

# **Programmation Python**

**(adoptez un serpent !)**

## **Partie 1- Bases et programmation objet**

**Pr ADG**

[dahirou.gueye@uadb.edu.sn](mailto:dahirou.gueye@uadb.edu.sn)

Université Alioune Diop de Bambe

Master SI/SR

Année académique: 2020-2021

# « Serpent scientifique et littéraire »



# Plan

1. Présentation du langage python
2. Calculs, variables, types, opérations et chaînes
3. La sélection conditionnelle if
4. Listes et boucles
5. Dictionnaires, Tuples et ensembles
6. Fichiers
7. Fonctions
8. Programmation orientée objet
9. Modules/packages
10. Exceptions
11. Interface graphique tKinter
12. Programmation réseau: socket
13. Base de données
14. Programmation web
15. Framework Django et APIs python
16. Traitement d'images avec python (analyse des données visuelles)

# Programmation orientée objet en Python

Chapitre 1 :

Présentation du langage Python

# Présentation

## ■ Python est un langage de programmation

- Orienté objet
  - ✗ Classes, objets, méthodes, héritage, etc.
  - ✗ Tout est objet en Python (Y compris les chaînes de caractères, les nombres, etc.)
- Multi-paradigme
  - ✗ Programmation objet mais aussi programmation impérative, fonctionnelle, etc.
- Interprété: ce qui signifie qu'il n'y a pas de phase de compilation qui traduit le programme en langage machine mais les instructions sont traitées au fur et à mesure de leur lecture par l'interprète.
- Dynamique: Tout peut être modifié au cours de l'exécution du programme
- Syntaxe souple
  - ✗ Faible typage, simplification commande, affectation multiple
- Multi-plateforme, libre et open-source (MacOS, Windows, Linux, Android)

## ■ Python optimise le temps du programmeur plutôt que le temps de calcul !

■ Si Python a gagné en popularité, c'est qu'il encourage une programmation intuitive reposant sur une syntaxe naturelle et des concepts fondamentaux puissants qui facilitent la programmation.

# Historique

- 1991 : Création de la première version publique de Python par Guido van Rossum au **CWI Centrum Wiskunde & Informatica** d'Amsterdam
- 1996 : sortie de la librairie NumPy
- 2001 : création de la Python Software Foundation (PSF) qui prend en charge le développement du langage
- 2008 : sortie de Python 2.6 et 3.0
- Python est resté peu connu pendant de longues années
- Croissance lente mais constante
- Depuis 2015, python est le langage le plus enseigné aux USA

# Avantages et inconvénients

## ■ Avantages :

- Gain de temps pour le programmeur
- Syntaxe très claire
- Nombreux modules
  - ✗ À rechercher et télécharger sur <http://pypi.python.org>
- Multi-plateforme

## ■ Inconvénients :

- Langage interprété donc peu rapide

# Python 2.x et Python 3.x

## ■ Deux versions de Python cohabitent encore :

- La version 2.7, stable et assez largement utilisé
- La version 3.x, dont le développement se poursuit

## ■ Ces deux versions sont très proches mais incompatibles

## ■ Nous étudierons la version 3.x

- Actuellement (juin 2021) on est à la version 3.9 qui est stable et la toute dernière est 3.9.5

## ■ Les deux versions sont téléchargeables sur

<https://www.python.org/downloads/>

# Que fait-on en Python ?

## ■ Python est utilisé pour :

- Scripting : Les scripts (petits programmes jetables)
- Le développement de prototype qui parfois, deviennent des applications finales
- Le développement de serveurs et de sites web dynamiques
  - Développement web avec des frameworks comme **Django** et **Flacon**
- Développement de jeux avec **Pygame**
- Applications mobiles avec des cadres comme **Kivy**. Android avec **SL4A**

# Que fait-on en Python ?

- Le développement d'application classique (desktop)
  - Python est également souvent utilisé par les admin système pour créer des tâches dites répétitives ou simplement de maintenance. D'ailleurs si vous voulez créer des applications java en codant en python, c'est possible grâce au projet Jython.
- Data Science – incluant le machine learning, l'analyse de données ainsi que la visualisation de données
- Chatbot avec le module **space**, **nltk**
- Réseaux de neurone avec le module **tensorflow**
- Programer arduino et Raspberry Pi en python
- WebRTC (Il permet à 2 navigateurs Web d'échanger des flux audio et vidéo en utilisant les modules **aiohttp** et **python-socketio**.)
- cloud computing (Concevoir et déployer rapidement des applications Python sur Google Cloud)

# Qui utilise python?

- **Google** (Guido van Rossum a travaillé pour Google de 2005 à 2012),
  - Python a été une partie importante de Google depuis le début et le reste à mesure que le système grandit et évolue.
- **Microsoft** avec
  - Microsoft store pour windows 10, visual studio,
  - Microsoft Azure: Générer et déployer vos applications Python dans le cloud, et aller plus loin avec l'IA et la science des données
- **Yahoo Groups** utilise Python "pour maintenir ses discussions de groupe".
- **YouTube** utilise Python pour produire des fonctionnalités maintenables en des temps records, avec un minimum de développeurs,
- **la Nasa** revendique l'utilisation de Python,
- **Etc.**

# Comment utiliser python?

Python présente la particularité de pouvoir être utilisé de plusieurs manières différentes :

- En mode interactif

- c'est à dire de manière à dialoguer avec lui directement depuis le clavier

```
Last login: Sat Jan 11 11:14:47 on console
You have new mail.
[macs-MBP:~ DAHIROU$ python3
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 05:52:31)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> ]
```

- Les trois caractères « supérieur à » constituent le signal d'invite, ou prompt principal, lequel vous indique que Python est prêt à exécuter une commande.
- Sous forme de programmes (scripts) et les sauvegarder sur disque.
  - (fichier complet avec l'extension .py) en utilisant par exemple un éditeur de fichier
- IDE (PyCharm, Visual studio code, TextWrangler, SublimeText, etc.)

# Installation de base

## ■ Windows

- Se rendre à la page <https://www.python.org/download> où vous trouverez un programme d'installation qui contient tout ce dont vous aurez besoin pour suivre le cours.
- Pour vérifier que vous êtes prêts, il vous faut lancer IDLE (quelque part dans le menu Démarrer) et vérifier le numéro de version.

## ■ MacOs

- Vous pouvez télécharger la version exécutable de python sur Mac OS à l'adresse suivante : <https://www.python.org/downloads/mac-osx/>
- Sachez aussi, si vous utilisez déjà MacPorts <https://www.macports.org>, que vous pouvez également utiliser cet outil pour installer, par exemple Python 3.6, avec la commande

`$ sudo port install python36`

# Installation de base

## ■ Fedora

```
sudo yum install python3-tools
```

## ■ Debian/Ubuntu

- Ici encore, Python-2.7 est sans doute déjà disponible. Procédez comme ci-dessus, voici un exemple recueilli dans un terminal sur une machine installée en Ubuntu-14.04/trusty :

```
$ python3
```

```
Python 3.6.2 (default, Jul 20 2017, 12:30:02)
```

```
[GCC 6.3.1 20161221 (Red Hat 6.3.1-1)] on linux
```

```
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> exit()
```

- Pour installer python

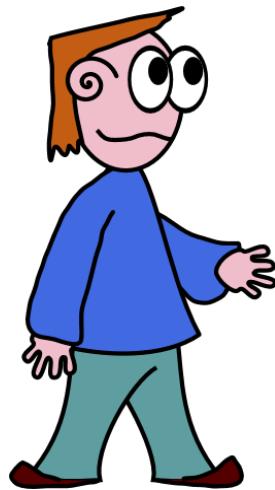
```
$ sudo apt-get install python3
```

- Pour installer idle

```
$ sudo apt-get install idle3
```

# Le génie de la programmation

Appelez-moi « ordinateur » !



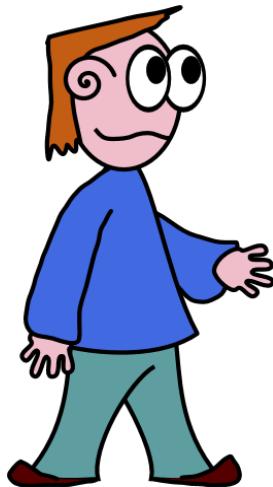
# Le génie de la programmation

Ordinateur, calcule  $5 + 2$  !

Ok, calcul effectué sans erreur.

Tu ne m'as pas donné le résultat ?

Tu ne m'as pas demandé de l'afficher.



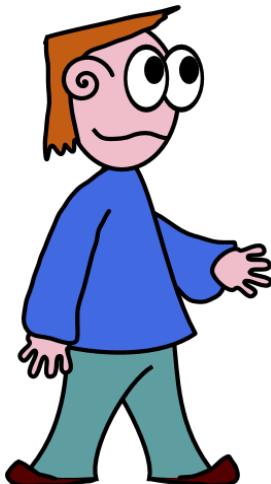
# Le génie de la programmation

Ordinateur, calcule  $5 + 2$  et affiche le résultat !

7.

Ajoute encore 3 à ce résultat !

Tu ne m'a pas demandé  
de le garder en mémoire.  
Je ne m'en souviens plus  
donc recommence de zéro !



# Le génie de la programmation

Ordinateur, calcule  $5 + 2$  et appelle ce résultat « r » !

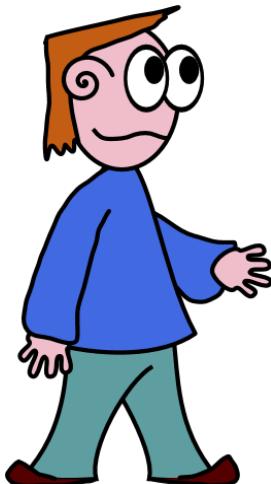
Affiche « r » !

7.

Ajoute 3 à « r » !

Affiche « r » !

10.



# Le génie de la programmation

`r = 5 + 2`

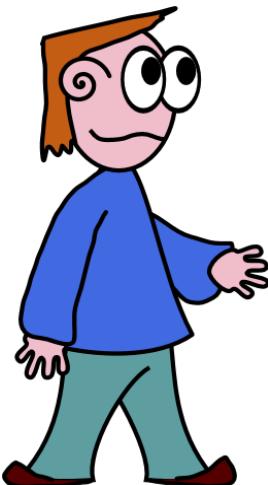
`print(r)`

7.

`r = r + 3`

`print(r)`

10.



*Ne jamais laisser le « génie »  
avoir le dernier mot !*



# Programmation orientée objet en Python

Chapitre 2:  
Syntaxe, Variables et Types de données

# Les commentaires

- Les commentaires ne sont pas exécutés par Python, ils sont destinés aux humains et non à l'ordinateur
- Ils facilitent la lecture et la compréhension du programme par d'autres personnes (ou par soi-même quelques années plus tard !)
- Commentaires simples (mono lignes)
  - Les commentaires commencent par un dièse #
  - Ils se prolongent jusqu'à la fin de la ligne
- # Ceci est mon premier commentaire !
- Commentaires longs
  - Il n'y a pas de commentaire long (type /\* \*/) à proprement parler en Python
    - ✗ Commenter chaque ligne
    - ✗ Utiliser une chaîne de caractères longue :

"""Ceci est une chaîne longue non utilisée,  
qui fait office de commentaire multi-ligne !  
"""

# Afficher à l'écran

■ La fonction `print()` permet d'écrire à l'écran (dans le terminal)

```
>>> print(5 + 2)
7
>>> print("Bonjour")
Bonjour
>>> print("Bonjour", "Master SI/SR")
Bonjour Master SI/SR
```

■ En ligne de commande, le `print()` peut être omis

```
>>> 5 + 2
7
```

# Aide

## ■ Plusieurs commandes permettent d'obtenir de l'aide en Python

- Aide sur une classe ou une méthode :

**help(list)**

**help(list.append)**

- Lister les méthodes d'un objet :

**dir(objet)**

## ■ Aide en ligne

- <https://docs.python.org/3/>
- Aussi disponible en téléchargement ou en paquet Linux

# Les types de données

Votre programme !

Modules Python

Tableaux :  
Module **numpy**  
(utilisation rare)

Classes et objets

Type de données de collection :

Séquences :

liste (**list**)

tuple (**tuple**)

ensemble (**set**)

ensemble fixe (**frozenset**)

Tableaux associatifs (**hash**) :

dictionnaire (**dict**)

dictionnaire ordonné

(**OrderedDict**)

Type de données de base :

Nombres :

booléen (**bool**)

entier (**int**)

flottant (**float**)

Textes :

chaîne de

caractères (**str**)

chaîne binaire (**bytes**)

Fonction

Fichier

(**file**)

# Nombres entiers

```
>>> print(1)  
1  
>>> print(1 + 2)  
3  
>>> print((3 + 4) * (5 + 8))  
91
```

■ Pas de limite aux nombres entiers en Python !

```
>>> print(123456789123456789123456789)  
123456789123456789123456789
```

# Nombres à virgule

```
>>> print(1.3)
```

```
1.3
```

```
>>> print(1.3e6)
```

```
1300000.0
```

- La précision du flottant Python correspond en fait à un double.

# Booléens

■ Deux valeurs possibles : vrai ou faux

✗ **True**

✗ **False**

# Variables

■ Une variable est un nom auquel on associe une valeur

✗ La valeur est souvent connu seulement à l'exécution du programme (par exemple c'est le résultat d'un calcul)

■ Sous Python, les noms de variables doivent en outre obéir à quelques règles simples :

✗ Un nom de variable est une séquence de lettres (a→z,A→Z) et de chiffres(0→9), qui doit toujours commencer par une lettre.

✗ Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que \$, #, @, etc. sont interdits, à l'exception du caractère \_ (souligné).

✗ Attention majuscules et minuscules sont différenciées !

