MOOC Python

# Introduction

progiciels = ensemble de plusieurs logiciels pouvant fonctionner ensemble, principalement utilisés dans le monde professionnel

IOT : Internet of Things

python = **langage de programmation "interprété"** -> instructions transcrites en langage machine (binaire) au fur et à mesure (= **langage script**)

différent des "**langages "compilés**" : il faut un logiciel spécialisé pour transformer le code en langage machine(ex : C++) (-> compilation)

langages interprété sont plus simples à utiliser (pas d'étape de compilation), et fonctionne qque soit l'environnement (mac, linux...) (= **portabilité**).

mais traduction au fur et à mesure peut ralentir l'exécution

"interpréteur de commande" = permet de tester directement du code. C'est la boite noire qui s'ouvre quand on ouvre python

pour écrire des commentaires : #blablabla

si le programme tourne dans le vide, faire ctrl + C pour l'arrêter

# opérateurs

/ : division

// : partie entière de la division

% : reste de la division

= affecter une valeur à une variable

nb +=3 équivaut à nb = nb + 3 (on a de même \*=, ou -=, ou /=...)

permuter des valeurs ; a,b = b,a

pour attribuer des valeurs à a,b,c, on peut aussi écrire a,b,c = 3,5,7

tests :

|  |  |
| --- | --- |
| < | Strictement inférieur à |
| > | Strictement supérieur à |
| <= | Inférieur ou égal à |
| >= | Supérieur ou égal à |
| == | Égal à |
| != | Différent de |

# types de données

Python différencie les entiers des nombres à virgule flottante: surtout pour une question de place en mémoire mais, pour un ordinateur, les opérations que l'on effectue sur des nombres à virgule ne sont pas les mêmes que celles sur les entiers, et cette distinction reste encore d'actualité de nos jours.

int = integer = nombre entier

float = nombres flottants = nombres à virgule

str = chaine de caractères : entre " ", ' ', ou """ """

bool = booléen (True et False)

on peut marquer 'j/'aime le python' : avec le /, l'apostrophe de J'aime n'est pas pris en compte

/n : pour passer à la ligne. Mais ne marche pas avec mon python...

(a,b,c) : collection (les virgules ne sont pas obligatoires)

Une des grandes puissances de Python est qu'il comprend automatiquement de quel type est une variable et cela lors de son affectation.

# fonctions basiques

print(a,b,c) affiche a, b, puis c

type(a) affiche le type de a

* boucle **if** :

if a>b:

print("a sup b")

elif a = b:

print ("a = b")

else:

print("b sup a")

puis ensuite il faut faire 2 enter pour que ça lance.

indentation (alinéas) est importante !!!

a>b est un **prédicat**

la ligne avec print est un **bloc d'instruction**

* on a la même syntaxe pour **while**

**continue** : on retourne au while sans faire les lignes suivantes

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

* opérateurs **and et or** :

if a == 3 or b ==5: #etc

if not b==5 : #etc

if (b==5) is not True: #etc

* Pour que l'utilisateur puisse entrer lui-même une donnée :

an = input("saisissez une année:") -> c'est une chaine de caractère

année = int(an)

* **boucle for :**

for lettre in "test":

print(lettre)

"in" peut être utilisé dans d'autres contextes :

if e in "une phrase":

print ("la phrase contient un e")

# Créer des fonctions

**modularité :** regrouper dans des fonctions des parties de notre code que nous serons amenés à réutiliser, et regrouper les fonctions ayant un rapport entre elles dans un fichier, pour constituer un module.

**def** nom\_de\_la\_fonction(parametre1, parametre2, parametre3, parametreN):

pour avoir valeurs par défaut, par exemple :

**def** nom\_de\_la\_fonction(parametre1 = 10, parametre2=0, parametre3=4, parametreN=5):

on peut taper un texte d'aide après les :; avec """ blablabla """

**def** nom\_de\_la\_fonction(parametre1, parametre2, parametre3, parametreN):

"""Fonction affichant la table de multiplication par nb

de 1\*nb à max\*nb

(max >= 0)"""

-> c'est une "docstring" = chaine d'aide

on l'appelle en faisant help(nom\_de\_la\_fonction)

-> on peut appeler les paramètres par leur nom

nom\_de\_la\_fonction(parametre3 = 5):

tous les autres paramètres restent par défaut

**signature** = éléments qui permettent au langage d'identifier ladite fonction. En python, c'est juste le nom de la fonction

-> On ne peut pas appeler 2 fonctions pareil

fonction "**return**" : permet de retourner la valeur

fonctions lambda : fonctions définies de manière très rapide

exemple : f = lambda a,b: a+b

il faut la stocker explicitement qque part

puis ensuite, f(4,5)

# Modules

importer des modules : (exemple avec le module math)

>>> import math

Puis, pour appeler une fonction du module : par exemple

math.sqrt(16)

on peut regarder les fonctions qu'il y a dans le module en tapant

help(math)

(Q pour revenir à l'interpréteur)

on peut aussi écrire

import math as m

En vérité, quand vous tapez import math, cela crée un espace de noms dénommé « math », contenant les variables et fonctions du module math. Quand vous tapez math.sqrt(25), vous précisez à Python que vous souhaitez exécuter la fonction sqrt contenue dans l'espace de nomsmath. Cela signifie que vous pouvez avoir, dans l'espace de noms principal, une autre fonctionsqrtque vous avez définie vous-mêmes. Il n'y aura pas de conflit entre, d'une part, la fonction que vous avez créée et que vous appellerez grâce à l'instruction sqrt et, d'autre part, la fonctionsqrtdu modulemathque vous appellerez grâce à l'instruction math.sqrt.

-> idem pour les variables : pas de risque de conflit

on peut aussi n'importer qu'une seule fonction du module math :

from math import sqrt

-> dans ce cas là, on l'appelle uniquement par sqrt, et non pas par math.sqrt...

et si on veut toutes les importer comme ça :

from math import \*

quand on crée un dossier avec une fonction, on peu marquer au tout début (avant même le def) une docstring (ligne d'aide) """ le module contient..."""

# Enregistrer son code

On peut enregistrer son code tout simplement en le copiant-collant dans un fichier texte et en rajoutant l'extension .py dans le nom du fichier.

Si, quand on double clique sur le fichier, il se ferme direct, c'est parce qu'il s'exécute très rapidement puis se ferme à la fin de l'exécution. -> il faut le mettre sur pause.

import os # module os qui dispose de variables et de fonctions utiles pour dialoguer avec votre système d'exploitation

puis à la fin du code :

os.system("pause")

pour créer un module, il suffit d'enregistrer un fichier code avec toutes les fonctions que ce module contient... (on appelle ce fichier multipli par exemple).

