29 août 2014

Résumé

```
from numpy import *
```

Document en perpétuelle évolution.

Table des matières

1 Commandes de base

1.1 Assigner/afficher/demander/aide

Action	Entrée	Sortie
Assigner	`x=y`	Stocke l'état de la variable `y` dans la variable `x`
Assigner (multiple)	`a,b=0,1`	Pareil que `a=0` et `b=1`
Incrémenter	`x+=y`	Change `x` en `x+y` *in place*
Décrémenter	`x-=y`	Change `x` en `x-y` *in place*
Afficher (sortie écran)	`print("salut")`	Affichage de `salut` à l'écran (et c'est tout)
Poser une question	`age=input("Quel âge as tu?")` (question entre "" optionnelle)	Stocke la réponse dans la variable `age`
Renvoyer un résultat (sortie mémoire)	`return resultat`	Renvoie `resultat` (uniquement dans une fonction) ($\Longrightarrow$ sortie de fonction)
Demander le type	`type(objet)`	type Python
Catalogue objet	`dir(objet)`	Liste fctions/méthodes relatives
Aide en ligne	`help(commande)`	Aide
Commentaires	`#`	Passe le reste de la ligne en mode commentaire

1.2 Opérations/fonctions usuelles

• **Remarque :** DE = « division euclidienne ».

Action	Python 2.7	Python 3+	Exemple
Addition	+	+	`2+3` → `5`
Multiplication	*	*	`2 * 3` → `6`
Puissance	**	**	`2 * 3` → `8`
Division décimaux	Cf. remarque	/	`5/2` → `2.5` en Python 3
Quotient DE	/	//	`7/3` → `2` en 2.7
Reste DE	%	%	`7 % 3` → `1`
Addition `array`	+	+	`array([1,2])+array([2,3])` → `array([3,5])`
Addition/concaténation `list` ou `str`	+	+	`[1,2]+[3,4]` → `[1,2,3,4]`
Multiplication/concaténation `list` ou `str`	*	*	`[1,2]*3` → `[1,2,1,2,1,2]`
Fonction usuelle	Nom usuel	Nom usuel	pas classique : `sqrt(2)` (racine) `floor(4.5)` (partie entière)

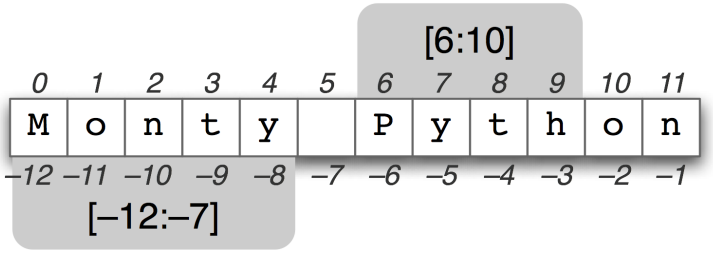
• **Remarque :**

➢(division décimale Python 2.7) « / » ne renverra le résultat attendu QUE si l'un des acteurs est de type `float`, ce qu'il faut parfois préciser à Python. Par exemple pour « 5 diviser par 2 » on tape `5./2` ou `5/2.` ou `float(5)/2`.

➢on peut s'affranchir de cette particularité en validant `from __future__ import division` qui passer la division en mode Python 3.

2 Types Python

• **Indexation des itérables :**



⚠l'indexation commence à zéro et pas un donc si `truc='Monty Python'` on accède à `'M'` par `truc[0]`.

Type	Ex. maths	Python	Exemple	Itérable
Booléen	Vrai ou Faux	`bool`	`True/False`	x
Entier	12	`int`	`12`	x
Décimal	1,2	`float`	`1.2`	x
Complexe	$1+2i$	`complex`	`1+2j`	x
Fonction	$f : x \mapsto e^{-x}$	`function`	`f = lambda x: exp(-x)`	x
Liste	[1,2,3]	`list`	`l=[1,2,3]`	`l[1]` → `2`
Liste (de listes)	[[1,2],[3,4]]	`list`	`l=[[1,2],[3,4]]`	`l[0]` → `[1,2]` `l[1][1]` → `4`
Ensemble	$\{1,2,3\}$	`set`	`s={1,2,3}`	`s[2]` → `3`
p-uplet	t=(1,2,3)	`tuple`	`t=(1,2,3)`	`t[0]=1`
Chaîne caractères	Texte normal	`str`	`mot="Salut"`	`mot(1)="a"`
Tableaux	$T = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$	`array`	`T=array([[1,2,3],[4,5,6]])`	`T[1,1]` → `5` `T[0:2,1]` → `[2,5]`
Conversion			Voir remarque	

• **Remarque :**

① **(conversion d'un objet en un autre type)** [a] `nouveau=nouveautype(ancien)`. Exemples :

➢conversion ensemble → liste : `a=list({1,2,3})` → `a= [1,2,3]` ;

➢conversion décimal → chaine de caractères : `mot=str(12)` → `mot="12"`.

