Journal d’apprentissage :

***Semaine 1 :***

Difficile de se motiver et de trouver le rythme. J’ai passé la semaine à me dire que ce premier projet était très rapide et donc que je pouvais bien le reporter à demain, à après-demain…

Je dois trouver une organisation cohérente entre les enfants, le travail, le sport, le couple et la formation.

J’ai repris ma lettre de demande de réduction de temps de travail et je vais bientôt la donner à ma directrice. Je vais me libérer plus de temps mais je ne dois pas attendre janvier pour m’impliquer.

Nous sommes vendredi, demain c’est l’anniversaire de Mélie, je speed sur mon projet pour débuter le prochain dimanche soir. La semaine prochaine devrait être plus calme.

***Semaine 2 :***

**Git et console**

*Console:*

**Pwd** : donne le répertoire courant

**Ls**: liste des fichiers et répertoires dans le répertoire courant (afficher sous forme de liste ajouter **-l** après ls, pour plus d’infos ajouter **–a**)

**Cd**: permet e se placer dans un répertoire (ajouter le nom du répertoire après cd)

**Touch** : permet de créer un fichier (ex : touch fichieracreer.txt)

**Mkdir** : permet de créer un dossier (ex : mkdir dossieracreer)

**Cat**: afficher le contenu d’un fichier (ex : cat monfichier.txt)

*Git :*

Pour activer un dossier comme repository il faut se placer dedans avec **cd** et taper **git init**

Lorsque l’on créer un fichier dans un repository il faut l’ajouter a l’index git : (**git add** nomdevotrefichier.extension)

Lorsqu’on modifie un repository il faut faire un commit (**git commit**). Ajouter –m permet d’envoyer un message décrivant les modif effectuées ( **git commit –m «**ajouté ma checklist-vancances.md **»**)

On entre d’abord le fichier dans l’index avec **git add** avant de le commiter avec **git commit –m**

Mais si on met a jour un fichier déjà ajouté a l’index auparavant il suffis de faire **git commit –a –m «** ajouté itinéraire dans checklist-vacances.md **»**

**Git log** affiche la liste de tous les commit réalisés (le plus récent est en haut de la list et le plus plus vieux en bas). Pour quiter le log appuyer sur **Q**

Pour se positionner sur un commit donné il faut taper **git checkout SHAducommit**

Pour revenir a la branche précédente taper **git checkout master**

On ne peut pas annuler une modification sur un commit mais on peut créer un nouveau commit qui fera l’inverse du précédente avec **git revert SHAducommit**

On peut modifier le message ajouté a la fin d’un commit (avec –m) en tapant **git commit --amend –m «** votre nouveau message **»** (il y a deux – entre commit et ammend)

*Mais il n’est possible de modifier le message que s’il n’a pas encore été « pushé » sur l’origine !*

Au cas où je n’ai pas encore fait mon nouveau commit mais je veux annuler tous les changements que je n’ai pas encore commité il faut taper **git reset -- hard**

*Github :*

**t** pour faire une recherche en tapant un mot ou une lettre clé

Pour copier un repo présent sur github sur ma machine il faut taper **git clone lienfournipargithub** (sur git dans le repo que l’on veut il faut cliquer sur clone url en bas a droite de l’écran)

Depuis le local Envoyer les modifications apportées à un commit local sur github avec **git push origin master**

Bien se placer dans le répertoire local pour effectuer les push.

Bien penser a ajouter avec add les fichiers copier depuis GitHub à Git et a en faire des commits sur Git.

Depuis GitHub **git pull origin master** pour récupérer en local les dernières modifs sur le fichier .

Les branche : la branche principale est appelée **master**. Pour voir les branches taper **git branch** (une étoile sera placée devant la branche ou on est placé)

Créer une nouvelle branche **git branch nouvelle-branche**

Pour se placer dans une branche a l’intérieur du repo **git checkout nouvelle-branche**

**Git checkout –b ma-branche** créer une branche et se positionne directement dedans

Pour fusionner deux branche (ajouter les modifs de la branche A à la branche B) se mettre dans la branche A avec **git checkout brancheA** puis **git merge brancheB** pour les fusionner

On doit toujours chercher à remettre les modifs de branche dans la branche principale **master**

Lorsque deux branches sont impossibles à fusionner car il existe des conflits (ex : une phrase est différente dans les deux.) après le message d’erreur taper **sublime nomdefichier.extension** cela affichera les différences entre les deux fichiers

Trouver qui a modifier un fichier **git blame nomdufichier.extension** cela liste toutes les modifs faites ligne par ligne. Cela rendra le début du SHA avec le nom de la personne.

Trouver pourquoi une modif a été faite soit **git log** et rechercher le commit dont le SHA commence par celui rendu lors de la recherche du qui a fait la modif. Soit **git show** qui renvoie directement les détails du commit recherché en saisissant le début de son SHA.

