Le cours d'algorithmique

Formation d'ingénieur de l'université des Antilles

Auteur : Vincent Pagé : vincent.page@univ-antilles.fr

Introduction

Dans ce cours, nous verrons les rudiments de l'algorithmique pour vos formations (Matériaux et Systèmes énergétiques). Mon objectif est simple : ne pas focaliser sur les détails, mais vous permettre de faire des choses rapidement. Ce document est extrêmement synthétique. Il ne vous dispense pas d'aller en cours ni d'essayer de programmer vous même.

Pour plus de détails, vous pouvez me poser des questions et/ou chercher un peu sur le net ou dans des cours plus détaillés. Si vous ne voulez pas chercher, vous trouverez quelques liens vers ce type de ressources ici

Les concepts présentés dans ce cours s'appliquent à la plupart les langages que je connais. Lorsque je devrais faire un vrai programme pour vous présenter quelque chose, je le ferais en **python**.

Installez donc python sur votre machine. Les exemples seront fait pour fonctionner en python 3. Je suis sûr qu'avec l'aide d'internet, vous devriez être en mesure d'installer **python 3** sur votre machine...

Le dépot dans lequel vous avez trouvé ce document contient également les exemples que nous avons vu en cours.

Vous pouvez repartir vers le Sommaire ____ ### Concepts d'algorithmique d'aujourd'hui. Ceci est ma vision de la programmation actuelle. Tout le monde ne s'y retrouvera peut être pas, mais il me semble que la plupart des développeurs en entreprise seront d'accord avec le constat suivant : Programmer (ou coder) fait appel à deux grands capacités : - la stratégie - la tactique

La stratégie

C'est, dans le domaine militaire, ce qui vous fait gagner une guerre (*si vous le faites bien et que vous avez de la chance*). C'est la planification vue de loin.

Les généraux, autour d'une table, décident d'envoyer tel bataillon à gauche, pendant que tel autre bataillon partira à droite pour prendre en sandwich l'armée

adverse. Ils peuvent aussi planifier de faire un siège pour affamer l'armée ennemie, et quand ils auront faim, ils attaqueront.

La tactique

C'est, dans le domaine militaire, ce qui vous fait gagner une bataille (*si vous le faites bien et que vous avez de la chance*). C'est la planification vue de très près.

Le sergent, au coeur du champ de bataille, décide d'envoyer 2 hommes a gauche de la petite colline, pendant que ...

Les sergents font souvent de mauvais généraux, les généraux font souvent de mauvais sergents.

Quel rapport avec l'algorithmique et la programmation ?

Prenons un exemple plus ou moins simple : Vous souhaitez faire un programme qui reconnaisse le visage des différents étudiants de la promotion.

Plus précisément, nous devons faire un programme auquel on donne une image, et il doit retrouver le nom de l'étudiant correspondant.

Je vais supposer que nous disposons d'images des visage chaque étudiant de la promotion, associé à son nom.

La partie **Stratégie** c'est la définition des grandes étapes de mon programme. Pour l'exemple, on pourrait penser à quelque chose comme ceci : lorsqu'on me donne un fichier image à reconnaitre, je vais :

1. lire l'image du fichier -> une image *inconnue*
2. lire l'ensemble de mes images connues -> *maBase*
3. pour chaque image de *maBase*

   3.1 Je mesure la distance de mon image *inconnue* a l'image connue -> dist

   3.2 Je retiens l'image connue ayant la plus distance *dist* -> *imageLaPlus-Proche*
4. J'annonce avoir reconnu l'étudiant dont le nom corresspond à *imageLaPlus-Proche*

Vous ne savez pas lire une image ? Ce n'est pas très grave (internet ou votre sergent le sait)

Vous ne savez pas définir une distance entre deux images ? Raffinez votre stratégie. Par exemple, une solution classique (assez peu efficace, il faut reconnaître) consiste à calculer la somme des carrés des différences pixel à pixel entre les 2 images.

Tout ce travail n'est pas forcément fait par un informaticien.

La partie **tactique** c'est, pour une partie précise de mon programme, la réalisation d'un algorithme dans un langage donné.

Pour l'exemple, Si je dois faire cette somme des carrés ... je pourrais faire quelque chose comme ceci :

```
distance = 0;
for x in range(len(image1.width) :
  for y in range()(image1.height)) :
    dist += (image1[x][y] - image2[x][y]) **2
```

Vous ne comprenez pas ce code ? c'est normal. Vous devriez en comprendre l'essentiel d'ici quelques cours.

La **tactique**, c'est de la programmation les mains dans le cambouis (*ou dans la boue si on conserve l'image du sergent*)

Les cours d'algorithmique focalisent souvent sur la tactique, alors que les langages actuels permettent d'en faire abstraction assez fréquemment. Par exemple, si je dois trier un tableau par ordre croissant, nous avons, dans tous les langages évolués, des fonctions pour le faire sans faire la moindre boucle.

La **stratégie**, est utile pour programmer mais aussi pour la coordination de n'importe quel projet que vous aurez à réaliser : vous aller choisir d'enchainer des actions, certaines requérant des résultats obtenus par des actions précédentes...

Ce cours a pour objectif de vous apprendre un peu de **tactique** (les variables, les boucles, les fonctions...) et beaucoup de **stratégie**.

Voyons donc le minimum de tactique à savoir pour commencer ### Les variables Comprenons une chose tout d'abord : un programme informatique ne fait qu'une chose : il manipule des variables. Toutes les informations que doit gérer votre programme doivent donc se retrouver dans des variables. Vous en connaissez vraisemblablement quelques types simples de variables : - les entiers (int) - les nombres à virgules (float) - les vrais ou faux (booleen) - les chaînes de caractères (string)

Dans les variables, je stocke des valeurs.

```
a=5
b=7
print(a+b)
```

j'ai créé une variable a, lui ai donné la valeur 5, puis crée une variable b, lui ai donné la valeur 7, puis affiché le résultat de la somme des valeurs des deux variables. ### Les tests conditionnel (*if*) Une grande partie de l'algorithmique consiste a dire ce que l'on fait dans tel ou tel cas. Au coeur de tout ceci se trouve le test conditionnel. Ici, on fait un programme qui compare la valeur de a avec celle de b. Si a est plus petit, on écrit qu'il est plus petit, sinon, on écrit qu'il est plus grand. Enfin, notre programme continuera à écrire des inepties. Voici la syntaxe en python :

```
if a < b :
    print ("a est plus petit")
    print ("mais il est vaillant")
else :
    print ("a est plus grand ou égal")

print ("la taille importe peu")
```

Notez le décalage horizontal qui signale ce qui est dans le if, et ce qui ne l'est pas. "Ce qui est dans le if" est appelé **un bloc d'instructions**.

En python, il est décalé (on dit **indenté**) et il y a deux points avant...