② **liste des objets vides :** `[]` liste vide, `{ }` ensemble vide, `""` chaine vide.

2.1 Zoom sur les listes/tableaux

a. À l'exception de la conversion `lst` → tableau multi-dimensionnel, voir la commande `reshape` en section 2.1.2.

2.1.1 Fonctions/méthodes

Action	Entrée	Sortie
Longueur d'une liste	`len( [1,2,3] )`	3
Rajouter à une liste	`a.append(4)`	Rajoute 4 à la fin *in place* de `a`
Concaténer listes	`[1,2,3]+[4]`	`[1,2,3,4]`
Répéter liste	`[1,2]` $*$ 3	`[1,2,1,2,1,2]`
Expulser un élément	`a.pop()`	Expulse le dernier élément de `a` *in place*
Trier	`a.sort()`	Trie `a` par ordre croissant *in place*
Symétrie centrale	`a.reverse()`	Inverse les éléments d'une liste *in place*
Tester l'appartenance	`1 in [1,2,3]` ou 5 in [1,2,3]	`True` ou `False`
Format tableau	`tableau.shape`	tuple (nbre lgn, nbre col,etc.)
Extraire (*sclicing*) 1-d	Si `a=[0,1,2,3,4,5]`	`a[0:3]`$\rightarrow$ `[0,1,2]`
Extraire tableau 2d	Si `a=array( [ [1,2,3] , [4,5,6] ] )`	`a[0:2,0:2]` $\rightarrow$ `[ [1,2],[4,5] ]`
Compter apparition	Si `a=[1,1,2,3]`	`a.count(1)` $\rightarrow 2$
Créer une copie de `a`	`b=a.copy()`	Crée un nouveau pointeur `b`
Tester égalité d'`array`s	`list(a)==list(b)`	Càd convertir en listes

2.1.2 Générer des listes/tableaux

Action	Entrée	Sortie
Générer liste d'entiers	`range(6)` ou `range(1,4)`	`[0,1,3,4,5]` ou `[1,2,3]` ⚠pas le dernier par ex. `range(a,b)` $\rightarrow [[a,b[[$
Générer subdivision de $[0,10]$ à 3 pts	`linspace(0,10,3)`	`array([ 0., 5., 10.])`
Génération par compréhension	`[i**2 for i in range(4)]`	`[0,1,4,9]`
Génération filtrée	`[i for i in range(6) if i%3 == 0]`	`[0,3]` ici filtre = « reste de i par 3=0 »
Multiplication/concaténation	`[0]*3` ou `[ [1,2]*2 ]`	`[0,0,0]` ou `[[1,2],[1,2]]`
Générer tableau zéros	`zeros((2,3))`	`[ [0,0,0],[0,0,0] ]`
Générer tableau contenant type x	`array( [lgn,col] ,dtype=x)`	Tableau lgn $\times$ col à *data type* x
Extraire (*sclicing*) 1-d	Si `a="python"`	`a[0:3]`$\rightarrow$ `"pyt"`
Conversion liste $\rightarrow$ tableau	`reshape(liste,(3,2))`	`liste` à 6 éléments $\rightarrow$ tableau 3 x 2

Plus de compléments sur le **slicing**, voir section 4.4 page 14.

2.2 Tableaux vus comme des matrices

• Les matrices peuvent être vus comme des tableaux 2-d, par exemple :
$M = \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}$ devient `M = array( [ [1,-1],[1,1] ] )` mais ⚠à la syntaxe du produit matriciel.

