RESUME DES COMMANDES DE BASE EN PYTHON

(Python version 3)

LES VARIABLES

Le typage est dynamique : la variable est créée et affectée en même temps et reçoit son type au vu de la valeur déclarée. Le type n'existe donc pas préalablement à la valeur.

⚠ On évitera des noms de variables comportant des caractères accentués.

On peut faire des affectations multiples :

```
>>>  ma_variable1 , ma_variable2 = valeur1 , valeur2
>>>  ma_variable1 , ma_variable2 =  variable2 , variable1  # échange de deux variables sans avoir
     besoin d'une variable auxiliaire.
```

On peut supprimer, réinitialiser une variable :

```
>>>  del ma_variable  # un appel à ma_variable provoque une erreur.
>>>  ma_variable = None # un appel à ma_variable ne provoque pas d'erreur, mais si ma_variable est
     incluse dans un calcul, il y aura une erreur.
```

On peut tester le type d'une variable ou demander quel est son type :

```
>>>  type(ma_variable)  # retourne la classe dont fait partie ma_variable
>>>  isinstance(ma_variable, classe)   # classe ∈ {int, float, str, ... }
```

On peut demander l'adresse en mémoire d'une variable :

```
>>>  id(ma_variable)
```

Les variables sont

soit modifiables (mutables) : on peut modifier leur valeur sans changer l'emplacement en mémoire

soit non modifiables (immutables) : changer leur valeur nécessite de réallouer un emplacement en mémoire.

Variables modifiables : listes, dictionnaires

variables non modifiables : entier, flottant, chaînes, tuples.

Conversions : on peut modifier le type d'une variable quand cela est possible.

float(mon_entier) convertit un nombre entier en nombre flottant.

complex(mon_flottant) convertit un flottant en complexe.

int(mon_flottant) donne l'entier le plus proche (en direction de 0) du nombre flottant.

int(ma_chaîne) ou float(ma_chaîne) donne l'entier (ou le flottant) correspondant si la chaîne de caractères est interprétable comme un entier (ou comme un flottant).

LES NOMBRES

Les entiers (type int) : calculs en précision illimitée.

Les nombres flottants (type float) : codés en double précision sur 8 octets (précision 15 à 16 chiffres significatifs).
Il y a différents modes d'arrondi (round pour l'arrondi au plus proche, int pour l'arrondi en direction de 0).

Les complexes (type complex) : de la forme a + bj où a et b sont des flottants.

Les bouléens (type bool) : Le résultat d'un test est soit True, soit False. Ce sont des nombres bouléens.

```
>>>  2**2 = = 4 # retourne Vrai.
>>>  3*0.1 = = 0.3 # retourne Faux à cause des erreurs d'arrondi.
>>>  (1+1j)**2 == 2*1j # retourne Vrai.
>>>  1/3 != 0.33 # retourne Vrai , != signifiant "différent".
```

LES CHAINES DE CARACTERES

Une chaîne est une suite finie de caractères alphanumériques (chiffres, lettres, symboles). Elle est de type *immutable* (ou immuable) : une fois créée, une chaîne ne peut plus être modifiée. Toute opération effectuée sur une chaîne aboutit donc à la création d'une nouvelle chaîne.

Syntaxe : une chaîne est une suite de caractères délimités par des apostrophes (quotes) ou guillemets (double quotes) ".

```
>>>  c1, c2 = 'python' , 'Python'
```

La *casse* est importante : il faut distinguer minuscules et majuscules.

```
>>>  ma_chaîne = ""  #chaîne vide
```

Insertion de caractères spéciaux : si on veut insérer un caractère réservé à l'intérieur d'une chaîne, on le fait précéder d'un \ (backslash). Liste des caractères spéciaux : " , ' , \ , % .
\n permet un retour à la ligne pour éditer les chaînes longues sur plusieurs lignes.
\t insère une tabulation.