Eventuellement, il pourrait être intéressant d'avoir 3 cas : - $a < b$ - $a > b$ - $a = b$

La solution consisterait à imbriquer des *if* ou a utiliser *elif* (cherchez sur le net, on le verra en cours mais je ne vais pas surcharger ce support)

Exercices

Vous **devez** être en mesure de faire les exercices suivants :

1. faites un programme qui demande à l'utilisateur un entier a et affiche un message adapté si l'on est dans l'un des deux cas suivants

- a < 10
- a >= 10

2. faites un programme qui demande à l'utilisateur son nom *nom* et affiche :

- "Bienvenue mr page" si le nom est "page"
- "Sortez d'ici" dans les autres cas

3. faites un programme qui demande à l'utilisateur un entier a et affiche un message adapté si l'on est dans l'un des trois cas suivants
    - a = 10
    - a < 10
    - a > 10
4. faites un programme qui demande à l'utilisateur un entier a et affiche un message adapté si l'on est dans l'un des 4 cas suivants
    - a <= -3
    - -3 < a < 0
    - 0 < a < 2

- 2 <= a ### Les boucles *Tant que* (while) Imaginons que je veuille écrire “bonjour à tous” 10 fois. Nous pouvons le faire en répétant 10 fois la ligne suivante :

  ```
  print ("bonjour à tous")
  ```

  Mais c’est laid et peu efficace (si je veux changer le message en “au revoir à tous”, il faudra que je fasse 10 modifications)

L’idée est de répéter une série d’instructions plusieurs fois. On parle d’une boucle (ici, une boucle Tant que ou **while**)

une premiere boucle infinie

Voici ce que l’on pourrait faire :

```
a = 0
while (a == 0)
  print ("bonjour à tous")
  print ("Appuyez sur Ctrl C pour quitter")
```

Le déroulement est le suivant : 1. on initialise *a* à 0 2. on arrive sur la ligne du *while*. On teste si *a* est égal à 0. Si oui, on execute le code du bloc en dessous. Dans notre cas, *a* vaut bien 0. 3. on affiche “bonjour a tous” 4. on affiche “Appuyez” 5. le bloc d’instruction est terminé. On remonte à la ligne du while (comme en 2.) et on recommence (on boucle)

Dit autrement, on effectue des tous de boucle. Avant chaque nouveau tour, on vérifie si la condition est vraie (**True**) ou fausse (**False**)

Dans le cas de notre programme, vu que *a* ne change pas de valeur dans la boucle, nous allons boucler indéfiniment. Pour quitter une boucle infinie, suivez les instructions affichées par notre programme (Ctrl + c)

Une boucle qui compte les tours.

En modifiant un peu notre programme, on va se servir de *a* pour compter le nombre de tours que l’on fait...

```
a = 0
while (a < 3):
  print ("bonjour à tous")
  print ("j'ai fait", a, "tours")
  a = a+1

print ("fin de la boucle")
```

Ici, à chaque tour de boucle, *a*, qui avait commencé à 0, est augmenté de 1. au bout de 3 tours, la condition *a<3* ne sera plus respectée et nous sortirons de la boucle pour afficher “fin de la boucle”

Nous verrons plus tard (ou vous chercherez) un autre célèbre type de boucle, la boucle **for** qui marche bien aussi, mais je n'en n'ai pas besoin pour le moment.

Exercices

Vous **devez** être en mesure de faire les exercices suivants : 1. faites un programme qui affiche les nombres entiers de 0 a 33

2. faites un programme qui affiche les nombres de 3 a 33
3. faites un programme qui calcule la somme des nombres de 0 à 33
4. faites un programme qui calcule la somme des inverses des nombres de 0 à 33
5. faites un programme qui calcule le produit des nombres de 1 à 33
6. faites un programme qui calcule le produit du carré des nombres de 1 à 33

Les fonctions.

Le code que j'ai présenté juste avant est le **programme principal**. C'est ce que fait mon programme.

Un vrai bon programme découpe le code en petites actions que le programme principal organise. Ces petites actions seront des fonctions.

Programmer en utilisant des fonctions va nous permettre de - simplifier notre programme principal - limiter les risques d'erreur - simplifier l'évolution de mon programme.

Création d'une fonction

Commençons par un exemple simple : un programme qui affiche 3 fois "bonjour" et le nombre de tours faits.

```
i = 0
while (i < 3):
  print ("bonjour à tous")
  i = i+1
```

Je vais créer une fonction qui devra effectuer cette action. ma fonction s'appellera *affiche3FoisBonjour*

```
def affiche3FoisBonjour():
    i = 0
    while (i < 3):
      print ("bonjour à tous")
      i = i+1
```

Cette fonction est comparable à un ouvrier spécialisé que l'on appelle chaque fois que l'on veut qu'il travaille.

Pour l'appeler, mon **programme principal** va utiliser son nom, sans oublier les parenthèses, comme suit :

```
affiche3FoisBonjour()
```

Je peux appeler ma fonctions plusieurs fois au besoin :

```
def affiche3FoisBonjour():
    i = 0
    while (i < 3):
      print ("bonjour à tous")
      i = i+1


affiche3FoisBonjour()
affiche3FoisBonjour()
```

le code ci dessus définit la fonction puis le **programme principal** l'appelle 2 fois.

Pour préparer la section suivante, je vais ré-écrire le code précédent en entrant le nombre de fois ou j'écris le message dans une variable :

```
def affiche3FoisBonjour():
    n=3
    i = 0
    while (i < n):
      print ("bonjour à tous")
      i = i+1


affiche3FoisBonjour()
affiche3FoisBonjour()
```

Le programme fait exactement la même chose qu'auparavant.

Passage d'un paramètre à une fonction

Le gros intérêt des fonctions est la possibilité de les paramétrer.

Si mon programme doit afficher un certain nombre de fois "bonjour a tous", il faudrait que le **programme principal** puisse dire à la fonction quelle valeur on veut donner à sa variable n

Le problème est que les fonctions travaillent avec des variables que elles seules connaissent (on parle de **variables locales**). Donc le programme principal ne

connait pas la variable n de la fonction. Il faut donc établir une communication entre le programme principal et la fonction.

On dit qu'on **passe un argument à la fonction**. Ceci se fait comme suit :

```
def afficheNFoisBonjour(n):
    i = 0
    while (i < n):
      print ("bonjour à tous")
      i = i+1


afficheNFoisBonjour(5)
```

Mon programme principal dit *5* à la fonction. La fonction récupère ce *5* et le met dans sa variable n (entre les parenthèses) et peut faire ce qu'on lui demande.

Notez que ma fonction s'appelle maintenant *afficheNFoisBonjour* vu qu'elle est paramétrable. Changez la valeur (ici *5*) passée a la fonction et ré-executez le code. Observez que ma fonction sait maintenant faire beaucoup plus de choses qu'avant.

Passage de plusieurs paramètres à une fonction

Je vais encore modifier ma fonction pour qu'elle affiche plusieurs fois un message variable quelconque.

Vu de la fonction, le message à afficher sera également une variable *message*, que le programme principal lui communiquera.

```
def afficheNFoisTruc(n,message):
    i = 0
    while (i < n):
      print (message)
      i = i+1


afficheNFoisTruc(5, "bonjour")
afficheNFoisTruc(3, "au revoir")
```

Lors de l'appel de la fonction par le programme principal, celui ci donne maintenant 2 informations à la fonctions (2 arguments) : le nombre et le message. C'est l'ordre qui détermine dans quelle variable finit la valeur passée par le programme principal.

Ainsi, dans le programme qui précède : - au premier appel, le 5 atterrit dans la variable n de la fonction et "bonjour" dans la variable *message* de la fonction. - au second appel, le 3 atterrit dans la variable n de la fonction et "au revoir" dans la variable *message* de la fonction.

Au final, mon programme affiche donc 5 fois “bonjour” puis 3 fois “au revoir”

Valeur de retour d’une fonction

Les fonctions que nous avons vues font des choses a partir des informations que leur donne le programme principal.

Comment faire si maintenant ces fonctions doivent communiquer des informations au programme principal ?

Par exemple, si je dois faire une fonction qui fait la somme de deux nombres :

```
def somme(a,b):
    resu = a +b

somme (3,5)
```

ma fonction calcule bien la somme de a et b et met le résultat dans resu. Le programme principal fournit les valeurs de a et de b lors de l’appel, mais il ne peut pas utiliser le résultat calculé car **resu** est locale à la fonction.

Il faut que la fonction *réponde* au programme. On dit qu’elle **retourne une valeur** au programme. Cela se fait avec le mot clef **return**.

```
def somme(a,b):
    resu = a +b
    return resu

resultat = somme (3,5)
print(resultat)
```

Dans ce qui précède la fonction calcule *resu* le renvoie au programme principal. Le programme principal récupère ce résultat et le place dans la variable *resultat*. On peut ensuite l’afficher.