Action	Entrée	Sortie
Inverse	`linalg.inv(M)`	`matrix([[ 0.5, 0.5], [-0.5, 0.5]])`
Produit matriciel	`dot(A,B)`	Produit $A \times B$
Déterminant, trace	`det(M),tr(M)`	`2,2`
Matrice identité	`identity(3)`	I_3
Générer diag(a,b,c)	`diag(liste)`	Matrice diagonale à coeff. formés par `liste`

• **Méthodes avancées** de calcul formel (donc HP, mais peut-être utile) :

Action	Entrée	Sortie
Puissance	`linalg.matrix_power(M,3)`	Puissance 3ème
Résolution $MX = Y$	`linalg.solve(M,Y)`	Solution(s) X
Valeurs propres	`linalg.eigvals(M)`	Vecteur des VP (avec répétition)
VP + Vecteur propres	`linalg.eig(M)`	Vecteur des VP + matrice constituée de $\overline{VP}$ unitaires associés dans l'ordre aux VP

2.3 Zoom sur les chaînes de caractères

Action	Entrée	Sortie
Longueur d'une chaîne	`len("python")`	`6`
Concaténer chaînes	`"pyt"+"hon"`	`"python"`
Aller à la ligne	`" Salut \ n ça va?"`	Va à la ligne à l'affichage
Répéter chaîne	`"Ok" * 3`	`"OkOkOk"`
Tester l'inclusion	`"py" in "python"` ou `"pi" in "python"`	`True` ou `False`
Remplacer dans	`"python".replace("py","zz")`	`"zzthon"`
Découper selon	si `mot="a b c"` alors `mot.split(" ")`	liste `["a","b","c"]`
Compter apparition	si `a="coco"`	`2`

3 Compléments pour les TIPE

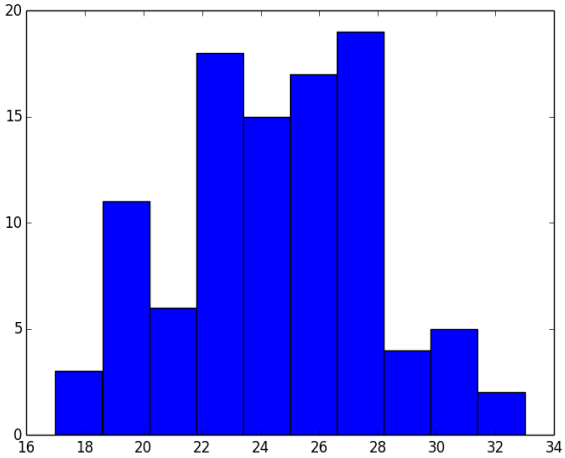
3.1 Générer de l'aléatoire/probabilités

• Tout vient du package `random` : `from random import *`

Action	Entrée	Sortie
Entier aléatoire ds $[1;5[$	`randint(1,5)`	`3`
Flottant aléatoire dans $[0,1[$	`random()`	`0.243`
Génération tableau 3 x 2 d'entiers aléatoires ds $[\![1\ ;\ 4]\!]$	`random.randint(1,5,(3,2))`	`[[2,3,1],[1,1,4]]`
4 tirages suivant une loi uniforme sur $[0,1]$	`random.rand(4)`	`[0.23423,...]`
4 tirages suivant une loi normale $\mathscr{N}(0,1)$	`random.randn(4)`	`[-2.124,...]`
50 tirages suivant une loi binomale $\mathscr{B}(n,p)$	`random.binomal(n,p,50)`	`[4,34,...]`
25 tirages suivant une loi géométrique $\mathscr{G}(p)$	`random.geometric(1/2,25)`	`[2, 5, ...]`
25 tirages suivant une loi de Poisson $\mathscr{P}(10)$	`random.geometric(1/2,25)`	`[ 6, 9, ...]`

• **Remarque :** pour une matrice aléatoire, on génère d'abord une liste que l'on `reshape` (voir section ?? page ??).

• **Exemple :** histogramme de 100 tirages `a=random.binomial(50,1/2,100)` par `hist(a)` puis `show()` :



3.2 Tracés 2d

```
from matplotlib.pyplot import *
```
[a]

• ⚠ les graphiques ne s'affichent qu'à la commande `show()`.