✗ Il est important de trouver des noms parlants

# Variables

■ En plus de ces règles, il faut encore ajouter que vous ne pouvez pas utiliser comme nom de variables les 33 « mots réservés » ci-dessous

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

# Affectations simples de variables

■ On crée une variable en lui donnant une valeur, avec =

```
>>> age = 36
```

■ On obtient la valeur d'une variable

En donnant son nom

```
>>> age
```

36

Ou bien avec print

```
>>> print(age)
```

36

■ La valeur de la variable peut être modifiée avec =

```
>>> age = 37
```

```
>>> age = age + 1 # Anniversaire
```

37

■ del permet de supprimer une variable.

del age

# Affectations multiples

- Sous Python, on peut assigner une valeur à plusieurs variables simultanément

```
>>> x = y = 2020
[>>> x
2020
[>>> y
2020
>>> █
```

- On peut aussi effectuer des affectations parallèles à l'aide d'un seul opérateur :

```
[>>> note1, note2 = 13, 18.77
[>>> note1
13
[>>> note2
18.77
>>> █
```

# Types de données

■ Python peut manipuler différents types de données :

- ✗ des nombres entiers (integer en anglais, abrégé en int),
- ✗ des nombres à virgule (souvent appelé flottants ou float en anglais),
- ✗ des chaînes de caractères (string en anglais, abrégé en str)
- ✗ et des booléens (valeur vraie ou fausse, bool) :

- **age** = 25 # Entier
- **poids** = 35.5 # Flottant
- **nom** = "Dahirou Gueye" # Chaîne de caractères
- **enseignant** = TRUE # Booléen
- **etudiant** = FALSE # Booléen
- **telephone** = "00221 77 501 25 35" # Chaîne de caractères!

# Opérations sur les nombres

## ■ Opérations basique :

✗ Addition : +

✗ Soustraction : -

✗ Multiplication : \*

✗ Division : /

✗ Division entière : //

✗ Modulo (reste de la division) : %

✗ Puissance : \*\*

## ■ D'autres opérations sont définies dans le module **math**

# Priorité des opérations

■ Sous Python, les règles de priorité sont les mêmes que celles qui vous ont été enseignées au cours de mathématique. Vous pouvez les mémoriser à l'aide de l'acronyme PEMDAS :

- P pour parenthèses
- E pour exposants
- M et D pour multiplication et division, qui ont la même priorité
- A et S l'addition A et la soustraction S

# Chaînes de caractères

## ■ Les chaînes de caractère s'écrivent entre guillemets (simples ou doubles)

- > `Structure="Département d'Informatique TIC"`
- > `chaine_vide= " "`

## ■ Les chaînes de caractère sont en Unicode

- Cela permet de gérer toutes les langues du monde et des symboles

## ■ Pas de type caractère en Python

- On utilise une chaîne d'un seul caractère

## ■ Caractères spéciaux

- Retour à la ligne : "`\n`"
- Tabulation : "`\t`"
- Antislash : "`\\"`"

# Chaînes de caractères

## ■ Chaînes avec des guillemets à l'intérieur

- « Antislasher » les guillemets à l'intérieur de la chaîne : "**Il se nomme "Dahirou"**."
- Tripler les guillemets extérieurs : "'''**Il se nomme "Dahirou"**.''''
- Alterner guillemets doubles et simples : '**Il se nomme "Dahirou"**.'

# Chaînes de caractères

## Opérations sur les chaînes

Exemples avec `s = "Bonjour"`

- ✗ Obtenir la longueur d'une chaîne    `len(s)`      => 7  
    (= le nombre de caractères)
- ✗ Obtenir un caractère de la chaîne    `s[0]`      => "B"      `s[-1]` => "r"
- ✗ Obtenir une partie de la chaîne    `s[0:3]`      => "Bon"
- ✗ Concaténer deux chaînes                  `"Hi " + s`
- ✗ Rechercher si une chaîne est  
    inclus dans une autre                      `s.find("jour")=> 3 # Trouvé en position 3`  
  (-1 si pas trouvé)

# Chaînes de caractères

## ■ Opérations sur les chaînes

✗ Passer une chaîne en minuscule      `s.lower()`

ou en majuscule      `s.upper()`

✗ Remplacer un morceau d'une chaîne      `s.replace("Bonjour", "Good morning")`

✗ Demander à l'utilisateur de saisir une chaîne      `saisie = input("Entrez un nom : ")`

# Chaînes de caractères

■ Formatage de chaînes de caractère : inclure des variables dans une chaîne : %

```
>>> nom = "Gueye"  
>>> prenom = "Dahirou"  
>>> "Salut %s !" % prenom  
      Salut Dahirou !  
>>> "Bonjour %s %s !" % (prenom, nom)  
>>> "Taux de réussite : %s %" % 100  
      Taux de réussite : 100 %
```

■ Attention aux faux nombres !

```
telephone1 = "00221 33 823 37 38"
```

```
telephone2 = 00221338233738
```

- Quel est le type de donnée des deux variables ci-dessus ?
- Pourquoi a-t-on choisi ce type de donnée pour le premier cas ?

# Conversions

■ Il est souvent nécessaire de convertir un type de données vers un autre

- Les fonctions `int()`, `float()`, `bool()` et `str()` permettent de convertir une valeur vers un entier, un flottant et une chaîne de caractère
- `int("8") => 8`  
`str(8) => "8"`
- `input()` retourne toujours une chaîne de caractères ; il faut penser à la transformer en entier ou en flottant si l'on demande la saisie d'un nombre !

`age = int(input("Entrez votre âge : "))`

`poids = float(input("Entrez votre poids : "))`

# Programmation orientée objet en Python

Chapitre 3 :

Instructions de Contrôle: la sélection conditionnelle

# Sélection conditionnelle

- Si nous voulons pouvoir écrire des applications véritablement utiles, il nous faut des techniques permettant d'aiguiller le déroulement du programme dans différentes directions, en fonction des circonstances rencontrées
- Les conditions permettent d'exécuter des commandes seulement dans certaines situations

**if condition :**

*commande exécutée si la condition est vraie*

*commande exécutée si la condition est vraie...*

*suite du programme (exécuté que la condition soit vraie ou fausse)*

- Attention à l'indentation (= les espaces blancs en début de ligne)
- La condition est une comparaison utilisant un booléen ou un opérateur :

Inférieur à : <

Supérieur ou égal à : >=

Supérieur à : >

Égal à : ==

Inférieur ou égal à : <=

Different de : !=

# Sélection conditionnelle

- Le résultat doit être ceci:

```
>>> a=120
>>> if(a>100):
...     print("a dépassé la centaine")
...

```

- Frappez encore une fois <Enter>. Le programme s'exécute, et vous obtenez :

```
a dépassé la centaine
>>>
```

# Instructions successives

- Il est possible d'ajouter plusieurs conditions successives, et un bloc par défaut (*else = sinon*)

**if** *condition1*:

*commande exécutée si condition1 est vraie*

*commande exécutée si condition1 est vraie...*

**elif** *condition2*:

*commande exécutée si condition1 est fausse et condition2 est vraie...*

**else:**

*commande exécutée si condition1 et condition2 sont fausses...*

*suite du programme (exécutée que les conditions soient vraies ou fausses)*

# Instructions successives

## ■ Exemple1 de condition :

```
[>>> a=0
[>>> if a>0:
[...     print("a est positif")
[... elif a<0:
[...     print("a est négatif")
[... else:
[...     print("a est nul")
[... ]
```

## Exemple2 de condition :

```
age = int(input("veuillez saisir votre âge : "))
```

```
if age == 0 :
    print("Vous êtes un nouveau-né.")
elif age < 18 :
    print("Vous êtes un enfant.")
elif age >= 65 :
    print("Vous êtes une personne âgée.")
else :
    print("Vous êtes un adulte.")
```

```
print("L'année prochaine vous aurez ", age + 1, " ans.")
```

# Instructions imbriquées

- Il est possible d'imbriquer les unes dans les autres plusieurs instructions composées, de manière à réaliser des structures de décision complexes

## ■ Exemple1:

```
[>>> if embranchement == "vertébrés":  
[...     if classe == "mammifères":  
[...         if ordre == "carnivores":  
[...             if famille == "félins":  
[...                 print("c'est peut-être un chat")  
[...             print("c'est en tous cas un mammifère")  
[...         elif classe == "oiseaux":  
[...             print("c'est peut-être un canari")  
... print("la classification des animaux est complexe")
```

# Instructions imbriquées

## ■ Exemple2 :

```
age = int(input("veuillez saisir votre âge : "))
poids = float(input("veuillez saisir votre poids : "))

if age == 0 :
    print("Vous êtes un nouveau-né.")

    if poids > 10.0 :
        print("Je pense qu'il y a une erreur sur le poids !")
```

## ■ Les opérateurs logiques **not**, **and** et **or** permettent de combiner plusieurs conditions entre elles

✗ Respectivement NON, ET et OU logique

```
if (age < 18) or (age > 65) :
    print("Vous ne travaillez probablement pas.")
```

# Exercice 1 : les champignons

■ On souhaite réaliser un programme Python permettant d'identifier des champignons. Afin de rester simple, nous nous limiterons aux 4 d'espèces de champignon suivant :

- Les Ascomycètes (sans chapeau), comprenant :
  - ✗ Les morilles (avec pied)
  - ✗ Les truffes (sans pied)
- Les Basidiomycètes (avec chapeau), comprenant :
  - ✗ Les bolets (sans lamelle)
  - ✗ Les autres champignons (avec lamelles)

■ Le programme posera à l'utilisateur des questions de la forme  
« Le champignon a-t-il un chapeau (o/n) ? », via la fonction input()

# Exercice 1 : les champignons

■ On souhaite réaliser un programme Python permettant d'identifier des champignons. Afin de rester simple, nous nous limiterons aux 4 espèces de champignon suivant :

- Demander si le champignon a un chapeau :
  - ✗ S'il n'a pas de chapeau :
    - ⌚ C'est un Ascomycète
    - ⌚ Demander si le champignon a un pied
      - S'il a un pied, c'est une morille
      - S'il n'a pas de pied, c'est une truffe
  - ✗ S'il a un chapeau :
    - ⌚ C'est un Basidiomycète
    - ⌚ Demander si le champignon a des lamelles
      - S'il n'a pas de lamelles, c'est un bolet
      - S'il a des lamelles, c'est un autre champignon

# Exercice 1 : les champignons



# Exercice 1 : les champignons



# Programmation orientée objet en Python

Chapitre 4:  
Listes et Boucles

# Listes

- Une liste comme une collection d'éléments séparés par des virgules, l'ensemble étant enfermé dans des crochets

```
>>> jour = ['lundi', 'mardi', 'mercredi', 2020, 18.5, 'jeudi', 'vendredi']
>>> jour
['lundi', 'mardi', 'mercredi', 2020, 18.5, 'jeudi', 'vendredi']
>>> print (jour)
['lundi', 'mardi', 'mercredi', 2020, 18.5, 'jeudi', 'vendredi']
>>> █
```

- il est possible de changer les éléments individuels d'une liste :

```
>>> print (jour)
['lundi', 'mardi', 'mercredi', 2020, 18.5, 'jeudi', 'vendredi']
>>> jour[3] = jour[3] +37
>>> jour
['lundi', 'mardi', 'mercredi', 2057, 18.5, 'jeudi', 'vendredi']
>>> █
```

- Fonction len()

```
>>> print(len(jour))
```

# Listes

## ■ Fonction del

```
[>>> del(jour[4])
[>>> jour
['lundi', 'mardi', 'mercredi', 2057, 'jeudi', 'vendredi']
>>> ]
```

## ■ Fonction append()

```
[>>> jour.append('samedi')
[>>> jour
['lundi', 'mardi', 'mercredi', 2057, 'jeudi', 'vendredi', 'samedi']
>>> ]
```

# Accès aux éléments

## ■ Soit les listes

```
>>> nombres = [15, 48, 20, 35]
```

```
>>> animaux = ["éléphant", "girafe", "rhinocéros", "gazelle"]
```

```
>>> liste3 = [2000, "Bambey", 3.14, ["Ablaye", "Daouda", 1980]]
```

>>> print(nombres[2])	>>> print(nombres[1:3])	>>> print(nombres[2:3])	>>> print(nombres[2:])	>>> print(nombres[:2])	>>> print(nombres[-1])	>>> print(nombres[-2])
20	[48, 20]	[20]	[20, 35]	[15, 48]	35	20

## ■ Notation

- la notation liste[i:j] désigne les éléments i à j-1 de la liste
- la notation [i:] désigne les éléments i et suivants
- la notation [:i] désigne les éléments 0 à i-1

# Les listes sont modifiables

## ■ Modification par affectation directe

### ■ Exemple1:

```
>>> nombres[0] = 12
```

```
>>> nombres
```

```
[12, 48, 20, 35]
```

### ■ Exemple2:

```
>>> liste3[3][1] = "Dahirou"
```

```
>>> liste3
```

```
[2000, "Bambey", 3.14, ["Ablaye", "Dahirou", 1980]]
```

# Les listes sont modifiables

## ■ Les listes peuvent être modifiées

- Ajouter un élément à la fin
- Ajouter un élément à une position donnée
- Enlever un élément
- Enlever un élément à une position donnée

## Exemples

```
animaux = ["éléphant", "girafe",  
          "rhinocéros", "gazelle"]
```

```
animaux.append("lion")
```

```
animaux.insert(0, "hippopotame")
```

```
animaux.remove("gazelle")
```

```
del animaux[-2]
```

Que contient alors la liste animaux ?