-> dans un autre fichier (MAIS contenu dans le même dossier qui ne possède que des fichiers python), on peut l'appeler :

"import Nom-Fonction from multipli"

On peut aussi tester le module directement dans le fichier le décrivant. -> Exécuter le module comme un programme à lui tout seul.

après la description de la fonction :

# test de la fonction table

if \_\_name\_\_ == "\_\_main\_\_":

table(4)

os.system("pause")

-> quand on lance le fichier décrivant le module, le test s'exécute. Mais quand on importe le module dans un autre fichier, il ne s'affiche pas.

if \_\_name\_\_ == "\_\_main\_\_":

-> veut dire "si le fichier appelé est le fichier exécuté", en d'autres termes, si j'ai cliqué sur le fichier contenant ce code, et qu'il n'est pas appelé par un autre programme

# Packages

package : regroupe des modules entre eux

-> on peu faire toutes les asborescences de dossiers qu'on veut, pour réunir des modules, réunir des fonctions...

on peut importer des packages exactement de la même manière que les modules

dans ce cas, pour importer un module ou un package, il faut préciser le chemin :

import nom\_bibliotheque.evenements.clavier

-> importe le module clavier compris dans un sous-package evenement lui même compris dans un package nom\_bibliothèque

# Exceptions

-> erreurs que peut rencontrer Python en exécutant un programme

>>> # Exemple classique : test d'une division par zéro

>>> variable = 1/0

Traceback (most recent call last):

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

ZeroDivisionError: int division or modulo by zero

* ZeroDivisionError : le type de l'exception ;
* int division or modulo by zero : le message qu'envoie Python pour vous aider à comprendre l'erreur qui vient de se produire.

on peut tester qque chose, pour voir si un erreur s'affichera ou pas (exemple pour le programme bissextile : si un utilisateur tape qque chose qui n'est pas convertible en chiffre)

**try**:

# Bloc à essayer

except:

# Bloc qui sera exécuté en cas d'erreur

exemple :

annee = input()

try: # On essaye de convertir l'année en entier

annee = int(annee)

except:

print("Erreur lors de la conversion de l'année.")

après le except on peu préciser le type d'erreur que l'on cherche.

ex :

except NameError:

print("La variable numerateur ou denominateur n'a pas été définie.")

-> ne traite que des exceptions NameError, et on peut mettre d'autres blocs except après.

ou encore :

except type\_de\_l\_exception **as** exception\_retournee:

print("Voici l'erreur :", exception\_retournee)

!! attention, essayer au maximum de spécifier les types d'erreurs possible.... Ne pas mettre de except tout seul, sinon, TOUTES les erreurs seront traitées de la même manière (et certaines ont nécessaire)

on peut coupler les except avec des **else** (-> que faire si aucune erreur ne survient ?)

**finally** permet d'exécuter du code après un bloc try, *quelle que soit le résultat de l'exécution dudit bloc*.

try:

# Test d'instruction(s)

except type\_de\_l\_exception: # Rien ne doit se passer en cas d'erreur

pass

**assert**:

Les assertions sont un moyen simple de s'assurer, avant de continuer, qu'une condition est respectée. En général, on les utilise dans des blocs try … except.

assert test

-> si le test est faux, alors ça lève une erreur "AssetionError" -> permet de générer une erreur là où il n'y en aurait pas forcément eu.

annee = input("Saisissez une année supérieure à 0 :")

try:

annee = int(annee) # Conversion de l'année

assert annee > 0

except ValueError:

print("Vous n'avez pas saisi un nombre.")

except AssertionError:

print("L'année saisie est inférieure ou égale à 0.")

**lever une exception** :

raise TypeDeLException("message à afficher")

# Les chaines de caractère

**un objet est une structure de données, comme les variables, qui peut contenir elle-même d'autres variables et fonctions**

si on a un objet chaine (par exemple une chaine de caractère), et qu'on veut appliquer une méthode de la classe de l'objet :

chaine.() -> pour tout mettre en minuscule par exemple

OU : str.lower(chaine) -> parce que lower est une méthode de string

avant le nom de la méthode, on peut donc mettre le nom de la classe ou le nom de l'objet

On peut cumuler les noms des fonctions :

chaine.lower().capitalize().center(40)

Pour voir toutes les méthodes d'une classe d'objet, taper par exemple :

help(str)

fonction format :

>>> prenom = "Paul"

>>> nom = "Dupont"

>>> age = 21

>>> print("Je m'appelle {0} {1} et j'ai {2} ans.".format(prenom, nom, age))

Je m'appelle Paul Dupont et j'ai 21 ans.

-> Fonction qui dans une chaine de caractère, remplace les {0} {1} {2} ou plus par les éléments qui sont rentrés dans une collection

-> on n'est pas obligés de les appeler dans l'ordre

-> Si c'est dans l'ordre, on n'est pas obligés de préciser les numéros dans l'accolade

au lieu de mettre numéros entre accolades, on peut donner des noms aux places :

# formatage d'une adresse

adresse = """

{no\_rue}, {nom\_rue}

{code\_postal} {nom\_ville} ({pays})

""".format(no\_rue=5, nom\_rue="rue des Postes", code\_postal=75003, nom\_ville="Paris", pays="France")

print(adresse)

concatenation : avec des +

ex : "bonjour" + ""+ "Paul" = "bonjour Paul"

mais on ne peut pas concaténer différents types de variables

nous avons dit que Python était un langage à **typage dynamique**, ce qui signifie qu'il identifie lui-même les types de données et que les variables peuvent changer de type au cours du programme.

accéder aux caractères d'une chaine

chaine[0] -> appelle le premier caractère

chaine[1] -> appelle le deuxième caractère

chaine [-1] -> appelle le dernier caractère

chaine[0:4] OU chaine[:4]-> appelle les 4 premiers caractères (place comprise entre 0 et 3)

chaine[4:] OU chaine[4:len(chaine)] -> appelle tous les caractères à partir du 5eme

chaine[4:6] : accède aux caractères 4 et 5

chaine[0:8:2] : les 8 premiers caractères, avec un pas de 2

len(chaine) : longueur de la chaine

Pour rechercher/remplacer, nous avons à notre disposition les méthodes count, find et replace, à savoir « compter », « rechercher » et « remplacer »

# Listes

Les listes sont des séquences. En fait, leur nom est plutôt explicite, puisque ce sont des objets capables de contenir d'autres objets de n'importe quel type. On peut avoir une liste contenant plusieurs nombres entiers (1, 2, 50, 2000 ou plus, peu importe), une liste contenant des flottants, une liste contenant des chaînes de caractères... et une liste mélangeant ces objets de différents types.