Pour des raisons de sécurité et de clarté, il est important d'ignorer certains fichiers dans Git, tels que :

* Tous les fichiers de configuration (config.xml, databases.yml, .env...)
* Les fichiers et dossiers temporaires (tmp, temp/...)
* Les fichiers inutiles comme ceux créés par votre IDE ou votre OS (.DS\_Store, .project...)

Le plus crucial est de ne **JAMAIS versionner une variable de configuration**, que ce soit un mot de passe, une clé secrète ou quoi que ce soit de ce type. Versionner une telle variable conduirait à une large faille de sécurité, surtout si vous mettez votre code en ligne sur GitHub !

Créer un fichier **.gitignore** pour lister les fichiers a ne pas versionner dans GIT. listez ces fichiers ligne par ligne dans .gitignore en indiquant leurs chemins complets . le fichier .gitignore doit être ajouté a l’index et commiter comme le reste

Pour mettre de côté des modifs d’une branche pour aller travailler sur une autre branche il faut taper **git stash** et lorsque le travail aill -eurs est fini revenir sur la branche concerné et taper **git stash pop** pour récupère les modis mise de côté. En remplacant pop par **apply** cela conserve les modifs dans la stash. Sinon au moment du pop la stash est vide et il faut commiter a la fin pour sauvegarder.

Pour ajouter des fichiers « untracked » a un repos avec des fichiers déjà **add** il faut taper **git add -A**

***Semaine 3-4 :***

**Python :**

* Pour exécuter un fichier .py : clic droit / ouvrir avec idle puis f5 ou run
* Tout au début du code préciser l’encodage fichier ( Utf-8 ou Latin-1) # -\*-coding: non de l’encodage -\*
* Pour mettre python en pause (uniquement pour windows) pour afficher le résultat : au début du code taper : import os / puis tout à la fin du code taper : os.system(« pause »)
* Sous linux ou mac pour mettre en pause il faut terminer le code par : input(« appuyez sur ENTREE pour fermer ce programme… »)
* **#** jusqu’à un saut de ligne permet de laisser des commentaires dans le code
* **/** permet de couper une instruction avant d’aller à la ligne. Cela dit à Python que la ligne de commande continue à la ligne d’en dessous

*Divisions :*

**/** donne le résultat d’un division (ex : 10/3 = 3.33333)

**//** donne la partie entière d’une division (ex: 10//3 = 3)

**%** (modulo) donne le reste d’une division (ex : 10%3 = 1)

*Variables :*

Pour donner une variable à python il suffit d’écrire :

**Nom\_de\_la\_variable = valeur**

Pour récupérer la valeur de la variable il faut taper **nom\_de\_variable** et python renvoie la valeur

Une variable ne peut être composé que de lettres, majuscule, minuscule, chiffre et d’underscore. Le nom ne peut pas commencer par un chiffre.

Possible d’augmenter une variable en faisant **nom\_de\_la\_variable = nom\_de\_la\_variable + 2**

Le meilleur moyen est d’utiliser **variable += 1** pour incrémenter (augmenter) la variable. On peut aussi utiliser **/=**, **-=** et **\*=**.

Une variabale peut être utiliser pour définir une autre variable (ex : **mon\_age\_x2 = mon\_age \* 2**)

On ne peut pas construire de variable portant les noms : **and, del, from, none, true, as, elif, global, nonlocal, try, assert, else, if, not, while, break, exept, import, or, with, class, false, in, pass, yield, continue, finally, is, raise, def, for, lambda, return**

*Nombres :*

**int**  nombre entier

**float** nombre à virgule (le virgule s’écrit avec un point **.** )

**str**  chaine de caractère

On peut faire considérer un nombre entier comme un nombre flottant en le notant avec un point et un zéro (ex : **52.0**)

*Chaine de caractères :*

On peut l’écrire de trois manière différentes :

**« ceci est une chaine de caractères »**  (guillemets)

**‘ceci est une chaine de caractère’** (apostrophe)

**« « «  ceci est une chaine de caractère » » »** (triple guillemets)

On peut stocker une chaine de caractère dans une variable (ex : **ma\_chaine= ‘bonjour, la foule !’**

Lorsque l’on écrit une variable contenant une chaine de caractère avec le même symbole que celui délimitant la chaine on obtient une erreur. (Ex : **chaine = ‘j’aime le python’**)

Dans ce cas il faut placer un antislash avant l’apostrophe interne (ex : **chaine = ‘j\’aime le python !’**. si on utilise les guillemets il faut aussi mettre un antislash (ex : **chaine2 = « \ »le seul individu formé(…)\ » »**

On appelle cela échapper. Pour écrire un antislash dans une chaine de caractère il faut l’échapper lui-même **\\**

**\n** permet de sauter une ligne (ex : **« essai\nsur\nplusieurs\nlignes »**

Utiliser les triples guillemets permet de s’affranchir d’échapper.

