Informatique

A- **Le type entier et ses opérateurs:**

Les opérateurs classiques sur les entiers sont
• l'addition +, la soustraction −, la multiplication ×1
• division entière//qui à deux entiers associe un entier: 7//2 = 3
• la division réelle/qui à deux entiers associe un réel: 7/2 = 3.5
• l'opérateur mod qui renvoie le reste de la division entière: 7 mod 2 = 1, 11 mod 3 = 2

B- **Boucles pour:**

Lorsqu'on utilise la boucle « Tant que », on ne sait pas a priori combien de passages dans la boucle l'algorithme va effectuer. C'est pour cela on doit utilizer un compteur. En python c'est WHILE ….:
Cependant, on peut connaître à l'avance le nombre de passages. Dans ce cas, on peut utiliser la boucle «Pour ». La syntaxe de la boucle «Pour» en pseudo-code est la suivante:
pour i allant de n1 à n2
    faire instruction
fin
En python c'est for i in range(debut,fin,pas):

C- **Les listes:**

→ Une liste en python est une collection d'objets ordonnés.
→ En python les listes s'écrivent avec des crochets [ ] type(L)
→ Le nombre d'éléments dans une liste L est len(L)
→ Les éléments d'une liste sont numérotés de 0 à len(L)-1
→ Pour accéder au ième élément d'une liste L, on tape L[i]
→ Les listes peuvent être modifiées :
Pour ajouter un élément à la fin d'une liste : L.append('element') **ou** L=L+[x] **ou** L.extend([x])

→ Suppression d'éléments:

- ➢ y=L.pop() à supprime et retourne le dernier élément
- ➢ L.remove(x) à supprime la première occurrence de x
- ➢ del L[i] à supprime l'élément d'indice i

→ Plages d'indices (slices):

- ➢ L[i:j] à éléments de L d'indice i…j-1 (j exclu)
- ➢ L[:n] à n premiers éléments de L (indices 0…n-1)
- ➢ L[n:] à derniers éléments de L, à partir de l'indice n
- ➢ L[-n:] à n derniers éléments, sauf pour n=0

→ Definition en comprehension:

Notation: [formule(i) for i in …… if condition(i)] Créée une liste de la bonne taille et la remplit avec la formule. **Exemple :** carrés des entiers de -2 à 3 : Formule : [n**2 for n in range(-2, 1+3)]

→ summation/ maximum/ minimum:

- ➢ **Exemple:** somme des premiers carrés, sum([i**2 for i in range(1, 1+n)])
- ➢ **Exemple:** maximum de n(n-i) pour i entier dans 1..n : max([n*(n-i) for i in range(1, 1+n)])
- ➢ **Exemple:** nombre de i dans 1..n premiers avec n : len([i for i in range(1, 1+n) if pgcd(i,n) == 1])
- ➢ **Exemple:** un élément au hasard

```
from random import choice
L=[1,5,8,6,-4]
–alea = choice(L)
```

→Tri / renversement:

- ➢ Pour trier une liste méthode sort:

```
L = [3,-7,5,8,9,-5]
L.sort() print(L)
```

- ➢ Ou fonction sorted:

```
L = [3,-7,5,8,9,-5]
L2 = sorted(L)
```

- ➢ Même chose pour le renversement reverse **ou** reversed:

```
L=[1,5,8,6,-4]
print(reversed(L))
```

D- **Les tuples:**

→ Un tuple en python est une collection d'objets ordonnés.
→ En python les tuple s'écrivent avec des parenthèses ()
type(T)
→ Un tuple avec un seul élément a s'écrit (a,)
→ Le nombre d'éléments dans un tuple T est len(T)
→ Les éléments d'un tuple sont numérotés de 0 à len(T)-1
→ Pour accéder au ième élément d'un tuple T, on tape T[i]
→ Les tuples ne peuvent pas être modifies

