2

Premiers pas

La programmation est donc l'art de commander à un ordinateur de faire exactement ce que vous voulez, et Python compte parmi les langages qu'il est capable de comprendre pour recevoir vos ordres. Nous allons essayer cela tout de suite avec des ordres très simples concernant des nombres, puisque ce sont les nombres qui constituent son matériau de prédilection. Nous allons lui fournir nos premières « instructions », et préciser au passage la définition de quelques termes essentiels du vocabulaire informatique, que vous rencontrerez constamment dans la suite de cet ouvrage.

Remarque préliminaire : Comme nous l'avons expliqué dans la préface (voir : Versions du langage, page 10), nous avons pris le parti d'utiliser dans ce cours la nouvelle version 3 de Python, laquelle a introduit quelques changements syntaxiques par rapport aux versions précédentes. Dans la mesure du possible, nous vous indiquerons ces différences dans le texte, afin que vous puissiez sans problème analyser ou utiliser d'anciens programmes écrits pour Python 1 ou 2.

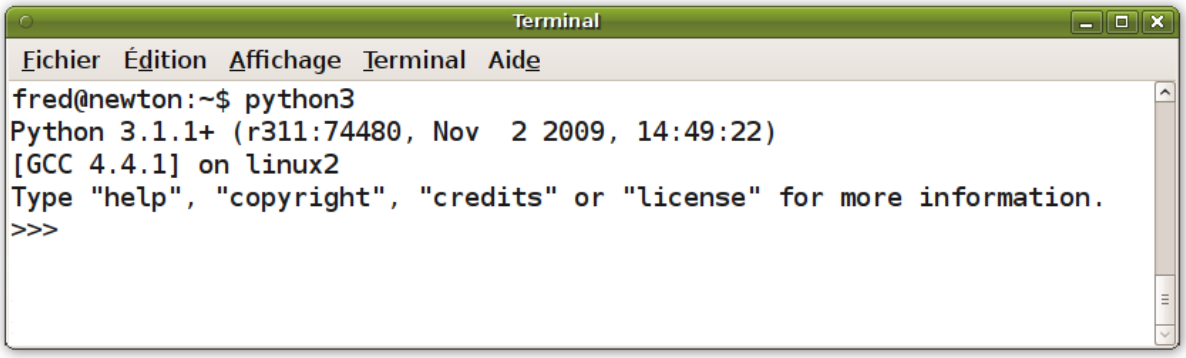
Calculer avec Python

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

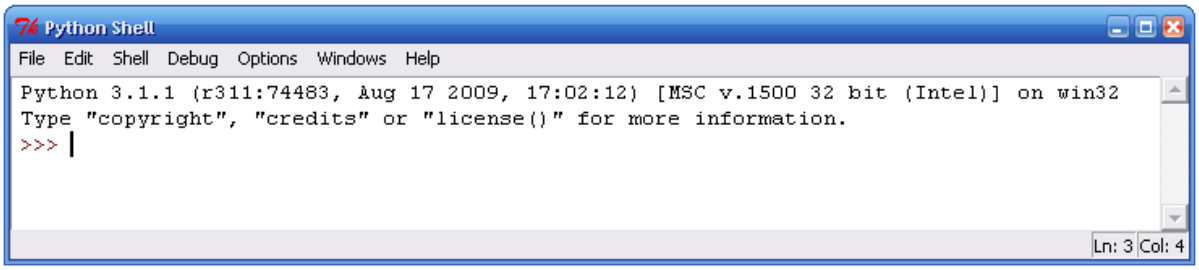
L'interpréteur peut être lancé directement depuis la ligne de commande (dans un « shell » *Linux*, ou bien dans une fenêtre *DOS* sous *Windows*) : il suffit d'y taper la commande **python3** (en supposant que le logiciel lui-même ait été correctement installé, et qu'il s'agisse d'une des dernières versions de Python), ou **python** (si la version de Python installée sur votre ordinateur est antérieure à la version 3.0).

Si vous utilisez une interface graphique telle que *Windows*, *Gnome*, *WindowMaker* ou *KDE*, vous préférerez vraisemblablement travailler dans une « fenêtre de terminal », ou encore dans un environnement de travail spécialisé tel que *IDLE*. Voici par exemple ce qui apparaît dans une fenêtre de terminal *Gnome* (sous *Ubuntu Linux*)[5] :

[5] Sous *Windows*, vous aurez surtout le choix entre l'environnement *IDLE* développé par Guido Van Rossum, auquel nous donnons nous-même la préférence, et *PythonWin*, une interface de développement développée par Mark Hammond. D'autres environnements de travail plus sophistiqués existent aussi, tels l'excellent *Boa Constructor* par exemple (qui fonctionne de façon très similaire à *Delphi*), mais nous estimons qu'ils ne conviennent guère aux débutants. Pour tout renseignement complémentaire, veuillez consulter le site Web de Python.