# Les listes sont modifiables

## Exemples

- Ajouter tous les éléments d'une autre liste      `animaux.extend(["lion", "buffle"])`
- Trier une liste (ordre numérique / alphabétique)      `animaux.sort()`
- Concaténer deux listes      `animaux + ["lion", "buffle"]`
- Inverser l'ordre d'une liste      `animaux.reverse()`
- Cloner une liste      `Lcopie=list(animaux)`

# Listes (Exemples (1))

## ■ Supprimer une entrée avec son index avec la fonction **del**

```
>>> liste = ["a", "b", "c"]
```

```
>>> del liste[1]
```

```
>>> liste
```

```
['a', 'c']
```

## □ Supprimer une entrée avec sa valeur avec la méthode **remove**

```
>>> liste = ["a", "b", "c"]
```

```
>>> liste.remove("a")
```

```
>>> liste
```

```
['b', 'c']
```

# Listes (Exemples (2))

- Inverser les valeurs d'une liste avec la méthode **reverse**

```
>>> liste = ["a", "b", "c"]
```

```
>>> liste.reverse()
```

```
>>> liste
```

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

- Compter le nombre d'items d'une liste

```
>>> liste = [1,2,3,5,10]
```

```
>>> len(liste)
```

5

# Listes (Exemples (3))

## □ Compter le nombre d'occurrences d'une valeur

```
>>> liste = ["a","a","a","b","c","c"]
```

```
>>> liste.count("a")
```

3

```
>>> liste.count("c")
```

2

## □ Trouver l'index d'une valeur

```
>>> liste = ["a","a","a","b","c","c"]
```

```
>>> liste.index("b")
```

3

# Listes (Exemples (4))

- Trouver un item dans une liste avec le mot clé **in**

```
>>> liste = [1,2,3,5,10]
```

```
>>> 3 in liste
```

True

```
>>> 11 in liste
```

False

- La fonction **range**

```
>>> range(10)
```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Listes (Exemples (5))

## ❑ Agrandir une liste

```
>>> x = [1, 2, 3, 4]
```

```
>>> y = [4, 5, 1, 0]
```

```
>>> x.extend(y)
```

```
>>> print x
```

```
[1, 2, 3, 4, 4, 5, 1, 0]
```

## ❑ Additionner deux listes

```
>>> x = [1, 2, 3]
```

```
>>> y = [4, 5, 6]
```

```
>>> x + y
```

```
[1, 2, 3, 4, 5, 6]
```

# Listes (Exemples (6))

## ❑ Multiplier une liste:

```
>>> x = [1, 2]
```

```
>>> x*5
```

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

## ❑ Initialiser une liste:

```
>>> [0] * 5
```

```
[0, 0, 0, 0, 0]
```

# Listes (Exemples (7))

- ❑ Obtenir le plus grand élément d'une liste, ou le plus petit :

```
>>> max([2, 1, 4, 3])
```

4

```
>>> min([2, 1, 4, 3])
```

1

- ❑ Faire la somme d'une liste:

```
>>> sum([2, 1, 4, 3])
```

10

# Listes et chaînes de caractères

## ■ Couper une chaîne de caractères en une liste de chaînes : **split**

✗ **split sans argument** : couper sur les espaces, sauts de ligne, tabulations

```
>>> "Une phrase avec des mots".split()
```

```
["Une", "phrase", "avec", "des", "mots"]
```

✗ **Split avec argument** : couper sur un séparateur donné

```
>>> "Plan-Jaxaay".split("-")
```

```
["Plan", "Jaxaay"]
```

## ■ Assembler une liste de chaînes en une seule chaîne : **join**

```
>>> "/".join(["répertoire", "sous_répertoire", "fichier"])
```

```
"répertoire/sous_répertoire/fichier"
```

# Boucles

■ Une boucle permet d'exécuter plusieurs fois les mêmes commandes

- La boucle parcourt une séquence

**for variable in sequence :**

*commande répétée plusieurs fois*

*commande répétée plusieurs fois*

*suite du programme (exécuté une seule fois)*

# Boucles

- La séquence peut être :

✗ une variable

- Chaîne de caractères
- Liste, ensemble, tuple
- ...

✗ une liste d'indice générée avec la fonction range()

`range(5) => [0, 1, 2, 3, 4]`

# Boucles

- Exemple de séquence de type chaîne de caractères:

```
[>>> prenom="Dahirou"
[>>> for car in prenom:
[...     print(car+' *',end=' ')
[...
D * a * h * i * r * o * u * >>> ]
```

- L'instruction `for` permet donc d'écrire des boucles, dans lesquelles l'itération traite successivement tous les éléments d'une séquence donnée.
- L'argument `end = " "` signifie que vous souhaitez remplacer le saut à la ligne par un simple espace. Si vous supprimez cet argument, les nombres seront affichés les uns en-dessous des autres.

# Boucles

## ■ Exemple de séquence de type Liste

```
[>>> liste=['SATIC','SDD','ECOMIJ']
[>>> for ufr in liste:
[...     print('longueur de la chaine',ufr,'=',len(ufr))
[...
 longueur de la chaine SATIC = 5
 longueur de la chaine SDD = 3
 longueur de la chaine ECOMIJ = 6
>>> █
animaux = ["éléphant", "biche", "rhinocéros", "brochet"]
environnements = ["savane", "forêt", "savane", "rivière"]
X Afficher un animal par ligne
```

```
for animal in animaux : print(animal)
print("C'est fini !")
```

X Afficher un animal par ligne avec son numéro

```
for i in range(len(animaux)) :
    print("Numéro " + str(i) + " : " + animaux[i])
print("C'est fini !")
```

X Afficher un animal par ligne avec son environnement

```
for i in range(len(animaux)) :
    print(animaux[i] + " dans la " + environnements[i])
```

# Boucles (boucler sur plusieurs listes avec zip)

## ■ Exemples

```
animaux = ["éléphant", "biche", "rhinocéros", "brochet"]
```

```
environnements = ["savane", "forêt", "rivière", "rivière"]
```

```
for animal, envir in zip(animaux, environnements):
```

```
    print(animal + " vit dans " + envir)
```

■ Ce qui donne :

éléphant vit dans savane

biche vit dans forêt

Rhinocéros vit dans savane

brochet vit dans rivière

# Boucles (boucler sur plusieurs listes avec des boucles imbriquées)

## ■ Exemples

```
animaux = ["éléphant", "biche", "rhinocéros", "brochet"]
```

```
environnements = ["savane", "forêt", "savane", "rivière"]
```

```
for animal in animaux:
```

```
    for envir in environnements:
```

```
        print (animal+ " dans " + envir )
```

## ■ Ce qui donne :

éléphant dans savane

éléphant dans forêt

éléphant dans rivière

biche dans savane

biche dans forêt

biche dans rivière

rhinocéros dans savane

rhinocéros dans forêt

rhinocéros dans rivière

brochet dans savane

brochet dans forêt

brochet dans rivière

# Boucles

- **break** permet d'interrompre la boucle
- **continue** permet de passer immédiatement à l'élément suivant
- La boucle **for** peut aussi avoir un bloc **else** :

**for variable in liste :**

**if condition d'exclusion :**

**continue**

*commande répétée plusieurs fois*

**if condition :**

*commande exécutée si condition satisfaite*

**break**

**else :**

*commande exécutée si condition jamais satisfaite*

# Boucles

■ **continue** : permet de passer immédiatement à l'élément suivant

**for variable in liste :**

**if condition d'exclusion : continue**

*commande répétée plusieurs fois*

**for i in range(10):**

*if i == 2:*

*continue*

*print(i)*

# Boucles

## ■ break et else :

✗ **break** permet d'interrompre la boucle

```
for i in range(10):
    if i == 2:
        break
```

```
    print(i)
```

✗ le bloc **else** est exécuté si la boucle est allée jusqu'au bout (sans rencontrer **break**)

## for variable in liste :

*commande répétée plusieurs fois*

### if condition :

*commande exécutée si condition satisfaite*

**break**

### else :

*commande exécutée si condition jamais satisfaite*

# Boucles

■ La boucle **while** dure tant qu'une condition est satisfaite :

**while condition :**

*commande répétée plusieurs fois*

*suite du programme (la boucle est finie)*

■ Elle peut aussi avoir un **break** et un **else**

■ Exemple: la liste des carrés et des cubes des nombres de 1 à 12

```
[>>> a=0
[>>> while a<12:
[...     a=a+1
[...     print(a , a**2 , a**3)
[...
```

■ Afficher les dix premiers termes d'une suite appelée « Suite de Fibonacci »

# Boucles

- Afficher les dix premiers termes d'une suite appelée « Suite de Fibonacci ». Il s'agit d'une suite de nombres dont chaque terme est égal à la somme des deux termes qui le précédent

```
[>>> a, b, c = 1, 1, 1
[>>> while c < 11 :
[...     print(b, end = " ")
[...     a, b, c = b, a+b, c+1
```

## ■ Résultat

```
[...
1 2 3 5 8 13 21 34 55 89 >>> ]
```

# Exercice 2 : le dot-plot

■ Le dot-plot est une technique très simple pour analyser visuellement deux séquences d'ADN

■ Principe :

- ✗ Mettre une séquence à l'horizontal et l'autre en vertical, sur une matrice
- ✗ Mettre un X dans les cases où les bases sont les mêmes dans les deux séquences

■ Faire un programme Python pour générer un dot-plot

- ✗ Les séquences d'ADN sont des chaînes de caractères
- ✗ On générera le dot-plot ligne par ligne et on écrira chaque ligne avec print()

A	T	G	G	T	C	A	A	T
A	X						X	X
T		X			X			X
G			X	X				
G			X	X				
T				X	X			
C						X		
A	X						X	X
A	X						X	X
T	X			X				X

# Exercice 2: Nombres paires d'une liste

■ Exemple2 : Garder seulement les nombres pairs d'une liste :

**nombres = [1, 4, 7, 8, 12, 15]**

**nombres\_pairs = [ ] ...????**

# Listes procédurales (*comprehension list*)

- Permet de créer une liste sans donner les éléments un à un mais en donnant une boucle pour la remplir

```
>>> [len(mot) for mot in "Un texte avec des mots".split()]
```

```
[2, 5, 4, 3, 4]
```

sum() retourne la somme d'une liste

```
>>> sum(len(mot) for mot in "Un texte avec des mots".split())
```

```
18
```

- Plusieurs boucles peuvent être combinées

```
print(" " + seq1)
```

```
print("\n".join(base2 + "".join([" ", "X"])[base1 == base2] for base1 in
seq1) for base2 in seq2 ))
```

- => Plusieurs approches différentes pour faire la même chose

✗ Mais en général, une approche est plus simple et claire !

# Programmation orientée objet en Python

Chapitre 5:  
Dictionnaires, Tuples et ensembles

# Dictionnaires

- Les types de données composites que nous avons abordés jusqu'à présent (chaines, listes) étaient tous des séquences, c'est à dire des suites ordonnées d'éléments.
- Dans une séquence, il est facile d'accéder à un élément quelconque à l'aide d'un index (un nombre entier), mais à la condition expresse de connaître son emplacement.
- Les *dictionnaires* que nous découvrons ici constituent un autre *type composite*.
- Ils ressemblent aux listes dans une certaine mesure (ils sont modifiables comme elles), mais ne sont pas des séquences.
- En revanche, nous pourrons accéder à n'importe lequel d'entre eux à l'aide d'un index spécifique que l'on appellera une *clé*, laquelle pourra être alphabétique, numérique, ou même d'un type composite sous certaines conditions.

# Dictionnaires

■ Un tableau associatif qui fait correspondre des valeurs à des clefs : dict

```
dict = { clef1 : valeur1, clef2 : valeur2, ... }
```

```
>>> definitions = {"os" : "système d'exploitation", "python" : "langage de programmation", "dictionnaire" : "association clef-valeur"}
```

```
>>> fiche = {"nom": "Gueye", "prenom": "Dahirou"}
```

```
>>> materiel = {}
```

```
>>> materiel['computer'] = 'ordinateur'
```

```
>>> materiel['mouse'] = 'souris'
```

```
>>> materiel['keyboard'] = 'clavier'
```

```
>>> print(materiel)
```

```
{'computer': 'ordinateur', 'keyboard': 'clavier', 'mouse': 'souris'}
```

# Opérations sur les dictionnaires

■ La recherche est optimisée dans les dictionnaires (table de hachage)

■ Principales opérations :

- Nombre de clés ou de valeur : **len (definitions)**      -> 3
- Obtenir la valeur associée à une clé: **definitions ["python"]** -> langage de programmation
- Ajouter un couple de clé: valeur: **definitions["flottant"] = "nombre décimal"**
- supprimer une clé et la valeur associée: **del definitions["os"]**
- Test d'appartenance

**>>> if "os" in definitions:**

...        **print("Intègre un système")**

...        **else:**

...        **print("Pas de système. Sorry") ...**

**Pas de systèmes. Sorry**

# Opérations sur les dictionnaires

## ■ Récupérer une valeur dans un dictionnaire: get

```
>>> data = {"os": "systeme", "age": 30}  
>>> data.get("os")  
'système'  
>>> data.get("adresse", "Adresse inconnue")  
'Adresse inconnue'
```

## ■ Méthode keys()

```
>>> print(data.keys())
```

## ■ Vérifier la présence d'une clé : haskey

```
>>> data.has_key("os")  
True
```

## ■ Méthode values()

```
>>> print(data.values())
```

# Dictionnaires et boucles

■ Une boucle sur un dictionnaire parcourt les clefs :

```
for clef in définitions :  
    print(clef, " : ", définitions[clef])
```

Ou

```
for clef in définitions.keys(): print cle
```

■ Récupérer les valeurs

```
for valeur in définitions.values() :  
    print(valeur)
```

■ Récupérer les couples (clés:valuers)

```
for clef, valeur in définitions.items() :  
    print(clef, " : ", valeur)
```

■ Dictionnaire procédural :

```
quadruple = { i : 4 * i for i in range(10) }
```

# Construction d'un histogramme à l'aide d'un dictionnaire

■ Supposons par exemple que nous voulions établir l'histogramme qui représente la fréquence d'utilisation de chacune des lettres de l'alphabet dans un texte donné.

