I

Révisions 1 : Rappels élémentaires de programmation (structures conditionnelles et itératives, fonctions, preuve et complexité des algorithmes)

PLAN DU CHAPITRE

I Quelques rappels essentiels sur les structures de programmation

I.1 Structures conditionnelles : `if .... elif .... else`

Les structures conditionnelles sont des *séquences d'instructions* que Python réalise uniquement lorsqu'une condition est vérifiée. Ces mots clé sont *if, else, elif*.

Elles s'écrivent ainsi dans leur forme la plus complète :

Listing I.1 –

```
if condition 1 (expression booléenne):
  bloc d'instruction␣1
␣␣.......
␣␣.......
elif␣condition␣2␣(optionnel)␣(expression␣booléenne):
␣␣bloc␣d'instruction 2
  .......
  .......
elif condition 3 (optionnel) (expression booléenne):
  bloc d'instruction␣3
␣␣.......
␣␣.......
else:
␣␣bloc␣d'instruction 4
  .......
  .......
```

La condition à vérifier est souvent formulée avec un opérateur de comparaison renvoyant le résultat booléen False ou True :

égal à	⇔	==
différent de	⇔	!=
plus grand que	⇔	>
plus petit que	⇔	<
supérieur ou égal à	⇔	>=
inférieur ou égal à	⇔	<=

Listing I.2 –

```
# Donne le signe d'un entier relatif
print "Entrer␣un␣nombre␣entier␣relatif:"
n=input()
if type(n)!=int: #vérifie si n est entier avec l'opérateur booléen "!=" différent de
  print("n␣n'est␣pas␣un␣entier")
elif n==0: #sinon vérifie si n est nul avec l'opérateur booléen "==" égal à
  print ("n␣est␣l'entier␣nul")
elif n>0:
  print ("n␣est␣entier␣positif")
else: # sinon
  print ("n␣est␣négatif")
print"Fin␣d'execution"
```

Ce qui donne par exemple pour n=2 :

```
Entrer un nombre entier relatif :
2
n est un entier positif
Fin d'execution
```

et pour n=-2 :

```
Entrer un nombre entier relatif :
-2
n est un entier négatif
Fin d'execution
```

NB : les conditions après un *elif* sont testées uniquement si les conditions antérieures n'ont pas été vérifiées.

I.2 Structures itératives

a - Boucles énumérées

Les boucles énumérées sont des structures itératives pour lesquelles le nombre d'itérations est préalablement fixé ; le nombre d'itérations est implémenté à l'aide d'un index apparent ou pas suivant l'instruction employée.

- Commande `range` :
  La commande `range` permet d'énumérer des entiers en précisant le premier (optionnel), le dernier-1, et le pas d'incrément (optionnel). Ses syntaxes possibles sont les suivantes :

```
>>> list(range(10))
[0,1,2,3,4,5,6,7,8,9]
```

```
>>> list(range(5,10))
[5,6,7,8,9]
```

```
>>> list(range(5,15,2))
[5,7,9,11,13]
```

- Commande de boucle indexée `for` : La commande for permet de lancer une boucle avec un index ou compteur ; ses syntaxes sont les suivantes :

```
>>> for x in range(2,10,3) :
... : print(x,x**2)
2 4
5 25
8 64
```

On peut également imbriquer les boucles :

```
for x in range(1, 6) :
... :for y in range(1, 6) :
... : print(x * y, end=' ')
... :print('/')
1 2 3 4 5 /
2 4 6 8 10 /
3 6 9 12 15 /
4 8 12 16 20 /
5 10 15 20 25 /
```

- Commande de boucle `for` pour itérer sur une chaîne de caractère :
  Afin d'éviter l'introduction d'un index de boucle (qui existe néanmoins de façon masquée), la commande `fort` permet également d'itérer sur une chaîne de caractère :