Avec *IDLE* sous *Windows*, votre environnement de travail ressemblera à celui-ci :



Les trois caractères « supérieur à » 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 (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          # attention : ceci fonctionnerait différemment sous Python 2

>>> 20 // 3
```

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

Sous *Linux*, nous préférons personnellement travailler dans une simple fenêtre de terminal pour lancer l'interpréteur Python ou l'exécution des scripts, et faire appel à un éditeur de texte ordinaire tel que *Gedit*, *Kate*, ou un peu plus spécialisé comme *Geany* pour l'édition de ces derniers.

> *Sous Python 3, l'opérateur de division / effectue une division réelle. Si vous souhaitez obtenir une division entière (c'est-à-dire dont le résultat - tronqué - ne peut être qu'un entier), vous devez utiliser l'opérateur //. Veuillez bien noter que ceci est l'un des changements de syntaxe apportés à la version 3 de Python, par rapport aux versions précédentes. Si vous utilisez l'une de ces versions, sachez que l'opérateur / y effectue par défaut une division entière, si on lui fournit des arguments qui sont eux-mêmes des entiers, et une division réelle, si au moins l'un des arguments est un réel. Cet ancien comportement de Python a été heureusement abandonné car il pouvait parfois conduire à des bugs difficilement repérables.*

```
>>> 20.5 / 3

>>> 8,7 / 5          # Erreur !
```

Veuillez remarquer au passage ce qui est la règle dans tous les langages de programmation, à savoir que les conventions mathématiques de base sont celles qui sont en vigueur dans les pays anglophones : le séparateur décimal y est donc toujours un point, et non une virgule comme chez nous. Notez aussi que dans le monde de l'informatique, les nombres réels sont souvent désignés comme des nombres « à virgule flottante » (*floating point numbers*).

Données et variables

Nous aurons l'occasion de détailler plus loin les différents types de données numériques. Mais avant cela, nous pouvons dès à présent aborder un concept de grande importance.

L'essentiel du travail effectué par un programme d'ordinateur consiste à manipuler des *données*. Ces données peuvent être très diverses (tout ce qui est *numérisable*, en fait[6]), mais dans la mémoire de l'ordinateur elles se ramènent toujours en définitive à *une suite finie de nombres binaires*.

Pour pouvoir accéder aux données, le programme d'ordinateur (quel que soit le langage dans lequel il est écrit) 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 *référence* désignant une adresse mémoire, c'est-à-dire un emplacement précis dans la mémoire vive.

À cet emplacement est stockée une *valeur* bien déterminée. C'est la donnée proprement dite, qui est donc stockée sous la forme d'une suite de nombres binaires, mais qui n'est pas nécessairement un nombre aux yeux du langage de programmation utilisé. Cela peut être en fait à peu près n'importe quel « objet » susceptible d'être placé dans la mémoire d'un ordinateur, par exemple : un nombre entier, un nombre réel, un nombre complexe, un vecteur, une chaîne de caractères typographiques, un tableau, une fonction, etc.

---

[6] *Que peut-on numériser au juste ?* Voilà une question très importante, qu'il vous faudra débattre dans votre cours d'informatique générale.

Pour distinguer les uns des autres ces divers contenus possibles, le langage de programmation fait usage de différents types de variables (le type *entier*, le type *réel*, le type *chaîne de caractères*, le type *liste*, etc.). Nous allons expliquer tout cela dans les pages suivantes.

Noms de variables et mots réservés

Les noms de variables sont des noms que vous choisissez vous-même assez librement. Efforcez-vous cependant de bien les choisir : de préférence assez courts, mais aussi explicites que possible, de manière à exprimer clairement ce que la variable est censée contenir. Par exemple, des noms de variables tels que *altitude* , *altit* ou *alt* conviennent mieux que x pour exprimer une altitude.

Un bon programmeur doit veiller à ce que ses lignes d'instructions soient faciles à lire.

Sous Python, les noms de variables doivent en outre obéir à quelques règles simples :

- Un nom de variable est une séquence de lettres (a → z , A → Z) et de chiffres (0 → 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 (les caractères majuscules et minuscules sont distingués).
  Attention : Joseph, joseph, JOSEPH sont donc des variables différentes. Soyez attentifs !

Prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules (y compris la première lettre[7]). Il s'agit d'une simple convention, mais elle est largement respectée. N'utilisez les majuscules qu'à l'intérieur même du nom, pour en augmenter éventuellement la lisibilité, comme dans *tableDesMatieres*.

En plus de ces règles, il faut encore ajouter que vous ne pouvez pas utiliser comme nom de variables les 33 « mots réservés » ci-dessous (ils sont utilisés par le langage lui-même) :

and	as	assert	break	class	continue	def
del	elif	else	except	False	finally	for
from	global	if	import	in	is	lambda
None	nonlocal	not	or	pass	raise	return
True	try	while	with	yield		

[7] Les noms commençant par une majuscule ne sont pas interdits, mais l'usage veut qu'on le réserve plutôt aux variables qui désignent des *classes* (le concept de classe sera abordé plus loin dans cet ouvrage). Il arrive aussi que l'on écrive entièrement en majuscules certaines variables que l'on souhaite traiter comme des pseudo-constantes (c'est-à-dire des variables que l'on évite de modifier au cours du programme).

Affectation (ou assignation)

Nous savons désormais comment choisir judicieusement un nom de variable. Voyons à présent comment nous pouvons *définir* une variable et lui *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 sa valeur (son contenu).

En Python comme dans de nombreux autres langages, l'opération d'affectation est représentée par le signe *égale*[8] :