E- **Les chaines de caractères:**

→ Une chaîne de caractère est une suite finie de caractères délimitée soit par des apostrophes, soit par des guillemets. Le type correspondant est str.
→ La barre oblique inversée \ (antislash) permet d'insérer certains caractères: **saut à la ligne \n**, **tabulation \t, apostrophe \' , guillemet \"** etc.).
→ La fonction str(n) convertit un nombre n au format chaîne de caractères.
→ La chaîne ch a len(ch) éléments numérotés à partir de 0: **ch[0], ch[1], ch[2], …**
→ Les chaînes ne sont pas modifiables.
→ Parcours des éléments d'une chaîne:

➢ Indices:
```
for i in range(len(ch)):
    print(ch[i])
```
➢ Direct:
```
for c in ch:
    print(c)
```

→ Découpage des chaînes (slices):

➢ ch[a:b]: chaîne constituée des éléments de ch d'indices a, … b-1 (b exclu)
➢ ch[:n]: n premiers éléments de ch (indices 0 à n-1)
➢ ch[n:]: derniers éléments de ch, à partir de l'indice

→ **Concaténation:** "L'université"+" de Tours" donne

"L'université de Tours"
→ **Multiplication:** "abc"*3 donne "abcabcabc".
→ L'opérateur in **teste l'appartenance** d'un élément à une chaîne: if x in ch: print(x,' est présent')
→ La méthode capitalize () renvoie une copie de la chaîne avec son premier caractère en majuscule et le reste en minuscule.
→ La méthode lower() renvoie une copie de la chaîne en lettres minuscules.
→ La méthode upper() renvoie une copie de la chaîne en lettres majuscules.
→ La méthode title() renvoie une copie de la chaîne où la première lettre de chaque mot est majuscule.
→ La méthode count(sub, start, end) donne le nombre d'occurrences de la chaîne sub sans recouvrement dans l'intervalle start : end. Les arguments facultatifs start et end sont interprétés comme pour des tranches.
→ La méthode find(sub, start, end) donne l'indice du premier caractère de la chaîne sub sans recouvrement dans l'intervalle start : end. Les arguments facultatifs start et end sont interprétés comme pour des tranches. La méthode retourne -1 si la chaîne recherchée n'a pas été trouvée.
→ s.rfind(sub, start, end) donne l'indice le plus élevé de la sous-chaîne sub dans la chaîne s[start:end]. Les arguments facultatifs start et end sont interprétés comme dans la notation des slices. La méthode donne -1 si sub n'est pas présente.
→ La méthode replace(old, new, count) renvoie une copie de la chaîne dont toutes les occurrences de la sous-chaîne old sont remplacés par new. Si l'argument optionnel count est donné, seules les count premières occurrences sont remplacées.
→ La méthode strip(chars) retourne une copie de la chaîne dans laquelle toutes les occurrences de tous les caractères de l'argument chars ont été supprimées en début et en fin de chaîne. Si l'argument chars est omis, les espaces et les sauts de ligne sont supprimés.

→ La méthode split(sep) renvoie la liste des mots de la chaîne, en utilisant l'argument sep comme séparateur de mots, l'espace par défaut. L'argument sep peut contenir plusieurs caractères.
→ s.join(iterable) concatène les éléments de l'argument iterable (par exemple type liste de chaînes, type chaîne de caractères) en mettant s entre.
→ La méthode format() s'utilise de plusieurs façons. Elle sert par exemple à afficher des nombres réels avec un nombre déterminé de décimales. **Exemple:** x=5.5897 print("x=", format(x,".3f")) affichera : x = 5.590
→ La méthode format() utilise aussi des champs de remplacement placés entre accolades. On peut accéder aux arguments par position. La chaîne à formater contient des champs {0}, {1}, {2} etc., qui sont remplacés par les valeurs formatées des arguments de la méthode format(). **Exemple:**
vil, dep, reg = 'Tours', 'Indre-et-Loire', 'Centre'
"{0} est dans l'{1}, en région {2}".format(vil,dep,reg) vaut "Tours est dans l'Indre-et-Loire, en région Centre."
→ Les accolades peuvent aussi contenir des indications de formatage des arguments. **Exemple:**
import math
"pi :{:9.5f}".format(math.pi) vaut "pi : 3.14159"
→ La fonction chr(i) retourne le caractère dont le code Unicode est l'entier i. **Exemple:** chr(65) vaut 'A'
→ La fonction ord(c) retourne le code Unicode du caractère c. **Exemple:** ord('A') vaut 65

F- **la complexité:**

Déterminer la complexité d'un algorithme, c'est évaluer comment le temps d'exécution augmente avec la taille des entrées. Dans cette partie nous donnerons simplement des exemples d'études de complexité d'algorithme.

