Variables, expressions et instructions

1 Introduction à l'algorithmique

1.1 Notion d'algorithme

Un algorithme est une procédure se terminant en nombre *fini* d'étapes qui permet de résoudre une classe de problèmes, écrite de façon suffisamment *détaillée* pour être suivie par un humain ne possédant pas de compétence particulière et qui n'est même pas obligé de comprendre le problème qu'il est en train de résoudre.

Par exemple à l'école primaire on apprend par exemple les algorithmes d'addition de deux nombres entiers, puis ceux de la multiplication de deux nombres entiers.

Ces algorithmes n'auraient que peu d'intérêt si ils ne permettaient que de calculer la somme particulière de deux nombres fixés une fois pour toutes. Ainsi un algorithme fonctionne avec des *données* précisant l'instance du problème qu'il traite. En retour, il construit un *résultat* répondant à cette instance du problème.

1.2 Notion de programme

Les algorithmes existent depuis l'antiquité dans le but de résoudre certaines tâches très calculatoires. Si l'exécution d'un algorithme à la main est en théorie toujours possible, certains algorithmes nécessitent un nombre de calculs très important, ce qui rend cette exécution d'une part longue et augmente le risque d'erreurs humaines d'autre part.

Avec l'avènement de l'informatique, il est devenu possible d'automatiser les algorithmes de sorte à ce qu'il soient exécutés de manière plus rapide et plus sûre. Le prix de cette automatisation est la traduction de l'algorithme sous une forme lisible par la machine. Un *programme* est la traduction d'un algorithme dans un langage particulier, qui est à la fois compréhensible pour l'homme et interprétable pour la machine. Il est exprimé dans un langage de programmation (nous utiliserons Python 2.7.5 dans notre cours) constitué d'un assemblage d'instructions regroupées dans un fichier texte appelé *code source* du programme. Ce code source est ensuite traduit en langage machine par un compilateur.

L'exécution d'un programme commence à la première instruction et continue en exécutant d'autres instructions en suivant des règles précises. Le parcours des instructions au cours de l'exécution d'un programme s'appelle le *flot d'exécution*.

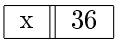
2 Variables

Un des impératifs du traitement automatisé des algorithmes est la nécessité de devoir stocker les données précisant l'instance du problème à résoudre ainsi que les calculs intermédiaires et les résultats dans la machine. Pour ce faire on introduit le concept de variable.

2.1 Notion de variable

Une variable est une représentation idéale d'une zone de la mémoire de l'ordinateur. Il s'agit d'un endroit où on peut stocker une valeur, y accéder et changer cette valeur.

Pour faire référence à une variable on utilise un nom de variable. On représentera dans ce cours et dans les suivants une variable par un diagramme en forme de rectangle, où l'on mettra le nom de la variable dans une case à gauche et sa valeur dans une case à droite. Par exemple, la variable x contient la valeur 36 sera noté :



On considère qu'une variable qui n'a pas ou plus de nom n'existe pas. C'est raisonnable dans la mesure où l'on considère qu'il s'agit d'espace mémoire perdu, que le système réutilise à d'autres fins.

L'ensemble des variables définies à un instant donné de l'exécution d'un programme est appelé un *état*

2.2 Noms de variables en Python

Comme on l'a vu une variable possède un nom et une valeur. Il y a en Python 3.3 un certain nombre de règles sur les noms que peuvent prendre les variables :

- Le nom d'une variable doit commencer par une lettre minuscule (a à z), une lettre majuscule (A à Z) ou un caractère souligné (_).
- Pour la suite des caractères du nom de cette variable on doit utiliser des lettres minuscules et majuscules, des chiffres (0 à 9) et le caractère souligné
- Enfin une variable n'a pas le droit de porter le nom d'un mot réservé : ce sont des mots-clés du langage qui ne sont pas interprétables comme noms de variables par le compilateur. Leur liste est en Python 3 : and, as, 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, with, yield.