```
>>> n = 7                       # définir n et lui donner 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
```

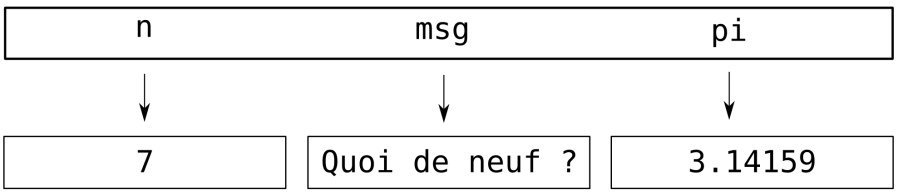
Les exemples ci-dessus illustrent des instructions d'affectation Python tout à fait classiques. Après qu'on les ait exécutées, il existe dans la mémoire de l'ordinateur, à des endroits différents :

- 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 chacune de réaliser plusieurs opérations dans la mémoire de l'ordinateur :

- créer et mémoriser un **nom de variable** ;
- lui attribuer un **type** bien déterminé (ce point sera explicité à la page suivante) ;
- créer et mémoriser une **valeur** particulière ;
- établir un **lien** (par un système interne de pointeurs) entre le nom de la variable et l'emplacement mémoire de la valeur correspondante.

On peut mieux se représenter tout cela par un diagramme d'état tel que celui-ci :



Les trois noms de variables sont des *références*, mémorisées dans une zone particulière de la mémoire que l'on appelle espace de noms, alors que les valeurs correspondantes sont situées ailleurs, dans des emplacements parfois fort éloignés les uns des autres. Nous aurons l'occasion de préciser ce concept plus loin dans ces pages.

[8] Il faut bien comprendre qu'il ne s'agit en aucune façon d'une égalité, et que l'on aurait très bien pu choisir un autre symbolisme, tel que **n ← 7** par exemple, comme on le fait souvent dans certains pseudo-langages servant à décrire des algorithmes, pour bien montrer qu'il s'agit de relier un contenu (la valeur 7) à un contenant (la variable n).

Afficher la valeur d'une variable

À la suite de l'exercice ci-dessus, nous disposons donc des trois variables **n**, **msg** et **pi**.
Pour afficher leur valeur à l'écran, il existe deux possibilités. La première consiste à entrer au clavier le nom de la variable, puis <*Enter*>. Python répond en affichant la valeur correspondante :

```
>>> n
7
>>> msg
'Quoi de neuf ?'
>>> pi
3.14159
```

Il s'agit cependant là d'une fonctionnalité secondaire de l'interpréteur, qui est destinée à vous faciliter la vie lorsque vous faites de simples exercices à la ligne de commande. À l'intérieur d'un programme, vous utiliserez toujours la fonction `print()`[9] :