Commandes	Syntaxe
Généré une égende axe des x (ou y)	`xlabel("texte")`
Générer une grille	`grid()`
Générer une ligne brisée de sommets d'abscisse/ordonnées connues	`plot(list_abscisses,list_ordonnées,options)` voir plus bas pour les options de tracé
Générer un tracé de $y = f(x)$ sur x dans $[a, b]$ discrétisé à 20 points	`abscisses=linspace(a,b,20)` `plot(x,f(x),options)`
Réglages axes	`axis([xmin,xmax,ymin,ymax])`
Afficher les tracés	`show()`
Histogramme d'une liste `liste`	`hist(liste)`
Vider les tracés générés	`clf()` (*clear figure*)

• **Options de tracé :**

Options	syntaxe
Epaisseur ligne	`linewidth="4"` pour 4 pixels
Style tiret ou pointillés	`linestyle="dashed"` ou `"dotted"`
Couleur	`color="blue"`
Associer une légende	`label="Texte"` (faire `legend(loc=0)` avant pour un bon emplacement
Désactiver lignes	`linestyle="none"`
Activer marqueurs aux sommets	`marker="o",markersize=choix` autres marqueurs `*, x`

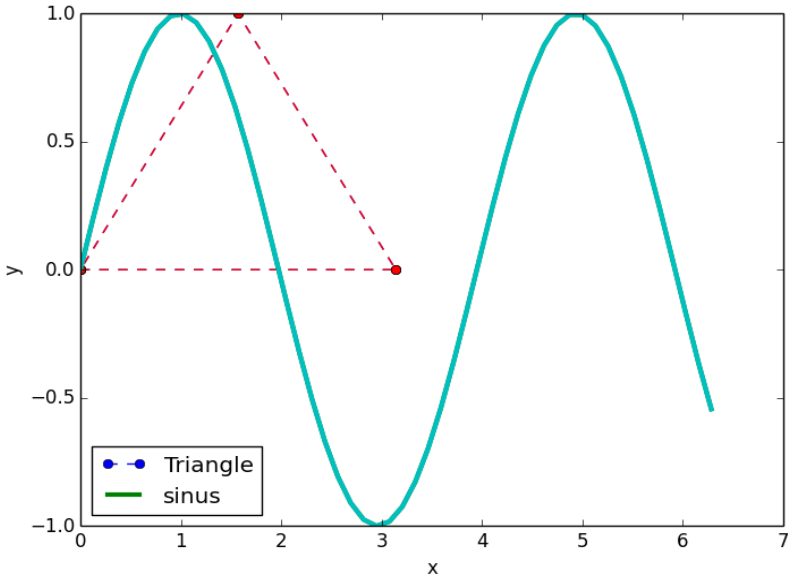
• Un petit exemple :

a. Des fonctions/types présentées dans ce document font partie de ce module qu'il faudra charger. La syntaxe présentée active automatiquement les fonctions/types. En cas de problème, on pourra toujours taper `import matplotlib.pyplot as pl`, `np` étant une abréviation arbitraire. Les fonctions/types associés devront impérativement être précédées du préfixe `pl`, par exemple `plot` deviendra `pl.plot`.

```
# -*- coding: utf-8 -*-

from numpy import *
from matplotlib.pyplot import *
f = lambda x : sin(x) # declaration fonction
##### BEL EMPLACEMENT DE LEGENDES
legend(loc=0)
##### LEGENDES SUR LES AXES
xlabel("x")
ylabel("y")
##### GENERATION D'UN TRIANGLE A(0,0),B(pi,0),C(pi/2,1)
absc=[0,pi,pi/2,0]
ordo=[0,0,1,0]
triangle=plot(absc,ordo,linestyle="dashed",marker="o",label="Triangle")
##### GENRATION y=f(x)
abs_discr=linspace(0,2*pi,50) # discretisation Ã  50 pts
ordonn=f(x) #gÃ©nÃ©ration ordonnees
graphe_sinus=plot(abs_discr,ordonn,linewidth="3",label="sinus")
##### ET MAINTENANT ON AFFICHE
show()
```

donne :



3.3 Autres tracés

On peut copier/coller/adapter des codes à partir de la page :
`http://matplotlib.org/gallery.html`

3.4 Mesurer le temps

- Tout vient du package `time` : `from time import *`
- Peu de choses à savoir :
  - (i) `a=clock()` : stocke dans `a` le temps indiqué par l'horloge interne ;
  - (ii) on écrit un [bloc] d'instructions ;
  - (iii) `b=clock()` : pareil ;

La différence `b-a` donne alors le temps qui s'est écoulé entre les lignes (i) et (iii) à l'exécution, soit le temps de calcul demandé par le [bloc] (ii)