■ Méthode:

- Pour chacun de ces caractères, nous interrogeons le dictionnaire à l'aide de la méthode **get()**, en utilisant le caractère en guise de clé, afin d'y lire la fréquence déjà mémorisée pour ce caractère.
- Si cette valeur n'existe pas encore, la méthode **get()** doit renvoyer une valeur nulle. Dans tous les cas, nous incrémentons la valeur trouvée, et nous la mémorisons dans le dictionnaire, à l'emplacement qui correspond à la clé

# Remarques sur les clés

- Les clés ne sont pas nécessairement des nombres ou chaînes de caractères
- En fait nous pouvons utiliser en guise de clés n'importe quel type de données non modifiables : des entiers, des réels, des chaînes de caractères, et même des tuples.
- Considérons par exemple que nous voulions répertorier les arbres remarquables situés dans un grand terrain rectangulaire
- Nous pouvons pour cela utiliser un dictionnaire, dont les clés seront des tuples indiquant les coordonnées x,y de chaque arbre :

```
>>> arb = {}  
>>> arb[(1,2)] = 'Peuplier'  
>>> arb[(3,4)] = 'Platane'  
>>> arb[6,5] = 'Palmier'  
>>> arb[5,1] = 'Cycas'  
>>> arb[7,3] = 'Sapin'  
>>> print(arb[1,2])
```

# Les dictionnaires ne sont pas des séquences

- Les éléments d'un dictionnaire ne sont pas disposés dans un ordre particulier. Des Opérations comme la concaténation et l'extraction (d'un groupe d'éléments contigus) ne peuvent donc tout simplement pas s'appliquer ici.

```
>>> print(arb[1:3])  
***** Erreur : TypeError
```

- Vous avez vu également qu'il suffit d'affecter un nouvel indice (une nouvelle clé) pour ajouter une entrée au dictionnaire. Cela ne marcherait pas avec les listes:

# Les dictionnaires ne sont pas des séquences

```
>>> invent ={"oranges":274, "poires":137, "bananes":312}
```

```
>>> invent['cerises']= 987
```

```
>>> print(invent)
```

```
{'oranges': 274, 'poires': 137, 'bananes': 312, 'cerises': 987}
```

```
>>> liste =['jambon', 'salade', 'confiture', 'chocolat']
```

```
>>> liste[4] ='salami'
```

```
***** IndexError: list assignment index out of range *****
```

# Contrôle du flux d'exécution à l'aide d'un dico

- Il arrive fréquemment que l'on ait à diriger l'exécution d'un programme dans différentes directions, en fonction de la valeur prise par une variable.
- Vous pouvez bien évidemment traiter ce problème à l'aide d'une série d'instructions if - elif - else , mais cela peut devenir assez lourd et inélégant

```
materiau = input("Choisissez le matériau : ")  
if materiau == 'fer':fonctionA()  
elif materiau == 'bois':fonctionC()  
elif materiau == 'cuivre':fonctionB()  
elif materiau == 'pierre':fonctionD()  
elif ... etc ...
```

# Contrôle du flux d'exécution à l'aide d'un dico

```
materiau = input("Choisissez le matériau : ")  
dico = {'fer':fonctionA,  
        'bois':fonctionC,  
        'cuivre':fonctionB,  
        'pierre':fonctionD,  
        ... etc ...}
```

- Les deux se résument:

**dico[materiau]**

- On peut améliorer la technique avec get()

**dico.get(materiau, fonctAutre)()**

# Tuples

- Du point de vue de la syntaxe, un tuple est une collection d'éléments séparés par des virgules :
- Très proches des listes. La différence est qu'ils ne sont pas modifiables
- Les tuples s'écrivent entre ou sans parenthèses

```
>>> triplet = (1, 2, 3)  
>>> couple = (1, 2)  
>>> tup = 'a', 'b', 'c', 'd', 'e'  
>>> print(tup)  
('a', 'b', 'c', 'd', 'e')
```

# Tuples

- Si un seul élément, il faut mettre une virgule après

```
>>> tuple_a_un_element = (1,)
```

- Les parenthèses ne sont pas obligatoires

```
>>> mon_tuple = 1, 2, 3
```

- Afficher une valeur d'un tuple

```
>>> mon_tuple[0]
```

1

# Tuples

## ■ Les tuples ne sont pas modifiables

- ✗ => ils peuvent servir de clef dans les dictionnaires
- ✗ C'est leur principale utilisation

Changement de valeur

```
>>> mon_tuple[1] = "ok"
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

# Opérations sur les tuples

■ NB: Les opérations que l'on peut effectuer sur des tuples sont syntaxiquement similaires à celles que l'on effectue sur les listes, si ce n'est que les tuples ne sont pas modifiables :

```
>>> print(tup[2:4])  
('c', 'd')
```

```
>>> tup[1:3] = ('x', 'y')  
=> ***** erreur ! *****
```

```
>>> tup = ('bambe',) + tup[1:]
```

```
>>> tup  
('bambe', 'b', 'c', 'd', 'e')
```

```
>>>
```

# Opérations sur les tuples

■ Les opérateurs de concaténation et de multiplication fonctionnent aussi. Mais puisque les tuples ne sont pas modifiables, vous ne pouvez pas utiliser avec eux, ni l'instruction `del` ni la méthode `remove()` :

```
>>> tu1, tu2 = ("a","b"), ("c","d","e")
>>> tu3 = tu1*4 + tu2
>>> tu3
('a', 'b', 'a', 'b', 'a', 'b', 'a', 'b', 'c', 'd', 'e')

>>> for e in tu3:
...     print(e, end=":")
...
a:b:a:b:a:b:c:d:e:
```

# Opérations sur les tuples

```
>>> del tu3[2]
```

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

TypeError: 'tuple' object doesn't support item deletion

# Tuples

- Le tuple permet une affectation multiple

```
>>> v1, v2 = 11, 22
```

```
>>> v1
```

```
11
```

```
>>> v2
```

```
22
```

- Il permet également de renvoyer plusieurs valeurs lors d'un appel d'une fonction:

```
>>> def renvoyer_nom():
```

```
...     return "Dahirou", "Gueye"
```

```
...
```

```
>>> renvoyer_nom()
```

```
('Dahirou', 'Gueye')
```

# Ensembles

## ■ Ensembles (**set**) au sens mathématique, proches des listes

✗ Les ensembles ne sont pas ordonnés

✗ Les doublons sont automatiquement enlevés des ensembles

## ■ Contrairement aux séquences comme les listes et les tuples dans lesquels chaque élément est indexé

## ■ Les ensembles s'écrivent entre accolades

```
>>> {1, 1, 2, 3}  
{1, 2, 3}
```

```
>>> {1, 1, 2, 3} == {2, 3, 1}  
True
```

```
>>> my_set = {1.0, "Hello", (1, 2, 3)}
```

# Ensembles

■ Les opérations ensemblistes sont disponibles, ex :

**Exemple 1:**

```
>>> {1, 2, 3}.intersection({2, 3, 4})  
{2, 3}
```

■ Convertir une liste en ensemble : `set([1, 2, 3])`

**Exemple2:**

```
>>> s1 = set([4, 6, 9])  
>>> s2 = set([1, 6, 8])  
>>> s1.intersection(s2)  
set([6])
```

# Ensembles

■ Union: Somme des éléments de deux ensembles

```
>>> s1 = set([4, 6, 9])
```

```
>>> s2 = set([1, 6, 8])
```

```
>>> s1.union(s2)
```

```
set([1, 4, 6, 8, 9])
```

```
>>> s1 | s2
```

```
set([1, 4, 6, 8, 9])
```

# Ensembles

- Un "**frozenset**" (ensemble figé) se comporte comme un ensemble,
- Sauf qu'il est immutable, c'est-à-dire qu'une fois créé, on ne peut pas le mettre à jour
- Il dispose donc des mêmes fonctions que le type "set", mais sans "add", "update", "pop", "remove", "discard" et "clear".
- Ce type de données prend en charge des méthodes telles que `copy()`, `difference()`, `intersection()`, `isdisjoint()`, `symmetric_difference()` et `union()`
- De plus, ils sont hachables, ce qui leur permet de faire partie d'ensembles.
- Ensemble immutable : **frozenset([1, 2, 3])** (clé de dictionnaire)

# Ensembles

```
>>> fs = frozenset([2, 3, 4])
```

```
>>> s1 = set([fs, 4, 5, 6])
```

```
>>> s1
```

```
{frozenset({2, 3, 4}), 4, 5, 6}
```

```
>>> fs.intersection(s1)
```

```
frozenset({4})
```

```
>>> fs.add(6)
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

**AttributeError: 'frozenset' object has no attribute 'add'**

```
>>>
```

# Programmation orientée objet en Python

## Chapitre 6: Fichiers

# Fichiers

Une manière de stocker des données de manière pérenne est de les stocker dans des fichiers.

## ■ Editer un fichier

- Pour éditer un fichier en python on utilise la fonction open .
- Cette fonction prend en premier paramètre le chemin du fichier (relatif ou absolu) et en second paramètre le type d'ouverture

## ■ Chemin relatif / chemin absolu

- Un chemin relatif est un chemin qui prend en compte l'emplacement de lecture.
- Un chemin absolu est un chemin complet qui peut être lu quelque soit l'emplacement de lecture.

# Fichiers

- Les fichiers sont ouverts avec la fonction `open()`:

```
fichier = open("chemin/nom_du_fichier", mode)
```

```
fichier = open("/tmp/test.txt")          # lecture (r) mode texte  
(valeur par défaut)
```

```
fichier = open("/tmp/test.txt", "w")      # écrire (w) mode texte
```

```
fichier = open("/tmp/test.bin", "rb")     # lecture (r) mode binaire (b)
```

```
fichier = open("/tmp/test.bin", "wb")     # écrire (w) mode binaire (b)
```

✗ Ouvrir un fichier en écriture crée le fichier s'il n'existe pas, et l'écrase sinon

- Python gère la conversion des sauts de ligne (Unix / Windows / Mac)

# Fichiers

## ■ Les types d'ouverture

**r**, pour une ouverture en lecture (READ).

**w**, pour une ouverture en écriture (WRITE), à chaque ouverture le contenu du fichier est écrasé. Si le fichier n'existe pas python le crée.

**a**, pour une ouverture en mode ajout à la fin du fichier (APPEND). Si le fichier n'existe pas python le crée.

**b**, pour une ouverture en mode binaire.

**t**, pour une ouverture en mode texte.

**x**, crée un nouveau fichier et l'ouvre pour écriture

# Fichiers

## ■ Principales opérations :

```
>>> contenu=fichier.read() # lire tout le contenu du fichier
```

```
>>> fichier.write("contenu") # écrire dans le fichier
```

```
>>> fichier.close() # fermer le fichier (automatiquement appelé quand l'objet fichier  
est détruit)
```

## ■ Lire le contenu d'un fichier

✗ Exemple1 :

```
fichier = open("test.txt", "r")
```

```
print fichier.read()
```

```
fichier.close()
```

✗ Exemple2 :

```
for ligne in open("/tmp/test.txt").read().split("\n"):  
    print(ligne)
```

# Fichiers

## ■ Ecrire dans un fichier

```
fichier = open("test.txt", "a")
```

```
fichier.write("Bonjour monde")
```

```
fichier.close()
```

■ Pour le mode d'ouverture **a** , si vous voulez écrire à la ligne, vous pouvez utiliser le saut de ligne **\n** :

```
fichier = open("test.txt", "a")
```

```
fichier.write("\nBonjour tout monde")
```

```
fichier.close()
```

# Fichiers

## ■ Le mot clé **with**

- Il existe une autre syntaxe plus courte qui permet de s'émanciper du problème de fermeture du fichier: le mot clé **with**

**with open("test.txt", "r") as fichier:**

**print fichier.read()**

# Fichiers: Exercices

## ■ Enoncé 1:

**Vous devez lire et écrire des données dans un fichier texte. Si le fichier n'existe pas, il doit être créé. Si le fichier a un contenu, le contenu doit être supprimé. Quel code devriez vous utiliser?**

- A. `open ("local_data", "r")`
- B. `open ("local_data", "r+")`
- C. `open ("local_data", "w+")`
- D. `open ("local_data", "w")`

# Fichiers: Exercices

## ■ Enoncé 2:

**Créer une fonction qui lit un fichier de données et imprime chaque ligne du fichier**

## ■ Enoncé 3:

**Ecrire un programme python qui permet de regrouper dans une liste les mots communs à deux fichiers textes: fichier1 et fichier2**

# Programmation orientée objet en Python

## Chapitre 7: Fonctions

# Fonctions

- Une fonction (ou *function*) est une suite d'instructions que l'on peut appeler avec un nom.
- Les fonctions sont créées avec l'instruction **def** :

**def nom\_fonction(nom\_paramètre1, nom\_paramètre2,...):**

*corps de la fonction*

**return valeur de retour**

✗ Exemple1:

```
>>> def indique_mon_age():
```

```
...     return 25
```

```
...
```

✗ Exemple2:

```
def table_de_multiplication(nombre):  
    for i in range(1, 11):  
        print(i, "x", nombre, "=", i * nombre)
```

# Fonctions

■ Les fonctions sont appelées avec des parenthèses

*nom\_fonction(valeur\_paramètre1, valeur\_paramètre2,...)*

>>> **indique\_mon\_age()**

25

>>> **table\_de\_multiplication(7)**

(1, 'x', 7, '=', 7)

(2, 'x', 7, '=', 14)

(3, 'x', 7, '=', 21)

(4, 'x', 7, '=', 28)

(5, 'x', 7, '=', 35)

(6, 'x', 7, '=', 42)

(7, 'x', 7, '=', 49)

(8, 'x', 7, '=', 56)

(9, 'x', 7, '=', 63)

(10, 'x', 6, '=', 70)

# Fonctions

Exemple 3:

```
def affiche_carre(n):  
    print(" le carre de " , n "vaut", n*n)
```

Qui s'utiliserait comme ceci:

```
affiche_carre(12)
```

Le carre de 12 vaut 144

Mais en fait il serait plus intéressant de retourner qq chose

```
Def carre(n):
```

```
    return n*n
```

```
If carre(8) <= 100:
```

```
    print (" petit appartement ")
```

# Fonctions

■ Possibilité de retourner plusieurs valeurs (à l'aide d'un tuple) :

```
>>> def extrémités(liste) :  
...     return liste[0], liste[-1]
```

```
>>> i, j = extrémités(range(10))
```

```
>>> i
```

```
0
```

```
>>> j
```

```
9
```