```
>>> print(msg)
Quoi de neuf ?
>>> print(n)
7
```

Remarquez la subtile différence dans les affichages obtenus avec chacune des deux méthodes. La fonction `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 apostrophes afin de vous rappeler que la variable traitée est du type « chaîne de caractères » : nous y reviendrons.

> *Dans les versions de Python antérieures à la version 3.0, le rôle de la* ***fonction*** `print()` *était assuré par une* ***instruction*** `print` *particulière, faisant d'ailleurs l'objet d'un mot réservé (voir page 14). Cette instruction s'utilisait sans parenthèses. Dans les exercices précédents, il fallait donc entrer « `print n` » ou « `print msg` ». Si vous essayez plus tard de faire fonctionner sous Python 3, des programmes écrits dans l'une ou l'autre version ancienne, sachez donc que vous devrez ajouter des parenthèses après chaque instruction* `print` *afin de convertir celle-ci en fonction (des utilitaires permettent de réaliser cela automatiquement).*
> *Dans ces mêmes versions anciennes, les chaînes de caractères étaient traitées différemment (nous en reparlerons en détail plus loin). Suivant la configuration de votre ordinateur, vous pouviez alors parfois rencontrer quelques effets bizarres avec les chaînes contenant des caractères accentués, tels que par exemple :*
> `>>> msg = "Mon prénom est Chimène"`
> `>>> msg`
> `'Mon pr\xe9nom est Chim\xe8ne'`
> *Ces bizarreries appartiennent désormais au passé, mais nous verrons plus loin qu'un programmeur digne de ce nom doit savoir de quelle manière sont encodés les caractères typographiques rencontrés dans différentes sources de*

[9] les *fonctions* seront définies en détail dans les chapitres 6 et 7 (voir page 49).

> *données, car les normes définissant ces encodages ont changé au cours des années, et il faut connaître les techniques qui permettent de les convertir.*

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 rattache à une famille particulière de langages où l'on trouve aussi par exemple *Lisp*, *Scheme*, et quelques autres. On dira à ce sujet que *le typage des variables 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 faut toujours – par des instructions distinctes – d'abord déclarer (définir) le nom et le type des variables, et ensuite seulement leur assigner un contenu, lequel doit bien entendu être compatible avec le type déclaré.

Le typage statique est préférable dans le cas des langages compilés, parce qu'il permet d'optimiser l'opération de compilation (dont le résultat est un code binaire « figé »).

Le typage dynamique quant à lui permet d'écrire plus aisément des constructions logiques de niveau élevé (métaprogrammation, réflexivité), en particulier dans le contexte de la programmation orientée objet (polymorphisme). Il facilite également l'utilisation de structures de données très riches telles que les listes et les dictionnaires.

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 des *affectations parallèles* 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**.

> *Les francophones que nous sommes avons pour 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'est-à-dire le point décimal. La virgule, quant à elle, est très généralement utilisée 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.*

Exercices

2.1 Décrivez le plus clairement et le plus complètement possible ce qui se passe à chacune des trois lignes de l'exemple ci-dessous :

```
>>> largeur = 20
>>> hauteur = 5 * 9.3
>>> largeur * hauteur
930
```

2.2 Assignez les valeurs respectives 3, 5, 7 à trois variables a, b, c.
Effectuez l'opération `a - b//c`. Interprétez le résultat obtenu.

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 les symboles spéciaux 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. À cette valeur, il attribue automatiquement un type, lequel dépend de ce qu'il y a 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. Nous avons déjà signalé l'existence de l'opérateur de division entière `//`. Il faut encore 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 caractère typographique **%**. Cet opérateur fournit *le reste de la division entière* d'un nombre par un autre. Essayez par exemple :

```
>>> 10 % 3       # (et prenez note de ce qui se passe !)
>>> 10 % 5
```

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.

Exercice

2.3 Testez les lignes d'instructions suivantes. Décrivez dans votre cahier ce qui se passe :

```
>>>  r , pi = 12, 3.14159
>>>  s = pi * r**2
>>>  print(s)
>>>  print(type(r), type(pi), type(s))
```

Quelle est, à votre avis, l'utilité de la *fonction* **type()** ?
(Note : les *fonctions* seront décrites en détail aux chapitres 6 et 7.)

Conversion de type

Il est possible de convertir certains types. Par exemple, la fonction **int** permet de transformer un réel en un entier en supprimant la partie à droite de la virgule.

```
>>> a=-1.93
>>> int(a)
-1
```

Priorité 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, les 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* et *D* pour *multiplication* et *division*, qui ont la même priorité. Elles sont évaluées avant *l'addition A* et *la soustraction S*, lesquelles sont donc effectuées en dernier lieu.
  Ainsi **2*3-1 = 5** (plutôt que 4), et **2/3-1 = -0.3333...** (plutôt que 1.0).
- 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 que l'opérateur // effectue une division entière, et vérifiez en effectuant 59*(100//60)).

Composition

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

Or l'une des grandes forces d'un langage de programmation de haut niveau est qu'il permet de construire des instructions complexes par assemblage de fragments divers. Ainsi par exemple, si vous savez comment additionner deux nombres et comment afficher une valeur, vous pouvez combiner ces deux instructions en une seule :

```
>>> print(17 + 3)
>>> 20
```

Cela n'a l'air de rien, mais cette fonctionnalité qui paraît si évidente va vous permettre de programmer des algorithmes complexes de façon claire et concise. Exemple :

```
>>> h, m, s = 15, 27, 34
>>> print("nombre de secondes écoulées depuis minuit = ", h*3600 + m*60 + s)
```

Attention, cependant : il y a une limite à ce que vous pouvez combiner ainsi :

Dans une expression, ce que vous placez à la gauche du signe égale doit toujours être une variable, et non une expression. Cela provient du fait que le signe égale n'a pas ici la même signification qu'en mathématique : comme nous l'avons déjà signalé, 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 un peu 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 auparavant, il nous faut encore aborder un autre concept de grande importance.