G- **La recursivité:**

Une fonction est dite récursive si elle comporte dans son corps un appel à elle-même. La récursivité se rapproche du principe de

récurrence.
La définition d'une fonction récursive doit nécessairement comporter:

- ➢ Des valeurs de résultat pour un ensemble de cas de base (évaluation accomplie, sans appel à ellemême).
- ➢ Des valeurs en fonction de la fonction elle-même: appel à elle-même avec des paramètres différents (ou modifiés auparavant).

Les appels successifs doivent mener à l'un des cas de base en un nombre fini d'étapes.
→ Chaque appel à une fonction implique la copie, dans la pile d'exécution, du contexte de l'appel (paramètres, valeur de retour…).
→ S'il s'agit d'une fonction récursive, le nombre d'appels récursifs doit être fini et pas trop élevé pour éviter le débordement de la pile et donc l'arrêt du programme.
→ Exemple d'exception générée par RecursionError: maximum recursion depth exceeded in comparison → Python fixe une limite pour se prémunir contre un débordement de pile. On peut augmenter cette limite (sys.setrecursionlimit(limit)) mais il est plutôt conseillé d'essayer d'optimiser le code.
→ Certaines fonctions ne peuvent être définies que sous forme récursive…
→ Lorsqu'une fonction s'appelle indirectement:
f1 → f2, f2 → f3, …, fn → f1
il s'agit de récursivité indirecte (on dit aussi récursivité croisée).
Exemple:

```
def f1(…):
    … f2(…)
def f2(…):
    … f1(…)
```

L'ensemble des fonctions f1, f2 est globalement récursif.
→ Avantage des algorithmes récursifs: concision d'écriture.
→ Inconvénient: encombrement de la pile d'exécution,

ralentissement d'exécution.

→ Un algorithme qui ne comporte :

➢ qu'un seul appel récursif (en particulier pas d'appel avec un paramètre étant lui-même un appel : profondeur d'appel d'ordre 1).

➢ tel que l'appel est la dernière action effectuée et n'est pas argument d'une fonction ou opérande d'un calcul,

est appelé algorithme à récursivité terminale.

Exemple:

fonction f avec appel à elle-même à l'intérieur:

➢ f(n/2)*3 → non terminale

➢ f(f(n-2)) → non terminale

➢ return f(n-3) → terminale

→ La "dérécursivation" d'un algorithme n'est pas toujours possible. Elle l'est lorsque la récursivité est terminale.

H- **Ensembles:**

Un ensemble est une collection finie et sans doublon où l'ordre n'importe pas.

type set

ensemble vide: set() et non pas {}

→ Notations (ordre et doublons disparaissent toujours) :

➢ **En extension:** E = {0,1,4,9} vaut aussi {4,0,9,1}

➢ **En compréhension:** E = {n**2 for n in range(-3, 3+1)} vaut {0,1,4,9}

➢ **Par conversion de liste ou de séquence:** Si L = [9,4,1,0,1,4,9], E = set(L) vaut {0,1,4,9}

→ **Parcours d'un ensemble:** si E = {0,1,4,9} alors for x in E: print(x) affiche 0 1 4 9, pas forcément dans cet ordre.