■ Des variables locales peuvent être définies à l'intérieur de la fonction

```
def calcul_compliqué(x, y):  
    résultat_intermédiaire = x + y  
    résultat_final = résultat_intermédiaire + 1  
    return résultat_final
```

# Fonctions

## ■ Les paramètres peuvent avoir une valeur par défaut

✗ On parle de paramètres optionnels, par exemple :

```
def table_de_multiplication(nombre, début = 1, fin = 10):  
    for i in range(début, fin):  
        print(i, "x", nombre, "=", i * nombre)
```

✗ Si la valeur par défaut doit être calculé à partir d'autres valeurs :

```
def créer_utilisateur(nom, login = None) :  
    login = login or nom
```

login vaut par défaut la valeur de nom

## ■ Les paramètres d'une fonction peuvent être nommés lors de l'appel, ce qui permet de les passer dans le désordre :

```
résultat = nom_fonction (paramètre2 = valeur_paramètre 2, paramètre1 =  
valeur_paramètre 1)
```

Exemple: `table_de_multiplication(nombre, fin = 20)`

# Fonctions

## ■ L'opérateur splat

L'opérateur splat : \* est très souvent utilisé en python.

```
def ma_function(*var)
```

```
def ma_function(**var)
```

```
ma_function(*var)
```

```
ma_function(**var)
```

# Fonctions

## ■ Une liste en paramètre

Exemple 1: On peut récupérer les valeurs renseignées via une liste:

```
>>> def additionne_moi(*param):
...     return param[0] + param[1] + param[2]
```

...

```
>>> additionne_moi(10, 20, 30)
```

60

L'utilisation de l'étoile permet de passer par une liste:

```
>>> data = [1, 2, 3]
>>> additionne_moi(*data)
```

# Fonctions

## ■ Utilisez un dictionnaire pour les paramètres

Vous pouvez utiliser un dictionnaire en paramètres. Pour cela, vous devez ajouter une double étoile: \*\*

```
>>> def ma_fiche(**parametres):  
...     return parametres["prenom"]
```

...

```
>>> ma_fiche(prenom="Dahirou")  
'Dahirou'
```

# Fonctions

## ■ Utilisation de splat dictionnaire au niveau des appels de fonctions

Prenons l'exemple de cette fonction:

```
>>> def test(firstname="", lastname=""):  
...     return "{} {}".format(firstname, lastname)
```

Créons notre dictionnaire:

```
>>> data = {'firstname':'Dahirou', 'lastname':'Gueye'}
```

Et envoyons notre variable avec une étoile \*

```
>>> test(*data)  
  
'lastname firstname'
```

Puis avec deux étoiles \*\*

```
>>> test(**data)  
  
'Dahirou Gueye'
```

# Fonctions

## Exemple 2 paramètres liste et dictionnaire :

- Une fonction peut avoir un nombre de paramètre variable, nommé ou non

```
>>> def fonction_à_paramètres_variables(*args, **kargs):    kargs = keyword  
...          print(args, kargs)                                arguments
```

```
>>> fonction_à_paramètres_variables(1, 2, 3, param = 4)  
(1, 2, 3) { "param" : 4 }
```

- Cette syntaxe peut aussi s'utiliser lors de l'appel d'une fonction

```
args = [5, 1, 10]  
table_de_multiplication(*args)
```

```
kargs = { "nombre" : 5, "début" : 1, "fin" : 10 }  
table_de_multiplication(**kargs)
```

✗ sont équivalents à :

```
table_de_multiplication(5, 1, 10)
```

# Fonctions

## ■ Portée des variables (variable globale et variable locale)

Une variable déclarée à la racine d'un module est visible dans tout ce module.  
On parle alors de variable globale.

```
>>> x = "hello"
```

```
>>> def test():
```

```
...     print x
```

```
...
```

```
>>> test() hello
```

# Fonctions

Et une variable déclarée dans une fonction ne sera visible que dans cette fonction. On parle alors de variable locale.

```
>>> x = False  
  
>>> def test():  
...     x = "hello"  
  
... >>> test()  
  
>>> x  
  
False
```

# Fonctions

■ En Python, les fonctions sont des objets

```
>>> def nombre_suivant(x) : return x + 1  
>>> variable_fonction = nombre_suivant  
>>> print(variable_fonction(5))  
6
```

# Fonctions prédéfinies

■ Il existe des fonctions internes à python:

**abs(x)** Retourne une valeur absolue

```
>>> abs(-1)
```

1

**all(iterable)** Retourne True si tous les éléments d'un élément itérable sont True

```
>>> liste = [True,True,True,1]
```

```
>>> all(liste)
```

True

**any(iterable)** Retourne True si au moins un élément d'un élément itérable est True

```
>>> liste = [True, False, True]
```

```
>>> any(liste)
```

True

# Fonctions prédéfinies

**bin(x)** Convertit un integer en chaîne de caractères binaires.

```
>>> bin(101)
```

```
'0b1100101'
```

**callable(object)** Détermine si un objet est *callable*.

```
>>> callable("A")
```

```
False
```

```
>>> callable(int)
```

```
True
```

**str.capitalize()** La méthode capitalize permet de mettre une chaîne de caractères au format Xxxxx

```
>>> "oLIViER".capitalize() 'Olivier'
```

# Fonctions prédéfinies

**choice([])** Retourne une valeur d'une liste aléatoirement.

```
>>> import random
```

```
>>> random.choice([1,2,3,4,5])
```

3

```
>>> random.choice([1,2,3,4,5])
```

2

**str.count(string)** La méthode count compte le nombre d'occurrences de la recherche demandée.

```
>>> "ifoad".count("o")
```

1

# Fonctions prédéfinies

**str.endswith(str)** La méthode endswith test si une chaîne de caractères se termine par la chaîne demandée

```
>>> a = "olivier"
```

```
>>> a.endswith("r")
```

True

```
>>> a.endswith("er")
```

True

```
>>> a.endswith("é")
```

False

**str.find(string)** La méthode find trouve la première occurrence de la recherche demandée.

```
>>> "sagna".find("a")
```

# Fonctions prédéfinies

**help(element)** Cette fonction vous retourne des informations sur l'utilisation de l'élément qui vous intéresse.

**Hex** Convertit un nombre en valeur hexadécimale.

```
>>> hex(16)
```

```
'0x10'
```

**str.islower()** Retoune True si tous les caractères sont en minuscule.

```
>>> "olivier".islower() True
```

```
>>> "Olivier".islower() False
```

**str.isspace()** Retourne True si il n'y a que des espaces et au moins un caractère.

```
>>> " ".isspace() True
```

```
>>> "jean louis".isspace() False
```

```
>>> " ".isspace() True
```

# Fonctions prédéfinies

**str.istitle()** Retourne True si la chaîne a un format titre.

```
>>> "Titre".istitle() True
```

```
>>> "TitrE".istitle() False
```

```
>>> "Titre de mon site".istitle() False
```

```
>>> "Titre De Mon Site".istitle() True
```

**str.isupper()** Retourne True si tous les caractères sont en majuscule et qu'il y a au moins un caractère.

```
>>> "OLIVIER".isupper()
```

True

```
>>> "Olivier".isupper()
```

False

```
>>> "OlivieR".isupper()
```

False

# Fonctions prédéfinies

**len(s)** Retourne le nombre d'items d'un objet.

```
>>> len([1,2,3])
```

```
3
```

```
>>> len("olivier")
```

```
7
```

**str.lower()** La méthode lower permet de mettre en minuscule une chaîne de caractères.

```
>>> "IFOAD".lower()
```

```
'ifoad'
```

**randint()** Retourne un int aléatoire.

```
>>> import random
```

```
>>> random.randint(1,11)
```

```
5
```

# Fonctions prédéfinies

**random()** Retourne une valeur aléatoire.

```
>>> import random
```

```
>>> random.random()
```

```
0.9563522652738929
```

**str.replace(string, string)** La méthode replace remplace un segment d'une chaîne de caractères par une autre:

```
>>> "Tic".replace("ic","IC")
```

```
'TIC'
```

**reverse()** La méthode reverse inverse l'ordre d'une liste.

```
>>> x = [1,4,7]
```

```
>>> x.reverse()
```

```
>>> x
```

```
[7, 4, 1]
```

# Fonctions prédéfinies

**shuffle([])** Mélange aléatoirement une liste.

```
>>> import random
```

```
>>> x = [1,2,3,4,5]
```

```
>>> random.shuffle(x)
```

```
>>> x
```

```
[2, 5, 4, 1, 3]
```

**list.sort()** La méthode sort permet de trier une liste.

```
>>> l = [5,1,4,2,10]
```

```
>>> l.sort()
```

```
>>> l
```

```
[1, 2, 4, 5, 10]
```

# Fonctions prédéfinies

**sorted(iterable)** Tri un élément itérable.

```
>>> sorted([3,2,12,1])
```

```
[1, 2, 3, 12]
```

**str.split(séparateur)** La méthode split transforme une chaîne de caractères en liste.

```
>>> "olivier:sagna".split(":")
```

```
['olivier', 'sagna']
```

**sum(iterable [,start])** Additionne les valeurs d'un élément itérable.

```
>>> sum([1,2,3])
```

6

# Exercice: Fonctions et Fichiers

## ■ Exercice comptage de mots :

- ✗ Nous allons écrire un programme pour compter le nombre d'exemplaire de chaque mot présent dans un texte
- ✗ Le programme :
  - ✗ Lit le fichier **texte.txt**, qui contient un poème
  - ✗ Passe le texte lu en minuscule et supprime les virgules (en les remplaçant par des chaînes vides)
  - ✗ Découpe le texte en une liste de mots
  - ✗ Compte le nombre de fois où chaque mot est rencontrée à l'aide d'un dictionnaire que l'on appellera **mot\_2\_nb**
  - ✗ Affiche le résultat avec un mot par ligne et le nombre de fois où il a été trouvé

# Tri

■ La méthode `sort()` permet de trier le contenu d'une liste (ordre numérique ou alphabétique)

```
>>> ma_liste = [2, 3, 1, 5, 4]
>>> ma_liste.sort()
>>> ma_liste
[1, 2, 3, 4, 5]
```

- Pour trier selon un ordre arbitraire, on utilise une fonction de tri
  - ✗ Cette fonction prend un élément en paramètre et retourne une valeur
    - ➊ Les éléments seront triés dans l'ordre des valeurs rentrées par la fonction

# Tri

- ✗ Exemple pour trier des mots par longueur :

```
>>> def fonction_de_tri(mot):
...     return len(mot)
>>> ma_liste = ["table", "voiture", "bal"]
>>> ma_liste.sort(key = fonction_de_tri)
>>> ma_liste
["bal", "table", "voiture"]
```

- ✗ sort() trie la liste au moment où il est appelé, mais ne garantit pas que la liste reste triée en cas de modification
- ✗ Le module **bisect** permet de garder une liste triée lorsqu'on ajoute un élément

Les fonctions suivantes sont fournies :

- ✗ bisect.bisect\_left(a, x, lo=0, hi=len(a))
- ✗ bisect.bisect\_right(a, x, lo=0, hi=len(a))
- ✗ bisect.insort\_left(a, x, lo=0, hi=len(a))
- ✗ bisect.insort\_right(a, x, lo=0, hi=len(a))¶

# Tri

## ✗ `bisect.bisect_left(a, x, lo=0, hi=len(a))`

Trouve le point d'insertion de `x` dans `a` permettant de conserver l'ordre. Les paramètres `lo` et `hi` permettent de limiter les emplacements à vérifier dans la liste, par défaut toute la liste est utilisée. Si `x` est déjà présent dans `a`, le point d'insertion proposé sera avant (à gauche) de l'entrée existante.

Le point d'insertion renvoyé, `i`, coupe la liste `a` en deux moitiés telles que, pour la moitié de gauche : `all(val < x for val in a[lo:i])`, et pour la partie de droite : `all(val >= x for val in a[i:hi])`.

## ✗ `bisect.bisect_right(a, x, lo=0, hi=len(a))`

Semblable à `bisect_left()`, mais renvoie un point d'insertion après (à droite) d'une potentielle entrée existante valant `x` dans `a`.

## ✗ `bisect.insort_left(a, x, lo=0, hi=len(a))`

insère `x` dans `a` en conservant le tri.

## ✗ `bisect.insort_right(a, x, lo=0, hi=len(a))¶`

Similaire à `insort_left()`, mais en insérant `x` dans `a` après une potentielle entrée existante égale à `x`.

# Exercice

## ■ Exercice comptage de mots :

- ✖ Modifier le programme précédent pour trier les résultats en fonction du nombre de fois où le mot a été rencontré
  - ✖ Astuce : pour cela, on transformera le dictionnaire en une liste de couple (clef, valeur)

# Programmation orientée objet en Python

Chapitre 8:  
Classes et objets

# Conventions de noms

■ Pas de notion de « privé » ou « public » en Python => importance des conventions

- Les noms de classe avec une majuscule, les mots séparés par une majuscule
- Les noms de variables en minuscule, les mots séparés par des soulignement « \_ »
- Les noms de collections (listes, dictionnaires,...) au pluriel
- Pour les dictionnaires, on peut aussi utiliser la forme « x\_2\_y » (2 = to en anglais)
- Les constantes entièrement en majuscule
- Les attributs « considérés comme privé » sont précédés d'un soulignement
- Les attributs et méthodes spéciaux de Python sont précédés et suivis de deux soulignements (ex : \_\_init\_\_ )

# Conventions de noms

- Exemple :
  - ✗ Personne ou EtreVivant : la classe des personnes ou des êtres vivants
  - ✗ dahirou ou bouna\_gueye : une personne
  - ✗ personnes : une liste / un dictionnaire contenant des personnes
  - ✗ oeuvre\_2\_artiste : un dictionnaire faisant correspondre des œuvres (les clefs) à des artistes (les valeurs)
  - ✗ \_oeuvre : un attribut privé

# Programmation orienté objet

■ Une approche de la programmation dans laquelle le programme est organisé sous la forme « d'objet »

- Un objet est une entité distincte et indépendante
- Un objet peut être relié à d'autres objets
- Un objet représente dans le programme une chose concrète manipulée par l'ordinateur : un utilisateur, un produit en vente, une pièce dans un musée,...

