Chapitre 1

Brève introduction

Ce chapitre n'a pas pour but de constituer un cours exhaustif sur l'emploi de *Python*, il donne simplement quelques clés pour appréhender les problèmes classiques de la programmation avec ce logiciel. L'outil d'aide principal reste la commande *help* suivie du nom de la fonction dont on souhaite avoir de plus amples informations.

1. Variables et affectation

1.1. Présentation

L'instruction d'affectation est la commande $=$

Exemples :

- L'instruction $\boxed{n=1}$ stipule que la variable n est de type entière (*int*) et qu'elle possède la valeur 1.
- L'instruction $\boxed{n=n+1}$ précise que la valeur numérique de la variable entière n est remplacée par sa valeur précédente incrémentée de 1. L'instruction d'affectation n'est pas symétrique : la variable est toujours à gauche.
- L'instruction $\boxed{R=8.314}$ indique que la valeur numérique de la variable réelle R vaut 8,314 et que cette variable est de type réelle ou « à virgule flottante (*float* en anglais) ».
- La commande $\boxed{T=\text{"}bonjour\text{"}}$ place la chaîne de caractères bonjour dans la variable T.
- $\boxed{A=[8,9,4]}$ correspond à une liste ou une matrice ligne $A=(a_i)_{1\leq i\leq 3}$ appartenant à $\mathcal{M}_{1,3}(\mathbb{R})$. Attention, le premier élément de cet liste est 8 et il s'obtient de la façon suivante $A[0]$: le premier indice d'une liste est l'indice nul.

– Il est également possible d'effectuer des affectations parallèles comme l'instruction : $\boxed{x, y = 4, 4.13}$, la virgule sépare les deux affectations, x vaut 4 et y vaut 4,13.

1.2. Opérations mathématiques

À droite du signe d'affectation il est possible d'effectuer toutes les opérations mathématiques courantes (+ : addition, – : soustraction, / : division, * : multiplication, ** : puissance) et toutes les fonctions usuelles si on dispose des modules nécessaires ou d'un éditeur efficace comme *spyder* par exemple.
Python dispose également de l'opérateur modulo représenté par le symbole %, ainsi la commande 10%2 renvoie 0 alors que la commande 21%2 donne 1.

1.3. Opérateurs logiques et relationnels

Les opérateurs suivants permettent de comparer des éléments :

Symbole Mathématique	Symbole python	Signification « numérique »
$=$	==	Egal à
$<$	<	Strictement inférieur à
$>$	>	Strictement supérieur à
$\leq$	<=	Inférieur
$\geq$	>=	Supérieur
$\neq$	!=	Différent de

L'opération de comparaison renvoie *True* si l'expression est vraie et *False* sinon.

1.4. Notion de fonctions

Un programme peut être écrit sous forme de script, il s'agit alors d'une suite d'instructions et commandes destinées à effectuer des opérations conduisant au résultat souhaité. On peut néanmoins préférer la notion de fonction dont le but est d'appeler un certain nombre d'arguments pour fournir en sortie les variables résultats. D'autres langages proposent les termes de *subroutine* ou de *procédure*.
La syntaxe est la suivante :

```
def  mafonction(arg1,arg2):
      rep=arg1+ arg2
      return rep
```

mafonction est le nom de la fonction qui renvoie la variable *rep* à partir des arguments *arg*1 et *arg*2. Les instructions nécessaires à la fonction sont indentées après le « : » qu'il ne faut pas oublier !
Cette fonction élémentaire permet de calculer la somme de deux termes, elle s'appelle de la façon suivante : a=mafonction(5,6) ; print a. Le résultat est bien sûr 11, il est affiché grâce à la commande *print*.