On peut permuter deux variables (**variable1, variable2 = variable2, variable1 # permutation**)

On peut affecter la même valeur a plusieurs variables (ex : **x = y = 3**)

L’antislash permet aussi de couper une instruction pour la poursuivre à la ligne suivante.

*Fonctions :*

**Nom\_de\_la\_fonction(parametre\_1,parametre\_2,…,parametre\_n)**

Même si la fonction n’a pas de paramètre on doit quand même mètre les parenthèses

On peut connaitre le type d’un objet par la variable **type(nom\_de\_la\_variable)**

(ex : a = 3, taper dans l’interpréteur : type(a). il renvoie {class ‘int’} pour dire que le chiffre est de la classe des entiers)

**Print()** affiche la valeur d’une ou plusieurs variables ou un message

(Ex : a=3 / print(a) / a=a+3 / b= a-2 / print(« a= », a, « et b= », b) / a=6 et b=4)

On demande a print d’afficher la chaine de caractère « a= » puis la fonction a, puis la chaine de caractère « et b= » et enfin la foinction de b. python les affiche dans l’ordre et cela donne a=6 et b=4

**Input()** donnée saisis par l’utilisateur

*Structures conditionnelles :*

**If :** (les deux points sont indispensable pour terminer la condition, la condition se place entre if et les eux points) permet d’établir une condition simple (ex : if a<0 :/ print (« a est inférieur à 0 ») )

**Else :** permet de définir le reste hors des conditions de if et elif . Si la condition explicité par if , n’est pas remplis, alors affiche cela.(ex : if Age >= 18 : /print(« vous êtes majeur »/ else : / print(« vous êtes mineur ») else ne figure qu’a la fin du bloc et ne peut être utilisé qu’une fois.

**Elif :** complète if par une autre condition on peut mettre autant de if et de elif que l’on veut et il n’est pas nécessaire e terminer par else

Les conditions (entre if, elif et les deux points) sont exprimées par les symboles

**<** (inférieur à)

**>** (supérieur à)

**<=** (inférieur ou égal)

**>=** (supérieur ou égal)

**==** (égal à)

**!=** (différent de)

*Prédicats et booléens :*

Les prédicats sont ce qui se trouve entre le if et les deux points

Les booléens sont la valeur que renvoie python par exemple sur une opération (ex :a=0 / a==5 / False) **False** et **True** sont des booléens et sont toujours écrit avec une majuscule.

On peut les définir comme variables (ex : majeur = False / if age >= 18 / majeur = True)

Ici majeur vaut False sauf si la variable age est supérieur ou égal a 18.

*Mots clés :*

La différence entre and et or : en tapant and on cherche à savoir si quelque chose est. Avec or on cherche si quelque chose n’est pas.

Les condition de and serons parentes. Les conditions de or seront séparées.

**And** permet ajoute une conditions parente dans if (ex : if a>=2 and a <=8 : / print(« a est dans l’intervalle ») / else / print(« a n’est pas dans l’intervalle »)

*si quelque chose est ceci ET cela alors print ça*

**Or** permet de tester une autre condition dans if en l’occurrence au lieu de chercher si a est dans l’intervalle on cherche à savoir s’il n’est pas dans l’intervalle. Or signifie « ou bien » donc on peut tester deux conditions différentes dans un même if (ex: if annee % 400==0 or (annee % 4 ==0 and annee % 100 != 0) : / print(« l’année saisise est bisextile. »)

*Si quelque chose est ceci OU cela alors print ça*

**Not** inverse un prédicat (ex : a ==5 équivaut donc à a !=5)

**Is** (peut être associer à not) (ex : majeur= False / **if** majeur **is not** True : / print(« vous n’êtes pas encore majeur »)

**In** peut être utilisé comme les précédents dans d’autres condition (par exemple if) pour dire dans quoi on veut travailler (une variable, une chaine de caractère …) (ex : chaine = « Bonjour les ZEROS » / for lettre in chaine : / if lettre in « AEIOUYaeiouy » : / print(lettre) / else : / print(« \* ») )

*Convertir une variable :*

Transforme une variable, par exemple une chaine de caractère en int

(ex : type(annee) / <type ‘str’> / annee=int(annee) / type(annee) / <type ‘int’> / print(annee) / 2009 *au lieu de printer ‘2009’*

Cela permet à l’utilisateur de rentrer des chiffres dans la variable et non une chaine de caractère.

