertoire : rep
rep = dict()                  # objet de type dict, vide.
del rep
rep = {}                      # objet de type dict, vide.
rep = {'nico':30, 'ahmed':25}
rep['nico']                   # vaut 30
rep.get('nico')               # vaut 30
rep['anne']=28                # ajoute 'anne'
rep. update({'ahmed':24, 'yuwei':22}) # ajoute ou modifie
rep
rep [10]= 'coucou'            # on peut mettre ce qu'on veut
rep [(4,2)]= 'bof'            # la clé peut être un tuple
rep
del rep[10], rep[(4,2)]
rep

#on peut faire des opérations proches des séquences
len(rep)                      # vaut 4
'nico' in rep                 # vaut True

# transformation en liste de clés
list(rep)                     # ['nico', 'ahmed', 'anne', 'yuwei']

# transformation en liste de tuples
l=list(rep.items())           # [('nico', 30), ('ahmed', 24),
                              #  ('anne', 28), ('yuwei', 22)]
l

# passage d'une liste de tuples à un dictionnaire
rep2=dict(l)                  # rep2 est identique à rep
rep2 == rep                   # True
rep is rep2                   # False
```

Exemples de code – 2 – unpacking et vues

```
# parcours d'un dictionnaire : tuple unpacking
for k,v in rep.items():
    print(f"{k} {v}")

for k in rep :       # ne parcourt que les clés
    print(k)

# récupération de « vues » de clés, valeurs et items
# la vue suit les modifications du dictionnaire d'origine

vkey=rep.keys()      # dict_keys(['nico', 'ahmed', 'anne'])
vval=rep.values()    # dict_values([30, 24, 28])

rep['ahmed']=20
vval                 # dict_values([30, 20, 28])

rep.clear()          # ça vide le dictionnaire
```

Dictionnaire et objet – v1

- Le dictionnaire python permet de gérer les objets (ou structure), qu'on trouve classiquement dans d'autres langages.
- On peut ensuite avoir des listes d'objets en python.

```
lesPersonnes = [
    {'nom': 'Pierre',  'age': 25, 'email': 'pierre@spam.com'},
    {'nom': 'Paul',    'age': 18, 'email': 'paul@spam.com'},
    {'nom': 'Jacques', 'age': 52, 'email': 'jacques@spam.com'},
]

personnes[0]['age'] += 1

for personne in lesPersonnes:
    print(30*"=")
    for info, valeur in personne.items():
        print(f"{info:8s} ->  {valeur}")
```

Dictionnaire et objets – v2

- Ce qui n'est pas pratique, c'est de passer par : personne[0]
- Ce qui serait pratique, ce serait de passer par personne['Pierre'] : pour cela, il faut que 'Pierre' soit une clé.
- On va attacher toutes les informations de chaque nom au nom en tant que clé.

```
# on part du tableau d'objets
lesPersonnes = [
    {'nom': 'Pierre',  'age': 25, 'email': 'pierre@spam.com'},
    {'nom': 'Paul',    'age': 18, 'email': 'paul@spam.com'},
    {'nom': 'Jacques', 'age': 52, 'email': 'jacques@spam.com'},
]

# on le tranforme en dictionnaire
dicoLesPersonnes = { personne['nom']: personne for personne in
lesPersonnes}

dicoLesPersonnes

dicoLesPersonnes['Pierre']
      # {'nom': 'Pierre', 'age': 25, 'email': 'pierre@spam.com'}
dicoLesPersonnes['Pierre']['age']          # 25
dicoLesPersonnes['Pierre']['age'] += 1
dicoLesPersonnes['Pierre']['age']          # 26

for nom, record in dicolesPersonnes.items():
    print(f"Nom : {nom} -> enregistrement : {record}")
```

3.5 - Les ensembles

Principes

- Les set sont proches des dictionnaires, mais ils ne stockent qu'une clé et pas de valeurs.
- Le set est utile pour garder les éléments uniques d'une séquence (supprimer les doublons) et pour faire des tests d'appartenance sur un élément d'une séquence.
- On n'a pas d'accès direct à un élément, puisque si on connaît la clé, on connaît l'élément ! Par contre, on peut faire un « in ».

```
s = set()                          # objet de type set, vide.
del s
s = {1, 2, 'a', True, (1, 'b')}  # objet de type set

s = {} # attention, c'est un dictionnaire

l=[1, 2, 1, 3, 2]
s = set(l)                         # {1, 2, 3}

3 in s                             # True
len(s)                             # 3
s.add(6)                           # {1, 2, 3, 6}
s.update([1, 2, 7, 8])             # {1, 2, 3, 6, 7, 8}
s.remove(6)                        # {1, 2, 3, 7, 8}
s.remove(7);s.remove(8)            # {1, 2, 3}

s1=s
s2 = {3, 4, 5}

s1 - s2              # difference    {1, 2}
s1 | s2              # union         {1, 2, 3, 4, 5}
s1 & s2              # intersection  {3}

s.clear()            # ça vide l'ensemble
```

Recommandation

- Si on veut éviter les doublons, il faut un set.
- Dès qu'on veut faire plus d'un test d'appartenance, on a intérêt à convertir la liste en set.

```
import time

def newListe(n):
   return [x for x in range(n)]

try:
  del l1, l2, l3, s1, s2, s3
except:
  pass

l1=newListe(100_000)
debut = time.time();
98_765 in l1
print(time.time() - debut) # donne le temps en secondes

debut = time.time()
s1=set(l1); 98_766 in s1
print(time.time() - debut) # donne le temps en secondes

debut = time.time()
98_767 in s1
print(time.time() - debut) # donne le temps en secondes
print('=====================================')

l3=newListe(1_000_000)
debut = time.time()
987_654 in l3
print(time.time() - debut) # donne le temps en secondes

debut = time.time()
s3=set(l3); 987_655 in s3
print(time.time() - debut) # donne le temps en secondes
debut = time.time()
987_656 in s3
print(time.time() - debut) # donne le temps en secondes
print('=====================================')

'''
RESULTATS :
0.0014836788177490234
0.00471806526184082
1.0967254638671875e-05
=====================================
0.01631784439086914
0.0439910888671875
1.0967254638671875e-05
=====================================
'''
```

3.6 – Les exceptions

Principes

- Une exception est un **mécanisme de communication d'erreur** dans un programme et particulièrement dans une fonction. Ainsi, une fonction peut retourner ce qu'elle a prévu de retourner ou une exception.
- On peut **capturer les exceptions** renvoyées par les fonctions (ou par le programme) et les traiter.
- Les exceptions fournissent de l'information sur l'erreur qui se produit : c'est une **mécanisme de notification d'erreur**.
- C'est un mécanisme utilisé couramment dans le **fonctionnement normal** d'un programme.

Test dans idle

- L'exception basique : division par 0.

```
def appel(x):        # fichier division.py
    divi(1, x)

def divi(a, b):
    print(a/b)
    print('continuons...')
```

- On teste appel(1,2) puis appel(1,0)
- Le print('continuons') ne se fait pas.
- On obtient un message d'erreur sur plusieurs lignes.
- La dernière ligne nous dit le nom de l'exception. Ici : ZeroDivisionError.
- En remontant, on trace le problème : dans le fichier division.py, ligne 4 dans la fonction divi, l'instruction print(a/b)
- En remontant, on trace le problème : dans le fichier division.py, ligne 2 dans la fonction appel, l'instruction divi(1,x)
- Enfin en remontant, on arrive dans le fichier pyshell (notre programme idle), la ligne 1 dans le module (le programme principal, ou le main), l'instruction appel(0)
- Cette suite d'appel de fonctions : appel(0) -> divi(1,x) -> print(a,b) : c'est la **pile d'exécution des fonctions**. print(a,b) est en haut de la pile, appel(0) en base de la pile.

try … except

- Le **try** permet d'essayer. Le **except** permet de réagir en cas d'exception dans l'essai.
- On peut faire le except où on veut dans la remontée de l'exception : **capturer l'exception où on veut** pour la traiter.
- Exemple 1

```
def appel(x):
    divi2(1, x)

def divi(a, b):
    try:
        print(a/b)
    except ZeroDivisionError:
        print('Division par 0')
    print('continuons...')
```

  - Si on teste : >>>appel(1,0)
  - On affiche 'Division par 0' et aussi 'continuons...' : le programme ne s'est pas arrêté.

- On peut traiter l'exception où on veut : ici au niveau de la fonction appel()

```
def appel(x):
    try:
        divi(1, x)
    except ZeroDivisionError:
        print('Division par 0')
    print('continuons...')

def divi(a, b):
    print(a/b)
```

  - Si on teste : >>>appel(1, 0)
  - On affiche 'Division par 0' et aussi 'continuons...' : le programme ne s'est pas arrêté.

Type d'exception

- Si on teste : >>>appel(1, '0'), on a une erreur :
  - TypeError: unsupported operand type(s) for /: 'int' and 'str'
- On ajoute un except TypeError dans le code :

```
def divi(a, b):
    try:
        print(a/b)
    except TypeError:
        print('Il faut des réels et pas des strings')
    except ZeroDivisionError:
        print('Division par 0')
    print('continuons...')
```

- On n'est pas obligé de préciser le type de l'exception (on peut juste mettre « except »), mais c'est à éviter pour une meilleure maintenabilité du code.

try … except as … finaly

- On ajoute un except TypeError dans le code :

```
def unSur(x):
    a = 'unSur:'
    try:
        b='try'
        c=''
        return 1/x
    except ZeroDivisionError as e:
        c='-except'
        # type(e): ZeroDivisionError
        # e: message de l'exception. Ici "division by 0"
        print(f"Spam!!!, {type(e)}, {e}")
        # on remplace le message de l'exception par le return
        return("Division par 0 !!!")
    finally:
        print("finally : on passe toujours ici, avant le return,
avec les variables mises à jour: " + a + b +c)

print(unSur(2))
print(unSur(0))
```

  - **except as** : permet d'accéder au message de l'exception : {e}
  - **le return dans l'except** : modifie le message de l'exception.
  - **finally** : est toujours exécuté en cas de try, qu'on passe par le try ou par le try et l'except. Les variables mises

raise

- On peut retourner des exceptions volontairement avec un raise.
- On choisit un type d'exception qui correspond au contexte. Ici ValueError : pour une valeur d'argument inapproprié. Par défaut, on peut choisir Exception.
- Toutes les exceptions natives -> ici.
- On peut passer plusieurs arguments à ValueError : ils sont accessibles dans l'attribut args. Traditionnellement, le 2ème argument peut être un code d'erreur.

```
def maFonction(a): # a doit être positif
    if a < 0:
        raise ValueError('Erreur : a doit être positif', 99)
    print('Exécution normale de la fonction')

try:
    maFonction(-1)
except Exception as e:
    print(f"type: {type(e)}, e: {e}, code: {e.args[1]}")
    print(e)

maFonction(1)
maFonction(-1)
```

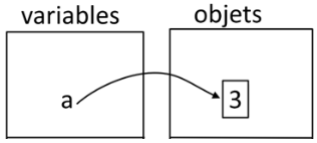
3.7 - Référence partagée, garbage collector, shallow copy, deep copy

Rappels

- On distingue entre espace des variables et espaces des objets
- Quand on écrit :