Tout se passe comme si l’appel de la fonction était remplacé par la valeur que la fonction renvoie.

Exercices

Vous devez être capables de faire ce qui suit :

- faire une fonction qui calcule le produit des nombres entiers de 1 à n et l’appeler pour que le programme principal affiche la somme. n sera fixé par l’utilisateur du programme.
- reprendre **tous** les exercices des sections précédentes pour les transformer en fonctions que le programme principal utilise. ## Cours 2 : Tableaux

intérêt

Avec les types de variables que nous avons vus dans le cours précédent, nous pouvons faire beaucoup de choses, mais cela devient vite pénible.

Imaginons que je doive faire un programme qui gère vos notes, je doiS stocker une note (un float) par étudiant. Je pourrait par exemple choisir un ordre pour rentrer les notes et stocker la note de chaque étudiant dans une variable. Disons que ma classe ne contient que 3 étudiants.

```
e1 = 12
e2 = 9.5
e3 = 14
```

Pour calculer la moyenne de ma promotion, je pourrais ajouter ceci au code qui précède :

```
moyenne = (e1+e2+e3) / 3
print (moyenne)
```

Si finalement, je dois ajouter un 4eme étudiant arrivé en retard, mon code est transformé à de multiples endroits pour devenir :

```
e1 = 12
e2 = 9.5
e3 = 14
e4 = 2 # oui, il est mauvais, en plus

moyenne = (e1+e2+e3+e4) / 4
print (moyenne)
```

A chaque ajout d'étudiant, il faudra que je pense à faire toutes les modifications. A terme, je suis sûr de faire une erreur...

De plus, dans le cas de votre promotion, il me faudrait une vingtaine de variables, ce qui devient lourd et pénible. Il est temp d'introduire les tableaux. En python, on les appelle des *listes*, mais c'est sans importance pour nous.

Voici ce que je voudrais : une seule variable contenant ces informations, rangées dans des cases séparées.

notes
12
9.5
14
2

Ici, *notes* est le nom de mon tableau. L'intérêt est simple : si j'ajoute un étudiant, j'ajoute juste une case à mon tableau. **Mes notes ne sont contenues que**

dans une seule variable : *notes*

Je peux créer un tableau de ce type comme ceci

```
notes = [12, 9.5, 14, 2]
print(notes)
```

Pour accéder à une case, je vais utiliser son numéro (on parle d'*indice* de la case). Les indices commencent à 0. Une bonne représentation de mon tableau serait la suivante :

indice	valeur
0	12
1	9.5
2	14
3	2

Si je veux accéder à la case numéro 2, j'utiliserais l'écriture suivante : notes[2]

Le code suivant : - affiche la valeur de la case 2 (qui vaut 14) - modifie la valeur de la case 3 pour y mettre la valeur 11 - affiche tout le tableau

```
print (notes [2])
notes[3] = 11
print(notes)
```

Manipulations de base sur les tableaux

Création d'un tableau

Pour créer un tableau, on peut le créer déja rempli, comme nous l'avons fait. Nous aurions pu également créer un tableau vide et le remplir quand nous voulons (ce qui permettrait d'ajouter des étudiants à n'importe quel moment)

Ce qui suit crée un tableau vide, et le remplit avec des chaines de caractères contenant les noms de chaque étudiant de ma promo.

```
noms = []

print (noms)

noms.append("moutoussamy")
noms.append("destouches")
print (noms)

noms.append("julan")
print (noms)
```

```
noms.append("naejus")
print (noms)
```

longueur d'un tableau

il peut être utile de connaitre le nombre de cases d'un tableau nommé *tab* : on l'obtient avec la fonction *len*

```
nbEtudiants = len(noms)
print (nbEtudiants)
```

parcours de tableaux

Si je veux afficher le contenu de chaque case du tableau de notes, je peux utiliser print(notes). Ici, je vais le faire d'une autre manière, que je réutiliserais de plusieurs façons différentes utilisant toutes la notion de parcours de tableau. Cela me permettra de faire des choses plus compliquées plus tard (comme calculer la moyenne ou trouver le nom du major de promo).

Si je veux afficher le tableau, ce que je veux faire est en fait afficher successivement le contenu de chaque case.

Je vais donc visiter chaque case du tableau afficher le contenu de la case en cours.

Nous allons voir 3 façon de le faire, chacune ayant ses intérêts (surtout les 2 dernières en fait)

parcours d'un tableau avec une boucle while.

je peux le faire comme suit avec la boucle *while* vue dans le cours précédent :

```
notes = [12, 9.5, 14, 2]

i = 0
while i<4 :
    print(notes[i])
    i = i+1
```

Mais ceci ne marche que pour un tableau de 4 cases. Je peux améliorer ceci facilement en modifiant légèrement mon code en prenant en compte la taille du tableau.

```
notes = [12, 9.5, 14, 2]

i = 0
while i<len(notes) :
  print(notes[i])
  i = i+1
```

C'est fonctionnel mais peu sympathique à écrire. Voyons une version plus pratique.

parcours d'un tableau avec une boucle for.

```
notes = [12, 9.5, 14, 2]

for e in notes :
  print (e)
```

Dans ce type de boucle, à chaque tour de boucle, la variable *e* va prendre la valeur du contenu de la case visitée. le *for* se débrouille tout seul pour se promener de case en case.

Le code précédent se lit quasiment comme en francais : pour chaque *e* dans le tableau *notes*, j'affiche *e*

Vous choisissez le nom que vous donnez à la variable qui visite les cases. Le nom du tableau (après *in*) est celui que vous avez donné à votre variable contenant le tableau. Je pourrais tout aussi bien écrire :

```
notes = [12, 9.5, 14, 2]

for biten in notes :
  print (biten)
```

C'est la version la plus simple pour parcourir tout tableau.

parcours avec une boucle for et l'indice de la case

il peut arriver que j'ai besoin de me déplacer dans mon tableau en utilisant le numéro des cases du tableau. C'est ce que nous allons essayer de faire ici...

Commençons par ce code. Vous devriez vite comprendre qu'il affiche les chiffres de 0 à 3.

```
indices = [0,1,2,3]

for i in indices :
  print (i)
```

Je peux me servir de ce *i* variable pour afficher le contenu de la case numéro *i* d'un tableau, comme ceci :

```
for i in indices :
  print (notes[i])
```

Le problème est que je dois définir manuellement le tableau indice. Ce qui est pénible, si mon tableau a de nombreuses cases. Pour générer un tableau allant de 0 à n-1, nous disposons de *range(n)*.

```
indices  = range(4)
for i in indices :
  print (notes[i])
```

ou encore

```
for i in range(4) :
  print (notes[i])
```

Mais ceci ne fonctionne que pour des tableaux de 4 cases. Il suffit d'intégrer la longueur du tableau à mon code et c'est réglé :

```
for i in range(len(notes)) :
  print (notes[i])
```

Quelques exemples

Si vous avez compris ces parcours, au lieu d'afficher simplement le contenu de chaque case, nous allons pouvoir faire des choses plus complexes.

Somme des éléments d'un tableaux

Pour faire la somme des éléments d'un tableau, il me suffit d'avoir une variable *somme* qui vaudra 0 au départ, et que je vais augmenter au cours de mon parcours de la valeur de chaque case visitée.

N'importe quel parcours parmi les 3 précédents fonctionne, je vais prendre le plus simple.

```
notes = [12, 9.5, 14, 2]

somme = 0
for n in notes :
    somme = somme + n

print ("somme :", somme)
```

A moindre frais, je peux aussi calculer la moyenne en ajoutant la ligne qui suit :

```
print ("moyenne :", somme/len(notes))
```

recherche du maximum d'un tableaux

Si je cherche le maximum de mon tableau, il suffit que je dispose d'une variable *maxi* qui vaut, disons 0. Je parcours mon tableau et chaque fois que je vois quelque chose de plus grand que ce que j'ai déja vu, mon *maxi* est mis à jour.

