Informatique Pour Tous

Cours 1 – Variables (simples)

Contrairement à une calculatrice « collège », les langages de programmation abstraits (on parle de langage « haut niveau », sous-entendu en degré d'abstraction par rapport au hardware sur lequel il est exécuté) permettent de gagner en versatilité en proposant d'utiliser des **variables**.

Prenons un exemple mathématique, dans un but illustratif : soit la fonction *f* définie par

$$f\begin{cases}\mathbb{R}^2 \to \mathbb{R}\\ x,y \to 3x+y\end{cases}$$

- L'**en-tête** de cette fonction nous donne le **type** des arguments : ici par exemple *x* et *y* sont deux réels
- *x* et *y* sont les arguments de la fonction, qui peuvent prendre plusieurs valeurs différentes
  - → On les décrira *a priori* par des variables
  - → On fixera leurs valeurs *a posteriori* quand on va **appeler** la fonction pour un couple particulier (x_0, y_0)

    Par exemple, pour les valeurs particulières des variables $x = 1$ et $y = 2$, $f(x,y) = 5$

Continuons avec une analogie : en sciences, plutôt que de travailler avec des valeurs numériques, on préfèrera utiliser des symboles (souvent des lettres) qui renvoient vers ces valeurs, de sorte que les expressions littérales soient facilement lisibles.

Exemple :

- Certains de ces symboles représenteront des grandeurs de valeur fixée, qui ne varie pas pendant le problème.

  π ; accélération de la pesanteur : g ; Nombre d'Avogadro : N_A

- D'autres de ces symboles représenteront des grandeurs susceptibles de varier au cours du problème.

  Temps : t ; Coordonnée spatiale : x etc…

De façon analogue, en informatique on préfère travailler avec des **variables**, souvent représentées par une lettre, qui contiennent une donnée, que l'on peut maintenir constante, ou bien faire varier au cours de l'exécution du code.

A / Variables simples

Nous verrons ensemble que tout nombre, chaîne de caractère, ou autre valeur pouvant être stockée dans une variable se ramène toujours *in fine* à une suite binaire finie, stockée dans la mémoire de l'ordinateur (mémoire vive, RAM).

A.1 / Déclaration (fait automatiquement en Python)

Dans les langages procéduraux comme le langage *C / C++* (contrairement à *Python*), il faut **déclarer** une **variable** avant de pouvoir lui **affecter** une valeur. La déclaration revient à :

1 – Indiquer le nombre de bits (espace) sur lequel la **donnée** sera stockée (on parle d'**allouer** de la mémoire)

2 – Trouver et bloquer un emplacement correspondant dans la mémoire vive

3 – Donner un nom à la variable faisant **référence** (pointeur) à l'adresse mémoire dans laquelle la donnée correspondante est stockée.

Pour déclarer une variable, il faut donc savoir sur combien de bits elle sera stockée, ce qui dépendra de son **type**.

Exemples : on a *bool* (tout ou rien, VRAI ou FAUX) : 1 bit

int : 4 octets

float : 8 octets

A.2 / Affectation

Affecter (ou **assigner**, anglicisme) une **valeur** à une **variable** revient à remplir l'espace mémoire vers lequel pointe l'**adresse** à laquelle elle fait **référence**.

Ceci s'écrit : variable = valeur

En algorithmique (pseudo-code) on écrira plutôt : variable ← valeur

Ceci se lit rigoureusement « on range la **valeur** donnée dans l'adresse mémoire à laquelle fait **référence** la **variable**.»
→ Il est très important de comprendre que le symbole = n'est donc pas symétrique !

```
a = 1       # Stocke 1 (entier) dans la variable a
1 = a       # Génère une erreur car on n'a pas le droit de donner le nom « 1 »
            # à une variable (car 1 désigne le chiffre 1).
```

En Python, l'**interpréteur** de commandes est programmé pour déduire le type (et donc le nombre de cases mémoire) d'une variable à partir de la valeur que vous cherchez à lui **affecter**. La **déclaration** est donc inutile car elle est faite automatiquement et implicitement par l'interpréteur.

Exemple : si l'on écrit *x = 21*

1. L'interpréteur comprend que *x* est de type *int* donc il le déclare sur 4 octets
2. Puis *x* est affecté : si à partir de cette ligne on **appelle** *x,* il fera référence à la valeur 21

Remarque: attention à cette interprétation automatique du **type** lors d'une **affectation** :

```
In [1]: a = 1

In [2]: type(a)
Out[2]: int

In [3]: b = 1.

In [4]: type(b)
Out[4]: float
```

```
In [5]: c = '1'

In [6]: type(c)
Out[6]: str
```

```
In [7]: e = 'TRUE'

In [8]: type(e)
Out[8]: str

In [18]: f = True

In [19]: type(f)
Out[19]: bool
```

A.3 / Nom de variable

Règle 1 : Choisir des variables avec des noms clairs et lisibles, ex : vit_mot pour une vitesse moteur, et surtout pas V
Règle 2 : Attention l'interpréteur est sensible à la **casse** : Vit_mot ≠ vit_mot ≠ VIT_MOT
Règle 3 : Certaines fonctions ou méthodes de Python ont déjà un nom, et donc aucune variable ne peut avoir le même.