Toutefois, la structure d'une liste fait que chaque objet *a sa place* et que l'ordre compte.

pour remplacer l'élément d'une liste, on peut mettre :

liste[0] = 2

(alors que ce n'est pas possible avec str, il faut utiliser la fonction replace)

pour ajouter un élément à la fin d'une liste, on peut utiliser append :

liste.append(6)

OU list.append(liste, 6)

!!!! aucune des méthodes de chaînes ne modifie l'objet d'origine mais elles renvoient toutes un nouvel objet (chaine.upper, chaine.lower...), qui est la chaîne modifiée. Ici c'est le contraire : les méthodes de listes ne renvoient rien mais modifient l'objet d'origine.

!! append ne renvoie rien. donc si on marque :

liste2 = liste1.append(-15)

on a liste2 est égal à None

none : objet vide de python

pour insérer un objet dans à la ieme place :

liste.insert(i, objet)

liste1.extend(liste2) : rajoute la liste 2 à la suite de la liste 1

OU : liste3 = liste1 + liste2

Pour supprimer une variable : del

-> del liste[3] -> supprime le quatrième élément de la variable

on peut aussi utiliser "remove", qui est utilisé avec le nom de l'objet à retirer :

essaiListe.remove("objet à la place 3")

-> !! ne retire que la première occurrence "objet à la place 3" dans la liste

pour parcourir une liste, on peut utiliser une boucle for, une boucle while, ou la fonction **enumerate** (couplée à une boucle for): renvoie une liste de 2 éléments : une valeur et la place qu'elle occupe :

for elt in enumerate(ma\_liste):

... print(elt)

...

(0, 'a')

(1, 'b')

(2, 'c')

(3, 'd')

(4, 'e')

(5, 'f')

(6, 'g')

(7, 'h')

on peut aussi écrire for i,j in enumerate(ma\_liste)

-> si on veut différencier la valeur de sa place de manière plus simple

suivi par exemple de:

print("À l'indice {} se trouve {}.".format(i, j))

# Tuples

tuples : comme des listes, sauf qu'on ne peut pas les modifier

identifiés par des parenthèses

ex\_tuple = ()

ex\_tuple = (1,)

ex\_tuple = 1,2

ex\_tuple = 1,

a,b = 1,2

rappel si on veut permuter des valeurs :

a, b = b, a

Certains cours encouragent à toujours placer des parenthèses autour des tuples quand on les utilise. Pour ma part, je pense que, si vous gardez à l'esprit qu'il s'agit de tuples, que vous n'avez aucune peine à l'identifier, cela suffit. Si vous faites la confusion, mettez des parenthèses autour des tuples en toutes circonstances.

# Entre chaines et listes

split :

permet de convertir une chaine de caractère en une liste, selon un séparateur qu'on peut préciser (par défaut un espace)

-> le met dans une autre variable

à l'inverse, la fonction join permet de concaténer les éléments d'une liste pour en faire une chaine de caractère. syntaxe bizarre :

"e".join(sep)

si on veut créer une fonction avec un nombre non fixé de paramètres (print par exemple):

avec une asterisque :

def fonction(\*parametres):

OU def donction(paramètres, ...)

on peut aussi fixer des paramètres obligatoires et des paramètres facultatifs:

def fonction\_inconnue(nom, prenom, \*commentaires)

dans ce cas, \*commentaires est un tuple

# Compréhension de listes

-> permet de filtrer ou modifier une liste très simplement

>>> liste\_origine = [0, 1, 2, 3, 4, 5]

>>> **[**nb \* nb **for** nb **in** liste\_origine**]**

[0, 1, 4, 9, 16, 25]

crochets correspondent à une instruction sur une liste

-> il faut capturer le résultat dans une nouvelle variable

on peut coupler avec un for :

>>> liste\_origine = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> [nb for nb in liste\_origine if nb % 2 == 0]

[2, 4, 6, 8, 10]

pour trier, on peut utiliser :

listeTriee = sorted(liste)

si on veut trier par ordre décroissant :

listeTriee = sorted(liste, reverse = True)

OU sort(liste) -> mais dans ce cas, ça modifie la liste directement

# Set

mon\_dictionnaire = {'pseudo', 'mot de passe'}

Avec une telle instruction, ce n'est pas un dictionnaire que vous créez, mais un set.

Un set (ensemble) est un objet conteneur (lui aussi), très semblable aux listes sauf qu'il ne peut contenir deux objets identiques. Vous ne pouvez pas trouver deux fois dans un set l'entier 3 par exemple. Je vous laisse vous renseigner sur les sets si vous le désirez.

# Dictionnaires

Les dictionnaires sont des objets pouvant en contenir d'autres, à l'instar des listes. Cependant, au lieu d'héberger des informations dans un ordre précis, ils associent chaque objet contenu à une clé.

pour accéder au contenu d'un dictionnaire, on n'utilise pas un indice, mais une clé (généralement sous forme de symbole, mais ça peut aussi être des nombres, des tuples...)

créer un dictionnaire :

exemple = dict()

OU exemple = {}

pour ajouter une clé et une valeur dans mon dictionnaire :

mon\_dictionnaire["pseudo"] = "Prolixe"

mon\_dictionnaire["mot de passe"] = "monmotdepasse"

pour accéder à la valeur : mon\_dictionnaire["mot de passe"]

si elle n'existe pas, exception de type KeyError

pour supprimer des entiers :

del

OU pop (methode de dictionnaire) (-> renvoie aussi la valeur qui est supprimée)

ex: mon\_dictionnaire.pop("pseudo")

on peut aussi mettre des fonctions dans un dictionnaire

* On commence par définir deux fonctions, fete et oiseau (pardonnez l'exemple).
* On crée un dictionnaire nommé fonctions.
* On met dans ce dictionnaire les fonctions fete et oiseau. La clé pointant vers la fonction est le nom de la fonction, tout bêtement, mais on aurait pu lui donner un nom plus original.
* On essaye d'accéder à la fonction oiseau en tapant fonctions[« oiseau »]. Python nous renvoie un truc assez moche, <function oiseau at 0x00BA5198>, mais vous comprenez l'idée : c'est bel et bien notre fonction oiseau. Toutefois, pour l'appeler, il faut des parenthèses, comme pour toute fonction qui se respecte.
* En tapant fonctions["oiseau"](), on accède à la fonction oiseau et on l'appelle dans la foulée.

>>> def fete():

... print("C'est la fête.")

...

>>> def oiseau():

... print("Fais comme l'oiseau...")

...

>>> fonctions = {}

>>> fonctions["fete"] = fete # on ne met pas les parenthèses

>>> fonctions["oiseau"] = oiseau

>>> fonctions["oiseau"]

<function oiseau at 0x00BA5198>

>>> fonctions["oiseau"]() # on essaye de l'appeler

Fais comme l'oiseau...

>>>

# Parcours d'un dictionnaire

on peut les parcourir avec une boucle for:

for cle in fruits:

... print(cle)

cela marche, mais pour etre vraiment précis, il faudrait marquer :

for cle in fruits.keys():

... print(cle)

méthode dict.keys() de la classe dictionnaire renvoit la liste de toutes les clés du dictionnaire

méthode dict.value fait la même chose avec les valeurs

on peut aussi l'utiliser dans des tests if :

>>> if 21 in fruits.values():

... print("Un des fruits se trouve dans la quantité 21.")