Quelque chose comme ceci.

```
notes = [12, 9.5, 14, 2]

maxi = 0
for n in notes :
    if n > maxi:
      maxi =  n

print ("somme :", somme)
```

Avec un peu d'expérience, je peux me méfier de la ligne qui dit *maxi = 0*. En effet si mon tableau ne contient que des éléments négatifs, aucune case ne va déclencher mon *if*. Par conséquent, a la fin de la boucle, le maximum de mon tableau sera resté à zéro, alors que tous les éléments sont négatifs... De fait, il vaut mieux initialiser mon maxi à la valeur de la premiere case du tableau, comme suit :

maxi = notes[0] for n in notes : if n > maxi: maxi = n

print ("maxi :", maxi)

parcours de 2 tableaux conjoints

Bon. Imaginons que je veuille maintenant gérer aussi les noms de mes étudiants. Je pourrais stocker le nom des étudiants dans un tableau de chaînes de caractères.

```
noms = ["moutoussamy","destouches","julan","naejus"]
```

L'idée est que l'étudiant dont le nom est dans la case numéro 2 du tableau *nom* a eu la note contenue dans la case numéro 2 du tableau *notes*

Si je souhaite afficher les noms et les notes de ma promotion : je vais me déplacer de case en case en utilisant un indice variant de 0 à 3. Pour chaque indice, j'affiche la case correspondante dans le tableau des noms et la case correspondante dans le tableau de notes.

```
noms = ["moutoussamy","destouches","julan","naejus"]
for i in range(len(notes)):
    print(noms[i], notes[i])
```

Cela commence a ressembler a quelque chose ! Voyons si l'on peut compliquer un tout petit peu.

affichage du nom du major

Pour afficher le nom du major de promo, c'est relativement simple : je vais chercher le numéro de la case contenant le maximum du tableau de notes. Je peux alors afficher le nom situé dans la case correspondante dans le tableau de noms.

Pour trouver la position du maximum : on veut retenir la valeur du maximum, mais aussi sa position. On aura donc ces variables à mémoriser.

Lors du parcours du tableau, chaque fois que je mets a jour mon maximum, je met également à jour sa position.

```
maxi = notes[0]
imaxi = 0

for i in range(len(notes)):
    if notes[i]> maxi:
        maxi = notes[i]
        imaxi = i

print ("major", noms[imaxi])
```

Nettoyage et mise en fonctions.

Pour avoir du code propre il est toujours recommandé de coder en utilisant des fonctions.

Chacune des petites actions que j'ai faites va donc être déplacée dans une fonction spécifique. A dire vrai, j'aurais tendance à le faire au fur et à mesure. Ici, et c'est la dernière fois de l'année, je le fais à la fin du chapitre pour ne pas tout mélanger.

En se souvenant de ce qui a été fait dans le cours précédent sur les fonctions, vous devriez pouvoir comprendre ce qui suit, qui reprend la totalité de nos travaux, mais proprement.

```
def afficher(tab):
    for e in tab :
        print(e)

def afficherPromo(noms, notes):
    for i in range(len(noms)) :
        print(noms[i], notes[i])

def calculerSomme(tab):
    somme = 0
    for e in tab :
        somme = somme + e

    return somme

def maximum(tab):
    maxi = tab[0]
    for e in tab :
```

```
        if e > maxi:
            maxi = e

    return maxi

def imaximum(tab):
    maxi = tab[0]
    imaxi = 0

    for i in range(len(tab)) :
        if tab[i] > maxi:
            maxi = tab[i]
            imaxi = i

    return imaxi

notes = [12, 9.5, 14, 2]
noms = ["moutoussamy","destouches","julan","naejus"]

afficher(noms)
afficher(notes)

afficherPromo(noms, notes)

somme = calculerSomme (notes)
print ("somme :", somme)

moyenne = somme / len(notes)
print ("moyenne :", moyenne)

maxi = maximum(notes)
print ("maxi :", maxi)

iMajor = imaximum(notes)

print ("major :", noms[iMajor])
```

les fichiers

Les sources de tout ce que nous avons fait dans ce cours sont dans le répertoire Sources. On y trouvera en particulier : - L'intro sur les tableaux - Les parcours de tableaux - Les exemples de calcul - La version finale ### Cours 3 : Réalisation d'un vrai projet

Projet : Présentation et Méthodologie

Présentation du projet

Ici, on s'intéresse au **Jeu des Allumettes**. - C'est un jeu à deux joueurs qui jouent chacun son tour. - au départ les allumettes sont disposées comme suit :

numéro de ligne	allumettes
0	I
1	III
2	IIIII
3	IIIIIII

- A chaque tour, un joueur choisit une ligne et un nombre d'allumettes à retirer sur cette ligne (entre 1 et 3)
- le joueur qui retire la dernière allumette a perdu.

Il s'agit pour nous de faire un programme qui permette à deux joueurs humains de jouer sur l'ordinateur.

Méthodologie de développement du projet

Dans un tel projet, peut être trop complexe pour vous, il va néanmoins falloir commencer pour avancer. Le plus simple est de définir les grandes lignes du projet (notre stratégie).

Ensuite, nous transformerons ces grandes lignes en un programme principal et coderons toutes les fonctions utiles.

Grandes lignes

Ici, on se contente de décrire ce qui se passe dans une partie... Nous traduirons ces grandes lignes en un programme principal plus tard.

```
1. Joueur départ : joueur 1
2. initialiser plateau

3. tant qu'il reste au moins une allumette
  - demander ligne et combien d'allumettes
  - retirer allumettes
  - afficher plateau
  - changer Joueur

4. afficherVainqueur
```

Transformation en programme principal.

Pour transformer proprement ces grandes lignes en un programme, je dois savoir quelles variables sont passées à toutes mes petites parties de programme.

La variable la plus importante est sans aucun doute l'état du plateau qui stocke quelles allumettes sont sur quelle ligne. Une rapide reflexion montre que la seule information importante est le nombre d'allumettes présent sur chaque ligne. Je peux donc coder ceci avec un tableau de 4 entiers. Mon initialisation du plateau peut donc s'écrire comme suit :

```
plateau = [1,3,5,7]
```

Ensuite, je vais pour chaque action de mon plan, supposer l'existence d'une fonction qui fait ce qu'on lui demande et répond ce dont j'ai besoin. Le code complet donnerait ceci :

```
numJoueur = 1
plateau = [1,3,5,7]

while ( compteAllumettes(plateau) > 0 ) :
  numLigne, nbAllu = choisirStrategie(plateau)
  retirerAllumettes(plateau, numLigne, nbAllu)
  afficherPlateau(plateau)
  numJoueur = changerJoueur(numJoueur)

print ("le joueur ", numJoueur, "a gagne")
```

Ce code met en évidence des fonctions : - *compteAllumettes* qui renvoie le nombre d'allumettes restantes - *choisirStrategie* qui renvoie le nombre d'allumettes que l'utilisateur veut retirer et le numéro de ligne sur laquelle les retirer. - *retirerAllumettes* qui retire effectivement ces allumettes. - *afficherPlateau* qui affiche l'état du plateau actuel. - *changerJoueur* qui fait la bascule entre joueur 1 et joueur 2.

Un grand intérêt doit être porté aux variables qui sont passées à chaque fonction et à l'enchainement des appels. Par exemple, *numLigne* est le numéro de ligne sur laquel le joueur veut retirer ses allumettes. Il a été choisi dans la fonction *choisirStrategie* et sera réutilisé par *retirerAllumettes*.

Quand chaque fonction aura été codée, tout le projet fonctionnera.

par exemple, coder (même salement) la fonction qui compte les allumettes peut se faire comme suit :

```
def compteAllumettes(tab) :
    somme = tab[0]+tab[1]+tab[2]+tab[3]

    return somme
```

Ci nous faisons ce travail préalable, nous pouvons maintenant lancer une équipe de développeurs sur chaque fonction séparément, ils travailleront en autonomie.