*Les boucles :*

**While :** c’est une variable qui permet de répéter un bloc d’instruction tant que celui-ci est vrai (while veux dire tant que) (ex : nb=7 / i=0 / while i < 10 : / print(i+1, « \* », nb, « = », (i+1) \* nb) / i += 1 . i est le compteur de la boucle, nb la valeur sur laquelle on travail. La dernière ligne i+=1 permet d’incrémenter (augmenter i de 1 a chaque tour de boucle). Utile pour vérifier une condition

**For :** instruction qui parcours une séquence. Il s’écrit **for element in sequence** : la variable element est créer par le for, elle prend successivement chacune des valeurs parcourue dans la séquence. Utile pour parcourir une séquence.

**Break** permet d’interrompre une boucle (instruction à éviter, préférer donner une condition en fin boucle claire dans while ou for)

**Continue** permet de continuer une boucle il n’ignore la fin du bloc que pour le tour de boucle courant.

**Variable += chiffre** permet d’ajouter (incrémenter) une valeur à celle de la variable à chaque tour de boucle

*Les fonctions :*

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

Permet de définir une fonction. On peut définir une limite maximum a une fonction dans ses paramètres (ex : def table(nb, max=10) ). Cette fonction permet d’affiche la tabled e multiplication de nb de 1\*nb a max\*nb.

Il est important de documenter une fonction en expliquant a quoi elle sert. Pour cela on utilise une chaine de caractère flottante (docstring) sous la table avec trois « « « au début et a la fin. Le texte d’aide est indenté au même niveau que la focntion.

Pour les afficher on peut exécuter **help(table)**

On peut appeler la fonction par son nom pour en afficher les paramètre ou bien pour les modifier (ex : fonc() affiche la fonction qui s’appelle fonc, fonc(4) affiche la fonction en incrémentant chaque paramètre de 4, fonc (b=8, d=5) affiche les paramètres de base et remplace ceux modifiés dans l’appel.

**Return** permet de renvoyer une valeur pour la récupèrer ensuite et par exemple la stocker dans une variable.

**Fonction lambda :**

**F = Nom de la fonction argument1, argument2 : instruction de la fonction**

on stocke la fonction dans une variable (f=) pour pouvoir l’appeler (ex : lambda x, y : x+y)

*Les modules :*

**Import** permet d’importer un module

Pour appeler une fonction d’un module on écrit le nom du module suivis d’un point et du nom de la fonction.

**Nom du module. Fonction du module(paramètre)**

(ex : math.sqrt(16) ) on précise que l’on fait appel a la fonction sort (racine carrée) du module math pour le chiffre 16.

**Help(« math »)** permet d’avoir accès a la documentation du module math.

Taper **Q** pour quitter, **espace** pour changer de page et **entré** pour changer de ligne.

On peut appeler l’aide directement sur une fonction particulière d’un module avec help(« math.sqtr ») par exemple.

On peut importer un module en le renommant (ex : import math as mathématiques) ensuite on l’appel par le nom qu’on lui a donné.

**From math import fabs** autre méthode d’import qui permet de n’importer qu’une fonction d’un module. On l’appel ensuite directement (ex ; fabs(5)). En tapant \* a la place du nom de la fonction on appelle toutes les fonctions.

*Créer un module :*

-Créer un dossier

-Y placer un fichier (.py) avec la fonction que l’on veut appeler

-Y placer un autre fichier ou l’on fait appel a cette fonction pour un cas précis

-Pour cela on tape : **from** nom du fichier sans l’extension (.py) **import \***

-On tape le nom de la fonction à appeler et la demande (ex : table(3, 20) ) là c’est pour la fonction def table(nb, max=10) :

-Python va créer dans le dossier au moment de l’import un fichier .pyc qui est le code du module compilé.

-On peut choisir de différencier ce qui va s’exécuter en fonction de si on appelle la fonction en module ou si on l’exécute directement.

Pour cela on doit après le code de la fonction créer la variable

**If \_\_name\_\_ == « \_\_main\_\_ » :**

**Nom de la fonction(valeur)**

En ajoutant ce code, ce qui est définit dans la variable s’affichera sur l’exécution directe du fichier en .py. S’il est appelé en module il ne s’exécutera pas.

Pour l’appeler en module il faut simplement écrire : **nom de la fonction(valeur)** et lors de l’import du module cela sera affiché directement.

