Table des matières

fiche 1 : premiers pas en PYTHON 3

1. préliminaires.

1.1. PYTHON 3.01

Python est un langage interprété. Un peu à la manière de Javascript, qui réinterprète le source à chaque exécution, un peu à la manière de Java puisqu'il a la possibilité de créer un pseudo-code optimisé interprété par une machine virtuelle.

Python est un langage qui évolue. Lorsqu'un langage évolue, ses promoteurs essaient de maintenir la "compatibilité ascendante". La version n intègre la version (n-1). Ce qui est écrit dans les versions anciennes continue à être correctement traité dans la version nouvelle. Avec un langage compilé (C, C++, Delphi) le source d'un programme écrit pour la version (n-1) est compilé sans problème par le compilateur de la version n. Dans un langage interprété (Java, Javascript), le texte ou le pseudo-code de la version (n-1) reste valide pour la machine virtuelle des versions ultérieures. Ce problème de la compatibilité n'a pas que des avantages, c'est aussi un vrai handicap.

Exemple : en Java, on a vu apparaître des objets et des méthodes "obsolètes". Un objet obsolète continue à être valide, mais le langage propose de le remplacer par un autre jugé plus performant ou plus sûr. À terme, les objets jugés désuets risquent de disparaître du langage.

Il existe des cas de rupture brutale. Certaines données ou formes syntaxiques ne sont plus maintenues, plus acceptés par le langage. C'est le cas en HTML, langage pour lequel les navigateurs cessent d'nterpréter certaines balises issues des premières générations du HTML. C'est par exemple le cas de, la balise **<font>** dans l'interpréteur de Firefox en version 2.5 et supérieur. Les pages web écrites avec ces anciennes balises HTML sont périmées.

Il y a une rupture du même type entre les versions 2 et la version 3 de Python. Que les textes sources écrits en version 3 ne soient pas correctement interprétés par des interpréteurs des versions 2 est assez logique. Mais en Python, les sources et pseudo-codes réalisés selon certains critères (syntaxiques et lexicaux) des versions 2 ne sont pas reconnus en version 3. Pour contourner ce problème d'incompatibilité, certain logiciels intègrent leur propre interpréteur. C'est la cas de Blender qui intègre son propre interpréteur de langage PYTHON. Cependant, on se trouve devant un dilemme en travaillant avec Blender :

* soit on travaille uniquement sur la version 2 de Python, avec le risque qu'un jour prochain Blender passe à la version 3,

* soit on travaille proprement le code Python en version 3, et en cas de besoin, on bricole le programme écrit dans cette version pour le rendre compatible avec l'interpréteur intégré à Blender.

Cette seconde solution n'est pas satisfaisante, mais pour une situation d'apprentissage, elle est la moins mauvaise. Il faut simplement prévoir des notes, indiquant les principales évolutions pour, le cas échéant, revenir à la vieille version. (Nous verrons que sous Linux, il est possible d'indiquer la version de l'interpréteur Python a utiliser sur la première ligne de chaque programme. Nous avons alors la cohabitation de 2 langages distincts).

Un autre inconvénient au niveau de l'apprentissage, est que les tutoriaux sont écrits dans les vieilles versions. Cependant, adapter en version 3 les exemples de la version 2 est une bonne occasion pour signaler les grosses différences et les évolutions sémantiquement significatives. On a tout à gagner en travaillant directement en version 3.

1.2. Environnement de développement ou pas ?

Un environnement de développement ou IDE (IDE - Integrated Development Environment) est un logiciel constitué d'outils qui facilitent l'écriture et les tests dans un langage défini, voire plusieurs. Un IDE comporte en général un éditeur avec coloration syntaxique, un système de gestion de fichiers (sauvegarde/chargement), un compilateur le cas échéant, un exécuteur de programme, un système d'aide en ligne, des indicateurs de syntaxe etc. Le plus connu est peut être Eclipse. On connaît aussi JBuilder et NetBean pour Java, CBuilder pour le C, Dreamweaver (sous Windows) et Quanta+ (sous

Linux) pour le HTML (et les langages satellites). Pour certains langages (PHP, Javascript) les IDE sont inexistants (avant que NetBean ne prenne en charge le PHP) ou confidentiels (pour les spécialistes).

Les adeptes de Python se partagent entre tenants du travail sur console (ou éditeur à coloration syntaxique + console) et IDE. Il existe une IDE qui s'appelle IDLE, assez élémentaire et en anglais. C'est l'IDE fournie par les créateurs/mainteneurs de Python autour de Guido van Rossum, inventeur de Python. Comme l'aide qu'il apporte permet de passer sans difficulté au mode console, nous adopterons cet IDE qui en plus accompagne le paquet chargé avec Python. Il existe évidemment de nombreux autres IDE, mais à caractère plus technique et/ou hors du monde du libre.

IDLE n'est pas un outil de production ; il fonctionne aussi bien sous Windows, que sous Linux et Mac. Ce qui est intéressant pour le travail de club même s'il y a de petites différences de présentation selon le système d'exploitation. La question sera traitée dans le chapitre "un langage de blocs" dans la fiche 2. Il est évident que le travail sur console sera évoqué en son temps car l'exécution d'un programme à travers un IDE peut cacher des problèmes de portabilité.

Quand à l'aide, elle passe de toute façon par un navigateur ; l'aide sur la machine est en anglais est en mode **`chm`**. L'aide en ligne est bien mise à jour… La traduction en français est engagée, mais ne tient pas ses promesses.

Il existe une abondante documentation en français. En tout état de cause, nous conseillons de passer en priorité sur le site **DEVELOPPER.COM, rubrique Python.** La majorité des sites et forums en français répertoriés par Google sont décevants, voire complètement creux ; à l'exception de ceux auxquels renvoie **DEVELOPPER.COM,** comme le site des **utilisateurs francophones de Python**.

En matière de livres en français, citons uniquement

* celui de Gérard Swinnen, "*Apprendre à programmer avec PYTHON*" disponible en PDF sur **`http://python.developpez.com/cours/TutoSwinnen/`**

***** la version française en PDF du "*Tutoriel Python*" par Guido van Rossum, sur **`http://python.developpez.com/cours/TutoVanRossum/.`**

* la version française de **`"Au coeur de Python"`** de Wesley J. Chun. Une partie de ce livre se retrouve en ligne.

Ces écrits demandent à être adaptés à la version 3.0x. **Le tutoriel (110 pages) de Guido van Rossum est un passage obligé**. Le livre de Swinnen est incomplet, mais très sûr et très pédagogique.