Dans la pratique, cette approche est trop stricte. Sur cet exemple, je suis capable de définir une stratégie générale sans trop de difficulté, mais pour des projets très complexes, c'est parfois délicat.

Ce que nous avons fait a surtout une valeur pédagogique : Comprendre cette notion d'enchainement d'actions et de passage de variables. Oublions donc un peu ce que nous venons de faire pour repartir de l'étape des grandes lignes.

Méthodologie de développement réel

Un développeur un peu chevronné fait souvent le travail d'élaboration des grandes lignes dans sa tête. Si vous n'y arrivez pas, posez le sur un papier (ou dans un fichier).

Il définit ensuite la ou les variables dont il a le plus besoin. Pour nous, c'est clairement *plateau*, que j'ai vu comment définir en python :

```
plateau = [1,3,5,7]
```

Puis il va élaborer directement une fonction parmi toutes celles à faire et la tester. Puis il s'intéressera a une autre fonction et à son chainage avec la première. Mais dans quel ordre ? La règle standard est la suivante : - les plus faciles d'abord - les plus utiles pour les tests d'abord.

Notez qu'on ne finalisera peut être pas tout de suite ces fonctions, mais on les mettra dans un état permettant à notre projet d'avancer.

Ici, par exemple, la fonction par laquelle je commencerais serait clairement *afficherPlateau*. Celle ci est facilement testable et quand j'aurais fait les autres fonctions (retirer des allumettes, les compter,. . .), l'affichage pourra me permettre de comparer ce que je vois avec ce que calculent ces fonctions.

A votre niveau, une première version d'affichage pourrait être la suivante, avec son test :

```
def afficherPlateau(tab) :
    print (tab)

plateau = [1,3,5,7]
afficherPlateau(plateau)
```

Cette version affiche tout le tableau. C'est moche. . . A moindre coût, on peut faire la meme chose comme ceci, ce qui est toujours moche, mais j'ai fait une boucle et chaque tour de boucle (une **itération** de la boucle) sert à traiter une case du tableau, donc une ligne de mon plateau.

```
def afficherPlateau(tab) :
    for i in range(len(tab)) :
```

```
        print (tab[i], end=" ")

    print("")

plateau = [1,3,5,7]
afficherPlateau(plateau)
```

Je peux améliorer mon affichage en créant une petite fonction responsable de l'affichage d'une ligne. Je lui donne un nombre entier et elle affiche autant de **I** que ce nombre avant de sauter à la ligne. Si je la code, je la teste dans la foulée.

```
def afficherLigne(n) :
    for i in range(n):
        print("I", end = "")
    print("")

afficherLigne(4)
afficherLigne(2)
afficherLigne(-1)
```

Je peux chainer les deux fonctions : mon programme principal appelle la fonction *afficherPlateau* qui, pour chaque ligne, appelle la fonction *afficherLigne*. Et je teste le tout.

```
def afficherLigne(n) :
    for i in range(n):
        print("I", end = "")
    print("")

def afficherPlateau(tab) :
    for i in range(len(tab)) :
        afficherLigne (tab[i])

plateau = [1,3,5,7]
afficherPlateau(plateau)
```

L'étape suivante consisterait : 1. à être capable de choisir un nombre représentant un numéro de ligne et un nombre d'allumettes... 2. retirer effectivement ces allumettes. 3. toutes les autres actions définies dans les grandes lignes 4. le bouclage sur un nouveau tour

Nous procèderons ainsi de proche en proche avec à chaque étape : - un programme qui fonctionne - un programme qui a été testé - un programme qui s'approche de plus en plus de l'objectif

En cours, nous sommes allés un peu plus loin que ceci. Le résultat obtenu en cours est ici. Cela inclue la fonction qui demande au joueur quelle ligne / combien d'allumettes il veut retirer et la fonction qui retire effectivement ces allumettes du plateau.

Pour comprendre ces parties, ce n'est pas compliqué, il vous manque simplement une information : pour lire un entier que l'utilisateur entre avec le clavier :

1. On utilise la fonction *input()* qui renvoie ce que l'utilisateur a tapé au clavier sous forme d'une chaine de caractères (*string*)
2. On transforme cette chaine de charactères en entier avec la fonction *int*

ce qui donne ceci pour afficher le carré d'un entier choisi par l'utilisateur :

```
print ("entrez un entier")
chaine = input()
monEntier = int(chaine)
print (monEntier**2)
```

ou en version courte

```
print ("entrez un entier")
monEntier = int(input())
print (monEntier**2)
```

Voila...

les fichiers

Les sources de tout ce que nous avons fait dans ce cours sont dans le répertoire Sources. On y trouvera en particulier : - Les grandes lignes - le programme principal final - le développement pas à pas

Exercices à faire

Vous **devez** être en mesure de faire les choses suivantes : - Essayer de ré-écrire les grandes lignes sans regarder le modèle. - refaire la traduction en un programme principal en comprenant pourquoi tout s'enchaine comme cela.

Mais vous ne pourrez pas le tester... vu que les fonctions manqueront.

Ensuite, vous **devez** être en mesure de : - écrire tout seul la fonction *afficherLigne* et la tester - écrire tout seul la fonction *afficherPlateau* et la tester - écrire tout seul les fonctions contenues dans ce fichier et les tester

Vous **devriez** essayer de faire les choses suivantes : - ajouter toutes les fonctions utiles dans une version simple (par exemple, on ne vérifie pas si le numéro de ligne et le nombre d'allumettes choisis par le joueur sont valide) - ajouter la boucle sur les tours de jeu. - modifier la fonction d'affichage du plateau pour qu'elle affiche le numéro de ligne avant les allumettes, comme suit :

```
0: I
1: III
2: IIIII
```

```
3: IIIIIII
```

- modifier la fonction *choisirStrategie* pour qu'elle affiche un message incluant le numéro du joueur, tel que :

  ```
  Joueur 2 :
  Entrez un numéro de Ligne
  ```

  il faudra sans doute modifier la fonction *afficherLigne* et peut être lui passer un paramètre supplémentaire.

Vous **pourriez** : - Essayer de modifier le programme pour que les joueurs manipulent des numéros de ligne allant de 1 à 4 (et pas de 0 à 3). Notre programme, lui, continuera à stocker le nombre d'allumettes dans un tableau indexé de 0 à 3. - Essayer de mettre en place la vérification du numéro de ligne et du nombre d'allumettes choisi par le joueur. Si ses choix sont non valides, on lui redemande... ### Cours 4 : Structures de données élaborées

Il s'agit ici de disposer de structures un peu plus compliquées que des variables isolées (vues au cours 1) ou des tableaux 1D (vus au cours2

un tableau 1D, c'est ca :

indices
cases

Tableaux 2D

Il peut être intéressant d'avoir sous la main un **tableau 2D** tel que celui décrit ci dessous. J'appellerais ce tableau *structure1*

ligne 0
ligne 1
ligne 2

Dans un tableau comme celui ci, on voudrait pouvoir acceder à la variable contenue dans la case sur la ième ligne et la jème colonne avec du code comme celui ci

```
structure1[0][1] = 5;
```

qui mettrait la valeur 5 dans la case située ligne 0, colonne 1

Toute l'astuce en algorithmique est de recycler ce qu'on a déjà fait.

Il est facile de créer un tableau 2D a partir d'un tableau 1D. Le tableau 2D est un tableau dont chaque case contient : un tableau...