→ Par exemple, les noms suivants sont interdits (renvoi d'une erreur)

and	**assert**	**break**	**class**	**continue**	**def**	**del**	**elif**
else	**except**	**exec**	**finally**	**for**	**from**	**global**	**if**
import	**in**	**is**	**lambda**	**not**	**or**	**pass**	**print**
raise	**return**	**try**	**while**	**yield**			

Exemple : *Parmi ces affectations lesquelles sont correctes ?*

```
>>> del=6          >>> moyenne=67      >>> 6_vitesse=78      >>> longueur=80.
>>> classe="PTSI 2"   >>> ordonnee == 5
```

A.4 / Types de variables simples

Il existe de nombreux **types** pour les variables, référencés sur le net. Un **type** correspond non seulement à un **espace mémoire** (nombre de bits de stockage) mais aussi à une façon de **coder** les données.

Les types les plus communs que nous rencontrerons en Python seront :

- `int` (entier sur 4 octets)
- `long` (plus rare, entier codé sur *N* octets, Python décide du nombre d'octets nécessaires, ce qui permet d'éviter les problèmes d'Overflow)
- `float` (décimal à virgule flottante, sur 8 octets)
- `str` (chaîne de caractère)
- `bool` (0 ou 1, «FALSE» ou «TRUE», binaire sur 1 bit)

Remarque : Dans *Python*, la fonction native `type(variable)` renvoie le type de la variable en argument.

Exemple :

```
>>> a = '1'          >>> type(a)      # renverra : str
>>> a = 1            >>> type(a)      # renverra : int
>>> b = 1.0          >>> type(b)      # renverra : float
```

On peut également forcer l'interprétation d'une variable sous un type imposé.

Exemple :

```
>>> a = '1'      >>> a2 = int(a)      >>> a2          #vaut : 1
                                       >>> type(a)     #renverra : int
>>> a3 = float(a)     # ici, idem à a3 = float(a2)
>>> a3     # vaut : 1.0                >>> type(a3)    #renverra :float
```

Remarque : si `x` est un *float*, `int(x)` est la partie entière de `x`. Exemple : `int(3.8)` vaut 3.
L'arrondi à l'entier le plus proche est renvoyé par `round(x)`. Exemple : `round(3.8)` vaut 4.

A.5 / Cas particulier de la multi-affectation (unpacking)

Il est possible d'affecter simultanément plusieurs variables.

Exemple :

```
>>> var1, var2 = '1', 4
# est équivalent à :
>>> var1 = '1'
>>> var2 = 4
```

Conseil : Cela rend souvent les codes difficiles à lire, l'utilisation de la multi-affectation est donc déconseillée, sauf dans deux cas :

1) Si une fonction renvoie deux valeurs. On écrira alors par exemple
`x, y = fonction(3.0)` `# avec fonction :` $\mathbb{R} \to \mathbb{R}^2$

2) Si besoin d'une permutation entre deux variables.
Problème posé : La variable `A1` a une certaine valeur. La variable `A2` a une autre valeur.
On veut permuter les valeurs de `A1` ↔ `A2`

2.a – Si on s'interdit l'usage de la multi-affectation :

```
>>> copie_A2 = A2  # on doit faire une copie
>>> A2 = A1  # si pas de copie, on aurait écrasé A2 (valeur perdue)
>>> A1 = copie_A2
```

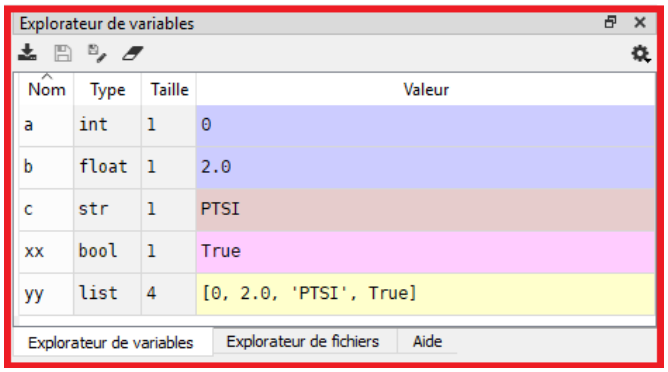
2.b – Si on s'autorise l'usage de la multi-affectation :

```
>>> A1, A2 = A2, A1
```

A.6 / Explorateur de variables

Sur les environnements Python type IDE, vous aurez accès à un explorateur de variables. Pensez à vous en servir, c'est un outil d'aide extrêmement pratique !

- cf ci-contre pour *Spyder*.