...

Un des fruits se trouve dans la quantité 21.

>>>

et si on veut parcourir les deux à la fois, on utilise

dict.items()

-> renvoie une liste de couples (clé, valeur)

fonctions aux paramètres inconnus.

si on marque fonction(parametre='a')

alors on ajoute ce paramètre à un dictionnaire

pour transformer un dictionnaire en paramètres d'une fonction :

idem, on peut metre directeement le nom d'un dictionnaire dans les parenthèses après le nom de la fonction

# Les fichiers

Pour savoir dans quel espace de travail on est :

os.getcwd()

Pour changer d'espace de travail :

os.chdir("D:\\Desktop\\Python ")

OU os.chdir("D:/Dekstop/Python")

**pour ouvrir un fichier dans python** :

open("chemin", mode d'ouverture)

modes d'ouverture :

- 'r' : read, pour la lecture

- 'w': write pour écrire dessus. le contenu du fichier est écrasé, et s'il n'existe pas, il est créé

- 'a': append : mode ajout. On écrit à la fin du fichier sans écraser le contenu pré-existant

-> l'objet créé est un objet de classe "TextIoWrapper"

après l'avoir utilisé, **il faut fermer le fichier** (car sinon, aucun autre programme ne pourra y accéder):

mon\_fichier.close() (méthode de classe TextIoWrapper)

**lire le fichier** :

contenu = mon\_fichier.read()

**écrire dans un fichier :**

mon\_fichier.write("texte à écrire")

-> renvoie le nombre de caractères écrits

le texte ne va se voir dans le fichier texte que quand on l'aura fermé

!! write n'accepte que les chaines de caractère

**mot clé "with"**

with open(mon\_fichier, mode\_ouverture) as variable:

-> on ne manipule le fichier que dans ce bloc, ailleurs, il reste fermé.

-> on l'importe sous le nom "variable"

permet d'éviter d'avoir des bugs parce qu'on n'a pas fermé le fichier

si une exception se produit, le fichier sera quand même fermé à la fin

ce "with" permet de créer un "**context manager**"

**enregistrer objets dans des fichiers : module pickle**

import pickle

with open('donnees', "wb)" as fichier #wb car on veut ouvrir le dossier en mode binaire

mon\_pickler = pickle.Pickler(fichier)

#on appelle un objet Pickler en lui passant en paramètre le fichier dans lequel nous allons enregistrer notre objet

mon\_picker.dump(score)

# le score est enregistré dans mon fichier

**Pour récupérer des objets enregistrés dans un fichier :**

with open('donnees', 'rb') as fichier:

... mon\_depickler = pickle.Unpickler(fichier)

... score\_recupere = mon\_depickler.load()

...

on récupère le premier objet qui a été lu dans le fichier. S'il y en a plusieurs, il faudra appeler cette méthode plusieurs fois

# Portée des variables et références

portée : quand et comment les variables sont-elles accessibles ?

espace local de la fonction = espace qui contient les paramètres passés à la fonction et les variables définies dans son corps.

espace local dans lequel la fonction a été appelé. ici c'est l'éditeur python : si une variable a été définie à cet endroit, alors la fonction va pouvoir la lire

variables définies dans un corps de fonction :

on ne peut pas y accéder une fois qu'on est sorti de la fonction... car l'espace de la fonction a été détruit

pour les variables extérieures à l'espace local de la fonction : on peut les lire, mais pas les modifier. On ne peut modifier que des variables définies DANS la fonction

mais si on fait **paramètre.** methode\_pour\_modifier\_paramètre, alors là ça marche

-> **ici, on modifie l'objet paramètre, on appelle une méthode de l'objet, plutôt que de changer la valeur d'un paramètre**

-> on modifie l'attribut d'un objet

# références

**Les variables cachent des références à des objets**

**Deux références peuvent pointer vers le même objet : si objet1 = objet2, alors si on modifie l'objet 2, on va modifier l'objet 1**

cela ne marche pas avec les entiers, les flottants et les chaines de caractère: ces objets n'ont aucune méthode de travail qui les modifie directement.

quand on fait V1 +=4, on crée une nouvelle variable V1 qui écrase l'ancienne

affectation n'est pas pareil que l'appel d'une méthode !

si on veut créer un nouvel objet depuis un autre:

ma\_liste2=list(ma\_liste1) # on copie le contenu de ma\_liste1

(on peut faire pareil avec dict)

**id(objet)** -> renvoie les paramètres de l'objet, là où il est enregistré dans Python. Si id = id, alors c'est qu'il n'y a qu'un seul objet

# Variables globales

on définit la valeur dans l'éditeur

dans le corps d'une fonction, on précise que cette variable est globale. On peut alors la modifier par affectation dans le corps de la fonction.

>>> i = 4 # Une variable, nommée i, contenant un entier

>>> def inc\_i():

... """Fonction chargée d'incrémenter i de 1"""

... **global** i # Python recherche i en dehors de l'espace local de la fonction

... i += 1

Python accède en lecture ET en écriture à la variable

# créer des classes

en opposition à quoi ?

C : non orienté objet

C++: orienté objet

Java : TOUT doit être contenu dans des classes

en python, tout est objet

dans une classe, on définit des méthodes et des attributs

**convention de nommage :**

Ds python, c'est mieux de nommer les classe dans la convention "**Camel Case**"

-> MaClasse par exemple

// **snake case** : ma\_classe

PEP : Python Enhancement Proposals: propositions d'amélioration de python

créer une nouvelle classe :

class NomDeLaClasse:

def \_\_init\_\_(self): #\_\_init\_\_ = constructeur

self.nom="Dupont" # instanciation de notre attribut nom, valeur par défaut

**méthodes constructeur** : méthode chargée de construire les attributs d'une classe

Elle a pour nom \_\_init\_\_, c'est invariable : en Python, tous les constructeurs s'appellent ainsi. Nous verrons plus tard que les noms de méthodes entourés de part et d'autre de deux signes soulignés (\_\_nommethode\_\_) sont des **méthodes spéciales**. Notez que, dans notre définition de méthode, nous passons un premier paramètre nommé self.

pour créer un objet, on fait ensuite :

-> cette méthode renvoie un objet self

bernard = Personne() # on appelle le constructeur de notre classe Personne

et on peut ensuite changer la valeur de son attribut :

bernard.nom = "Robert"

Pour que les noms et prénoms ne soient pas définis par défaut, mais plutôt entrés comme paramètres lors de la création de l'objet :

def \_\_init\_\_(sef, nom, prenom):

self.nom = nom

self.prenom = prenom

ici, on a des **attributs contenus dans nos objets.** Mais on peut aussi avoir des **attributs de classe**:

class Compteur:

objets\_crees=0 # on définit notre attribut avant la définition du constructeur

def \_\_init\_\_(self)