Listing I.3 –

```
1 def iter_carac(mot):
2   for c in mot:
3     print(c,end='␣'*2)
4 iter_carac("informatique-de-tronc-commun")
```

```
i n f o r m a t i q u e - d e - t r o n c - c o m m u n
```

- Commande de boucle `for` pour itérer sur une chaîne de caractère avec énumération de l'index :
  On peut également balayer une structure itérable (comme ci-dessus avec les chaînes de caractères) en extrayant également son index dans la structure à l'aide de la commande `enumerate` :

```
for (i,c) in enumerate("informatique-de-tronc-commun") :
... : print(i,c,sep="->",end=' ')
0->i 1->n 2->f 3->o 4->r 5->m 6->a 7->t 8->i 9->q 10->u 11->e 12->- 13->d 14->e 15->- 16->t
17->r 18->o 19->n 20->c 21->- 22->c 23->o 24->m 25->m 26->u 27->n
```

b - Boucles conditionnelles : commande `while`

La commande *while* permet de réaliser une série d'instructions de manière itérative dans une boucle dite **conditionnelle** ; la série d'instructions sera réalisée tant que la condition énoncée sera vérifiée ; on notera deux choses importantes :

- la condition peut ne jamais être réalisée, et dans ces conditions la boucle n'est jamais exécutée.
- si l'instruction booléenne à évaluer comporte une variable, **celle-ci devra être initialisée avant l'instruction `while`**

L'exemple ci-dessous, qui réinvestit un test *if*, calcule la factorielle de n à l'aide d'une boucle *while* :

Listing I.4 –

```
1 print"Entrer␣la␣valeur␣d'un␣entier␣positif"
2 n=input()
3 if type(n)!=int: # Teste si n n'est pas un entier ("vrai" si le type de n n'est pas 'int'
```

```
4     print("vous␣n'avez␣pas␣entré␣un␣entier")
5   else: #
6     fact=1
7     while n>0: #teste la condition sur n et stoppe la boucle dès que n=0
8       fact=fact*n
9       n=n-1
10    print(fact)
```

Un autre exemple plus amusant qui donne l'heure courante tant que la demande de sortie du programme n'est pas ordonnée par l'utilisateur :

Listing I.5 – L'horloge non parlante!!!

```
1 import time      # importation du module time
2 quitter = 'n'    # initialisation
3 while quitter != 'o':
4   # ce bloc est exécuté tant que la condition est vraie
5   # strftime() est une fonction du module time
6   print('Heure␣courante␣',time.strftime('%H:%M:%S'))
7   quitter=input('Voulez-vous␣quitter␣ce␣programme␣(o/n)')
8 print("A␣bientôt")
```

I.3 Fonctions

a - Principe de définition

Lorsqu'une série d'instructions **visant à renvoyer un résultat** est destinée à être reproduite de nombreuses fois dans un programme, il peut être utile de définir cette suite d'instructions **comme une fonction** à laquelle on pourra faire appel à tout moment. Une fonction est désignée par une étiquette et reçoit un certain nombre d'arguments/paramètres. Son définition dans un programme se fait à l'aide du mot clé `def`. Par exemple, le script suivant calcule la factorielle de n rentrée au clavier

Listing I.6 –

```
1 def factorielle(n):
2   if type(n)!=int:
3     print ("n␣n'est␣pas␣un␣entier!!!")
4   else:
5     f = 1
6   for i in range(1, n+1):
7     f *= i
8   return(f) #renvoie le résultat de la fonction
9
10 ##### Procédure d'appel à la fonction #####
11 for i in range(5):
12   print(factorielle(i+1)) #+1 car décalage d'indice dans une liste générée par range
```

```
1
2
6
24
120
```

Il est également possible de transmettre ses arguments à une fonction sans que le nombre de ces derniers soit nécessairement défini. On ajoute alors "*" devant le nom générique choisi pour les arguments.