```
a=3
```

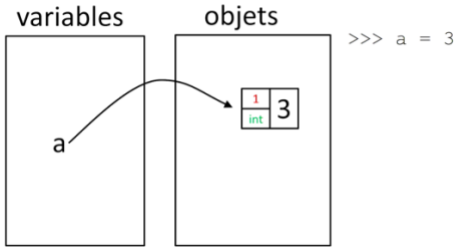
  - Il y a en fait 3 opérations :
    1. Python crée l'objet 3 dans l'espace des objets
    2. Python crée la variable a dans l'espace des variables.
    3. Python fait le lien entre la variable et l'objet



- Une variable est toujours une référence vers un objet.

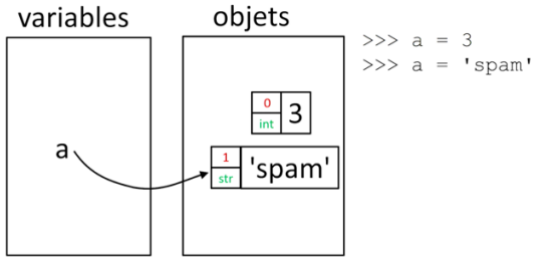
Garbage collector

- Dans l'espace des objets, Python précise, pour chaque objet, son type et le nombre de variables qui le référence.



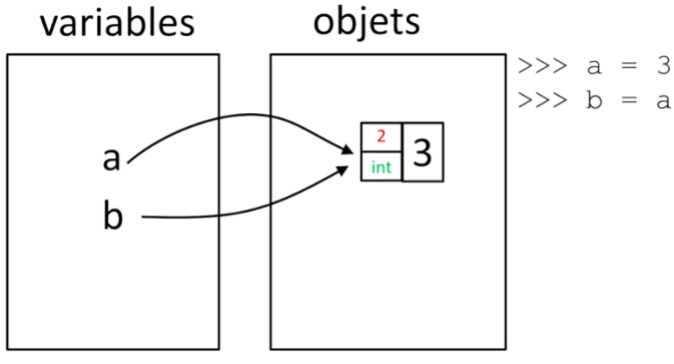
- Si on change de valeur de a, l'objet initial (3) est toujours présent, mais le nombre de variables qui le référence vaut 0.

  ⇒ Le mécanisme de Garbage Collector (module gc) s'occupe de libérer la mémoire dans l'espace des objets quand les objets ne sont plus référencés par personne. On ne s'en occupe pas nous-même.

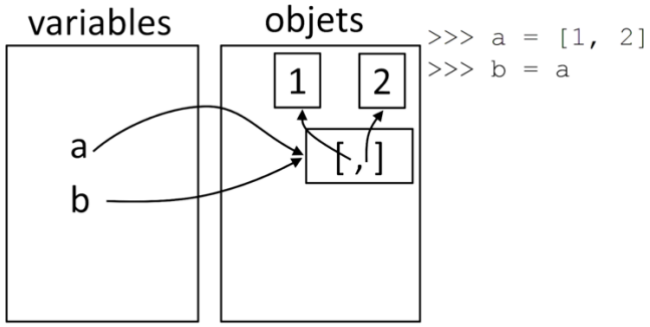


Référence partagée

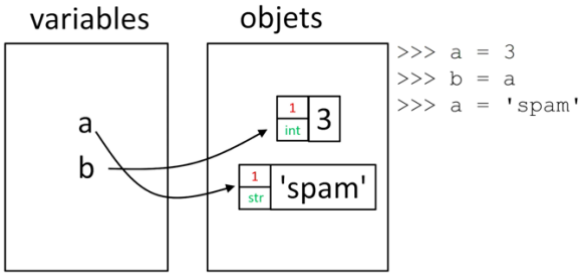
- Quand 2 variables font référence au même objet, on parle de référence partagée. 2 cas se présentent :
  - Référence partagée d'un objet immuable (non mutable)



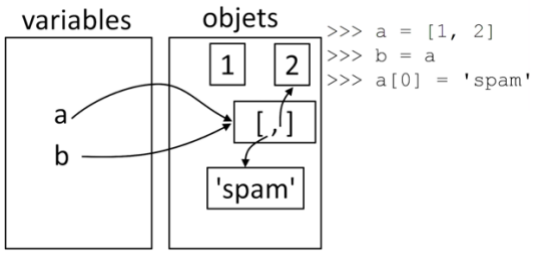
  - Référence partagée d'un objet mutable (par exemple, une liste)



Changement de référence des variables : pas de problème

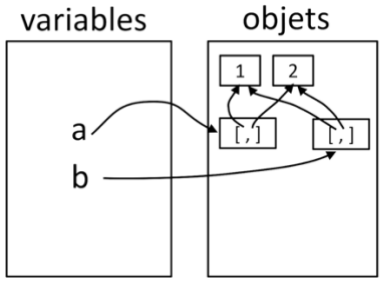


Changement de référence dans l'espace des objets : effet de bord



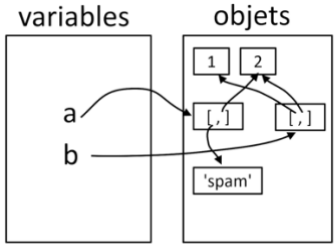
- On parle d'effet de bord quand une modification de la valeur d'une variable entraine la modification de la valeur d'une autre variable : ici, on modifie « a » et « b » est aussi modifié.
- L'instruction « b=a » n'est pas une copie mais un partage de référence.

Shallow copy – 1 : objets de 2ème niveau immuables




```
>>> a = [1, 2]
>>> b = a[:]
```

- L'instruction « b= a[:] » va dupliquer l'objet référencé par a. C'est une « shallow copy »
- Par contre, les objets référencés de 2ème niveau ne sont pas dupliqués : les références internes restent les mêmes.
- Il n'y a pas de risque d'effet de bord si les objets référencés de 2ème niveau sont des objets immuables :

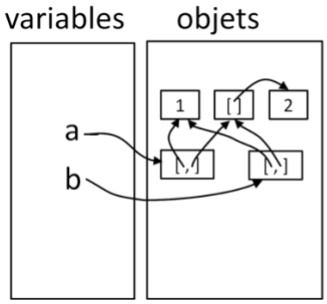



```
>>> a = [1, 2]
>>> b = a[:]
>>> a[0] = 'spam'
```

  - Ici, a contient ['spam', 2] et b [1, 2]

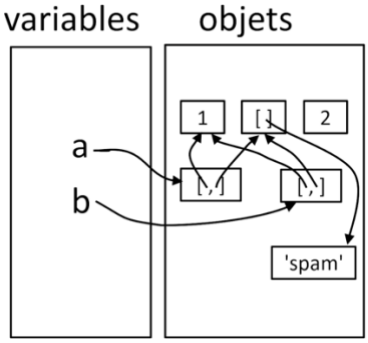
Shallow copy – 1 : objets de 2ème niveau mutable

- Il y a un risque d'effet de bord si les objets référencés de 2ème niveau sont des objets mutables.
  - Ici, la liste contient une liste, donc un objet mutable :




```
>>> a = [1, [2]]
>>> b = a[:]
```

  - Si on écrit a[1][0] = 'spam', ça modifie aussi l'objet b : il y a un effet de bord.

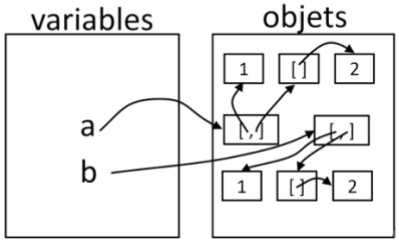



```
>>> a = [1, [2]]
>>> b = a[:]
>>> a[1][0] = 'spam'
```

Deep copy

- Une deep-copy va dupliquer tous les objets de l'espace des objets.
- Elle utilise la fonction « deepcopy » définie dans le module « copy ».
- Avec une deepcopy, il y a une indépendance complète entre les objets (les références) de 2 variables.
  - Dans l'exemple ci-dessous, a et b sont deux variables complètement indépendantes. Il ne peut donc plus y avoir d'effet de bord.

```
>>> a = [1, [2]]
>>> import copy
>>> b = copy.deepcopy(a)
```



- En réalité, Python ne duplique que les objets mutables. Les objets immuables (comme 1 et 2) seront des « singletons », c'est-à-dire des objets qui n'existe qu'une seule fois. Ca permet d'économiser de la mémoire. Comme ce sont des objets immuables, il n'y a pas de risque d'effet de bord.

Opérateur « is »

- a is b vaut True si a et b font référence au même objet.
- Si a est un objet immuable, a is b est équivalent à a == b
- Par exemple, avec un objet immuable :

```
a='spam'
b=a
a == b # True
a is b # True
c='spam'
a == c # True
a is c # True
```

- Par exemple, avec un objet mutable :

```
a=['egg']
b=a
a == b # True
a is b # True
c=['egg']
a == c # True
a is c # False
```

3.8 - Première approche des classes

Rappels des types built-in

- Tous les types proposés par Python s'appellent type « natifs » ou « built-in ».
- Tous les types sont des classes.
- Certains types sont immuables, certains types sont mutables.
- Des fonctions d'informations peuvent être appliquées aux types: dir(type), help(type.méthode)
- Pour toutes les variables, on peut: les supprimer = del(variable), consulter leur type = type(variable), etc.
- De nombreuses fonctions, techniques et méthodes sont communes aux types contenant un nombre d'informations variables (str, list, dict, tuple, set) : len(collection), print(variable), [], [:], collection.pop(), collection.add(value), collection.remove(value), etc.

Ecrire ses propres objets

- On peut écrire ses propres objets qui auront les mêmes fonctionnements que des types built-in : del, len, [], etc.
- Une classe c'est un type.
- On crée des classes, c'est-à-dire nos propres types. A ces types on va associer de l'information qu'on appelle « attributs » et des fonctions de traitement qu'on appelle « méthode ».

Exemple de base : une classe qui ne fait rien

```
class C :
   pass # instruction qui ne fait rien

c=C()   # on créé un objet c de classe C
type(c) # <class '__main__.C'>
```

– On crée un objet c de la classe C = on crée une variable c qui fait référence à un objet de la classe C. Mais cette classe ne fait rien.

Exemple de plus subtil : une classe de liste de mots à partir d'une phrase

- Le mot-clé « self » correspond à l'objet sur lequel on travaille.
- La méthode __init__ est l'initialisateur : la méthode qui donne les valeurs de départ à l'objet qu'on crée.

```
class Phrase:
   def __init__(self, phrase) : # constructeur ou initialisateur
      self.phrase=phrase        # attribut phrase
      self.mots=phrase.split()  # attritut mots

phrase=Phrase('Coucou, c'est moi')
type(phrase)          # <class '__main__.Phrase'>
phrase.phrase         # 'Coucou, c'est moi'
print(phrase.mots)    # ['Coucou,', 'c'est', 'moi']
```

– On crée une variable phrase qui fait référence à un objet de la classe Phrase. A la création, on passe une string en paramètre. A la création, le constructeur est appelé et 2 attributs sont initialisés. On peut accéder à la valeur de ces attributs.

Exemple de plus subtil : on ajoute une méthode

- On va définir une méthode upper() pour notre classe qui mettra tous les mots en majuscule.
- On le code avec une compréhension de liste.
- On pourrait coder un len() qui retourne le nombre de mots, un contains() qui vérifie si un mot est présent.

```
class Phrase:
   def __init__(self, phrase) : # constructeur ou initialisateur
      self.phrase=phrase        # attribut phrase
      self.mots=phrase.split()  # attritut mots

   def upper(self) :
      self.mots=[m.upper() for m in self.mots]

phrase=Phrase('Coucou, c'est moi')
type(phrase)          # <class '__main__.Phrase'>
phrase.phrase         # 'Coucou, c'est moi'
phrase.upper()
print(phrase.mots)    # ['COUCOU,', 'C'EST', 'MOI']
```

– ATTENTION : la ligne vide avant le 2ème def doit avoir les espaces pour un niveau de bloc. Sinon, l'interpréteur pense que la classe est terminée.

Exemple de plus subtil : transformer un objet en chaine

- On va définir une méthode __str__() qui va retourner une chaine de caractère.
- Cette méthode permet de faire un print(objet) et d'afficher la chaine de caractère retournée par __srt__
- Cette méthode permet aussi de faire un str(objet) et de récupérer la chaine de caractères retournée par __str__.
- Du coup, on supprime l'attribut phrase qui ne sert plus.

```
# avec le code précédent
print(phrase)       # <__main__.Phrase object at 0x7fd06c398820>

class Phrase:
   def __init__(self, phrase) : # constructeur ou initialisateur
      self.mots=phrase.split()  # attritut mots

   def upper(self):
      self.mots=[m.upper() for m in self.mots]

   def __str__(self):
      return " ".join(self.mots)

phrase=Phrase("Coucou, c'est moi")
type(phrase)          # <class '__main__.Phrase'>
phrase.upper()
print(phrase.mots)    # ['COUCOU,', 'C'EST', 'MOI']
print(phrase)
print("Voici la phrase de départ : "+str(phrase))
```

- On peut faire pareil avec __len__. Usage : len(phrase)
- On peut faire pareil avec __contains__. Usage « MOI » in phrase

4 – Précisions – Test – Fonction – Portée – Paramètres (surtout théorique)

4.1 - Les fonctions

Une fonction est un objet

- Une fonction est un objet comme un autre. Quand on crée une fonction, on crée une variable du nom de la fonction qui référence la fonction.

```
def f(a, b, c):
   print(a, b, c)

type(f)    #  <class 'function'>
g=f
g(1, 2, 3) # 1 2 3
```

Paramètre immuable en référence

- ATTENTION : les explications ci-dessous sont compliquées à comprendre ! Accrochez-vous !
- Quand on passe un paramètre à une fonction, on passe un objet (de l'espace des objets) qui correspond à ce paramètre. La variable qui correspond au paramètre dans la fonction va référencer cet objet. C'est pour cela qu'on dit que le paramètre est en référence.
- Dans l'exemple ci-dessous :
  - Si on passe l'objet « 1 » en paramètre, la variable « a_f » référence l'objet 1 qui est immuable. Puis a_f référence l'objet 2. Quand on quitte la fonction, la variable « a_f » est supprimée. L'objet « 2 » n'est plus référencé par aucune variable. Il sera supprimé par un garbage collector.
  - Si on passe une variable « a » en paramètre qui référence un objet immuable, ici l'objet 3, alors, la variable « a_f » du paramètre de la fonction va référencer le même objet : c'est une référence partagée. « a_f » référence l'objet 3 qui est immuable, puis l'objet 4.
  - Quand on quitte la fonction, la variable « a_f » est supprimée. L'objet « 4 » n'est plus référencé par aucune variable. Il sera supprimé par un garbage collector.
  - La variable « a » continue de référencer l'objet « 3 ».

```
def inc(a_f):
    print(a_f, end = '')
    a_f = a_f+1
    print(f" -> {a_f}")

inc(1)    # 1 -> 2

a=3
inc(a)    # 3 -> 4
print(a) # 3
```

Paramètre mutable en référence

- ATTENTION : les explications ci-dessous sont compliquées à comprendre ! Accrochez-vous !
- Les principes sont les mêmes que précédemment.
- Quand on passe un paramètre à une fonction, on passe un objet (de l'espace des objets) qui correspond à ce paramètre. La variable qui correspond au paramètre dans la fonction va référencer cet objet. C'est pour cela qu'on dit que le paramètre est en référence.
- Dans l'exemple d'une fonction qui append 10 à une liste (ci-dessous) :
  - Si on passe l'objet [10, 20] en paramètre, la variable « list_f » référence cet objet qui est mutable. Si on y ajoute l'objet 10 à la liste, on peut l'afficher dans la fonction. Quand on quitte la fonction, la variable est supprimée. L'objet n'est plus référencé par aucune variable. Il sera supprimé par un garbage collector.
  - Si on passe une variable « l » en paramètre qui référence un objet mutable, ici la liste [30, 40, 50], la variable « list_f » du paramètre de la fonction va référencer le même objet : c'est une référence partagée. « list_f » référence l'objet [30, 40, 50], qui est mutable. On y ajoute l'objet 10. La variable « listt_f » référence alors l'objet [30, 40, 50, 10], comme la variable « a » par effet de bord.
  - Quand on quitte la fonction, la variable « l_f » est supprimée. Mais l'objet qu'elle référence est toujours référencée par la variable « l »

```
Def app10(list_f):
   print(str(list_f),end='')
   list_f.append(10)
   print(f" -> {list_f}")

app10([10, 20])   # [10,20] -> [10,20,10]
l=[50, 60, 70]
app10(l)          # [50,60,70] -> [50,60,70,10]
print(l)          # [50,60,70,10]
```

Exemple : la fontion sort(liste)

- La fonction list.sort(liste) permet de trier une liste passée en paramètre.