Voici le schéma correspondant :

tab2D
ligne0
ligne1
ligne2

Et chaque ligne est une variable contenant un tableau. par exemple, *ligne0* serait le tableau suivant.

case [0,0]

case [0,1]

case [0,2]

...

case [0,5]

Le tout nous donne cette structure pour mon tableau 2D : c'est ma *structure2*

case [0,0]

case [0,1]

case [0,2]

...

case [0,5]

case [1,0]

case [1,1]

case [1,2]

...

case [1,5]

case [2,0]

case [2,1]

case [2,2]

...

case [2,5]

Elle est équivalente dans son utilisation au modèle de la *structure1*. Relisez ca calmement si cela vous pose problème.

Implémentation en python.

Maintenant, implémentons ça en python. Nous allons créer un tableau contenant les noms et les notes de la promo.

```
promo = [["Moutoussamy", 13],["Madasaib",17]]
```

On aurait pu le créer comme cela aussi :

```
promo = []
promo.append(["Moutoussamy", 13])
promo.append(["Madasaib",17])
```

Pour l'afficher, pour le moment, je vais faire :

```
print (promo)
```

Je vous rappelle que mon tableau a la forme suivante :

"Moutoussamy"

13

"Madasaib"

17

- *promo* est le **tableau externe**.
- *promo[0]* est le **tableau interne** de la première ligne.
- *promo[0][0]* est donc la première case de ce tableau interne et contient *"Moutoussamy"*
- *promo[0][1]* est la seconde case de ce tableau interne et contient *13*

Je peux aussi penser directement en termes lignes colonnes en disant :

- *promo[i][j]* est la valeur de la case située ligne i, colonne j

Pour bien comprendre ces notions, vous pouvez essayer le code suivant (n'hésitez pas à modifier les chiffres entre crochets) :

```
print(promo)
print(promo[0])
print(promo[0][1])
```

Quelques fonctions qui travaillent sur un seul indice

Afficher une ligne

Je veux un programme qui affiche une ligne de mon tableau *promo*

1. je fais une fonction a qui on passe le tableau interne (une ligne)

2. j'appelle cette fonction en lui donnant une case du tableau externe

```
def afficherEtudiant(ligne) :
    for case in ligne :
        print (case, end = " ")

    print("")
```

je peux appeler cette fonction comme suit dans mon programme principal :

```
afficherEtudiant(promo[0])
afficherEtudiant(promo[1])
```

J'aurais pu faire ma fonction comme ceci, en changeant la forme du *for*

```
def afficherEtudiantV2(tabE) :

    for j in range(len(tabE)) :
        print (tabE[j], end = " ")

    print("")

afficherEtudiantV2(promo[0])
afficherEtudiantV2(promo[1])
```

Afficher une colonne

Je veux un programme qui affiche les noms contenus dans mon tableau *promo*. Le problème est que mon tableau promo ne contient pas de structure contenant la colonne.

Je peux m'en sortir comme ceci sur chaque ligne *i*, j'affiche juste la case tab[i][0]

```
def afficherNoms(tab) :

    for i in range(len(tab)) :
        print (tab[i][0])

afficherNoms(promo)
```

Parcours de tableaux 2D

Avec ce que nous avons fait, nous pouvons commencer à nous déplacer à la fois sur les lignes et les colonnes.

Voyons donc un parcours dont l'objectif sera d'afficher correctement toutes les cases du tableau.

Faisons une fonction qui affiche le tableau de la promotion. Il s'agit, pour chaque ligne du tableau de promo, d'afficher la ligne. Afficher la ligne sera fait en appelant la fonction *afficherEtudiantV2*.

```
def afficherTab2D(tab) :

    for i in range(len(tab)) :
        afficherEtudiantV2(tab[i])

afficherTab2D(promo)
```

Si vous comprenez tout ceci, c'est bien. Encore un petit effort : Je vais faire tout le boulot en une seule fonction. Le code de la fonction *afficherEtudiantV2* sera intégré à la fonction *afficherTab2D*

```
def afficherTab2DV2(tab) :

    for i in range(len(tab)) :
      ligne = tab[i]
      for j in range(len(ligne)) :
        print (ligne[j], end = " ")

      print ("")

afficherTab2DV2(promo)
```

Enfin, le code standard pour un programmeur est obtenu en faisant disparaitre la variable *ligne* (je la remplace par *tab[i]*) :

```
def afficherTab2DV3(tab) :

    for i in range(len(tab)) :

      for j in range(len(tab[i])) :
        print (tab[i][j], end = " ")

      print ("")

afficherTab2DV2(promo)
```

Ceci est appelé une **double boucle imbriquée** (une boucle dans une boucle). Cela terrorise les jeunes informaticiens jusqu'à ce qu'ils les comprennent. Passez donc cette étape au plus vite :

Un programmeur, même débutant, interprètera ce code comme suit :

1. au coeur de la boucle, il y a *print(tab[i][j])* : J'affiche la case de la ligne i, colonne j d'un tableau 2D.

2. Cette instruction est itérée par la boucle *for* sur j : je me déplace donc sur les colonnes : cette boucle affiche toutes les colonnes de la ligne i.
3. Tout ce que je viens de faire est itérée par la boucle *for* sur i : cette boucle travaille donc successivement sur toutes les lignes.

J'ai bien une fonction qui parcours toutes les lignes, et pour chaque ligne, affiche le contenu de toutes les colonnes de la ligne.

Allez y calmement, ca va rentrer...

les fichiers

Les sources de tout ce que nous avons fait dans ce cours sont dans le répertoire Sources. Regardez les fichiers commençants par *04_...* dans l'ordre ou ils sont listés

Exercices à savoir faire

Vous **devez** être en mesure de faire les exercices suivants :

- faire un programme qui calcule la moyenne de ma promo.
- créer un tableau 2D ou chaque étudiant a 3 notes (3 contrôles continus). On pourra au plus simple avoir quelque chose comme ceci :

"Moutoussamy"

13

12

9

avec 2 étudiants.

- calculer la moyenne d'un étudiant.
- calculer la moyenne de tous les étudiants au CC 2
- calculer la moyenne de générale de ma promo

Voyons si vous êtes capable de jouer avec des doubles boucles imbriquées, meme s'il n'y a pas de tableau en jeu. Avant de commencer, on va faire simple :

- faites un programme qui, pour une variable $n = 5$ afficherait :

  `XXXXX`

- faites un programme qui, pour une variable n donnée, afficherait n lignes de n x. Ex, si n=5 on afficherait :

```
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
```

- faites un programme qui, pour une variable n donnée, afficherait une sorte de triangle. Ex, si *n=5* on afficherait :

```
X
XX
XXX
XXXX
XXXXX
```

- faites un programme qui, pour une variable n donnée, afficherait les motifs suivants (alternance sur chaque ligne de 'X' et de '_') Ex, si *n=3* on afficherait 3 lignes de 3 caracteres

```
X_X
X_X
X_X
```

si *n=5* on afficherait 5 lignes de 5 caracteres :

```
X_X_X
X_X_X
X_X_X
X_X_X
X_X_X
```

- faites un programme qui, pour une variable n donnée, afficherait les un damier (alternance sur chaque ligne de 'X' et de '_') Ex, si *n=3* :

```
X_X
_X_
X_X
```

si *n=5* on afficherait 5 lignes de 5 caracteres :

```
X_X_X
_X_X_
X_X_X
_X_X_
X_X_X
```

Cours 5 : rappels stratégie et CC1

stratégie

Suite à vos demandes, le début de ce cours a été consacré à la réalisation d'un exemple de programme. Il s'agissait de réaliser un programme permettant de gérer le déplacement d'un drone, simplement en prévoyant les variables et fonctions nécessaires.

