2 Notions élémentaires

1 Installation de Python

Avant de commencer à programmer, il est important d'installer l'interpréteur Python, c'est-à-dire le programme qui comprend les instructions que vous allez écrire dans le langage Python. Sans interpréteur, votre ordinateur ne comprendra pas ces instructions et vos programmes ne fonctionneront pas.

L'installation de l'interpréteur pour Windows et Mac OS se fait à partir du site officiel de Python, www.python.org. En arrivant sur ce site, vous verrez au sommet une liste de liens (About, Downloads, Documentation, ...).

FIGURE 2.1 – Fenêtre de téléchargement pour système Mac OS X

Cliquez sur le lien **Downloads** afin d'obtenir la version de Python 3 désirée. Pour ce cours, nous utiliserons la version 3.7 qui peut être téléchargée via l'onglet **All releases**. Pour Windows, choisissez la version ***Windows x86-64 executable installer*** et cliquez dessus pour télécharger Python 3.7.

Double-cliquez sur le fichier que vous venez de télécharger afin de procéder à l'installation de Python (si l'installation ne se lance pas, faite un clic droit sur l'icône du fichier et choisissez **Install**). Une fois, l'installateur activé, cliquez sur le bouton ***Next*** et acceptez les choix que le programme vous propose. Une fois l'installation terminée, cliquez sur ***Finish***.

L'installation pour Mac OS est analogue si ce n'est qu'en lieu et place du fichier .exe, vous devez télécharger le fichier .pkg. Le lien de ce fichier est intitulé ***macOS 64-bit installer*** sur la page de téléchargement du site. Finalement, si votre système d'exploitation est Ubuntu, vous pouvez installer Python en ouvrant une fenêtre terminal (cliquez sur Applications > Accessories > Terminal) et en y entrant la commande `sudo apt-get install python3` suivie de <`Enter`>. Le mot de passe de l'administrateur de votre ordinateur vous sera alors demandé avant que ne commence l'installation.

En cas de difficulté dans l'installation de Python, n'hésitez pas à visionner la vidéo d'installation proposée à l'adresse inventwithpython.com/videos/.

2 La console Python

Python présente la particularité de pouvoir être utilisé de deux manières différentes :

- en **mode interactif,** l'utilisateur peut dialoguer directement avec Python depuis le clavier grâce à la boucle d'évaluation de l'interpréteur qui exécute chaque instruction immédiatement.
- en **mode script,** le programmeur peut enregistrer un ensemble de commandes Python dans un fichier (appelé **script**) qu'il pourra par la suite exécuter à l'aide d'une touche spéciale.

Si votre système d'exploitation est *Windows*, l'interpréteur peut être lancé en choisissant **Start > Tous les programmes > Python 3.7 > IDLE (Python GUI).** IDLE (***I****nteractive* ***D****eve****L****opment* ***E****nvironment*) est un environnement développement auquel nous donnerons la préférence dans ces notes.

Si vous êtes sous Linux, vous pouvez lancer IDLE depuis la ligne de commande en tapant `idle3` ou depuis le menu **Applications > Programming > IDLE 3** ou **Applications > Programmation > IDLE 3** :

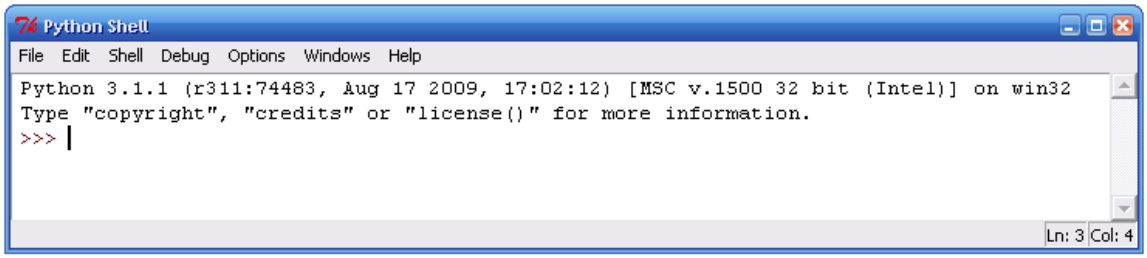
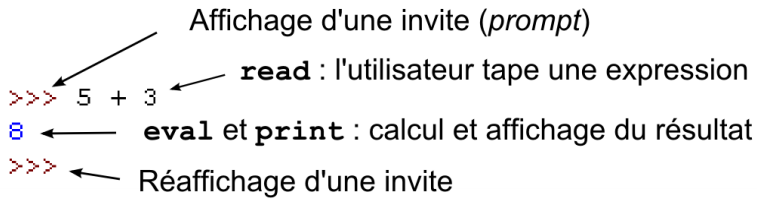


FIGURE 2.2 – Console interactive IDLE sous Windows

La fenêtre qui apparaît lorsque vous ouvrez IDLE pour la première fois est appelée **console interactive**. Il s'agit d'un programme permettant de taper des instructions Python qui seront directement transmises à l'interpréteur pour être évaluées. La console étant interactive, le résultat de cette évaluation est alors directement retourné. Les trois caractères ">>>" constituent le signal d'**invite** ou **prompt principal**, lequel indique à l'utilisateur que Python est prêt à exécuter une commande :



Vous pouvez tout de suite utiliser l'interpréteur comme une simple calculatrice de bureau. Veuillez donc vous-même tester les commandes ci-dessous :

```
>>> 2 - 9          # les espaces sont optionnels
7
>>> 7 + 3 * 4      # la hiérarchie des opérations mathématiques est-elle respectée ?
19
>>> (7+3)*4
40
>>> 20 / 3         # l'opérateur de division / effectue une division réelle
6.666666666666667
>>> 20 // 3        # l'opérateur de division // effectue une division entière
6
>>> 20.5 / 3       # le séparateur décimal peut-il être une virgule ?
6.833333333333333
```

Comme vous pouvez le constater, les **opérateurs** arithmétiques pour l'addition, la soustraction, la multiplication et la division sont respectivement +, -, * et /. Vous remarquerez au passage que les parenthèses ont la fonction attendue et que le séparateur décimal est le point et non la virgule.