→ **Nombre d'éléments de E** (cardinal): **len(E)**

→ Tester **si l'élément x est dans E** (x∈E): **x in E**

→ **Ajouter l'élément x à E: E.add(x)** (modifie E)

→ **Supprimer l'élément x de E : E.remove(x)**

→ Opérations ensemblistes (ne modifient ni E ni F) :

➢ **union: E | F** (éléments dans E ou dans F)

➢ **intersection: E & F** (éléments dans E et dans F)
➢ **différence: E – F** (éléments dans E mais pas dans F)

→ Tests:

➢ **égalité: E == F**
➢ **inclusion: E <= F** (E est-il inclus dans F?)

→ Si A est une liste, plus A est longue, plus le test x in A est lent, Si A est un ensemble, le test x in A est garanti rapide.

I- **Dictionnaire:**

Dictionnaire est une table qui associe des clefs et des valeurs.

→ Type: dict

→ Principe: 2 colonnes, on cherche une clef à gauche, on donne la valeur correspondante à droite

→ Les clefs forment un ensemble (ni doublon ni ordre)

→ Notation en extension: {clef:valeur, … }

→ Dictionnaire vide: {} ou dict()

→ **Recherche:**

table[clef]: valeur, exception si clef absente
table.get(clef,x):valeur, x si clef absente
clef in table: teste si la clef est présente

→ **Modification:**

table[clef] = val: ajoute la clef et la valeur si clef absente
modifie la valeur si clef présente
del table[clef]: supprime la clef, exception si absente

→ **Compréhension (ex : carrés parfaits et leurs racines):**

C = {n**2:n for n in range(0, 4+1)} C vaut {0:0, 1:1, 4:2, 9:3, 16:4}

→ **Parcours d'un dictionnaire** (clefs, ordre indéfini):

```
for c in D :
    print("clef:", c, "valeur:", D[c])
```

Soit d={"Arthur":15 , "Alfred":18 , "Anne":19}

➢ **Liste des clefs**: **list(d.keys())** liste des clefs du dictionnaire d **exemple :** print(list(d.keys())) affiche ["Arthur","Alfred","Anne"]

➢ **Liste des valeurs**: **list(d.values())** liste des valeurs du

dictionnaire d **exemple :** print(list(d.values())) affiche [15,18,19]

➢ **Liste des couples (clef,valeur) : list(d.items())** liste des couples (clef,valeur) du dictionnaire d

➢ **exemple :** print(list(d.items())) affiche [("Arthur",15), ("Alfred",18),("Anne",19)]

J- **Exceptions:**

→ Il arrive qu'un programme génère des erreurs (division par 0, dépassement d'indice dans un tableau…). Pour éviter un arrêt intempestif lors de l'exécution, on peut utiliser un gestionnaire d'erreurs appelé gestionnaire d'exceptions.

→ Une exception est un signal qui se déclenche en cas de problème. Si l'exception n'est pas capturée on obtient un message d'erreur et le plus souvent l'arrêt du programme.

→ **Exception:**

➢ NameError une des variables intervenant dans une instruction n'a pas été définie avant d'être utilisée.

➢ TypeError une des variables n'a pas le type adéquat pour l'opération demandée.

→ la **partie du code risquant de générer un problème est incluse** dans un bloc

try:

…

suivi **d'un ou de plusieurs bloc(s)**

except Exception1:

…

except Exception2 :

…

suivi éventuellement d'un bloc

finally:

…

→ Le bloc **finally** est optionnel, il contient des instructions qui seront exécutées quoi qu'il arrive, erreur survenue ou pas, et dans le cas d'une erreur survenue même si une erreur non prévue par les except se produit.
→ Attention l'ordre des blocs **except** a une importance, on les ordonne suivant l'ordre de risque d'apparition des erreurs.
→ On cerne au maximum les instructions risquant de lever l'exception dans le **try…**
→ On peut mettre un bloc **else** pour les instructions à suivre si aucune exception n'a été levée.
→ On peut récupérer le message correspondant à l'erreur levée avec le mot-clé **as.**
Exemple: except ZeroDivisionError as msg:
print("Une erreur hélas : ", msg)
affichera Une erreur hélas: division by zero
→ On peut lever soi-même une exception grâce au mot-clé **raise** suivi du type de l'exception.
Exemple:

```
b=8
try:
    a=b-8
    if( a==0 ):
        raise ZeroDivisionError("a est nul, pb pour la division future")
except ZeroDivisionError as precision:
    print("stop: ",precision)   affichera stop: a est nul, pb pour la division future
```

