IPT : cours 5

Les instructions de base en Python

MPSI : Prytanée National Militaire

Pascal Delahaye

3 novembre 2015

1 Structure d'un programme

Les programmes en Python se tapent dans la fenêtre "Editeur" de Spyder.

Structure d'un programme Python :

```
EDITEUR : (Rédaction du programme)

              ##### Importation des fonctions utiles

              from module1 import f1, f2 ...
              from module2 import g1, g2 ...

              ##### Définition du programme

              def nom_programme(var_1, var_2, ...):
                  instruction1
                  instruction2
                  ...
CONSOLE : (utilisation du programme)

              ##### Utilisation du programme

              nom_programme(valeur_1, valeur_2, ...)
```

Remarque 1. Avant d'exécuter un programme, il faut commencer par l'enregistrer. Vous pouvez le faire dans un répertoire personnel sur le disque dur de l'ordinateur ou sur une clé USB.
On peut alors le lancer en tapant sur la touche F5.

Remarque 2. Après avoir appuyé sur la touche F5, la fonction que l'on vient de créer dans l'éditeur peut être utilisée depuis la console dédiée à celui-ci. On utilise la même commande :

```
nom_programme(valeur_1, valeur_2, ...)
```

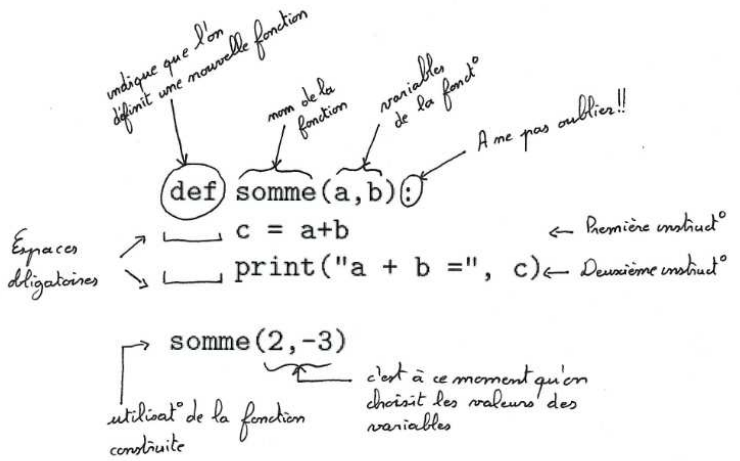
`Exemple 1.` Taper dans l'éditeur le programme suivant :

Python
```
EDITEUR :
              ##### Importation des fonctions utiles (aucune ici !)
              ##### Définition de la fonction "somme"

              def somme(a,b):
                  c = a+b
                  print("a + b =", c)
CONSOLE :
              ##### Utilisation de la fonction "somme"

              somme(2,-3)
```

La construction d'un programme doit respecter certaines règles de syntaxe.
Analysons la façon dont le programme précédent a été rédigé :



Dans le programme ci-dessus, nous :

1. Définissons une fonction de deux variables appelée `somme` qui comprend deux instructions :
   — la première instruction crée une variable c qui contient la valeur "a+b"
   — la deuxième instruction demande d'afficher le résultat obtenu précédemment.
2. Appelons cette fonction pour les valeurs `a = 2` et `b = -3`

Remarque 3. Voici une version interactive du programme précédent :

Python
```
    def somme():
        print("Nous allons calculer la somme de deux nombres.")
        a = eval(input("Entrer une premiere vale  ur : "))
        b = eval(input("Entrer une deuxieme valeur : "))
        print("la somme des deux valeurs est : ", a+b)
```

Même si l'objectif reste l'addition de deux nombres, nous notons ici plusieurs différences avec le programme précédent :

1. La fonction `somme()` n'a plus de variable.
   C'est lors de l'exécution du programme que les valeurs de "a" et "b" sont demandées à l'utilisateur grâce à la commande `eval(input())`.
2. ⚠ La fonction `input()` utilisée sans le `eval()` donne automatiquement la structure de *chaîne de caractères* aux valeurs entrées au clavier.

2 Les principales instructions !

2.1 L'affectation

L'*affectation* consiste à attribuer une valeur à une variable.
L'instruction se présente sous la forme suivante :

Affectation d'une variable

`A = a`	*On affecte la valeur a à la variable A*

1. `A` repésente le nom de la variable
2. `a` représente la valeur que l'on attribue à cette variable

Remarque 4. Si A n'a pas encore été créée, l'affectation engendre en même temps la création de la variable A.