```
Help(list.sort)
```

- The sort is in-place (i.e. the list itself is modified).

Shallow copy pour éviter la modification du paramètre

- Pour éviter la modification de la liste de départ, il faut faire une shallow-copy.
- On reprend l'exemple de la fonction qui append 10 à une liste : app10(liste)
- On peut passer en paramètre une shallow-copy :

```
app10(l[:])           # [50,60,70] -> [50,60,70,10]
print(l)              # [50,60,70]
```

- Si on veut récupérer la liste modifiée dans la fonction, travailler sur une shallow-copy et la retourner

```
def app10(l_f):
    new_l=l_f[:]              # shallow copy
    print(str(new_l),end='')
    new_l.append(10)
    print(f" -> {new_l}")
    return new_l              # return de la shallow copy

app10([10, 20])    # [10,20] -> [10,20,10]
l=[50, 60, 70]
new_l= app10(l)    # [50,60,70] -> [50,60,70,10]
print(l)           # [50,60,70]
print(new_l)       # [50,60,70,10]
```

Utilisation d'une fonction non définie

- On peut écrire une fonction qui utilise une fonction qui n'est pas définie.
- Ca posera un problème uniquement à l'exécution

```
def f1():
   return f2()
# l'interpréteur est d'accord

f1() # ca va planter : NameError: name 'f2' is not defined
```

Première approche du polymorphisme

```
def f(a, b):
   print(f"{a} et {b}")
   return a + b

f(1, 2) #
f(1.1, 2.2)
f("coucou", "python")
f([1,2], [3,4])
```

- La fonction est polymorphe puisqu'elle peut fonctionner avec des types différents : des entiers, des float, des string et des listes.
- C'est pratique pour factoriser le code.
- Ce sera aussi pratique pour faire évoluer le code plus facilement.

4.2 - if, elif, else

Détails

- if, elif, else, <expression>, <bloc>

```
if <expression>:
   <bloc 1>
elif:
   <bloc 2>
elif:
   <bloc 3>
else:
   <bloc 4>
```

- elif et else sont facultatifs.
- Un <bloc> contient des instructions.
- Une <expression> est une formule de calcul qui est évaluer. Son résultat est True ou False.
- Est Faux : False, 0, None, [], {}, (), ‘’
- Est Vrai : Tout le reste : 18, ‘coucou’, [1,2,3]

Opérateurs booléens

- >, >=, <, <=, !=, == exemple : a==5
- In exemple : a in [1, 2, 3]
- and, or, not exemple : a==5 and not b

Pas de switch

- Il n’y a pas de switch en python.
- Basiquement, les elif joue le rôle du switch. On peut aussi trouver d’autres solutions plus élégantes selon les situations.

4.3 - while, continue, break

while <expression>

- Tant que expression est True, on répète la boucle

```
a=list(range(1,10))
while a:
    a.pop()
    print(a)
```

- Continue permet de passer au suivant sans faire ce qui suit

```
a=list(range(1,10))
while a:
    a.pop()
    if len(a)%2:
        continue
    print(a)
```

- Break permet de quitter la boucle. C'est nécessaire avec les boucles sans fin : while True

```
while True: # boucle sans fin
    s=input('Quelle est votre question ? ')
    if 'aucune' in s:
        break
    print('Très intéressant, mais encore ?', end=' ')
```

- Petit jeu basique avec un boucle sans fin et des tests

```
while True:
   s=input('quelle est votre question ? ')
   if s.startswith('bonjour'): # starts whith
      print('Bonjour, comment allez-vous ? ')
   elif s=='bien':
      print('tant mieux !')
   elif 'java' in s.lower():
      print('je ne parle pas le java')
   elif 'bye' == s:
      break
   else:
      print('je n\'ai pas compris')
```

  - Notez les différentes façons de vérifier la présence d'un mot dans une chaine : ==, in, startswith.

4.4 : portée et durée de vie des variables

Durée de vie

- Une variable existe à partir du moment où elle a été créée.
- Une variable créée dans une fonction meurt avec la fin de l'exécution de la fonction.
- Une variable créée en dehors de toute fonction meurt avec le restart de l'interpréteur python.

Portée des variables : local et global

- Une variable créée dans une fonction est une variable locale.
- Une variable créée en dehors de toute fonction est une variable globale.
- Une variables est visible par toutes les fonctions qui interviennent après sa création.

Recherche de la valeur d'une variable : remontée LEGB

- Pour savoir ce que vaut une variable on recherche :
  - L : Si elle est définie localement
  - E : Si elle est définie dans une fonction appelante de la pile des fonctions appelantes (Englobante).
  - G : Si elle est définie globalement directement dans le programme et en dehors de toute fonction
  - B : Si elle est définie par le module builtins

Module builtins

- Le module builtins contient tout ce qui est prédéfinie dans Python : les types, les fonctions, les classes.
- https://docs.python.org/fr/3/library/builtins.html#module-builtins

```
import builtins
dir(builtins)
help(buitins.oct)
```

- Quand on utilise une fonction builtins, comme print() par exemple, elle est cherchée en dernière instance dans le module builtins. Mais elle pourrait être redéfinie à différents niveaux.
- On peut toujours y accéder en préfixant son accès par le nom de son module :

```
import builtins
a=0
print=5
print(a) # erreur, TypeError: 'int' object is not callable
builtins.print(a)      # 0
builtins.print(print)  # 5
```

4.5 : global et non local

global

- Les variables globales sont accessibles en lecture dans les fonctions mais on ne peut pas les modifier.
- Pour modifier une variable globale dans une fonction, on commence par déclarer la variable globale dans la fonction avec le mot clé « global ». Les modifications qui sont faites seront visibles en dehors de la fonction.
- Par principe, il vaut mieux toujours éviter les variables globales.
- Exemple 1 : on affiche une variable globale dans une fonction

```
x = 5
def f():
    print(x)    # x est la variable globale
f()
print(x)
```

- Exemple 2 : on essaye de modifier une variable globale dans une fonction

```
x = 5
def f():
    print(x) # Erreur: x est locale et n'a pas de valeur
    x=10
f()
print(x)
```

- Exemple 3 : on essaye de modifier une variable globale dans une fonction

```
x = 5
def f():
    global x     # x est explicement déclaré comme le x global
    print(x)
    x=x+10
f()
print(x)
```

- Exemple 4 : on modifie la globale sans globale
  - On va mieux nommer les variables et la fonction. On va passer un paramètre à la fonction.

```
monX=5
def add_10(x):
    return x+10
monX=add_10(monX)
print(monX)
```

- Exemple 5 : on modifie la globale sans globale en définissant une classe : c'est le plus propre.
  - On va mieux définir une classe qui permet de créer une variable et de l'incrémenter.

```
class MonX :
    def __init__(self,x):
        self.x=x
    def add_10(self):
        self.x=self.x+10
    def __str__(self):      # pour le print
      return str(self.x)

monX=MonX(5)
monX.add_10()
print(monX)
```

non local

- Le mot clé « global » permet de modifier les variables globales.
- Mais on peut aussi vouloir modifier une variable locale d'une fonction englobante.
- Dans ce cas, on lieu de la déclarer « global » on la déclarera « non locale », mais les principes sont les mêmes que pour une « global », et c'est à éviter.

```
x = 5
def f():
    y=10
    def g():
        global x
        x=x+1
        nonlocal y
        y=y+1
    g()
    print(y)

f()
print(x)
```

4.6 : paramètres et arguments d'une fonction

Vocabulaire

- Quand on définit une fonction, les variables qu'on passe entre parenthèse sont les « paramètres de la fonction ». On parle aussi de paramètres formels.

```
def f(a, b, c):
    print(a*b+c)
```

- Quand on utilise une fonction, les expressions qu'on passe entre parenthèses (souvent des variables ou des valeurs « en dur ») sont les arguments de la fonction. On parle aussi de paramètres d'appel.

```
n=10
f(n, 5, 3*n+4)
```

- Il y a plusieurs manières de définir les paramètres d'une fonction et de passer des arguments à une fonction.

Arguments nommés

- Passage classique d'arguments : dans l'ordre de définition des paramètres.
- Passage d'arguments nommés : on nomme les arguments et on peut les placer dans n'importe quel ordre

```
f(c=3*n+4, b=5, a=n)
```

- Ca peut être pratique quand le nom des paramètres est explicite. Par exemple, avec une fonction qui initialise des valeurs pour un dictionnaire dont chque clé est explicite et correspond au nom des paramètres :

```
def personne(nom, prenom, tel):
    return{ 'nom':nom, 'prenom':prenom, 'tel':tel }

print( str( personne( tel='0606',nom='van Rossum',prenom='') ) )
```

Paramètre optionnel : en dernier

- Un paramètre optionnel est un paramètre qui a une valeur par défaut. Il est toujours en fin de liste des paramètres.

```
def personne(nom, prenom, tel='?'):
    return{ 'nom':nom, 'prenom':prenom, 'tel':tel }

print( str( personne('van Rossum','Guido') ) )
print( str( personne('van Rossum','Guido', '0606') ) )
```

Paramètre en forme étoile : tuple au choix, *t

- On peut passer une série d'arguments qui seront rangés dans un tuple.

```
def f(*t):     # on va passer une liste d'arguments
    print(t)
    print(t[0])

f(3)                          # (3,) : tuple à 1 élément – t[0]:3
f('Guido')                    # ('Guido',) - t[0]:'Guido'
f( [1, 2, 3] )                # ([1, 2, 3],) - t[0]: [1, 2, 3]
f(1, 2, 3)                      # (1, 2, 3)
f('van Rossum','Guido', '0606') # ('van Rossum','Guido', '0606')
f('Guido', 999, [1, 2, 3] )     # ('Guido', 999, [1, 2, 3] )
```

Paramètre en forme double étoile : dictionnaire au choix, **d

- On peut passer une série d'arguments qui seront rangés dans un dictionnaire. Il faut donc nommer les arguments pour avoir les clés du dictionnaire.

```
def f(**d):     # on va passer une liste d'arguments
    print(d)
    # print(d[0])

f(nom='van Rossum', prenom='Guido', tel='0606')
# {'nom': 'van Rossum', 'prenom': 'Guido', 'tel': '0606'}
# 'van Rossum'
```

Argument en forme étoile : passer une liste en argument, *l

- On peut passer une liste en argument : ses éléments viennent prendre la places des paramètres.
- Il faut qu'il y ait autant de paramètres dans la fonction que d'éléments dans la liste.

```
def f(a, b):
    print(a, b)

l=[1, 2]
f(*l)               # 1 2
```

Argument en forme double étoile : passer un dictionnaire en argument, **d

- On peut passer un dictionnaire en argument : ses values viennent prendre la places des paramètres.
- Il faut qu'il y ait autant de paramètres dans la fonction que values dans le dictionnaire.

```
d={'a':1, 'b':2}
f(**d)               # 1 2
```

Un usage pratique : sep et end du print

- Dans print, on peut ajouter « sep » et « end »

```
print(1, 2, sep=';', end='. ')  # 1;2.>>>
```

- On peut définir le couple « sep » et « end » dans un dictionnaire et le passer en ** dans un print :

```
sepend={'sep':';', 'end':'.'}
print(1, 2, **sepend)           # 1;2.>>>
```

Assert

- En Python un assert est une aide au débugage qui vérifie des conditions. Si la condition n'est pas vérifiée alors une AssertionError est soulevée avec, si besoin, un message d'erreur.
- Contrairement au « if », les « assert » ne servent pas à signaler une erreur mais à stopper le programme dans le cas où ça arriverait. Une sortie par un « assert » signifie que le programme doit être modifié : il y a un bug. Les « assert » ne doivent donc pas faire partie du code final livré. C'est un code intermédiaire pour le développeur.

https://docs.python.org/3/reference/simple_stmts.html#the-assert-statement

```
assert expression1, expression2
```

équivaut à :

```
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

51 - Itérateur – Itérable – Lamba expression – Compréhension de liste, de set, de dict

51.1 - Itérateur

Itérateur, itérable, méthode iter( )

- Un **itérable** est un objet **parcourable avec un itérateur**.
  - Toutes les séquences des itérables.
  - Les fichiers sont des itérables.
  - La fonction builtins **iterateur = iter(iterable)** permet de retourner un objet itérateur spécifique à chaque type de séquence. On peut aussi écrire iterable.__iter()__ (c'est équivalent)
- **Un itérateur** est un objet qui définit un **protocole d'itération**.
  - Un itérateur ne peut être **parcouru qu'une seule fois**.
  - La méthode **next(iterateur)** retourne l'élément en cours, jusqu'au dernier. Ensuite, elle retourne une exception : StopIteration. On peut aussi écrire iterateur.__next__()

```
l = [1, 2, 3]
for i in l:
   print(str(i))


print([i for i in l])        # [1, 2, 3]

iterateur=iter(l)
print(next(iterateur))       # 1
print(next(iterateur))       # 2
print(iterateur.__next__()) # 3
print(iterateur.__next__()) # StopIteration

iterateur=l.__iter__()       # ou iterateur=iter(l)
while True:
   try:
      n=next(iterateur)
      print(n)               # 1 puis 2 puis 3
   except StopIteration:
      break                  # pas de message quand on quitte
```

Utilité

- Créer un itérateur est très peu couteux. C'est une technique d'optimisation.
- Par exemple, la méthode builtins « zip(l1, 12, l3) permet de faire une liste de tuples en prenant les éléments 1 par 1 dans chaque liste. Mais z est un itérateur, pas une liste de tuples. On parcourt z comme un itérateur : avec un boucle for ou une compréhension.

```
l1=[11, 12, 13]
l2=[21, 22]
l3=[31, 32, 33]
z=zip(l1, l2, l3)
for i in z:
   print(i)

for i in z:
   print(i)  # z est vide : c'est un itérateur
```

Principe python : itérateur plutôt que structure de données

- On manipule des itérateurs plutôt que des listes qui prennent de la place.
- De façon générale, on des itérateurs plutôt que structures de données (listes, dictionnaires, set, etc.) qui prennent de la place.
- Ca a du sens quand on travaille sur les listes de grande taille.