1.3. Téléchargement.

Le paquet est disponible pour Windows comme pour Linux à l'adresse :

`http://www.python.org/download/releases/3.1/`

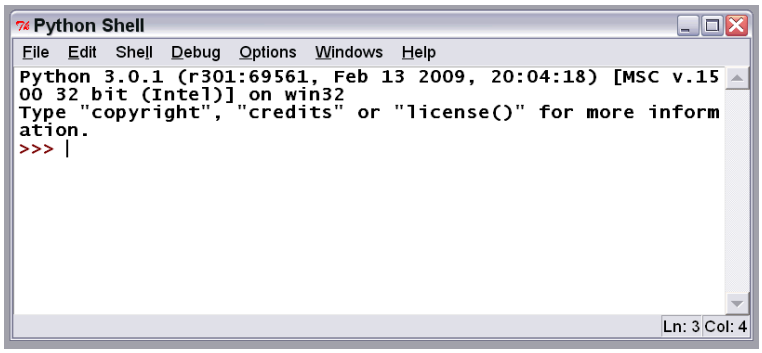
Le format de fichier pour Windows est **`.MSI`** , en anglais : **`Microsoft Software Installe`**r (en français : Installateur de Logiciels Microsoft). Un package Windows avec l'extension **`.msi`** a un processus d'installation simplifié. Le "msi" installe et configure les services d'une application ou effectue une installation depuis le réseau. Il est reconnus par XP et suivants. Il suffit donc de cliquer le package, comme pour un fichier exe.

Pour Linux, on trouve tout le matériel (sources, binaires) à la même adresse ; mais la majeure partie des distributions Linux incluent Python dans l'installation par défaut. Cependant, en 2009, ce n'est pas la version 3 de Python qui est installée mais une version 2.5. Il est possible d'installer et de faire cohabiter les version 2 et 3 de Python. On peut installer la version 3.1 de Python et l'IDLE adapté à partir du dépôt d'Ubuntu 9.4 (et suivants)

2. présentation de l'IDE.

2.1. IDLE

Une fois le package installé, on peut lancer l'IDE qui s'appelle IDLE. Une fenêtre qui ressemble à la copie d'écran (sous Windows ; sous Linux, IDLE est un peu différent) ci-dessous s'ouvre :



* On peut regarder ce qui correspond à chaque élément de menu ; mais, pour l'instant, le premier élément important est de choisir une fonte adaptée. On le fait dans le menu option ; il faut choisir une fonte à espacements fixes (courier, lucida typewriter etc). Ici, pour avoir des copies d'écran bien denses, on a choisi un lucida console gras en 12 points. Les autres options ne sont pas à changer.

* le symbole « **>>>**» est le prompt de Python ; la position du curseur est notée en bas, à droite.

Comme les langages de shell ou le vieux basic, Python peut être utilisé de façon interactive, c'est à dire qu'on peut exécuter une ligne de commande immédiatement. L'ordre d'exécution est la touche entrée. L'en-tête incite à exécuter les commandes **`copyright`**, **`crédits`** et **`licence()`.**

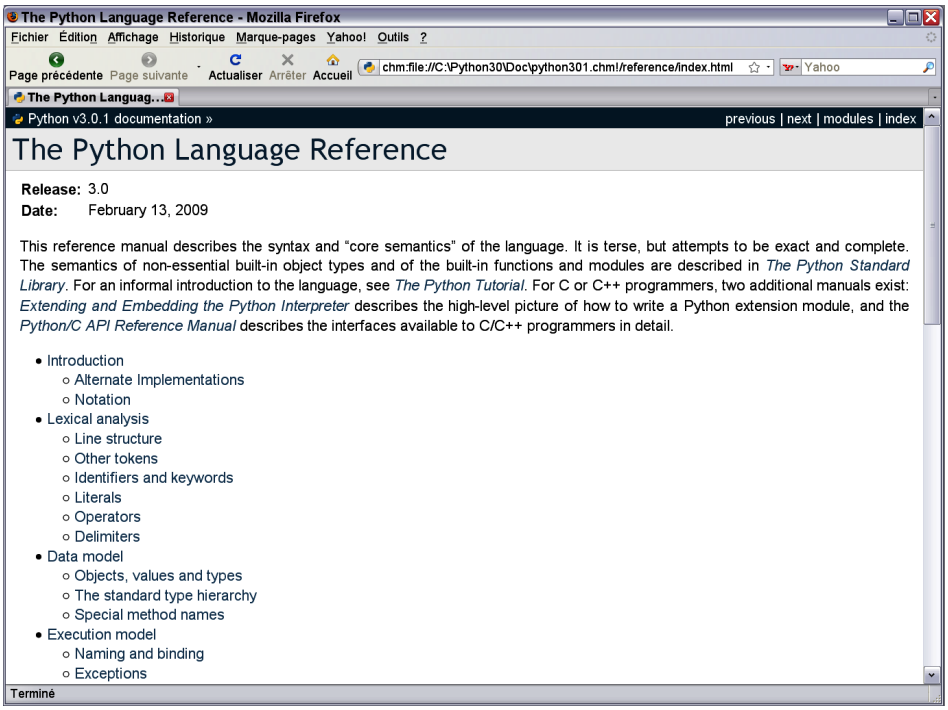
```
>>> copyright
Copyright (c) 2001-2009 Python Software Foundation.
All Rights Reserved.

Copyright (c) 2000 BeOpen.com.
All Rights Reserved.

Copyright (c) 1995-2001 Corporation for National Research Initiatives.
All Rights Reserved.

Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
>>>
```

2.2. L'aide.

* Le menu Help permet d'accéder aux raccourcis clavier. Quant à l'aide Python proprement dite, on peut choisir l'aide sur la machine, contenue dans le fichier **`python301.chm`**, du répertoire Doc de l'installation, à ouvrir avec un navigateur ad hoc (sous Windows, il existe un lecteur natif ; avec Firefox, il faut ajouter un plugin, **`chm reader`** à télécharger ; sous Linux, il existe plusieurs lecteurs de fichiers chm). On peut choisir aussi l'accès, par le menu, de l'aide en ligne. Cette option suppose que le centre de diffusion de Python n'ait pas changé l'adresse de la documentation ! En cas d'adresse incorrecte, rectifier dans le fichier EditorWindows.py, ligne :

```
EditorWindow.help_url = "http://docs.python.org/3.0/"
```

On trouvera ci-dessous une copie d'écran de la fenêtre d'aide sous Firefox (**`Fichier > Ouvrir un fichier CHM`** ….), avec le plugin chargé.



3. Premier pas.

3.1. le mode calculette

Voici quelques exemples de fonctionnement avec pour commande, l'exécution d'un calcul :

```
>>> 3
3
>>> 3+3-8
-2
>>> 2**8
256
>>> 15 / 4
3.75
>>> 15 // 4
3
>>> 15 % 4
3
>>> 
```

* un nombre est une commande.
* les opérations fonctionnent classiquement ; les règles de priorité et de parenthésage sont les règles usuelles.
* la puissance se note avec deux étoiles successives.
* la division avec le slash fonctionne en nombre flottant.
* la division entière se fait avec un double slash.
* le modulo avec le signe %
* les nombres flottants se notent avec un point.