*Les exceptions :*

**Try :** information ou actions à tester

**Except :** instructions à exécuter en cas d’erreur

Par exemple si on demande à l’utilisateur de saisir un chiffre et qu’il saisit une chaine de caractères. On peut mettre exept : / print(« erreur lors e la conversion de l’année »)

Entre le except et les : on peut préciser le type d’erreur pour laquelle on veut exécuter une action précise. (ex : exept NameError : / print(«la variable numérateur ou dénominateur n’a pas été définie »)

On peut capturer le type d’erreur (NameError, TypeError…) dans une variable pour la renvoyer a l’utilisateur. (ex : try : / except type\_de\_l\_execption as exception\_retournee : / print(« voici l’erreur : », exception\_retournee)

**Else :** permettra de renvoyer un message si aucune exception n’est trouvée

**Finally :** permet d’exécuter du code après un bloc try quel que soit le résultat (erreurs ou non)

**Pass :** permet de passer ou de laisser des conditions vides

**Assert :** permet de s’assurer d’une condition particulière dans un bloc try par exemple que la valeur tapée par l’utilisateur est bien < a 0.

On peut donc dans les conditions du exept (entre le except et les : ) mettre AssertionError et demander de renvoyer un message particulier pour ce type d’erreur précise. L’intérêt est que l’on choisit directement l’erreur.

**Raise :** permet de lever une exception

**Semaine 6 :**

*La POO :*

Les variables sont des **objets**, les fonctions de ces variables sont des **méthodes**. On appelle une méthode d’un objet avec objet.methode()

Les types de données (int, str…) sont des **classes**.

Une classe est un modèle qui servira à construire un objet ; c’est dans la classe qu’on va définir les méthodes propres à l’objet.

**.lower()** permet de mettre une variable (chaine de caractère) en minuscule. On l’utilise comme ceci nom\_de\_la\_variable.lower() pour une variable contenant une chaine de caractère.

**.upper()** permet de mettre une chaine de caractère en majuscule

**.capitalize()** permet de mettre la première lettre en majuscule

**.strip()** retire les espace au début et à la fin de la chaine

**.center()** centre une chaine de caractère. On peut définir la taille souhaité en mettant en paramètre (entre les parenthèse) la taille (en nombre de caractère) totale de la chaine. Des espaces seront ajouter au début et à la fin pour atteindre le nombre de caractère voulue t centré la chaine.

**On peut appeler plusieurs méthodes en même temps (ex : objet.methode1().methode2() )**

**Dans ce cas la seconde méthode travaillera sur la première méthode et non sur l’objet de base.**

**Pour connaitre les méthodes d’une classe help(nomdeclasse)**

*Formater et afficher une chaine :*

*La méthode des chaînes renvoie un nouvel objet mais ne modifie pas l’objet d’origine. Pour modifier la variable chaîn\_1 ou créer chaîne\_2. La modification sera appliquée dans la variable chaine 2, la variable chaine\_1 sera intacte. On pourra appeler les deux séparément.*

*Syntaxe 1 :*

**.format(variable1, variable2, variable3)** pour appeler la méthode on choisis l’ordre dans lequel elle va s’appliquer (ex : « prenom = « Paul » / nom = « Dupont » / age = 21 / print (« je m’appelle **{0} {1}** et j’ai **{2}** ans. »**.format(prenom, nom, age)**)

Ainsi on remplace {0} par la première variable et ainsi de suite.

On peut mettre l’ordre que l’on veut. On peut aussi appeler une méthode dans la méthode (ex : .format(prenom, nom, age, **nom.upper()** ) ca permet d’afficher dans le cas {4} le nom en majuscule

Il n’y a pas besoin de mettre de chiffre entre les {} si les variables sont appelées dans l’ordre souhaité dans .format

*Syntaxe 2 :*

On peut nommer directement les variables que l’on va afficher. On va donner directement les valeur des variables que l’on souhaite dans .format :

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)

*Concaténation de chaînes (regrouper des chaînes en une seule) :*

**+** permet de concaténer deux chaînes. On les appels par le nom de variable (ex : chaîne complète = nomdevariable1 + nom de variablel2 ). Si on veux mettre un espace entre les deux chaine il faut procéder comme ceci : chaine\_complete = variable1 + «  »  + variable 2

On a introduit à la main un espace qu’on a placé entre les deux guillemets.

Pour concaténer des chaîne de classe différentes il faut formater a la classe désiré. Pour cela on utilise **nom\_de\_classe(variable)** . (ex : age = 21 / message = « j’ai  » + str(age) + «  ans. »)

*Parcours et sélection de chaînes :*

Pour afficher un caractère donné on tape le **nom de la variable suivis de la position entre []**

variable[0] affiche le premier caractère de la variable chaine.

Variable[2] affiche le troisième caractère

Variable[-1] affiche le dernier caractère

**Len()** donne la longueur de la chaine en nombre de caractères len(nom\_de\_variable)

On peut parcourir une chaîne avec **while** (ex : chaine = « salut » / i = 0 / while i < len(chaine) : / print (chaine[i]) / i += 1

Ce code veux dire qu’**on part du compteur i qui est à zéro (i = 0)** donc au premier caractère, et **on ajoute 1 au compteur à chaque tour de boucle (i += 1)** **jusqu’à la fin de la longueur de la chaine de caractère (while i < len(chaine) :** )

On peut sélectionner une partie de la chaine **[0 : 2]** ici les caractères entre 0 et 2 (donc 0 et 1 hein ! piège !)