51.2 - Expression lambda – variable fonction

Fonction comme paramètre d'une fonction

- On peut passer une fonction en paramètre d'une fonction.
- On peut mettre une fonction dans une variable. La variable devient une fonction.

```
def f_de_x(f) :
   for x in range(10):
      print( f"f({x})={f(x)}" )

def carre(x):
   return x**2-1

f_de_x(carre)

c=carre
type(c)   # <class 'function'>
f_de_x(c)
```

Expression lambda

- La programmation fonctionnelle consiste à écrire des expressions lambda ou fonction lambda.
- Une fonction lambda est une expression
- Cette expression peut être stockée dans une variable
- Syntaxe : lambda paramètre : expression retournée
- C'est une autre façon de définir des fonctions.

```
carre = lambda x : x**2-1
print( carre(10) )
for x in range(10):
   print( f"f({x})={carre(x)}" )

exemple = lambda x, y : x*y-1
print( exemple(2,3) )
```

map et filter

- map() applique une fonction à chaque élément d'un itérable et retourne un itérable.

```
m = map ( carre, range(10) )
print(list(m))
print(list(m))  # [] : m est vide : l'itérateur est parcouru
```

- filter() applique un filtre à chaque élément d'un itérable et retourne un itérable.

```
f = filter ( lambda x:x%2==0, range(10) )
for x in f :
   print(x)

print(list(f))  # [] : f est vide : l'itérateur est parcouru
```

compréhension de liste

- Fonction lambda et filter, ça rappelle l'écriture en compréhension.
- En python moderne, on va privilégier cette écriture en compréhension.
- La différence importante est qu'il ne s'agit alors pas d'un itérateur mais bien d'une liste.

```
f= [ x for x in range(10) if x%2==0]
for x in f :
   print(x)

print(f)  # f n'est pas vide : c'est une liste
```

51.3 – Compréhesion de liste, de dictionnaire, de set

Syntaxe intuitive

- La syntaxe des compréhensions est simples et intuitives, c'est pour cela qu'elle rencontre un grand succès.

```
multi3 = [ k for k in range(100) if k%3==0 ]
print(multi3)
```

- On peut écrire des codes très courts tout en restant lisibles grâce à cette syntaxe :
  - Exemple : on veut extraire les prénoms qui commencent par a est les mettre en minuscule.
  - On récupère les prénoms si la première lettre vaut a et on met le résultat en minuscules :

```
pns = ['ana', 'eve', 'ALICE', 'Anne', 'bob']
pns_a = [ p.lower() for p in pns if p.lower().startswith('a') ]
```

compréhension d'ensemble

- Un ensemble n'a pas de doublon.
- Si notre liste de départ a des doublons et qu'on veut les supprimer, on peut faire :

```
pns = ['ana', 'eve', 'ALICE', 'Anne', 'bob']
pns.extend(pns) # extend rajoute pns à pns : tout est duppliqué
pns_a = [ p.lower() for p in pns if p.lower().startswith('a') ]
pns_a = set(pns_a) # pns_a est un set sans doubloon
# pns_a vaut {'anne', 'alice', 'ana'}
```

- On peut faire directement :

```
pns_a = { p.lower() for p in pns if p.lower().startswith('a') }
```

  - C'est une comprehension qui produit un set : une compréhension d'ensemble.
- Remarque :

```
pns = {'ana', 'eve', 'ALICE', 'Anne', 'bob'}
pns_a = [ p.lower() for p in pns if p.lower().startswith('a') ]
```

  - C'est une compréhension de liste (pns_a est une liste) mais qui part d'un ensemble (pns est un ensemble).

compréhension de dictionnaire

- On peut aisément produire un dictionnaire à partir d'une liste

```
pns= ['ana', 'eve', 'ALICE', 'Anne', 'bob']
pns_a= { p.lower():20 for p in pns if p.lower().startswith('a') }
pns_a  # {'anne': 20, 'alice': 20, 'ana': 20) }
```

- On aussi partir d'un dictionnaire et produire un dictionnaire :

```
pers = {('ana', 20), ('EVE', 30), ('bob', 40)}
ages = dict(ages)
pers_u40 = {p.lower():a for p, a in ages.items() if a < 40}
pers_u40  # {'ana': 20, 'eve': 30}
```

compréhension de liste de dictionnaires – équivalent SQL

- On peut faire des requêtes type SQL avec des compréhension de liste de dictionnaires

```
pers= [
   {'nom': 'eve',    'age': 20},
   {'nom': 'ALICE',  'age': 30},
   {'nom': 'Anne',   'age': 25},
   {'nom': 'bob',    'age': 40}
]
# REQUETE 1
# les personnes dont le noms commencent par a et dont l'age <40
# avec leurs noms en minuscule
pns_a= [
   {'nom':p['nom'].lower(), 'age':p['age']}
   for p in pers
   if p['nom'].lower().startswith('a') and p['age']<40
]
pns_a
# [
#    {'nom': 'alice', 'age': 30},
#    {'nom': 'anne',  'age': 25}
# ]
#
# REQUETE 2
# somme des ages
# des personnes dont le noms commencent par a et dont l'age <40
sumAge= sum([
   p['age']
   for p in pers
   if p['nom'].lower().startswith('a') and p['age']<40
])
sumAge # 55
```

- La function « sum » est une fonction builtins. On trouve aussi les fonctions max et min. Pour la moyenne, il suffit de diviser sum par len.
- Version compacte :

```
pers= [ {'nom': 'eve',    'age': 20}, {'nom': 'ALICE',  'age':
30}, {'nom': 'Anne',   'age': 25}, {'nom': 'bob',    'age': 40} ]

pns_a= [ {'nom':p['nom'].lower(), 'age':p['age']}
         for p in pers
         if p['nom'].lower().startswith('a') and p['age']<40 ]
print(pns_a)

requete = [ p['age']
            for p in pers
            if p['nom'].lower().startswith('a') and p['age']<40 ]

moyAge= sum(requete)/len(requete)
print(moyAge)
```

51.41 – Expressions génératrices

Compréhension de liste

- Une compréhension de liste retourne une liste qui occupe la mémoire.

```
carre = [x**2 for x in range(1000)]
type(carre)  # <class 'list'>
len(carre)   # 1000
sum(carre)   # 332833500
```

Expressions génératrice

- Une expression génératrice est une compréhension qui retourne un itérateur et pas une liste. L'itérateur n'occupe pas la mémoire.
- Syntaxiquement, il suffit de mettre l'expression entre parenthèses.
- Ca retourne un itérateur et pas une liste. Il ne sera parcourable qu'une seule fois.

```
carre = (x**2 for x in range(1000))
type(carre)  # <class 'generator'>
sum(carre)   # 332833500
sum(carre)   # 0 : l'itérateur a déjà été parcouru
```

  - Carre n'est utilisable qu'une fois. Mais l'instruction d'expression génératrice ne coûte quasiment rien. Si on a de nouveau besoin de carré, il n'y a qu'à le réécrire.
  - Pour éviter de le réécrire, on pourra écrire des fonctions génératrices.

Chaîner des expressions génératrices

```
carre = (x**2 for x in range(100))
palin = (x for x in carre if str(x)==str(x)[::-1])
type(palin)  # <class 'generator'>
list(palin)  # [0, 1, 4, 9, 121, 484, 676]
```

- Dans le code ci-dessus, aucune liste temporaire n'est enregistrée.
- On peut aussi tout écrire d'un coup, mais c'est moins lisible :

```
palin=( x**2 for x in range(100) if str(x**2)==str(x**2)[::-1] )
list(palin) # [0, 1, 4, 9, 121, 484, 676]
```

Parcours de fichier

- Un fichier sera un itérateur et le parcours d'un fichier fonctionnera de la même manière.

```
with open("file/fichier.txt", "r", encoding='utf-8') as fichier:
   filtre = (ligne.split() for ligne in fichier)
   lignes=list(filtre)

with open("file/fichier.txt", "r", encoding='utf-8') as fichier:
   filtre = (int(x.split()[0]) for x in fichier)
   filtre = (x for x in filtre if x <10)
   lesXsingleton=list(filtre)
# On chaine les expressions, ça simplifie l'écriture.
# En une seule expression, il faudrait faire :
# un if sur int(x.split()[0]) au lieu d'un simple x en 2 lignes.

with open("file/fichier.txt", "r", encoding='utf-8') as fichier:
   filtre = (int(x) for ligne in fichier
                    for x in ligne.split()
                    if int(x) <10)
   lesX2=list(filtre)
```

51.42 – Fonctions génératrices

Principes

- Les fonction génératrice fonctionnent comme les fonctions sauf qu'elles n'utilisent plus le return : elle « yield ». On fait autant de « yield » qu'on veut : le « yield » permet de fournir des données à un itérateur.
- L'appel à la fonction retourne un itérateur qu'on peut exploiter comme tel : avec un next(), en le transformant en liste, dans un for in.

Exemple de base

```
def gen(x):
   if type(x)==int:
      yield x
   elif type(x)==list:
      for i in x:
         yield i

g=gen(10)
type(g) # <class 'generator'>
next(g) # 10
next(g) # StopIteration

g=gen([1, 2])
list(g) # [1, 2])
list(g) # []

g=gen([1, 2])
next(g) # 1
next(g) # 2
next(g) # StopIteration

g=gen([1, 2])
for i in g:
   print(i)
# 1
# 2
```

Exemple 2

```
def gen(x):
   yield x
   x=x+1
   yield x

g=gen(10)
type(g) # <class 'generator'>
list(g) # [10, 11]
list(g) # []

g=gen(10)
next(g) # 10
next(g) # 11
next(g) # StopIteration
```

Exemple 3

```
def listeCarres(a,b):
   for i in range(a, b):
      yield i**2

g=listeCarres(1,10)
type(g) # <class 'generator'>
list(g) # [10, 11]
list(g) # []

g=listeCarres(5,7)
next(g) # 25
next(g) # 26
next(g) # StopIteration
```

Exemple 4

- Fonction génératrice pour un palindrome

```
def palin(it):
   for i in it:
      if ( isinstance(i,(int, str)) and
           str(i)==str(i)[::-1] ):
         yield(i)
```

  - La fonction est classique sauf qu'elle « yield » les résultats à la volée.

- Utilisation de la fonction génératrice avec un range()

```
# on passe le range de départ
p=palin(range(10,50))
list(p) # [11, 22, 33, 44]
```

  - On peut obtenir le même résultat avec un code complet « en dur » :

```
palin=( x for x in range(10, 50)
        if str(x)==str(x)[::-1] )
list(palin) # [11, 22, 33, 44]
```

- Utilisation de la fonction génératrice avec une liste « en dur » :

```
p=palin([1, 12, 22, 'a', 'ab ', 'aba'])
list(p) # [1, 22, 'a', 'aba']
```

  - On peut obtenir le même résultat avec un code complet « en dur » :

```
palin=( x for x in [1, 12, 22, 'a', 'ab ', 'aba']
        if str(x)==str(x)[::-1] )
list(palin) # [1, 22, 'a', 'aba']
```

- Utilisation de la fonction génératrice avec expression génératrice

```
p=palin(x**2 for x in range(100)))
list(p) # [0, 1, 4, 9, 121, 484, 676]
```

  - On peut obtenir le même résultat avec un code complet « en dur » :

```
palin=( x**2 for x in range(100) if str(x**2)==str(x**2)[::-1] )
list(palin) # [0, 1, 4, 9, 121, 484, 676]
```

52 - Importation – Espace de nommage (théorique)

52.1 – Espace de nommage

Principes

- Un module est un fichier .py.
- Quand on importe ce fichier avec l'instruction import, on accède aux fonctions et aux variables qu'il définit.
- Les variables d'un module sont isolées des variables d'un autre module. Cette isolation permet d'éviter des erreurs.
- En Python, chaque module (donc fichier) correspond à un espace de nommage. Ainsi toute variable est définie dans un espace de nommage.
- En C, il n'y a pas d'espace de nommage. En Java, les classes définissent les espaces de nommage.
- De façon plus générale, en python, les modules, les fonctions, les classes et les instances définissent des espaces de nommage. Un espace de nommage regroupe un ensemble de variable appartenant à un objet.
- Un espace de nommage peut être créée et être détruit pendant le cycle de vie du programme. C'est le cas pour les fonctions : chaque fonction a son propre espace de nommage créé à l'appel de la fonction et détruit dès que la fonction retourne son résultat.