B/ Expressions et instructions

B.1 / Expression

Lorsque l'on souhaite traiter des données, on va réaliser des opérations binaires à l'aide d'**objets** (variables ou non) et d'**opérateurs**.

Exemple : pour des **objets** « nombres », les **opérateurs** les plus courants sont :

- Addition, soustraction, multiplication, division : + , - , * , /
- Puissance : ** exemple 3^2 s'écrit 3**2
- Division Euclidienne et reste de la division Euclidienne (parfois très pratique) : // et %
  Ex : 13 // 2 donne 6, et 13 % 2 donne 1. → utile pour savoir si un nombre est un multiple de l'autre

Lorsque l'**interpréteur** repère une **expression** dans une **ligne de commande**, c'est toujours le calcul de l'expression qui est fait avant toute autre **instruction** (notamment d'**affectation**). Si l'on souhaite effectuer une **opération** avant une autre dans une même **expression,** on peut utiliser des parenthèses.

Remarque : attention, la division de 2 entiers ($\mathbb{Z}$) renvoie toujours un flottant (disons $\mathbb{R}$) !

Exemple :

```
>>> num = 4              # type int
>>> denom = 2            # type int aussi
>>> num/denom            # vaut 2.0
>>> type(num/denom)      # renvoie float
```

Solutions : utiliser `int(num/denom)` , ou `round(num/denom)` , ou la division Euclidienne `num//denom`.

B.2 / Instruction

Une **instruction** est une commande qui demande l'exécution d'une tâche. Nous avons par exemple déjà vu **l'affectation** d'une variable, nous verrons également **l'appel à une fonction**, l'**import d'une bibliothèque**, etc.

Il convient ici de s'arrêter sur le cas particulier d'une **affectation** de variable qui nécessite une **expression**. Dans ce cas :
1 – nous avons vu que c'est **l'expression** qui est traitée en premier
2 – le **type** du résultat permet de **déclarer** la **variable** (ce qui est fait automatiquement par Python)
3 – enfin, le résultat est **affecté** à la **variable**

Exemple 1 : *x* = 4*3.0

1. L'expression 4*3.0 est calculée, donne 12.0 (*int* * *float* = *float*, int*int = int)
2. 12.0 est un *float* donc Python va **allouer** 8 octets de mémoire et **déclarer** *x*
3. *x* ← 12.0 : on « range» 12.0 dans l'**espace mémoire** à laquelle fait **référence** *x*

Exemple 2 : $x = x + 1$

- Si x n'a jamais été affecté : renvoie une erreur car l'expression ne peut pas être calculée
- Si x est un *int* (ex : $x = 2$) : a/ L'expression 2 + 1 est calculée, donne 3 (*int + int = int*)
  b/ 3 est un *int* donc 4 octets de mémoire allouée
  c/ $x \leftarrow 3$
- Si x est un *float* (ex : $x = 1.12$) : a/ L'expression 1.12 + 1 est calculée, donne 2.12 (*float + int = float*)
  b/ *float* donc 8 octets
  c/ $x \leftarrow 2.12$

B.3 / Quelques exemples

```
>>> var = 2
>>> var = var*4    # ceci peut aussi s'écrire var *= 4
>>> var    # vaut : 8 (int)
>>> var = var/2    # ceci peut aussi s'écrire var /= 2
>>> var    # vaut : 4.0 (float)
```

La fonction `input(texte)` prend en argument un texte (de type *str*). Elle stoppe momentanément l'exécution du programme, affiche le texte dans le terminal et attend que l'opérateur tape une réponse.
L'exécution du programme reprend alors, et la fonction `input` renvoie, là où on l'a appelée, le **texte** (*str*) tapé par l'utilisateur.

Objectif : On veut demander à l'opérateur son âge, et le stocker comme une valeur **entière** (*int*) dans une variable `age`.

```
>>> question = 'Quel âge avez-vous ?'

# Si on tape simplement ici : input(question)
# → la fonction input est appelée, mais on ne fait rien de la réponse, qui est donc perdue.

>>> reponse = input(question)    # on stocke la réponse dans une variable
                                 # attention, type(reponse) est str !

>>> age = int(reponse)           # forcer l'interprétation en int
```

On a une variable `N` qui contient une valeur entière positive. On veut affecter à une variable `M` la valeur de la plus grande puissance de 2 qui soit $\leq$ `N`. Pour cela, on s'autorise à exécuter en boucle, manuellement (pour le moment), une suite d'instructions.

```
>>> N = 100      # les puissances de 2 sont : 1, 2, 4, 8, 16, 32, 64, 128...
                 # donc la réponse attendue est 64.

>>> puiss = 0    # initialisation

>>> print( 2**puiss )      # la fonction print() permet d'afficher
                           # la valeur d'une variable ou d'un calcul

>>> puiss = puiss + 1   # idem : puiss += 1

                 # On exécute ces 2 lignes en boucle, jusqu'à afficher un nombre
                 # > 100 auquel cas :

>>> M = 2**(puiss-1)
```