Compteur.objets\_crees +=1 # préfixe = nom de la classe. chaque fois qu'on crée un objet, on incrémente le compteur

de même, on peut accéder en dehors de la classe :

Compteur.objets\_crees

Pour résumer, quand vous devez travailler dans une **méthode de l'objet sur l'objet lui-même**, vous allez passer par self.

méthodes de classe : se définissent comme les méthodes d'instance, mais avec **cls** à la place de **self**

class Compteur:

"""Cette classe possède un attribut de classe qui s'incrémente à chaque

fois que l'on crée un objet de ce type"""

objets\_crees = 0 # Le compteur vaut 0 au départ

def \_\_init\_\_(self):

"""À chaque fois qu'on crée un objet, on incrémente le compteur"""

Compteur.objets\_crees += 1

def combien(**cls**):

"""Méthode de classe affichant combien d'objets ont été créés"""

print("Jusqu'à présent, {} objets ont été créés.".format(

cls.objets\_crees))

combien = classmethod(combien)

Notez que vous pouvez appeler la méthode de classe depuis un objet instancié sur la classe. Vous auriez par exemple pu écrire a.combien() OU Compteur.combien()

Enfin, pour que Python reconnaisse une méthode de classe, il faut appeler la fonction classmethod qui prend en paramètre la méthode que l'on veut convertir et renvoie la méthode convertie.

**méthodes statiques** : travaillent indépendamment de toutes données

class Test:

"""Une classe de test tout simplement"""

def afficher():

"""Fonction chargée d'afficher quelque chose"""

print("On affiche la même chose.")

print("peu importe les données de l'objet ou de la classe.")

afficher = **staticmethod**(afficher)

# introspection des classes

**méthode dir** : prend en paramètre un objet ou une classe et renvoit la liste de ses attributs et méthodes

méthode **\_\_dict\_\_** (méthode spéciale): prend en paramètre un objet et renvoie un dictionnaire avec ses attributs et leurs valeurs

# les propriétés

**encapsulation :** principe qui consiste à cacher ou protéger certaines données de notre objet -> les données sont alors **privées (//public)**

Dans cerains langages (Java, C++, PhP) on ne peut pas accéder à l'objet depuis l'extérieur de la classe (mon\_objet.mon\_attribut n'est pas possible).

dans ce cas, on accède aux attributs par des méthodes un peu particulières :

- **accesseurs**. pour accéder : mon\_objet.get\_mon\_attribut()

- et **mutateurs.** pour modifier: mon\_objet.set\_mon\_attribut(valeur)

Dans python, c'est plus subtil : il y a encapsulation de certaines données seulement (pour celles dont on attend une action particulière) -> dans ce cas, on crée des propriétés

ds les autres langages, on définit les **principes d'accès** aux données lors de la définition de la classe

**Dans Python, tout est public.** encapsulation fondée sur la définition de propriétés : "quand un utilisateur souhaite modifier cet attribut, fait celà". cela repose sur le bon sens.

définition attribut avec un \_ avant :

self.\_lieu\_residence au lieu de self.lieu\_residence -> marque le fait que l'on n'accède pas à cette attribut depuis l'extérieur de la classe (- > convention, dans les faits, rien n'est interdit).

puis, dans le corps de la classe, on définit les méthodes \_get\_lieu\_residence et \_set\_lieu\_residence (noter ici aussi l'underscore au début)

puis, à la fin de la définition de la classe :

lieu\_résidence = property(\_get\_lieu\_residence, \_set\_lieu\_residence)

--> ici on définit lieu\_résidence (sans underscore) comme une propriété fonction de (dans l'ordre, tout est facultatif : accesseur, mutateur, suppresseur, accès à l'aide)

ici, on peut toujours faire jean.lieu\_residence = Dublin

mais alors, on appelle la fonction \_set\_lieu\_residence : on peut mettre messages d'alertes, lever une erreur...

de cette manière, on peut définir les méthodes qui seront appelées si on fait :

- del objet.lieu\_résidence (à la troisième place)

- help(objet.lieu\_residence) (à la quatrième place)

**Résumé :**

Dans cette section, je vais condenser un peu tout le chapitre. Nous avons vu qu'en Python, quand on souhaite accéder à un attribut d'un objet, on écrit tout bêtementobjet.attribut. Par contre, on doit éviter d'accéder ainsi à des attributs ou des méthodes commençant par un signe souligné \_, question de convention. Si par hasard une action particulière doit être menée quand on accède à un attribut, pour le lire tout simplement, pour le modifier, le supprimer…, on fait appel à des propriétés. Pour l'utilisateur de la classe, cela revient au même : il écrit toujoursobjet.attribut. Mais dans la définition de notre classe, nous faisons en sorte que l'attribut visé soit une propriété avec certaines méthodes, accesseur, mutateur ou autres, qui définissent ce que Python doit faire quand on souhaite lire, modifier, supprimer l'attribut.

# Méthodes spéciales

se notent \_\_methodespeciale\_\_

**méthodes qui permettent de travailler sur l'objet :**

**\_\_init\_\_**

**\_\_del\_\_**

si on crée un objet à l'intérieur d'un espace de nom (local, corps d'une fonction par exemple), il sera supprimé quand on sortira de l'espace de nom.

de même que pour init, si on marque def \_\_del\_\_(self), on peut contrôler ce qui se passe quand on supprime l'objet (afficher un message par exemple)

si on ne définit pas une méthode spéciale, python aura un comportement par défaut

**méthodes qui permettent de contrôler comment l'objet est représenté et affiché**

**\_\_repr\_\_ :** comment l'objet est affiché quand on l'appelle dans la console

ex :

def \_\_repr\_\_(self):

return "je suis un objet"

**\_\_str\_\_** : détermine ce qui est affiché si on fait print(objet)

si rien n'est défini, print renvoie par défaut repr(objet)

**accéder aux attributs de notre objet :**

**\_\_getattr\_\_ :**  méthode qui est appelée quand on fait objet.attribut

**-> si l'attibut n'existe pas, alors python va exécuter \_\_getattr**

!! différence de l'établissement d'une propriété avec \_set : méthode qui est appelée quand l'attribut existe

**par exemple :**

def \_\_getattr\_\_(self):

print("il n'y a aucun attribut de ce nom là !!!")

si je fais objet.blablabla() -> il va exécuter cette méthode

**\_\_setattr\_\_ :**

méthode qui est appelée si on fait **objet\_attribut = nouvelle valeur**

ça équivaut à objet.\_\_setattr(attribut, nouvelle valeur)

def \_\_setattr\_\_(self, nom\_attr, val\_attr):

"""Méthode appelée quand on fait objet.nom\_attr = val\_attr.