Exemple de base

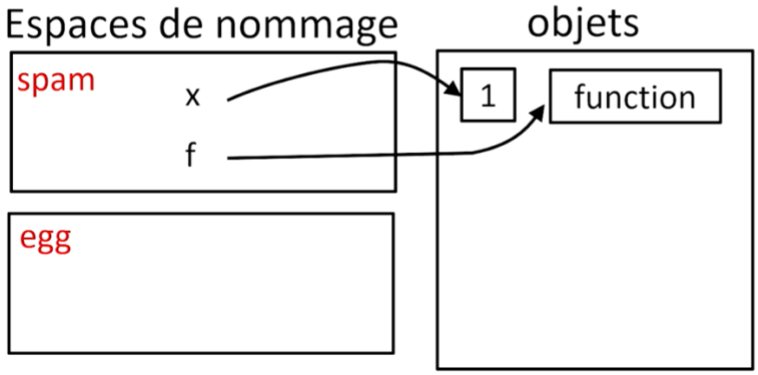
- Un module 1 (fichier egg.py) va importer un autre module 2 (fichier spam.py) : import spam
- On a 2 variables et 2 fonctions de même nom dans les modules : par exemple x et f().
- Si le fichier egg veut accéder à la variable x ou à la fonction f() du module spam, il suffit d'écrire : spam.x et spam.f()
- La notation objet.attribut permet d'accéder à attribut dans l'espace de nommage de objet.

spam.py

```
x = 1
def f():
    print(x)
```

egg.py

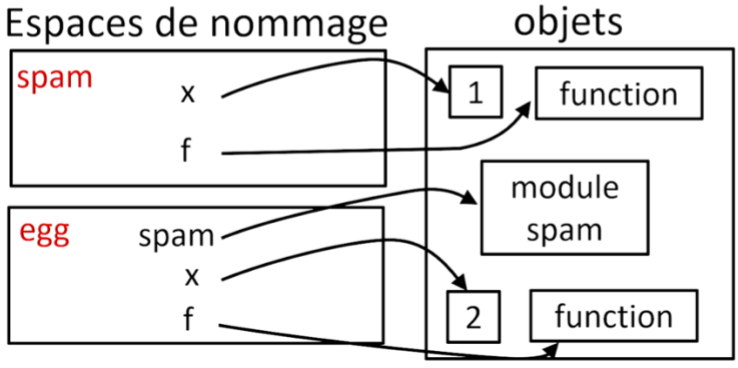
```
import spam
x = 2
def f():
    print(x)
f()
spam.f()
print(spam.x)
```



- Étapes suivantes :

spam.py
```
x = 1
def f():
    print(x)
```

egg.py
```
import spam
x = 2
def f():
    print(x)
f()
spam.f()
print(spam.x)
```



Objet partagé

- 2 variables dans 2 espaces de nommage peuvent référencer le même objet. La modification d'une variable entrainera donc la modification de l'autre par effet de bord.

Implémentation des espaces de nommage

- Techniquement, un espace de nommage est géré avec un dictionnaire : la clé correspond au nom de la variable, la valeur correspond à la référence vers l'objet.

52.23 – Précisions sur l'importation

Principes de l'importation d'un module

- import os : ça importe le fichier import.py. En général, on importe du Python (parfois du C).
- os est désormais une variable qui référence l'objet module : elle vaut le nom absolu du fichier os.py. Un print(os) permet de voir ça.
- L'interpréteur va chercher os.py d'abord dans le répertoire courant, ensuite l'interpréteur cherche dans un répertoire défini dans la variable système PYTHONPATH qui est une clé du dictionnaire os.environ pas forcément définie. Ensuite, l'interpréteur va chercher dans la librairie standard qui a été installé à l'installation.
- Le chemin de recherche des modules est définie dans la variable sys.path. C'est une liste qu'on peut modifier en cours de programme.
- Quand le ficher de code est importé, il est précompilé par l'interpréteur : ça crée un fichier .pyp contenant du « bytecode » placé dans un répertoire __pycache__ dans le répertoire courant.
- Pour finir, l'interpréteur évalue le bytecode pour générer l'objet module. Les blocs de code de fonction ne seront évalués que lors des appels de la fonction, l'objet fonction ayant été créé à l'interprétation du code de définition de la fonction.
- Un objet module n'est importé qu'une seule fois, même s'il y a plusieurs importation du même module. Il y aura ensuite des références partagées vers cet objet. Cet objet étant mutable, les modifications seront partagées par toutes les références partagées.

Affichage des nom d'un espace de nommage : dir(), vars()

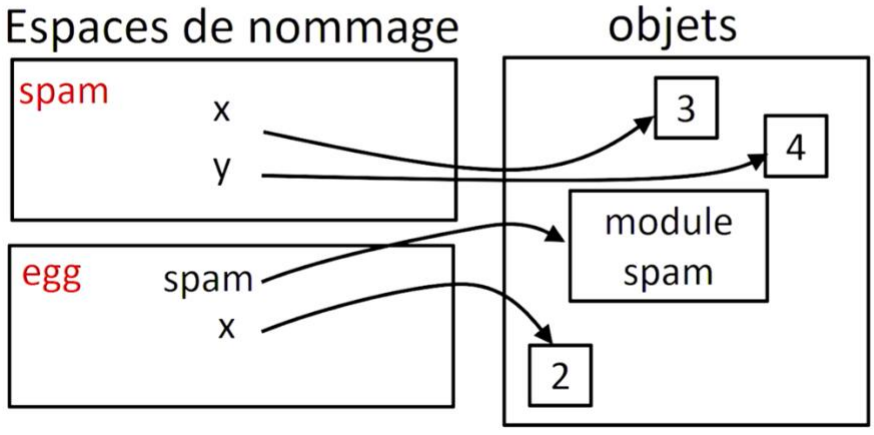
- dir(), dir(module) : pour afficher les nom de l'espace de nommage courant ou de l'espace de nommage d'un module particulier.
- vars() : permettra d'afficher ne plus la valeur des variables.

Création de variables dans un espace de nommage importé

- On peut créer de nouvelles variables dans un espace de nommage importé.

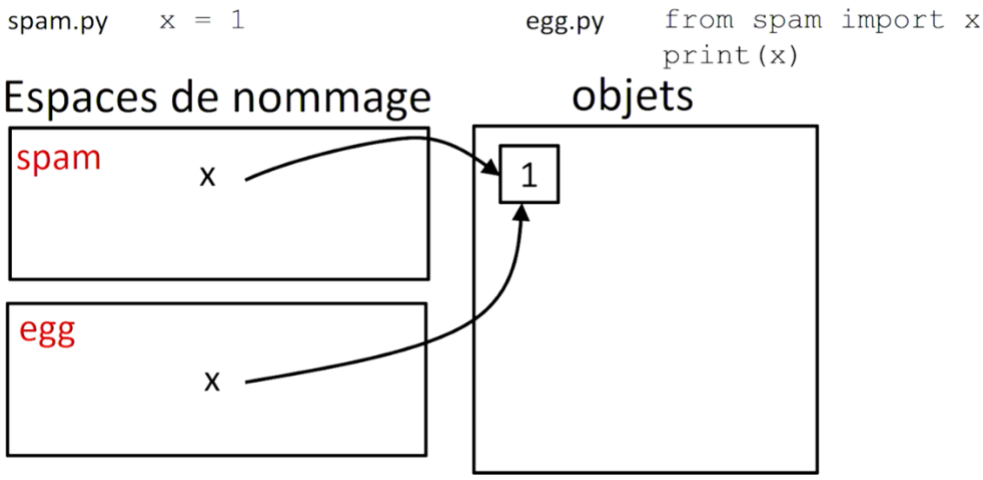
spam.py
```
x = 1
```

egg.py
```
import spam
x = 2
spam.x = 3
spam.y = 4
print(spam.x)
print(spam.y)
```



From module import attribut

- Autre technique d'importation : from spam import x
- Dans ce cas, on n'accède pas au module spam. On va seulement se doter d'une variable x qui référencera l'objet référencé par la variable x du module spam. On ne peut pas modifier la valeur de x pour spam.



6 – Classe

6.1 - Introduction

Notions importante

- Paradigme de programmation.
- Classe, méthode, instance, méthode spéciale, héritage.
- Créer ses propres ses propres exceptions, ses propres littérateurs, ses propres contextes managers.

Tout est objet

- En Python, tout est objet : entier, liste, chaîne de caractères, fonctions, les modules : tout !
- Un objet est caractérisé par son type.
- Une classe permet de définir son propre type. Les classes sont aussi des objets ! Une classe est définie dans un module. Quand le module est importé, l'objet classe est créé.

Usine à instance

- Une classe est une « usine à instance » : un service qui permet de créer de nouveaux objets avec leurs caractéristiques et leurs services (leurs méthodes).
- L'instance hérite de la classe : elle hérite de tous les attributs et de toutes les méthodes définies dans la classe.
- En langage naturel, on peut dire que l'instance « est une » sa classe. Par exemple >>> phrase=Phrase('Coucou, c'est moi') , phrase est un objet, Phrase sa classe. « phrase » est une « Phrase ».
- L'objet et la classe ont leur propre espace de nommage. On cherche les informations dans l'arbre d'héritage des espaces de nommage, de l'instance à la classe, en remontant les héritages entre classes.

Exemple 1

```
class Phrase:
    maPhrase = 'test de class en python'

p1 = Phrase()    # un objet p1
p2 = Phrase()
p1 is p2 # false # ni le même objet
p1 == p2 # false # ni la même valeur

print(Phrase.__dict__)  # mappingproxy({ ...'ma_phrase': 'test de
class en python' ... })
print(p1.__dict__)      # {} rien de spécifique, hérite de Phrase
print(p2.__dict__)      # {}

print(vars(Phrase))     # plus pratique que .__dict__
print(vars(p1))         #
print(vars(p2))         #

print(p1.maPhrase)      # vaut 'test de class en python'
```

- vars(Classe), vars(objet) : donne des informations sur la classe ou l'objet.
- Les caractéristiques sont dans un dictionnaire.
- '__module__': '__main__' : la classe et l'objet sont dans le module « main ».

Exemple 2 : class et objet mutables

- On peut ajouter et modifier les attributs des classes et des objets à tout moment (pas de principe d'encapsulation).

```
Phrase.ma_phrase = 'modif de class en python'
print(p1.maPhrase)    # vaut 'modif de class en python'
print(p1.maPhrase)    # comme Phrase.mots
print(p2.maPhrase)    # idem
```

- On peut modifier au niveau de la classe : ça vaut pour tous les objets

```
Phrase.mots = Phrase.ma_phrase.split()
print(vars(Phrase))   # on trouve mots
print(vars(p1))       # on ne trouve pas mots
print(Phrase.mots)    # ['modif', 'de', 'class', 'en', 'python']
print(p1.mots)        # comme Phrase.mots
print(p2.mots)        # idem
```

- On peut modifier au niveau de l'objet : ça ne vaut que pour l'objets

```
p1.nbMots = len(p1.mots)
print(Phrase.nbMots)  # AttributeError: pas d'attribut 'nbMots'
print(p1.nbMots)      # 5
print(p2.nbMots)      # AttributeError: pas d'attribut 'nbMots'
```

Exemple 3 : implémentation des méthodes

- Une méthode est une fonction définie dans une classe qui peut travailler sur les attributs de l'instance.

```
class Phrase:
    def initPhrase(self, maPhrase):
        self._maPhrase = maPhrase
    def nbLettres(self):
        return len(self._maPhrase)

print(vars(Phrase))  # on y trouve initPhrase mais pas ma_phrase
                     # maPhrase est de niveau instance

maPhrase = 'test de méthodes en python'
p1 = Phrase()
p1.initPhrase(maPhrase)

print(p1.maPhrase)      # 'test de méthodes en python'
print(p1.nbLettres())   # 26

Phrase.initPhrase(p1, 'coucou') # on peut faire ça : on évite !
print(p1.maPhrase)              # 'coucou'

print(Phrase.maPhrase)  # AttributeError, pas d'attribut maPhrase
```

- A noter l'usage : Phrase.initPhrase(p1, 'coucou'). On ne fait jamais ça. Ca montre qu'on peut accéder à la méthode à partir de la classe (en tant qu'objet), en passant tous ses paramètres : le self.
- Par contre, quand on accède à initPhrase( ) à partir d'une instance, il n'y a plus de paramètre self. Le lien est fait automatiquement.

6.2 – Méthodes spéciales

Présentation

- On part de la classe Phrase en y ajoutant l'attribut « mots » dans la méthode initPhrase
- On voudrait pouvoir utiliser les usages habituels du Python :

```
class Phrase:
    def initPhrase(self, maPhrase):
        self.maPhrase = maPhrase
        self.mots=maPhrase.split()
    def nbLettres():
        return len(self.maPhrase)

p = Phrase()
p.initPhrase('test de méthodes spéciales')
p.mots # ['test', 'de', 'méthodes', 'spéciales']

# on voudrait :
len(p)
'de' in p
p[2]
p1 + p2
```

  - Comment faire ça ?
  - Les méthodes spéciales permettent ça.