- Le drone part d'un point déterminé au début du programme grace au GPS.
- Il doit parcourir une liste de points de passages notés sur une carte (Maps ?)
- Il doit revenir au point de départ et se poser.
- il volera à une altitude prédéterminée (disons 30m)

Les variables

- Position actuelle du drone :

Notre programme va gérer la position du drone en l'air. On pourrait stocker cette position dans un tableau 1D de float (si ce sont des coordonnées GPS). Ce tableau sera nommé *position*

x	y	z
127.14	23.47	30

- Points de passsage :

On dispose d'une liste de point, reperés par leurs coordonnées 2D (x,y) sur une carte

x	y
127.14	23.47
127.14	20.47
125.14	20.47
125.14	23.47

Il semble naturel de placer ces points dans un tableau 2D nommé *trajet*

- objectif actuel : Dans ce trajet, il faut que notre programme sache a chaque instant quel est l'objectif actuel (un des points du tableau). On peut par exemple stocker l'indice du tableau correspondant a l'objectif actuel. Cet indice sera noté *iobjectif*
- l'altitude voulue est un simple float.

Les fonctions

1. Pour connaitre la position du drone : il nous faudrait la fonction *lireGPS()* qui renvoie une position (3D)
2. Pour connaitre le trajet a effectuer, on utilisera la fonction lireTrajet() qui renvoie un tableau tel que *trajet*. Cette fonction pourrait lire un fichier par exemple
3. Pour savoir si un objectif est atteint (avec une marge d'erreur) il nous faut connaître la position du drone et la position de l'objectif. On utilisera la fonction *objectifAtteint* qui renvoie True ou False.
4. Pour déplacer le drone, on utilise la fonction *deplacer* a qui l'on communique la position 3D du drone, la position 2D de l'objectif, et l'altitude de vol voulue.
5. Pour faire atterir le drone, on utilise la fonction *atterir*.

Le programme principal

```
depart3D = lireGPS()

trajet = lireTrajet()

altitude = 30.0

# ajoutons le point de départ au trajet.
depart2D = [depart3D[0], depart3D[1]]
trajet.append(depart2D)

# notre premier objectif est dans la case 0 du tableau trajet.
iobjectif  = 0

# décollage
monter(depart3D, altitude)

# la boucle qui gère les déplacement du drone
fini = False
while (fini == False):
  # on récupere la position du drone
  position = lireGPS()

  # on déplace un peu le drone vers l'objectif, à une altitude donnée
  deplacer (position, trajet[iobjectif], altitude)

  # Si l'objectif en cours est atteind
  if (objectifAtteint(position, trajet[iobjectif] )) :
```

```
    # on passe à l'objectif suivant
    iobjectif +=1

    # si on a fini tous les objectifs
    # il est temps d'atterrir !
    if iobjectif>= len(trajet):
      fini = True

atterir (départ3D)
```

CC1

La suite du cours a été consacrée au premier Contrôle continu. Le sujet est disponible dans les annales ### Structures de données plus complexes

Au delà des variables primitives et des tableaux, on peut avoir besoin de choses plus spécifiques. Nous traiterons ici des **tableaux associatifs** et des **classes**

Tableaux associatifs

Les tableaux associatifs sont les objets permettant de stocker une association entre une **clef** et une **valeur**

Reprenons notre tableau de notes par étudiant.

indice

nom

note

0

“Moutoussamy”

13

1

“Madasaib”

17

Les tableaux associatifs permettent de remplacer les indices par autre chose (par exemple une chaine de caractères, comme le nom de l’étudiant)

Le nom (la clef) permet ainsi d’accéder à la valeur (la note de l’étudiant)

En **Python** les dictionnaires sont une forme de tableaux associatifs.

Dictionnaires, Introduction

On peut créer un dictionnaire et le remplir comme suit :

```
notes={}

notes["Moutoussamy"]=13

notes["Madasaib"]=17

notes["Naejus"] = 21

print (notes)
```

Pour modifier une note, il suffit de réaffecter la valeur associée à sa clef :

```
notes["Naejus"] = 7
print (notes)
```

Pour parcourir un dictionnaire, on utilise la syntaxe suivant, qui permet de parcourir les clefs du dictionnaire :

```
print (notes)

for clef in notes :
    print (notes[clef])
```

Si, lors du parcours, on veut afficher la clef (le nom de l'étudiant) et la valeur (la note associée), on procèderait comme suit :

```
for clef in notes :
    print (clef, notes[clef])
```

Autre application

On peut également se servir de cela pour coder des associations entre une variable et une action à effectuer.

```
action= {}
action["haut"] = "je monte"
action["bas"] = "je descends"
action["gauche"] = "je cours a gauche"
action["droite"] = "je rampe vers la droite"

direction = "droite"

print (action[direction])
```

Stockages plus élaborés

La valeur stockée peut être plus complexe qu'un simple réel. Pour stocker les notes de la promotion la valeur associée à un nom pourrait être le tableau des notes de l'étudiant :

```
notes2={}

notes2["Moutoussamy"]=[13, 8, 18]

notes2["Madasaib"]=[17, 9, 5]

notes2["Naejus"] = [12, 9, 14]

print (notes2["Naejus"])

print (notes2["Naejus"][1])
```

Stockages encore plus élaborés

Je ne ferais jamais ce qui suit dans un programme réel, j'utiliserais plutôt des classes, ce qui fera l'objet de notre prochain cours.

Cependant, cela me permettra de vous montrer deux ou trois choses et de gagner du temps après...

Imaginons que pour chaque étudiant, je veuille stocker :

- son nom
- son prénom
- son annee de naissance
- ses notes.

Je pourrais envisage de stocker les informations relatives à chaque étudiant dans un dictionnaire :

```
etu = {}
etu["nom"]= "Doe"
etu["prenom"] = "John"
etu["anneeNaissance"] = 1996
etu["notes"]=[12,9.5,18]
```

Quand je dispose d'un étudiant, je peux afficher ses informations a l'aide d'une fonction telle que *afficherEtu*

```
def afficherEtu(e):
    print ("Je m'appelle :", e["nom"])
    print ("vous pouvez m'appeler :", e["prenom"])
```

```
    print ("Je suis né en :", e["anneeNaissance"])
    print("voici mes résultats :",e["notes"])
```

Je pourrais aussi stocker tous mes étudiants dans un tableau de promo, chaque case contenant un dictionnaire des infos sur l'étudiant...

```
promo = []

etu = {}
etu["nom"]= "Doe"
etu["prenom"] = "John"
etu["anneeNaissance"] = 1996
etu["notes"]=[12,9.5,18]

promo.append(etu)

etu = {}
etu["nom"]= "Sarr"
etu["prenom"] = "Bouna"
etu["anneeNaissance"] = 1998
etu["notes"]=[14,8.5,16]

promo.append(etu)
```

Pour afficher les infos d'un étudiant de la promo, je ferais par exemple :

```
afficherEtu(promo[0])
```

pour afficher toutes les infos de toute la promo :

```
for e in promo :
    afficherEtu(e)
```

Les fichiers sources

Les sources de tout ce que nous avons fait dans ce cours sont dans le répertoire Sources.

Les Classes

Ici, on rentre dans le domaine de la **Programmation Orientée Objet** (ou POO pour les intimes).

Nous allons pouvoir programmer des objets qui disposent de propriétés et des fonctions spécifiques...

Reprenons nos étudiants et leurs informations : nous allons créer une **Classe** *Etudiant* qui correspondra à un nouveau type de données.

Pour créer une Classe, on ecrit simplement :

```
class Etudiant:
```

Tout ce qui concerne cette classe est indenté (décalé vers la droite).

La première chose à faire est de définir ce qu'il faut faire quand on construit un objet de type Etudiant. C'est la fonction ***init*** qui s'en charge (on pourrait parler de **constructeur** de la classe)

Cette fonction est une fonction interne de la classe (ce que l'on appelle une **méthode**). Toutes les méthodes ont comme premier paramètre la variable **self** qui représente l'objet lui même.

D'ou le code suivant :

```
class Etudiant:
    def __init__(self):
        self.nom=""
        self.prenom=""
        self.anneeNaissance=0
        self.notes=[]
```

Ma classe *Etudiant* dispose d'un constructeur, qui pour chaque objet de type étudiant lui crée un champ *nom*, un champ prénom, un champ *anneeNaissance* et un tableau de notes.