2. Les structures

2.1. La structure de test : *if…elseif…else*

La syntaxe est la suivante :

Syntaxe	*Exemple*

Déclaration de la fonction :
  if expression 1 *vérifiée :*
   Instruction 1
  elseif expression 2 *vérifiée :*
   Instruction 2
  elseif expression 3 *vérifiée :*
   Instruction 3
…(*autant de elseif que désiré*)
  else :
  Instruction 4
	`def fonctionF(x):`
	`    if (x<0):`
	`        y=0`
	`    elif (x>=0)and(x<1):`
	`            y=x`
	`    elif (x>=1)and(x<2):`
	`            y=-x+2`
	`    else:`
	`            y=0`
	`    return y`

L'indentation est essentielle, c'est elle qui délimite les différentes conditions.
L'exemple proposé est en fait une alternative à la fonction :

$$x \mapsto \frac{|x| - 2|x-1| + |x-2|}{2}$$

Remarque : Cet exemple montre l'utilisation de tests conditionnels multiples « et » réalisés grâce à la commande « and ». Le test « ou » s'effectue avec l'instruction « or ».

2.2. La structure de répétition : *while...end*

La syntaxe est la suivante :

Syntaxe	*Exemple*

while expression vérifiée :
   Instruction
	`x=0`
	`while (x<25):`
	`    x=x+1 # ou x+=1`
	`    print x`

Dans l'exemple, x vaut initialement 0 puis tant que x reste strictement inférieur à 25, sa valeur augmente d'une unité ; les valeurs affichées vont de 1 à 25 par pas de 1.

2.3. La structure de répétition avec compteur *for...end*

La syntaxe est la suivante :

Syntaxe	*Exemple*
for affectation du compteur : *Instruction*	`for x in range(1,26,1):` `  print x`

Comme précédemment, le programme affiche les valeurs de 1 à 25 par pas de 1. Attention la valeur 26 est exclue !

3. Application : la suite de Syracuse

3.1. Présentation

Le principe de sa construction est le suivant. La suite est initialisée par un entier naturel, s'il est pair, il faut le diviser par deux et s'il est impair on le multiplie par trois et on ajoute 1. Ces opérations permettent de construire la suite $(u_n)_{n\in\mathbb{N}}$ telle que, $u_0 \in \mathbb{N}$:

$$\forall n \in \mathbb{N}^* \; u_{n+1} = \frac{u_n}{2} \text{ si } u_n \text{ est pair et } u_{n+1} = 3u_n + 1 \text{ si } u_n \text{ est impair}$$

Ainsi, à partir de $u_0 = 5$ apparaît la suite de nombres : 5 ; 16 ; 8 ; 4 ; 2 ; 1 puis à partir de 1 à nouveau 4 ; 2 ; 1 et ainsi de suite pour un cycle sans fin. La conjecture de Syracuse annonce qu'à partir de n'importe quelle valeur de départ, la suite finit toujours par retomber sur le cycle 4 ; 2 ; 1. Il s'agit d'une conjecture car ce résultat n'a toujours pas trouvé de preuve malgré les nombreuses tentatives de mathématiciens émérites. Dans le cas des suites de Syracuse, le temps de vol correspond au nombre de valeurs prises par la suite avant de retomber sur 1 (5 valeur pour l'exemple précédent) ; et l'altitude représente la valeur maximale de la suite (16 dans l'exemple).

3.2. Codes

Il existe de nombreuses façons de procéder pour programmer une suite selon le résultat espéré. Dans tous les cas, il faut se donner une valeur de départ pour u_0.

Pour connaître uniquement le $n^{ème}$ terme de la suite, il est inutile de stocker toutes les valeurs précédentes qui peuvent donc être écrasées au fur est à mesure de l'exécution du code.

Fonction	*Commentaires*
`def syrac1(u0,n):`	Déclaration de la fonction : la valeur de u_0 et l'indice final n sont les arguments d'entrée.
`    u=u0`	Initialisation de la variable u
`    for i in range(n):`	Boucle for avec compteur i sur l'indice de la suite. range(n) est équivalent à range(0,n,1)
`        if u%2==0:`	Si u est pair
`            u=u/2`	Division par 2
`        else:`	Sinon
`            u=3*u+1`	Multiplication par 3 et ajout de 1
	Fin du test
	Fin de la boucle avec compteur.
`    return u`	La fonction renvoie la valeur de u_n.