3.5 Manipulation de fichiers

• Les réactions des commandes sont ici données avec un fichier `essai.txt` :

➢dont le chemin est `C:\ Users\ Florent \ Documents\ Python \ essai.txt` ;

➢constitué de deux lignes (avec un passage à la ligne)

```
------
Salut !
Ca va ?
------
```

Action	Entrée	Sortie
Charger en mode lecture	`truc=open("chemin","r")`	Stockage dans `truc` en mode "Read"
Lire la prochaine ligne	`fichier.readline()`	`"Salut !\n"` (un `\n` est un retour à la ligne)
Lire la `liste` de *toutes* les lignes	`fichier.readlines()`	`["Salut !\n","Ca va ?"]` (un `\n` est un retour à la ligne)
Charger fichier en mode écriture (⚠efface contenu)	`truc=open("chemin","w")`	Stockage dans `truc` en mode "Write"
Écrire dans `truc`	`truc.write("Youpi !")`	Rajoute `"Youpi !"` dans truc ⚠penser à refermer
Refermer/sauvegarder	`truc.close()`	Comme dit
Gérer répertoire courant	`import os`	Charge le module adéquat
Afficher répertoire courant	`os.getcwd()`	Comme dit
Changer répertoire courant	`os.chdir("C:\skynet\Document\")`	Déplace le répertoire courant à l'adresse indiquée
Sauver figure	`savefig("nom_fichier")`	Sauvegarde une figure dans le répertoire courant (format `.png`)

⚠il y a parfois un bug sur la donnée des chemins ; par exemple `"C:\Users\Documents\"` peut poser problème. L'explication est que le groupe `\ U` est une commande reconnue et automatiquement executée par l'éditeur/encodeur (qui bugge car elle est hors-propos). Le bug se résout en rajoutant un « r » au début du chemin, en l'ocurrence en changeant en :
`r"C:\Users\Documents\"`

3.6 Manipuler/Générer des images

• Le package utilisé est :
```
from matplotlib.image import *
from matplotlib.pyplot import *
```

• **Format des `array` affichables :** `Python` peut convertir des tableaux `tab` de format $(n, p, 3)$ en images à $n \times p$ pixels (repérage identique aux matrices en maths : n lignes numérotées en partant du haut, p colonnes de gauche à droite). Pour un tel tableau :

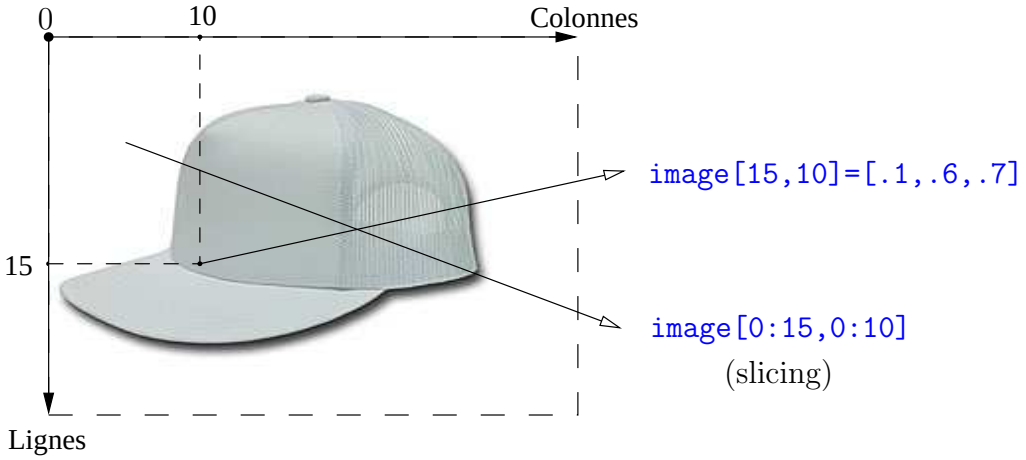
➢`tab[x,y]` est donc un `array` de type `[R,G,B]` avec `R,G,B` flottants dans $[0, 1]$ reflétant l'intensité du *Red, Green* ou *Blue* (couleurs primaires) au pixel (x, y). Par exemple `tab[x,y]=[0,0,0]` est un pixel noir, `tab[x,y]=[1,0,0]` est un pixel rouge.