→ Le mot-clé **pass** permet de ne rien faire (il peut être d'ailleurs aussi utilisé en dehors du contexte des exceptions, dans une fonction par exemple…)

K- Fichiers:

→ Un fichier contient des bits (0/1) qui sont ensuite interprétés en fonction de l'application qui ouvre le fichier. Les fichiers

binaires contiennent souvent des en-têtes (octets de données au début du fichier) servant à identifier le type du fichier et d'autres informations descriptives. Les fichiers **texte** eux ne contiennent que des données de type texte (caractères, saut de ligne,…) et peuvent être ouverts avec un éditeur de texte, ils seront compréhensibles.

→ **L'accès à un fichier** est assuré par l'intermédiaire d'un **descripteur de fichier** créé à l'aide de la **fonction open(nom, mode)**. **Par exemple**, pour ouvrir un fichier texte nommé 'essai.txt' en lecture seule: **f = open('essai.txt', 'r')**

→ L'objet f est **un descripteur de fichier**, il établit le lien avec le fichier, toutes les opérations seront effectuées par son intermédiaire. Les possibilités dépendent du mode d'ouverture, précisé en paramètre: mode lecture, écriture.

→ **Après les traitements, le fichier doit être refermé par : f.close()**

→ On peut rencontrer **l'adjonction de t ou b** dans le mode d'ouverture pour **préciser s'il s'agit d'un fichier texte ou binaire:** f = open(fname, "rt")

→ **Pour ouvrir un fichier** on peut aussi utiliser **with**, **par exemple** pour ouvrir un fichier texte nommé 'essai.txt' en lecture seule:

```
with open('essai.txt', 'r') as f :
    print(f.read())
```

Les instructions qui suivent doivent être indentées. Dès que l'on sort de l'indentation, **le fichier est fermé, on n'a pas besoin d'écrire l'instruction f.close().**

→ **Les modes d'ouverture** de la fonction open sont des chaînes de caractères :

- ➢ **'r'** : ouverture en **lecture** seule.
- ➢ **'w'** : ouverture en **écriture** (si le fichier n'existe pas, il est créé, sinon son contenu est écrasé).
- ➢ **'a'** : **ouverture en ajout** (si le fichier n'existe pas, il est créé, sinon l'écriture s'effectue à la suite du contenu

existant).

→ La fonction **open(nom, mode)**peut provoquer une erreur d'entrée-sortie (fichier introuvable, disque plein etc.). Cette erreur peut être gérée par une exception.

```
try:
    with open('essai.txt', 'r') as f :
        ….
except IOError :
    …
```

→ La méthode **read()** lit un fichier texte et affecte son contenu dans un chaîne de caractères : s=f.read()

→ La méthode **readlines()** renvoie la liste des lignes d'un fichier texte : lignes=f.readlines() OU

```
with open('essai.txt', 'r') as f :
    for ligne in f :
        ….
```

→ La méthode **write()** écrit des chaînes de caractères dans un fichier texte. Elle n'ajoute pas de caractère saut de ligne '\n'. Les instructions :

```
f.write('Université de Tours\n')
f.write('Informatique')
```

permettent d'écrire 'Informatique' en dessous de 'Université de Tours'.

→ Le module **pickle** permet de sauvegarder dans un fichier, au format binaire, n'importe quel objet Python (liste, dictionnaire, tuple) puis de le restaurer sans manipulation particulière.

→ La méthode **dump()** permet de sauvegarder dans le fichier.

→ La méthode **load()** permet de restaurer l'objet sauvegardé

L- **Programmation orientée objet:**

→ En **programmation orientée objet** on s'intéresse d'abord aux objets concernés par un problème, plutôt qu'à l'action à effectuer pour résoudre ce problème. On part des objets intervenants dans l'action et on définit des méthodes qui vont agir sur cet objet.