Si la fonction est appelée dans le script du programme, il suffit d'ajouter les instructions u=syrac1(2919,120) ; print(u) pour connaître par exemple le $120^{ème}$ terme de la suite démarrant à partir de 2919. En revanche, si la fonction a par exemple été enregistrée dans le fichier syracuse.py et que l'on souhaite utiliser la fonction syrac1 dans un autre fichier, il est alors nécessaire de l'importer avant :

Commandes	*Commentaires*
`from syracuse import syrac1`	La fonction syrac1 est importée depuis le fichier syracuse.py
`u=syrac1(2919,120)`	Utilisation de la fonction
`print(u)`	Affichage du résultat

Dans le cas où l'utilisateur souhaite conserver l'ensemble des valeurs de la suite, il faut utiliser une liste.

Fonction	*Commentaires*
`def syrac2(u0,n):`	Déclaration de la fonction : la valeur de u_0 et l'indice final n sont les arguments d'entrée.
`    u=[0]*(n+1)`	Initialisation de la liste des valeurs de la suite $(u_n)_n$ avec n+1 valeurs nulles (cf. de l'indice 0 à n)
`    u[0]=u0`	Initialisation par la valeur u_0 à l'indice 0
`    for i in range(0,n):`	Boucle for avec compteur i sur l'indice de la suite
`        if u[i]%2==0:`	Si u_i est pair
`            u[i+1]=u[i]/2`	Division par 2
`        else:`	Sinon
`            u[i+1]=3*u[i]+1`	Multiplication par 3 et ajout de 1
	Fin du test et fin de la boucle avec compteur (cf. indentation)
`    return u`	La fonction renvoie la liste des valeurs de la suite $(u_n)_n$

Une autre façon de programmer cette suite est de considérer la liste *u* comme un objet et de lui appliquer la méthode append (qui signifie ajouter en anglais), il s'agit d'une sorte de fonction qui est attachée aux objets du type liste.

Fonction	*Commentaires*
`def syrac3(u0,n):`	Déclaration de la fonction : la valeur de u_0 et l'indice final n sont les arguments d'entrée.
`    u=u0`	
`    u_liste=[u0]`	Initialisation de la liste avec le premier élément
`    for i in range(1,n+1):`	Boucle for avec compteur i sur l'indice de la suite
`        if u%2==0:`	Si u est pair
`            u=u/2`	Division par 2
`        else:`	Sinon
`            u=3*u+1`	Multiplication par 3 et on ajout de 1
	Fin du test
`        u_liste.append(u)`	Ajout de la valeur à la liste
`    return u_liste`	La fonction renvoie la liste des valeurs de la suite $(u_n)_n$

L'exécution de la fonction permet alors d'obtenir une représentation graphique du résultat.

Script	*Commentaires*
`from syracuse import syrac2`	Importation des modules nécessaires
`import matplotlib.pylab as plt`	
`u=syrac2(2919,190)`	Appel de la fonction syrac2 pour les arguments d'entrée u_0 = 2919 et n = 190
`indice=range(0,191,1)`	Définition des indices de 0 à 190 (191 exclu) par pas de 1
`plt.plot(indice,u,'k-')`	Représentation graphique (couleur noire : blac<u>k</u>)
`plt.xlabel('Indices',fontsize='medium')`	Les axes sont étiquetés
`plt.ylabel('Valeurs de la suite',fontsize='medium')`	
`plt.show()`	

Dans le cas de cette suite, il est utile d'observer ses valeurs jusqu'au premier cycle. Une boucle *for* est donc inappropriée puisque le temps de vol est inconnu. Il faut donc utiliser une boucle *while*, tournant tant que la suite n'a pas atteint la valeur 1. L'insertion d'un test (*if*) permet de déterminer l'altitude. La fonction ne nécessite alors que la connaissance de la valeur initiale mais retourne 3 variables réponses : la suite, l'altitude et le temps de vol.