3.2. les chaînes de caractère.

Les exemples suivant illustrent comment se comporter avec les chaînes de caractère :

```
>>> "n"
'n'
>>> 'n'
'n'
>>> n
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    n
NameError: name 'n' is not defined
>>> "cette chaîne n'a rien de spécial ?"
"cette chaîne n'a rien de spécial ?"
>>> 'cette chaîne n'a rien pour elle'
SyntaxError: invalid syntax (<pyshell#4>, line 1)
>>> """cette chaîne est "spécifique" de Python"""
'cette chaîne est "spécifique" de Python'
>>> '''cette chaîne l'est-elle ?'''
"cette chaîne l'est-elle ?"
>>> 
```

* une chaîne se met entre quotes simples ou quotes doubles ;
* il y a erreur si les quotes sont omises (voir plus loin la notion de variable).
* bien observer comment s'alternent simples et doubles quotes.
* les quotes simples ou doubles données en réponse lors de l'utilisation comme commande d'une chaîne valide sont optimisées par la machine Python.

Un exemple avec caractères d'échappement :

```
>>> """le cheval, l'âne et le "jacktalope" juvénile"""
'le cheval, l\'âne et le "jacktalope" juvénile'
>>> "le cheval, l'âne et le \"jacktoalope\" juvénile"
'le cheval, l\'âne et le "jacktoalope" juvénile'
>>>
```

Le **saut de ligne** utilise aussi le caractère d'échappement **** et le caractère s'écrit **\n** :

3.3.  nombres et opérateurs en version 2

Les nombres n'ont pas le même statut en version 2 et en version 3. Les types ne sont pas identiques. La division avec simple slash est entière si les deux arguments le sont, et flottante si l'un des argument l'est.

Par exemple, **`15 / 4`** s'écrit en version 3 : **`15 // 4 ;`** et si l'on veut une division flottante en version 2, il faut écrire **`15.0 / 4`** alors qu'en version 3, on écrit **`15 / 4 qui`** donne un flottant.

3.4. identificateurs.

Comme dans tout système de programmation, il existe des "mots", créés par le programmeur, pour identifier un objet. Par exemple, en HTML, une balise peut être nommée par un identificateur :

`<h1 id="maBalise" class="grosTitre">`, le mot **`maBalise`** identifie la balise **`h1`** actuellement utilisée ; c'est ce mot qui sert pour la désigner dans un script javascript destiné à en changer une caractéristique graphique (la couleur, la marge…), ou encore à récupérer le "contenu" de la balise.

Pyhon possède aussi des identificateurs ; **quelles en sont les règles de construction ?**

* il n'y a pas de restriction sur la longueur ; tout dépend de l'implémentation. Disons simplement qu'il convient d'être raisonnable (16 caractères).

* les seuls caractères utilisables sont les minuscules, les majuscules, le souligné; les chiffres. Il faut éviter les caractères accentués, les cédilles, même s'ils ne sont pas interdits pour éviter les problèmes lorsque l'on change de système d'exploitation. Majuscules et minuscules sont différentiés.

* tout caractère autorisé sauf un chiffre peut être la première lettre d'un identificateur.

note : il faut éviter de commencer un identificateur par le souligné ; ce type d'identificateur est licite, mais il a des utilisations particulières en Python. Une bonne façon de procéder est de créer des identificateurs pas trop courts, expressifs, lisibles et faciles à écrire : maBalise, laMairieDeSaintJean. On rappelle la recommandation suivante : n'utiliser que les caractère ASCII (pas d'accents…)

3.5. variables.

Les seuls objets rencontrés jusqu'ici sont les nombres (entier ou flottants) et les chaînes de caractères. Les exemples donnés le sont avec ces objets ; on généralisera plus tard.

* on peut affecter la valeur d'un objet à un identificateur. L'affectation occupe une ligne :

`<identificateur> = <expression>`

* une affectation vaut déclaration, c'est-à-dire que l'identificateur peut se substituer à l'expression qui lui a été affectée.

Il n'y a pas de déclaration de variable, comme en Pascal, C ou Java. Une variable est créée par une affectation.

En particulier, l'identificateur devient une commande en mode shell. Sinon, il n'est pas reconnu par l'interpréteur et suscite une erreur si on l'utilise comme commande.

* une affectation ne retourne rien contrairement au PHP ; ainsi en PHP l'affectation x = 0 retourne 0 et l'affectation peut être utilisée comme booléen. En Python, c'est une erreur.

note : Si on a programmé avec d'autres langages, on a l'habitude de la notion de variable. Dans un premier temps, on utilise cette notion comme d'habitude. On verra cependant que la notion de variable est loin d'être naïve en Python (dans tous les langages objets d'ailleurs). Une étude plus pointue sera menée ultérieurement.

```
>>> bibi=3
>>> Bibi=7
>>> bibi
3
>>> Bibi
7
>>> img1="mairie"
>>> img1
'mairie'
>>> _img1="école"
>>> _img1
'école'
>>> 1tab="Jean"
SyntaxError: invalid syntax (<pyshell#8>, line 1)
>>> toto
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    toto
NameError: name 'toto' is not defined
>>>
```

3.6. quelques particularités.

* l'affectation en cascade est autorisée.

* une variable déclarée s'utilise comme la valeur qu'elle représente.

* les variables ne sont pas typées : on peut affecter à une variable des objets de natures diverses.

```
>>> x1=x2=x3=3.1406
>>> x1
3.1406000000000001
>>> x1=x2=x3=3.1416
>>> x1
3.1415999999999999
>>> x2
3.1415999999999999
>>> x3='message à l\'affiche'
>>> x3
"message à l'affiche"
>>> x1**4
97.410002176508314
>>>
```

Une petite surprise : à l'affichage, la valeur numérique flottante entrée et la valeur affichée ne sont pas semblables ! On a le même comportement avec les chaînes : lorsque l'on utilise un objet comme commande, c'est la valeur interne de l'objet qu'il fournit . Pour les flottants, la version 3.1. a "normalisé" l'affichage, qui est ainsi plus naturel.

* On peut même faire des affectation en bloc :

```
>>> r, pi, autre = 5, 3.1416, "c'est autre chose !"
>>> r
5
>>> pi
3.1415999999999999
>>> autre
"c'est autre chose !"
>>> type(r)
<class 'int'>
>>> type(pi)
<class 'float'>
>>> type(autre)
<class 'str'>
>>>
```

note : on a affiché le **type** de l'objet affecté**,** qui est appelé aussi sa **classe**; pour l'instant, on dispose des entiers (classe `'int'`), les flottants (classe `'float'`), et les chaînes de caractères ou strings (classe `'str'`).