A la différence d'une vulgaire calculatrice, l'interpréteur Python permet de traiter du texte. Traditionnellement, le premier programme gérant du texte qu'un utilisateur écrit dans un nouveau langage de programmation est le programme qui affiche `"Hello, World!"` à l'écran. En Python, ce programme est très simple :

```
>>> print("Hello, World!")
Hello, World!
```

La **fonction**[1] `print` permet d'afficher la valeur qui lui est transmise entre parenthèses. Notons que les guillemets marquent le début et la fin du texte à afficher mais qu'ils n'apparaissent pas à l'affichage.

1. Une **fonction** est une sorte de petit programme à l'intérieur de votre programme. Elle contient des lignes de code qui doivent être exécutées lorsque la fonction est appelée. Python intègre un certain nombres de fonctions que l'utilisateur peut appeler.

3 Données et types

L'essentiel du travail effectué par un programme consiste à manipuler des **données**. Ces éléments de base peuvent être très divers mais, dans la mémoire de l'ordinateur, ils se ramènent toujours en définitive à une suite finie de nombres binaires. Jusqu'à présent, nous avons rencontré des données de différents **types** :

- les **entiers** (`int`) : -4, 0, 99
- les **nombres à virgules flottantes** (`float`) : 20.5, 10., .001
- les **chaînes de caractères** (`string`) : "Hello, World", 'Oui'

Comme nous le verrons par la suite, il existe d'autres types de données gérés par Python. Par ailleurs, si vous doutez du type d'une donnée, vous pourrez recourir à la fonction **type** :

```
>>> type(10)
<class 'int'>
>>> type(10.)                    # le type float est caractérisé par un point décimal
<class 'float'>
>>> type('10')                   # le type string est caractérisé par les apostrophes
<class 'str'>
```

En outre, certaines fonctions permettent de transformer une donnée numérique de type **integer** en une donnée de type **float** (`float`) ou même de type **string** (`str`) et inversement comme l'illustre l'exemple suivant :

```
>>> float(10)
10.0
>>> str(10)
'10'
>>> int(10.6)                    # la conversion de float à int provoque une troncation
10
>>> int('10')                    # selon le contexte, un entier peut être considéré
10                               # comme un nombre ou comme une chaîne de caractères
```

3.1 Type integer

Les données de type **integer** (`int`) regroupent tous les nombres entiers. Python est capable de traiter des nombres entiers de taille illimitée. Cet état de fait pourrait cependant intriguer ceux d'entre vous qui s'interrogent sur la représentation interne des entiers dans un ordinateur. Vous savez probablement en effet que le *cœur* de celui-ci est constitué par un circuit intégré électronique (une *puce* de silicium) à très haut degré d'intégration, qui peut effectuer plus d'un milliard d'opérations en une seule seconde, mais seulement sur des nombres binaires de taille limitée : 32 bits actuellement. Or, la gamme de valeurs décimales qu'il est possible d'encoder sous forme de nombres binaires de 32 bits s'étend de -2147483648 à +2147483647.

Les opérations effectuées sur des entiers compris entre ces deux limites sont donc très rapides, parce que le processeur est capable de les traiter directement. En revanche, lorsqu'il est question de traiter des entiers plus grands, les interpréteurs et compilateurs doivent effectuer un gros travail de codage/décodage, afin de ne présenter en définitive au processeur que des opérations binaires sur des nombres entiers de 32 bits au maximum. Ainsi, les calculs impliquant de grands nombres devront être décomposés par l'interpréteur Python en calculs sur des nombres plus simples, ce qui pourra nécessiter un temps de traitement plus long.

3.2 Type float

Une donnée numérique contenant dans sa formulation un élément tel qu'un point décimal ou un exposant [2] de 10 est du type *nombre réel*, ou *nombre à virgule flottante*, désigné en anglais par l'expression **floating point number**. Ce type sera pour cette raison appelé type `float` sous Python :

```
3.14        10.        .001        1e100        3.14e-10
```

Ce type autorise les calculs sur de très grands ou très petits nombres, avec un degré de précision constant. En fait, le type `float` permet de manipuler des nombres (positifs ou négatifs) compris entre 10^{-308} et 10^{308} avec une précision de 12 chiffres significatifs. Ces nombres sont encodés sur 8 octets (64 bits) en mémoire : une partie du code correspond aux 12 chiffres significatifs, et une autre à l'ordre de grandeur (exposant de 10).

2. La notation scientifique "e+n" signifie "fois dix à l'exposant n".

3.3 Type string

Un programme peut traiter des caractères alphabétiques, des mots, des phrases, ou des suites de symboles quelconques. Il existe donc des structures de données particulières appelées **chaîne de caractères** ou **string** en anglais. En Python, on peut délimiter une telle suite de caractères par des apostrophes ou des guillemets :

```
>>> print('Que faites-vous ?')
Que faites-vous ?
>>> print("Etes-vous à l'école?")
Etes-vous à l'école ?
>>> print('"Oui, je suis en cours", répondit-il')
"Oui, je suis en cours", répondit-il
```

Remarquez l'utilisation des guillemets pour délimiter une chaîne dans laquelle il y a des apostrophes, ou l'utilisation des apostrophes pour délimiter une chaîne qui contient des guillemets.

Le caractère spécial « \ » (**antislash**) permet quelques subtilités complémentaires :

- Il permet d'écrire sur plusieurs lignes une commande qui serait trop longue pour tenir sur une seule.
- À l'intérieur d'une chaîne de caractères, l'**antislash** permet d'insérer certains codes spéciaux :

Séquence d'échappement	Description
\n	Saut de ligne
\t	Tabulateur horizontal
\r	Retour de ligne
\\	Antislash
\"	Guillemet
\'	Apostrophe

```
>>> print('"N\'est-ce pas ?" répondit-elle.')
"N'est-ce pas ?" répondit-elle.

>>> print("Ceci est une chaîne plutôt longue\n contenant plusieurs lignes \
... de texte (Ceci fonctionne\n de la même façon en C/C++.\n\
...     Notez que les blancs en début\n de ligne sont significatifs.\n")
Ceci est une chaîne plutôt longue
contenant plusieurs lignes de texte (Ceci fonctionne
de la même façon en C/C++.
   Notez que les blancs en début
de ligne sont significatifs.
```