Les noms de variables sont sensibles à la casse, ainsi toto, Toto et tOto ne désignent pas la même variable. Par ailleurs, les variables dont les noms commencent par un caractère souligné ne sont pas exportés lorsqu'ils se trouvent dans un module.

Ces règles différent suivant le langage de programmation utilisé.

Au-delà du côté contraignant de ces règles, il faut garder à l'esprit qu'un programme doit pouvoir être réutilisé ou modifié pour répondre à de nouveaux besoins, soit par son concepteur originel, soit par d'autres utilisateurs. On veillera donc à utiliser des noms de variables parlants.

Exercice 1 : Analyser les résultats du programme suivant. Proposer une modification de ce programme de sorte à ce qu'il soit mieux documenté

```
a=input()
b=int(a)
c=60*(a-b)
d=int(c)
e=60*(c-d)
f=int(e)
```

2.3 Valeurs : types de base

Les valeurs sont dites *typées* en Python, autrement dit elles sont classées selon le type d'objet qu'elles représentent. Une valeur peut être de type entier, flottant, chaîne de caractères, ... Des types similaires existent dans la plupart des langages de programmation. Leurs représentations en mémoire varient beaucoup d'un langage à l'autre mais ce sont souvent les mêmes types d'objets que l'on cherche à représenter.

Si il existe de nombreux types de base en Python, et si il est possible de créer soi-même des types qui répondent à un problème donné (cela sera approfondi dans le cours de deuxième année sur la programmation orientée objet), il est nécessaire de bien connaître quelques types de base que l'on manipule tout le temps.

2.3.1 Types simples

Ce sont les types les plus fondamentaux. On distingue notamment :

- Le entiers relatifs. Ils sont codés soit par le type int sur 32 bits soit par le type long arbitrairement long par Python.

- Les nombres décimaux. Ils sont codés selon la norme IEEE754, que l'on étudiera ultérieurement. On parlera de "réel flottant" à précision simple (codé sur 32-bits) et à précision double (codé sur 64-bits)
- Les booléens. Ceux-ci ne se composent que de deux états : vrai ou faux. Python utilise `True` ou `False` et est comme toujours sensible à la casse. Ces types sont utilisés lors des tests logiques.
- Les nombres complexes. Ils sont codés sous la forme $a + bj$ où a et b sont deux réels (flottants) et j vérifie $j^2 = -1$

Exercice 2 : Choisissez le type qui vous semble adapté pour représenter chacune des valeurs suivantes :

- Représenter la taille d'un individu
- Représenter le nombre d'Avogadro
- Calculer le plus petit multiple de 42 supérieur à 2^{200}
- Tenir la comptabilité d'un établissement bancaire

2.3.2 Types composés

On appelle types composés des valeurs formées de plusieurs valeurs de types plus simples. Par exemple un couplet d'entiers est un type composé. On distingue notamment :

- Les n-uplets (tuple en anglais), est une généralisation de la notion de couplet ou de triplet. On peut voir ce type comme la traduction informatique d'un produit cartésien d'ensembles. Pour construire un n-uplet il suffit de placer des expressions entre parenthèses séparées par des virgules.
  Par exemple voici la création d'une variable t triplet composé d'un entier et de deux flottants :

  $$t = (1, 2., 3.45)$$

  L'état est le suivant :

t	(1,2.,3.45)

  Pour accéder aux composantes du n-uplet, on utilise l'expression t[i] où i est le numéro de la composante, que l'on appelle indice. Attention ! En Python, comme dans la plupart des langages on commence à numéroter à partir de 0 et non de 1.
  Ainsi dans l'exemple précédent, l'expression print(t[1]) renverra 2.0
  Les n-uplets sont *immuables* c'est à dire qu'il n'est pas possible d'affecter de nouvelles valeurs aux composantes.
- Les chaînes de caractères (string en anglais ou pour Python) : ce sont des n-uplets particuliers, composés uniquement de caractères. Ainsi ils sont également immuables. Leur intérêt par rapport aux n-uplets réside dans le fait que certaines opérations particulières qui ne sont possibles que sur les chaînes de caractères sont implémentées. Ils feront l'objet d'un cours ultérieur.
- Les listes sont une généralisation des n-uplets. La différence majeure est que les listes sont modifiables là où les n-uplets ne le sont pas. Elles feront également l'objet d'un cours ultérieur.