La fonction qui suit calcule par exemple la somme alternée d'une série d'arguments numériques entiers transmis en nombre quelconque :

Listing I.7 –

```
def sommeAlt(*args):
  s=0
  sg=1
  badtest=0
  for i in args:
    if type(i)!=int:
      print ("Attention : au moins un de vos arguments n'est pas un entier!")
      badtest=1
    else:
      s+=sg*i
      sg=-sg
  if badtest==0:
    return s
######## Appels à la fonction #########
print(sommeAlt(1,2,3),sommeAlt(4,5,6))
```

```
2 5
```

Remarque I-1: OMISSION D'ARGUMENTS - ARGUMENTS PAR DÉFAUT

Une tentative d'appel à une fonction avec un nombre d'arguments ne correspondant pas à celui figurant dans sa définition conduit à une erreur :

b - Portée des variables

- Cas simple: variables locales et globales dans les fonctions

On appelle **portée des variables** le domaine de visibilité de ces dernières au sein de la définition d'une fonction. On distingue les variable locales à une fonction des variables globales.

Les exemples suivants illustre cela :

Listing I.8 –

```
x = 42
def f():
  return x #renvoie la valeur de x variable globale soit 42
def g():
  x = 3
  return x #renvoie la valeur de x locale car définie dans le programme soit 3
def h():
  global x
  x = 17
  return x #renvoie la valeur de x globale modifiée dans le programme
```

Ce qui donne la sortie suivante :

```
>>> f()
42
>>> g()
3
>>> x
42
>>> h()
17
>>> x
17
```

- Cas plus "fin": variables locales dans les sous-fonctions

Une sous-fonction est une fonction définie à l'intérieur même d'une autre fonction.
A RETENIR :

Propriété I-1:

- une sous-fonction n'est pas appelable en dehors de la fonction dans laquelle elle est définie.
- les variables locales de la sous-fonction ne sont pas visibles de l'extérieur, donc ni de la fonction dans laquelle elle est définie, et fatalement pas non plus de l'extérieur des deux fonctions.
- comme toujours, les variables définies comme globales sont visibles de partout.

Listing I.9 – Variable locale

```
### Définitions fonction et sous-fonction ###
def fonction1():
  x=1
  def fonction2():
    x=2
    print (3*"␣" + "niveau␣2,␣on␣a␣x=",x)
    return None
  ### Appel à la sous-fonction ###
  fonction2()
  print(1*'-' + "niveau␣1␣(après␣appel␣à␣fonctio
  return None
```

Listing I.10 – Variable globale

```
### Définitions fonction et sous-fonction ###
def fonction1():
  global x
  x=1
  def fonction2():
    global x
    x=2
    print (3*'-'+'niveau␣2,␣on␣a␣x=',x)
    return None
  ### Appel à la sous-fonction ###
  fonction2()
  print (1*'-'+'niveau␣1␣(après␣appel␣à␣fonction2),␣x=',
  return None
```

```
>>> fonction1()
---niveau 2, on a x= 2
-niveau 1 (après appel à fonction2), x= 1
```

```
>>> fonction1()
---niveau 2, on a x=2
-niveau 1 (après appel à fonction2), on a x=2
```

II Preuve des algorithmes

II.1 Définition

Définition II-1: PREUVE

On appelle preuve d'un algorithme, la propriété qui assure à ce dernier :
- de terminer. On appelle cela la TERMINAISON de l'algorithme
- de réaliser ce qu'on attend de lui. On appelle cela la CORRECTION de l'algorithme.

II.2 Comment «prouver» ?

a - Terminaison

Il est fréquent dans l'établissement d'un algorithme qu'un programmeur ait recours à une structure de boucle. Lorsque cette dernière est **conditionnelle** (while), et que l'algorithme exécute une première fois les instructions contenues dans la boucle dans le cas où la condition de boucle est vérifée, il est important de s'assurer que **l'algorithme sortira de la boucle et terminera.** Cette propriété de l'algorithme s'appelle **la terminaison**

Ainsi :

le groupe d'instructions de la boucle doit permettre une modification de la condition de boucle.