On se charge d'enregistrer l'objet"""

object.\_\_setattr\_\_(self, nom\_attr, val\_attr)

#ici on appelle la méthode setattr définie au niveau de la classe objet (dont toutes les classes créées héritent). dans la définition de \_\_setattr\_\_, on ne peut pas faire self.attribut = nouvelle\_valeur, ça ferait tourner le truc en boucle !

self.enregistrer()

**\_\_delattr\_\_**

appelée quand on fait del objet.attribut

ici encore, lors de la définition de la fonction, on ne peut pas faire

del self.attribut

il faut écrire :

objet.\_\_delattr(self, attribut)

!! **différence entre les propriétés et les méthodes \_\_getattr\_\_ et \_\_setattr\_\_** :

Ton \_set\_attribut est propre à un attribut, il ne sera appelé que lors de modifications sur l'attribut attribut. \_\_setattr\_\_ est plus général, et est appelé à chaque modification d'attribut. Les propriétés (et plus largement les descripteurs), sont un comportement permis par les méthodes \_\_getattribute\_\_, \_\_setattr\_\_ et \_\_delattr\_\_ de la classe object

**les méthodes de conteneur**

**surcharge d'opérateurs** : expliquer à Python ce qu'il doit faire quand il utilise tel ou tel opérateur

**accéder à un élément d'un conteneur :**

**objets conteneurs** : chaines de caractères, listes et dictionnaires : il contiennent plusieurs objets grâce auxquels on peut accéder grâce à []

**\_\_getitem\_\_:** ce qu'on fait quand on fait objet[index]

**\_\_setitem\_\_,** -> objet[index] = nouvelleValeur

**\_\_delitem\_\_** -> del objet[index]

**méthode spéciale derrière le mot-clé "in" : \_\_contains\_\_**

-> renvoie un booléen True ou False

faire

8 in ma\_liste

revient au même que

ma\_liste.\_\_contains\_\_()

**\_\_len\_\_: connaitre la taille d'un conteneur, via len()**

**les méthodes mathématiques:**

**\_\_add\_\_(self, obj) :** que faire si je veux additionner deux objets ? (ce que je fais si je fais d1 + obj)

**\_\_sub\_\_** : signe "-"

**\_\_mul\_\_** : signe \*

**\_\_truediv\_\_** : signe "/"

**\_\_floordiv\_\_** : //

**\_\_mod\_\_** : % (modulo)

**\_\_pow\_\_**: \*\* puissance

si on veut faire le travail dans l'autre sens (**4 + d1** plutôt que d1 + 4):

on rajoute un r devant la méthode spéciale : \_\_radd\_\_, \_\_rmod\_\_...

et si on veut faire **d1 +=4**, on rajoute un i : **\_\_iadd\_\_**

**méthodes de comparaison :**

| **Opérateur** | **Méthode spéciale** | **Résumé** |
| --- | --- | --- |
| == | def \_\_eq\_\_(self, objet\_a\_comparer): | Opérateur d'égalité (*equal*). Renvoie True si self et objet\_a\_comparer sont égaux, False sinon. |
| != | def \_\_ne\_\_(self, objet\_a\_comparer): | Différent de (*non equal*). Renvoie True si self et objet\_a\_comparer sont différents, False sinon. |
| > | def \_\_gt\_\_(self, objet\_a\_comparer): | Teste si self est strictement supérieur (*greather than*) à objet\_a\_comparer. |
| >= | def \_\_ge\_\_(self, objet\_a\_comparer): | Teste si self est supérieur ou égal (*greater or equal*) à objet\_a\_comparer. |
| < | def \_\_lt\_\_(self, objet\_a\_comparer): | Teste si self est strictement inférieur (*lower than*) à objet\_a\_comparer. |
| <= | def \_\_le\_\_(self, objet\_a\_comparer): | Teste si self est inférieur ou égal (*lower or equal*) à objet\_a\_comparer. |

si Python n'arrive pas à faire objet1 < objet2, il essayera l'opération inverse, soit objet2 >= objet1. Cela vaut aussi pour les autres opérateurs de comparaison que nous venons de voir.

**méthodes spéciales à pickle** :

\_\_getstate\_\_ : appelée au moment de sérialiser l'objet. Quand on enregistre l'objet à l'aide du module pickle, \_\_getstate\_\_ est appelée juste avant l'enregistrement. Sinon, pickle enregistre la valeur renvoyée par \_\_getstate\_\_.

"Dans le premier cas, on modifie le dictionnaire des attributs *avant* la sérialisation. Le dictionnaire des attributs enregistré est celui que nous avons modifié avec la valeur de notre attribut temporaire à 0."

\_\_setstate\_\_ : appelée au moment de désérialiser l'objet, donc après le \_\_getstate\_\_

Dans le second cas, on modifie le dictionnaire d'attributs *après* la désérialisation. Le dictionnaire que l'on récupère contient un attribut attribut\_temporaire avec une valeur quelconque (on ne sait pas laquelle) mais après avoir récupéré l'objet qui est déjà instancié (et avant le retour de la désérialisation !), on met cette valeur à 0.

-> les deux sont à peu près équivalents

# le tri de liste en python

méthode "list.sort()" : travaille sur la liste elle même

méthode "sorted()" : renvoie un nouvel objet. !! à la différence de "sort", elle travaille sur n'importe quel type de séquence (tuple, liste ou dictionnaire)

la méthode de tri dépend du type d'éléments que la liste contient (chaine de caractère ou nombres)

si on veut trier un tableau selon une colonne donnée ?

exemple : #liste de tuples

etudiants = [

("Clément", 14, 16),

("Charles", 12, 15),

("Oriane", 14, 18),

("Thomas", 11, 12),

("Damien", 12, 15),

]

sorted(etudiants)

-> cela nous sort les étudiants triés selon la première colonne, cad leurs noms

si on veut trier selon une autre colonne : argument **key** -> paramètre optionnel, sous forme d'une fonction lambda (le plus simple, sinon on définit la fonction avant)

fonction lambda :

- en entrée les éléments de la liste à trier (-> ici les tuples noms, âge, note)

- en sortie l'élément qui doit être utilisé pour trier (- exemple, note)

sorted(etudiants, **key=lambda colonnes: colonnes[2]**)

pour trier une liste qui contient plusieurs objets Etudiant

etudiants = [

Etudiant("Clément", 14, 16),

Etudiant("Oriane", 14, 18),

Etudiant("Thomas", 11, 12),

Etudiant("Damien", 12, 15)]

listes\_etudiants=sorted(etudiants, **key = lambda x:x.moyenne**)

trier dans l'ordre inverse : argument facultatif reverse :

sorted(etudiants, key=lambda etudiant: etudiant.age, reverse=True)