# Programmation orienté objet

- Un objet rassemble à la fois les données et les traitements (appelés méthodes) qui s'applique sur ces données : c'est l'encapsulation
  - ✗ Données : nom, prenom, age, etc.
  - ✗ Traitement : poser une question à la personne à propos de son vieillissement

**personne1**

**Données :**

nom : Thiam

prenom : Moussa

...

**Traitements :**

Poser une question  
à cette personne

**personne2**

**Données :**

nom : Sow

prenom : Magatte

...

**Traitements :**

Poser une question  
à cette personne

**personne3**

**Données :**

nom : Diedhiou

prenom : Maria

...

**Traitements :**

Poser une question  
à cette personne

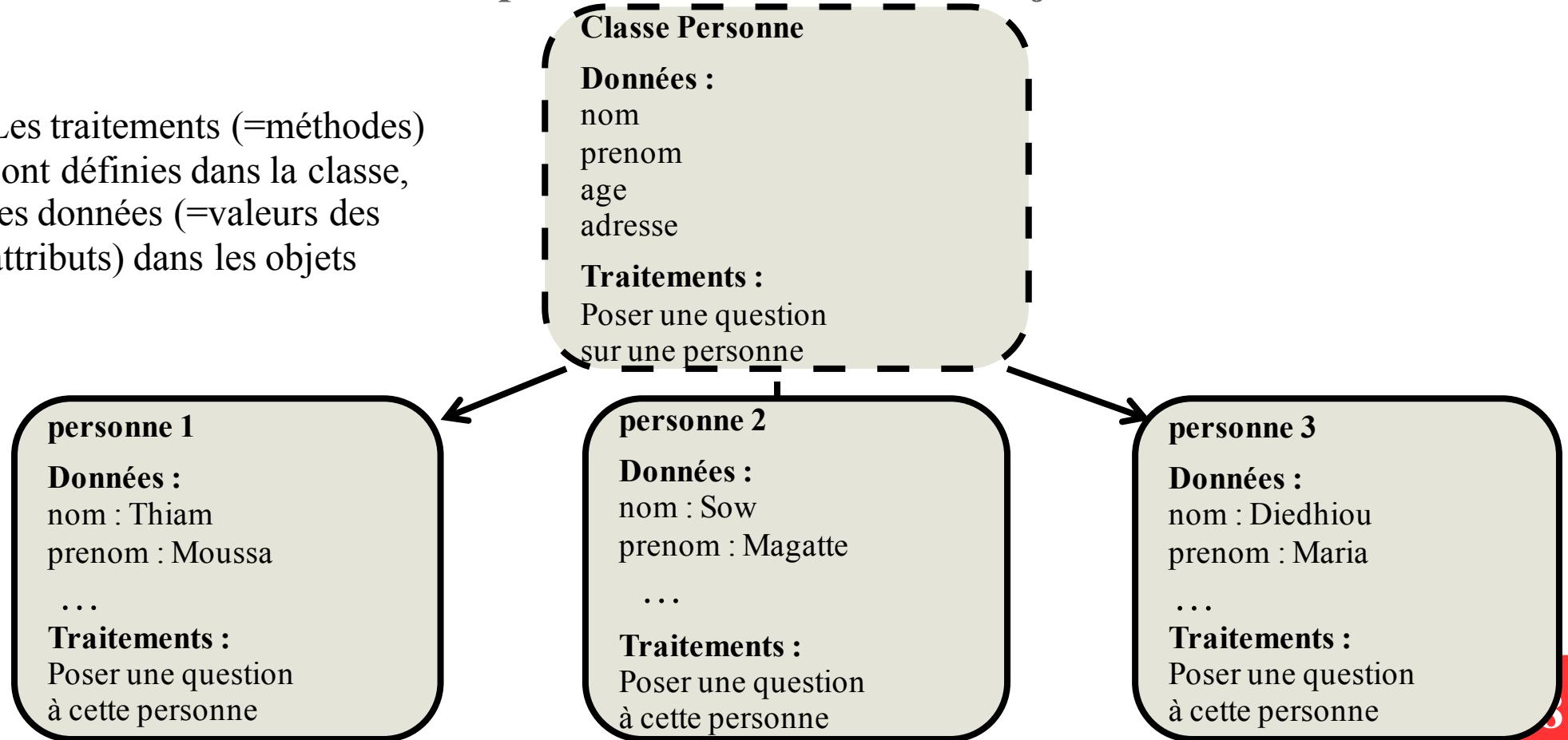
# Programmation orienté objet

■ Un objet est créé à partir d'une classe

✗ Une classe est une catégorie abstraite d'objet

✗ C'est un « moule » permettant de créer des objets similaires

Les traitements (=méthodes) sont définies dans la classe, les données (=valeurs des attributs) dans les objets



# Classes

■ Les classes sont créées avec l'instruction **class**

■ Les méthodes à l'intérieur des classes avec **def** (comme les fonctions)

**class nom\_de\_la\_classe (classe\_parente1, classe\_parente2,...):**

*attribut\_de\_classe = valeur*

**def \_\_init\_\_(self, paramètres...):** # constructeur  
*self.attribut\_de\_l'objet = valeur*

**def nom\_de\_méthode(self, paramètres...):**  
*corps de la méthode*  
**return valeur de retour**

■ Pour une classe « vide » (sans méthode ni attribut) :

**class nom\_de\_la\_classe (classes.parentes...):**  
**pass**

# Classes

- La classe parente est **object** s'il n'y en a pas de plus précise
- Le premier paramètre « **self** » est l'objet auquel est appliquée la méthode
  - ✗ Équivalent au « **this** » en Java
  - ✗ Il est nommé « **self** » par convention, mais on peut choisir un autre nom
- Les objets sont ensuite créés en appelant la classe comme une fonction  
*nom\_de\_l'objet = nom\_de\_la\_classe(paramètres du constructeur...)*

# Classes

■ Le constructeur `__init__()` est appelé à la création de l'objet, pour l'initialiser

- ✗ Il peut prendre des paramètres si nécessaires
- ✗ En général, les valeurs des paramètres sont stockés dans l'objet

■ Les attributs de l'objet

- ✗ Ils ne sont pas déclarés ni typés (comme pour les variables)

■ Exemple 1:

```
class Personne(object):
```

```
    def __init__(self, nom, prenom, age, adresse):
```

```
        self.nom=nom
```

```
        self.prenom=prenom
```

```
    ...
```

# Quelques remarques

- Tous les attributs et méthodes des classes Python sont « publics » au sens de C++, parce que « nous sommes tous des adultes ! » (citation de Guido van Rossum, créateur de Python).
- Les attributs ne sont pas déclarés ni typés (comme pour les variables): on peut se contenter de les initialiser dans le constructeur !
- Le constructeur d'une classe est une méthode spéciale qui s'appelle `__init__()`.
  - ✗ Il est appelé à la création de l'objet, pour l'initialiser
  - ✗ Il peut prendre des paramètres si nécessaires
  - ✗ En général, les valeurs des paramètres sont stockées dans l'objet
- Toutes les méthodes prennent une variable `self` (sauf les méthodes de classe dont nous parlerons plus tard) comme premier argument. Cette variable est une référence à l'objet manipulé.

# Classes

■ Exemple (NB on peut mettre plusieurs classes dans un fichier) :

```
class Musée(object) :
```

```
    def __init__(self, nom) :
```

```
        self.nom = nom
```

```
        self.oeuvres = [ ]
```

```
    def add_œuvre(self, œuvre) :
```

```
        self.oeuvres.append(œuvre)
```

```
        œuvre.musée = self
```

```
class Oeuvre(object) :
```

```
    def __init__(self, titre, année, photo, artiste) :
```

```
        self.titre = titre
```

```
        self.année = année
```

```
        self.photo = photo
```

```
        self.artiste = artiste
```

```
    def poser_question(self, question) :
```

```
        print("On demande", question, "à propos de l'œuvre", self.titre, "!!")
```

# Objets

■ Les objets sont ensuite créées en appelant la classe comme une fonction

*nom\_de\_l'objet = nom\_de\_la\_classe ( paramètres du constructeur... )*

■ Exemple 1 par rapport à la classe Personne:

```
perso1= Personne("Cisse", "Abdallah ", 55, "Carapaces")
```

```
perso2= Personne("Gueye", "Dahirou ", 37, "Bambey")
```

```
print perso1.prenom, perso1.nom, perso1.age, perso1.adresse
```

```
print type(perso1)
```

```
print perso1.__class__.__name__
```

Exécution:

Abdallah Cissé 55 Carapaces

<class '\_\_main\_\_.Personne'>

Personne

# Objets

■ Exemple 2 par rapport aux classes Musée et Oeuvre:

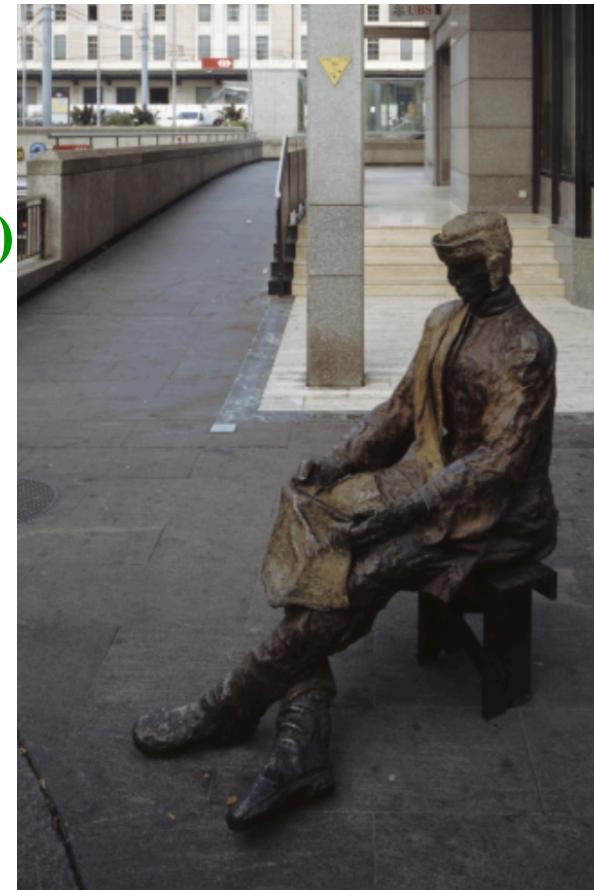
```
>>> mos = Musée("Maison Ousmane Sow")
```

```
>>> immigré = Oeuvre("L'Immigré", 2008, "immigré.jpg", "Ousmane  
Sow", "Genève")
```

■ Les méthodes sont appelées avec la « notation pointée » :

*✗ nom\_de\_l'objet . nom\_de\_la\_méthode ( paramètres... )*

```
>>> mos.add_oeuvre(immigré)
```



# Objets

■ Eemple3 par rapport aux classes Musée et Oeuvre:

```
>>> louvre = Musée("Le Louvre")
```

```
>>> liberté = Oeuvre("La Liberté guidant le peuple", 1830, "liberte.jpg",  
"Eugène Delacroix")
```

■ Les méthodes sont appelées avec la « notation pointée » :

*✗ nom\_de\_l'objet . nom\_de\_la\_méthode ( paramètres... )*

```
>>> louvre.add_oeuvre(liberté)
```

# Objets

■ De même pour les attributs :

✗ *nom\_de\_l'objet . nom\_de\_l'attribut*

>>> liberté.titre

"La Liberté guidant le peuple"

>>> liberté.titre = "(nouveau titre)"

✗ **del** permet aussi de supprimer l'attribut

>>> **del** liberté.titre

■ L'objet nul / vide s'appelle **None**

# Classes et objets

## ■ Exercice : le musée virtuel 1

- Nous allons reprendre l'exemple du musée et de l'œuvre d'art
  - ✗ Une partie du code source est donnée dans les slades précédents. Veuillez continuer l'implémentation en créant un objet de type Musée et des œuvres (liberté, jaconde et venus) de type Œuvre.
  - ✗ ajouter ces trois œuvres et poser une question à propos de l'oeuvre
  - ✗ Tout ça dans le fichier nommé exo\_musee\_1.py
- Ajouter à la classe Musée une méthode permettant de calculer la période couverte par les collections du musée
  - ✗ c'est-à-dire retournant l'année de l'œuvre la plus ancienne et l'année de l'œuvre la plus récente
- Tester cette méthode en l'appelant à la fin du programme

# Sous le capot ?

■ L'un des points fort de Python est la transparence de son système objet : il est possible de comprendre comment cela fonctionne

■ Un objet = un dictionnaire et une référence à une classe

objet oeuvre\_1

Dictionnaire : ( \_\_dict\_\_ )

```
{ "titre" : "La Liberté guidant le peuple",
  "photo" : "liberte.jpg",
  "artiste" : "Eugène Delacroix" }
```

Classe : ( \_\_class\_\_ )

Référence à la classe Oeuvre

■ oeuvre\_1 . attribut\_x

- 1) chercher attribut\_x dans le dictionnaire \_\_dict\_\_
- 2) si trouvé, retourner \_\_class\_\_.attribut\_x
- 3) Si pas trouvé, déclencher une erreur !