Propriété II-1: TERMINAISON

On assure la **terminaison** d'un algorithme lorsque toutes les structures de boucles conditionnelles de celui-ci «terminent».

b - Correction

En outre, la seconde préoccupation du programmeur sera d'assurer que son algorithme réalise bien le travail demandé. Cette propriété de l'algorithme s'appelle **la correction**. Elle est généralement plus délicate à établir, et repose sur la notion d'invariant de boucle.

Propriété II-2: CORRECTION MONTRÉE PAR INVARIANT DE BOUCLE

On assure la **correction** d'un algorithme avec boucle en dégageant une propriété vérifiée avant l'entrée dans la boucle et qui le restera durant chaque itération i de boucle ; soit $\mathscr{P}_i$ cette propriété au rang i. Cette propriété doit permettre de renvoyer le résultat attendu au dernier rang de boucle. On l'appelle **l'invariant de boucle**. La correction est généralement démontrée **par récurrence**.

Définition II-2: INVARIANT DE BOUCLE

On appelle invariant de boucle une propriété vraie avant l'exécution de boucle et qui le restera à chaque itération.

II.3 Quelques exemples

a - Factorielle

On considère le script python suivant :

Listing I.11 – Factorielle de n

```
n=input()
if type(n)==int and n>=0:
  k=1
  f=1
  while k<=n :
    f=f*k
    k=k+1
  print (f)
else:
  vprint ("Impossible")
```

- TERMINAISON :
  - ◇ Si n entré au clavier est négatif, le programme termine sur un message ("impossible").
  - ◇ Si n est nul la boucle n'est pas exécutée, 1 est renvoyé et le programme termine.
  - ◇ Si $n>0$ k étant initialement à 1, la boucle est exécutée. A chaque itération, k est incrémenté de 1 et finit par être supérieur à n donc pour k=n+1, on sort de la boucle,le programme renvoie f, et termine.

  CONCLUSION : la terminaison est assurée.

- CORRECTION :
  Un invariant de boucle $\mathscr{P}_i$ est par exemple :

  « *après la* $i^{\text{ième}}$ *itération* k *contient* $i+1$ *et* f *contient* $i!$ »

  Cette propriété est vraie au rang 0. Supposons la vraie au rang i, et montrons qu'elle est héréditaire :

  - ◇ au rang $i+1$, on a : k qui contient $i+1$ en début d'itération et $f = i! \times (i+1) = (i+1)!$
  - ◇ en fin d'itération k contient $i+2$

  Ceci est bien la propriété au rang $i+1$

  CONCLUSION : la correction est assurée.

b - Puissance de 2

On considère le code python calculant la puissance $n^{\text{ième}}$ de 2 :

Listing I.12 – Puissance de 2

```
n=input()
if type(n)==int and n>=0 :
p=1
while n>0 :
p=2*p
n=n-1
print p
else :
print "Impossible"
```

- TERMINAISON :
  - ◇ Si n entré au clavier n'est pas un entier positif ou nul le programme termine sur un message ("impossible").
  - ◇ Si n est nul la boucle n'est pas exécutée, 1 est renvoyé et le programme termine.
  - ◇ Si $n > 0$, la boucle est exécutée. A chaque itération, n est décrémenté de 1 et finit par être nul, on sort de la boucle, le programme renvoie p, et termine.

  CONCLUSION : la terminaison est assurée.

- CORRECTION :
  Un invariant de boucle est par exemple : «*après la* $i^{ième}$ *itération* p *contient* $2^{n_0-(n_0-i)} = 2^i$ *et* n *contient* $n_i = n_0 - i$»
  Les conditions initiales assure qu'au rang 0 la propriété est vraie. Supposons la vraie au rang i, et montrons qu'elle est héréditaire :
  - ◇ au rang $i+1$, on a : p qui contient $2 \times 2^{n_0-(n_0-i)} = 2^{n_0-(n_0-(i+1))} = 2^{i+1}$
  - ◇ en fin d'itération n contient $n_{i+1} = n_0 - (i+1)$

  Ceci est bien la propriété au rang $i+1$

  CONCLUSION : la correction est assurée.