2.2 L'impression à l'écran

Impression à l'écran

`print("texte que l'on souhaite imprimer")`	*Le texte entre guillemets s'imprime à l'écran*
`print(A)`	*Le contenu de la variable A s'imprime à l'écran*
`print("texte",A)`	*Le texte et le contenu de A s'impriment à l'écran*

2.3 Le renvoi d'une valeur

Un programme doit renvoyer un résultat à l'utilisateur, en général sous la forme d'une impression à l'écran.
Cependant, certains programmes destinés à construire des programmes plus complexes (appelé des *sous-programmes*), génèrent un résultat qui n'est pas destiné à être affiché à l'écran mais qui doit être utilisé dans un traitement ultérieurement. Dans ce cas, le résultat est renvoyé par la commande `return`.

Renvoi d'une valeur

`return A`	*La valeur de la variable A est renvoyée*

La structure dans l'éditeur est alors la suivante :

Python
```
EDITEUR :
                ##### Définition des sous-programmes

                def sous_prog1():
                    instructions
                    return A

                def sous_prog2():
                    instructions
                    return A

                ##### Définition du programme principal

                def prog_principal():
                    instructions utilisant les sous programmes précédents
                    print()
```

`Exemple 2.` Un exemple :

Python
```
EDITEUR :
          ##### Définition des sous-fonctions

          def somme(x,y):         **** Renvoie la somme de 2 nombres
              return x+y
          def produit(x,y):       **** Renvoie le produit de 2 nombres
              return x*y

          ##### Définition de la fonction principale

          def calcul(x,y):        **** Renvoie le double de la somme de 2 nombres
              print(produit(somme(x,y),2))
CONSOLE :
          ##### Utilisation de la fonction principale

          calcul(2,3)
```

2.4 Les structures de contrôle

On utilise une "structure de contrôle" lorsque la série d'instructions à effectuer par l'ordinateur dépend de la valeur d'une condition booléenne.
La structure de contrôle utilisée par Python doit respecter la syntaxe suivante :

Structure de contrôle Simple :

```
if condition :
   instruction 1
   ...
else :
   instruction a
   ...
```

Si la condition est vérifiée, faire alors la première série d'instructions, sinon, faire la seconde

`Exemple 3.` Les solutions réelles d'une équation du second degré dépendent de la valeur du discriminant.
Ce qui donne le programme suivant :

Python
```
EDITEUR :
                    def solution (a,b,c) :
                        Delta = b**2 - 4*a*c
                        if Delta < 0 :
                           print("Il n'y a pas de solutions réelles")
                        else :
                           print("Il y a une ou deux solutions réelles")
```

Dans l'exemple précédent, on aimerait bien utiliser une structure de contrôle faisant apparaître 3 conditions.
Cela est possible grâce à la structure :

Structure de contrôle Multiple :

```
if condition 1 :
   instruction a
   ...
elif condition 2 :
   instruction a'
   ...
else
   instruction a''
```

Si la condition 1 est vérifiée, faire alors la première série d'instructions, sinon, si la condition 2 est vérifiée faire la seconde série d'instructions et sinon, faire la dernière.

`Exemple 4.` Le programme précédent se transforme alors de la façon suivante :

Python
```
EDITEUR :
    from math import sqrt

    def solution (a,b,c) :
        Delta = b**2 - 4*a*c
        if Delta < 0 :
           print("Il n'y a pas de solutions réelles")
        elif Delta == 0 :
           print("Il y a une unique solution réelle : ", -b/(2*a))
        else :
           print("Il y a deux solutions réelles distinctes : ",
                 (-b+sqrt(Delta))/(2*a), " et " ,(-b-sqrt(Delta))/(2*a))
```

2.5 Les boucles "for"

On emploit une boucle "for" lorsque l'on souhaite effectuer les mêmes instructions un nombre de fois déterminé.

DÉFINITION 1 : **Itérable**

Les 3 structures de données "liste", "chaine de caractères" et "tuple", vues en début d'année sont des itérables.
Nous verrons plus tard qu'elles partagent un certain nombre de fonction communes dont `len()`.
Elles sont caractérisées par le fait qu'elles correspondent à une séries de "composantes".

1. la liste $[1, 2, 3, 4]$ a pour composantes les valeurs 1, 2, 3 et 4
2. la chaine "bahut" a pour composantes les lettres "b", "a", "h", "u" et "t"
3. le tuple (a, b, c, d) a pour composante les valeurs a, b, c et d.