Insérer des données numériques formatées dans une chaîne :
chaîne.format(suite des éléments à inclure) (valable en Python3)
A chaque emplacement de la chaîne où on veut insérer une donnée numérique, on place {}. Il y a donc autant d'éléments à inclure que de {}.
On peut préciser le formatage à l'intérieur des accolades : {:d}, {:n.pf}, {:s}, ...

```
>>>  rayon = 2.165
>>>  'L\aire d\'un cercle de rayon {:.2f} cm est {:.2f} cm² '.format( rayon, math.pi*rayon**2)
```

on obtient :

l'aire d'un cercle de rayon 2.17 cm est de 14.73 cm²

Méthodes sur les chaînes
Les chaînes sont des *instances* de la *classe* str. La classe dispose de méthodes pour agir sur les attributs de la chaîne.
Mais rappelons que les chaînes sont immutables et que ces méthodes vont fournir de nouvelles chaînes.
dir(str) liste toutes les méthodes de la classe.

- Opérations
  - chaîne1 + chaîne2 *concaténation* des deux chaînes (les chaînes sont mises bout à bout) .
  - chaîne1 += chaîne2 équivaut à chaîne1 = chaîne1 + chaîne2 (concaténation puis affectation)
  - chaîne*n renvoie n copies de chaîne concaténées.
  - chaîne.lower() et chaîne.upper() transforme tout en minuscules et en majuscules respectivement.
  - etc ...

```
>>>  x = '45'
>>>  x = x + '67' # donne '4567' ; une nouvelle chaîne de nom x est créée à un nouvel
       emplacement mémoire (effectuer id(x) avant et après l'instruction).
>>>  ':-)'*3  # renvoie ':-) :-) :-)'
```

- Extraction (techniques de slicing pour les chaînes)
  - extraire un seul caractère : chaîne[i]
    ⚠ **les index commencent à 0** ! le premier caractère s'obtient donc par chaîne[0].
    ⚠ ma_chaîne[i] = nouveau_caractère renvoie une erreur : une liste est immutable
    Les index peuvent être négatifs : chaîne[−1] retourne le dernier caractère.
  - la syntaxe générale est ma_ chaîne[start : end : step] :
    on extrait de ma_chaîne les caractères de l'index « start » à l'index « end » (mais end exclu) avec le *pas* « step ».
    Si start est omis, par défaut c'est 0 ; si end est omis, par défaut on va jusqu'au bout de la chaîne.
    Enfin step = 1 par défaut.

```
>>>  'abcdefghijkl'[1:10:2]  # donne 'bdfhj'.
>>>  x = 'informatique'
>>>  x[ :5] , x[::2] , x[2 :]  # retourne ('infor' , 'ifraiu' , 'formatique').
```

- Informations et sous-chaînes
  - len(chaîne) calcule la longueur de la chaîne.
  - ma_sous_chaîne in ma_chaîne : teste si « ma_sous_chaîne » est présente dans ma_chaîne.
    Réponse bouléenne.
  - ma_chaîne.count(ma_sous-chaîne) : compte le nombre d'occurrences de ma_sous_chaîne dans ma_chaîne.
  - chaîne.isalpha(), chaîne.isdigit() teste si le caractère est alphanumérique (resp. numérique). Réponse bouléenne.
  - ma_chaîne.find(ma_sous-chaîne) recherche d'une sous-chaîne : renvoie l'indice de la première occurrence de la sous-chaîne.
  - chaîne.replace(chaîne ancienne, chaîne nouvelle) remplace dans « chaîne » toutes les occurrences de « chaîne ancienne » par « chaîne nouvelle ».
  - ma_chaîne.split() ou ma_chaîne.split(séparateur) : découpe la chaîne à chaque occurrence du séparateur (espace par défaut) et renvoie la liste des sous-chaînes qui apparaissent ainsi.
    On a aussi list(chaîne) qui renvoie la liste de tous les caractères composant la chaîne.
  - séparateur.join(liste de sous-chaîne) effectue l'opération inverse.

```
>>>  'car' in 'carte, truc, artiste, écart'  # retourne True.
>>>  'carte, truc, artiste, écart' .count('car')  # retourne 2.
>>>  'carte'.replace('te','pe')  # renvoie 'carpe'.
>>>  '01407809'.split('0')  # retourne  ['', '14', '78', '9'].
>>>  list('carte')  # renvoie ['c','a','r','t','e'].
>>>  '_'.join(['Loire','Atlantique'])  # retourne 'Loire_Atlantique'
```

