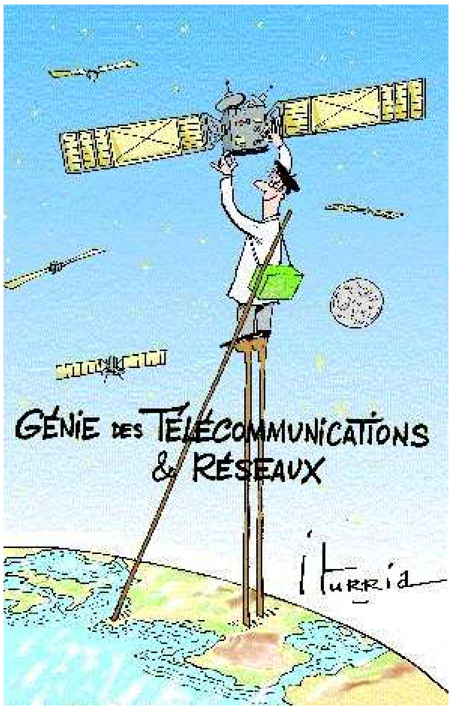


Université de Pau et des Pays de l'Adour

INSTITUT UNIVERSITAIRE DE TECHNOLOGIE DES PAYS DE L'ADOUR

Département Génie des Télécommunications et Réseaux

371, rue du Ruisseau
BP 201
40004 Mont-de-Marsan Cedex
tél : 05 58 05 76 00
fax : 05 58 06 83 70

Algorithmique

Manuel Munier
E-mail: munier@univ-pau.fr

Version du 10 septembre 2002

Table des matières

1 Introduction . . . **3**
- 1.1 Qu'est-ce que la programmation ? . . . 3
- 1.2 Langage machine, langage de programmation . . . 3

2 Algorithmique & Python . . . **4**
- 2.1 Instructions élémentaires . . . 5
  - 2.1.1 Premiers pas en Python . . . 5
  - 2.1.2 Utilisation de variables . . . 6
  - 2.1.3 Opérateurs et expressions . . . 9
  - 2.1.4 Composition . . . 10
- 2.2 Instructions de contrôle . . . 10
  - 2.2.1 Séquence d'instructions . . . 10
  - 2.2.2 Exécution conditionnelle - l'instruction `if` . . . 10
  - 2.2.3 Opérateurs de comparaison . . . 12
  - 2.2.4 Opérateurs booléens . . . 12
  - 2.2.5 Instructions composées - Blocs d'instructions . . . 13
  - 2.2.6 Instructions imbriquées . . . 13
  - 2.2.7 Quelques règles de syntaxe Python . . . 14
- 2.3 Instructions répétitives . . . 15
  - 2.3.1 Ré-affectation . . . 15
  - 2.3.2 Premières itérations - l'instruction `while` . . . 16
  - 2.3.3 Elaboration de tables . . . 17
  - 2.3.4 Construction d'une suite mathématique . . . 17
  - 2.3.5 Autre forme d'itérations - l'instruction `for` . . . 18
- 2.4 Comment conserver nos programmes ? . . . 19
  - 2.4.1 Premiers scripts . . . 19
  - 2.4.2 Utilisation de fonctions prédéfinies . . . 20
  - 2.4.3 Interaction avec l'utilisateur - l'instruction `input()` . . . 22
- 2.5 Fonctions . . . 22
  - 2.5.1 Définir une fonction . . . 23
  - 2.5.2 Variables locales, variables globales . . . 26
  - 2.5.3 "Vraies" fonctions et procédures . . . 28
  - 2.5.4 Utilisation des fonctions dans un script . . . 29
  - 2.5.5 Valeurs par défaut pour les paramètres . . . 30
  - 2.5.6 Arguments avec étiquettes . . . 30

1 Introduction

Certains voient, à tort, dans l'ordinateur une machine pensante et intelligente, capable de résoudre bien des problèmes. En fait, celui-ci ne serait capable de rien si quelqu'un (l'**analyste-programmeur** en l'occurence) ne lui avait fourni la liste des actions à exécuter. Cette description doit être faite de manière non ambigüe car il ne faut pas s'attendre à la moindre interprétation des ordres fournis. Ils seront exécutés de manière purement mécanique.

1.1 Qu'est-ce que la programmation ?

Le but de ce cours est de vous apprendre à penser et à réfléchir comme un analyste-programmeur. Ce mode de pensée combine des démarches intellectuelles complexes, similaires à celles qu'accomplissent les mathématiciens, les ingénieurs et les scientifiques.

Comme le mathématicien, l'analyste-programmeur utilise des langages formels pour décrire des raisonnements : les algorithmes. Comme l'ingénieur, il conçoit des dispositifs, assemble des composants pour réaliser des mécanismes et évalue leurs performances. Comme le scientifique, il observe le comportement de systèmes complexes, il ébauche des hypothèses explicatives, il teste des prédictions.

L'activité essentielle d'un analyste-programmeur est **la résolution de problèmes**. Il s'agit là d'une compétence de haut niveau qui implique des capacités et des connaissances diverses : être capable de (re)formuler un problème de plusieurs manières différentes, être capable d'imaginer des solutions innovantes et efficaces, être capable d'exprimer ces solutions de manière claire et complète.

La **programmation** d'un ordinateur consiste en effet à *expliquer* en détail à une machine ce qu'elle doit faire, en sachant d'emblée qu'elle ne peut pas véritablement *comprendre* un langage humain, mais seulement effectuer un traitement automatique sur des séquences de caractères. Un programme n'est rien d'autre qu'une suite d'instructions encodées en respectant de manière très stricte un ensemble de conventions fixées à l'avance que l'on appelle un **langage de programmation**. La machine est ainsi pourvue d'un mécanisme qui décode ces instructions en associant à chaque *mot* du langage (opérations élémentaires) une action précise.

De plus, les opérations élémentaires que peut exécuter un ordinateur sont en nombre restreint et doivent être communiquées de façon précise dans un langage qu'il comprendra. Le problème principal de l'analyste-programmeur est donc de lui décrire la suite des actions élémentaires permettant d'obtenir, à partir des données fournies, les résultats escomptés. Cette description doit être précise, envisager le moindre détail et prévoir les diverses possibilités de données.

Au cours de cette activité de programmation, le rôle de l'**algorithme** est fondamental. En effet, sans algorithme, il n'y aurait pas de programme (qui n'est jamais que sa traduction dans un langage compréhensible par l'ordinateur). De plus, les algorithmes sont fondamentaux en un autre sens : ils sont indépendants à la fois de l'ordinateur qui les exécute et des langages de programmation dans lesquel ils sont énoncés et traduits.

Vous allez donc apprendre à programmer, activité déjà intéressante en elle-même parce qu'elle contribue à développer votre intelligence. Mais vous serez également amené à utiliser la programmation pour réaliser des projets concrets dans les différentes matières du GTR, ce qui vous procurera certainement de grandes satisfactions. . . !

1.2 Langage machine, langage de programmation

A strictement parler, un ordinateur, si puissant qu'il puisse être, n'est rien d'autre qu'une machine effectuant des opérations simples sur des séquences de signaux électriques, lesquels sont conditionnés de manière à ne pouvoir prendre que deux états seulement (par exemple un potentiel électrique maximum ou minimum). Ces séquences de signaux obéissent à une logique de "tout ou rien" et peuvent donc être considérés conventionnellement comme des suites de nombres ne prenant jamais que les deux valeurs 0 et 1. Un système numérique ainsi limité à deux chiffres est appelé système binaire.

Sachez dès à présent que dans son fonctionnement interne, un ordinateur est totalement incapable de traiter autre chose que des nombres binaires. Toute information d'un autre type doit être convertie (ou codée) en **format binaire**. Cela est vrai non seulement pour les données que l'on souhaite traiter (les textes, les images, les sons, les nombres,...) mais aussi pour les programmes, c'est-à-dire les séquences d'instructions que l'on va fournir à la machine pour lui dire ce qu'elle doit faire avec ces données.

Le seul *langage* que l'ordinateur puisse véritablement *comprendre* est donc très éloigné du langage naturel que nous utilisons. Il s'agit d'une suite de 1 et de 0 (les *bits*) souvent traités par groupes de 8 (les *octets*), 16, 32, ou même 64. Ce **langage machine** est évidemment presqu'incompréhensible pour nous. Pour *parler* à un ordinateur il nous faudra utiliser des systèmes de traduction automatiques capables de convertir en nombres binaires des suites de caractères formant des mots-clés (anglais en général) qui seront plus significatifs pour nous. Ces systèmes de traduction automatique seront établis sur la base de toute une série de conventions dont il existera évidemment de nombreuses variantes.

Le système de traduction proprement dit s'appellera **interpréteur** ou bien **compilateur** suivant la méthode utilisée pour effectuer la traduction (voir ci-après). On appellera **langage de programmation** un ensemble de mots-clés (choisis arbitrairement) associé à un ensemble de règles très précises indiquant comment on peut assembler ces *mots* pour former des *phrases* que l'interpréteur ou le compilateur puisse traduire en langage machine (binaire).

Suivant son niveau d'abstraction on pourra dire d'un langage qu'il est "de bas niveau" (ex : assembleur) ou "de haut niveau" (ex : Python, Pascal, C, C++, Java,...). Un langage de bas niveau est constitué d'instructions très élémentaires, très "proches de la machine". Un langage de haut niveau comporte des instructions plus abstraites ou plus puissantes. Cela signifie que chacune de ces instructions pourra être traduite par l'interpréteur ou le compilateur en un grand nombre d'instructions machine élémentaires.

Le premier langage de programmation que vous allez apprendre en **Python**. Il s'agit d'un langage de haut niveau dont la traduction en codes binaires est complexe et prend donc toujours un certain temps. Cela pourrait vous paraître un inconvénient. En fait les avantages que procurent les langages de haut niveau sont énormes :

- il est **beaucoup plus facile** d'écrire un programme dans un langage de haut niveau
- l'écriture du programme prend donc beaucoup moins de temps
- la probabilité d'y faire des erreurs est nettement plus faible
- la maintenance (i.e. l'apport de modifications ultérieures) et la recherche des erreurs (les fameux *bugs*) sont grandement facilitées

De plus, un programme écrit dans un langage de haut niveau sera souvent **portable**, c'est-à-dire que l'on pourra le faire fonctionner sans trop de modifications sur des machines ou des systèmes d'exploitation différents (Window$ 9x, Linux, HP-UX,...). A l'inverse, un programme écrit dans un langage de bas niveau (par exemple l'**assembleur** que vous étudierez plus tard en cours d'achitecture) ne peut fonctionner que sur un seul type de machine : pour l'exécuter sur un autre type de machine, il faut le réécrire entièrement.