Méthodes spéciales

- Les méthodes spéciales commencent toutes par __ et finissent toutes par __
- Elles sont appelées automatiquement sur les mots-clés, les opérateurs, les fonctions builtin du Python.
  - __len__() : len
  - __str__() : print
  - __contains__() : in

```
class Phrase:
    def initPhrase(self, maPhrase):
        self.maPhrase = maPhrase
        self.mots=maPhrase.split()
    def __len__(self):
        return len(self.mots)
    def __str__(self):
        return str(self.mots)
    def __contains__(self, mot):
        return mot in self.mots
    def nbLettres():
        return len(self.maPhrase)

p = Phrase()
p.initPhrase('test de méthodes spéciales')
p.nbLettres() # 26
len(p)        # 4
print(p)      # ['test', 'de', 'méthodes', 'spéciales']
'test' in p   # True
```

  - A noter : __str__ retourne une str

Méthode spéciale spéciale : le constructeur

- Le constructeur en Python, c'est la méthode spéciale __init__().
- __init__ va permettre de passer des paramètres au moment de l'instanciation. (Formellement, il y a un constructeur par défaut, sans paramètre qui fera appel à la méthode spéciale __init__()

```
class Phrase:
    def __init__(self, maPhrase):
        self.maPhrase = maPhrase
        self.mots=maPhrase.split()
    def __len__(self):
        return len(self.mots)
    def __str__(self):
        return str(self.mots)
    def __contains__(self, mot):
        return mot in self.mots
    def nbLettres(self):
        return len(self.maPhrase)

p = Phrase('test de méthodes spéciales')
len(p)        # 4
print(p)      # ['test', 'de', 'méthodes', 'spéciales']
'test' in p   # True
```

Toutes les méthodes spéciales

- Il y a plein de méthodes spéciales
  - __add__ : + >>> def __add__(self, other):
  - Les apparentés : __mul__, __sub__, __div__, __and__, etc.
  - Il y en a d'autres.
- On peut les utilser pour définir ce qu'on appelle un « protocole » : notamment le « **protocole d'itération** » et le « **protocole context manager** ».

getter et setter classiques

- Chaque attribut peut être récupéré avec un getter et modifié avec un setter

```
class Personne:
    test='coucou'
    def __init__(self, nom, mail):
        self.nom = nom
        self.mail = mail
    def getNom(self):
        return self.nom
    def setNom(self, nom):
        self.nom=nom
    def getMail(self):
        return self.mail
    def setMail(self, mail):
        self.mail=mail

p=Personne('egg', 'egg@spam.com')

p.getNom()          # p.nom
p.getMail()
p.setNom('bacon')   # p.nom='bacon'
p.getNom()
```

6.34 – Héritage

Présentation : relation est un

- La relation d'héritage est une relation « est un » : elle traduit une inclusion ensembliste.
- Un objet hérite de sa classe : l'objet est inclus dans l'ensemble correspondant à sa classe. L'objet est un « sa classe ». Par exemple : p, p1, p2 sont des « Phrase ». Ces objets appartiennent à l'ensemble correspondant à la classe.
- Un objet qui hérite de sa classe a accès aux attributs et aux méthodes qui sont définies dans sa classe.
- C'est la même chose pour l'héritage entre classe : si la classe A() hérite de la classe B(), c'est que A est un B. A pourra accéder aux attributs et aux méthodes de B. Donc si « a » est une instance de A, c'est que « a » est un « A » et aussi un « B ». « a » accède aux attributs et aux méthodes de A et de B.
- Il suffit d'écrire : >>>class A(B) pour dire que la classe A hérite de la classe B.
- La classe A est dite : classe enfant.
- La classe B est dite : classe parent.

Usage

- Quand **A hérite de B,** on peut dire que **A spécifie B** : cela veut dire que A est B mais avec des **spécifications supplémentaires** : des attributs et des méthodes en plus ou des redéfinitions des méthodes qu'on trouvait dans B.
- L'héritage est donc utilisé pour partir d'une classe existante et n'avoir qu'à rajouter les spécifications qui nous intéressent, sachant qu'on doit avoir, autant que possible, la **relation logique « enfant est parent »**.
- Autrement dit, l'héritage est utilisé pour **créer une classe à partir d'une autre en modifiant un peu son comportement**.
- **Exemple** : on veut **une phrase avec une liste de mots tout en minuscule**. On va créer une classe PhaseLower qui va hériter de Phrase.

```
class PhraseLower(Phrase):
    def __init__(self, maPhrase):
        Phrase.__init__(self, maPhrase)
        self.motsLower = [m.lower() for m in self.mots]
    def __contains__(self, mot):
        return mot.lower in self.motsLower
    def getPhrase(self):
        return self.maPhrase.lower()

p = PhraseLower('Test DE L'Héritage')
p.nbLettres()               # 18
p.mots                      # ['Test', 'DE', 'L'Héritage']
p.motsLower                 # ['test', 'de', 'l'héritage']
isinstance(p, PhraseLower)  # True
isinstance(p, Phrase)       # True
'De' in p                   # True

p2 = Phrase('Test DE L'Héritage')
'De' in p2                  # False

p.getPhrase()
```

- Dans le__init__ on appelle le __init__ de la classe parente.
- On accède aux attributs de la classe parente avec le self
- La méthode __contains__ redéfinit la méthode __contains__ de la classe parente.

Arbre héritage et héritage multiple

- Toute classe peut hériter d'une autre classe. On peut avoir A qui hérite de B qui hérite de C. Dans ce cas, on a un arbre d'héritage.
- Toute classe peut aussi hériter de plusieurs classe. C'est un mécanisme qu'il faut utiliser avec prudence (il n'existe pas en Java). Dans ce cas, il faut préciser l'ordre d'héritage pour savoir qui est prioritaire en cas de conflit d'héritage (il y a conflit d'héritage quand 2 classes parentes d'un héritage multiple propose le même attribut ou la même méthode.
- Pour toute instance, on peut écrire : >>> C.__class__ : cela donne
- Pour tout classe C, on peut écrire : >>> C.__bases__ : cela donne la classe parente.
- Pour tout classe C, on peut écrire : C.mro() : cela toutes les classes classes parentes dans l'ordre, du parent le plus proche au parent le plus éloigné. Le plus éloigné est toujours la classe « object ».
- MRO : Methode Resolution Order.
- Le MRO est logique en cas d'héritage simple.
- Le MRO est plus subtil en cas d'héritage multiple. L'ordre dépend de l'ordre de définition des héritage : >>>A(B, C) : ici A hérite de B et C, d'abord de B, ensuite de C et enfin de « object ».

6.5 – Variable, attribut, instance, objet

Variable, attribut, mécanisme de résolution

- **Variable** : un nom référence directement un **objet**. Le mécanisme pour faire le lien est la **résolution lexicale** (ou liaison lexicale).
- **Attribut** : un nom référence un objet en utilisant la **notation « objet.attribut »**. Le mécanisme pour faire le lien avec l'objet correspondant à l'attribut est la **résolution d'attribut**.
- Les mécanismes de **résolution** ont pour but de savoir dans **quel espace de nommage** le nom a été défini.

Attribut

- Pour un **attribut**, l'espace de nommage est **celui de l'objet**. L'objet peut être un module, une classe ou une instance (attention, objet ne veut pas dire instance).
- Si c'est un module, on cherche l'attribut dans l'espace de nommage du module (les variables globales du module).
- Si c'est une classe ou une instance, on cherche l'attribut le long de l'arbre d'héritage.

Variable

- Une variable définie dans une **fonction**, une **classe** ou un **module** devient **locale** au lieu de définition, sauf si elle est déclaré global ou nonlocal.
- Les différentes déclaration de variables :
  - x=1 : variable x
  - def f(a) : variable a, paramètre de la fonction, variable f, la fonction
  - class A() : variable A
  - for i in maListe : i est une variable
  - import sys : sys est une variable
  - from math import sqrt : sqrt est une variable
  - etc.
- Dans quel espace de nommage je trouve la variable : **règle LEGB** (local, englobant, global, builtins).
- Attention dans les classes : une variable dans une méthode (donc pas un attribut, pas précédé par self) est résolue en sautant la classe : elle ne fait pas référence à un attribut de la classe. Elle fera référence au bloc dans lequel est défini la classe.

```
x = 1
class A():
 x = 2
 def f(self):
  print(self.x)    # 2
  print(A.x)       # 2
  print(x)         # 1

a=A()
a.f() # 2 2 1
```

6.6 – Classe productrice d'objet itérable

Rappels : Itérateur, itérable, méthode iter( )

- Un **itérable** est un objet qu'on peut itérer (on peut appliquer une boucle sur lui). Ce parcours de boucle se fait avec un objet itérateur.
- L'itérateur est produit avec la méthode iter() appliquée à l'itérable : **iterateur = iter(iterable)**.
- La méthode **next(iterateur)** retourne l'élément en cours, jusqu'au dernier. Ensuite, elle retourne une exception : StopIteration.

Définir un itérateur

- On part de la classe Phrase à minima.
- On veut en faire une classe productrice d'objet itérable. xploitable avec un next : le next parcourera les mots de la phrase en tant qu'itérateur.
- Il faut donc définir un itérateur qui corresponde aux mots.
- La méthode spéciale __iter__ va nous permettre de faire ça, avec un « yield » qui permet de fournir des données à un itérateur : on yield les mots un par un, et on a notre itérateur.
- De là, on peut faire un >>>it=iter(p) et next(it)
- On peut aussi faire un >>> for m in p: ou un >>> if 'classe' in p:
- Ca remplace la méthode __contains__

```
class Phrase:
    def __init__(self, maPhrase):
        self.maPhrase = maPhrase
        self.mots=maPhrase.split()
    def __iter__(self):
        for m in self.mots:
            yield m

p = Phrase('test classe itérateur')
it = iter(p)

next(it)        # 'test'
next(it)        # 'classe '
next(it)        # 'itérateur'
next(it)        # Erreur : StopIteration

for m in p:
    print(m)
# test
# classe
# itérateur

if 'classe' in p:
    print(1)
else:
    print(0)
# 1
```

6.7 – Créer ses exceptions personnalisées

Principes

- Ca permet d'avoir une exception qui correspond exactement à notre besoin.
- Une exception qu'on crée en tant que classe hérite de la classe Exception ou de ses sous-classes.
- En cas d'erreur on « raise » une exception avec des paramètres au choix. En général, un message d'erreur et éventuellement un code d'exception.
- C'est une technique simple pour documenter les comportements erronés de nos programmes.

Exemple : pas de phrase vide

- Dans la classe phrase, si on instancie une phrase vide, on aura une exception.

```
class Phrase:
    def __init__(self, maPhrase):
        if maPhrase.strip() == '' :
            raise PhraseVideError('Une phrase doit contenir du
texte', 99)
        self.maPhrase = maPhrase
        self.mots=maPhrase.split()
    def __str__(self):
        return str(self.mots)

class PhraseVideError(Exception):
    pass

try:
    p = Phrase('Phrase correcte')
except PhraseVideError as e:
    print(e.args)

print(p)
del p

try:
    p = Phrase('')
except PhraseVideError as e:
    print(e.args)
    print(e.args[0])
    print(e.args[1])

print(p)
```

6.8 – Context manager

Notion de context manager

- On a déjà vu les context manager avec les fichiers.
- Ils permettent de gérer automatiquement les erreurs et la fermeture des fichiers.
- Ils s'utilisent avec un « whith … as ».

Notion de finalisation

- Un « finally » dans un « try … except … finally » permet de finaliser des traitements, qu'on soit passé par le cas général (le try) ou par l'exception.
- La fermeture d'un fichier est un cas de finalisation.

Protocle de context manager

- Quand on écrit :

```
with expression as e :
   bloc d'instructions
```

  - « e » est un objet ContextManager.qui correspond au retour de l'expression mais qui va exécuter deux méthodes spéciales automatiquement :
    → la méthode __enter__() qui est exécutée au début du with
    → la méthode exit__() qui est exécutée à la sortie du with.

- On peut donc écrire :

```
with MaClasse() as e :
```

- Dans ce cas, on peut ajouter une méthode __enter__ et méthode exit__ pour gérer une entrée et une sortie du with.

Exemple

```
import time
class MonTimer:
    def __enter__(self):
        self.start=time.time() # heure de début
        print("__enter__")
        return self

    def __exit__(self, *args):
        self.stop=time.time()  # heure de sortie
        duree=self.stop-self.start
        print(f"__exit__ durée : {duree:.2f}s")
        return False           # pour stopper en cas d'erreur

    def __str__(self):         # print de temps intermédiaire
        duree = time.time()-self.start
        return f"durée en cours : {duree:.2f}s"

with MonTimer() as mt:
                                        # __enter__
    print("début du with")              # début du with
    sum(x for x in range(10_000_000))   # 49999995000000
    print(mt)                           # durée en cours : 0.57s
    sum(x for x in range(10_000_000))   # 49999995000000
    print("fin du with")                # fin du with
                                        # __exit__ : durée = 1.13s
```

6.9 – Attribut de classe - Méthode de classe – Méthode static - Décorateur

Notion d'attribut de classe

- On peut définir dans une classe des attributs qui sont accessibles via la classe et pas via l'instance.
- Si on considère la classe comme un ensemble, et l'instance comme un objet de l'ensemble, l'attribut de classe est une information qui s'applique à l'ensemble et pas à un objet en particulier.
- De la même façon, on peut définir des méthodes qui soient des méthodes de classe

Exemple classique d'attribut de classe : un compteur d'instances

- On veut pouvoir compter les instances d'une classe.
- On définit un attribut nbI dans la class qu'on initialise à 0.
- On met à jour cet attribut dans la méthode spéciale __init__ : on y accède en le préfixant par le nom de la class

```
class Test():
    nbI = 0
    def __init__(self) :
        Test.nbI += 1

t1=Test()
t2=Test()
Test.nbI # 2
t1.nbI   # 2
```

Méthode de classe

- On va se doter d'une méthode qui nous donne un accès à la valeur de l'attribut.

```
class Test():
    nbI = 0
    def __init__(self) :
        Test.nbI += 1
    def getNbI():
        return Test.nbI

t1=Test()
t2=Test()
Test.getNbI() # 2
t1.getNbI()   # bug !!
```

⇒ On ne peut pas accéder à la méthode via une instance.