Pour insérer plus aisément des caractères spéciaux ou « exotiques » dans une chaîne, sans faire usage de l'**antislash**, ou pour faire accepter l'**antislash** lui-même dans la chaîne, on peut encore délimiter la chaîne à l'aide de **triples guillemets** ou de **triples apostrophes** :

```
>>> print("""Usage: trucmuch [OPTIONS]
    { -h
      H hote
    }""")
Usage: trucmuch [OPTIONS]
    { -h
      H hote
    }
```

Les chaînes de caractères constituent un cas particulier d'un type de données plus général que l'on appelle des **données composites**. Une donnée composite est une entité qui rassemble dans une seule structure un ensemble d'entités plus simples : dans le cas d'une chaîne de caractères, par exemple, ces entités plus simples sont évidemment les caractères eux-mêmes. La technique utilisée par Python pour mémoriser et traiter les caractères typographiques dans la mémoire de l'ordinateur permet de mélanger dans une même chaîne des caractères latins, grecs, cyrilliques, arabes, ainsi que des symboles mathématiques, etc.

En fait, sous Python 3, les chaînes de caractères sont des chaînes **Unicode**[3], ce qui signifie que les identifiants numériques de leurs caractères sont uniques (il ne peut exister qu'un seul caractère typographique pour chaque code) et universels (les identifiants choisis couvrent la gamme complète de tous les caractères utilisés dans les différentes langues du monde entier).

À l'origine du développement des technologies informatiques, alors que les capacités de mémorisation des ordinateurs étaient encore assez limitées, on n'imaginait pas que ceux-ci seraient utilisés un jour pour traiter d'autres textes que des communications techniques, essentiellement en anglais. Il semblait donc tout-à-fait raisonnable de ne prévoir pour celles-ci qu'un jeu de caractères restreint, de manière à pouvoir représenter chacun de ces caractères avec un petit nombre de bits, et ainsi occuper aussi peu d'espace que possible dans les coûteuses unités de stockage de l'époque. Le jeu de caractères **ASCII**[4] fut donc choisi en ce temps là, avec l'estimation que 128 caractères suffiraient (à savoir le nombre de combinaisons possibles pour des groupes de 7 bits[5]). En l'étendant par la suite à 256 caractères, on put l'adapter aux exigences du traitement des textes écrits dans d'autres langues que l'anglais, mais au prix d'une dispersion des normes. Ainsi par exemple, la norme **ISO-8859-1 (latin-1)** codifie tous les caractères accentués du français ou de l'allemand (entre autres), mais aucun caractère grec, hébreu ou cyrillique. Pour ces langues, il faudra respectivement utiliser les normes **ISO-8859-7, ISO-8859-8, ISO-8859-5** et d'autres normes encore pour l'arabe, le tchèque, le hongrois...

L'intérêt résiduel de ces normes anciennes réside dans leur simplicité. Elles permettent en effet aux développeurs d'applications informatiques de considérer que **chaque caractère typographique est assimilable à un octet**, et que par conséquent une chaîne de caractères n'est rien d'autre qu'une séquence d'octets. C'est ainsi que fonctionnait l'ancien type de données **string** de Python (dans les versions antérieures à la version 3.0).

Toutefois, les applications informatiques modernes ne peuvent plus se satisfaire de ces normes étriquées. Il faut désormais pouvoir encoder, dans un même texte, tous les caractères de n'importe quel alphabet de n'importe quelle langue. Une organisation internationale a donc été créée : le **Consortium Unicode**, laquelle a effectivement développé une norme universelle sous le nom de **Unicode**. Cette nouvelle norme vise à donner à tout caractère de n'importe quel système d'écriture de langue un nom et un identifiant numérique, et ce de manière unifiée, quelle que soit la plate-forme informatique ou le logiciel.

Une difficulté se présente, cependant. Se voulant universelle, la norme **Unicode** doit attribuer un identifiant numérique **différent** à plusieurs dizaines de milliers de caractères. Tous ces identifiants ne pourront évidemment pas être encodés sur un seul octet. À première vue, ils serait donc tentant de décréter qu'à l'avenir, chaque caractère devra être encodé sur deux octets (cela ferait 65536 possibilités), ou trois (16777216 possibilités) ou quatre (plus de 4 milliards de possibilités). Chacun de ces choix rigides entraîne cependant son lot d'inconvénients. Le premier, commun à tous, est que l'on perd la compatibilité avec la multitude de documents informatiques préexistants, qui ont été encodés aux normes anciennes, sur la base du paradigme *un caractère égale un octet*. Le second est lié à l'impossibilité de satisfaire deux exigences contradictoires : si l'on se contente de deux octets, on risque de manquer de possibilités pour identifier des caractères rares ou des attributs de caractères qui seront probablement souhaités dans l'avenir ; si l'on impose trois, quatre octets ou davantage, par contre, on aboutit à un monstrueux gaspillage de ressources : la plupart des textes courants se satisfaisant d'un jeu de caractères restreint, l'immense majorité de ces octets ne contiendraient en effet que des zéros.

Afin de ne pas se retrouver piégée dans un carcan de ce genre, la norme **Unicode** ne fixe aucune règle concernant le nombre d'octets ou de bits à réserver pour l'encodage. Elle spécifie seulement la valeur numérique de l'identifiant associé à chaque caractère. En fonction des besoins, chaque système informatique est donc libre d'encoder *en interne* cet identifiant comme bon lui semble, par exemple sous la forme d'un entier ordinaire. Comme tous les langages de programmation modernes, Python s'est donc pourvu d'un type de données *chaîne de caractères* (le type **string**) qui respecte scrupuleusement la norme **Unicode**, et la représentation *interne* des codes numériques correspondants est sans importance pour le programmeur.

Nous verrons un peu plus loin dans ce chapitre qu'il est effectivement possible de placer dans une chaîne de ce type un mélange quelconque de caractères issus d'alphabets différents (qu'il s'agisse de caractères `ASCII` standards, de caractères accentués, de symboles mathématiques ou de caractères grecs, cyrilliques, arabes, etc.), et que chacun d'eux est effectivement représenté *en interne* par un code numérique unique.