Caractères accentués

Le jeu de caractères de base est le code ASCII (american standard code for information interchange). Chaque caractère est codé sur un octet (256 possibilités). En fait le code ASCII monopolise seulement les 128 premiers octets. Les autres sont utilisés pour coder des caractères supplémentaires propres à chaque langue. Chaque codage est normalisé. En occident, on utilise la norme Latin-1 (ISO-8859-1) pour manipuler tous les caractères accentués (é , è , ï, ù, etc ...).
La commande chr(n) retourne le caractère ascii codé par l'entier n (≤ 255). Par exemple, chr(233) retourne 'é', en effet, 'é' se code en hexadécimal par e9, donc par l'entier 14×16 + 9 = 233. Opération inverse : ord('é') retourne 233.

La norme utf8 permet le codage au format unicode. Ce format est compatible avec le codage ascii et la norme Latin-1 mais il est conçu pour permettre le codage de tous les caractères utilisés dans toutes les langues, un identifiant numérique étant attribué à chaque caractère.

Données rentrées au clavier

La commande raw_input permet une interaction de l'utilisateur avec le programme via le clavier.
Syntaxe : ma_variable = raw_input('texte à l'adresse de l'utilisateur')
Ce qui est enregistré est forcément au format str. On peut le convertir ensuite, si possible, au format entier, float, etc .

LES LISTES

Une liste est un *conteneur* : elle contient des objets de type divers (nombres, chaînes, autres listes, ...). Une liste est un objet *itérable* : ses éléments sont indexés et peuvent être parcourus par un *itérateur*. Une liste est *mutable* (ou modifiable) : tout en gardant le même identifiant et le même emplacement en mémoire, on peut modifier son contenu.

Syntaxe : une liste est une suite délimitée par des crochets et composée d'objets de type quelconque, séparés par des virgules. Chaque objet est repéré dans la liste par son *index* (qui débute à 0).

```
>>>  ma_liste = []  # liste vide.
>>>  notes = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]  #  liste d'entiers.
>>>  notes = list(range(21))  # retourne [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
>>>  appreciations  = ['TB' , 'B' , 'AB' , 'M' , 'I']  # liste composée de chaînes.
>>>  departements = [[29, 'Finistère] , [35, 'Ille-et-Vilaine'] , [53, 'Mayenne']]  # le type des objets d'une
     liste n'est pas forcément homogène en Python. On peut en particulier imbriquer des listes.
```

Méthodes sur les listes

Les listes sont des instances de la classe list ; dir(list) énumère toutes les méthodes de la classe.
On décrit un certain nombre de méthodes.
Avec les plus basiques d'entre elles (ajouter ou retirer un élément, longueur de la liste), on peut facilement écrire la plupart des autres à l'aide de boucles élémentaires.

- Extraction : on dispose des mêmes techniques de slicing que pour les chaînes.
  - liste[i] renvoie le terme d'index i dans la liste. Les index commencent à 0.
  - liste[start : stop : step] renvoie la liste qui commence au terme d'index « start », se termine au terme d'index « stop » (mais stop exclu) avec un pas « step ».

```
>>>  notes[0] renvoie12
>>>  notes = [12,9,11,15,10,14 ]
>>>  notes[0:3] renvoie [12,9,11]
>>>  notes[2:] renvoie [11,15,10,14]
>>>  notes[– 3,–1] renvoie [15,10]
```

- Informations sur une liste :
  - len(liste) : nombre d'éléments de la liste.
  - x in liste : teste si l'objet x fait partie de la liste.
  - max(liste) : pour une liste composée de nombres, renvoie le plus grand élément. idem pour min(liste)
  - liste.index(x) : renvoie l'index de l'objet x dans la liste (index de la première occurrence si x présent plusieurs fois.
  - liste.count(x) : renvoie le nombre d'occurrences de x dans la liste.

```
>>>  prenoms = ['Didier', 'Eric', 'Marie', 'Bruno', 'Eric', 'Damien', 'Nathalie']
>>>  len(prenoms) renvoie 7
>>>  prenoms.count('Eric') renvoie 2
>>>  prenoms.index('Eric') renvoie 1
```