Au cours de votre cursus au sein du GTR de Mont-de-Marsan vous apprendrez encore deux autres langages de programmation de haut niveau : la langage **C** (plus courant mais syntaxiquement pointilleux) en première année, puis le langage **Java** (langage orienté objets) en deuxième année. Mais chaque chose en son temps. Pour l'instant, le langage **Python** sera amplement suffisant pour vous initier aux concepts de base de l'algorithmique et de la programmation `;-)`

2 Algorithmique & Python

Etant donné que cette première partie du cours est consacrée à l'algotithmique, il peut vous sembler déroutant que nous parlions de Python. Nous avons en effet affirmé page 3 que *"les algorithmes sont indépendants à la fois de l'ordinateur qui les exécute et des langages de programmation dans lesquel ils sont énoncés et traduits"*. Pourquoi déjà s'intéresser à un langage de programmation particulier ? Et dans ce cas, pourquoi ne pas démarrer immédiatement avec le langage C ? Ces deux questions appellent en fait la même réponse. L'objectif d'un programme est de décrire très précisément à l'ordinateur les différentes actions à exécuter pour calculer le résultat attendu. Le langage naturel n'étant pas adapté à ce besoin de précision (problèmes d'ambiguïté, de redondance,...), sont apparus des langages formels pouvant être facilement analysés par la machine : les langages de programmation. Cette précision avait cependant un coût : les premiers langages tels que Pascal ou C étaient affublés d'une foule de détails qui leur étaient propres (type de données, entête d'un programme, gestion des chaînes de caractères, etc...).

Si ces "détails" apparaissent anodins à un programmeur confirmé, ils ont cependant le mérite de "noyer" le programmeur débutant (c'est-à-dire des personnes telles que vous...). Lors de la mise au point d'un programme, ce dernier passera ainsi plus de temps sur ces détails que sur l'algorithme proprement dit, c'est-à-dire l'enchaînement désiré des actions. C'est pour cette raison que des langages intermédiaires ont été définis : les langages algorithmiques. Il s'agit toujours de langages formels (i.e. fortement structurés) mais débarassés des détails spécifiques à un langage de programmation particulier. De plus, les mots-clés sont généralement traduits en français afin de ne pas effrayer nos programmeurs débutants. Par contre, ils ne peuvent être compilés que sur le papier...

Plus récemment sont apparus de nouveaux langages de programmation tels que Python (développé depuis 1989 par Guido van Rossum et de nombreux contributeurs bénévoles). Bien qu'à la base il s'agissait surtout de regrouper au sein d'un même langage toutes les "propriétés" d'autres langages de programmation très évolués, les développeurs ont décidé, une fois n'est pas coutume, de définir un langage de programmation qui soit facile à utiliser. Tout en restant un véritable langage de programmation, Python s'est donc rapproché des langages algorithmiques définis auparavant.

2.1 Instructions élémentaires

2.1.1 Premiers pas en Python

Python présente la particularité de pouvoir être utilisé de plusieurs manières. Nous allons d'abord l'utiliser en mode *interactif*, c'est-à-dire d'une manière telle que nous pourrons dialoguer avec lui directement depuis le clavier. Cela nous permettra de découvrir très vite un grand nombre de fonctionnalités du langage. Dans un second temps nous verrons comment créer nos premiers programmes (*scripts*) et les sauvegarder.

L'interpréteur peut être lancé directement depuis la ligne de commande (sous Linux, M$DOS) : il suffit de taper la commande `"python"` (en supposant que le logiciel ait été correctement installé). Si vous utilisez une interface graphique telle que Window$, Gnome ou KDE, vous préférerez vraisemblablement travailler dans une "fenêtre de terminal" ou encore dans un environnement de travail spécialisé tel que IDLE (sur votre bureau Window$ c'est l'icône avec un pyhon). Au démarrage de l'interpréteur vous obtenez un message d'introduction tel que celui-ci :

```
Python 2.1 (#15, Apr 16 2001, 18:25:49) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>>
```

Les trois caractères `">"` constituent le signal d'invite, ou *prompt principal*, lequel vous indique que Python est prêt à exécuter une commande. Par exemple, vous pouvez tout de suite utiliser l'interpréteur comme une simple calculatrice de bureau. Veuillez donc vous-même tester les commandes ci-dessous (soit dit au passage, prenez l'habitude d'utiliser votre cahier d'exercices pour noter les résultats qui apparaissent à l'écran) :

```
>>> 5+3

>>> 2 - 9                  # les espaces sont optionnels

>>> 7 + 3 * 4              # la hiérarchie des opérations mathématiques
                           # est-elle respectée ?
```

```
>>> (7+3)*4

>>> 20 / 3                  # surprise !!!
```

Comme vous pouvez le constater, les opérateurs arithmétiques pour l'addition, la soustraction, la multiplication et la division sont respectivement "+", "-", "*" et "/". Les parenthèses sont fonctionnelles.

Par défaut, la division est cependant une division entière, ce qui signifie que si on lui fournit des arguments qui sont des nombres entiers, le résultat de la division est lui-même un entier (tronqué), comme dans le dernier exemple ci-dessus.

Si vous voulez qu'un argument soit compris par Python comme étant un nombre réel, il faut le lui dire explicitement en fournissant au moins un point décimal. A noter que dans tous les langages de programmation les conventions mathématiques de base sont celles en vigueur dans les pays anglophones : le séparateur décimal sera donc toujours un point, et non une virgule comme chez nous[1]. Essayez par exemple :

```
>>> 20.0 / 3                # comparez le résultat avec celui obtenu à
                            # l'exercice précédent

>>> 8./5
```

Si une opération est effectuée avec des arguments de types mélangés (entiers et réels), Python convertit automatiquement les opérandes en réels avant d'effectuer l'opération. Essayez :

```
>>> (4 * 2.5) / 3.3
```

2.1.2 Utilisation de variables

Nous reviendrons plus loin sur les différents types de données numériques. Mais avant cela, nous pouvons dès à présent aborder un concept essentiel.

La programmation d'un ordinateur - à l'aide d'un langage quelconque - fait abondamment usage d'un grand nombre de **variables** de différents types. Une variable apparaît dans un langage de programmation sous un **nom de variable** à peu près quelconque (voir ci-après), mais pour l'ordinateur il s'agit d'une simple étiquette reliée à une **adresse mémoire**, c'est-à-dire un emplacement précis dans la mémoire vive.

A cet emplacement est stocké une **valeur** bien déterminée. Cette valeur n'est pas nécessairement une valeur numérique. Cela peut être en fait à peu près n'importe quel "objet" susceptible d'être placé dans la mémoire d'un ordinateur, comme par exemple : un nombre entier, un nombre réel, un vecteur, un nombre complexe, une chaîne de caractères typographiques, un tableau, une fonction, etc...

Toutes ces valeurs doivent évidemment être d'abord encodées sous la forme d'un certain nombre d'octets (Voir cours d'informatique générale). Étant donnée la grande variété des contenus possibles, il a fallu définir conventionnellement différents **types de variables**. Nous y reviendrons plus loin.

Noms de variables et mots réservés Les noms de variables que vous utilisez sous Python doivent obéir à quelques règles simples :

- Un nom de variable est une séquence de lettres (a $\rightarrow$ z, A $\rightarrow$ Z) et de chiffres (0 $\rightarrow$ 9), qui doit toujours commencer par une lettre.
- Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que `$`, `#`, `@`, etc... sont interdits, à l'exception du caractère `_` (souligné).
- La casse est significative (c.à.d. que les caractères majuscules et minuscules sont distingués).
  Attention : `Joseph`, `joseph`, `JOSEPH` sont donc des variables différentes. Soyez attentifs !

[1]Dans le monde de l'informatique, les nombres réels sont souvent désignés comme des nombres "à virgule flottante", ou encore de nombres "de type **float**".

Prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules (y compris la première lettre[2]). N'utilisez les majuscules qu'à l'intérieur même du nom, pour en augmenter éventuellement la lisibilité, comme dans `tableDesMatieres`, par exemple.

En plus de ces règles, il faut encore ajouter que vous ne pouvez pas utiliser comme noms de variables les 28 *mots réservés* ci-dessous (ils sont utilisés par le langage lui-même) :

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

Affectation (ou assignation) Maintenant que nous savons comment définir un nom de variable, voyons comment y **affecter** une valeur. Les termes "affecter une valeur" ou "assigner une valeur" à une variable sont équivalents. Ils désignent l'opération par laquelle on établit un lien entre le nom de la variable et son contenu. L'instruction d'affectation la plus commune sous Python utilise le signe "égale" :

```
>>> n = 7                        # donner à n la valeur 7
>>> msg = "Quoi de neuf ?"       # affecter la valeur "Quoi de neuf ?" à msg
>>> pi = 3.14159                 # assigner sa valeur à la variable pi
```

Les exemples ci-dessus illustrent des instructions d'affectation Python tout à fait classiques. Après qu'on les ait exécutées, il existe dans la mémoire de l'ordinateur, à des endroits différents :
- trois noms de variables, à savoir **n**, **msg** et **pi**
- trois séquences d'octets, où sont encodées le nombre entier `7`, la chaîne de caractères `Quoi de neuf ?` et le nombre réel `3,14159`.

Les trois instructions d'affectation ci-dessus ont eu pour effet, à la fois de créer les noms des 3 variables, de leur attribuer un type, et d'établir un lien symbolique (par un système interne de pointeurs) entre le nom de chaque variable et la valeur correspondante. On peut mieux se représenter tout cela par un petit diagramme (diagramme d'état) :

n	**msg**	**pi**
↓	↓	↓
`7`	`Quoi de neuf ?`	`3,14159`

Les trois noms de variables sont mémorisés dans une zone particulière de la mémoire que l'on appelle "espace de noms". Nous aurons plus loin l'occasion de revenir sur ce concept.

Puisqu'il s'agit, à la base, d'un cours d'algorithmique, voici la syntaxe d'une affectation en pseudo-langage algorithmique. Cette notation a le mérite d'indiquer clairement qu'une affectation consiste à prendre (copier) la valeur qui se trouve à droite du symbole pour la mettre dans la variable dont le nom est indiqué à gauche du symbole.

n $\leftarrow$ 7 # donner à n la valeur 7
msg $\leftarrow$ "Quoi de neuf ?" # affecter la valeur "Quoi de neuf ?" à msg
pi $\leftarrow$ 3.14159 # assigner sa valeur à la variable pi

Afficher la valeur d'une variable A la suite de l'exercice ci-dessus, nous disposons donc des trois variables **n**, **msg** et **pi**. Pour afficher leur valeur, il existe deux possibilités. La première consiste simplement à entrer au clavier le nom de la variable (suivi de `<Enter>`). Python répond en affichant la valeur :

```
>>> n
7
```

[2]Les noms commençant par une majuscule ne sont pas interdits, mais il est d'usage de les réserver aux variables qui désignent des classes (le concept de classe sera étudié l'année prochaine ne Programmation Orientée Objets).

```
>>> msg
"Quoi de neuf ?"
>>> pi
3.14159
```

Il s'agit là seulement d'une fonctionnalité secondaire de l'interpréteur, destinée à vous faciliter la vie lorsque vous faites de simples exercices à la ligne de commande. A l'intérieur d'un programme, vous utiliserez toujours l'instruction `print` :

```
>>> print msg
Quoi de neuf ?
```

Remarquez la subtile différence dans les affichages obtenus avec chacune des deux méthodes. L'instruction print n'affiche strictement que la valeur de la variable, telle qu'elle a été encodée, alors que l'autre méthode (celle qui consiste à entrer seulement le nom de la variable) affiche aussi des guillemets (pour vous rappeler le type de la variable : nous y reviendrons).

En pseudo-langage algorithmique l'affichage du contenu de la variable `msg` sera réalisé par l'instruction "afficher(msg)", tout simplement.