3. Dans les versions antérieures de Python, les chaînes de caractères de type **string** étaient en fait des séquences d'octets et il existait un deuxième type de chaîne, le type **unicode** pour traiter les chaînes de caractères au sens où nous l'entendons désormais.
4. ASCII = *American Standard Code for Information Interchange*
5. En fait, on utilisait déjà les octets à l'époque, mais l'un des bits de l'octet devait être réservé comme bit de contrôle pour les systèmes de rattrapage d'erreur. L'amélioration ultérieure de ces systèmes permit de libérer ce huitième bit pour y stocker de l'information utile : cela autorisa l'extension du jeu ASCII à 256 caractères (normes ISO-8859, etc.).

4 Variables et affectation

Pour pouvoir stocker et accéder aux données, un programme d'ordinateur fait abondamment usage d'un grand nombre de **variables** de différents types. Une variable apparaît dans un programme sous un **nom de variable**, mais pour l'ordinateur il s'agit d'une **référence** désignant une adresse mémoire. À cet emplacement de la mémoire vive est stockée une **valeur** bien déterminée. C'est la donnée proprement dite, stockée sous la forme d'une suite de nombres binaires, mais qui peut, selon son type, apparaître aux yeux du langage de programmation sous la forme d'un entier, d'un nombre réel, d'une chaîne de caractères, d'une fonction, etc.

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.

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 !

Prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules (y compris la première lettre[6]). 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		

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, l'opération d'affectation est représentée par le signe = (qui n'a rien à voir avec l'égalité mathématique !). Dans une affectation, le membre de gauche reçoit le membre de droite ce qui nécessite d'évaluer la valeur correspondant au membre de droite avant de l'affecter au membre de gauche :

```
>>> n = 7                      # définir n et lui donner la valeur 7
>>> msg = "Quoi de neuf ?"     # affecter la valeur "Quoi de neuf ?" à msg
>>> pi = 3 + 0.14159           # assigner la valeur 3.14159 à la variable pi
```

Les exemples ci-dessus illustrent des instructions d'affectation Python tout à fait classiques. Après qu'on les a 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`.

6. 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).

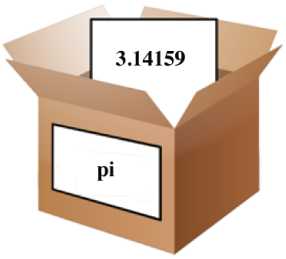
Les trois instructions d'affectation ci-dessus ont eu pour effet chacune de réaliser plusieurs opérations dans la mémoire de l'ordinateur :

- créer et mémoriser un **nom de variable** ;
- lui attribuer un **type** bien déterminé ;
- 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 une variable comme une boîte qui va contenir une valeur. L'affectation

```
pi = 3+0.14159
```

a pour effet de déposer dans cette boîte la valeur réelle 3.14159.



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.

4.1 Afficher la valeur d'une variable

À la suite de l'exemple 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 dans la console Python. À l'intérieur d'un script, vous utiliserez toujours la fonction `print` :

```
>>> 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 que la valeur de la variable, telle qu'elle a été encodée, alors que l'autre méthode affiche aussi des apostrophes afin de vous rappeler que la variable traitée est du type *chaîne de caractères*.

4.2 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'exemple précédent, les variables **n**, **msg** et **pi** ont été créées automatiquement chacune avec un type différent (`int` pour **n**, `string` pour **msg**, `float` pour **pi**).

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.

4.3 Variantes de l'affectation

Outre l'affectation simple présenté précédemment, Python permet certaines variantes d'affectation :

- Il est possible de **ré-affecter** une nouvelle valeur à une même variable autant de fois que souhaité :

```
>>> altitude = 320
>>> print(altitude)
320
>>> altitude = 375
>>> print(altitude)
375
>>> altitude = altitude + 50
>>> print(altitude)
425
```

  L'instruction `altitude = altitude + 50` **incrémente** le contenu de la variable `altitude` de 50 unités, ce qui signifie que l'on affecte à la variable `altitude` une nouvelle valeur, égale à la valeur précédente augmentée de 50 unités. Cette instruction d'incrémentation est souvent abrégée `altitude += 50`. Le symbole d'incrémentation `+=` ajoute la valeur de l'expression située à droite de `+=` à la valeur de la variable située à sa gauche et stocke le résultat dans la variable située de gauche. D'autres variantes existent :

Symbole	Exemple	Equivalent
+=	a += 1	a = a + 1
-=	a -= 1	a = a – 1
*=	a *= 2	a = a * 2
/=	a /= 2	a = a / 2

- Il est possible d'effectuer des **affectations multiples**, c'est-à-dire d'affecter une même valeur à plusieurs variables simultanément :

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

- Il est possible d'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`.

Finalement, rappelons que le symbole `=` utilisé sous Python pour réaliser une affectation ne doit pas être confondu avec un symbole d'égalité tel qu'il est compris en mathématiques. Il est tentant d'interpréter l'instruction `altitude = 320` comme une affirmation d'égalité, mais ce n'en est pas une pour les raisons suivantes :

- L'égalité est **commutative**, alors que l'affectation ne l'est pas. En mathématiques, les écritures `a = 7` et `7 = a` sont équivalentes. En programmation, une instruction telle que `320 = altitude` est illégale.
- L'égalité est **permanente**, alors que l'affectation peut être remplacée. Lorsqu'en mathématiques, nous affirmons une égalité telle que `a = b` au début d'un raisonnement, alors `a` continue à être égal à `b` durant tout le développement qui suit. En programmation, une première instruction d'affectation peut rendre égales les valeurs de deux variables, et une instruction ultérieure en changer ensuite l'une ou l'autre :

```
>>> a = 5
>>> b = a          # a et b contiennent des valeurs égales
>>> a = 7
>>> print(b)       # b n'ayant pas été modifié contient toujours la valeur 5
5
>>> b = 2          # a et b sont maintenant différentes
>>> print(a)       # a n'a pas changé de valeur
7
```

5 Expressions et opérateurs

Une **expression** est une portion de code que l'interpréteur peut évaluer pour obtenir une valeur. Une expression est formée par la combinaison de valeurs et de variables avec des **opérateurs** :