- Opérations
  - range(start, end, step) où start, end, step sont des entiers. Renvoie une liste d'entiers commençant à l'index start inclus, se terminant à l'index end exclu avec le pas step. Ce sont donc les termes de la suite arithmétique de raison step et de premier terme start.
    Il s'agit en fait d'une liste virtuelle (type range); list(range(...)) fait la conversion en liste.
    C'est une commande utile dans les boucles.
  - liste1 + liste2 : renvoie une liste unique composée dans l'ordre, des éléments de liste1 suivis de ceux de liste2.
  - ma_liste*n : liste unique obtenue en répétant n fois ma_liste.

```
>>>  list(range(1,10,2)) renvoie [1,3,5,7,9]
>>>  [0]*5 renvoie [0,0,0,0,0]
>>>  [0,1,2] + [3,4,5,6] = [0,1,2,3,4,5,6]
```

- Modifications
  - ma_liste[i] = nouveau_objet : modifie l'objet d'index i dans la liste. Rappel : une liste est mutable, on peut modifier ses éléments. Cette opération est impossible avec une chaîne.
  - ma_liste.append(nouveau_élément) : rajoute nouveau_élément en fin de liste.
  - ma_liste.pop() : renvoie le dernier de ma_liste et le supprime dans la liste.
    ma_liste. pop(i) : retourne l'élément d'index i et le supprime dans la liste. pop et append permettent d'implémenter une pile.
  - ma_liste.insert(index, nouveau_objet) : rajoute nouveau_objet à l'index spécifié. Equivaut à liste[index :index] = nouveau_objet
  - del(liste[i]) ou bien liste[i : i+1]= [] : suppression de l'objet d'index i.

- modifications simultanées : ma_liste[i :j] = ma_sous_liste (les objets d'indice i à j – 1 sont remplacés par ceux de ma_sous_liste). Si i = j, cela revient à insérer; si ma_sous_liste a moins de j – i éléments, ceux restant sont supprimés.
  Suppression simultanée : liste[i :j] = [].

- liste.remove(x) : supprime la première occurrence de x dans la liste.
- liste.sort() ou bien liste.sort(fonction de comparaison) : trie la liste par ordre croissant (par défaut ou à l'aide de la fonction de comparaison préalablement définie).
  ⚠ La liste triée remplace la liste initiale. On dit que le tri se fait *en place*.
- liste.reverse() : renverse l'ordre des éléments
- sum(liste) : calcule la somme des éléments de la liste (si cela a un sens).

```
>>>  Bretagne, villes = [29,22,56,35] , ['Quimper', 'St-Brieux', 'Vannes', 'Rennes']
>>>  Bretagne.append(44)  # la liste Bretagne est désormais [29,22,56,35,44]
>>>  Bretagne.pop()  # renvoie 44 et la liste Bretagne est désormais  [29,22,56,35]
>>>  villes.insert(1,'Brest')  # villes est désormais ['Quimper', 'Brest', 'St-Brieux', 'Vannes', 'Rennes']
>>>  villes[−1] = 'Saint-Malo'  # villes est  désormais ['Quimper', 'Brest', 'St-Brieux', 'Vannes',  'Saint-Malo']
>>>  Bretagne.reverse()   # Bretagne est désormais [35,56,22,29] et l'adresse en mémoire n'a pas changé.
```

- Duplication : ⚠ si on veut faire une copie de liste1, on a envie d'écrire liste2 = liste1. Mais ce n'est pas une vraie copie de liste1, c'est un *alias*, c'est à dire un autre nom pour désigner le même emplacement en mémoire. Toute modification sur l'une des deux listes se répercute sur l'autre.
  On peut écrire liste2 = liste1[:] pour obtenir une vraie copie de liste1. On peut ensuite modifier liste2 sans toucher à liste1.
  Essayer liste = [[True]*3]*3 puis l[0][0] = False et regarder l'effet sur l !!!

Instructions de branchement

Test simple

```
SI < expression bouléenne > ALORS
|          < instructions >
FIN SI
```

Test « si .. alors … sinon »

```
SI < expression bouléenne > ALORS
|          < instructions 1 >
SINON
|          < instructions 2 >
FIN SI
```

Tests multiples « si … alors si … sinon … »

```
SI < expression bouléenne1 > ALORS
|          < instructions1 >
SINON SI < expression bouléenne 2> ALORS
|          < instructions 2 >
SINON
|          < instructions 3 >
FIN SI
```

Traduction en Python

```
if condition1 :
|       instructions1
elif condition2 :
|       instructions2
elif condition3 :
|       instructions3
...
else :
        instructions
```

Ne pas oublier l'indentation ni les : en fin de ligne.