➢pour générer un tableau de pixels noirs ($[R, G, B] = [0, 0, 0]$) à 100 lignes et 50 colonnes :

⇝ `tab= [ [0,0,0] for _ in range(100 * 50)]`

⇝ `tab=reshape(tab(100,50,3))`

➢pour afficher : `imshow(tableau,interpolation="nearest")`. Sans l'option interpolation, `Python` lisse les différences entre pixels ce qui peut nuire à la lisibilité.



➢**Images .png** : une image `casquette.png` située dans le répertoire courant [a] peut être pixellisée en un tableau au format précédent via la commande `imread("nom_fichier.png")`. On peut la manipuler, et sauvegarder le résultat dans un nouveau fichier.

Action	Entrée	Sortie
Générer tableau noir RGB	`tab= [ [0,0,0] for _ in range(100 * 50)]` `tab=reshape(tab(100,50,3))`	Cf. Ci dessus
Pixellisation d'un .png	`tableau=imread("casquette.png")`	`image` devient un `array`
Format image	`tableau.shape`	[nbre lignes,nbres colonnes, 3] 3 pour dosage RGB
Ajouter tableau à l'image	`imshow(tableau)`	Crée une image virtuelle
Afficher l'image	`show()`	Affiche
Sauvegarder tableau→.png	`imsave("casquette2.png",tableau)`	Sauvegarde rép. courant

3.7 Calcul scientifique (HP)

• Les commandes qui suivent sont HP mais peuvent être pratiques :

Action	Entrée	Sortie

Valeur approchée $\int_a^b f$	`> import scipy.integrate as spi`
	`> f = lambda x : exp(-x)`
	`> spi.quad(f,0,Infinity)`
		Tuple : Valeur approchée, Majoration de l'erreur

Résolutions Eq. diff.	COMING SOON	COMING SOON

4 Algorithmique

• Ceci *n'est pas* un précis d'algorithmique (il y aura un cours), juste un rappel de la rédaction globale des principaux éléments d'algorithmiques accompagnés de quelques exemples basiques.

4.1 Test `if`

• **Principe général** :

a. Sinon indiquer son chemin complet ou déplacer le répertoire courant (voir section 3.5 page 8).

```
if bool:
    #le test crée une indentation
    #->[instructions à éxécuter si bool==True]
#la fin de l'indentation marque la fin du test
else: # sinon (OPTIONNEL)
    #->[instructions à exécuter si bool==False]
#la fin de l'indentation marque la fin du else
```

Typiquement, `bool` est un booléen (càd une assertion vraie=`True` ou fausse=`False`) défini par une condition de type :

➢`a ==b` (test d'égalité [a]) entre deux éléments de même type ;

➢`a !=b` (test de différence) entre deux éléments de même type ;

➢`a in b` (test d'appartenance) pour savoir si `a` est un élément [b] de `b` ;

➢`a <=b` ou `a >=b` pour comparer deux éléments qui peuvent l'être ;

➢des liaisons `and` (et), `or` (ou), `not` (négation) entre une ou plusieurs des propositions précédentes.

• **Exemple :**

① **Exemples de booléens :**

```
>>> 1 in [1,2,3]
True
>>> 1 in [ [0,1],[1,2] ] #Attention les éléments de cette liste... sont des listes
False
>>> [0,1] in [ [0,1],[1,2] ]#ça va aller mieux
True
>>> "bon" in "bonjour" and not 4 in [1,2,3]
True
>>>
```

② **Exemples de test :**

```
>>> a=23
>>> if a%5==3: # rappel % = reste division entiers
        print('le reste de la division de '+str(a)+' par 5 est 3')
else:
        print('le reste de la division de '+str(a)+' par 5 est pas 3')

#cela donne :

le reste de la division de 23 par 5 est 3
```

4.2 Boucles

• On peut forcer la sortie d'un boucle à tout moment en s'offrant un `break`.

4.2.1 Boucles `for`

• **Principe général :**

a. L'égalité dans un test se rédige par DEUX symboles « = », pas naturel au début et source de beaucoup de plantages...

b. Pour les `str`, test si `a` est une sous-chaine de `b`, par exemple `"bon" in "bonjour"` est `True`.

```
for compteur in iterable:
    #les deux points creent une indentation
    #->[taper les instructions a exécuter pour chaque valeur de l'iterable]
# revenir sur l'indentation marque la fin de la boucle
```

① Typiquement, l'itérable est un ensemble d'entiers $[\![a\ ;\ b]\!]$=`range(a,b+1)` ou une liste.

