Révisions 1: Rappels élémentaires de programmation

(Révisions MPSI)

JL GRAYE

MP3 Lycée Montaigne

Plan

Variables informatiques : manipulation de base

Une variable informatique est l'élément de base de stockage des informations dans un programme informatique. Elle est généralement désigné par une lettre (au minimum) **a,b,c,x,y etc...**, ou un assemblage de plusieurs lettres et chiffres **ab,bf0, xy, var6, essai etc....**
Lorsque l'on stocke une valeur dans une variable, cette dernière désigne en fait une adresse dans la mémoire dans laquelle est stockée la valeur. On appelle cela un pointeur.
Pour connaître la correspondance entre le nom choisi de la variable et l'adresse mémoire correspondant à la valeur stockée, Python exploite une table d'allocation mémoire.

Variables informatiques : manipulation de base

En Python, l'affectation d'une valeur à un nom de variable se fait par le signe "=", et l'interrogation de la valeur pour affichage par simple rappel du nom de la variable :

Variables informatiques : manipulation de base

En Python, l'affectation d'une valeur à un nom de variable se fait par le signe "=", et l'interrogation de la valeur pour affichage par simple rappel du nom de la variable :

```
>>> x = 9
>>> x
9
```

Variables informatiques : manipulation de base

En Python, l'affectation d'une valeur à un nom de variable se fait par le signe "=", et l'interrogation de la valeur pour affichage par simple rappel du nom de la variable :

```
>>> x = 9
>>> x
9
```

Affectation simultanée :

```
>>> x,a = 9,3
>>> x
9
>>> a
3
```

Variables informatiques : manipulation de base

Echange de deux variables :

Variables informatiques : manipulation de base

Echange de deux variables :

```
>>> x,a = a,x
>>> x
3
>>> a
9
```

Variables informatiques : manipulation de base

Echange de deux variables :

```
>>> x,a = a,x
>>> x
3
>>> a
9
```

QUESTION : Comment fait-on la même chose dans un script, c'est à dire un programme Python ?

Variables informatiques : manipulation de base

RÉPONSE : Seul l'affichage se fait par une nouvelle commande : *print*

```
1 a,b = 9,3
2 print "La variable a contient :",a,"et la variable b contient :
      ",b
3 a,b=b,a
4 print "La variable a contient :",a,"et la variable b contient :
      ",b
```

qui donne la sortie suivante :

Variables informatiques : manipulation de base

RÉPONSE : Seul l'affichage se fait par une nouvelle commande : *print*

```
1 a , b = 9 , 3
2 print "La␣variable␣a␣contient :" , a , "␣et␣la␣variable␣b␣contient :
      " , b
3 a , b=b , a
4 print "La␣variable␣a␣contient :" , a , "␣et␣la␣variable␣b␣contient :
      " , b
```

qui donne la sortie suivante :

```
La variable a contient : 9 et la variable b contient : 3
La variable a contient : 3 et la variable b contient : 9
```

Variables informatiques : manipulation de base

Exercice n°1:

Jouons un peu avec l'addition ! Interpréter les deux scripts suivants "à la main" et expliquer leur rôle :

Le premier script :

```
1 a,b = 9,3
2 print "La variable a contient :",a,"et la variable b contient :",b
3 c=a
4 a=b
5 b=c
6 print "La variable a contient :",a,"et la variable b contient :",b
```

Variables informatiques : manipulation de base

... et le second :

```
1 a,b = 9,3
2 print "La variable a contient :",a,"et la variable b contient :",b
3 a=a+b
4 b=a-b
5 a=a-b
6 print "La variable a contient :",a,"et la variable b contient :",b
```

Intérêt du second script : économie de mémoire car occupation de celle-ci par 2 variables seulement a, b et non 3 a, b, c !

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

égal à	$\Leftrightarrow$	==
différent de	$\Leftrightarrow$	!=
plus grand que	$\Leftrightarrow$	>
plus petit que	$\Leftrightarrow$	<
supérieur ou égal	$\Leftrightarrow$	>=
inférieur ou égal	$\Leftrightarrow$	<=

Par exemple ...

Structures conditionnelles : *if, elif, else*

```
1  # Donne le signe d'un entier relatif
2  print "Entrer un nombre entier relatif :"
3  n=input()
4  if type(n)!=int:
5      print "n n'est pas un entier"
6  elif n==0:
7      print "n est l'entier nul"
8  elif n>0:
9      print "n est entier positif"
10 else:
11     print"n est negatif"
12 print"Fin d'execution"
```

ATTENTION : après une instruction *if* ou *else* ou *elif* on écrit un caractère **" :"** et le bloc d'instructions est écrit après une **indentation** (touche *tabulation*).