L'égalité se teste par = =
La différence se teste par !=

Opérateurs bouléens
NOT, AND, OR
NOT est prioritaire sur AND, lui même prioritaire sur OR.
AND et OR sont paresseux :
dans condition1 OR condition2, si condition1 est satisfait, condition2 n'est par évaluée.
dans condition1 AND condition2, si condition1 est faux, condition2 n'est par évaluée.

LES BOUCLES

Boucles inconditionnelles
Le nombre d'itérations est fixé à l'avance.
Syntaxe :

```
for i in liste_indices :
    instructions dépendant a priori de i
```

La variable i qui contrôle le nombre de passages dans la boucle, est une variable muette.
Si Liste_indices contient des entiers consécutifs, la liste est générée à l'aide de range.
On peut utiliser n'importe quelle liste a priori, pourvu que les calculs dans le corps de la boucle ait un sens pour tous les éléments de la liste. N'importe quel objet itérable peut d'ailleurs servir.

```
>>>  somme = 0
     for i in range(1:1000):
          somme +=1/i**2
     print(somme)
>>>  somme = 0
     liste_indices = [i**2 for i in range(1:1000)]
     for i in liste_indices:
          somme += 1/i
     print(somme)
```

```
>>>  compteur = 0
     for c in 'l\'encyclopédie des mathématiques':
          if  c == 'e':
               compteur +=1
     print(compteur)  # on compte le nombre de 'e' dans la chaîne. On trouve 4.
```

On peut construire des listes par compréhension à l'aide de for :

```
>>>  [x**2 for x in [0,1,2,3,4]]
>>>  [x**2 for x in range(100) if x%3 ==0]  # on sélectionne les indices à l'aide d'un
     test.
```

Boucles conditionnelles

On ne connaît pas le nombre d'itérations à l'avance. L'arrêt repose sur un test bouléen dont la valeur dépend de l'état des données. S'il n'est jamais satisfait, la boucle ne s'arrête jamais.

Les problèmes de terminaison des programmes sont liés au boucles conditionnelles.

Syntaxe :

```
while condition :
     instructions
```

La condition bouléenne doit toujours être évaluable. Les variables qui la composent doivent donc être initialisées. Si la condition porte sur les termes d'une liste, il faut faire attention aux problèmes éventuels de débordement.

Mot clé « break » pour interrompre prématurément une boucle.

FONCTIONS

```
def  ma_fonction(paramètre1, paramètre2, ... ) :
     " description de la fonction "
     instructions  # corps de la fonction
     return résultat
```

FICHIERS TEXTE

On peut créer des fichiers texte à accès séquentiels.
Pour créer un fichier texte ou bien ouvrir un fichier existant :
>>> f = open(nom fichier , option) nom fichier est une chaîne de caractères se terminant par .txt
avec les options :

- 'w' fichier en écriture, si le fichier existe déjà, il est écrasé
- 'r' fichier en lecture seule
- 'a' on peut écrire mais à la suite du texte existant (a = append)
- 'r+' lecture et écriture
- 'b' mode binaire

Le fichier est créé par défaut, dans le même répertoire que le script qui le construit.
On peut rajouter comme option encoding = 'latin1' ou bien 'utf8' pour spécifier le système d'encodage.

Ecriture d'une chaîne :
>>> f.write(chaîne)
Saut de ligne :
>>> f.write('\n')

On ne peut pas écrire de nombre, seulement des chaînes. Convertir avant les nombres : str(nombre)

On lit le texte de façon séquentielle : il y a trois modes de lecture

- lecture en une seule fois : f.read()
- lecture d'un nombre fixé de caractères à partir de la position courante du « curseur » de lecture : f.read(n)
- lecture ligne par ligne :
  f.readline() lecture d'une seule ligne
  f.readlines() liste de toutes les lignes

Fermer le fichier :
>>> f.close()