Mémo Python

- Python est sensible à la casse, aux espaces en début de ligne. *Version 2.x et 3.x pas 100 % compatibles*.
- Commentaires : `#` jusqu'en fin de ligne, pas de commentaire multi-lignes, sauf pour documenter les fonctions avec les paires de trois guillemets doubles : `"""Descriptif ici, y compris avec des sauts de ligne pris en compte à l'affichage ..."""`.
- Le `\` permet de poursuivre sur la ligne suivante comme s'il n'y avait pas de saut de ligne.
- Les blocs d'instructions sont indiqués par l'indentation (précédés de `:` en fin de ligne précédente). Ne pas mélanger tabulation et espaces, faire remplacer la tabulation par 4 espaces dans l'éditeur.

Mots réservés

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

Variables

Le type est donné dynamiquement, par exemple lors d'une affectation.

Nombres

Entiers : `int` ; réels (représentation non exacte en « flottants ») : `float`

```
entierLu = int( input("Saisis un entier STP") )
```

Opérations : `+`, `-`, `*`, `/` (**la division en Python3.*x* retourne un `float`** ; en en 2.*x*, la division sur des entiers est la division entière), `//` (division entière), `%` (modulo, reste dans la division entière).
On peut « résumer » `a = a+2` avec `a += 2`, (idem `-=`, etc.) mais il n'y a pas de `p++`.

Conversions chaînes / nombres : `chaine=str(nbre_ou_liste)` `n=int(chaine)` `x=float(chaine)`

Chaînes de caractères

Séparateurs : `"` ou `'` (ou `"""` pour le « multi-ligne »). Le premier caractère est le n°0 : `chaine[0]`.
Pas de type « caractère » (unique) comme en C par exemple (on utilise une chaîne de longueur 1).
Chaînes non modifiables localement, par affectation directe (pas : `chaine[2]="t"`...)

Opérations : `+` pour concaténer ; si `mot='ab'*3`, alors `mot` vaut `'ababab'`...

```
majuscules=chaine.upper(); longueur=len(chaine)
liste_decoupe = chaine.split(separateur)  # ou par lignes : liste = chaine.splitlines()
ch2 = ch.replace(ch_init,ch_de_rempl)     #    compte = ch.count(mot)
# pour les afficher :
print(chaine, var2, ch3, sep=' ~ ', end='')  # séparateur/fin par défaut : " " / "\n"
```

Listes (pour les « tableaux » notamment)

Ordonnées, éléments de types différents possibles : `liste=[1,'deux',liste2,['a','e','i','o']]`

```
liste = []                  # initialisation indispensable pour la suite
liste2 = [ i*i for i in range(5) ]   # ici :  [0, 1, 4, 9, 16]
                            # syntaxe : [ element for var in range(nb_val) ]
liste.append(el)            # ajoute el en fin de liste, list2=liste.
liste1.extend(list2)        # concatène deux listes (⇒ ajout de multiples valeurs OK)
for el in liste:            # il y a aussi     if el in liste:
     print(liste.index(el), el)          # pour obtenir sa position ; on part de 0
  # pour enlever des éléments :
del liste[3]
liste.remove(el)
  # pour trier :
liste.sort(liste1)
liste.reverse(liste1)
```

Tableaux de dimension > 2 : on imbrique des listes ; avec `table=[['a','b'],['c','d']]` le `table[0][1]` vaut `'b'` et `table[0]` vaut la sous-liste `['a','b']`. Attention *on numérote toujours en partant de* **0**...

Slicing : `listeOuChaine[2:7]` va du n°**2** (en partant de 0) au n°**6** < **7**. Autres : `debut[:7]` ou `fin[3:]`
En partant de la fin : `dernier=listeOuChaine[-1]; saufDernier=listeOuChaine[:-1]`

Attention, **pour les listes, l'affectation `b=a` ne copie que l'adresse en mémoire**, donc `a` et `b` désignent la même liste, si l'une liste est modifiée, l'autre aussi. Pour une **recopie**, utiliser `b=a[:]`

Chaînes ↔ listes :

```
listeDepuisChaine = list(lachaine) # liste des lettres de la chaîne
sep=':' ; chaine2 = sep.join(listeDepuisChaine)  # avec sep="", chaîne reconstituée
```

Structures de contrôle

Tests et booléens

`True` ou `False` (ou `None`), attention/majuscules. `0`, `0.0`, `""`, `()`, `[]`, `{}` valent `False`, les autres `True`.

Pour la logique : `or`, `and`, `not`. Pour les comparaisons : `<`, `>`, `<=`, `>=`, `!=` et `==` (attention, pas le `=`)

Instructions conditionnelles (structures de tests)

```
if test:
     # instruction(s)
elif test2:
     # instruction(s)
else:
     # instruction(s)
# Ici la suite, après les blocs du «si»
```

```
    # Variantes :
    # 1°) sur une seule ligne
    if test : # instruction unique
    # Ici la suite, après le «si»

    # 2°) Syntaxe alternative:
    min = x if x<y else y
```

Boucles « tant que... »

```
while test:
     # instruction(s)
```

Boucles «pour...»

```
for i in range(8) :
     # instruction(s) répétées ici 8 fois
# La suite ici, après le bloc du « for »
```

```
# ou :range(1,10) pour 1, 2, .., 9 < 10
# range(1,10,2) pour 1, 3, 5, .. 9 < 10
# range(9,0,-1) pour 9, 8, .., 1 > 0
```

Python permet en fait de parcourir ainsi tout «***iterable***», comme une chaîne, une liste, on peut utiliser par exemple `for caract in "ma chaine":` ou encore `for j in ['lun', 'mar', 'mer']:`

Sauts et sorties anticipées, boucle pseudo «répéter… jusqu'à ce que...»

```
continue   # Passe directement à l'itération suivante (boucles seulement)
break      # Fait sortir immédiatement du bloc d'instructions

while 1:   # Idem   « while True: »
    reponse = input("Ta réponse ?")
    if reponse != '':      # Exemple de condition de sortie en fin de boucle
        break
```

Fonctions

Fonction sans valeur de retour (procédure)

```
  # 1°) Définition («déclaration») des fonctions avant de pouvoir les utiliser :
def affiche3fois(txt) :
    for i in range(3) :
          print(txt)

def foncSansParametre() : # Portée des variables : la "déclaration" avec 'global'
    global score          #  permet ici de modifier cette variable "globale"
    print("J'incrémente la variable globale 'score' !")
    score += 1

  # 2°) Utilisation («appel») des fonctions
affiche3fois("affiche-moi trois fois !")
foncSansParametre()
```

Fonction avec valeur de retour

```
def f1(x) :           # 1°) Déclarations:
    y = 2*x           # Portée des variables : cette variable y est "locale" (*)
    return y

def premiere_lettre(chaine) :
    return chaine[0]

T = "Chaîne de test" # 2°) Appels aux fonctions:
print("Première lettre de la chaine T :", premiere_lettre(T) )
ordonnee = f1(33.5)
print(y)              # Message d'erreur : y n'est pas définie ici, cf.*
```

Modules

Première solution

```
import math, random
racine = math.sqrt(5)      # => préfixer les «fonctions» («méthodes», en fait, ici)
de_six_faces = random.randint(1,6)
```

Autre approche

```
from math import sqrt      # ou pour tout importer :    from math import *
racine_de_cinq = sqrt(5)   # => sans préfixer ; pose problème en cas de synonymes...
```