```
>>> a, b = 7.3, 12
>>> y = 3*a + b/5      # la partie droite de cette instruction est une expression
```

Dans cet exemple, nous commençons par affecter aux variables `a` et `b` les valeurs `7,3` et `12`. 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`. Python évalue chaque expression 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 `float`, parce que l'expression évaluée pour déterminer sa valeur contient elle-même au moins un `float`.

Les **opérateurs** sont les symboles spéciaux utilisés pour représenter des opérations simples, telles l'addition ou la multiplication. Les **opérandes** sont les valeurs combinées à l'aide des opérateurs :

Opérations	Opérateur	Expression
Addition	+	a + b
Soustraction	−	a − b
Multiplication	*	a * b
Division réelle	/	a / b
Division entière	//	a // b
Exponentiation	**	a ** b
Modulo	%	a % b

L'opérateur **modulo**, représenté par `%`, fournit le **reste de la division entière** d'un nombre par un autre :

```
>>> 20 // 3              # résultat tronqué de la division de 20 par 3
6
>>> 20 % 3               # reste de la division entière de 20 par 3
2
>>> 20 // 4
5
>>> 20 % 4               # 20 est un multiple de 4, donc pas de reste
0
```

Les opérateurs Python ne sont pas seulement les opérateurs mathématiques de base. Il faut encore ajouter un certain nombre d'opérateurs logiques, des opérateurs agissant sur les chaînes de caractères, des opérateurs effectuant des tests d'identité ou d'appartenance que nous étudierons dans la suite de ces notes.

5.1 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, il s'agit des mêmes qu'en mathématiques. 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 (et non 59).

5.2 Composition

L'une des forces d'un langage 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
```

Cette fonctionnalité vous permettra de programmer des algorithmes complexes de façon claire et concise :

```
>>> h, m, s = 15, 27, 34
>>> print("nombre de secondes écoulées depuis minuit =", h*3600 + m*60 + s)
nombre de secondes écoulées depuis minuit = 55654
```

Dans l'exemple ci-dessus, remarquons tout d'abord que la fonction **print** permet d'afficher plusieurs valeurs fournies entre parenthèses et séparées les unes des autres par des virgules. En outre, l'interpréteur Python évalue la valeur correspondant à l'expression `h*3600 + m*60 + s` avant de l'afficher.

5.3 Opérations sur les chaînes de caractères

En fonction des circonstances, nous souhaiterons traiter une chaîne de caractères, tantôt comme un objet, tantôt comme une collection de caractères distincts. Un langage de programmation tel que Python doit donc être pourvu de mécanismes qui permettent d'accéder séparément à chacun des caractères d'une chaîne.

Python considère qu'une chaîne de caractères est un objet de type **séquences**, lesquelles sont des **collections ordonnées d'éléments**. Les caractères d'une chaîne sont donc toujours disposés dans un certain ordre. Ainsi, chaque caractère de la chaîne peut être désigné par sa place dans la séquence, à l'aide d'un **index**.

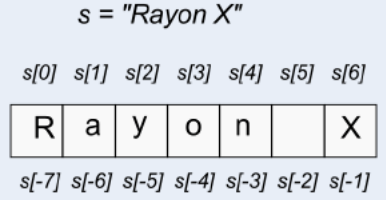
Pour accéder à un caractère bien déterminé, on utilise le nom de la variable qui contient la chaîne et on lui accole, entre deux crochets, l'index numérique qui correspond à la position du caractère dans la chaîne. Comme vous aurez l'occasion de le vérifier, les données informatiques sont souvent numérotées **à partir de zéro** (et non à partir de un). C'est le cas pour les caractères d'une chaîne :

```
>>> s = "Rayon X"
>>> print(s[0], s[1], s[4])
R a n
```

De plus, il est souvent utile de pouvoir désigner l'emplacement d'un caractère par rapport à la fin de la chaîne. Pour ce faire, il suffit d'utiliser des indices négatifs : -1 pour le dernier caractère, -2 pour l'avant-dernier, etc. :

```
>>> print(s[-1], s[-2], s[-4], s[-7])
X   o R
```

L'indexation de la chaîne `s` ci-dessus peut être visualisée à l'aide du schéma suivant :



Notons que Python intègre de nombreuses **fonctions** qui permettent d'effectuer divers traitements sur les chaînes de caractères (conversions majuscules/minuscules, découpage en chaînes plus petites, recherche de mots, etc.). Pour l'instant, contentons-nous de savoir qu'il est possible de :

- assembler plusieurs petites chaînes pour en construire de plus grandes. Cette opération s'appelle **concaténation** et on la réalise sous Python à l'aide de l'opérateur `+` :

```
>>> a = 'Petit poisson'
>>> b = ' deviendra grand'
>>> c = a + b
>>> print(c)
Petit poisson deviendra grand
```

- répéter une chaîne de caractères à l'aide de l'opérateur * :

```
>>> d = 'zut ! ' * 4
>>> print(d)
zut !  zut !  zut !  zut !
```

Remarquez que les opérateurs + et * peuvent aussi être utilisés pour l'addition et la multiplication lorsqu'ils s'appliquent à des arguments numériques. Le fait que les mêmes opérateurs puissent fonctionner différemment en fonction du contexte dans lequel on les utilise est un mécanisme fort intéressant que l'on appelle **surcharge** des opérateurs. Dans d'autres langages, la surcharge des opérateurs n'est pas toujours possible : on doit alors utiliser des symboles différents pour l'addition et la concaténation.

- déterminer la longueur (c'est-à-dire le nombre de caractères) d'une chaîne, en faisant appel à la fonction intégrée `len` :

```
>>> ch ='Georges'
>>> print(len(ch))
7
```

- convertir en nombre véritable une chaîne de caractères qui représente un nombre :

```
>>> ch = '8647'
>>> print(ch + 45)
    *** erreur *** : on ne peut pas additionner une chaîne et un nombre
>>> n = int(ch)
>>> print(n + 65)
8712                    # OK : on peut additionner 2 nombres
```

Dans cet exemple, la fonction intégrée `int` convertit la chaîne en nombre entier. Il serait également possible de convertir une chaîne de caractères en nombre réel, à l'aide de la fonction intégrée `float`.