■ oeuvre\_1 . méthode\_y(arg1, arg2)

oeuvre\_1 . \_\_class\_\_ . méthode\_y(oeuvre\_1, arg1, arg2)



# Conditions et objets

■ Pour tester si une variable contient bien un objet (et pas **None**) :

```
if personne : print("la personne existe, elle s'appelle :",
    personne.nom)
else :      print("il n'y a personne (personne = None)")
```

■ Pour tester si deux variables contiennent le même objet :

```
if objet_1 is objet_2: print("ce sont les mêmes !")
else : print("ce sont deux objets distincts")
```

- Attention, **objet\_1 == objet\_2** teste l'égalité des objets
  - ✗ Ce n'est pas la même chose
  - ✗ Python permet de redéfinir l'opération **==**  
(via la méthode **\_\_eq\_\_()**)

# Conditions et objets

■ Pour tester si un objet est de la classe X (ou une de ses sous-classes) :

```
if isinstance(mon_objet, X) : print("j'ai trouvé un X !")
```

■ Pour tester si une classe hérite d'une autre :

```
if issubclass(fille, mère) : print("elles sont parentes !")
```

# Gestion de la mémoire

■ Python gère automatiquement la mémoire et la destruction des objets

■ Deux méthodes sont employées en parallèle :

- Un compteur de référence (compte le nombre de référence pointant sur chaque objet, s'il descend à zéro => destruction)
- Un ramasse-miette qui, à intervalle régulier, détecte les objets non joignables
  - ✗ Similaire à Java
  - ✗ Il sert à casser les cycles de référence

# Gestion de la mémoire

■=> La destruction des objets peut être :

- ✗ Immédiate (dès qu'ils ne sont plus nécessaires, pour les objets qui ne sont pas pris dans un cycle)
- ✗ Retardée (quand le ramasse-miette se lance, pour les objets pris dans un cycle)

```
class Bombe(object):  
    def __del__(self):  
        print("BOUM !")
```

```
>>> bombe = Bombe()  
>>> bombe = None # Destruction immédiate  
BOUM !
```

# Méthodes spéciales

■ De nombreuses méthodes spéciales sont disponibles en Python, de la forme `__nom_de_méthode__()`

- ✖ `__init__(paramètres...)`: constructeur
- ✖ `__del__()`: destructeur (appelé lorsque l'objet est détruit)
- ✖ `__repr__()`: retourne une chaîne de caractères pour afficher au programmeur
- ✖ `__str__()`: retourne une chaîne de caractères pour afficher à l'utilisateur final
  - `print(objet)` appelle `__str__()` en priorité (à défaut, `__repr__()`)
  - `str(objet)` appelle `__str__()` en priorité (à défaut, `__repr__()`)
  - `repr(objet)` appelle `__repr__()` en priorité (à défaut, `__str__()`)
  - Dans le terminal Python, `__repr__()` est utilisé en priorité :

```
>>> objet  
<chaîne retournée par repr()>
```

# Quelques astuces...

■ Obtenir la classe d'un objet : `objet.__class__`

■ Obtenir le nom d'une classe : `MaClasse.__name__`

✗ Exemple :

```
class Machine(object) :
    def __repr__(self):
        return "<%s>" % self.__class__.__name__
```

```
class Ordinateur(Machine) :
    pass
```

```
>>> o = Ordinateur()
```

```
>>> o
```

```
<Ordinateur>
```

# Héritage

■ Python supporte l'héritage multiple : une classe peut hériter de plusieurs autres

✗ En cas de conflit (= plusieurs classes ayant des méthodes de même noms), les premières classes parentes sont prioritaires

■ L'héritage permet de créer des sous-classes (= sous-catégories) d'objets

✗ La sous-classe « hérite » (= récupère) les attributs et méthodes de la super-classe

✗ On parle aussi de « classe fille » et de « classe mère »

✗ Permet la réutilisation des attributs et méthodes, et d'éliminer des redondances

■ Pour dire qu'une sous classe B hérite d'une classe de base ou classe parente A:

```
>>> class B(A):
```

# Héritage

- Nous pouvons par exemple définir une classe **Mammifere()**, qui contient un ensemble de caractéristiques propres à ce type d'animal. À partir de cette classe parente, nous pouvons dériver une ou plusieurs classes filles, comme : une classe **Primate()**, une classe **Rongeur()**, une classe **Carnivore()**, etc., qui hériteront toutes les caractéristiques de la classe **Mammifere()**, en y ajoutant leurs spécificités.
- Au départ de la classe **Carnivore()**, nous pouvons ensuite dériver une classe **Belette()**, une classe **Loup()**, une classe **Chien()**, etc., qui hériteront encore une fois toutes les caractéristiques de la classe parente avant d'y ajouter les leurs.
- Exemple :

# Héritage

```
>>> class Mammifere(object):
...     caract1 = "il allaite ses petits ;"
>>> class Carnivore(Mammifere):
...     caract2 = "il se nourrit de la chair de ses proies ;"
>>> class Chien(Carnivore):
...     caract3 = "son cri s'appelle aboiement ;"
>>> mirza = Chien()
>>> print(mirza.caract1, mirza.caract2, mirza.caract3)
```

# Héritage

- Nb: les attributs utilisés dans cet exemple sont des attributs des classes (et non des attributs d'instances).
- L'instance **mirza** peut accéder à ces attributs, mais pas les modifier :

```
>>> mirza.caract2 = "son corps est couvert de poils ;" #1
```

```
>>> print(mirza.caract2) #2
```

son corps est couvert de poils ; #3

```
>>> fido = Chien() #4
```

```
>>> print(fido.caract2) #5
```

il se nourrit de la chair de ses proies ; #6

# Classes et héritages

## ■ Exercice animaux

✗ Créer une classe **Animal** avec :

- une méthode **\_\_str\_\_()** qui retourne une chaîne de la forme « un » + le nom de la classe (Animal ou une sous-classe)
- une méthode **déplacer()** qui affiche (print) le nom de l'animal (donnée par la méthode précédente) suivi de « se déplace »

✗ Créer une classe **AnimalVolant** héritant de **Animal** avec :

- une méthode **voler()** qui affiche le nom de l'animal + « vole »
- redéfinir la méthode **déplacer()** pour qu'elle appelle **voler()**

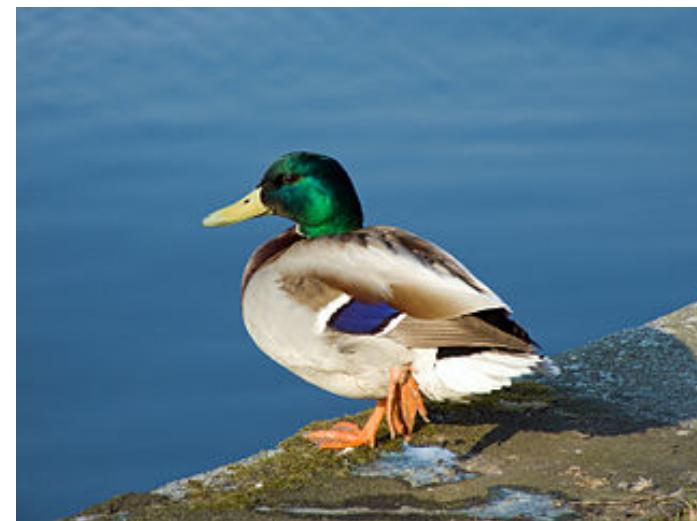
✗ De même, créer une classe **AnimalNageant** avec une méthode **nager()**

✗ et une classe **AnimalMarchant** avec une méthode **marcher()**

# Classes et héritages

## ■ Exercice animaux (suite)

- ✖ Créer une classe **Pigeon** qui est un animal qui peut marcher et voler ; son mode de déplacement « par défaut » est le vol (**déplacer()** doit donc appeler **voler()**)
- ✖ Créer un **Pigeon** et le déplacer. Tester le programme
- ✖ Créer une classe **Canard** qui est un animal qui peut marcher, nager et voler ; son mode de déplacement « par défaut » est la nage
- ✖ Créer un **Canard** et le déplacer. Tester le programme



# Changement de classe

■ En Python, la classe d'un objet peut changer !

■ Il suffit de modifier `__class__`

✗ Exemple :

```
mon_objet = AncienneClasse()
```

```
mon_objet.__class__ = NouvelleClasse
```

■ Exercice animaux (suite et fin)

✗ Créer une classe **Papillon** qui est un animal volant

✗ Créer une classe **Chenille** qui est un animal marchant

- Ajouter à cette classe une méthode **metamorphoser()** qui change la classe de la **Chenille** par la classe **Papillon**

✗ Créer une **Chenille**, la déplacer, la métamorphoser puis la déplacer de nouveau. Tester le programme

# Polymorphisme

- Le polymorphisme permet d'attribuer des comportements différents à des objets dérivant les uns des autres, ou au même objet ou en fonction d'un certain contexte
- Les méthodes des classes parentes peuvent être redéfinies dans les classes filles
- Dans ce cas, la méthode de la classe fille peut appeler celle de la classe parente :

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

```
class PlatEnSauce(Plat) :  
    def __init__(self, nom, sauce) :  
        super().__init__(nom)           Appelle Plat.__init__()  
        self.sauce = sauce
```

- Les appels via `super()` peuvent retourner une valeur

```
valeur_de_retour_de_la_classe_parente = super().ma_méthode()
```

- `super()` gère le cas de l'héritage multiple

# Polymorphisme

■ Exemple : classes de cylindres et de cônes : *Classes dérivées - Polymorphisme*

**class Cercle(object):**

**def \_\_init\_\_(self, rayon):**

**self.rayon = rayon**

**def surface(self):**

**return 3.1416 \* self.rayon\*\*2**

**class Cylindre(Cercle):**

**def \_\_init\_\_(self, rayon, hauteur):**

**Cercle.\_\_init\_\_(self, rayon)**

**self.hauteur = hauteur**

**def volume(self):**

**return self.surface()\*self.hauteur**

*# la méthode surface() est héritée de la classe parente class*

# Polymorphisme

■ Exemple : classes de cylindres et de cônes : *Classes dérivées - Polymorphisme*

**class Cone(Cylindre):**

**def \_\_init\_\_(self, rayon, hauteur):**

        Cylindre.\_\_init\_\_(self, rayon, hauteur)

**def volume(self):**

        return Cylindre.volume(self)/3

*# cette nouvelle méthode volume() remplace celle que*

*# l'on a héritée de la classe parente (exemple de polymorphisme)*

**# Programme test :**

cyl = Cylindre(5, 7)

print("Surf. de section du cylindre =", cyl.surface())

print("Volume du cylindre =", cyl.volume())

co = Cone(5,7)

print("Surf. de base du cône =", co.surface())

print("Volume du cône =", co.volume())

# Héritage

## ■ Exercice : le musée virtuel 2

### ✗ Créer une classe **Personne** avec

- Deux attributs : nom et prénom
- Une méthode `__str__()` qui retourne une chaîne de la forme :  
*"prénom nom"*
- Une méthode `__repr__()` qui retourne une chaîne de la forme :  
*"<Classe prénom nom>"*

### ✗ Créer une classe **Utilisateur** qui hérite de **Personne**

- Utilisateur possède deux attributs supplémentaires :
  - ➊ `mot_de_passe`
  - ➋ `login`, attribut optionnel qui vaut "*prénom\_nom*" s'il n'est pas renseigné

### ✗ Créer trois sous-classes d'**Utilisateur** : **Visiteur**, **Conservateur**, **Administrateur**

# Polymorphisme

## ■ Exercice : le musée virtuel 2 (suite)

### ✗ Créer une autre sous-classe de Personne : Artiste

- **Artiste** possède un attribut **oeuvres** (la liste des œuvres de cet artiste)
- Une méthode **add\_œuvre(œuvre)** pour ajouter une œuvre à l'artiste
- Une méthode **\_\_repr\_\_()** qui retourne une chaîne de la forme :

"<*Classe prénom nom, xxx œuvre(s)*>"

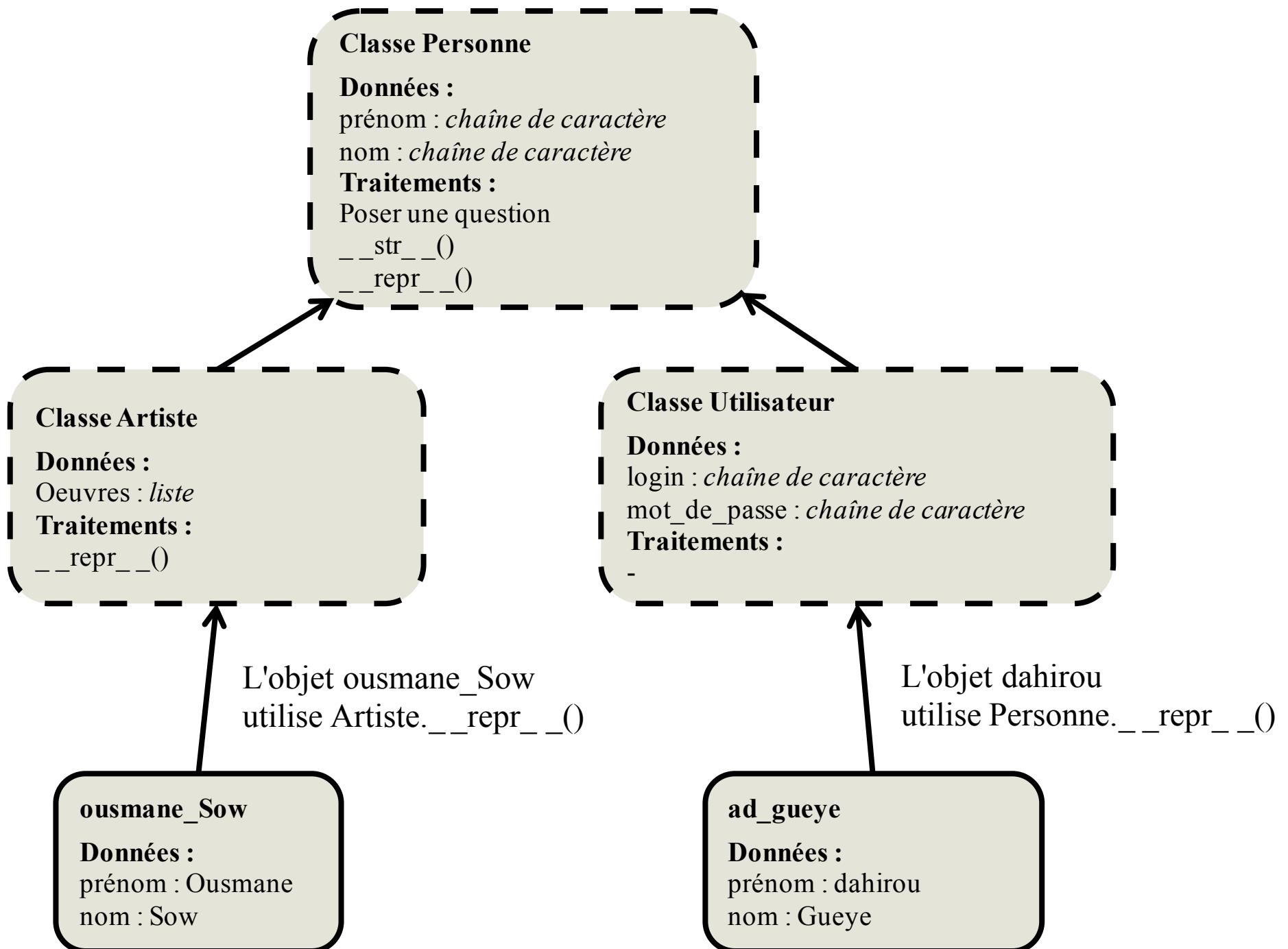
"<Artiste Ousmane Sow, 1 œuvre(s)>" (par exemple)