2.4 Python et typage

Python utilise le typage dynamique. Celui-ci consiste consiste à laisser l'ordinateur réaliser l'opération de typage 'à la volée', lors de l'exécution du code, et de décharger donc le programmeur de la tâche contraignante de déclarer expressément, pour chaque variable qu'il introduit dans son code, son typage.

C'est une solution commode pour un développement rapide de programmes, Python se chargeant lorsqu'il l'estime possible de convertir de changer le type d'une variable. Ainsi le calcul 42+3.14 renvoie un flottant alors que c'est une somme d'un entier et d'un flottant. Python a donc silencieusement converti l'entier 42 en flottant 42.

Le typage n'est pas vérifié lors de l'interprétation d'un programme. De ce fait, des opérations sur un objet peuvent échouer, signifiant que l'objet en question n'est pas du bon type. Malgré l'absence de typage

statique, Python est fortement typé, interdisant des opérations ayant peu de sens (comme, par exemple, additionner un nombre à une chaîne de caractères) au lieu de tenter silencieusement de la convertir en une forme qui a du sens. Python propose des fonctions permettant de transformer les variables dans un autre type. Ainsi :

Exercice 3 Commentez le programme suivant :

```
points = 3.14 #
print("Tu as " + points + " points !") #
points = int(points) #
print("Tu as " + points + " points !") #
points = str(points) #
print("Tu as " + points + " points !") #
```

3 Expressions et instructions

3.1 Notion d'expression

Une expression est une suite de caractères définissant une valeur. Pour calculer cette valeur la machine doit évaluer l'expression.

Le résultat du calcul peut dépendre de l'environnement au moment où le calcul est effectué. Ainsi une expression est plus difficile à évaluer qu'une valeur constante. On parlera donc *des* valeurs possibles d'une expression. Le résultat d'une expression est une valeur, celle-ci pourra être stockée dans une variable.

3*2 ; x+2 ; 0.1+0.1+0.1==0.3 sont des exemples d'expressions.

Les expressions en Python n'ont pas de type, car le type de la valeur d'une expression dépend de l'environnement par le biais des types de ses sous-expressions.

En Python comme dans la plupart des langages de programmation, une expression est :

- soit une constante, comme 42
- soit un nom de variable , comme x
- soit une expression entre parenthèses comme (3+2)
- soit composée de plusieurs expressions réunies à l'aide d'un opérateur comme dans 1 + (3 * 4) où les expressions 1 et (3 * 4) sont réunies à l'aide de l'opérateur +
- composée d'une fonction appliquée à une ou plusieurs autres expressions comme sqrt(9). Nous étudierons les fonctions ultérieurement.

Dans les prochains paragraphes sont présentés les constantes et opérateurs sur les entiers et les décimaux.

3.2 Constantes entières, opérateurs sur les entiers et précédence

Les constantes entières sont d'une longueur arbitraire en Python. Le type <int> codé sur 32 bits est automatiquement remplacé par le type <long> (qui est en réalité géré comme une liste), dont la seule limitation est la place en mémoire dès lors qu'il y a dépassement de capacité (overflow).

Les opérateurs sur les entiers sont résumés dans le tableau suivant :

Opérateur	Opération
+	Addition
-	Soustraction
*	Multiplication
//	Division entière
%	Reste de la division entière
-x	Opposé en préfixe
abs(x)	Valeur absolue
**	Exponentiation

La question de l'ordre dans lequel sont évaluées les expressions qui comprennent plusieurs opérateurs sans parenthèses (par ex. 2+3*4) fait l'objet de règles généralisant celles sur les priorités opératoires. On parle de précédence. Concrètement, en l'absence de parenthèses qui sont toujours évaluées en premier on évalue dans l'ordre :

1. les exponentiations
2. les multiplications, divisions entières et modulos
3. les additions et les soustractions

L'évaluation des fonctions est traitée comme un traitement de parenthèses. C'est à dire que sont d'abord évaluées les expressions qui sont des paramètres des fonctions appelées, puis les résultats des fonctions elles-mêmes et enfin les règles de précédence sont appliquées.