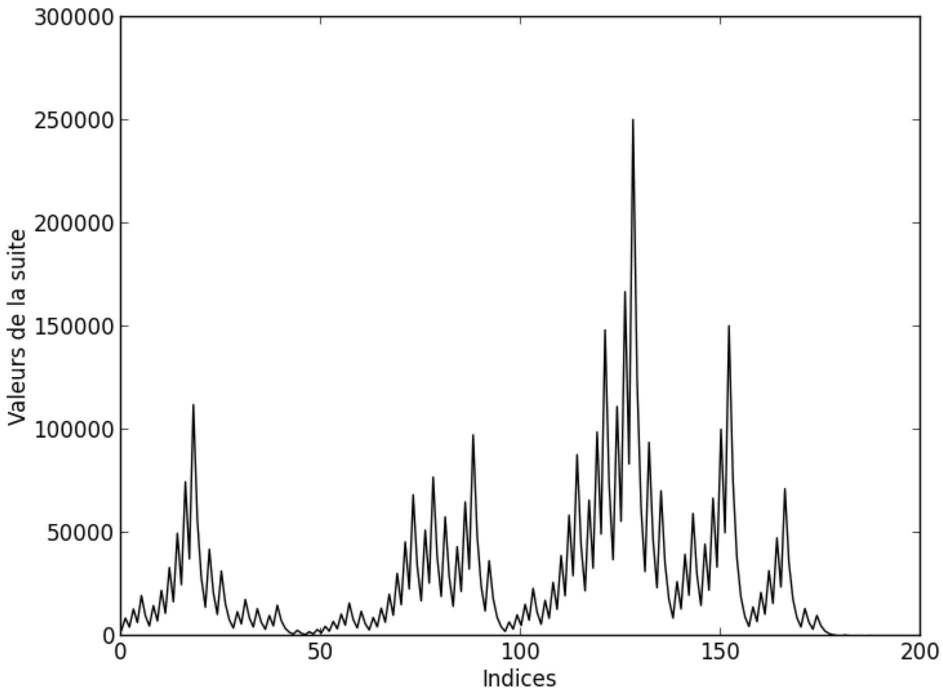


Figure 1 : Représentation graphique des 190 premiers termes de la suite de Syracuse obtenue à partir de la valeur initiale 2919.

Fonction dans syracuse	*Commentaires*
`def syrac4(u0):`	Déclaration de la fonction : arguments d'entrée : la valeur de u_0
`    u_liste=[u0]`	Initialisation de la liste avec le premier élément
`    alti=u0`	Initialisation de l'altitude
`    n=0`	Initialisation de l'indice (ou compteur)
`    u=u0`	
`    while u[n]>1:`	Boucle while tant que u n'atteint pas la valeur 1
`        if u%2==0:`	Si u est pair
`            u/=2`	Division par 2 (en place)
`        else:`	Sinon
`            u=3*u+1`	Multiplication par 3 et ajout de 1
`        n+=1`	Incrément du compteur des indices
`        u_liste.append(u)`	Ajout de l'élément suivant dans la suite
`        if u>alti:`	Si la valeur de la suite est supérieure à l'altitude maximum précédente
`            alti=u[n]`	alors cette valeur devient la nouvelle altitude
	Fin du test de l'altitude
	Fin de la boucle while
`    return u_liste,n,alti`	Réponses : la suite, le temps de vol et l'altitude

Ainsi la suite partant de 2919 possède une altitude de 250504 et un temps de vol de 216. Ces valeurs s'obtiennent avec les instructions suivantes :

Script	*Commentaires*
`from syracuse import syrac4`	Importation de la fonction nécessaire
`u,n,alti=syrac4(2919)`	Appel de la fonction syrac4 pour les arguments d'entrée u_0 = 2919
`print(n,alti)`	Affichage du temps de vol et de l'altitude

Chapitre 2

Les tris

Un algorithme est composé d'un ensemble d'opérations élémentaires, organisé selon un schéma et des règles précises dont le but est de résoudre un problème donné c'est-à-dire de répondre aux contraintes imposées par l'utilisateur.

Le tri est une opération courante surtout pour les personnes ordonnées, ce procédé est notamment utilisé par les banques, les bibliothèques, les concours,… L'optimisation des tris est un enjeu de taille pour l'algorithmique, en effet des études tendent à montrer qu'un quart des cycles machine sont occupés par le tri. Il s'agit également d'un problème fondamental de l'algorithmique c'est pourquoi il existe des dizaines d'algorithmes répondant à ce problème, utilisant quelques principes de base auxquels s'ajoutent des variantes.
Dans la suite la liste à trier est notée A, il s'agit d'un tableau (une matrice) à 1 ligne et n colonnes ; n représente alors le nombre d'éléments à trier.