- accéder à l'identifiant numérique Unicode d'un caractère à l'aide de la fonction `ord` :

```
>>> ord('A')
65
>>> ord('é')
233
```

- accéder à un caractère à partir de son identifiant numérique Unicode à l'aide de la fonction `chr` :

```
>>> chr(65)
'A'
>>> chr(233)
'é'
```

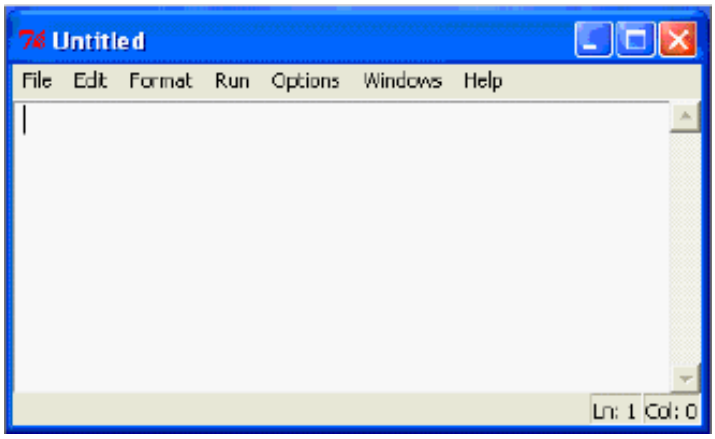
6 Éditer des scripts Python

Jusqu'à présent, vous avez toujours utilisé Python **en mode interactif,** ce qui vous a permis d'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 vous avez écrites disparaissent irrémédiablement dès que vous fermez l'interpréteur. Avant de poursuivre plus avant votre étude, il est donc temps que vous appreniez à sauvegarder vos programmes dans des fichiers, sur disque dur ou clef USB, de manière à pouvoir les retravailler par étapes successives, les transférer sur d'autres machines, etc.

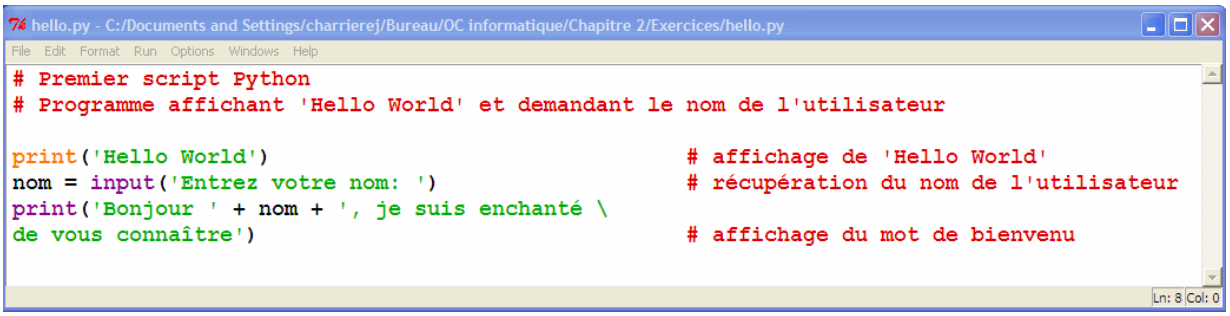
Pour ce faire, vous allez désormais rédiger vos séquences d'instructions dans l'**éditeur de texte** incorporé à l'interface de développement IDLE. Ainsi vous écrirez un **script**, que vous pourrez ensuite sauvegarder, modifier, copier, etc. comme n'importe quel autre texte traité par ordinateur. Suivez le menu **File → New window** (ou taper **Ctrl-N**) situé dans la barre des menus de la console Python. Une nouvelle fenêtre apparaîtra dans laquelle vous pourrez écrire votre script. Cette fenêtre est l'éditeur de texte IDLE [7].

7. Il serait possible d'utiliser un système de traitement de textes, à la condition d'effectuer la sauvegarde sous un format "texte pur" (sans balise de mise en page). Il est cependant préférable d'utiliser un véritable éditeur "intelligent" tel que **IDLE, Notepad++, Sublime Text** ou encore **Visual Code Studio**, muni d'une fonction de coloration syntaxique pour Python, qui vous aidera à éviter les fautes de syntaxe.

Le coin inférieur droit de la fenêtre d'édition de IDLE vous indique le numéro de la ligne sur laquelle se trouve le curseur. Cette information vous permettra de repérer rapidement les endroits du code où se trouvent les erreurs signalées par le **debugger**. Nous reviendrons sur cet outil en temps voulu.



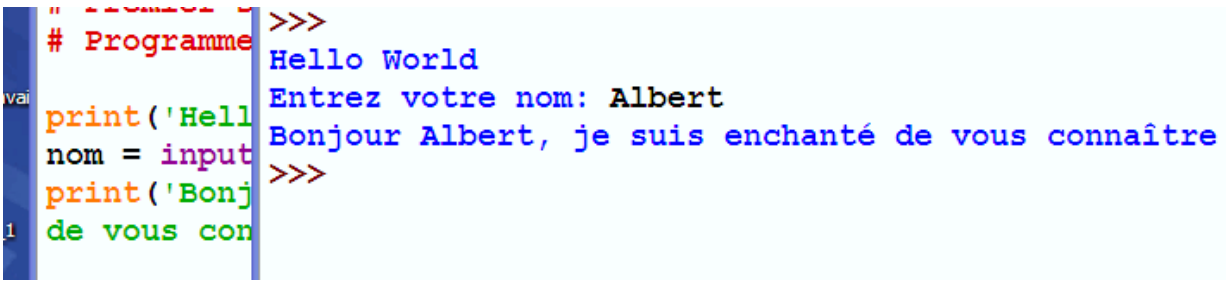
Traditionnellement, le premier programme qu'un utilisateur écrit dans un nouveau langage de programmation est le programme qui affiche 'Hello world' à l'écran. Aussi, allons-nous rédiger ce premier script à l'aide de l'éditeur IDLE. Ouvrez donc l'éditeur de texte et recopier les lignes suivantes :



Remarquez que l'éditeur de texte est muni d'une fonction de coloration syntaxique mettant les différents types d'instructions en évidence à l'aide de plusieurs couleurs, ce qui rend le code plus lisible.