On peut sélectionner une partie de la chaîne jusqu’à la fin par exemple du caratere 2 a la fin de la chaine **[2 : len(nom\_de\_le\_variable)]**

Il n’est pas forcément nécessaire de préciser le départ ou la fin de la chaine si on veut que cela parte du début ou aille jusqu’à la fin.

**Du début *nom\_de\_la\_variable[ : 2]* jusqu’à la troisième lettre non comprise**

**De la troisième lettre comprise *nom\_de\_le\_variable[2 : ]* jusqu’à la fin**

*Listes et tuples :*

*La méthode des listes ne renvoie rien mais modifie l’objet d’origine ! Par exemple pour modifie la variable liste\_1 on créer liste\_2 pour par exemple ajouter un élément. La variable liste\_2 sert de transition mais n’existera plus. La liste\_1 de base aura été modifié. On ne pourra appeler que la liste\_1 modifié. L’ancienne liste\_1 n’existe plus.*

*Listes :*

Pour créer une liste : nom\_de\_variable = list() / type(nom\_de\_variable) / <class ‘list’>

Ou bien : **nom\_de\_variable = []**

La liste peux être composé de ce qu’on veut (chaine de caractère entiers, flottant, autre liste…) tous séparés par une virgule et contenues ensemble par []

Pour afficher un élément de la liste on tape **nom\_de\_variable[0]** le zero affiche le prmeir élément et ainsi de suite.

On peut remplacer un élément d’un liste : **nom\_de\_ variable[4] = ‘Z’**

**.append()** pour ajouter un élément à la fin d’une liste.

Sous la forme **nom\_de\_variable.append(élément à ajouter)**

**.insert()** pour ajouter un élément là ou l’on veut dans une liste

Nom\_de\_varibale.insert(place, ‘caractère à insérer’)

**.extend()** pour concaténer des listes. Insère une liste à la fin d’une autre

**Nom\_de\_variable1.extend(nom\_de\_variable2)**

On peut aussi utiliser le **+** (ex : nom\_de\_liste1 + nom\_de\_liste2). Dans ce cas cela renverra le résultat de la liste concaténé

Ou encore **+=** ce qui donne la même chose que extend

**Del variable\_a\_supprimer** pour supprimer une variable

**Del nom\_de\_variable[5]** pour supprimer l’élément 6 d’une liste

**.remove()** en paramètre on nomme l’élément à supprimer et non sa place dans la liste. Cette méthode ne retire que la première occurrence trouvé dans la liste. Si par exemples il y a deux 32. Seul le premier trouvé est effacé.

*Parcourir des listes :*

Avec **while** de la même manière que pour créer une boucle avec len(ma\_liste) methode longue a écrire

Avec **for** de la manière for elt in ma\_liste : / print(elt) elt va prednre successivement les valeur de la liste ne permet pas de savoir ou sont placés les élément dans la liste

**Enumerate()** affiche des tuplesLe forme est : **for nom\_de\_variable\_a créer in enumerate(ma\_liste) :**

Quand on utilise **enumerate** on capture l’indice et l’élément dans deux variables distinctes.

On utilise **enumerate** dans une boucle **for** comme suit (i et elt sont des variables):

ma\_liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

for i, elt in enumerate(ma\_liste):

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

À l'indice 0 se trouve a.

À l'indice 1 se trouve b.

À l'indice 2 se trouve c.

À l'indice 3 se trouve d.

À l'indice 4 se trouve e.

À l'indice 5 se trouve f.

À l'indice 6 se trouve g.

À l'indice 7 se trouve h.

Plus simplement cela donne :

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 obtient l’affichage de la position puis de la valeur (indice, ‘élément’)

**.split()** permet de transformer une chaîne en liste (ex : ma\_chaine = « bonjour à tous » / ma\_chaine.split(« ») / [‘bonjour’, ‘à’, ‘tous’]

Avec split on peut donc découper la chaine au paramètre donné (). La méthode va découper la chaine en prenant en fonctions des espaces, tabulation et sauts de ligne.

**.join()** réalise l’inverse de split, transforme une liste en chaine.

On peut donner des paramètres inconnus à une fonction. Dans ce cas on met **\*** avant le nom du paramètre qui contiendra les inconnus

(ex : def fonction\_inconnue(nom, prénom, \*commentaires) :

Python va placer ces paramètres dans un tuple.

On peut transformer ce tuple en liste pour en permettre l’affichage et la modification.