Ce qui donne par exemple pour n=2 :

Structures conditionnelles : *if, elif, else*

```
1  # Donne le signe d'un entier relatif
2  print "Entrer un nombre entier relatif :"
3  n=input()
4  if type(n)!=int:
5      print "n n'est pas un entier"
6  elif n==0:
7      print "n est l'entier nul"
8  elif n>0:
9      print "n est entier positif"
10 else:
11     print"n est negatif"
12 print"Fin d'execution"
```

ATTENTION : après une instruction *if* ou *else* ou *elif* on écrit un caractère **" :"** et le bloc d'instructions est écrit après une **indentation** (touche *tabulation*).

Ce qui donne par exemple pour n=2 :

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

Structures conditionnelles : *if, elif, else*

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.

Boucle conditionnelle *while*

La commande *while* permet de réaliser une série d'instructions de manière itérative dans une boucle dite **conditionnelle**, c'est à dire tant qu'une condition est vérifiée. L'exemple ci-dessous, qui réinvestit un test *if* calcule la factorielle de n à l'aide d'une boucle *while* :

```
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 entrÃ© un entier"
5  else:
6      fact=1
7      while n>0: #teste la condition sur n et stoppe la boucle
       des que n=0
8          fact=fact*n
9          n=n-1
10     print(fact)
```

Boucle conditionnelle *while*

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. on fait appel ici au module time dont la *méthode* strftime lit l'heure et la convertit en chaine de caractère affichable :

```
1 import time      # importation du module time
2 quitter = 'n'    # initialisation
3 while quitter != 'o':
4     # ce bloc est exÃ©cutÃ© tant que la condition est vraie
5     # strftime() est une methode du module time
6     print('Heure courante ',time.strftime('%H:%M:%S'))
7     quitter = input("Voulez-vous quitter le programme (o/n) ?
      ")
8 print("A bientÃ´t")
```

Boucle inconditionnelle *for*

Préliminaire : commande *range* et les listes

La commande *range* permet de générer une **liste** de nombres. Elle possède plusieurs syntaxes possibles. Par exemple, pour générer une liste composée des dix premiers entiers naturels, on peut écrire :

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

On remarque que dans la seconde syntaxe proposée, le second argument (10) est encore exclu de la liste.

Boucle inconditionnelle *for*

Préliminaire : commande *range* et les listes

On peut également compter à rebourd, **et nous avons toujours ce souci avec le dernier nombre de la liste** :

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

Boucle inconditionnelle *for*

Commande *for*

La commande *for* permet de réaliser une série d'instructions de manière itérative dans une boucle inconditionnelle un nombre de fois déterminé. Le compteur de boucle prend ces valeurs successives dans une liste indiquée comme argument selon la syntaxe suivante :

```
1 print "Entrer␣un␣nombre␣entier:"
2 n=input()
3 L=range(1,n+1) #On remarque la nÃ©cessitÃ© de mettre n+1 en
      second argument pour parcourir tous les entiers jusqu'Ã  n
       et non pas n-1
4 fact=1
5 for i in L:
6     fact*=i #Formulation "compacte" du calcul et de l'
      affectation; vaut pour fact=fact*i
7 print(fact)
```

Boucle inconditionnelle *for*

Commande *for*

Un autre exemple simple est l'algorithme de chiffrement élémentaire dit Code de César qui consiste à décaler de n rangs d'alphabet toutes les lettres d'un mot. n constitue ce que l'on appelle la clé. Cryptons par exemple le mot **"bonjour"**en choisissant $n = 3$:

```
1 def cesar(ch):
2     alphabet="abcdefghijklmnopqrstuvwxyz"
3     res=""
4     for c in ch:
5         for i in range(26):
6             if c==alphabet[i]:
7                 res=res+alphabet[(i+3)%26] #permet une
    rotation circulaire sur les 26 lettres (ie dÃ©calage de 3
    modulo 26)
8     return res
9 print cesar("bonjour")
```

Boucle inconditionnelle *for*

Commande *for*

Exercice n°1:

On donne une liste de nombres stockés dans la variable liste.
Ecrire un programme permettant :

- *d'extraire de* liste *les éléments positifs et les classer par ordre croissant dans une nouvelle liste* listep
- *d'afficher cette liste une fois réalisée et d'indiquer le nombre de ces éléments.*
- *d'indiquer si la liste est vide le cas échéant*

Boucle inconditionnelle *for*

Commande *for*

Solution :

```
liste=[5,-9,-4,-6,-75,-8,4,-48,2]
listep=[]
#Tri des positifs
for x in liste:
    if x>=0:
        listep=listep+[x]
print("La liste des entiers positifs est:"), listep
# Tri par ordre croissant
for n in range(len(listep)):
    for m in range(n+1,len(listep)):
        if listep[m]<listep[n]:
            interm=listep[m]
            listep[m]=listep[n]
            listep[n]=interm
print (u"La liste triÃ©e est:"),listep
print ("longueur de la liste:"),len(listep)
if len(listep)==0:
    print("La liste des entiers positifs est vide")
```

Fonctions

Principe de définition

On peut être amené dans un programme à faire appel plusieurs fois à la même suite d'instructions. Il est alors possible de définir une procédure désignée par un nom pour executer ce bloc d'instructions. Cet objet porte le nom impropre de **fonction** même si l'appellation **procédure** lui conviendrait mieux.