Méthode de classe accessible via une instance : décorateur @staticmethod

- Pour accéder à la méthode via une instance, il suffit d'ajouter le décorateur @staticmethod

```
class Test():
    nbI=0
    def __init__(self) :
        Test.nbI+=1
    @staticmethod
    def getNbI():
        return Test.nbI

t1=Test()
t2=Test()
Test.getNbI() # 2
t1.getNbI()   # 2 !! grace au décorateur @staticmethod
```

6.10 décorateur dataclass

- Python 3.7 simplifie la définition de classes dites "de données" : en utilisant le décorateur @dataclass. Il faut l'importer.
- Les attributs d'instance se déclare dans la classe avec la syntaxe : attribut : type
- On peut mettre des valeurs par défaut
- La méthode __init__ est prédéfinie
- La méthode __str__ est prédéfinie

```
from dataclasses import dataclass

@dataclass
class Personne:
    nom: str
    age: int
    mail: str=''  # valeur par défaut

p1=Personne('egg', 20, 'egg@spam.com')
p2=Personne('bacon', 25)
print(p1)  # Personne(nom='egg', age=20, mail='egg@spam.com')
print(p2)  # Personne(nom='bacon', age=25, mail='')
p1.mail    # 'egg@spam.com'
p2.age     # 25
```

7 – Quelques modules standards

- https://docs.python.org/fr/3/library/index.html

7.1 – Math

```
import math
help(math)
for i in dir(math)
   print(i)
math.pi
math.e
```

ceil(), floor(), trunc()

pow(), sqrt(), pi,

trigonométrie, log, exponentiel, etc.

7.2 – Time

https://docs.python.org/fr/3/library/time.html?highlight=time#module-time

Mesurer le temps de calcul

- La méthode la plus simple est de faire une mesure avant et après et de faire une soustraction.
- Si on crée de grosses variables, on a intérêt à les deleter avant de commencer le calcul.

```
import time
del l1, l2, l3 # on del pour toujours partir du même point

def f(n):
   return [x for x in range(n)]

debut = time.time()
l1=f(100_000)
print(time.time() - debut) # donne le temps en secondes

debut = time.time()
l2=f(1_000_000)
print(time.time() - debut) # donne le temps en secondes

debut = time.time()
l3=f(10_000_000)
print(time.time() - debut) # donne le temps en secondes
```

7.3 – OS

Module Python : os

- https://docs.python.org/fr/3/library/os.html

Doc Python

- https://docs.python.org/fr/3/howto/regex.html

Autre doc

- http://calmettes.arnaud.free.fr/python_linux.pdf
- https://python.doctor/page-expressions-regulieres-regular-python

os.system('commande')

- os.systm() permet de passer des commandes de l'OS.

```
import os
os.system('clear')  # effacer l'écran sous linux-mac
os.system('cls')    # effacer l'écran sous windows
os.getlogin()       # l'utilisateur connecté
```

7.4 – pathlib

parcours de l'arborescence des répertoires et des fichiers : pathlib

- ancien module : os.path (obsolète)
- nouveau module : pathlib
- pathlib offre une interface orienté objet en offrant un accès à la classe Path.
- Documentation -> ici

Associer un path à un fichier

```
nomFichier='test.py'
from pathlib import Path  # on importe la class Path
path = Path(nomFichier)   # on crée un objet relié à un fichier
path.exists()             # si le fichier n'existe pas,
                          # la méthode exists() retourne False
# on va créer notre fichier avec un context manager
with open(nom, 'w', encoding='utf-8') as output:
    output.write('0123456789\n')
path.exists()             # maintenant c'est True
print(path.name)          # nom du fichier
stat=path.stat()          # la méthode stat() retourne les
                          # métadonnées du fichier
print(stat.st_size)       # c'est la taille du fichier
print(stat.st_mtime)      # c'est la date de modification
                          # en secondes, depuis le 1/1/1970
from datetime import datetime
mtime = datetime.fromtimestamp(stat.st_mtime)
print(f"{mtime:%Y%M%D-%H:%M}")  # affichage propre !

del path                  # libère le lien
path.unlink()             # supprime le fichier
```

Associer un path à un dossier

```
from pathlib import Path     # importation de la class Path
path = Path('./data/')       # objet path lié au dossier data
files=dirpath.glob("*.json") # récupèration des fichiers du ls
for f in files :
    print(f)
```

7.5 – Test unitaires : unittest

Pour gérer les tests unitaires, on utilise le module « unittest » :
https://docs.python.org/fr/3.10/library/unittest.html

Tuto :
https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python/2235416-creez-des-tests-unitaires-avec-unittest

7.6 – Tkinter

Quid

Interface graphique en Python
https://docs.python.org/fr/3/library/tkinter.html?highlight=tkinter#module-tkinter

Doc

- La documentation de la bibliothèque permet de se former : https://docs.python.org/fr/3/library/tkinter.html
- Guide de survie : https://docs.python.org/fr/3/library/tkinter.html?highlight=tkinter#tkinter-life-preserver
- Autre tuto : https://gayerie.dev/docs/python/python3/tkinter.html

Installation

➢ ***Sur PC***

```
pip install tk
```

⇒ Test basique sur PC :

```
python -m tkinter
```

➢ ***Sur MAC***

```
Brew install python-tk
```

Code minimum:

```
from tkinter import *

root = Tk()

root.mainloop()
```

Code plus élaboré :

```
from tkinter import *
from tkinter import ttk
root = Tk()
frm = ttk.Frame(root, padding=10)
frm.grid()
ttk.Label(frm, text="Hello World!").grid(column=0, row=0)
ttk.Button(frm, text="Quit", command=root.destroy).grid(column=1,
row=0)
root.mainloop()
```

7.7 – Expression régulière

Module Python : re

- https://docs.python.org/fr/3/library/re.html

Doc Python

- https://docs.python.org/fr/3/howto/regex.html
- https://python.doctor/page-expressions-regulieres-regular-python

Autre doc

- Chaine de caractère 2.2

7.8 – Programmation asynchrone

- asyncio est une bibliothèque permettant de faire de la programmation asynchrone en utilisant la syntaxe async/await.
- La programmation asynchrone est surtout utile en programmation WEB pour la mise à jour asynchrone de la page HTML -> on retrouve async et await dans Nodes.js.
- https://docs.python.org/fr/3/library/asyncio.html
- https://docs.python.org/fr/3/library/asyncio-task.html

7.9 – Programmation parallèle et threads

threading
https://docs.python.org/fr/3/library/threading.html?highlight=thread#module-threading

https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python/2235545-faites-de-la-programmation-parallele-avec-threading

7.10 – Programmation fonctionnelle

https://docs.python.org/fr/3/howto/functional.html

7.11 – Mot de passe

https://docs.python.org/fr/3/library/getpass.html

7.etc.

8 - Data Sciences – Modules non standards

- https://pypi.org : c'est la plateforme qui distribue les bibliothèques non standards via l'outil pip ou pip3.

8.0 - Introduction

Triple compétences Data-Sciences

- **Datasciences** : triple compétences : **statistiques** (mathématiques), **programmation**, expertise domaine (**métier**)
- **Si double compétence :**
  - stat + métier = stat classiques
  - prog + métier = danger ! il manque le métier statistique !
  - prog + stat = danger ! il manque le métier d'application !
- **Exemple** : En machine learning, il y a bien sûr des stat et de la prog. Mais il aussi faut préparer ses données et choisir son algo d'apprentissage, ce qui passe par une compétence métier d'application.

Liste d'outils

- Les outils de base : notebook, Jupyter, Numpy, Pandas
- Python : 1994 (simple et généraliste) – Numpy : 2006 – Pandas : 2008 (spécialisés et plus complexe).
- Numpy : librairie de référence pour manipuler des tableaux en Python. Tableaux multidimensionnels. C'est très performant, particulièrement la vectorisation
- Pandas : librairie de référence pour ajouter des labels aux index des tableaux Numpy, et ainsi les rendre plus explicites. De plus, on trouve dans Pandas les opérations classiques de BD : regroupement (group by), jointure, pivot.
- Numpy et Pandas, c'est ce qui se fait de mieux actuellement. Mais ce n'est pas parfait. Il faut faire avec.
- Les notebooks sont l'environnement de choix de la communauté datascience car ils permettent de faire des « runnable papers » : des papiers exécutables ! Du texte formaté mélangé avec du code et avec lequel on peut interagir avec nos données.

Installation pip ou pip3

- pip3 install numpy
- pip3 install panda
- pip3 install seaborn
- etc.

Documentation

- Il y a beaucoup de ressources sur internet

➢ ***Jupyter, notebook (et numpy, et pandas)***

https://openclassrooms.com/fr/courses/4452741-decouvrez-les-librairies-python-pour-la-data-science/5560976-familiarisez-vous-avec-lecosysteme-python

➢ ***Chaine de calcul python – datasciences***

https://www.youtube.com/watch?v=zZkNOdBWgFQ

➢ ***Pandas***

http://www.python-simple.com/python-pandas/panda-intro.php

8.1 - Mise en bouche

Vitesse de calcul

```
import numpy as np

l = list(range(1000))
a = np.array(L)

[x**2 for x in L]     # 300 micro-sec
[x**2 for x in a]     # un peu plus rapide : 200
a**2                  # beaucoup plus rapide : 1,5
```

- a**2 : on met le tableau au carré. C'est de la « vectorisation ».

Titanic – panda – group by

- On peut récupérer le fichier titanic sur github : https://github.com/mwaskom/seaborn-data
- Le fichier csv directement -> ici. Il faut le copier-coller et l'enregistrer dans titanic.csv.

```
# pip install pandas matplotlib
import pandas as pd
import matplotlib.pyplot as plt

# Modèle : chargement des données
print("------------------------------------------------------------")
data = pd.read_csv('mpl_0_titanic.csv')
print("shape:", data.shape) # [1309, 14]
print("head:", data.head())
print("[]:", data[['pclass', 'survived', 'sex', 'age']])

# Modèle : création de colonnes calculées
# agePourcent ramène la moyenne des ages à un pourcentage est donne un graphique lisible
data["agePourcent"] = (data["age"] / 100).astype(float)

# Contrôleur : traitement des données
print("------------------------------------------------------------")
result1 = data.groupby(['sex', 'pclass']).agg(
    agePourcent_moyenne=('agePourcent', lambda x: x[data['survived'] == 1].mean()), # Moyenne de 'agePourcent' pour les
survivants seulement
    survived_moyenne=   ('survived', 'mean') # Moyenne de 'survived' (qui sera toujours 1 dans ce filtrage)
)

data['not_survived'] = 1 - data['survived']
result2 = data.groupby(['sex', 'pclass']).agg(
    agePourcent_moyenne=('agePourcent', lambda x: x[data['not_survived'] == 1].mean()), # Moyenne de 'agePourcent'
pour les survivants seulement
    not_survived_moyenne=   ('not_survived', 'mean') # Moyenne de 'not survived' (qui sera toujours 1 dans ce filtrage)
)

print("group by 1:\n", result1)
print("group by 2:\n", result2)
'''
group by 1:
               agePourcent_moyenne  survived_moyenne
sex    pclass
```

```
female 1                0.349390        0.968085
       2                0.280809        0.921053
       3                0.193298        0.500000
male   1                0.362480        0.368852
       2                0.160220        0.157407
       3                0.222742        0.135447
'''

print("-------------------------------------------------------")
result3 = data.groupby(['pclass']).agg(
    nb_survivants=      ('survived', 'sum'),
    nb_non_survivants=  ('survived', lambda x: (x == 0).sum()),
    taux_survivants=    ('survived', lambda x: round(x.sum() / len(x) * 100, 2)),
    taux_non_survivants=('survived', lambda x: round((x == 0).sum() / len(x) * 100, 2))
)
print("group by 3:\n", result3)

# Vue : affichage des résultats
print("-------------------------------------------------------")
print('graphique')

# Figure 1
plt.figure(1)  # Création de la figure 1
ax = result1.plot(kind='bar')  # Trace le graphique en barres et retourne l'objet Axes
ax.set_ylim(0, 1)# pour fixer les valeurs de l'axe des Y
plt.title("Figure 1: Graphique en barres - Moyenne d'age et de survivants par sexe et classe")
plt.xlabel("Sexe et Classe")
plt.ylabel("Moyenne")

# Figure 2
plt.figure(2)  # Création de la figure 1
ax = result2.plot(kind='bar')  # Trace le graphique en barres et retourne l'objet Axes
ax.set_ylim(0, 1) # pour fixer les valeurs de l'axe des Y
plt.title("Figure 2: Graphique en barres - Moyenne d'age et de non survivants par sexe et classe")
plt.xlabel("Sexe et Classe")
plt.ylabel("Moyenne")

# Figure 2
plt.figure(3)  # Création de la figure 2
result3.plot(kind='bar')
plt.title("Figure 2: Statistiques par classe")
```

```
plt.xlabel("Classe")
plt.ylabel("Valeurs")

plt.show()
```