Ainsi dans l'expression n+n**2+fact(n+2) la machine évaluera tout d'abord n-2 puis sa factorielle, puis l'exponentiation et finalement les additions.

Malheureusement ces règles ne suffisent pas toujours à lever les ambiguïtés. Comment comprendre les expressions $2-3-1$ et 2**3**2 ? Dans la plupart des cas les expressions sont évaluées de gauche à droite. Il y a néanmoins une exception pour l'exponentiation, qui est évaluée de droite à gauche. Ainsi $2-3-1$ est évalué comme -2 là où 2**3**2 est évalué comme 2**9 soit 512.

3.3 Opérateurs sur les flottants et précédence

Les constantes flottantes sont définies à l'aide d'un ".". Lorsque l'on souhaite typer une valeur entière comme un flottant on peut la définir en l'écrivant avec un point à al fin de la partie entière. Par exemple, t=2 est un typage d'entier, là où t=2. est un typage de flottant.

Les opérations entre flottants et entiers ont des résultats qui sont toujours des flottants. L'entier est converti par Python en flottant à la volée. Les opérateurs sur les flottant sont les mêmes que ceux sur les entiers, à l'exception de la division flottante qui est notée /. Les règles de précédence sont les mêmes que pour les entiers.

On remarquera que de nombreuses fonctions mathématiques manquent à l'appel. Celles-ci sont contenues dans un *module* de Python appelé "math" (il existe de nombreux modules et on se servira à l'occasion du cours d'ingénierie numérique de plusieurs d'entre eux. Ils rajoutent des fonctions pré-codées à Python, ceci afin d'éviter de devoir résoudre, souvent de manière suboptimale, des problèmes qui ont déjà été résolus par d'autres utilisateurs). Il faut l'importer à l'aide d'une commande "import math as ..". Heureusement ce module est pré-importé dans l'environnement de développement que nous avons choisi (Spyder).

3.4 Opérateurs sur les autres types de base

Nous étudierons les opérateurs sur les booléens séparément, à l'occasion de notre étude de l'algèbre de Boole, qui sera menée lors du cours sur les instructions conditionnelles. De même, nous étudierons les opérateur des chaînes de caractères à l'occasion du cours et des TPs qui leur sont consacrés.

3.5 Notion d'instruction

Comme vu précédemment, l'état de la mémoire lors de l'exécution d'un programme à un instant est l'ensemble des variables définies (leur nom, leur valeur typée et éventuellement d'autres propriétés) à cet instant.

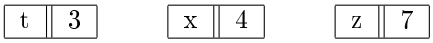
Une instruction est une modification de l'état d'un programme. Cela peut être la création d'une nouvelle variable, comme la modification de variables existantes.

Les instructions différent des expressions en ceci qu'une expression ne modifie jamais l'état d'un programme, là où une instruction modifie toujours son état.

Supposons par exemple qu'un programme soit dans l'état suivant :

$t + x$ est une expression, et sa valeur est 7. L'état du programme ne change en effet pas.

Par contre $z = t + x$ est une instruction, puisque la valeur 7 est affectée à la variable z. Le nouvel état du programme devient en effet :



On distingue les instructions simples qui manipulent directement l'état courant (il s'agit de la déclaration et de l'affectation) et les instructions composées qui permettent d'assembler d'autres instructions et de modifier le flot d'exécution en fonction de l'état courant. Celles-ci sont au nombre de trois : il 'agit de la séquence, du test et de la boucle.