Une boucle "for" a la structure suivante :

Structure d'une boucle "for" :

```
for i in iterable :
    instruction_1
    instruction_2
          .
          .
          .
    instruction_n
```

Pour i parcourant tour à tour toutes les composantes de l'itérable, effectuer la série d'instructions.

Remarque 5. La variable courante "i" variant souvant de p à q par pas de 1, l'itérable choisi sera la liste donnée par :

`range(p,q+1)`

`Exemple 5.`

1. Affichages multiples :

   Si l'on souhaite afficher tous les nombres pairs de 0 à 20 :

   Python
   ```
   for i in range(1,11) :
       print(2*i)
   ```

2. Si l'on souhaite calculer la somme $S_{100} = \sum_{k=1}^{100} \frac{1}{k}$:

   Python
   ```
   S = 0
   for i in range(1,101) :
      S = S + 1/i
   ```

`Exemple 6.` Contruire les programmes permettant de calculer :

1. $n!$
2. a^b

Exercice : 1

(∗) Construire un programme de codage d'un texte dont le principe est de décomposer le texte en deux textes regroupant pour le premier les lettres de rang paire et pour le deuxième les lettres de rang impaire.
Construire ensuite la procédure de décodage associée.

2.6 Les boucles "while"

Lorsque l'on souhaite effectuer une série d'instructions un nombre de fois difficile à déterminer par avance, on utilise une boucle "while". Les boucles `while` se structurent de la façon suivante :

Structure des boucles "while"

```
while "condition booléenne" :
      instruction 1
      instruction 2
          .
          .
          .
```

Tant que la condition est vérifiée, on effectue les instructions qui suivent.

Remarque 6.

1. Il est possible de remplacer une boucle "for" par une boucle "while" mais cela n'a pas grand intérêt...
2. Les boucle "while "présente un risque de *boucle infinie*. Pour éviter cela, il faut s'assurer que la condition est bien modifiée par les instructions.

`Exemple 7.` Deux exemples de boucles infinies :

Python

```
          # Exemple 1                          |          # Exemple 2
S = 0                                        |   L = []
while S == 0 :                               |   while 0 == 0 :
   print("je suis une boucle infinie !")     |        L.append(1)
```

- Dans l'exemple 1, le programme ne se termine jamais, seule une action de l'utilisateur peut l'interrompre.
- Dans l'exemple 2, le programme s'arrête avec un message d'erreur car la liste L atteind une taille maximale.

`Exemple 8.` La suite de Syracus est définie de la façon suivante : $\begin{cases} u_0 = A \\ u_{k+1} = u_k/2 \text{ si } u_k \text{ est pair} \\ u_{k+1} = 3u_k + 1 \text{ si } u_k \text{ est impair} \end{cases}$.

Que pensez-vous du programme suivant ?

Python

```
       def syracus(A) :
              U=A
              while U != 1 :
                     if U\%2 = 0 : U = U/2
                     else :        U = 3U + 1
```

`Exemple 9.` Programme de jeu où il s'agit de deviner un entier compris entre 0 et 9.

Python

```
from math import floor
from random import random

def jeu() :
    inconnu = int(10*random())
    guess = -1
    while guess != inconnu :
          guess = eval(input("devinez un entier compris entre 0 et 9 : "))
    print("Enfin gagné !")
```

Améliorer ce programme afin :

1. qu'il renvoie le nombre d'essais qu'il vous a fallu pour trouver le nombre inconnu.
2. qu'il indique à l'utilisateur si le nombre proposé est trop grand ou trop petit.

Quelques situations usuelles à connaître :

1. Suites divergeant vers $+\infty$:

   Ecrire en prseudo-langage, un programme permettant de savoir à partir de quelle valeur de n le terme général u_n de la suite dépasse une valeur A.

   Application : (u_n) définie par $u_n = \sum_{k=1}^{n} \frac{1}{k}$.

2. Suites adjacentes :

   Ecrire en pseudo-langage, un programme qui donne une approximation de la limite commune de deux suites adjacentes (u_n) et (v_n) à ε près.

   Application : On considère les deux suites suivantes :

   $$(u_n) : u_n = \sum_{k=1}^{n} \frac{1}{k!} \qquad (v_n) : v_n = u_n + \frac{1}{n!}$$

   Nous verrons en cours de math que les suites (u_n) et (v_n) sont adjacentes et convergent vers e.
   Concevez un programme donnant la valeur de e à une erreur ε près.

Exercice : 2

$(*)$ Concevez un programme permettant de tester si un entier naturel est un nombre premier.