Typage des variables Sous Python, il n'est pas nécessaire d'écrire des lignes de programme spécifiques pour définir le type des variables avant de pouvoir les utiliser. Il vous suffit en effet d'assigner une valeur à un nom de variable pour que celle-ci soit **automatiquement créée avec le type qui correspond au mieux à la valeur fournie**. Dans l'exercice précédent, par exemple, les variables **n**, **msg** et **pi** ont été créées automatiquement chacune avec un type différent ("nombre entier" pour **n**, "chaîne de caractères" pour **msg**, "nombre à virgule flottante" (ou "float", en anglais) pour **pi**).

Ceci constitue une particularité intéressante de Python, qui le distingue de nombreux autres langages. On dira à ce sujet que **le typage sous Python est un typage dynamique**, par opposition au typage statique qui est de règle par exemple en C++ ou en Java. Dans ces langages, il aurait fallu - par des instructions distinctes - d'abord définir le nom et le type des variables, et ensuite leur assigner un contenu compatible.

Affectations multiples Sous Python, on peut assigner une valeur à plusieurs variables simultanément. Exemple :

```
>>> x = y = 7
>>> x
7
>>> y
7
```

On peut aussi effectuer de **multiples affectations à l'aide d'un seul opérateur** :

```
>>> a, b = 4, 8.33
>>> a
4
>>> b
8.33
```

Dans cet exemple, les variables **a** et **b** prennent simultanément les nouvelles valeurs `4` et `8.33`.

Attention *: les francophones que nous sommes avons l'habitude d'utiliser la virgule comme séparateur décimal, alors que les langages de programmation utilisent toujours la convention en vigueur dans les pays de langue anglaise, c.à.d. le* ***point*** *décimal. La* ***virgule****, quant à elle, est utilisée dans Python pour séparer différents éléments (arguments, etc...) comme on le voit dans notre exemple pour les variables elles-mêmes ainsi que pour les valeurs qu'on leur attribue.*

2.1.3 Opérateurs et expressions

On manipule les valeurs et les variables qui les référencent en les combinant avec des **opérateurs** pour former des **expressions**. Exemple :

```
a, b = 7.3, 12
y = 3*a + b/5
```

Dans cet exemple, nous commençons par affecter aux variables **a** et **b** les valeurs `7.3` et `12`. Comme déjà expliqué précédemment, Python assigne automatiquement le type "réel" à la variable **a**, et le type "entier" à la variable **b**.

La seconde ligne de l'exemple consiste à affecter à une nouvelle variable **y** le résultat d'une expression qui combine les **opérateurs** `*` , `+` et `/` avec les **opérandes a**, **b**, `3` et `5`. Les opérateurs sont des symboles spéciaux qui sont utilisés pour représenter des opérations mathématiques simples, telles l'addition ou la multiplication. Les opérandes sont les valeurs combinées à l'aide des opérateurs.

Python évalue chaque expression qu'on lui soumet, aussi compliquée soit-elle, et le résultat de cette évaluation est toujours lui-même une valeur. A cette valeur est attribué automatiquement un type, lequel dépend de ce qu'il y avait dans l'expression. Dans l'exemple ci-dessus, **y** sera du type "réel", parce que l'expression évaluée pour déterminer sa valeur contient elle-même au moins un réel.

Les opérateurs Python ne sont pas seulement les quatre opérateurs mathématiques de base. Il faut y ajouter l'opérateur `**` pour l'exponentiation, ainsi qu'un certain nombre d'opérateurs logiques, des opérateurs agissant sur les chaînes de caractères, des opérateurs effectuant des tests d'identité ou d'appartenance, etc... Nous reparlerons de tout cela plus loin.

Signalons au passage la disponibilité de l'opérateur **modulo**, représenté par le symbole `%`. Cet opérateur fournit **le reste** de la division entière d'un nombre par un autre. Exemples :

```
>>> 10 % 3
1
>>> 10 % 5
0
```

Cet opérateur vous sera très utile plus loin, notamment pour tester si un nombre `A` est divisible par un nombre `B`. Il suffira en effet de vérifier que `A % B` donne un résultat égal à zéro.

Ordre des opérations Lorsqu'il y a plus d'un opérateur dans une expression, l'ordre dans lequel les opérations doivent être effectuées dépend de **règles de priorité**. Sous Python, ces règles de priorité sont les mêmes que celles qui vous ont été enseignées au cours de mathématique. Vous pouvez les mémoriser aisément à l'aide d'un "truc" mnémotechnique, l'acronyme **PEMDAS** :

- **P** pour parenthèses. Ce sont elles qui ont la plus haute priorité. Elles vous permettent donc de "forcer" l'évaluation d'une expression dans l'ordre que vous voulez. Ainsi `2*(3-1) = 4`, et `(1+1)**(5-2) = 8`.
- **E** pour exposants. Les exposants sont évalués ensuite, avant les autres opérations. Ainsi `2**1+1 = 3` (et non `4`), et `3*1**10 = 3` (et non `59049` !).
- **M**ultiplication et **D**ivision ont la même priorité. Elles sont évaluées avant l'**A**ddition et la **S**oustraction, lesquelles sont donc effectuées en dernier lieu. Ainsi `2*3-1 = 5` (plutôt que `4`), et `2/3-1 = -1` (rappelez-vous que par défaut Python effectue une division entière).
- Si deux opérateurs ont la même priorité, l'évaluation est effectuée de gauche à droite. Ainsi dans l'expression `59*100/60`, la multiplication est effectuée en premier, et la machine doit donc ensuite effectuer `5900/60`, ce qui donne `98`. Si la division était effectuée en premier, le résultat serait `59` (rappelez-vous ici encore qu'il s'agit d'une division entière).

2.1.4 Composition

Jusqu'ici nous avons examiné les différents éléments d'un langage de programmation : variables, expressions et instructions, mais sans traiter de la manière dont nous pouvons les combiner.

Or l'une des grandes forces des langages de programmation est qu'ils permettent de construire des instructions complexes par assemblage de fragments divers. Ainsi nous savons à présent comment additionner deux nombres et comment afficher une valeur. Nous pouvons faire les deux en une seule instruction :

```
>>> print 17 + 3
>>> 20
```

Cela n'a l'air de rien, mais cela va nous permettre de programmer des algorithmes complexes de façon claire et concise. Exemple :

```
print "nombre de secondes écoulées depuis minuit = ", h*3600 + m*60 + s
```

Attention cependant : il y a une limite à cette fonctionnalité. Ce que vous placez à la gauche du signe égal dans une expression doit toujours être une variable, et non une expression. Cela provient du fait que le signe égal n'a pas ici la même signification qu'en mathématique : il s'agit d'un symbole d'affectation (nous plaçons un certain contenu dans une variable) et non un symbole d'égalité. Le symbole d'égalité (dans un test conditionnel, par exemple) sera évoqué plus loin.

Ainsi par exemple, l'instruction `m + 1 = b` est tout à fait illégale.

Par contre, écrire `a = a + 1` est inacceptable en mathématique, alors que cette forme d'écriture est très fréquente en programmation. L'instruction `a = a + 1` signifie en l'occurrence "augmenter la valeur de la variable **a** d'une unité" (ou encore : "incrémenter **a**").

Nous aurons l'occasion de revenir bientôt sur ce sujet. Mais avant, il nous faut aborder un autre concept de grande importance.

2.2 Instructions de contrôle

2.2.1 Séquence d'instructions

Sauf mention explicite, les instructions s'exécutent les unes après les autres, dans l'ordre où elles ont été écrites à l'intérieur du programme.

Le "chemin" suivi par Python à travers un programme est appelé un **flux d'instructions**, et les constructions qui le modifient sont appelées des **instructions de contrôle de flux**.

Python exécute normalement les instructions de la première à la dernière, sauf lorsqu'il rencontre une instruction conditionnelle comme l'instruction `if` décrite ci-après (nous en rencontrerons d'autres plus loin, notamment à propos des boucles). Une telle instruction permet au programme de suivre différents chemins suivant les circonstances.

2.2.2 Exécution conditionnelle - l'instruction `if`

Si nous voulons pouvoir écrire des applications véritablement utiles, il nous faut des techniques permettant d'aiguiller le déroulement du programme dans différentes directions, en fonction des circonstances rencontrées. Pour ce faire, nous devons disposer d'instructions capables de **tester une certaine condition** et de modifier le comportement du programme en conséquence.

La plus simple de ces instructions conditionnelles est l'instruction `if`. Veuillez donc entrer dans votre éditeur Python les deux lignes suivantes :

```
>>> a = 4
>>> if (a > 0):
...
```

La première commande affecte la valeur 4 à la variable **a**. Jusqu'ici rien de nouveau. Lorsque vous finissez d'entrer la seconde ligne, par contre, vous constatez que Python réagit d'une nouvelle manière. En effet, et à moins que vous n'ayez oublié le caractère " :" à la fin de la ligne, vous constatez que le prompt principal (>>>) est maintenant remplacé par un prompt secondaire constitué de trois points.

Si votre éditeur ne le fait pas automatiquement, vous devez à présent effectuer une tabulation (ou entrer 4 espaces) avant d'entrer la ligne suivante, de manière à ce que celle-ci soit indentée (c.à.d. en retrait) par rapport à la précédente. Votre écran devrait se présenter maintenant comme suit :

```
>>> a = 150
>>> if (a > 100):
...     print "a dépasse la centaine"
...
```

Frappez encore une fois <Enter>. Le programme s'exécute, et vous obtenez :

```
a dépasse la centaine
```

Recommencez le même exercice, mais avec a=20 en guise de première ligne : cette fois Python n'affiche plus rien du tout.

L'expression que vous avez placée entre parenthèses est ce que nous appellerons désormais une **condition booléenne**[3]. L'instruction if permet de tester la validité de cette condition. Si la condition est vraie, alors l'instruction que nous avons indentée après le " :" est exécutée. Si la condition est fausse, rien ne se passe. Notez que les parenthèses utilisées ici sont optionnelles sous Python. Nous les avons utilisées pour améliorer la lisibilité. Dans d'autres langages, (C, Java,...) elles seraient obligatoires.

Recommencez encore, en ajoutant deux lignes comme indiqué ci-dessous. Veillez bien à ce que la quatrième ligne débute tout à fait à gauche (pas d'indentation), mais que la cinquième soit à nouveau indentée (de préférence avec un retrait identique à celui de la troisième) :

```
>>> a = 20
>>> if (a > 100):
...     print "a dépasse la centaine"
... else:
...     print "a ne dépasse pas cent"
...
```

Frappez <Enter> encore une fois. Le programme s'exécute, et affiche cette fois :

```
a ne dépasse pas cent
```

Comme vous l'aurez certainement déjà compris, l'instruction else ("sinon", en anglais) permet de programmer une exécution alternative, dans laquelle le programme doit choisir entre deux possibilités. On peut faire mieux encore en utilisant aussi l'instruction elif (contraction de "else if") :

```
>>> a = 0
>>> if a > 0 :
...     print "a est positif"
... elif a < 0 :
...     print "a est négatif"
... else:
...     print "a est nul"
...
```

[3]Ce terme vient de l'algèbre de Boole en logique. Un booléen ne peut prendre que deux valeurs : vrai ou faux (respectivement true ou false en anglais).