Ces cinq instructions à elles seules suffisent à exprimer tous les algorithmes que l'on peut écrire. C'est l'objet de la thèse de Church-Turing sur les procédés de calcul, dont le formalisme dépasse largement le niveau de ce cours.

Exercice 4 : Les suites de symboles suivantes sont-elles des expressions ou des instructions ?

- `a`
- `b=a`
- `b==a`
- `c*2`
- `b = b+1`
- `float(input())`
- `x=int(input())`

3.6 Instruction simples : déclaration et affectation

Déclarer une variable consiste à l'ajouter à l'état. En Python, cela se fait en évaluant une instruction de la forme :

```
nom_de_variable = expression
```

Comme cette instruction n'a pas de valeur, Python n'affiche rien, même si l'on travaille dans une console.

Le programme suivant :

```
x=3
y=x
```

va créer l'état :

L'instruction `y=x` est bien une déclaration de variable et n'a pas pour effet de renommer x. On considèrera pour simplifier qu'il est impossible de supprimer une variable en Python.

Une affectation est une instruction qui modifie la valeur d'une variable. Elle se note exactement de la même manière qu'une déclaration en Python. Dans le programme suivant on a ainsi une déclaration, suivie d'une affectation :

```
x=2
x=3
```

On utilisera très souvent ce genre d'écritures condensées :

```
x=x+2
```

La variable x joue deux rôles très différents à gauche et à droite de cette instruction. À gauche, il s'agit du nom de la variable sur laquelle va s'effectuer l'affectation. À droite, il s'agit d'une expression qui doit être évaluée où la variable x apparaît. Il est à noter que cette instruction renvoie une valeur si la variable x n'a pas été précédemment déclarée. Il ne pourra en aucun cas s'agir d'une déclaration donc.

Exercice 5 : Écrire une suite d'instructions qui échange deux variables x et y

3.7 Instructions composées

Passons maintenant aux instructions composées. On distingue :

- La *séquence d'instructions* qui consiste à assembler deux ou plusieurs instructions, on parle de bloc d'instructions. La manière la plus simple de le faire et des places ces instructions les unes à la suite des autres sur des lignes différentes dans le code source. L'exécution commencera à la première instruction et passera ensuite à la seconde, puis à la troisième, ... Ce type de programmation est appelé programmation séquentielle (par opposition à la programmation événementielle notamment).
- L'instruction *conditionnelle* consiste à exécuter une instruction si une certaine condition est remplie. Elle est la plupart du temps codée par un bloc "if ... (else ...)" en Python. Nous l'étudierons en détail lors d'un prochain cours.
- La *boucle* simple ou conditionnelle consiste à répéter un certain nombre de fois un bloc d'instructions, éventuellement sous certaines conditions. Elle est introduite par les mots clés "for" ou "while". Elle sera également étudiée en détails lors d'un cours ultérieur.

Exercice 6 : Décrire l'évolution de l'état du programme suivant en précisant si chaque ligne de code est une expression ,une instruction de déclaration ou une instruction d'affectation. On part de l'état suivant :

b	5

```
a = 2

a = b-7

b+1

b = b+1

c = a-b

a= 1./a + a - 1./a
```

3.8 Cas particulier des entrées-sorties

On appelle entrées-sorties les constructions qui permettent d'arrêter le flot d'exécution d'un programme pour communiquer avec un utilisateur.

On a ainsi déjà rencontré la commande `print()` qui permet d'afficher à l'écran un texte ou la valeur d'une variable. `print()` est considéré comme une instruction, au sens où même si il ne modifie pas à proprement parler l'état d'un programme il modifie une interface - à savoir l'écran de l'utilisateur.

On a également rencontré l'expression `input()` qui interrompt un programme et attend que l'utilisateur tape au clavier une séquence de caractères et qui prend cette valeur. L'expression `input()` sera le plus généralement associée à une instruction de déclaration ou d'affectation pour stocker la valeur entrée par l'utilisateur (à la rare exception de "appuyez sur entrée pour continuer")