(ex : def print(\*values, sep=’ ‘, end=’\n’) : ) cette fonction simule print.

On peut aussi transformer une liste en paramètre de fonction. (ex : liste\_de\_parametres = [1, 4, 9, 16, 25, 36] / print(\*liste\_des\_parametres) / 1 4 9 16 25 36

Si on n’avait pas mis \* l’affichage rendue par print aurait été [1, 4, 9, 16, 25, 36]

On utilise une étoile **\*** dans les deux cas. Si c'est dans une définition de fonction, cela signifie que les paramètres fournis non attendus lors de l'appel seront capturés dans la variable, sous la forme d'un tuple. Si c'est dans un appel de fonction, au contraire, cela signifie que la variable sera décomposée en plusieurs paramètres envoyés à la fonction.

Compréhension de liste :

Parcours simple :

On parcourt une liste pour en renvoyer des paramètres modifiés. Pour cela on utilise une boucle for (ex : liste\_origine = [0, 1, 2, 3, 4, 5] / [nb \* nb for nb in liste\_origine] / [0, 1, 4, 9, 16, 25]

On a demandé à capturer dans la variable nb la mise au carré de cette même variable. En utilisant une boucle for elle parcours donc les éléments de la liste\_origine et les met au carré et les renvois dans une liste.

Filtrage avec un branchement conditionnel :

On ajoute une condition (if) au cas précèdent.

(ex : 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]

Ici on demande à la boucle **for** de renvoyer dans nb seulement (if) les éléments divisibles par deux (donc dont le reste de division est égal à zéro). La liste renvoyé est donc deux celle des chiffres pairs.

nouvelle\_squence = [element for element in ancienne\_squence if condition]

*Tuples :*

**Les tuples sont des listes qu’on définit mais qui sont ensuite non modifiables. Elles sont entre () alors que les listes sont entre []**

Si on créer un tuple contenant un seul élément on doit quand même mettre la virgule après.

*Dictionnaires :*

**Dict()** créer un dictionnaire (ex : **mon\_dictionnaire = dict()** ) ou bien **mon\_dictionnaire = {}**

On peut aussi le créer comme ceci : **placard = {"chemise":3, "pantalon":6, "tee-shirt":7}** ici le dictionnaire est déjà rempli. Il est important de placer les : après le nom de la clé et les virgules entre valeur et clé suivante. Par contre pour appeler lé clé il est important de les nommées entre **[]**.

Comme pour en supprimer une (ex : placard = {"chemise":3, "pantalon":6, "tee shirt":7} / **del** placard["chemise"] )

Pour effectuer une suppression de clé on utilise de préférence la méthode de dictionnaire **.pop** (ex : placard = {"chemise":3, "pantalon":6, "tee shirt":7} / placard.pop("chemise") / 3 ). La différence est que la méthode .pop renvoie la valeur de la clé supprimé.

Une liste est entre [] un tuple entre () et un dictionnaire entre **{}**

*On accède à un dictionnaire avec une clé* (ex : mon\_dictionnaire[« pseudo »] = « 6ori1 » / mon\_dictionnaire[« mot de passe »] = « 15481 »

Pour obtenir la valeur il suffis d’appeler mon\_dictionnaire[« mot de passe »] et la valeur de la clé est renvoyée. Ici 15481.

On ne peut pas mettre deux clé identique. Si une nouvelle clé qui existe déjà est créer elle remplace la première. Dans l’exemple on utilise des chaines de caractère mais ce n’est pas obligatoire.

En tapant juste mon\_dictionnaire Python renvoie les clé suivie des valeurs entre {} et séparée par des ,

On peut stocker des fonctions dans des dictionnaires pour cela (ex : **dictionnaire[« clé »] = nom de la fonction** ) au moment de l’appel on doit le noter **dictionnaire[« clé »]()**

Parcours des clés :

Soit avec une boucle **for** (**for** nom\_de\_variable\_qui\_va\_se\_créer **in** nom\_de\_dictionnaire **:**) comme pour les liste. Mais cela renvoie uniquement les clés et dans un ordre aléatoire car les dictionnaires ne sont pas des structures ordonnées.

**.keys()** (ex : **for** nom\_de\_variable\_qui\_va\_se\_créer **in** nom\_de\_dictionnaire**.keys() :** )

Parcours des valeurs :

**.values()** (ex : **for** nom\_de\_variable\_qui\_va\_se\_créer **in** nom\_de\_dictionnaire**.values() :** )

On peut utiliser cette méthode dans une condition if (ex : **if** 21 **in** fruits**.values():** / print("Un des fruits se trouve dans la quantité 21.") / Un des fruits se trouve dans la quantité 21.