2.2.3 Opérateurs de comparaison

La condition évaluée après l'instruction `if` peut contenir les **opérateurs de comparaison** suivants (on parle alors d'expression booléenne) :

```
x == y          # x est égal à y
x != y          # x est différent de y
x > y           # x est plus grand que y
x < y           # x est plus petit que y
x >= y          # x est plus grand que, ou égal à y
x <= y          # x est plus petit que, ou égal à y
```

Exemple :

```
>>> a = 7
>>> if (a % 2 == 0):
...     print "a est pair"
...     print "parce que le reste de sa division par 2 est nul"
... else:
...     print "a est impair"
...
```

Notez bien que l'opérateur de comparaison pour l'égalité de deux valeurs est constitué de deux signes "égal" et non d'un seul[4]. Le signe "égal" utilisé seul est un opérateur d'affectation, et non un opérateur de comparaison. Vous retrouverez le même symbolisme en C, C++ et Java.

2.2.4 Opérateurs booléens

Il est également possible de combiner des conditions booléennes pour construire des expressions booléennes bien plus complexes.

```
a and b         # vraie si a et b sont vraies toutes les deux
a or b          # vraie si a est vraie ou si b est vraie (ou les deux)
not a           # vraie uniquement si a est fausse
```

Exemple :

```
>>> ue1,ue2,ue3 = 12,9.5,8.6
>>> moy = (ue1 + ue2 + ue3) / 3.0
>>> if (moy >= 10):
...     print "avoir la moyenne n'était pas si dur..."
...     if ( (ue1 >= 8) and (ue2 >= 8) and (ue3 >= 8) ):
...         print "année validée"
...     else:
...         print "redoublement"
... else:
...     print "réorientation !"
...
```

A noter que cette suite d'instructions se traduit de la manière suivante en pseudo-langage algorithmique. Bien que les mots-clés "si", "alors", "sinon" et "fsi" (fin-de-si) délimitent sans ambiguïté les instructions devant être exécutées quand la condition est vrai ou fausse, il vous est fortement conseillé de respecter

[4] Rappel : l'opérateur `%` est l'opérateur modulo : il calcule le reste d'une division entière. Ainsi par exemple, `a % 2` fournit le reste de la division de `a` par 2.

l'indentation, même dans vos algorithmes. Ceux-ci n'en seront que plus lisibles et, surtout, il vous sera plus facile de les corriger.

```
ue1 ← 12
ue2 ← 9.5
ue3 ← 8.6
moy ← (ue1 + ue2 + ue3) / 3
si (moy ≥ 10) alors
    afficher("avoir la moyenne n'était pas si dur...")
    si ( (ue1 ≥ 8) et (ue2 ≥ 8) et (ue3 ≥ 8) ) alors
        afficher("année validée")
    sinon
        afficher("redoublement")
    fsi
sinon
    afficher("réorientation !")
fsi
```

2.2.5 Instructions composées - Blocs d'instructions

La construction que nous avons utilisée avec l'instruction `if` est notre premier exemple d'instruction composée. Vous en rencontrerez bientôt d'autres. Toutes les instructions composées en Python ont toujours la même structure : une ligne d'en-tête terminée par un double point, suivie par une ou plusieurs instructions indentées sous cette ligne d'en-tête. Exemple :

```
Ligne d'en-tête:
    première instruction du bloc
    ... ...
    dernière instruction du bloc
```

S'il y a plusieurs instructions indentées sous la ligne d'en-tête, elles doivent l'être exactement au même niveau (comptez un décalage de 4 caractères, par exemple). Elles constituent ce que nous appellerons désormais un **bloc d'instructions** (ou une suite d'instructions). Dans l'exemple du paragraphe précédent, les deux lignes d'instructions indentées sous la ligne contenant l'instruction `if` constituent un même bloc logique. Ces deux lignes ne sont exécutées toutes les deux que si la condition testée par if est vraie.

2.2.6 Instructions imbriquées

Il est parfaitement possible d'imbriquer les unes dans les autres plusieurs instructions composées, de manière à réaliser des structures de décision complexes. Exemple :

```
if embranchement == "vertébrés":                     # 1
    if classe == "mammifères":                       # 2
        if ordre == "carnivores":                    # 3
            if famille == "félins":                  # 4
                print "c'est peut-être un chat"      # 5
        print "c'est en tous cas un mammifère"       # 6
    elif classe == 'oiseaux':                        # 7
        print "c'est peut-être un canari"            # 8
print"la classification des animaux est complexe"    # 9
```

Ce fragment de programme n'imprime la phrase `"c'est peut-être un chat"` que si les quatre premières conditions testées sont vraies.

Pour que la phrase `"c'est en tous cas un mammifère"` soit affichée, il faut et il suffit que les deux premières conditions soient vraies. L'instruction d'affichage de cette phrase (ligne 4) se trouve en effet au même niveau d'indentation que l'instruction `if ordre == "carnivores" :` (ligne 3). Les deux font donc partie d'un même bloc, lequel est entièrement exécuté si les conditions testées aux lignes 1 et 2 sont vraies.

Pour que la phrase `"c'est peut-être un canari"` soit affichée, il faut que la variable embranchement contienne `"vertébrés"`, et que la variable classe contienne `"oiseaux"`.

Quant à la phrase de la ligne 9, elle est affichée dans tous les cas, parce qu'elle fait partie du même bloc d'instructions que la ligne 1.

2.2.7 Quelques règles de syntaxe Python

Tout ce qui précède nous amène à faire le point sur quelques règles de syntaxe :

1. **Les limites des instructions et des blocs sont définies par la mise en page**

   Dans d'autres langages de programmation, il faut terminer chaque ligne d'instructions par un caractère spécial (souvent le point-virgule). Sous Python, c'est le caractère de saut à la ligne qui joue ce rôle. (Nous verrons plus loin comment étendre une instruction complexe sur plusieurs lignes). On peut également terminer une ligne d'instructions par un commentaire. Un commentaire Python commence toujours par le caractère spécial `"#"`. Tout ce qui est inclus entre ce caractère et le saut à la ligne suivant est complètement ignoré par le compilateur (ou l'interpréteur).

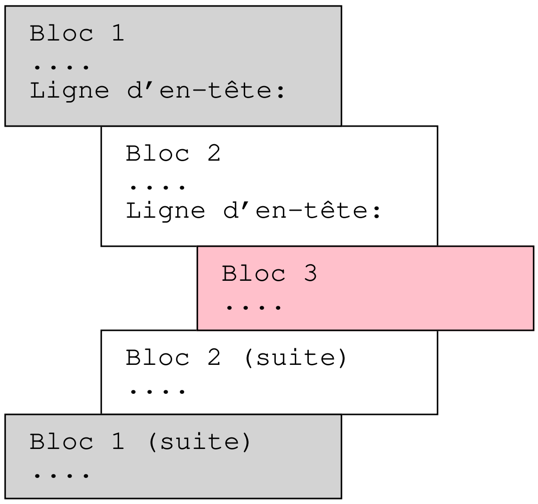
   Dans la plupart des autres langages, un bloc d'instructions doit être délimité par des symboles spécifiques (parfois même par des instructions, telles que `BEGIN` et `END` en Pascal). En C/C++ et en Java,par exemple, un bloc d'instructions doit être délimité par des accolades (`"{"` et `"}"`). Cela permet d'écrire les blocs d'instructions les uns à la suite des autres, sans se préoccuper d'indentation ni de sauts à la ligne, mais cela peut conduire à l'écriture de programmes confus, difficiles à relire pour les pauvres humains que nous sommes. On conseille donc à tous les programmeurs utilisant ces langages de se servir **aussi** des sauts à la ligne et de l'indentation pour bien délimiter visuellement les blocs.

   Avec Python, vous **devez** utiliser les sauts à la ligne et l'indentation, mais en contrepartie vous n'avez pas à vous préoccuper d'autres symboles délimiteurs de blocs. En définitive, Python vous force à écrire du code lisible et à prendre de bonnes habitudes `;-)`

2. **Instruction composée = en-tête, double point, bloc indenté**

   Nous aurons de nombreuses occasions d'approfondir le concept de "bloc d'instructions" et de faire des exercices à ce sujet, dès le chapitre suivant. Le schéma ci-contre en résume le principe :

   – Les blocs d'instructions sont toujours associés à une ligne d'en-tête contenant une instruction bien spécifique (`if`, `elif`, `else`, `while`, `def`,...) et **se terminant par un double point**.
   – **Les blocs sont délimités par l'indentation** : toutes les lignes d'un même bloc doivent être indentées exactement de la même manière (c.à.d. décalées vers la droite d'un même nombre d'espaces[5]). Le nombre d'espaces à utiliser pour l'indentation est quelconque, mais la plupart des programmeurs utilisent des multiples de 4.
   – Notez que le code du bloc le plus externe (bloc 1) ne peut pas lui-même être écarté de la marge de gauche (il n'est imbriqué dans rien).



3. **Les espaces et les commentaires sont normalement ignorés**

A part ceux qui servent à l'indentation, en début de ligne, les espaces placés à l'intérieur des instructions et des expressions sont presque toujours ignorés (sauf s'ils font partie d'une chaîne de caractères). Il en va de même pour les commentaires : ceux-ci commencent toujours par un caractère dièse (#) et s'étendent jusqu'à la fin de la ligne courante.

2.3 Instructions répétitives

2.3.1 Ré-affectation

Nous ne l'avions pas encore signalé explicitement : sous Python, il est permis de ré-affecter une nouvelle valeur à une même variable, autant de fois qu'on le voudra. L'effet d'une ré-affectation est de remplacer l'ancienne valeur d'une variable par une nouvelle.

```
>>> altitude = 320
>>> print altitude
320
>>> altitude = 375
>>> print altitude
375
```

Ceci nous amène à attirer une nouvelle fois votre attention sur le fait que le symbole égale utilisé sous Python pour réaliser une affectation ne doit pas être confondu avec un symbole d'égalité tel qu'il est compris en mathématique. Il est tentant d'interpréter l'instruction `altitude = 320` comme une affirmation d'égalité, mais ce n'en n'est pas une !

- Premièrement, l'égalité est commutative, alors que l'affectation ne l'est pas. Ainsi, en mathématique, les écritures `a = 7` et `7 = a` sont équivalentes. Sous Python, une instruction telle que `375 = altitude` serait illégale.
- Deuxièmement, l'égalité est permanente, alors que l'affectation peut être remplacée comme nous venons de le voir. Lorsqu'en mathématique, nous affirmons une égalité telle que `a = b` maintenant, alors **a** continue à être égal à **b** pour toute la suite du raisonnement. Sous Python, une instruction d'affectation peut rendre égales les valeurs de deux variables, mais une instruction ultérieure peut changer l'une ou l'autre.

```
>>> a = 5
>>> b = a      # a et b contiennent des valeurs égales
>>> b = 2      # a et b sont maintenant différentes
```

Rappelons ici que Python permet d'affecter leurs valeurs à plusieurs variables simultanément :

```
>>> a, b, c, d = 3, 4, 5, 7
```

Cette fonctionnalité de Python est bien plus intéressante encore qu'elle n'en a l'air à première vue. Supposons par exemple que nous voulions maintenant échanger les valeurs des variables **a** et **c**. (Actuellement, **a** contient la valeur `3`, et **c** la valeur `5`. Nous voudrions que ce soit l'inverse). Comment faire ?

```
>>> # Première solution
>>> tampon = a   # car nous allons "écraser" la valeur de a
>>> a = c
>>> c = tampon   # on retrouve ainsi l'ancienne valeur de a !
```

Comme il s'agit d'une opération courante, les langages de programmation proposent souvent des raccourcis pour l'effectuer (par exemple des instructions spécialisée, telle l'instruction `SWAP` du langage Basic). Sous Python, l'affectation multiple permet de programmer l'échange d'une manière élégante :

```
>>> a, b = b, a
```

(On pourrait bien entendu échanger d'autres variables en même temps, dans la même instruction).