Une classe avec des propriétés

Pour créer un objet de type *Etudiant* (on parle d'une **instance** de la classe Etudiant), on procèderait comme suit :

```
etu = Etudiant()
etu.nom= "Doe"
etu.prenom = "John"
etu.anneeNaissance = 1996
etu.notes=[12,9.5,18]
```

Les champs *nom*, *prenom*, *anneeNaissance* et *notes* sont des **propriétés** de la classe.

Je pourrais faire une fonction qui permette d'afficher les informations relatives à un étudiant comme suit :

```
def afficherEtu(e):
    print ("Je m'appelle :", e.nom)
    print ("vous pouvez m'appeler :", e.prenom)
    print ("Je suis né en :", e.anneeNaissance)
    print("voici mes résultats :",e.notes)
```

et on peut appeler cette fonction comme suit :

```
afficherEtu(etu)
```

Une classe avec des méthodes

La fonction définie ci-dessus *afficherEtu* est une fonction extérieure à la classe. On peut la transformer en fonction interne à la classe. Cela devient alors une **méthode** de la classe.

Notre classe deviendrait :

```
class Etudiant:

    def __init__(self):
        self.nom=""
        self.prenom=""
        self.anneeNaissance=0
        self.notes=[]

    def afficher(self):
        print ("Je m'appelle :", self.nom)
        print ("vous pouvez m'appeler :", self.prenom)
        print ("Je suis né en :", self.anneeNaissance)
        print("voici mes résultats :",self.notes)
```

La méthode *afficher* se comprend comme suit : - son argument est *self*, car une méthode doit disposer d'une variable correspondant à l'objet sur lequel elle travaille.

- dans le corps de la fonction, si l'objet doit afficher son nom, il utilise bien *self.nom* soit "mon nom à moi..."

Nous pouvons maintenant créer un étudiant et afficher ses informations comme suit :

```
etu = Etudiant()
etu.nom= "Doe"
etu.prenom = "John"
etu.anneeNaissance = 1996
etu.notes=[12,9.5,18]

etu.afficher()
```

Encore une fois, on peut utiliser ces objets en placant nos étudiants dans un tableau représentant la promotion complète :

```
promo = []

etu = Etudiant()
etu.nom= "Doe"
etu.prenom = "John"
etu.anneeNaissance = 1996
etu.notes=[12,9.5,18]
```

```
promo.append(etu)

etu = Etudiant()
etu.nom= "Thauvin"
etu.prenom = "Florian"
etu.anneeNaissance = 1998
etu.notes=[14,8.5,16]

promo.append(etu)
```

et pour afficher le tout

```
for e in promo:
    e.afficher()
```

Enfin, on peut également préparer quelques méthodes supplémentaires :

- Une méthode qui ajoute une note au tableau de notes :

Cette méthode est intéressante, car elle on lui passe un paramètre (en plus de self) : la note à inserer dans le tableau.

```
  def ajouterNote(self, note):
    self.notes.append(note)
```

- la méthode qui calcule la moyenne d'un étudiant :

```
  def calcMoyenne(self):
    moyenne = 0
    for n in self.notes:
        moyenne += n
    moyenne = moyenne / len(self.notes)

    return moyenne
```

- une méthode qui compare l'etudiant à un autre :

Celle ci est aussi intéressante car on lui passe en argument un autre Etudiant auquel *self* doit se comparer...

```
  def meilleurQue(self,e):
    moyenneMoi = self.calcMoyenne()
    moyenneLui = e.calcMoyenne()

    if moyenneMoi > moyenneLui :
        print (e.nom," t'es un gros loser")
    else :
        print (e.nom," un jour je t'aurais")
```

et les appels de ces méthodes :

```
for e in promo:
    e.afficher()
    print(e.calcMoyenne())

promo[0].meilleurQue(promo[1])
```

Les Classes, héritage, surcharge

Ici, nous allons travailler avec un monde rempli d'animaux de toutes sortes. La classe la plus générale sera celle des animaux.

Un animal a une race et un age... On peut écrire ce qui suit :

```
class Animal:

    def __init__(self):
        self.race=""
        self.age=0

    def presenter(self):
        print ("je suis de la race des", self.race)
```

Et on peut créer des animaux comme suit :

```
a = Animal()
a.race = "ravet"

a.presenter()

c = Animal()
c.race = "vache"
c.presenter()
```

Tous les animaux savent se présenter, c'est chouette...

Si l'on veut aller plus loin, nous allons créer une Classe *Mammifere*. Un Mammifère étant un Animal, nous souhaiterions que notre Mammifère ait aussi une race, un age et sache se présenter. Cela sera obtenu par le fait que la classe *Mammifere* **hérite** de la classe *Animal*, grâce à cette ligne

```
class Mammifere(Animal):
```

De plus, quand on construit un Mammifère, on commence par construire un animal. Le constructeur de la classe *Mammifere* appelle donc le constructeur de la classe *Animal*

```
    def __init__(self):
        Animal.__init__(self)
```

Ajoutons aux objets de la classe *Mammifere* la méthode qui décrit eur grossesse :

```
    def decrireGrossesse(self):
        print ("je porte mes petits dans mon ventre")
```

Nous pouvons maintenant créer un Mamifère et utiliser tout ce qu'il sait faire comme suit :

```
b = Mammifere()
c.race = "singe"
c.presenter()
c.decrireGrossesse()
```

Il faut noter que la méthode *presenter* vient de la classe *Animal* alors que la méthode *decrireGrossesse* vient de la classe *Mammifere*. Mais comme un Mammifere est un Animal, tout fonctionne.

La classe *Mammifere* **hérite** des méthodes de *Animal*.

On peut aller encore un peu plus loin avec une classe *Chien*

```
class Chien(Mammifere):
    def __init__(self):
        Mammifere.__init__(self)
        self.race = "chien"

    def cri(self):
        print ("ouaf")

    def decrireGrossesse(self):
        Mammifere.decrireGrossesse(self)
        print ("temps de gestation : 2 mois")
```

Notez que la classe *Chien* propose la méthode *cri* qui donne le cri du chien.

Notez surtout la méthode *decrireGrossesse* qui vient remplacer la méthode du même nom de la classe *Mammifere*. Dans cette méthode, on appelle la méthode de Mammifère (car elle reste valable) et on y ajoute des informations spécifiques aux chiens).

On dit que *decrireGrossesse* **surcharge** la méthode de la classe parente...

Voila !

Les fichiers sources

Les sources de tout ce que nous avons fait dans ce cours sont dans le répertoire Sources. ## Quelques liens externes

Le problème quand on enseigne l'informatique (ou quoi que ce soit d'autre), est d'adapter son discours a son public. Ce que je propose dans ce cours est un kit de survie pour l'algorithmique et la programmation utile pour tout ingénieur.

J'y fais d'énormes raccourcis. Si vous souhaitez aller un peu plus loin voici quelques liens que j'ai glané sur internet.

Un tutoriel pour se former :

Si vous souhaitez des informations plus détaillées, d'autres exemples, des exercices à faire, voici un tutoriel Je vous conseille de faire tout le début, vous pourrez vous arrêter quand vous voudrez.

Le tutoriel officiel :

Vous voulez quelque chose de plus rigoureux, toujours sous forme de tutoriel ? Voici un autre tutoriel Il est bien aussi, mais il suppose souvent que vous travaillez sous Linux et/ou que vous ayez quelques bases en informatique...

Un cours :

Vous souhaitez un gros pdf avec tout ce qu'on peut vouloir savoir sur python ? Voici un gros cours J'aurais tendance à louer son côté très complet, tout en lui reprochant de ne pas être ...synthétique...

Autres ressources

Si vous trouvez d'autres ressources utiles, n'hésitez pas à m'en faire part, je pourrais les ajouter...

Vous pouvez repartir vers le Sommaire _____