# les fonctions du module operator

**fonction itemgetter** dans le module operator, pour trier une **liste de tuples**

from operator import itemgetter

sorted(etudiants, key=itemgetter(2))

qui équivaut à

sorted(etudiants, key=lambda etudiant: etudiant[2])

**fonction attegetter**, pour trier une **liste d'objets**

from operator import attrgetter

sorted(etudiants, key=attrgetter("moyenne"))

qui équivaut à

listes\_etudiants=sorted(etudiants, key = lambda x:x.moyenne)

on peut trier selon plusieurs critères :

sorted(etudiants, key=attrgetter("moyenne", "age"))

-> attributs à entrer selon leur ordre d'importance

si deux éléments sont sensés arriver à la même place après le tri, python garde l'ordre dans lequel ils étaient dans la première liste à trier

-> propriété qui est appelée "**stabilité**"

si on veut croiser selon 1 critère croissant et un critère décroissant, on peut utiliser la propriété de stabilité et faire deux tris consécutifs

liste1 = sorted(etudiants, key=attrgetter("moyenne"))

liste2=sorted(liste1, key=attrgetter("age"), reverse = True)

= **chaînage de tri**

# la notion d'héritage

si une classe b hérite de la classe a (classe mère ou classe parente), alors les objets de type b classés auront accès aux méthodes et attributs de la classe a. MAIS peut aussi en définir d'autres.

pour définir une classe "héritière":

class A:

...

class B(A):

...

si des méthodes sont définies dans les 2 classes, alors ce sera celle de la classe B qui sera appelé en premier.

Si méthode définie uniquement dans la classe A, elle pourra être appliquée à la classe B.

!! s'il deux classes \_\_init\_\_ sont définies, alors Python ne va lire que celle de la classe B, et donc ne définir que les attributs qui sont définis à ce niveau là.

mais on peut aussi dire explicitement à Python d'aller chercher la méthode de la classe :

MaClasse.maMethode(mon\_objet)

et on peut l'appeler DANS l'espace de la classe B(A).

class Personne:

"""Classe représentant une personne"""

def \_\_init\_\_(self, nom):

"""Constructeur de notre classe"""

self.nom = nom

self.prenom = "Martin"

def \_\_str\_\_(self):

"""Méthode appelée lors d'une conversion de l'objet en chaîne"""

return "{0} {1}".format(self.prenom, self.nom)

class AgentSpecial(Personne):

"""Classe définissant un agent spécial.

Elle hérite de la classe Personne"""

def \_\_init\_\_(self, nom, matricule):

"""Un agent se définit par son nom et son matricule"""

# On appelle explicitement le constructeur de Personne :

**Personne.\_\_init\_\_(self, nom**)

self.matricule = matricule

def \_\_str\_\_(self):

"""Méthode appelée lors d'une conversion de l'objet en chaîne"""

return "Agent {0}, matricule {1}".format(self.nom, self.matricule)

**fonction isinstance(objet, classe)**

-> renvoie un booléen vrai en si un objet est une instance d'une classe donnée

**fonction issubclass(classe1, classe2)**

-> renvoie un booléen vrai si la classe1 hérite de la classe2

**Héritage multiple**

-> quand un objet hérite de 2 classes

class MaClasseHeritee(MaClasseMere1, MaClasseMere2)

La recherche des méthodes se fait d'abord dans MaClasseHeritee, puis dans MaClasseMere1, puis dans MaClasseMere2.

les types d'erreurs sont des classes, qui peuvent hériter les unes des autres.

* BaseException : la classe mère de *toutes* les exceptions. La plupart du temps, si vous faites hériter votre classe de BaseException, ce sera pour modéliser une exception qui ne sera pas foncièrement une erreur, par exemple une interruption dans le traitement de votre programme.
* Exception : c'est de cette classe que vos exceptions hériteront la plupart du temps. C'est la classe mère de toutes les exceptions « d'erreurs ».

# les itérateurs

for element in ma\_liste

-> python appelle l'iterateur de ma\_liste (objet qui est chargé de parcourir la liste)

méthode spéciale \_\_iter\_\_ dans la classe de l'objet (ici liste) -> renvoie un objet : un itérateur.

Puis à cet itérateur, on lui applique \_\_next\_\_, qui doit renvoyer l'élément suivant du parcours ou bien lever l'exception StopIteration si le parcours touche à sa fin.

class RevStr(str):

"""Classe reprenant les méthodes et attributs des chaînes construites

depuis 'str'. On se contente de définir une méthode de parcours

différente : au lieu de parcourir la chaîne de la première à la dernière

lettre, on la parcourt de la dernière à la première.

Les autres méthodes, y compris le constructeur, n'ont pas besoin

d'être redéfinies"""

def \_\_iter\_\_(self):

"""Cette méthode renvoie un itérateur parcourant la chaîne

dans le sens inverse de celui de 'str'"""

return ItRevStr(self) # On renvoie l'itérateur créé pour l'occasion

class ItRevStr:

"""Un itérateur permettant de parcourir une chaîne de la dernière lettre

à la première. On stocke dans des attributs la position courante et la

chaîne à parcourir"""

def \_\_init\_\_(self, chaine\_a\_parcourir):

"""On se positionne à la fin de la chaîne"""

self.chaine\_a\_parcourir = chaine\_a\_parcourir

self.position = len(chaine\_a\_parcourir)

def \_\_next\_\_(self):

"""Cette méthode doit renvoyer l'élément suivant dans le parcours,

ou lever l'exception 'StopIteration' si le parcours est fini"""

if self.position == 0: # Fin du parcours

raise StopIteration

self.position -= 1 # On décrémente la position

return self.chaine\_a\_parcourir[self.position]

# les générateurs

si on a un générateur mon\_generateur, on peut tout simplement faire :

for nombre in mon\_generateur(): # Attention on exécute la fonction (c'est obligatoire pour que ça marche)

print(nombre)

dans la fonction mon\_générateur, on a des "yield" : ce sont ces yield qui vont être lus les uns après les autres

**exemple** :

def intervalle(borne\_inf, borne\_sup):

"""Générateur parcourant la série des entiers entre borne\_inf et borne\_sup.

Note: borne\_inf doit être inférieure à borne\_sup"""

borne\_inf += 1

while borne\_inf < borne\_sup:

yield borne\_inf

borne\_inf += 1

def intervalle(borne\_inf, borne\_sup):

"""Générateur parcourant la série des entiers entre borne\_inf et borne\_sup.

Note: borne\_inf doit être inférieure à borne\_sup"""

borne\_inf += 1

while borne\_inf < borne\_sup:

yield borne\_inf

borne\_inf += 1

Si, dans une classe quelconque, la méthode spéciale \_\_iter\_\_ contient un appel à yield, alors ce sera ce générateur qui sera appelé quand on voudra parcourir la boucle.

**notion de co-routine**

-> moyen pour altérer le parcours pendant le parcours

interrompre la boucle : méthode **close** de la classe générateur.