4. La fonction print().

La plupart des langages possèdent une fonction **`print()`**. Elle envoie l'argument de la fonction sur le terminal de sortie par défaut (**`stdout`** ; en général c'est l'écran, mais ce peut être autre chose, comme une imprimante, un port usb etc). C'est le cas en Pascal de la fonctions **`write()`** et de **`writeln()`**. Javascript dispose de la fonction **`write()`**, mais son terminal de sortie est la page HTML où se trouve le script; il en est de même avec le **`echo()`** du PHP.

 Dans la version 2 de Python, **`print`** n'est pas une fonction mais une commande du langage et on peut l'utiliser sans parenthèses.

Voici ce qui se passe dans IDLE lorsque l'on commence la frappe :

```
>>> print(
        print(value, ..., sep=' ', end='\n', file=sys.stdout)
```

La syntaxe apparaît dans une fenêtre popup ! C'est un des intérêts de l'IDE. **`print()`** est multi-argument comme on peut le voir dans l'exemple qui suit ; **`sep`** désigne le séparateur des valeurs affichées, formatage sommaire, mais pratique. Le **`end`** désigne la chaîne affichée en final ; **`"\n"`** désigne le saut de ligné ; l'absence de saut de ligne se marque par une chaîne vide **`""`**. On ne va pas modifier le **`file`** (on remarque que par défaut, c'est la sortie standard du système, **`sys.stdout`**).

```
>>> print(x1,x2,x3,x1**4,sep=" ... ")
3.1416 ... 3.1416 ... message à l'affiche ... 97.4100021765
>>>
```

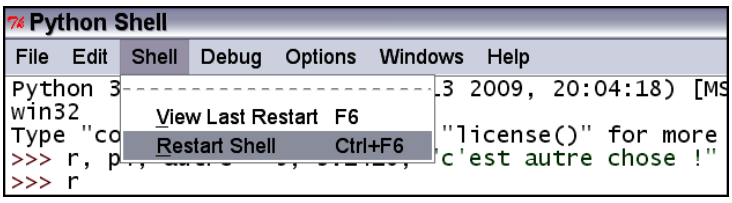
On remarque que les quotes externes du message, ainsi que le caractère d'échappement ont disparus. Voici un exemple de l'utilisation d'un caractère échappé (saut de ligne)

```
>>> print ("toto\ntiti\ntata")
toto
titi
tata
>>>
```

5. La session shell.

Jusque maintenant, on a vu que les variables déclarées "vivaient" dans la session IDLE tout au long de l'exercice. Peut-on, dans le mode interactif, et sans fermer l'IDE, initier une nouvelle session ? C'est à dire provoquer un oubli de tout ce qui a été fait auparavant ?

En PYTHON, une session est appelée un shell ; le changement de shell se fait par le menu Shell :



Ce qui donne :

```
>>> pi
3.1415999999999999
>>> type(pi)
<class 'float'>
>>> ================================ RESTART ================================
>>> pi
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    pi
NameError: name 'pi' is not defined
>>>
```

On peut ainsi travailler par "sessions" indépendantes comme si on avait des programmes différents.

Il y a cependant une limite au redémarrage des sessions shell : si dans une session shell on a utilisé une fonction résidente (comme **`input()`**) elle continue à être active lors d'un changement de session. Aussi on recommande de fermer le shell pour changer de session ! C'est un peu radical, mais évite bien des interrogations.

fiche 2 : considérations générales

1. la fonction input().

1.1. Saisie sur l'entrée standard.

On a déjà rencontré la sortie sur le périphérique standard, la fonction **`print()`**. Il existe évidemment une fonction symétrique, la lecture du périphérique d'entré standard (en général, le clavier), la fonction **`input()`**. **Cette fonction retourne toujours une chaîne de caractère**.

Attention. La fonction **`input()`** est très différente de celle du même nom dans les versions 2 ; Dans les versions 2, la fonction **`input()`** retourne un objet dont le type dépend de la forme de la valeur d'entrée : entier, flottant, chaîne de caractère… **Ceci n'a plus lieu d'être pris en considération.**

exemple en version 3 :

```
>>> x1=input()
12
>>> x1
'12'
>>> type(x1)
<class 'str'>
>>> x2=input("entrer une chaîne avec apostrophe : ")
entrer une chaîne avec apostrophe : ceci est l'adresse de mon "papy"
>>> x2
'ceci est l\'adresse de mon "papy"'
>>> type(x2)
<class 'str'>
>>> x3=input()
'12'
>>> x3
"'12'"
>>> x4=input(
            input([prompt]) -> string
```

On constate que si on entre un entier, la valeur retournée est une chaîne formée des chiffres entrés.

On peut ajouter un prompt comme argument, qui remplace le prompt classique de Python (**>>>**) lors d'une saisie.

Avec IDLE, un popup est associé, qui rappelle que la fonction a un argument éventuel et retourne toujours une chaîne.

1.2. cast.

Si l'on veut saisir un entier ou un flottant, il faut caster la valeur entrée (convertir son type) ; on trouvera ci-dessous l'illustration de l'entrée pour les types rencontrés :

```
>>> x4=input("entrer un entier")
entrer un entier
>>> x4=input("entrer un entier : ")
entrer un entier : 12
>>> x4 = int(x4)
>>> type(x4)
<class 'int'>
>>> x4
12
>>> x5=input('entrer un flottant : ')
entrer un flottant : 12
>>> x5=float(x5)
>>> x5
12.0
>>> x6= input ('entrer un entier : ')
entrer un entier : 12.63
>>> x6=int(x6)
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    x6=int(x6)
ValueError: invalid literal for int() with base 10: '12.63'
>>>
```

1.3. saisie de booléens.

```
>>> 1==1
True
>>> type(1==1)
<class 'bool'>
>>> b1= bool(2.6)
>>> b1
True
>>> b1= bool("")
>>> b1
False
>>> 1<23
True
>>>
```

Il faut savoir qu'en Python, comme dans de nombreux langages, le 0 ou la chaîne vide sont équivalents à False (faux) et que toutes les autres valeurs, entières, flottantes, chaînes, sont castées automatiquement en True (vrai) là où un booléen est attendu. L'égalité a pour signe le double égal (==).