- Explications détaillées : https://www.youtube.com/watch?v=zZkNOdBWgFQ

- Les print :

---

shape: (891, 15)

head:

survived pclass sex age sibsp parch fare embarked class who adult_male deck embark_town alive alone

0 0 3 male 22.0 1 0 7.2500 S Third man True NaN Southampton no False

1 1 1 female 38.0 1 0 71.2833 C First woman False C Cherbourg yes False

2 1 3 female 26.0 0 0 7.9250 S Third woman False NaN Southampton yes True

3 1 1 female 35.0 1 0 53.1000 S First woman False C Southampton yes False

4 0 3 male 35.0 0 0 8.0500 S Third man True NaN Southampton no True

[]: pclass survived sex age

0 3 0 male 22.0

1 1 1 female 38.0

2 3 1 female 26.0

3 1 1 female 35.0

4 3 0 male 35.0

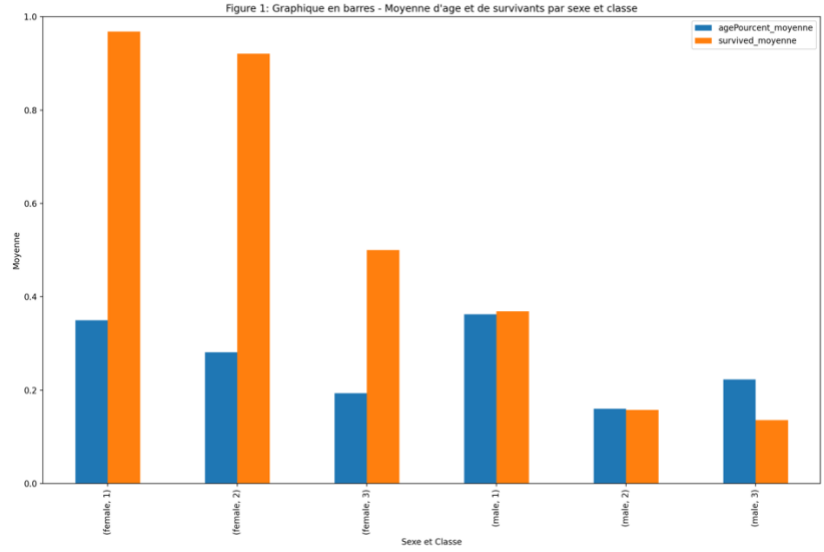
.. ..

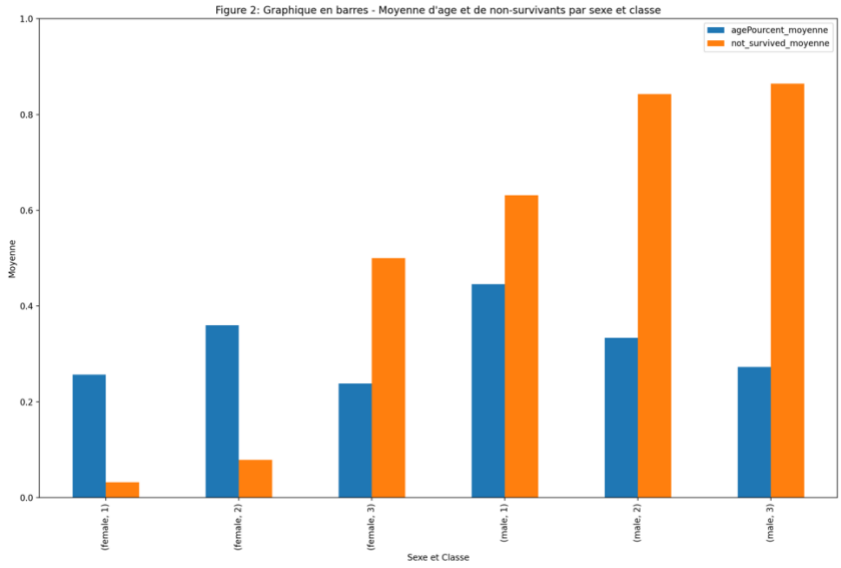
886 2 0 male 27.0

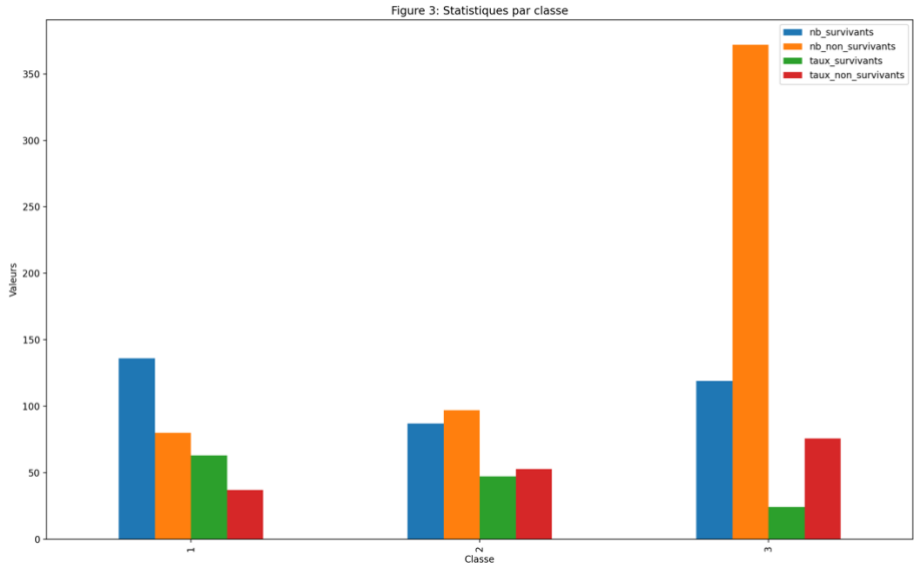
887 1 1 female 19.0

```
888      3        0  female   NaN
889      1        1    male  26.0
890      3        0    male  32.0

[891 rows x 4 columns]
----------------------------------------------------------
group by 1:
               agePourcent_moyenne  survived_moyenne
sex    pclass
female 1                 0.349390         0.968085
       2                 0.280809         0.921053
       3                 0.193298         0.500000
male   1                 0.362480         0.368852
       2                 0.160220         0.157407
       3                 0.222742         0.135447

group by 2:
               agePourcent_moyenne  not_survived_moyenne
sex    pclass
female 1                 0.256667             0.031915
       2                 0.360000             0.078947
       3                 0.238182             0.500000
male   1                 0.445820             0.631148
       2                 0.333690             0.842593
       3                 0.272558             0.864553

group by 3:
        nb_survivants  nb_non_survivants  taux_survivants  taux_non_survivants
pclass
1                 136                 80            62.96                37.04
2                  87                 97            47.28                52.72
3                 119                372            24.24                75.76
----------------------------------------------------------
```

Vue des graphiques Matplotlib







Titanic – pivot - pandas

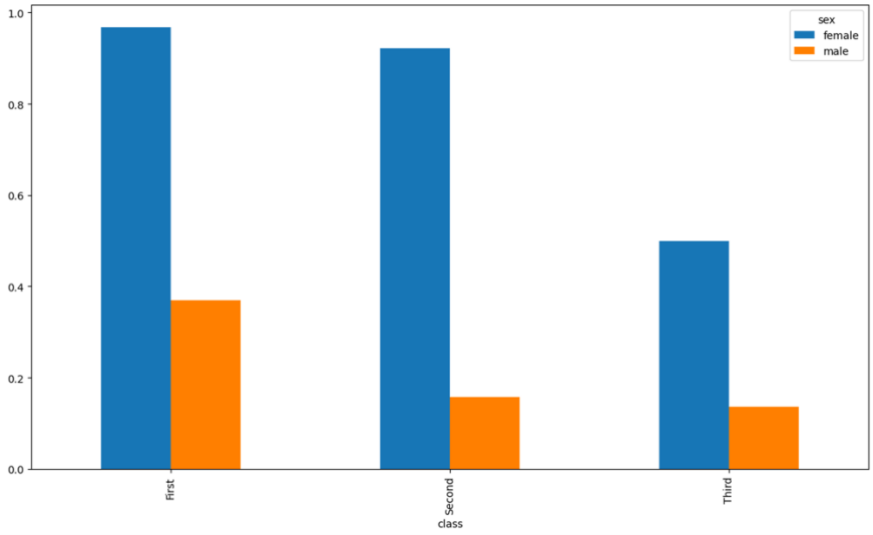
- La fonction pivot_table( ) permet de produire un tableau croisé (qu'on appelle pivot).
- A la suite du code précédent, on peut ajouter

```
##########################################################
# Pivot

res=data[['pclass', 'survived', 'sex']].pivot_table(index =
'pclass', columns = 'sex')
print(res)
'''
        survived
sex       female      male
pclass
1       0.968085  0.368852
2       0.921053  0.157407
3       0.500000  0.135447

96% des femmes de 1ère classe ont survécu, 13% des hommes de 3ème
classe ont survécu
'''
```

Titanic – pivot - seaburn

- Avec seaburn et la fonction load_dataset(), on récupère directement des données de github.
- Ici, on va récupérer le fichier titanic
- On a les mêmes résultats, mais avec une vraie fonction de pivot et pas un simple group by.
- Le code est structuré MVC

```
# pip install numpy, pandas, seaborn, matplotlib
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Modèle : chargement des données
titanic = sns.load_dataset('titanic') # csv de gitub
titanic.columns
titanic.head()

# Contrôleur : traitement des données
datas = titanic.pivot_table('survived',
                    aggfunc=np.mean,
                    index='class',
                    columns='sex')
print(datas)
'''
sex       female      male
class
First   0.968085  0.368852
Second  0.921053  0.157407
Third   0.500000  0.135447
'''

# Vue : affichage des résultats
print('graphique')
datas.plot(kind='bar')
plt.show()
```



Exemple de code complet :

```
import pandas as pd
data = pd.read_csv('titanic.csv')
print("-------------------------------------------------------")
print("shape:", data.shape) # [1309, 14]
print("head:", data.head())
print("[]:", data[['pclass', 'survived', 'sex', 'age']])
print("-------------------------------------------------------")
print()

data = pd.read_csv('titanic.csv')
data = data.drop([
'sibsp', 'parch', 'fare','embarked', 'class', 'who',
'adult_male', 'deck', 'embark_town', 'alive', 'alone'
], axis=1)

print("-------------------------------------------------------")
print("columns:", data.columns) # ['pclass', 'survived', 'sex', 'age']
print("describe:\n", data.describe())
print("-------------------------------------------------------")

print("-------------------------------------------------------")
result1 = data.groupby(['sex', 'pclass']).mean()

print("group by 1:\n", result1)
print("-------------------------------------------------------")
```

```
print("-------------------------------------------------------")
result2 = data.groupby(['pclass']).agg(
    nb_survivants=      ('survived', 'sum'),
    nb_non_survivants=  ('survived', lambda x: (x == 0).sum()),
    taux_survivants=    ('survived', lambda x: round(x.sum() / len(x) * 100, 2)),
    taux_non_survivants=('survived', lambda x: round((x == 0).sum() / len(x) * 100, 2))
)

print("group by 2:\n", result2)
print("-------------------------------------------------------")

print("-------------------------------------------------------")
print('graphique')
import matplotlib.pyplot as plt

# on retire l'age pour avoir des données lisibles dans le graphique
data = data.drop(['age',], axis=1)
result1 = data.groupby(['sex', 'pclass']).mean()

result1.plot(kind='bar')
plt.show()

result2.plot(kind='bar')
plt.show()
```

Résultats texte :

---

shape: (891, 15)

head: survived pclass sex age sibsp parch fare embarked class who adult_male deck embark_town alive alone

0 0 3 male 22.0 1 0 7.2500 S Third man True NaN Southampton no False

1 1 1 female 38.0 1 0 71.2833 C First woman False C Cherbourg yes False

2 1 3 female 26.0 0 0 7.9250 S Third woman False NaN Southampton yes True

```
3       1      1  female  35.0     1     0  53.1000       S  First  woman      False    C  Southampton
yes  False
4       0      3    male  35.0     0     0   8.0500       S  Third    man       True  NaN  Southampton
no   True
[]:     pclass  survived     sex   age
0        3        0    male  22.0
1        1        1  female  38.0
2        3        1  female  26.0
3        1        1  female  35.0
4        3        0    male  35.0
..     ...      ...     ...   ...
886      2        0    male  27.0
887      1        1  female  19.0
888      3        0  female   NaN
889      1        1    male  26.0
890      3        0    male  32.0

[891 rows x 4 columns]
------------------------------------------------------------


------------------------------------------------------------
columns: Index(['survived', 'pclass', 'sex', 'age'], dtype='object')
describe:
         survived      pclass         age
count  891.000000  891.000000  714.000000
mean     0.383838    2.308642   29.699118
std      0.486592    0.836071   14.526497
min      0.000000    1.000000    0.420000
25%      0.000000    2.000000   20.125000
50%      0.000000    3.000000   28.000000
75%      1.000000    3.000000   38.000000
max      1.000000    3.000000   80.000000
------------------------------------------------------------
```

---

group by 1:

		survived	age
sex	pclass		
female	1	0.968085	34.611765
	2	0.921053	28.722973
	3	0.500000	21.750000
male	1	0.368852	41.281386
	2	0.157407	30.740707
	3	0.135447	26.507589

---

group by 2:

	nb_survivants	nb_non_survivants	taux_survivants	taux_non_survivants
pclass				
1	136	80	62.96	37.04
2	87	97	47.28	52.72
3	119	372	24.24	75.76

---

graphique

Résultats graphiques :