Parcours des clés et valeurs simultanément :

**.items()** (ex : fruits = {"pommes":21, "melons":3, "poires":31} / **for** cle, valeur **in** fruits**.items():** / **print("**La clé **{}** contient la valeur **{}**."**.format(**cle, valeur**))** / La clé melons contient la valeur 3. / La clé poires contient la valeur 31. / La clé pommes contient la valeur 21.

Les valeurs sont renvoyées sous forme de tuple.

On peut récupérer les paramètre non précisés d’une fonction dans un dictionnaire en faisant précéder le paramètre de \*\* alors qu’en liste c’est avec \* (ex : **def** fonction\_inconnue(\*en\_liste, \*\*en\_dictionnaire): )

*Utiliser des fichiers :*

Changer le répertoire courant : import os / os.chdir(« C:/ nom du répertoire déjà créer ou l’on veut travailler » sur le gros Pc le rep est C:/Travaux\_Python

Ouverture :

**R** (read) ouvre le fichier en lecture

**W** (write) ouvre le fichier en écriture et écrase le contenu précédent. Si le fichier n’existe pas il est créé.

**A** (append) ouvre le fichier en ajout, ce qu’on ajoute va se mettre a la fin du fichier. Le contenu n’est pas écrasé. S’il n’existe pas le fichier est créer.

**Open()**

**Mon\_fichier = open(« fichier.txt », « r »)**

Les fichiers .txt sont de la classe **TextIoWrapper**

Mon\_fichier.**close()** pour fermer le fichier

Mon\_fichier = open(« fichier.txt », « r ») / contenu = mon\_fichier.**read() /** print(contenu)

Pour afficher l’intégralité d’un fichier.

**L’affichage est rendu en chaîne. On peut donc demander tout ce qu’on pourrait demande a une chaine !!**

**With** créer un « context manager » **s’assure de l’ouverture et de la fermeture d’un fichier.** S’il n’est pas ferme le fichier est fermé automatiquement

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

Module pickle : **import pickle**

Pour créer un fichier oui on pourra stocker des objets

**.Pickler()**

(ex : with open(‘donnees’, ‘wb’) as fichier : / mon\_pickler = pickle.Pickler(fichier) créer un fichier pickler ou on va pouvoir ensuite stocker des objets.

*Ici le dossier ou le fichier est créer est ‘donnees’ , il est nomé fichier et on demande de l’ouvrir en ecriture (w) en binaire (b).*

On peut créer un dictionnaire par exemple : score = { « joueur 1 » : 5, « joueur 2 » : 35, « joueur 3 » :20, } / with open(‘donnees’, ‘wb’) as fichier : / mon\_pickler = pickle.Pickler(fichier) / mo\_pickler.dump(score)

On **dump** (vide) le dictionnaire score dans le fichier pickler.

Pouir lire le fichier il faudra utiliserr **.Unpickler()** et le mot clé **load**

TP1 ALAIN :

Les variables doivent être déclarée au début.

On peut ensuite les appeler dans une fonction avec le mot clé **global** *nom de la variable.*

Mettre une variable en paramètre de fonction permet de l’appeler depuis une autre fonction pour l’utiliser ou en utiliser la valeur.

Pour créer une application L’appel des fonctions peux ensuite se faire par l’enchaînement :

Variable Z = nom\_fonction\_A

Nom\_fonction\_B(variable z)

Ici on créer une variable avec comme valeur fonction A

On appel cette variable pour l’utiliser en paramètre dans la fonction B.

Une boucle **while** peux être utiliser pour contenir ces lignes pour renvoyer au début de l’action (dans le tp la demande du nom et prénom) sans avoir recours a return.

Return est utilisé ici à la fin de la fonction a pour que les résultats soient récupérables pour la fonction b.

*Les classes :*

Les classes sont de sortes de super fonctions. Des mini programmes regroupant des fonctionnalités données et des méthodes d’applications qui lui sont propres.

En objet on utilise plus le global mais on passe en argument les variables que l’on veux utiliser.

@staticmethod

On définis une fonction inaltérable (static) que l’on pourra appeler par la classe.

@classmethod

Définit une méthode qui utilisera les variables définies dans la classe (l’équivalent d’un global au niveau de la classe).

Pour les utiliser on appellera la variable définie dans la classe par **cls.nomdelavariable**

Pour appeler une méthode d’une classe on utilise **nomdeclasse.nomdemethode**

*Diagramme de flux :*

Pour préparer au code. Vue de l’esprit du fonctionnement de l’application. Ça commence et fini par un ovale. Les actions sont représentées par des rectangles. Les question/choix par des losanges menant à deux réponse oui ou non. (Ex : l’utilisateur appuie sur le bouton il se passe ça. S’il n’appuie pas il se passe ceci.

Draw.io un site qui permet de réaliser des diagrammes de flux.