Attention aux majuscules : True, False

On peut avoir besoin d'entrer une valeur booléenne : il suffit alors d'utiliser le 0, la chaîne vide, ou toute autre valeur ad hoc, ou le True et le False.

```
>>>
>>> # chaîne vide pour False :
>>> x1=bool (input ("êtes-vous d'accord ? "))
êtes-vous d'accord ?
>>> x1
False
>>>
>>> # l'entier 0 pour False
>>> x1=bool(int(input ("êtes-vous d'accord ? ")))
êtes-vous d'accord ? 0
>>> x1
False
>>>
>>> # littéral pour faux :
>>> x1=false
Traceback (most recent call last):
  File "<pyshell#85>", line 1, in <module>
    x1=false
NameError: name 'false' is not defined
>>> x1=False
>>> x1
False
```

note : le dièse # sert pour les commentaires (sur la ligne).

Attention à l'usage de 0 comme booléen : **`dans l'exemple, input()`** retourne la chaîne `'0'` qui n'est pas vide ; il ne faut donc pas oublier de caster en entier ; de même, True et False sont des identificateurs, et s'utilisent sans quotes.False apparaît comme une chaîne non vide est se cate donc en True !

```
>>> x1=bool (input ("êtes-vous d'accord ? "))
êtes-vous d'accord ? False
>>> x1
True
>>> # le piège
>>> x1=bool (input ("êtes-vous d'accord ? "))
êtes-vous d'accord ? 0
>>> x1
True
>>>
>>> # autre
>>> x1=bool (input ("êtes-vous d'accord ? "))
êtes-vous d'accord ? 3.1416
>>> x1
True
>>>
```

On dispose donc pour l'instant de quatre types de données : entiers, flottants, chaînes et booléens. "**type**" est équivalent à "**classe**" en Python.

2. langage de blocs.

2.1. "Les accolades sont inutiles" (Guido von Rossum)

A l'exception près des langages de liste (Lisp, Logo), la majeure partie les langages sont des langages de bloc. En Pascal, un bloc de code est délimité par les mots clefs **`begin`** et **`end`**. En C, Java, Javascript, les blocs de code sont délimités par des accolades ouvrante/fermante {…}. Une instruction est terminée par un point-virgule (en Pascal c'est un séparateur d'instruction ; pour les autre langages, une terminaison d'instruction ; nuance…). **Pour Guido van Rossum, les délimiteurs sont inutiles ! et donc pas d'accolades ni de point-virgule.**

Mais on ne s'en tire pas à si bon compte ! Python est un langage de blocs, c'est-à-dire qu'il permet de traiter comme une instruction unique une séquence d'instructions. Il faut bien un truc pour marquer la fin d'une instruction élémentaire et indiquer la fin d'un bloc !

C'est ici que l'auteur de Python reprend une idée importante des tenants de la programmation structurée : **l'indentation du code**. Dans la plupart des langages, la structure du code a sa lecture facilitée par un système d'indentation devenu classique : tout ce qui est indenté de la même façon est une instruction du même bloc, c'est-à-dire de même niveau syntaxique, et si les blocs s'emboîtent, on le marque en augmentant l'indentation. D'un point de vue lexical (déchiffrement du code par l'analyseur), cela ne sert habituellement à rien, tout comme le saut de ligne, assimilé à un séparateur (l'espace, la tabulation…).

En Python, on utilise cette habitude des bons programmeurs en en faisant un élément du langage :

* il y a une instruction **élémentaire** par ligne ; le saut de ligne remplace le point-virgule ;
* les lignes qui ont la même indentation appartiennent au même bloc lexical (à préciser).
* une instruction obligatoire mais vide est définie : **`pass`**.

Ces deux règles ne sont malheureusement pas suffisantes pour un langage qui utilise des programmes et qui peut être interactif au besoin (on dit parfois qu'il est utilisée en mode calculette). Les différences de présentation d'avec les langages plus habituels peuvent un peu dérouter ; mais en Python, on est contraint par l'analyseur à utiliser une bonne présentation, puisque **la présentation, c'est la syntaxe !**

2.2. Voici trois schémas simples de structures :

Pascal

```
if (cond)
      begin
         inst1 ;
         inst2
```

```
    end
else
    begin
        inst3 ;
        inst4 ;
        inst5
    end ;
suite du texte
```

Java (C, PHP, Javascript)

```
if (cond){
    inst1 ;
    inst2 ;
}
else {
    inst3;
    inst4;
    inst5;
}
suite du texte
```

Python

```
if cond :
    inst1
    inst2
else :
    inst3
    inst4
    inst5

suite du texte
```

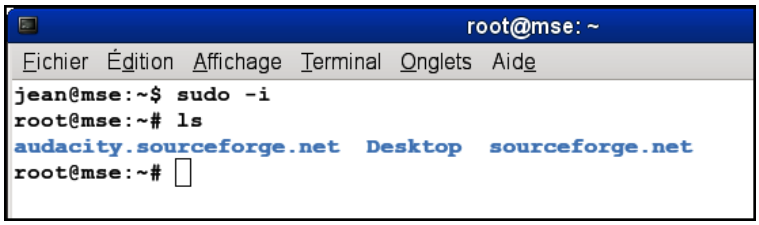
Ici, l'indentation est de quatre caractères ; le choix du nombre de caractère est toujours à la disposition du programmeur. Il faut remarquer, en Python, la présence du caractère deux-points après **`if`** et **`else`** ! Rien n'est parfait !

2.3. Problème avec les interpréteurs interactifs.

Il n'y a pas de difficulté particulière avec **les programmes sous forme de fichiers**. La difficulté vient de l'interactivité ! Tous les interpréteurs de commande (tant sous Windows que sous Linux) indiquent leur disponibilité à saisir une commande par un prompt, qui est un caractère ou une suite de caractères convenus. On connaît bien le prompt par défaut du DOS-Microsoft avec le signe supérieur :

```
C:\WINDOWS\SYSTEM32>cd d:\python
```

Voici un exemple Linux (avec prompt utilisateur et prompt racine):



Le prompt de python est un triple supérieur plus un espace (4 caractères) : **>>> .** Dès lors, comment évaluer les alignements dans les commandes un peu complexes ?

Il y a évidemment deux solutions :

* soit on ne tient pas compte du prompt et l'on fait comme si on avait affaire à un fichier ; mais cela oblige à dédoubler la première indentation sur la ligne suivante, qui ne comporte pas de prompt. D'un point de vue visuel la première ligne est un peu boiteuse.

* soit on invente un prompt secondaire (trois points et un espace **...**) pour les lignes qui ne comporte pas le prompt principal.

IDLE sous Windows a opté pour la première solution ; d'autres éditeurs/interpréteurs pour la second ; ce n'est pas très important, mais il vaut mieux le savoir lorsque l'on se réfère à un texte de tutorial. Par exemple, dans les livres cités (Swinnen, Rossum) il y a systématiquement le prompt secondaire.