② L'instruction `break`, si elle est rencontrée, provoque une sortie forcée de la boucle.

• **Exemple :**

① afficher a^2 pour a dans $[\![0\ ;\ 4]\!]$:

```
>>> for a in range(5):
        print(a**2)

0
1
4
9
16
```

Ici, le compteur est `a` qui décrit l'itérable est la liste `range(5)=[0,1,2,3,4]` de type `liste`.

② Compter[a] le nombre de "o" dans "bonjour" :

```
>>> mot,valeur='bonjour',0
>>> for lettre in mot:
        if lettre=='o':
            valeur+=1
>>> valeur
2
```

Ici, le compteur est `lettre` qui décrit l'itérable `mot="bonjour"` de type `str`.

4.2.2 Boucles `while`

• **Principe général :**

```
while bool :
    #les deux points creent une indentation
    #->[instructions à exécuter tant que bool est True]
    #ATTENTION s'assure que bool sera False au bout d'un moment...
    #...sinon boucle infinie et plantage.
```

• **Exemple :** pour une suite définie par $u_0 = 0$, on définit $u_{n+1} = u_n^2 + 1$ et on veut déterminer le plus petit n tel que $u_n \geqslant 50$.

```
>>> valeur,compteur=1,0
>>> while valeur<50:   #tant que [NEGATION DE CE QUE L'ON VEUT]
        compteur+=1
        valeur=valeur**2+1
        print('Rang '+str(compteur)+', suite= '+str(valeur))


Rang 1, suite= 2
Rang 2, suite= 5
Rang 3, suite= 26
Rang 4, suite= 677
```

a. Peut se faire directement avec la méthode `count`.

C'est la variable `valeur` qui prend les valeurs successives de la suite, à ne pas confondre avec la variable `compteur` qui joue ici le rôle de n.

4.2.3 Variants, invariants, complexité

• Lorsqu'on programme une boucle, il faut :

(i) S'assurer qu'elle se termine (pour les boucles `while`) ;

(ii) s'assurer qu'elle donne le bon résultat ;

(iii) évaluer le temps qu'elle va demander.

• **(i) : variant de boucle**. C'est une quantité **positive et strictement décroissante** au cours de la boucle, qui va donc forcer sa terminaison.

• **(ii) : invariant de boucle**. C'est, comme son nom l'indique, une quantité constante au cours de la boucle et dont on va pouvoir déduire le résultat.

• **(iii) : complexité** [a] **de la boucle**.

➢C'est le nombre d'*opérations élémentaires* rencontrées dans la boucle, càd :

⇝ d'assignations ;

⇝ de comparaisons ;

⇝ de calcul élémentaire ($+, \times$ ou évaluation d'une fonction arithmétique.

Ainsi, l'opération `if a<b: a=b**2+1` coûte : 1 comparaison + 1 assignation + 1 calcul de b^2 + 1 addition = 4 opérations élémentaires.

➢L'évaluation en temps se mesure en $O(\text{truc})$ (≪ grand O ≫), càd du type cte$\times$ truc sans chercher détailler la constante (on la majore par la pire situation).

• **Remarque :** on ne se préoccupe ici que de la complexité en temps (pas en espace mémoire).

• **Exemple :** étude de l'algorithme de division euclidienne.

```
1  # -*- coding: utf-8 -*-
2
3  def division(a,b):
4      assert a>=0 # provoque une sortie si a<0
5      assert b>0
6      q=0 # quotient
7      while a>=b:
8          a=a-b
9          q+=1
10     return q,a # a la fin de la boucle, a est le reste
```

➢**Variant de boucle :** `v=a-b`, qui est bien :

⇝ positif tant qu'on est dans la boucle (condition du `while`) ;

⇝ strictement décroissant dans la boucle car `a-b` → `a-2b` au cours d'un passage.

➢**Invariant de boucle :** on vérifie que :

`a+bq=cte` $\qquad (*)$

puisque si l'on a $(*)$ à la ligne 7, les modifications aux lignes 8 et 9 changent `a+bq` en `a-b+b(q+1)=a+bq` qui est donc conservé. Ceci permet de vérifier que le reste est bien `a` en sortie de boucle (ligne 10) puisque :

⇝ la cte vaut l'entrée de l'utilisateur à la ligne 6 (lorsque `q=0`) ;

a. La notion de complexité peut tout aussi bien s'évaluer pour un algorithme dans son ensemble.