III Complexité des algorithmes

III.1 Outils et notations

Si le programmeur veille à la preuve de son algorithme, il est également soucieux d'optimiser les performances de ce dernier afin de limiter l'occupation du microprocesseur. Toute opération ordonnée par l'algorithme au microprocesseur représente un coût en terme de **temps d'occupation de ce dernier**.

Le coût total cumulé une fois l'algorithme terminé est appelé **complexité temporelle**.

Par ailleurs, le fonctionnement d'un algorithme en machine occupe également de la mémoire.

Le coût total cumulé en occupation mémoire est appelé **complexité spatiale**.

Définition III-1: COMPLEXITÉ D'UN ALGORITHME

Soient (f_n) et (g_n) deux suites de réels positifs.

- (f_n) est dite **minorée** par (g_n) si et seulement si :

$$\exists N \in \mathbb{N} \quad \exists \lambda > 0 \quad \forall n \geqslant N \quad \lambda g_n \leqslant f_n$$

On note $f_n = \Omega(g_n)$ et on lit souvent : « f_n est *un* grand omega de g_n ».

- (f_n) est dite **majorée** par (g_n) si et seulement si :

$$\exists N \in \mathbb{N} \quad \exists \mu > 0 \quad \forall n \geqslant N \quad f_n \leqslant \mu g_n$$

On note $f_n = O(g_n)$ et on lit souvent : « f_n est *un* grand O de g_n ».

- (f_n) et (g_n) sont dites **du même ordre** si et seulement si :

$$f_n = \Omega(g_n) \text{ et } f_n = O(g_n)$$

soit :

$$\exists N \in \mathbb{N} \quad \exists \lambda > 0 \quad \exists \mu > 0 \quad \forall n \geqslant N \quad \lambda g_n \leqslant f_n \leqslant \mu g_n$$

On note $f_n = \Theta(g_n)$ et on lit souvent : « f_n est *un* grand theta de g_n ».

De manière très schématique, la notation O permet d'évaluer la **complexité dans le pire des cas** ; la notation Θ la **complexité «en gros»**.

EXEMPLE :

Listing I.13 – Boucles imbriquées

```
1 from numpy import *
2 n = 3
3 A = empty((n,n))
4 for i in range(n):
5   for j in range(i):
6     A[i,j] = i + j
```

Dans cet exemple, on exécute n itérations de la première boucle, et pour chaque itération de celle-ci au rang i, la seconde s'execute i fois pour remplir la matrice A.

Ainsi, le nombre total d'itérations de la structure imbriquée est :

$$f(n) = (1+2+3+\ldots+n) = n \times \frac{1+n}{2}$$

Notons $g(n) = n^2$. Il est possible de trouver deux constantes positives λ et μ telles que :

$$\lambda g(n) \leqslant f(n) \leqslant \mu g(n)$$

Par exemple, $\lambda = 1/2$ et $\mu = 1$ conviennent. On établit ainsi :

$$f(n) = O(n^2) \qquad f(n) = \Omega(n^2) \qquad f(n) = \Theta(n^2)$$

La notation $O(n^2)$ peut s'interpréter en terme de **complexité asymptotique** : lorsque n devient grand, $f(n)$ est de l'ordre de n^2. L'algorithme précédent est $O(n^2)$; son temps d'exécution n'excède pas un certain μn^2, avec $\mu > 0$.