Voici ce que cela donne dans les deux cas :

IDLE WINDOWS

```
>>> if (cond):
        inst1
        inst2
else:
        inst3
        inst4
        inst5
  < ligne vide >
```

<suite de l'interaction, suivie du prompt>

Prompt secondaire

```
>>> if (cond):
...     inst1
...     inst2
... else:
...     inst3
...     inst4
...     inst5
...
```

<suite de l'interaction, suivie du prompt>

note : par la suite, et pour plus de simplicité, les saisie d'écran seront faites sur IDLE. Pour ce qui concerne les textes Python écrits directement, on utilisera la convention fichier, sans prompt, en écrivant avec un caractère à espacement fixe et en encadrant le code.

fiche 3 : premiers contrôles de code

introduction.

Tous les langage de programmation connaissent les ruptures de séquences, c'est à dire des dispositions du code qui permettent, si certaines conditions sont réalisées d'exécuter un bloc de code donné, et éventuellement un autre bloc code dans le cas contraire. Une telle rupture de séquence s'appelle **une conditionnelle**.

Un second type standard est **la boucle** : un bloc de code doit être exécuté tant qu'une certaine condition est remplie ; le bloc de code d'une boucle peut ne jamais être exécuté, ; il peut l'être au moins une fois, ou un nombre prédéfini de fois, un nombre de fois qui dépend d'un événement extérieur (frappe d'une touche au clavier par exemple), voire une infinité (toute relative) de fois.

Les auteurs de langage ont fait preuve d'une grande imagination quant à la forme syntaxique à donner à ces ruptures de séquence, et l'équipe de Python n'a pas failli en la matière. Il est vrai que les caractéristiques d'un langage incitent à moduler de façon spécifique les ruptures de séquences. Par exemple PHP qui est un langage qui use et abuse des tableaux associatifs (ou dictionnaires) est généreux en structure d'énumération de ces tableaux. Si le Pascal ou le C sont au départ très spartiates en matière de structure de rupture de séquence, la diversité s'est ensuite installée. Il va falloir plusieurs fiches pour examiner toutes les possibilités de Python en la matière.

1. La conditionnelle.

1.1. Un petit problème.

On interroge le client sur son âge ; une fois celui-ci rentré (un nombre !) le programme affiche si celui-ci est majeur et a la possibilité de voter.

1.2. Solution interactive.

```
>>> age = int(input("donnez votre âge : "))
donnez votre âge : 72
>>> if (age<18):
        print("vous êtes mineur")
        print("vous ne pouvez pas voter")
else:
        print("vous êtes majeur")
        print("n'oubliez pas de voter")


vous êtes majeur
n'oubliez pas de voter
>>>
```

remarquer :

* le cast sur le **`input()`** qui permet de saisir un nombre entier.
* les blocs, avec un second bloc contenant une ligne vide
* le symbole deux-points (:)
* l'absence de prompt secondaire sur IDLE/Windows.

Le **`else`** peut être omis et dans ce cas, une ligne vide est nécessaire pour terminer le premier bloc en mode interactif.

1.3. Un peu plus élaboré :

On demande aux mineurs de 17 ans de s'inscrire sur les listes électorales. Il y a alors deux conditionnelles imbriquées :

```
>>> age = int(input("donnez votre âge : "))
donnez votre âge : 17
>>> if (age<18):
        print("vous êtes mineur")
        print("vous ne pouvez pas voter")
        if (age==17):
                print("mais vous pourrez le faire l'an prochain")
                print("inscrivez-vous sur les listes électorales")
else:
        print("vous êtes majeur")
        print("n'oubliez pas de voter")


vous êtes mineur
vous ne pouvez pas voter
mais vous pourrez le faire l'an prochain
inscrivez-vous sur les listes électorales
>>>
```

noter que le **`else`** se rapporte au premier **`if`** : question d'alignement ! Bien faire attention également à l'imbrication des commandes conditionnelles.

Il est évident que pour la production, une telle disposition n'est pas acceptable. On va examiner dans le paragraphe qui suit quelques autres présentations du travail avec Python

2. Le travail avec Python sans interactivité immédiate.

2.1. Elaboration d'un programme enregistré avec IDLE.

En tout état de cause, il faut d'abord **élaborer une programme et l'enregistrer.**

Pour cela on peut utiliser IDLE en mode programme (éditeur avec coloration syntaxique) :

`options > configure IDLE > general > open edit windows > OK`

Relancer IDLE : c'est un éditeur débugueur (il recherche et détecte les erreurs) avec coloration syntaxique.

Ecrire le programme, le sauvegarder, l'exécuter (**`Run`**). Une console d'exécution (Shell) s'ouvre.

```
age = int(input("donnez votre âge : "))
if (age<18) :
        print("vous êtes encore mineur")
        print("vous ne pouvez pas voter")
        print("----")
        if (age==17) :
                print("vous pourrez voter l'année prochaîne")
                print("inscrivez-vous sur les listes électorales")
else:
        print("vous êtes majeur")
        print("n'oubliez pas de voter")
```

La seule précaution à prendre est de ne pas mélanger les espaces et les tabulations. Ce n'est pas interdit, mais cela vaut mieux pour la maintenance du texte.

On aurait pu élaborer le programme avec n'importe quel éditeur (Pspad; Gedit etc) et le charger (commande **`File > Open`**). L'éditeur peut avoir un faciliteur d'écriture (tabulation automatique, coloration syntaxique, contrôle du code). **Sauvegardé en UTF-8**,. le résultat est le même.

Rappel : ne pas mélanger espaces et tabulations dans l'éditeur !

La console comporte une commande Run qui, si ce n'est fait, ouvre une console IDLE interactive, et un nouveau shell. On peut exécuter le programme autant qu'on le veut, chaque fois une nouvelle session du shell est créée automatiquement. On a vu qu'il valait mieux fermer la fenêtre du shell si on utilise le **`input()`** en mode programme.

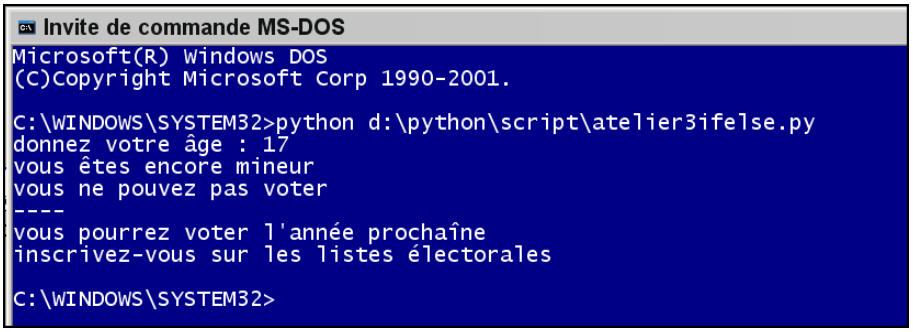
```
>>> ================================ RESTART ================================
>>>
donnez votre âge : 17
vous êtes encore mineur
vous ne pouvez pas voter
----
vous pourrez voter l'année prochaîne
inscrivez-vous sur les listes électorales
>>> ================================ RESTART ================================
>>>
donnez votre âge : 72
vous êtes majeur
n'oubliez pas de voter
>>> ================================ RESTART ================================
>>>
donnez votre âge : 14
vous êtes encore mineur
vous ne pouvez pas voter
----
>>>
```

2.2. exécution directe dans une console du système d'exploitation.