Une fois le code de votre programme tapé, sauvez-le sur votre disque dur de telle sorte que vous n'ayez pas à le retaper à chaque fois que IDLE est lancé. A ce dessein, veillez à choisir un nom de fichier qui se termine par l'extension `.py`. Si vous respectez cette convention, vous pourrez lancer l'exécution du script simplement en cliquant sur l'icône correspondante dans le gestionnaire de fichiers. Ce dernier sait en effet qu'il doit lancer l'interpréteur Python chaque fois que l'utilisateur essaie d'ouvrir un fichier dont le nom se termine par `.py`. La même convention permet en outre à l'éditeur de reconnaître automatiquement les scripts Python et d'adapter sa coloration syntaxique en conséquence.

Si vous travaillez avec IDLE, vous pouvez lancer l'exécution du script en cours d'édition en suivant le menu **Edit → Run script** ou plus directement à l'aide de la combinaison de touches **Ctrl-F5**. Si votre script ne comporte aucun erreur, il devrait s'exécuter comme suit :



Si vous avez rédigé votre script avec une version récente de IDLE, le script décrit ci-dessus devrait s'exécuter sans problème avec la version actuelle de Python 3. Si votre logiciel éditeur est ancien ou mal configuré, par contre, il se peut que vous obteniez un message d'erreur similaire à celui-ci :

```
File "hello.py", line 6
SyntaxError: Non-UTF-8 code starting with '\xe0' in file hello.py on line 6, but no
encoding declared; see http://python.org/dev/peps/pep-0263/ for details
```

Ce message vous indique que le script contient des caractères typographiques encodés suivant une norme ancienne (vraisemblablement la norme ISO-8859-1 ou Latin-1). Dans ce cas, vous devez :

- soit reconfigurer votre éditeur de textes pour qu'il encode les caractères en Utf-8 (suivez le menu **Options → Configue IDLE...,** atteignez l'onglet **General** et cochez la case radio **UTF-8** de l'option **Default Source Encoding**). Vous serez ainsi certain à l'avenir de travailler en accord avec les conventions de standardisation actuelles, qui finiront tôt ou tard par remplacer partout les anciennes.
- soit inclure le pseudo-commentaire suivant au début de tous vos scripts (obligatoirement à la 1^{e} ligne) :

```
# -*- coding:Latin-1 -*-
```

Le pseudo-commentaire ci-dessus indique à Python que vous utilisez dans votre script le jeu de caractères **ASCII étendu** correspondant aux principales langues de l'Europe occidentale, encodé sur un seul octet suivant la norme **ISO-8859-1**, laquelle est souvent désignée par l'étiquette **Latin-1** .

Python peut traiter correctement les caractères encodés suivant toute une série de normes, mais il faut alors lui signaler laquelle vous utilisez à l'aide d'un pseudo-commentaire en début de script. Sans cette indication, Python considère que vos scripts ont été encodés en Utf-8[8], suivant en cela la nouvelle norme **Unicode**, laquelle a été mise au point pour standardiser la représentation numérique de tous les caractères spécifiques des différentes langues mondiales, ainsi que les symboles mathématiques, scientifiques, etc.

Les considérations techniques ayant été abordées, il est maintenant temps de mettre l'accent sur trois concepts importants utilisés dans le script que vous venez de rédiger.

6.1 Commentaires

Un script Python contient des séquences d'instructions identiques à celles que vous avez expérimentées en mode interractif. Puisque ces séquences sont destinées à être conservées et relues plus tard par vous-même ou par d'autres, **il vous est très fortement recommandé d'expliciter vos scripts le mieux possible, en y incorporant de nombreux commentaires**. La principale difficulté de la programmation consiste en effet à mettre au point des algorithmes corrects. Afin que ces algorithmes puissent être vérifiés, corrigés, modifiés dans de bonnes conditions, il est essentiel que la signification des parties non triviales soient expliquée par un commentaire de leur auteur.

Un commentaire commence par le caractères # et s'étend jusqu'à la fin de la ligne courante :

```
# Premier script Python
# Programme affichant 'Hello World' et demandant le nom de l'utilisateur

print('Hello World')                          # affichage de 'Hello World'
nom = input('Entrez votre nom: ')             # récupération du nom de l'utilisateur
print('Bonjour ' + nom + ', je suis enchanté \
de vous connaître')                           # affichage du mot de bienvenu
```

Afin de vous montrer tout de suite le bon exemple, nous avons commencé notre script par deux lignes de commentaires, qui contiennent une courte description de la fonctionnalité du programme. Prenez l'habitude de faire de même dans vos propres scripts. Certaines lignes de code sont également documentées. Si vous procédez ainsi, c'est-à-dire en insérant des commentaires à la droite des instructions correspondantes, veillez à les écarter suffisamment de celles-ci, afin de ne pas gêner leur lisibilité.

8. Dans les versions de Python antérieures à la version 3.0, l'encodage par défaut était ASCII. Il fallait donc toujours préciser en début de script les autres encodages (y compris Utf-8).

6.2 Entrées

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 en mode texte, la méthode la plus simple consiste à employer la fonction intégrée **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 chaîne de caractères correspondant à ce que l'utilisateur a entré. Cette chaîne peut alors être assignée à une variable quelconque, convertie, etc.