2.3.2 Premières itérations - l'instruction `while`

L'une des choses que les machines font le mieux est la répétition sans erreur de tâches identiques. Il existe bien des méthodes pour programmer ces tâches répétitives. Nous allons commencer par l'une des plus simples : la boucle construite à partir de l'instruction `while`. Entrez les commandes ci-dessous :

```
>>> a = 0
>>> while (a < 7):          # (n'oubliez pas le double point !)
...     a = a + 1           # (n'oubliez pas l'indentation !)
...     print a
```

Frappez encore une fois `<Enter>`. Le programme s'exécute, et vous obtenez :

```
1
2
3
4
5
6
7
```

Que s'est-il passé ? Le mot `while` signifie "tant que" en anglais. Cette instruction utilisée à la seconde ligne indique à Python qu'il lui faut répéter continuellement le bloc d'instructions qui suivent, **tant que** le contenu de la variable **a** est (ou reste) inférieur à 7. Comme l'instruction `if` abordée au chapitre précédent, l'instruction `while` amorce une instruction composée. Le double point à la fin de la ligne introduit le bloc d'instructions à répéter, lequel doit obligatoirement se trouver en retrait. Comme vous l'avez appris au chapitre précédent, toutes les instructions d'un même bloc doivent être indentées exactement au même niveau (c.à.d. décalées à droite d'un même nombre d'espaces).

Ainsi nous avons construit notre première boucle de programmation, laquelle répète un certain nombre de fois le bloc d'instructions indentées. Voici comment cela fonctionne :

- Avec l'instruction `while`, Python commence par évaluer la validité de la condition fournie entre parenthèses (celles-ci sont optionnelles ; nous ne les avons utilisées que pour clarifier notre explication).
- Si la condition se révèle fausse, alors tout le bloc qui suit est ignoré et l'exécution du programme se termine[6].
- Si la condition est vraie, alors Python exécute tout le bloc d'instructions constituant **le corps de la boucle**, c.à.d. :
  - l'instruction `a = a + 1` qui incrémente d'une unité le contenu de la variable **a** (c.à.d. que l'on affecte à la la variable a une nouvelle valeur, qui est égale à la valeur précédente augmentée d'une unité).
  - l'instruction `print` qui affiche la valeur courante de la variable **a**.
- Lorsque ces deux instructions ont été exécutées, nous avons assisté à une première **itération**, et le programme boucle, c.à.d. que l'exécution reprend à la ligne contenant l'instruction `while`. La condition est à nouveau évaluée, et ainsi de suite. Tant que la condition `a < 7` reste vraie, le corps de la boucle est à nouveau exécuté et le bouclage se poursuit.

Cette boucle `while` en Python correspond à la boucle "tant que" en pseudo-langage algorithmique ("ftant" signifiant fin-de-tant-que).

a $\leftarrow$ 0
tant que (a $<$ 7) faire
    a $\leftarrow$ a+1
    afficher(a)
ftant

[6]Du moins dans cet exemple. Nous verrons un peu plus loin qu'en fait l'exécution continue avec la première instruction suivant le bloc indenté, et qui fait partie du même bloc que l'instruction `while` elle-même.

Remarques (algo et Python) :

- La variable évaluée dans la condition doit exister au préalable (il faut qu'on lui ait déjà affecté au moins une valeur).
- Si la condition est fausse au départ, le corps de la boucle n'est jamais exécuté.
- Si la condition reste toujours vraie, alors le corps de la boucle est répété indéfiniment (tout au moins tant que Python lui-même continue à fonctionner). Il faut donc veiller à ce que le corps de la boucle contienne au moins une instruction qui change la valeur d'une variable intervenant dans la condition évaluée par `while`, de manière à ce que cette condition puisse devenir fausse et la boucle se terminer. Exemple de boucle sans fin (à éviter) :

```
>>> n = 3
>>> while n < 5:
...     print "hello !"
```

2.3.3 Elaboration de tables

Recommencez à présent le premier exercice, mais avec une petite modification :

```
>>> a = 0
>>> while a < 12:
...     a = a +1
...     print a , a**2 , a**3
```

Vous devriez obtenir la liste des carrés et des cubes des nombres de `1` à `12`. Notez au passage que l'instruction `print` permet d'afficher plusieurs expressions l'une à la suite de l'autre sur la même ligne : il suffit de les séparer par des virgules. Python insère automatiquement un espace entre les éléments affichés.

2.3.4 Construction d'une suite mathématique

Le petit programme ci-dessous permet d'afficher les dix premiers termes d'une suite appelée "Suite de Fibonacci". Il s'agit d'une suite de nombres, dont chaque terme est égal à la somme des deux termes qui le précèdent. Analysez ce programme (qui utilise judicieusement l'affectation multiple). Décrivez le mieux possible le rôle de chacune des instructions.

```
>>> a,b,c = 1,1,1
>>> while c<11:
...     print b,
...     a,b,c = b,a+b,c+1
```

Lorsque vous lancez l'exécution de ce programme, vous obtenez :

```
1 2 3 5 8 13 21 34 55 89
```

Les termes de la suite de Fibonacci sont affichés sur la même ligne. Vous obtenez ce résultat grâce à la virgule placée à la fin de la ligne qui contient l'instruction `print`. Si vous supprimez cette virgule, les nombres seront affichés l'un en-dessous de l'autre.

Voici la version algorithmique de ce programme :

a $\leftarrow$ 1
b $\leftarrow$ 1
c $\leftarrow$ 1
tant que (c $<$ 11) faire
    afficher(b)
    z $\leftarrow$ a   # ATTENTION : c'est le "vieux" a qui est utilisé dans a+b!!!
    a $\leftarrow$ b

b ← z+b
c ← c+1
ftant

2.3.5 Autre forme d'itérations - l'instruction `for`

Si la boucle `while` en Python est très similaire à ses équivalents trouvés dans les autres langages, la boucle `for` est quant à elle assez particulière. Celle-ci permet d'itérer sur les différents éléments d'un ensemble de valeurs. Cet ensemble de valeurs peut être défini :

- soit explicitement (à la main) :
  - `[0, 1, 2, 3, 4, 5]`
  - `[-2, 0, 5, 7, -10, 6, -1]`
- soit à l'aide de la fonction `range`[7] du langage Python :
  - `range(5)` ⇝ `[0, 1, 2, 3, 4]`
  - `range(5,10)` ⇝ `[5, 6, 7, 8, 9]`
  - `range(0,10,2)` ⇝ `[0, 2, 4, 6, 8]`

Voici un exemple de boucle `for` en Python :

```
>>> for i in [0,1,2,3,4,5]:
...     print "le carré de",i,"est",i*i
```

Lorsque vous lancez l'exécution de ce programme, vous obtenez :

```
le carré de 0 est 0
le carré de 1 est 1
le carré de 2 est 4
le carré de 3 est 9
le carré de 4 est 16
le carré de 5 est 25
```

En utilisant la fonction `range`, nous aurions pu écrire ce même programme de la manière suivante :

```
>>> for i in range(6):
...     print "le carré de",i,"est",i*i
```

En langage pseudo-algorithmique cette forme d'itération est représentée par la boucle `pour`. Les trois variantes les plus fréquemment rencontrées sont les suivantes :

1. Ensemble de valeurs explicite :

   pour i dans [0, 1, 2, 3, 4, 5] faire
   afficher("le carré de ",i," est ",i*i)
   fpour

2. Entre deux bornes :

   pour i variant de 0 à 5 faire
   afficher("le carré de ",i," est ",i*i)
   fpour

3. Entre deux bornes avec un certain pas :

   pour i variant de 0 à 10 par pas de 2 faire
   afficher(i," est pair")
   fpour

[7] Syntaxe : `range([start,] stop [,step])`

2.4 Comment conserver nos programmes ?

Jusqu'à présent, nous avons utilisé Python en mode interactif. Ainsi nous avons pu apprendre très rapidement les bases du langage par expérimentation directe. Cette façon de faire présente toutefois un gros inconvénient : toutes les séquences d'instructions que nous avons écrites disparaissent irrémédiablement dès que nous fermons l'interpréteur. Avant d'aller plus loin dans notre étude, il est donc temps que nous apprenions sauvegarder nos programmes dans des fichiers sur disque, de manière à pouvoir les retravailler par étapes successives, les transférer sur d'autres machines, etc...

2.4.1 Premiers scripts

Pour ce faire, nous allons maintenant rédiger nos séquences d'instructions en utilisant un éditeur de textes quelconque (par exemple Kedit ou Kwrite sous KDE-Linux, Edit sous MS-DO$, Notepad sous Window$, etc...), ou encore une interface graphique de développement telle que PythonWin ou IDLE, et nous allons ainsi **écrire un script** que nous pourrons ensuite sauvegarder, modifier, copier,...comme n'importe quel autre texte traité par ordinateur[8].

Lorsque nous voudrons faire s'exécuter notre programme, il nous suffira de lancer l'interpréteur Python en lui fournissant le nom du fichier contenant le script. Par exemple, si nous avons placé un script dans un fichier nommé `"MonScript"`, il suffit d'entrer la commande suivante dans une fenêtre de terminal pour que le script s'exécute :

```
python MonScript
```

Nous pouvons faire mieux encore, en donnant au fichier un nom qui se termine par l'extension **.py** . Si nous respectons cette convention, nous pouvons faire s'exécuter le script simplement en cliquant sur son nom (à l'aide de la souris) dans dans le gestionnaire de fichiers (l'explorateur), parce que le système d'exploitation de l'ordinateur "sait" qu'il doit lancer l'interpréteur Python chaque fois que l'on essaye d'ouvrir un fichier dont le nom se termine par **.py** . (Ceci suppose bien entendu que le gestionnaire de fichiers de notre machine ait été correctement configuré).

Un script Python contiendra donc des séquences d'instructions identiques à celles que nous avons expérimentées jusqu'à présent. Puisque ces séquences sont destinées à être conservées et relues peut-être beaucoup plus tard, il est très fortement conseillé de les expliciter le mieux possible en y adjoignant de nombreuses remarques. Un bon programmeur s'efforce toujours d'insérer un grand nombre de commentaires dans ses scripts, de manière à ce que d'autres que lui puissent aisément comprendre ses algorithmes. Lui-même sera d'ailleurs très heureux de retrouver ses commentaires s'il doit retravailler un des ses propres programmes après quelques années.

On peut insérer des remarques quelconques à peu près n'importe où dans un script Python. Il suffit de les faire précéder d'un caractère `#`. Lorsqu'il rencontre ce caractère, l'interpréteur ignore tout ce qui suit jusqu'à la fin de la ligne courante.