→ Par convention les identifiants de classes **commencent par une lettre majuscule.**
→ Un objet est une structure de données avec des fonctions agissant sur ces données. Il regroupe donc données et moyens de traiter ces données. Il contient:

- ➢ **Des attributs**: variables qui caractérisent l'objet,
- ➢ **Des méthodes**: fonctions qui traitent les données, qui servent d'interface.

Le moule qui sert à construire l'objet est appelé classe.
→ Une fois la classe définie, on va pouvoir créer différents exemplaires de l'objet, les exemplaires ainsi créés sont appelés **instances** de la classe.
→ Les attributs et les méthodes d'un objet sont appelés ses **membres**. Dans la plupart des langages, on note les membres d'un objet: objet.membre de façon à savoir à qui appartient le membre considéré.
→ Cette classe Meuble peut donner naissance à d'autres classes qui seront ses descendants. Ces **descendants** vont tous hériter des caractéristiques propres à leur **ancêtre**, c.à.d. ses attributs et ses méthodes. Les descendants ont la possibilité d'avoir en plus leurs propres attributs et méthodes : soit en en définissant de nouveaux, soit en spécialisant les existants.
→ En programmation orientée objet on parle **d'encapsulation**. Ce terme regroupe deux notions:

- ➢ Le fait de réunir sous la même entité les données et les moyens de les gérer.
- ➢ Pouvoir masquer aux yeux d'un programmeur extérieur tous les rouages d'un objet servant à sa gestion interne. L'intérêt étant d'assurer l'intégrité des données.

Ici interviennent deux philosophies:

- ➢ Empêcher l'accès extérieur aux attributs, ils ne pourront être lus ou modifiés qu'au travers de certaines méthodes (langages: Java, C++, PHP,…)
- ➢ Autoriser l'accès aux attributs que ce soit en lecture ou en

écriture (modification) (langages: Python, Ruby,…)

→ La visibilité **publique** définit des attributs et méthodes accessibles depuis toutes les classes et dans tous les programmes.

→ La visibilité **privé** restreint l'accessibilité d'un attribut ou d'une méthode à l'intérieur de la classe où il (elle) est défini(e).

→ La visibilité **protégé** limite la visibilité des membres à l'objet luimême et à ses descendants, quel que soit leur degré de descendance (enfants, petits-enfants, …).

→ Parmi les méthodes publiques, on peut trouver:

- ➢ **Les constructeurs:** servent à construire l'objet, ils réservent la place mémoire pour les données, initialisent éventuellement les attributs, mettent en place ce qui concerne l'héritage.
- ➢ **Les destructeurs:** servent à détruire l'instance (libération de la mémoire allouée, décrémentation de compteur si besoin, …)
- ➢ Les accesseurs : permettent d'accéder aux attributs privés d'un objet. On distingue:
  - o **Les getters:** accesseurs en lecture pour récupérer la valeur d'un attribut, on les nomme get_XXXX(),
  - o **Les setters:** accesseurs en écriture, appelés aussi mutateurs, pour modifier la valeur d'un attribut, on les nomme set_XXXX().

→ Lorsque l'on définit une classe, on peut avoir besoin de préciser que l'attribut considéré est l'attribut de l'objet lui-même (notamment lorsqu'un même identifiant est utilisé pour plusieurs variables). Pour cela on dispose **d'un pointeur interne vers la classe elle-même.** Le mot clé **self permet de désigner l'objet dans lequel on se trouve.**

→ La méthode **__init__(self,…)** sert à initialiser les attributs ("__" correspond à deux caractères "trait du bas" ou underscore) __init__ n'est pas un constructeur (ce serait __new__)

→ Le premier paramètre **self indique que la méthode est**

propre à l'objet.
→ **On peut aussi initialiser les attributs sans les passer en paramètres.**
→ La ligne avec **" " "** est une ligne de commentaire.
→ On peut avoir besoin de **donner une valeur par défaut à un attribut** lorsque celui-ci n'est pas renseigné. C'est possible:
Exemple:

```
class Point(object):
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
```