Fonctions

Principe de définition

Définissons par exemple une fonction permettant de calculer la factorielle d'un entier entré au clavier :

```
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 rÃ©sultat de la fonction
9  for i in range(5):
10     print (factorielle(i+1)) #+1 toujours en raison de ce
       dÃ©calage d'indice dans une liste gÃ©nÃ©rÃ©e par range
```

Qui donnera la sortie suivante ...

Fonctions

Principe de définition

Fonctions

Principe de définition

```
1
2
6
24
120
```

Fonctions

Principe de définition

Nous pouvons également 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 :

Fonctions

Principe de définition

```
1  def sommeAlt(*args):
2      s=0
3      sg=1
4      badtest=0
5      for i in args:
6          if type(i)!=int:
7              print"Attention : au moins un de vos arguments n'
       est pas un entier !"
8              badtest=1
9          else:
10             s+=sg*i
11             sg=-sg
12     if badtest==0:
13         return s
14 print sommeAlt(1,2,3),sommeAlt(4,5,6)
```

La sortie est la suivante :

Fonctions

Principe de définition

```
1  def sommeAlt(*args):
2      s=0
3      sg=1
4      badtest=0
5      for i in args:
6          if type(i)!=int:
7              print"Attention : au moins un de vos arguments n'
       est pas un entier !"
8              badtest=1
9          else:
10             s+=sg*i
11             sg=-sg
12     if badtest==0:
13         return s
14 print sommeAlt(1,2,3),sommeAlt(4,5,6)
```

La sortie est la suivante :

```
2 5
```

Fonctions

Portée des variables

On appelle **portée des variables** le domaine de visibilité de ces dernières au sein de la définition d'une fonction. On distingue les variables locales à une fonction des variables globales. On illustre cela avec les exemples suivants :

```
1  x = 42
2  def f():
3      return x #renvoie la valeur de x variable globale soit 42
4  def g():
5      x = 3
6      return x #renvoie la valeur de x locale car dÃ©finie dans
       le programme soit 3
7  def h():
8      global x
9      x = 17
10     return x #renvoie la valeur de x globale modifiÃ©e dans le
        programme
```

Fonctions

Portée des variables

Ce qui donne la sortie suivante :

Fonctions

Portée des variables

Ce qui donne la sortie suivante :

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

Fonctions

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

Une sous-procédure ou sous-fonction consiste en une procédure ou fonction définie à l'intérieur même d'une procédure ou fonction.

Fonctions

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

Une sous-procédure ou sous-fonction consiste en une procédure ou fonction définie à l'intérieur même d'une procédure ou fonction.

A RETENIR :

Fonctions

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

Une sous-procédure ou sous-fonction consiste en une procédure ou fonction définie à l'intérieur même d'une procédure ou fonction.

A RETENIR :

- une sous-fonction n'est pas appelable en dehors de la fonction dans laquelle elle est incluse.

Fonctions

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

Une sous-procédure ou sous-fonction consiste en une procédure ou fonction définie à l'intérieur même d'une procédure ou fonction.

A RETENIR :

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

Fonctions

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

Une sous-procédure ou sous-fonction consiste en une procédure ou fonction définie à l'intérieur même d'une procédure ou fonction.

A RETENIR :

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

Fonctions

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

Variable locale

```
1 def fonction1():
2     x=1
3     def fonction2():
4         x=2
5         print "au niveau 2, on a x=",x
6         return None
7     fonction2()
8     print "au niveau 1 (aprÃ¨s appel Ã  fonction2), on a x=",x
9     return None
```

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

Fonctions

Cas plus "fin" : variables globales dans les sous-procédures ou sous-fonctions

Variable globale

```
1  ### DÃ©finitions fonction et sous-fonction ###
2  def fonction1():
3      global x
4      x=1
5      def fonction2():
6          global x
7          x=2
8          print "au niveau 2, on a x=",x
9          return None
10     ### Appel Ã  la sous-fonction ###
11     fonction2()
12     print "au niveau 1 (aprÃ¨s appel Ã  fonction2), on a x=",x
13     return None
```

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