III.2 Classification

Ces outils de comparaison permettent de classer les algorithmes selon leur complexité [1] :

- **complexité constante** en $O(1)$;
- **complexité logarithmique** en $O(\log_2 n)$;
- **complexité linéaire** en $O(n)$;
- **complexité quasi-linéaire** en $O(n \log_2 n)$;
- **complexité polynomiale** en $O(n^k)$;
- **complexité exponentielle** en $O(2^n)$;

Le tableau suivant compare ces complexités pour des tailles n de données croissantes.

n	10^2	10^3	10^4
$\ln n$	$4,6$	$6,9$	$9,2$
$n \ln n$	461	$6,9 \times 10^3$	$9,2 \times 10^4$
n^2	10^4	10^6	10^8
2^n	$> 10^{30}$	$> 10^{300}$	$> 10^{3000}$

III.3 Exemples

a - Valeur moyenne

Le script python suivant définit une fonction qui calcule la valeur moyenne des éléments de la liste *uneListe* donnée en argument :

Listing I.14 – Valeur moyenne des éléments d'une liste

```
1 def moyenne(uneListe) :
2 """Calcule de la moyenne des éléments d'une liste de nombres passée en argument"""
3   # calcul de la somme des éléments de la liste
4   somme=0.     # initialisation
5   for elt in uneListe :     # boucle sur les éléments de la liste
6     somme=somme+elt     # ajout de l'élément courant
7   # division de la somme par le nombre de termes
8   return somme/len(uneListe)
```

En toute rigueur, le calcul de complexité doit également tenir compte des opérations d'affectation qui représentent un coût temporel pour le processeur. On recense donc :

- 1 affectation $somme = 0$, soit 1 opération
- 1 affectation et une addition pour chaque itération, soit au total $2n$ opérations

1. Le logarithme en base 2, noté $\log_2$, apparait régulièrement dans les calculs de complexité. C'est pourquoi les résultats sont exprimés en ses termes.

- 1 division pour le calcul final de la moyenne soit 1 opération

Le coût total en opération est donc : $f(n) = 2n+2$

Posons la fonction $g(n) = n$. On peut trouver (λ, μ) tel que $\lambda g(n) < f(n) < \mu g(n)$; par exemple le couple $(\lambda = 2, \mu = 4)$

Ainsi : $\begin{cases} \text{la complexité est «en gros» est } \Theta(n) \\ \text{la complexité «au pire» est } O(n) \end{cases}$

b - Tri «bulle»

PRINCIPE : étant donnée une liste S d'éléments, on cherche à renvoyée la liste triée de ces éléments en faisant "remonter" en surface les éléments les plus grands, d'où l'appellation de `tri-bulle`.

L'algorithme en langage naturel est le suivant :

```
pour i de n à 2, faire:
   pour j de 1 à i-1, faire:
      si S[j]>S[j+1] faire:
         permutation S[j] et S[j+1] dans la liste S
```

ce qui donne en Python :

Listing I.15 –

```
1 def bulle(L):
2   for i in range(len(L)-1,0,-1):
3     for j in range(0,i):
4       if L[j]>L[j+1]:
5         L[j],L[j+1]=L[j+1],L[j]
6       print(L)
7 liste=[5,9,1,3,2,85,45,34]
8 bulle(liste)
```

NB : ce script inscrit l'état de la liste à chaque itération permettant de visualiser l'effet "bulle" : remontée du plus grand en fin de liste.

On recense :

- une première boucle procédant à $n-1$ itérations
- une seconde boucle procédant à $i-1$ itérations, i étant le rang de la première
- Une permutation correspondant à deux opérations élémentaires (intervention d'une troisième adresse mémoire intermédiaire, non visible ici)

Soit finalement en nombre d'opérations :

$$f(n) = (1+2+3+\ldots+(n-1)) \times \underbrace{C_{perm}}_{=2} = 2 \times \underbrace{(n-1)}_{nb\ termes} \times \frac{1+n-1}{2} = n(n-1)$$

On pose $g(n) = n^2$, ainsi :
Pour $n \geq 2$, on a : $\lambda g(n) < f(n) = \mu g(n)$ avec le couple $(\lambda = 0.5, \mu = 1)$

Ainsi : $\begin{cases} \text{la complexité est «en gros» est } \Theta(n^2) \\ \text{la complexité «au pire» est } O(n^2) \end{cases}$