→ **Les valeurs par défaut sont attribuées en remontant les paramètres du dernier jusqu'au premier.**
→ L'attribut spécial **__dict__** permet de lister la valeur des attributs d'une instance. **Exemple:** r1=Rectangle(5)
r1.__dict__
on obtient: {'side1': 5, 'side2': 4}
→ L'attribute spécial __doc__ permet d'afficher les commentaires d'une classe ou d'une instance. **Exemple:**
r1=Rectangle(5,2)
Rectangle.__doc__ ou r1.__doc__
on obtient: ' classe basique avec juste longueur, largeur '
→ **Rien n'empêche d'ajouter des attributs** spécifiques à une instance, en dehors de la classe. **Les attributs ne sont pas protégés, ils peuvent être modifiés en dehors de la classe.**
→ On peut avoir besoin de définir **une variable commune à toutes les instances** de la classe (pour compter par exemple le nombre d'exemplaires créés). Ce type de variable est appelée **attribut de classe**. Pour cela on écrit **Nomclasse.nomAttribut au lieu de self.nomAttribut.**
→ On peut vouloir écrire une méthode indépendamment de toute instance, pour cela **il faut définir une méthode de classe.** Pour cela, il faut ajouter **le décorateur @classmethod** avant la méthode et le **premier paramètre de la méthode est cls (pour**

classe) au lieu de self.
→ On peut avoir besoin **d'une fonction indépendamment de toute instance et de la classe**, tout en souhaitant l'écrire à l'intérieur de la classe pour qu'elle soit facilement accessible/identifiable, il faut **alors définir une méthode statique:** Pour cela, il faut ajouter **le décorateur @staticmethod** avant la méthode, **l'appel se fait ensuite sous la forme NomClasse.NomMéthode().**
→ Pour **détruire** une instance on utilise **del:** Un destructeur se nomme **__del__()**, il complète ce que fait le destructeur par défaut.

Exemple:

```
class Point(object):
        nb = 0
        def __init__(self, x, y):
                self.x = x
                self.y = y
                Point.nb += 1
        def __del__(self):
                Point.nb -= 1
p1 = Point(-5, 2)
p2 = Point(8,3)
print("On a créé ", Point.nb , "point(s).")
# affichera : on a créé 2 point(s).
del(p1)
print("On a créé ", Point.nb , "point(s).")
# affichera : on a créé 1 point(s).
```

M- **Base 2:**
→ Lorsqu'on écrit des nombres, on utilise le plus souvent l'écriture décimale, c'est-à-dire la base 10: on utilise les chiffres 0–9 pour représenter les puissances de 10. Par exemple lorsqu'on écrit 34765 on veut dire $34765 = 3 \times 104 + 4 \times 103 + 7 \times 102 + 6 \times 101 + 5 \times 100$. Mais il existe d'autres bases

possibles pour écrire un nombre.

Exemple:

En base 10, A = {0, 1, . . . , 9}, $9738 = (9738)_{10} = 9 \times 10^3 + 7 \times 10^2 + 3 \times 10^1 + 8 \times 10^0$

En base 2, A = {0, 1}, $(10011)_2 = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 16 + 0 + 0 + 2 + 1 = (19)_{10}$

$2^0=1$
$2^1=2$
$2^2=4$
$2^3=8$
$2^4=16$
$2^5=32$
$2^6=64$
$2^7=128$
$2^8=256$

N- **Ecriture hexadécimale: base 16:**

→ La numération hexadécimale correspond à la base a = 16, on prend l'alphabet

A = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F }.

Donc A vaut 10, B vaut 11, C vaut 12, D vaut 13, E vaut 14, F vaut 15.

Exemple : $35C_{16} = 3 \times 16^2 + 5 \times 16^1 + 12 \times 16^0 = 768 + 80 + 12 = 860_{10}$

$16^0=1$
$16^1=16$
$16^2=256$