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

```
nom = input("Entrez votre nom: ")
print("Bonjour,", nom)
```

ou encore :

```
print("Veuillez entrer un entier positif quelconque: ")
chaine = input()
entier = int(chaine)              # conversion de la chaîne en un nombre entier
print("Le carré de", entier, "vaut", entier**2)
```

La fonction `input()` renvoie toujours une chaîne de caractères. Si vous souhaitez que l'utilisateur entre une valeur numérique, vous devrez donc convertir la valeur entrée (qui sera donc de toute façon de type string) en une valeur numérique du type qui vous convient, par l'intermédiaire des fonctions intégrées `int()` (si vous attendez un entier) ou `float()` (si vous attendez un réel) :

```
>>> a = input("Entrez une donnée numérique: ")
Entrez une donnée numérique: 52.37
>>> type(a)
<class 'str'>
>>> b = float(a)            # conversion de la chaîne en un nombre réel
>>> type(b)
<class 'float'>
```

6.3 Sorties

En mode interactif, Python **lit-évalue-affiche** les instructions entrées par l'utilisateur mais la fonction `print` reste indispensable aux affichages en mode script. Cette fonction permet d'afficher n'importe quel nombre de valeurs fournies en arguments (c'est-à-dire entre les parenthèses). Par défaut, ces valeurs sont séparées les unes des autres par un espace, et le tout se termine par un saut à la ligne :

```
>>> a, b = 2, 5
>>> print("Le produit de", a, "par", b, "vaut:", a*b)
Le produit de 2 par 5 vaut: 10
```

Vous pouvez remplacer le séparateur par défaut (l'espace) par un autre caractère quelconque (ou même par aucun caractère), grâce à l'argument `sep` :

```
>>> print("Bonjour", "à", "tous", sep="*")
Bonjour*à*tous
 >>> print("Bonjour", "à", "tous", sep="")
Bonjouràtous
```

De même, vous pouvez remplacer le saut à la ligne terminal avec l'argument `end` :

```
>>> print("zut")
>>> print("zut", end = " ")
>>> print("zut", end = " ")
>>> print("zut", end = " ")
zut
zut zut zut
```

7 Glossaire du chapitre

La liste suivante répertorie dans l'ordre alphabétique les principaux termes introduits dans ce chapitre. La connaissance de leur définition est indispensable à la suite de votre apprentissage :

affectation : action d'assigner une valeur à une variable.

affectation multiple : opération visant à affecter une même valeur à plusieurs variables simultanément.

affectation parallèle : opération visant à affecter des valeurs (non forcément identiques) à plusieurs variables simultanément.

commentaire : information située dans un script ayant pour but d'expliquer la signification de certaines parties de code non triviales à d'autres utilisateurs et ayant aucun effet sur l'exécution du programme.

concaténer : joindre deux opérandes bout à bout.

donnée : élément de base tel qu'un nombre ou une chaîne de caractères qu'un programme manipule.

évaluer : simplifier une expression en exécutant les opérations jusqu'à obtenir une seule valeur.

expression : combinaison de valeurs, variables et opérateurs produisant une valeur lors de son exécution.

floating-point : type de données représentant les nombres réels.

fonction : séquence nommée d'instructions qui exécutent une opération prédéfinie.

instruction : partie de code qui représente une commande ou une action à entreprendre.

integer : type de données représentant les nombres entiers.

mode interactif : manière d'utiliser l'interpréteur Python en tapant des commandes et des expressions directement à la suite du prompt.

mode script : manière d'utiliser l'interpréteur Python pour lire et exécuter des instructions dans un script.

opérande : une des valeurs sur lesquelles agit un opérateur.

opérateur : symbole spécial représentant une action simple telle qu'une addition, une multiplication ou une concaténation.

opérateur modulo : opérateur représenté par le symbole % qui agit sur des entiers et retourne le reste de la division entière du premier opérande sur le deuxième.

prompt : suite de caractères (>>> sous IDLE) affichée par l'interpréteur pour inviter l'utilisateur à entrer la prochaine instruction à évaluer.

règles de priorité : ensemble des règles déterminant l'ordre dans lequel les opérations qui composent une expression doivent être effectuées.

script : programme stocké dans un fichier et habituellement destiné à être interprété.

string : type de données représentant une chaîne de caractères.

type : catégorie de données telles que les entiers (type `int`), les nombres réels (type `float`) ou les chaînes de caractères (type `str`).

variable : nom référençant une valeur et son type.

8 Exercices

Exercice 2.1
Supposons que les affectations suivantes aient été exécutées :

```
>>> largeur = 17
>>> hauteur = 12.0
>>> delimiteur = ','
```

Dans chacun des cas suivants, déterminez la valeur de l'expression et son type :

(1) `largeur // 2`
(2) `largeur / 2`
(3) `6-hauteur // 3`
(4) `hauteur / 3`
(5) `1 + 2 * hauteur`
(6) `(str(largeur) + delimiteur)*5`

Exercice 2.2
Ecrivez un programme permettant d'échanger la valeur de deux variables `a` et `b` initialisées à 3 et 5 :

(1) sans affectation parallèle.
(2) avec affectation parallèle.

Exercice 2.3
Ecrivez un programme qui permette d'afficher chacune des figures suivantes en utilisant les opérateurs de répétition et de concaténation :

(1)
```
    *
   * *
  *   *
 *     *
  *   *
   * *
    *
```

(2)
```
  *
 ***
*****
  *
  *
  *
  *
```

Exercice 2.4
Ecrivez un programme qui affiche chacun des caractères de la variable `mot='Salut'`

(1) en colonnes
(2) en ligne et séparés d'un %.

Exercice 2.5
Ecrivez un programme qui affiche le dernier caractère d'une chaîne entrée par l'utilisateur.

Exercice 2.6
Ecrivez un programme qui affiche la somme, la différence, le produit et le quotient de deux nombres transmis par l'utilisateur.

Exercice 2.7
Ecrivez un programme qui transforme en km/h une vitesse transmise en m/s.

Exercice 2.8
Ecrivez un programme qui retourne la décomposition d'un entier n entré par l'utilisateur sous la forme

$$n = 7 * q + r \text{ avec } 0 <= r < 7$$

(Pour $n = 39$, le programme affichera $39 = 7 * 5 + 4$)

Exercice 2.9
Ecrivez un programme qui convertit un nombre entier de secondes fourni au départ en un nombre d'années, de mois, de jours, d'heures, de minutes et de secondes. Pour cet exercice, on considérera des mois de 30 jours.

Exercice 2.10
Ecrivez un programme qui affiche les premières lettres minuscules de l'alphabet grec sachant que leurs identifiants Unicode sont situés entre 945 et 949.

Exercice 2.11
Ecrivez un programme qui convertit un octet entré par l'utilisateur en l'entier naturel correspondant. Comme,

$$0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 64 + 32 + 16 + 2 = 114,$$

la chaîne "01110010" produira 114 en sortie.