La première précaution à prendre, sans que ce soit une obligation, est de situer le chemin de Python dans le path du système. Un petit rappel pour Windows,

`Panneau de configuration > Systeme > Avancé > Variables d'environnement > `*`modifier le path.`*

On passe alors sous console et on tape la commande d'appel. Exemple sous Windows :



On rappelle que sous DOS, le path n'est valable que pour les fichiers de commande (.com, .exe, .bat). Evidemment, pour les mises au points complexes, un fichier batch peu être utile. Sous Linux on peu faire usage des liens symboliques pour éviter des frappes fastidieuses ; le path se situe dans **`/etc/profile`**.

3. le substitut de la commande case.

La structure de commande **`case`** (ou **`switch`**) se trouve dans de nombreux langages (Pascal, C, Java, PHP etc). Mais elle n'existe pas en Python. C'est une commande dont la destinées première est d'éviter les **`si...alors...sinon...`** en cascade.

Elle est remplacée par une complexification de la commande **`if`**, avec l'introduction du mot clés **`elif`** (raccourci pour **`else if`**).

On va le voir sur un exemple élémentaire :

```
nombre = int(input("entrer un nombre entier inférieur à 1000 : "))
#
#
if (nombre<0):
    print("ce nombre est négatif")
elif (nombre < 10):
    print("ce nombre est de l'ordre des unités")
elif (nombre <100):
    print("ce nombre est de l'ordre des dizaines")
elif (nombre <1000):
    print("ce nombre est de l'ordre des centaines")
else:
    print("on vous avait demandé un nombre inférieur à 1000")
```

On remarque que les divers cas sont examinés dans l'ordre et que si l'un d'entre eux est traité, on termine l'instruction. Le **`else`** récupère les cas résiduels ; il n'est pas obligatoire.

```
>>> ================================ RESTART ================================
>>>
entrer un nombre entier inférieur à 1000 : -5
ce nombre est négatif
>>> ================================ RESTART ================================
>>>
entrer un nombre entier inférieur à 1000 : 50
ce nombre est de l'ordre des dizaines
>>> ================================ RESTART ================================
>>>
entrer un nombre entier inférieur à 1000 : 2000
on vous avait demandé un nombre inférieur à 1000
>>>
```

4. La boucle Tant Que.

Un bloc peut voir son exécution répétée tant qu'une certaine condition est satisfaite. On teste d'abord la condition ; si elle est satisfaite, le bloc qui suit est exécuté et sinon on passe à la fin du bloc.

4.1. un exemple d'énumération.

On demande d'entrer un entier positif, puis d'afficher tous les multiples de 3 inférieurs à ce nombre.

```
print ("liste de multiples de 3")
borne = int(input("donner l'entier qui borne la liste : "))
#
# la boucle while :
#
variableDeBoucle = 0; # initialisation
#
while (variableDeBoucle <= borne) :
    if (variableDeBoucle + 3 > borne):
        print (variableDeBoucle)
    else:
        print (variableDeBoucle,"",sep=", ", end="")
    variableDeBoucle = variableDeBoucle + 3

# fin de programme
```

* une variable, appelée ici **`variableDeBoucle`** sert de régulateur au déroulement du programme ; à chaque exécution du **corps de boucle** qui est le bloc défini par les indentations, cette variable augmente de 3.

* le **`print()`** a un paramétrage différent selon qu'il s'agit du dernier élément de la liste ou non ; si c'est le dernier élément la variable a l'affichage par défaut (le séparateur est un espace, la fin d'affichage un passage à la ligne). Sinon, on affiche une chaîne vide, avec pour séparateur la virgule et un espace ; et la fin d'affichage est réduite à rien.

* dans le cas étudié, on sait, en entrant dans la boucle, combien de fois elle sera exécutée. Il existe dans les autres langages mais aussi en Python des structures de boucles appropriées à ce genre de situation (on les appelle les boucles **for**). Il faut donc prendre le modèle ci-dessus comme exemple de présentation, pas comme modèle d'algorithmique !

```
>>>
liste de multiples de 3
donner l'entier qui borne la liste : 47
0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45
>>>
>>> ================================ RESTART ================================
>>>
liste de multiples de 3
donner l'entier qui borne la liste : 51
0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51
>>>
```

4.2. un exemple où la durée de vie est imprévisible.

```
# initialisation
print ("la frappe de zéro stoppe l'affichage")
variableDeBoucle = int(input('tapez un chiffre : '))
#
# boucle à durée de vie imprésisible
while (variableDeBoucle != 0) :
    print ("c'est O.K.")
    variableDeBoucle = int(input('tapez un chiffre : '))
print("\n","vous avez interrompu le bouclage")
```

* la boucle peut ne pas être exécutée du tout.
* on a du doubler l'instruction d'entrée ; on pourrait espérer ne pas être soumis à cette contrainte !
* la boucle **while** comporte toujours une partie d'initialisation.

```
>>>
la frappe de zéro stoppe l'affichage
tapez un chiffre : 9
c'est O.K.
tapez un chiffre : 6
c'est O.K.
tapez un chiffre : 5
c'est O.K.
tapez un chiffre : 0

 vous avez interrompu le bouclage
>>>
```

5. rupture d'exécution d'une boucle.

Pour les puristes de l'algorithmique, un corps de boucle doit toujours être exécuté en entier. Il est vrai que c'est une bonne méthode pour le contrôle des algorithmes ; mais une contrainte qui alourdit souvent inutilement les programmes. Il existe deux instructions classiques, **break** et **continue** qui permettent de modifier le déroulement d'une boucle ; d'autre part, les boucles Python peuvent être munie d'une clause **else**.

5.1. l'instruction break.

Elle fait sortir de la boucle et donc la termine, quelle que soit la condition de boucle ; il faut donc toujours prendre avec précaution les variables de boucle à la sortie d'une boucle car il se peut que la condition de bouclage soit encore vraie.

5.2. l'instruction continue.

A la différence de l'instruction précédente, le déroulement du corps de boucle est stoppé, mais pas la

boucle. On retourne, en l'état, à l'évaluation du test de boucle. Tout ce qui a été modifié avant le **continue** reste comme il était quand on est arrivé sur l'instruction.