1. Tri par sélection (ou extraction)

1.1. Principe

La liste est parcourue pour trouver le minimum qui est alors placé en première position par permutation, puis ce principe est réitéré sur le reste de la liste.
Remarque : On peut également prendre le maximum et le placer en dernière position.

1.2. Algorithme

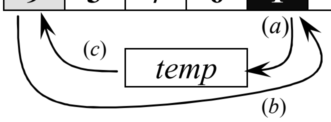
L'algorithme est présenté avec un langage universel, les codes sont donnés dans le paragraphe 1.3.

1	***pour*** $i \leftarrow 0$ *à* $n-2$ *faire*	Lecture de la liste de l'indice 0 à *n*–2 (gauche à droite)
2	$\min \leftarrow i$	Initialisation de l'indice de la valeur minimale recherchée
3	***pour*** $j \leftarrow i+1$ *à* $n-1$ *faire*	Lecture de la liste sur les indices à gauche de *i*
4	***si*** $A[j] < A[\min]$	Compare la valeur lue à la valeur précédente du minimum. Si la valeur lue est inférieure
6	$\min \leftarrow j$	Alors on a trouvé l'indice du nouveau minimum
7	***fin si***	Fin de la boucle de test
8	***fin pour***	Fin de la boucle de lecteur
9	***si*** $\min > i$	Si l'indice du minimum est plus grand que l'indice lu (boucle pour ligne 1)
10	$temp \leftarrow A[\min]$	On effectue la permutation entre les valeurs de la liste aux indices *i* et min. Il est nécessaire d'utiliser une variable tampon (*temp*) pour effectuer la permutation.
11	$A[\min] \leftarrow A[i]$	
12	$A[i] \leftarrow temp$	
13	***fin si***	Fin de la boucle de test
14	***fin pour***	Fin de la lecture de la liste (boucle pour)

On considère la liste d'entiers $[9, 3, 7, 6, 1, 8]$

0	1	2	3	4	5	: *indices*
9	**3**	**7**	**6**	**1**	**8**	: éléments à trier

0	1	2	3	**4**	5	: *indice* 0 pour le début de lecture (boucle pour de la ligne 3) et indice 4 du min.
9	**3**	**7**	**6**	**1**	**8**	Première lecture de 0 à 5 pour trouver le minimum (noir) en 4

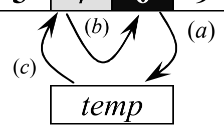


Comme l'indice 4 (du minimum) est strictement supérieur à 0, on permute le 1 et le 9 selon l'ordre des flèches (*a*), (*b*) puis (*c*)

0	**1**	2	3	4	5	: *indice* 1 du début de lecture et indice 1 du minimum suivant
1	**3**	**7**	**6**	**9**	**8**	Deuxième lecture de 1 à 5 pour trouver le minimum (noir) en 1

Comme l'indice 1 (du minimum) n'est pas strictement supérieur à l'indice de début de lecture, il n'y a pas de permutation.

0	1	**2**	**3**	4	5	: *indice* 2 du début de lecture et indice 3 du minimum suivant
1	**3**	**7**	**6**	**9**	**8**	Troisième lecture de 2 à 5 pour trouver le minimum (noir) en 3



Comme l'indice 3 (du minimum) est strictement supérieur à 2, on permute le 6 et le 7 selon l'ordre des flèches (*a*), (*b*) puis (*c*)

0	1	2	**3**	4	5	: *indice* 3 du début de lecture et indice 3 du minimum suivant
1	**3**	**6**	**7**	**9**	**8**	Quatrième lecture de 3 à 5 pour trouver le minimum (noir) en 3

Comme l'indice 3 (du minimum) n'est pas strictement supérieur à l'indice de début de lecture, il n'y a pas de permutation.