**exemple** :

generateur = intervalle(5, 20)

for nombre in generateur:

if nombre > 17:

generateur.close() # Interruption de la boucle

attention, il faut avoir stocké intervalle(5,20) avant dans une variable, sinon ça ne marche pas

envoyer des données au générateur :

On configure le générateur de telle manière qu'il accepte une valeur éventuelle au cours du parcours, avec

valeur\_recue = (yield valeur\_a\_renvoyer)

et ensuite, on peut utiliser la methode generator.send(valeur)

def intervalle(borne\_inf, borne\_sup):

"""Générateur parcourant la série des entiers entre borne\_inf et borne\_sup.

Notre générateur doit pouvoir "sauter" une certaine plage de nombres

en fonction d'une valeur qu'on lui donne pendant le parcours. La

valeur qu'on lui passe est la nouvelle valeur de borne\_inf.

Note: borne\_inf doit être inférieure à borne\_sup"""

borne\_inf += 1

while borne\_inf < borne\_sup:

valeur\_recue = (yield borne\_inf)

if valeur\_recue is not None: # Notre générateur a reçu quelque chose

borne\_inf = valeur\_recue

borne\_inf += 1

PUIS POUR INTERRAGIR AVEC LE GENERATEUR:

generateur = intervalle(10, 25)

for nombre in generateur:

if nombre == 15: # On saute à 20

generateur.send(20)

print(nombre, end=" ")

le résultat est : 11 12 13 14 21 15 20 21 22 23 24

# VOCABULAIRE A CONNAITRE

- introspection des classes

- langage script (=interprété)/langage compilé

- portabilité des langages

- méthodes d'instances, méthodes de classe, méthodes statiques

- convention de nommage : snake case, camel case

- valeurs locales/valeurs globales

- référence

- portée des variables

- "context manager"

- set, dictionnaire, tuple...

- typage dynamique

- lever une exception

- package, module

- docstring

- modularité

- prédicat

- bloc d'instruction

- progiciel

- encapsulation

- accesseurs/mutateurs

- attribut privé ou public

- principes d'accès, propriétés

- fichier source

- méthodes spéciales

- objets conteneurs

- surcharge d'opérateurs

# logiciels de gestion de versions

- **logiciels centralisés** : un serveur conserve les anciennes versions des fichiers et les développeurs s’y connectent pour prendre connaissance des fichiers qui ont été modifiés par d’autres personnes et pour y envoyer leurs modifications.

- **logiciels distribués** : il n’y a pas de serveur, chacun possède l’historique de l’évolution de chacun des fichiers. Les développeurs se transmettent directement entre eux les modifications, à la façon du *peer-to-peer*.

- ***Un logiciel de gestion de versions distribué avec un serveur****.* Le serveur sert de point de rencontre entre les développeurs et possède lui aussi l’historique des versions

-> Comme Git : Il a l’avantage d’être à la fois flexible et pratique. Pas besoin de faire de sauvegarde du serveur étant donné que tout le monde possède l’historique des fichiers, et le serveur simplifie la transmission des modifications.

**comparaison avec d'autres logiciels** :

Je n’entrerai pas dans les détails de la comparaison de Git avec les autres outils concurrents comme SVN et Mercurial. Retenez simplement que :

* CVS est le plus ancien et il est recommandé de ne plus l’utiliser car il est le moins puissant et n’est plus très bien mis à jour ;
* SVN est le plus connu et le plus utilisé à l’heure actuelle, mais de nombreux projets commencent à passer à des outils plus récents ;
* Mercurial, Bazaar et Git se valent globalement, ils sont récents et puissants, chacun a des avantages et des défauts. Ils sont tous distribués, donc chaque développeur possède l’historique des modifications et ils permettent en théorie de se passer de serveur (bien qu’on utilise toujours un serveur pour des raisons pratiques).

Dans un logiciel de gestion de versions comme Git, un **dépôt** représente une copie du projet. Chaque ordinateur d’un développeur qui travaille sur le projet possède donc une copie du dépôt.  
Dans chaque dépôt, on trouve les fichiers du projet ainsi que leur historique (voir les schémas du début du chapitre).

pour aller dans le fichier qu'on veut :

cd D:\\Desktop\\Python\\Git

pour créer un nouveau fichier :

mkdir PremierEssai

pour télécharger un dépôt déjà existant sur internet

git clone http://github.com/symfony/symfony.git

dans les 2 cas, on crée un dossier caché .git dans lequel Git stocke l'historique des fichiers et de la configuration

dossier courant de travail = working directory

**git status**

pour afficher les fichiers qu'on a modifié récemment

(nothing to commit = rien n'a été modifié)

Lorsqu’on travaille avec Git, on suit en général toujours les étapes suivantes :

1. modifier le code source ;
2. tester votre programme pour vérifier si cela fonctionne ;
3. faire un commit pour « enregistrer » les changements et les faire connaître à Git ;
4. recommencer à partir de l’étape 1 pour une autre modification.

1 commit = un ensemble de changements du code source -> il faut que l'ensemble soit cohérent (= résoudre un seul bug ou développer une nouvelle fonctionnalité)

commit sont là pour valider l'avancement du projet

**git diff**

Pour voir ce qui a changé concrêtement, à l'intérieur des fichiers

lignes avec - (ce qui a été enlevé), et avec un + (ce qui été ajouté)

**effectuer un commit des changements**

Il faut explicitement préciser les fichiers que vous voulez « commiter ». Pour cela, trois possibilités :

* faire git add nomfichier1 nomfichier2 pour ajouter les fichiers à la liste de ceux devant faire l’objet d’un commit, puis faire un git commit. Si vous faites un git status après un git add, vous les verrez alors en vert ;
* fairegit commit -apour « commiter » tous les fichiers qui étaient listés dans git status dans les colonnes « *Changes to be committed* » et « *Changed but not updated* » (qu’ils soient en vert ou en rouge) ;
* faire git commit nomfichier1 nomfichier2 pour indiquer lors du commit quels fichiers précis doivent être « commités ».

J’utilise personnellement la seconde solution lorsque je veux « commiter » tous les fichiers que j’ai modifiés, et la troisième solution lorsque je veux « commiter » seulement certains des fichiers modifiés.  
Faire appel àgit addest indispensable lorsque vous venez de créer de nouveaux fichiers que Git ne connaît pas encore. Cela lui permet de les prendre en compte pour le prochain commit.

Lorsque la commande *commit* est lancée, l’éditeur par défaut (généralement nano ou Vim) s’ouvre. Vous devez sur la première ligne taper un message qui décrit à quoi correspondent vos changements.

$ git commit -m "changement Yaml 2"

-> permet d'entrer directement tous les changements sous un commit de nom "bablabla"

-> avec cette méthode, on enregistre un **commit localement**