# Polymorphisme

- Modifier la classe **Oeuvre** pour :

- Que l'attribut **artiste** soit un **Artiste**
- Une œuvre anonyme a **None** pour artiste
- Dans le constructeur de **Oeuvre**, appeler la méthode **add\_oeuvre()** de l'artiste (si l'artiste n'est pas **None**)
- Ajouter une méthode **\_\_repr\_\_()** qui retourne une chaîne :
  - "*<Classe titre artiste=<Artiste prénom nom>>*"
- Ajouter une méthode **\_\_str\_\_()** qui retourne une chaîne :
  - "*titre de artiste*" ou "*titre (anonyme)*" si l'artiste est inconnu (**None**)

# Polymorphisme



# Classes et objets



# Chapitre 9:

## Modules/Packages

# Modules

- Les modules sont des fichiers qui regroupent des ensembles de fonctions
- Autrement, les modules Python permettent d'organiser un programme en différentes parties
  - ✗ Similaires aux packages de Java
  - ✗ L'importation du module est nécessaire pour accéder à son contenu
    - Par exemple pour créer des objets à partir des classes définies dans le module  
`objet = mon_module.MaClasse()`
    - En revanche, l'importation n'est pas nécessaire pour utiliser des objets déjà créés  
`def ma_fonction(objet) :  
 objet.ma_méthode() # pas besoin d'importer le module où est définie la classe`
  - 1 module = 1 fichier de code .py

# Modules

- Les modules peuvent aussi être organisés de manière hiérarchique
  - ✗ `module.sous_module.sous_sous_module`
  - ✗ Cela correspond à des sous-réertoires :
    - `module/sous_module/sous_sous_module.py`
- Python propose un grand nombre de modules de base
- D'autres modules peuvent être téléchargés en ligne : <http://pypi.python.org>
- Il est possible de créer ses propres modules

# Importer des modules

## ■ Importer le module

- ✗ Import d'un module ou d'un sous-module, le contenu est ensuite accessible avec la notation pointée :

```
import math  
print(math.sqrt(2.0))  
# Racine de 2
```

- ✗ Import de plusieurs modules en une ligne :

```
import sys, os.path
```

## ■ Import d'un sous-module :

```
import os.path  
from os import path
```

## ■ Importer du contenu du module

- ✗ Importer une fonction / une classe

```
from math import sqrt  
print(sqrt(2.0))
```

- ✗ Importer tout le contenu

```
from math import *  
print(sqrt(2.0))
```

- ✗ Attention aux conflits si plusieurs modules sont importés avec des fonctions de même noms !

- Le dernier importé l'emporte

# Importer des modules

■ Les importations réalisées au début du fichier de code sont valables dans tout le fichier :

```
import math  
[...]  
print(math.sqrt(2.0))
```

■ Les importations peuvent aussi être réalisées dans une fonction :

```
def ma_fonction(x) :  
    import module_long_à_charger  
    return module_long_à_charger.traite(x)
```

✗ Le module ne sera chargé qu'à l'exécution de la fonction

- Si elle n'est jamais exécuté, le module n'est pas chargé

# Créer son premier script python

- Lorsque du code est enregistré dans un fichier exécutable on parle de script .
- Créons un fichier avec l'extension .py de nom par exemple test.py dans le dossier que vous voulez

```
#!/usr/bin/python 2.7  
#-*- coding : utf-8 -*-
```

```
print("Hello world")
```

- La première ligne indique qu'il s'agit de code python.
- La deuxième ligne indique le type d'encodage utilisé

- Exécuter un script

```
python /chemin_vers_votre_script/test.py
```

# Importer des fonctions d'autres fichiers

■ Créons un autre fichier que nous nommerons func.py dans le même dossier que le fichier test.py

- func.py

```
# coding: utf-8
def ajoute_un(v):
    return v + 1
```

- test.py

```
# coding: utf-8
from func import *
age = input("Quel est votre age? : ")
print("Vous avez %d ans" % age)
age_plus_un = ajoute_un(age)
print("Dans un an vous aurez %d ans" % age_plus_un)
```

# Modules existants

## ■ Quelques modules intégrés à Python et souvent utiles :

✗ **sys** : module « système » (au sens Python)

- **sys.argv** : listes des arguments passés en ligne de commande
- **sys.path** : listes des répertoires où sont cherchés les modules

✗ **os** : module « operating système »

- **os.listdir(répertoire)** : liste les fichiers présents dans le répertoire

✗ **os.path** : gestion des noms de fichiers

- **os.path.join(répertoire, sous-répertoire,..., fichier)** : joint les noms de répertoires passés en paramètres avec des / ou des \ (selon l'OS)
- **os.path.exists(fichier)** : teste si un fichier existe

# Modules existants

■ Quelques modules intégrés à Python et souvent utiles :

- ✗ **glob** : recherche de fichiers avec des motifs du type « \*.py »
- ✗ **tempfile** : création de fichiers temporaires
- ✗ **math** : fonctions mathématiques (log, sin, cos, tan, sqrt,...)
- ✗ **random** : génération de nombres aléatoires
- ✗ **re** : expressions régulières (=recherche complexe dans des chaînes de caractères)
- ✗ **numpy**
- ✗ **opencv-python**

# Modules existants

## ■ Quelques modules intégrés à Python et souvent utiles :

- ✗ **io** : gestion des flux (input / output)
- ✗ **gzip, bz2, zipfile** : compression / décompression GZip, BZ2, ZIP
- ✗ **socket** : réseau bas niveau
- ✗ **ssl** : *secure socket layer*
- ✗ **urllib** : gestion des URL, téléchargement de fichiers
- ✗ **xml** : parseur XML
- ✗ **html** : parseur HTML
- ✗ **json** : parseur / sérialiseur JSON
- ✗ **email** : manipulation des emails (lecture, etc)
- ✗ **time** : gestion du temps

La doc des modules :

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

# Modules existants

## ■ Quelques modules intégrés à Python et souvent utiles :

✗ **date** : gestion des dates

✗ **locale** : localisation

`import locale`

`locale.setlocale(locale.LC_ALL, "fr_FR") # en français`

`locale.setlocale(locale.LC_ALL) # langue par défaut de l'ordinateur`

✗ **gettext** : système de traduction GNU Gettext

# Modules disponibles en ligne

■ Installation des modules avec **pip** (dans un terminal Unix) :

✗ **pip install nom\_du\_module**

■ Quelques modules intéressants :

✗ Pillow (PIL, Python Imaging Library) : gestion des images bitmap (JPEG, PNG, ...)

```
>>> import PIL.Image
```

```
>>> image = PIL.Image.open("/home/da/image.jpeg")
```

```
>>> print(image.size)
```

```
(2176, 3152)
```

```
>>> image.thumbnail((100, 100))
```

```
>>> image.save("/home/da/image.jpeg")
```

✗ PyQt : interface graphique Qt5

✗ Tkinter: interface graphique

# Modules

## ■ Exercice Module 1 :

- ✗ Le répertoire « images\_musee » contient 4 images au format JPEG
- ✗ Écrire un programme qui affiche le chemin complet de chaque fichier image (.jpeg) dans ce répertoire, par exemple :

Fichier image trouvé : /home/da/cours/prog\_python\_M1/images\_musee/joconde.jpg

Fichier image trouvé : /home/da/cours/prog\_python\_M1/images\_musee/venus.jpg

Fichier image trouvé : /home/da/cours/prog\_python\_M1/images\_musee/liberte.jpg

- ✗ On utilisera pour cela les modules os et os.path :

- **os.listdir(répertoire)**
- **os.path.splitext(fichier)**
- **os.path.join(repertoire, fichier)**

# Modules

## ■ Exercice Module 2 :

- ✗ Installer le module Pillow (PIL)
- ✗ Modifier le programme précédent pour qu'il crée une miniature de 400 pixels de haut de chaque image trouvée (on réduira la largeur proportionnellement)

# Package

- Lorsque l'on cherche à regrouper des modules, on parle de package
- Pour créer votre propre package, commencez par créer dans le même dossier que votre programme - un dossier portant le nom de votre package. Dans notre exemple, nous le nommerons " **utils** ".
- Dans ce dossier, créons le fichier suivant: **`__init__.py`**, cela indique à python qu'il s'agit d'un package . Ce fichier peut être vide, seule sa présence est importante.
- Ensuite créons un fichier toujours dans ce répertoire **utils** que nous nommerons par exemple " **operations.py** "

# Package

## ■ Contenu du dossier de votre projet

- Dossier utils
- test.py
- func.y

## ■ Contenu du dossier utils:

- init\_.py
- operations.py

## ■ Editons le fichier operations.py et créons une nouvelle fonction

```
# coding: utf-8
def ajoute_deux(v):
    return v + 2
```

# Package

- Puis ajoutons un appel vers cette fonction dans le fichier test.py

```
# coding: utf-8

from func import *

from utils.operations import ajoute_deux

age = input("Quel est votre age? : ")

print("Vous avez %d ans" % age)

age_plus_un = ajoute_un(age)

print("Dans un an vous aurez %d ans" % age_plus_un)

age_plus_deux = ajoute_deux(age)

print("Dans deux ans vous aurez %d ans" % age_plus_deux)
```

# Package

## ■ Remarques:

- Tout d'abord on importe un package avec les mots clé **from** et **import**
- ensuite pour appeler une fonction précise, on passe par la hiérarchie suivante:  
`from package.module import fonction`
- Si vous voulez importer toutes les fonctions d'un module, vous pouvez indiquer une étoile \*

# Créer ses propres modules

## ■ Un fichier .py => un module

module\_simple.py (fichier) → import module\_simple

module/ (répertoire)

  \_\_init\_\_.py (fichier) → import module

  sous\_module1.py (fichier) → import module.sous\_module1

  sous\_module2.py (fichier) → import module.sous\_module2

  sous\_module3/ (répertoire)

    \_\_init\_\_.py (fichier) → import module.sous\_module3

    sous\_sous\_module1.py (fichier) → import module.sous\_module3.sous\_sous

## ■ Le fichier \_\_init\_\_.py est nécessaire pour que le répertoire soit considéré comme un package

✗ Attention, importer un sous-module (**module.sous\_module1** par ex) importe automatiquement le module (ici, **module**)

- Attention aux dépendances cycliques si **module** importe **sous\_module1** !
- Il est souvent préférable de laisser \_\_init\_\_.py vide pour éviter ces problèmes

# Créer ses propres modules

■ Les fichiers doivent être placés dans un des répertoires où Python cherche les modules :

✗ Variable d'environnement **PYTHONPATH** :

- Équivalent au **CLASSPATH** de Java
- Créer un répertoire **src** où l'on placera ses codes sources
- Définir la variable d'environnement sous Unix :

**export PYTHONPATH=/home/da/src**

- sous Windows :

**set PYTHONPATH=C:\\src**

# Créer ses propres modules

✗ En Python, les répertoires où les modules sont recherchés sont disponibles dans la liste **sys.path** (modifiable)

- Cette liste peut être modifiée pour ajouter ou supprimer des répertoires

```
>>> import sys  
>>> sys.path  
[', '/home/da/src', '/usr/lib/python35.zip', '/usr/lib/python3.5',  
 '/usr/lib/python3.5/plat-linux', '/usr/lib/python3.5/lib-dynload',  
 '/usr/lib/python3.5/site-packages', '/usr/lib/python3.5/site-packages/gtk-  
 2.0']
```

# Créer ses propres modules

- Dans un module Python, la variable spéciale `__file__` contient le nom de fichier du module
  - ✗ Pratique pour aller chercher d'autres fichiers dans le même répertoire par exemple

```
import os.path  
  
répertoire_parent = os.path.dirname(__file__)  
  
fichier = os.path.join(répertoire_parent, "autre_fichier.txt")
```

- La variable spéciale `__name__` contient le nom du module
  - ✗ Si le fichier n'est pas un module importé mais celui exécuté par Python, `__name__` vaut "`__main__`"

# Créer ses propres modules

## ■ Exercice musée virtuel 5 - module

Nous allons reprendre l'exercice du musée virtuel

- ✗ Crée un répertoire **src**
- ✗ Crée un sous-répertoire **musee\_virtuel**
- ✗ Dans ce répertoire, créer un fichier **\_\_init\_\_.py** vide
- ✗ Dans ce répertoire, créer un fichier **personne.py** qui contiendra le modèle objet des personnes :
  - la classe Personne et ses descendantes (filles, petites-filles, etc)
- ✗ Dans ce répertoire, créer un fichier **musee.py** qui contiendra le modèle objet du musée virtuel :
  - les classes Musée, Salle, Ouvre, SalleDécoré
  - ce module **musee** a-t-il besoin d'importer le module **personne** ?
- ✗ Dans ce répertoire, créer un fichier **test.py** qui importera les deux sous-modules précédents et créera le musée avec les 4 oeuvres
- ✗ Vérifier que le programme (**test.py**) fonctionne bien