$\rightsquigarrow$ en sortie de boucle `a<b` donc $(*)$ donne entrée=`bq+a` avec $|a| < b$: c'est bien la division euclidienne de l'entrée par b, a en est bien le reste.

➢**Complexité :**

$\rightsquigarrow$ 1 affectation ligne 6 ;

$\rightsquigarrow$ chaque passage dans la boucle coûte 1 comparaison, 2 affectations et 2 calculs (+ et -) soit 5 opérations élémentaires ;

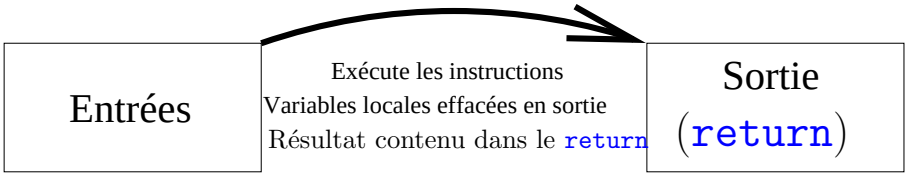
$\rightsquigarrow$ il faut évaluer le nombre de boucles, ici c'est facile, c'est la variable `q` (quotient), qui vaut la partie entière $E[a/b]$.
La complexité vaut donc $1+5E[a/b]$, pour un grand a/b (vu $E[x] \underset{+\infty}{\sim} x$) on a une complexité en $5a/b = O(a/b)$.

4.3 Fonctions `def`

• **Principe général :**

```
1 >>> def nom_fonction(entrees):
2         #les deux points créent une indentation
3         #->[instructions à exécuter]
4         #la procédure s'arrête au premier return
5         return(sortie)
```

Schématiquement, la fonction est la flèche dans le schéma suivant :



Les variables *autres que celles de l'entrée* intervenant dans la procédure sont des variables dites *locales* : utilisée de manière interne à la procédure, effacées à la sortie (on parle de portée *locale*). Si l'on veut conserver l'état d'une variable `x` (n'existant pas ailleurs dans le programme) après la sortie on la déclare comme *globale* dans la procédure par un `global x` situé juste après la première indentation.

• **Exemple :**

① Fonction `f` définie par :

➢Entrée : une variable `x` ;

➢Sortie : la valeur `x+1`.

```
1 >>> def f(x): #Ici l'entrée se résume à une variable x
2         return(x+1) #Ici la sortie se résume à une variable (contenant x+1)
3
4 >>> f(4)
5 5
```

• **Remarque :** peut aussi être défini par la lambda-fonction `f = lambda x : x+1`

② Fonction `test` définie par :

➢Entrée : une liste notée... `liste` ;

➢Sortie : un booléen `resultat` qui est `True` si la liste est réversible (càd se lit pareil dans les deux sens), `False` sinon.

```
>>> def test(liste):
        resultat=True
        longueur,indice=len(liste),0
        while resultat==True and indice<=len(liste)/2-1:
                if liste[indice]!=liste[longueur-1-indice]:
                        resultat=False
                indice+=1
        return resultat

>>> test([1,2,2,1]),test([1,2,3,1])
(True, False)
```

• **Remarque :** vu sa sortie, cette fonction est dite *booléenne* et peut intervenir dans un test, par exemple dans un « `if test(liste):` ».

③ Fonction `recherche` définie par :

➢Entrées : deux chaînes de caractères `mot` et `lettres` ;

➢Sortie : la liste (éventuellement vide) des indices i où la lettre i de `mot` vaut `lettre`.

```
>>> def recherche(mot,lettre): # ici deux entrées
        sites=[] #liste vide
        for indice in range(len(mot)): #pour parcourir tous les indices
                if mot[indice]==lettre:
                        sites.append(indice) #on rajoute l'indice à
                        #la liste des sites
        return(sites) #la procédure s'arrête, avec sortie = sites

>>> recherche('marseille','e')
[4, 8]
#oui il y a bien un e aux emplacements 4 et 8
```

4.4 Illustration du slicing

Exemple sur un tableau 6x6 :

```
>>> a[0,3:5]
array([3,4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,22,52])

>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```