Ouvrez donc un éditeur de texte, et rédigez le script ci-dessous :

[8]Il est évidemment préférable d'utiliser un éditeur "intelligent" tel que IDLE qui vous aide à éviter les fautes de syntaxe. Avec IDLE sous Window$, suivez le menu `File` → `New window` (ou frappez `CTRL-N`) pour ouvrir une nouvelle fenêtre dans laquelle vous écrirez votre script. Pour l'exécuter, il vous suffira de le sauvegarder puis de suivre le menu `Edit` → `Run script` (ou de frapper `CTRL-F5`).

```
#!/usr/bin/env python

# Premier essai de script Python
# petit programme simple affichant une suite de Fibonacci, c.à.d. une suite
# de nombres dont chaque terme est égal à la somme des deux précédents.

a,b,c = 1,1,1               # a & b servent au calcul des termes successifs
                            # c est un simple compteur
print 1                     # affichage du premier terme
while c<15:                 # nous afficherons 15 termes au total
    a,b,c = b,a+b,c+1
    print b
```

Comme vous l'aurez remarqué, nous avons inséré tout au début du script une ligne contenant le "faux commentaire" suivant :

```
#!/usr/bin/env python
```

Il vous est conseillé (ce n'est pas indispensable) de toujours commencer un script Python ainsi, parce que cette ligne rend le script directement exécutable depuis la ligne de commande d'un shell dans les environnements de type Unix (Linux, etc...). Cela signifie qu'il suffira de frapper son nom, ou de cliquer dessus, pour le lancer.

Afin de montrer tout de suite le bon exemple, nous avons ajouté ensuite deux autres lignes de pur commentaire contenant une courte description de ce que le programme est censé faire. Faites de même dans vos propres scripts.

Après avoir bien vérifié votre texte, sauvegardez-le et exécutez-le (consultez votre professeur concernant les autres possibilités de lancement sur d'autres systèmes d'exploitation) :

- Si vous travaillez en mode texte sous Linux ou sous MS-DO$, entrez la commande suivante dans une console : "`python <nom du script>`".
- Si vous travaillez en mode graphique sous Linux, vous pouvez ouvrir une fenêtre de terminal, et faire la même chose. Sous KDE ou Window$, et à la condition d'avoir respecté les conventions décrites plus haut, vous pouvez lancer l'exécution de votre script en effectuant un simple clic sur l'icône correspondante.
- Si vous travaillez avec IDLE, vous pouvez lancer l'exécution du script en cours d'édition, directement à l'aide de la combinaison de touches <Ctrl-F5>.

2.4.2 Utilisation de fonctions prédéfinies

L'un des concepts les plus importants en programmation est celui de **fonction**[9]. Les fonctions permettent en effet de décomposer un programme complexe en une série de sous-programmes plus simples, lesquels peuvent à leur tour être décomposés eux-mêmes en fragments plus petits, et ainsi de suite. D'autre part, les fonctions sont réutilisables : si nous disposons d'une fonction capable de calculer une racine carrée, par exemple, nous pouvons l'utiliser un peu partout dans nos programmes sans avoir à la réécrire à chaque fois.

Vous avez peut-être déjà rencontré des fonctions intégrées au langage lui-même, comme la fonction `len()`, par exemple, qui permet de connaître la longueur d'une chaîne de caractères. Il va de soi cependant qu'il n'est pas possible d'intégrer toutes les fonctions imaginables dans le corps standard de Python, car il en existe virtuellement une infinité : vous apprendrez d'ailleurs très bientôt comment en créer vous-même de nouvelles.

[9] Sous Python, le terme de "fonction" est utilisé indifféremment pour désigner à la fois de véritables fonctions mais également des procédures. Nous indiquerons plus loin la distinction entre ces deux concepts proches.

Les fonctions intégrées au langage sont peu nombreuses : ce sont seulement celles qui sont susceptibles d'être utilisées très fréquemment. Les autres sont regroupées dans des fichiers séparés que l'on appelle des **modules**.

Les modules sont donc des fichiers qui regroupent des ensembles de fonctions. Vous verrez plus loin comme il est commode de découper un programme important en plusieurs fichiers de taille modeste pour en faciliter la maintenance. Une application Python typique sera alors constituée d'un programme principal accompagné de un ou plusieurs modules contenant chacun les définitions d'un certain nombre de fonctions accessoires.

Il existe un grand nombre de modules pré-programmés qui sont fournis d'office avec Python. Vous pouvez en trouver d'autres chez divers fournisseurs. Souvent on essaie de regrouper dans un même module des ensembles de fonctions apparentées que l'on appelle des **librairies**.

Le module `math`, par exemple, contient les définitions de nombreuses fonctions mathématiques telles que sinus, cosinus, tangente, racine carrée, etc... Pour pouvoir utiliser ces fonctions, il vous suffit d'incorporer la ligne suivante au début de votre script :

```
from math import *
```

Cette ligne indique à Python qu'il lui faut inclure dans le programme courant toutes les fonctions (c'est là la signification du symbole `*`) du module `math`, lequel contient une librairie de fonctions mathématiques pré-programmées. On parle d'**importation de modules**.

Dans le corps du script lui-même, vous écrirez par exemple :

- `racine = sqrt(nombre)` pour assigner à la variable `racine` la racine carrée de `nombre`,
- `sinusx = sin(angle)` pour assigner à la variable `sinusx` la valeur du sinus de `angle` (en radians!), etc...

Exemple de programme utilisant des fonctions du module `math` :

```
#!/usr/bin/env python
# Démo: utilisation des fonctions du module <math>

from math import *

nombre = 121
angle = pi/6   # soit 30 degrés (la librairie math inclut aussi la définition de pi)
print 'racine carrée de', nombre, '=', sqrt(nombre)
print 'sinus de', angle, 'radians', '=', sin(angle)
```

L'exécution de ce script provoque l'affichage suivant :

```
racine carrée de 121 = 11.0
sinus de 0.523598775598 radians = 0.5
```

Ce court exemple illustre déjà fort bien quelques caractéristiques importantes des fonctions :

- une fonction est constituée d'**un nom quelconque associé à des parenthèses** ; exemple : `sqrt()`
- dans les parenthèses, on indique à la fonction un ou plusieurs **arguments** ; exemple : `sqrt(121)`
- la fonction fournit une **valeur en retour** (on dira aussi qu'elle "retourne" une valeur) ; exemple : `11.0`

Nous allons développer tout ceci dans les pages suivantes. Veuillez noter au passage que les fonctions mathématiques utilisées ici ne représentent qu'un tout premier exemple. Un simple coup d'oeil dans la documentation des librairies Python vous permettra de constater que de très nombreuses fonctions sont d'ores et déjà disponibles pour réaliser une multitude de tâches, y compris des algorithmes mathématiques très complexes (Python est couramment utilisé dans les universités pour la résolution de problèmes scientifiques de haut niveau). Il est donc hors de question de fournir ici une liste détaillée. Une telle liste est aisément accessible dans le système d'aide de Python :

`Documentation HTML` → `Python documentation` → `Modules index` → `math`

Au chapitre suivant, nous apprendrons comment créer nous-mêmes de nouvelles fonctions. Avant cela, analysons encore un exemple avec une fonction intégrée particulièrement importante : la fonction `input()` pour la saisie d'informations.

2.4.3 Interaction avec l'utilisateur - l'instruction `input()`

La plupart des scripts élaborés nécessitent à un moment ou l'autre une intervention de l'utilisateur (entrée d'un paramètre, clic de souris sur un bouton, etc...). Dans un script simple en mode texte (comme ceux que nous avons créés jusqu'à présent), la méthode la plus simple consiste à employer la fonction `input()`. Cette fonction provoque une interruption dans le programme courant. L'utilisateur est invité à entrer des caractères au clavier et à terminer avec `<Enter>`. Lorsque cette touche est enfoncée, l'exécution du programme se poursuit, et la fonction fournit en retour une valeur correspondant à ce que l'utilisateur a entré. Cette valeur peut alors être assignée à une variable quelconque.

On peut invoquer la fonction `input()` en laissant les parenthèses vides. On peut aussi y placer en argument un message explicatif destiné à l'utilisateur. Exemple :

```
#!/usr/bin/env python
# Utilisation de la fonction intégrée input()

from math import *
print 'Veuillez entrer un nombre positif quelconque : ',
nn = input()
print 'La racine carrée de', nn, 'vaut', sqrt(nn)
```

Notes :

- La fonction `input()` retourne une valeur dont le type correspond à ce que l'utilisateur a entré. Dans notre exemple, la variable `nn` contiendra donc un entier, une chaîne de caractères, un réel, etc... suivant ce que l'utilisateur aura décidé. Si l'utilisateur souhaite entrer une chaîne de caractères, il doit l'entrer comme telle, c.à.d. incluse entre des apostrophes ou des guillemets. Nous verrons plus loin qu'un bon script doit toujours vérifier si le type ainsi entré correspond bien à ce que l'on attend pour la suite du programme.
- Pour cette raison, vous préférerez peut-être utiliser la fonction similaire `raw_input()`, laquelle retourne toujours une chaîne de caractères.

2.5 Fonctions

Les scripts que vous avez écrits jusqu'à présent étaient à chaque fois très courts, car leur objectif était seulement de vous faire assimiler les premiers éléments du langage. Lorsque vous commencerez à développer de véritables projets, vous serez confrontés à des problèmes souvent fort complexes, et les lignes de programme vont commencer à s'accumuler...

L'approche efficace d'un problème complexe consiste souvent à le décomposer en plusieurs sous-problèmes plus simples qui seront étudiés séparément. Ces sous-problèmes peuvent éventuellement être eux-mêmes décomposés à leur tour, et ainsi de suite. Or il est important que cette décomposition soit représentée fidèlement dans les algorithmes pour que ceux-ci restent clairs.

D'autre part, il arrivera souvent qu'une même séquence d'instructions doive être utilisée à plusieurs reprises dans un programme, et on souhaitera bien évidemment ne pas avoir à la reproduire systématiquement.

2.5.1 Définir une fonction

Les **fonctions**[10] et les **classes d'objets** sont différentes structures de sous-programmes qui ont été imaginées par les concepteurs des langages de haut niveau afin de résoudre les difficultés évoquées ci-dessus. Nous allons commencer par décrire ici la définition de fonctions sous Python. Les objets et les classes seront examinés en deuxième année lors du cours "Programmation Orientée Objets".

Nous avons déjà rencontré diverses fonctions pré-programmées. Voyons à présent comment en définir nous-mêmes de nouvelles. La syntaxe Python pour la définition d'une fonction est la suivante :

`def NOM(`*`liste de paramètres`*`) :`
    bloc d'instructions

- Vous pouvez choisir n'importe quel nom pour la fonction que vous créez, à l'exception des mots réservés du langage (voir page 7), et à la condition de n'utiliser aucun caractère spécial ou accentué (le caractère souligné `"_"` est permis). Comme c'est le cas pour les noms de variables, il vous est conseillé de n'utiliser que des lettres minuscules, notamment au début du nom (les noms commençant par une majuscule sont généralement réservés aux classes).
- Comme les instructions `if` et `while` que vous connaissez déjà, l'instruction `def` est une instruction composée. La ligne contenant cette instruction se termine obligatoirement par un double point, lequel introduit un bloc d'instructions que vous ne devez pas oublier d'indenter.
- La liste de **paramètres** spécifie quelles sont les informations qu'il faudra fournir en guise d'arguments lorsque l'on voudra utiliser cette fonction. Les parenthèses peuvent parfaitement rester vides si la fonction ne nécessite pas d'arguments.
- Une fonction s'utilise pratiquement comme une instruction quelconque. Dans le corps d'un programme, un **appel de fonction** est constitué du nom de la fonction suivi de parenthèses. Si c'est nécessaire, on place dans ces parenthèses le ou les arguments que l'on souhaite transmettre à la fonction. Il faudra en principe fournir un argument pour chacun des paramètres spécifiés dans la définition de la fonction, encore qu'il soit possible de définir pour ces paramètres des valeurs par défaut (voir plus loin).

Fonction simple, sans paramètre Pour notre première approche concrète des fonctions, nous allons travailler à nouveau en mode interactif. Le mode interactif de Python est en effet idéal pour effectuer des petits tests comme ceux qui suivent. C'est une facilité que n'offrent pas dans tous les langages de programmation !

```
>>> def table7():
...     n = 1
...     while n <11 :
...         print n * 7,
...         n = n+1
...
```

En entrant ces quelques lignes, nous avons défini une fonction très simple qui calcule et affiche les 10 premiers termes de la table de multiplication par 7. Notez bien les parenthèses[11] , le double point et l'indentation du bloc d'instructions qui suit la ligne d'en-tête (c'est ce bloc d'instructions qui constitue le corps de la fonction proprement dite).

Pour utiliser la fonction que nous venons de définir, il suffit de l'appeler par son nom. Ainsi :

[10] Il existe aussi dans d'autres langages des **routines** (parfois appelés sous-programmes) et des **procédures**. Python utilise le même terme de **fonction**, pour désigner à la fois les fonctions au sens strict (qui fournissent une valeur en retour), et les procédures (qui n'en fournissent pas, ou retournent plutôt une valeur sans objet).

[11] Un nom de fonction doit toujours être accompagné de parenthèses, même si la fonction n'utilise aucun paramètre. Il en résulte une convention d'écriture qui veut que dans un texte quelconque traitant de programmation d'ordinateur, un nom de fonction soit toujours accompagné d'une paire de parenthèses vides. Nous respecterons cette convention dans la suite de ce texte.

```
>>> table7()
```

provoque l'affichage de :

```
7 14 21 28 35 42 49 56 63 70
```

Nous pouvons maintenant réutiliser cette fonction à plusieurs reprises, autant de fois que nous le souhaitons. Nous pouvons également l'incorporer dans la définition d'une autre fonction, comme dans l'exemple ci-dessous :

```
>>> def table7triple():
...     print 'La table par 7 en triple exemplaire :'
...     table7()
...     table7()
...     table7()
...
```

Utilisons cette nouvelle fonction, en entrant la commande :

```
>>> table7triple()
```

l'affichage résultant devrait être :

```
La table par 7 en triple exemplaire :
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
```

Une première fonction peut donc appeler une deuxième fonction, qui elle-même en appelle une troisième, etc... Au stade où nous sommes, vous ne voyez peut-être pas encore très bien l'utilité de tout cela, mais vous pouvez déjà noter deux propriétés intéressantes :

- Créer une nouvelle fonction vous offre l'opportunité de donner un nom à tout un ensemble d'instructions. De cette manière, vous pouvez simplifier le corps principal d'un programme, en dissimulant un algorithme secondaire complexe sous une commande unique, à laquelle vous pouvez donner un nom très explicite, en français si vous voulez.
- Créer une nouvelle fonction peut servir à raccourcir un programme, par élimination des portions de code qui se répètent. Par exemple, si vous devez afficher la table par 7 plusieurs fois dans un même programme, vous n'avez pas à réécrire chaque fois l'algorithme qui accomplit ce travail.

Une fonction est donc en quelque sorte une nouvelle instruction personnalisée, que vous ajoutez vous-même librement à votre langage de programmation.

Fonction avec paramètre Dans nos derniers exemples, nous avons défini et utilisé une fonction qui affiche les termes de la table par 7. Supposons à présent que nous voulions faire de même avec la table par 9. Nous pouvons bien entendu réécrire entièrement une nouvelle fonction pour cela. Mais si nous nous intéressons plus tard à la table par 13, il nous faudra encore recommencer. Ne serait-il donc pas plus intéressant de définir une fonction qui soit capable d'afficher n'importe quelle table, à la demande ?

Lorsque nous appellerons cette fonction, nous devrons bien évidemment pouvoir lui indiquer quelle table nous souhaitons afficher. Cette information que nous voulons transmettre à la fonction au moment même où nous l'appelons s'appelle un **argument**. Nous avons déjà rencontré à plusieurs reprises des fonctions intégrées qui utilisent des arguments. La fonction `sin(a)`, par exemple, calcule le sinus de l'angle `a`. La fonction `sin()` utilise donc la valeur numérique de `a` comme argument pour effectuer son travail.

Dans la définition d'une telle fonction, il faut prévoir une variable particulière pour recevoir l'argument transmis. Cette variable particulière s'appelle un **paramètre**. On lui choisit un nom en respectant les mêmes règles de syntaxe que d'habitude (pas de lettres accentuées, etc...), et on place ce nom entre les parenthèses qui accompagnent la définition de la fonction.

Voici ce que cela donne dans le cas qui nous intéresse :

```
>>> def table(base):
...     n = 1
...     while n <11 :
...         print n * base,
...         n = n +1
```

La fonction `table()` telle que définie ci-dessus utilise le paramètre `base` pour calculer les dix premiers termes de la table de multiplication correspondante.

Pour tester cette nouvelle fonction, il nous suffit de l'appeler avec un argument. Exemples :

```
>>> table(13)
13 26 39 52 65 78 91 104 117 130
>>> table(9)
9 18 27 36 45 54 63 72 81 90
```

Dans ces exemples, la valeur que nous indiquons entre parenthèses lors de l'appel de la fonction (et qui est donc un argument) est automatiquement affectée au paramètre `base`. Dans le corps de la fonction, `base` joue le même rôle que n'importe quelle autre variable. Lorsque nous entrons la commande `table(9)`, nous signifions à la machine que nous voulons exécuter la fonction `table()` en affectant la valeur `9` à la variable `base`.

Utilisation d'une variable comme argument Dans les deux exemples qui précèdent, l'argument que nous avons utilisé en appelant la fonction `table()` était à chaque fois une constante (la valeur `13`, puis la valeur `9`). Cela n'est nullement obligatoire. **L'argument que nous utilisons dans l'appel d'une fonction peut être une variable** lui aussi, comme dans l'exemple ci-dessous. Analysez bien cet exemple, essayez-le concrètement, et décrivez le mieux possible dans vos notes de cours ce que vous obtenez, en expliquant avec vos propres mots ce qui se passe. Cet exemple devrait vous donner un premier aperçu de l'utilité des fonctions pour accomplir simplement des tâches complexes :

```
>>> a = 1
>>> while a <20:
...     table(a)
...     a = a +1
...
```

Remarque importante : Dans l'exemple ci-dessus, l'argument que nous passons à la fonction `table()` est le **contenu** de la variable `a`. A l'intérieur de la fonction, cet argument est affecté au paramètre `base`, qui est une tout autre variable. Notez donc bien dès à présent que **le nom d'une variable que nous passons comme argument n'a rien à voir avec le nom du paramètre correspondant dans la fonction**. Ces noms peuvent être identiques si vous le voulez, mais vous devez bien comprendre qu'ils ne désignent pas la même chose (en dépit du fait qu'ils puissent contenir la même valeur).

Fonction avec plusieurs paramètres La fonction `table()` est certainement intéressante, mais elle n'affiche toujours que les dix premiers termes de la table de multiplication, alors que nous pourrions souhaiter qu'elle en affiche d'autres. Qu'à cela ne tienne. Nous allons l'améliorer en lui ajoutant des paramètres supplémentaires, dans une nouvelle version que nous appellerons cette fois `tableMulti()` :

```
>>> def tableMulti(base, debut, fin):
...     print 'Fragment de la table de multiplication par', base, ':'
...     n = debut
...     while n <= fin :
...         print n, 'x', base, '=', n * base
...         n = n +1
```

Cette nouvelle fonction utilise donc trois paramètres : la base de la table comme dans l'exemple précédent, l'indice du premier terme à afficher, l'indice du dernier terme à afficher. Essayons cette fonction en entrant par exemple :

```
>>> tableMulti(8, 13, 17)
```

ce qui devrait provoquer l'affichage de :

```
Fragment de la table de multiplication par 8 :
13 x 8 = 104
14 x 8 = 112
15 x 8 = 120
16 x 8 = 128
17 x 8 = 136
```

Notes :

- Pour définir une fonction avec plusieurs paramètres, il suffit d'inclure ceux-ci entre les parenthèses qui suivent le nom de la fonction, en les séparant à l'aide de virgules.
- Lors de l'appel de la fonction, les arguments utilisés doivent être fournis **dans le même ordre** que celui des paramètres correspondants (en les séparant eux aussi à l'aide de virgules). Le premier argument sera affecté au premier paramètre, le second argument sera affecté au second paramètre, et ainsi de suite.
- A titre d'exercice, essayez la séquence d'instructions suivantes et décrivez dans vos notes de cours le résultat obtenu :
  ```
  >>> t, d, f = 11, 5, 10
  >>> while t<21:
  ...     tableMulti(t,d,f)
  ...     t, d, f = t+1, d+3, f+5
  ...
  ```

2.5.2 Variables locales, variables globales

Lorsque nous définissons des variables à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des **variables locales** à la fonction. C'est par exemple le cas des variables `base`, `debut`, `fin` et `n` dans l'exercice précédent.

Chaque fois que la fonction `tableMulti()` est appelée, Python réserve pour elle (dans la mémoire de l'ordinateur) un nouvel **espace de noms**. Les contenus des variables `base`, `debut`, `fin` et `n` sont stockés dans cet espace de noms qui est inaccessible depuis l'extérieur de la fonction. Ainsi par exemple, si nous essayons d'afficher le contenu de la variable `base` juste après avoir effectué l'exercice ci-dessus, nous obtenons un message d'erreur :

```
>>> print base
Traceback (innermost last):
  File "<pyshell#8>", line 1, in ?
    print base
NameError: base
```

La machine nous signale clairement que le symbole `base` lui est inconnu, alors qu'il était correctement imprimé par la fonction `tableMulti()` elle-même. L'espace de noms qui contient le symbole `base` est strictement réservé au fonctionnement interne de `tableMulti()`,et il est automatiquement détruit dès que la fonction a terminé son travail.

Les variables définies à l'extérieur d'une fonction sont des variables globales. Leur contenu est "visible" de l'intérieur d'une fonction, mais la fonction ne peut pas le modifier. Exemple :

```
>>> def mask():
...     p = 20
...     print p, q
...
>>> p, q = 15, 38
>>> mask()
20 38
>>> print p, q
15 38
```

Analysons attentivement cet exemple : Nous commençons par définir une fonction très simple (qui n'utilise d'ailleurs aucun paramètre). A l'intérieur de cette fonction, une variable `p` est définie, avec `20` comme valeur initiale. Cette variable `p` qui est définie à l'intérieur d'une fonction sera donc une **variable locale**.

Une fois terminée la définition de la fonction, nous revenons au niveau principal pour y définir les deux variables `p` et `q` auxquelles nous attribuons les contenus `15` et `38`. Ces deux variables définies au niveau principal seront donc des **variables globales**.

Ainsi le même nom de variable `p` a été utilisé ici à deux reprises, **pour définir deux variables différentes** : l'une globale et l'autre locale. On peut constater dans la suite de l'exercice que ces deux variables sont bel et bien des variables distinctes, indépendantes, obéissant à une règle de priorité qui veut qu'à l'intérieur d'une fonction (où elles pourraient entrer en compétition), ce sont les variables définies localement qui ont la priorité.

On constate en effet que lorsque la fonction `mask()` est lancée, la variable globale `q` y est accessible, puisqu'elle est imprimée correctement. Pour `p`, par contre, c'est la valeur attribuée localement qui est affichée.

On pourrait croire d'abord que la fonction `mask()` a simplement modifié le contenu de la variable globale `p` (puisqu'elle est accessible). Les lignes suivantes démontrent qu'il n'en est rien : en dehors de la fonction `mask()`, la variable globale `p` conserve sa valeur initiale.

Tout ceci peut vous paraître compliqué au premier abord. Vous comprendrez cependant très vite combien il est utile que des variables soient ainsi définies comme étant locales, c.à.d. en quelque sorte confinées à l'intérieur d'une fonction. Cela signifie en effet que vous pourrez toujours utiliser quantités de fonctions sans vous préoccuper le moins du monde des noms de variables qui y sont utilisées : ces variables ne pourront en effet jamais interférer avec celles que vous aurez vous-même définies par ailleurs. On parle règles de de **visibilité** d'une variable (locale).

Cet état de choses peut toutefois être modifié si vous le souhaitez. Il peut se faire par exemple que vous ayez à définir une fonction qui soit capable de modifier une variable globale. Pour atteindre ce résultat, il vous suffira d'utiliser l'instruction `global`. Cette instruction permet d'indiquer -à l'intérieur de la définition d'une fonction- quelles sont les variables à traiter globalement.

Dans l'exemple ci-dessous, la variable `a` utilisée à l'intérieur de la fonction `monter()` est non seulement accessible, mais également modifiable, parce qu'elle est signalée explicitement comme étant une variable qu'il faut traiter globalement. Par comparaison, essayez le même exercice en supprimant l'instruction `global` : la variable `a` n'est plus incrémentée à chaque appel de la fonction.

```
>>> def monter():
...     global a
...     a = a+1
...     print a
...
```

```
>>> a = 15
>>> monter()
16
>>> monter()
17
```

Nous ne pouvons toutefois que vous encourager à **ne pas utiliser de variables globales** dans vos programmes. Si vous désirez transmettre des informations à une fonction, la meilleure solution consiste à utiliser le passage de paramètres. Ceci vous évitera, dans une fonction, de vous poser des questions pour savoir si telle ou telle variable est locale ou globale. En n'utilisant que des variables locales vous éviterez ainsi de modifier par inadvertance des variables (globales) utilisées par d'autres fonctions.

2.5.3 "Vraies" fonctions et procédures

Pour les puristes, les fonctions que nous avons décrites jusqu'à présent ne sont pas tout à fait des fonctions au sens strict, mais plus exactement des procédures[12]. Une "vraie" fonction (au sens strict) doit en effet **retourner une valeur** lorsqu'elle se termine. Une "vraie" fonction peut s'utiliser à la droite du signe "égale" dans des expressions telles que `y = sin(a)`. On comprend aisément que dans cette expression, la fonction `sin()` retourne une valeur (le sinus de l'argument `a`) qui est directement affectée à la variable `y`.

Voici un exemple de fonction extrêmement simple :

```
>>> def cube(w):
...     return w*w*w
...
```

L'instruction `return` définit ce que doit être la valeur "retournée" par la fonction. En l'occurrence, il s'agit du cube de l'argument qui a été transmis lors de l'appel de la fonction. Exemple :

```
>>> b = cube(9)
>>> print b
729
```

Autre exemple de fonction retournant un résultat calculé à partir des trois arguments transmis lors de l'appel de la fonction :

```
>>> def jury(ue1,ue2,ue3):
...     moy=(ue1+ue2+ue3)/3.0
...     if (moy>=10):
...         if ( (ue1>=8) and (ue2>=8) and (ue3>=8) ):
...             decision="admission"
...         else:
...             decision="redoublement"
...     else:
...         decision="reorientation"
...     return decision
...
```

[12]Dans certains langages de programmation, les fonctions et les procédures sont définies à l'aide d'instructions différentes. Python utilise la même instruction `def` pour définir les unes et les autres.

```
>>> jury(12,10,9)
'admission'
>>> jury(15,12,7)
'redoublement'
>>> jury(9,9,9)
'reorientation'
```

Notes :

- Comme nous l'avons vu dans l'exemple précédent, l'instruction `return` définit ce que doit être la valeur "retournée" par la fonction[13].
- Il n'est nullement indispensable que la valeur retournée par une fonction soit affectée à une variable (comme nous l'avons fait dans l'exemple de la fonction `cube()` par souci de clarté). Nous aurions pu tester la fonction `cube()` en entrant les commandes `"print cube(9)"` ou plus simplement encore `"cube(9)"`, comme nous l'avons fait pour la fonction `jury()`.

2.5.4 Utilisation des fonctions dans un script

Pour cette première approche des fonctions, nous n'avons utilisé jusqu'ici que le mode interactif de l'interpréteur Python. Il est bien évident que les fonctions peuvent aussi s'utiliser dans des scripts. Veuillez donc essayer vous-même le petit programme ci-dessous, lequel calcule le volume d'une sphère à l'aide de la formule que vous connaissez certainement : $V = \frac{4}{3}\pi R^3$

```
def cube(n):
    return n**3

def volumeSphere(r):
    return 4 * 3.1416 * cube(r) / 3

r = input('Entrez la valeur du rayon : ')
print 'Le volume de cette sphère vaut', volumeSphere(r)
```

A bien y regarder, ce programme comporte trois parties : les deux fonctions `cube()` et `volumeSphere()`, et ensuite le corps principal du programme.

Dans le corps principal du programme, il y a un appel de la fonction `volumeSphere()`.

A l'intérieur de la fonction `volumeSphere()`, il y a un appel de la fonction `cube()`.

Notez bien que les trois parties du programme ont été disposées dans un certain ordre : d'abord la définition des fonctions, et ensuite le corps principal du programme. Cette disposition est nécessaire, parce que l'interpréteur exécute les lignes d'instructions du programme l'une à la suite de l'autre, dans l'ordre où elles apparaissent dans le code source. Dans le script, **la définition des fonctions doit donc précéder leur utilisation**.

Pour vous en convaincre, intervertissez cet ordre (en plaçant par exemple le corps principal du programme au début), et prenez note du type de message d'erreur qui est affiché lorsque vous essayez d'exécuter le script ainsi modifié.

En fait, le corps principal d'un programme Python constitue lui-même une fonction un peu particulière, qui est toujours reconnue dans le fonctionnement interne de l'interpréteur sous le nom réservé `__main__` (le mot `"main"` signifie "principal", en anglais. Il est encadré par des caractères "souligné" en double, pour éviter toute confusion avec d'autres symboles). L'exécution d'un script commence toujours avec la première

[13] A titre purement indicatif, sachez que `return` peut également être utilisé sans aucun argument, à l'intérieur d'une fonction, pour provoquer sa fermeture immédiate. La valeur retournée dans ce cas est `None` (valeur nulle, chaîne ou liste vide).

instruction du module `__main__`, où qu'elle puisse se trouver dans le listing (en général ce sera donc plutôt vers la fin).

Les instructions qui suivent sont alors exécutées l'une après l'autre, dans l'ordre, jusqu'au premier appel de fonction. Un appel de fonction est comme un détour dans le flux de l'exécution : au lieu de passer à l'instruction suivante, l'interpréteur exécute la fonction appelée, puis revient au programme appelant pour continuer le travail interrompu.

Dans notre exemple, la fonction principale `__main__` appelle une première fonction qui elle-même en appelle une deuxième. Cette situation est très fréquente en programmation. Si vous voulez comprendre correctement ce qui se passe dans un programme, vous devez donc apprendre à lire un script, non pas de la première à la dernière ligne, mais plutôt en suivant un cheminement analogue à ce qui se passe lors de l'exécution de ce script. Cela signifie concrètement que vous devrez souvent analyser un script en commençant par les dernières lignes !

2.5.5 Valeurs par défaut pour les paramètres

Dans la définition d'une fonction, il est possible (et souhaitable !) de définir un argument par défaut pour chacun des paramètres. On obtient ainsi une fonction qui peut être appelée avec une partie seulement des arguments attendus. Exemple :

```
>>> def question(annonce, essais=4, please='Oui ou non, s.v.p.!'):
...     while essais >0:
...         reponse = raw_input(annonce)
...         if reponse in ('o', 'oui','O','Oui','OUI'):
...             return 1
...         if reponse in ('n','non','N','Non','NON'):
...             return 0
...         print please
...         essais = essais-1
...
```

Cette fonction peut être appelée de différentes façons, telles par exemple (essayez et décortiquez par vous même ces exemples) :

- `question('Voulez-vous vraiment terminer ? ')`
- `question('Faut-il effacer ce fichier ? ', 3)`
- `question('Avez-vous compris ? ', 2, 'Vous devez répondre par oui ou par non !')`

2.5.6 Arguments avec étiquettes

Dans la plupart des langages de programmation, les arguments que l'on fournit lors de l'appel d'une fonction doivent être fournis exactement dans le même ordre que celui des paramètres correspondants dans la définition de la fonction.

Python autorise cependant une souplesse beaucoup plus grande. Si les paramètres annoncés dans la définition de la fonction ont reçu chacun une valeur par défaut, sous la forme déjà décrite ci-dessus, on peut faire appel à la fonction en fournissant les arguments correspondants dans n'importe quel ordre, à la condition de désigner nommément les paramètres correspondants. Exemple :

```
>>> def oiseau(voltage=100, etat='allumé', action='danser la java'):
...     print 'Ce perroquet ne pourra pas', action
...     print 'si vous le branchez sur', voltage, 'volts !'
...     print "L'auteur de ceci est complètement", etat
...
```

```
>>> oiseau(etat='givré', voltage=250, action='vous approuver')
Ce perroquet ne pourra pas vous approuver
si vous le branchez sur 250 volts !
L'auteur de ceci est complètement givré

>>> oiseau()
Ce perroquet ne pourra pas danser la java
si vous le branchez sur 100 volts !
L'auteur de ceci est complètement allumé
```

Index

affectation, 7, 8, 15
assignation, voir affectation

bloc d'instructions, 13
boucle, 16

composition, 10
condition booléenne, 11

expression, 9

fonction, 22
- appel, 23
- définition, 23
- paramètre, 24

indentation, 11, 14
instructions
- def, 23
- elif, 11
- else, 11
- for, 18
- if, 10
- input, 22
- print, 8
- range, 18
- while, 16

itération, voir boucle

mots réservés, 7

opérateurs, 9
- booléens, 12
- comparaison, 12
- modulo, 9
- règles de priorité, 9

procédure, voir fonction

séquence (d'instructions), 10
script, 19
suite (d'instructions), voir séquence

variable, 6
- globale, 27
- locale, 27
- nom, 6
- type, 8
- valeur, 6
- visibilité, 27