5.3. la clause else.

Cette clause fonctionne comme celle qui accompagne (éventuellement) le **if**. Elle est exécutée lorsque le test de boucle devient faux. Elle est sautée en cas de **break**. On ne l'utilise pas souvent.

5.4. un exemple de break.

Un nombre premier (autre que 2) est un nombre non divisible par un nombre qui lui est inférieur (sauf par 1). Un test de primarité peut limiter ses essais aux nombres impairs dont le carré est inférieur au nombre testé (la démonstration est évidente !). Un tel test est peu performant mais rapide à programmer. c'est l'objet de l'exercice suivant : on commence par vérifier si le nombre est impair, puis on boucle les essais sur les impairs de carrés inférieurs au nombre testé. Si on trouve un diviseur, on stoppe la boucle (**break**), et la clause **else** est exécutée uniquement en cas de primarité.

```
# primarité d'un nombre : un nombre premier est un nombre non pair
# et non divisible par un nombre inférieur à sa racine carrée.
examen = int(input("entrer un entier pour tester sa primalité : "))
#
# boucle d'examen
i=1
if (examen % 2 == 0):
    print ("le nombre entré",examen,"est pair","\n")
else:
    while (i*i < examen) :
        i = i + 2
        if (examen % i == 0):
            print ("le nombre entré est multiple de ",i)
            break
    else:
        print ("le nombre",examen,"est premier")

print('\n','fin du programme')
```

Trois essais :

```
>>>
entrer un entier pour tester sa primalité : 24
le nombre entré 24 est pair


 fin du programme
>>> ================================ RESTART ================================
>>>
entrer un entier pour tester sa primalité : 49
le nombre entré est multiple de  7

 fin du programme
>>> ================================ RESTART ================================
>>>
entrer un entier pour tester sa primalité : 71
le nombre 71 est premier

 fin du programme
>>>
```

5.5. un exemple avec "continue".

Cet exemple n'est pas un modèle de programmation ; son intérêt réside dans le jeu qui est fait de la conditionnelle et des instructions de rupture de boucle.

```
# on veut tester si un nombre supérieur à 10, entré au clavier, n'est pas
# multiple de 3, ni de 5, ni de 7
#
while True :
    examen = int(input("entrer un nombre au moins égal à 10 : "))
    if (examen < 10):
        continue
    else :
        if (examen % 3):
            if (examen % 5):
                if (examen % 7):
                    pass
                else:
                    print (examen,"admet 7 comme diviseur")
                    break
            else:
                print (examen,"admet 5 comme diviseur")
                break
        else:
            print (examen,"admet 3 comme diviseur")
            break
#
    print(examen,"est conforme")
    break
#
print("\n","fin de programme")
```

* La condition du **`while`** est un littéral booléen. Le parenthésage des conditions est obligatoire dans certains langages comme PHP ; il ne l'est pas en Python, même si c'est une pratique recommandable.

* Pour les **`if`**, la condition est représentée par un entier : le cast est automatique dans ce cas.

* On a ici un exemple d'utilisation d'une boucle infinie, dont on ne peut sortir que par un **`break`**.

```
>>>
entrer un nombre au moins égal à 10 : 6
entrer un nombre au moins égal à 10 : 8
entrer un nombre au moins égal à 10 : 14
14 admet 7 comme diviseur

 fin de programme
>>> ================================ RESTART ================================
>>>
entrer un nombre au moins égal à 10 : 22
22 est conforme

 fin de programme
>>> ================================ RESTART ================================
>>>
entrer un nombre au moins égal à 10 : 25
25 admet 5 comme diviseur

 fin de programme
>>>
```

6. Deux types d'erreur.

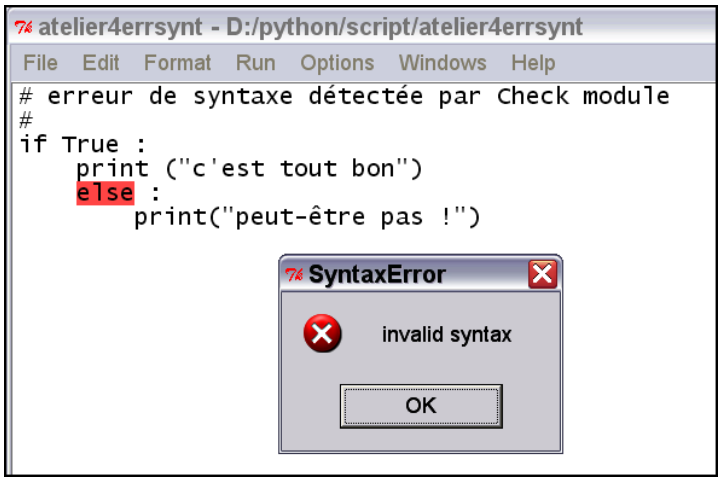
6.1. erreur de syntaxe.

Une erreur de syntaxe est une erreur dans l'écriture : une règle de présentation a été transgressée :

```
>>> print ("une erreur ,')

SyntaxError: EOL while scanning string literal (<pyshell#2>, line 1)
```

On est ici en mode interactif dans IDLE (<pyshell#2>) ; l'essai d'exécution de la ligne conduit à un message : Erreur de syntaxe : fin de ligne détectée lors de l'examen d'une chaîne littérale à la ligne 1.

L'analyseur ayant détecté une quote double attend une double quote fermante ; or il interprète la quote simple comme une apostrophe et va inutilement jusque la fin de la ligne ! L'erreur est fatale.



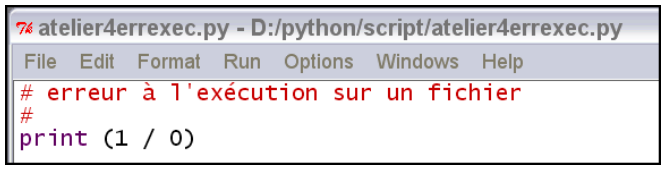
On a ici créé un fichier (**`D:/python/script/atelier4errsynt.py`**) grâce à IDLE et demandé l'exécution par **`Run > Check module (alt X)`**. Une fenêtre popup de IDLE signale une erreur de syntaxe et l'endroit où cette erreur est détectée ; le détail de l'erreur n'est pas précisée : il s'agit ici d'une erreur d'indentation, qui conduit à un **`else`** orphelin. L'erreur est fatale.

6.2. erreur d'exécution ou exception.

```
>>> print ( 1 / 0)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    print ( 1 / 0)
ZeroDivisionError: int division or modulo by zero
>>>
```

En mode interactif, sous IDLE, la ligne de commande est syntaxiquement correcte. Mais à l'exécution il y a une erreur mathématique, division par zéro, que la machine ne sait évidemment pas effectuer ! Le message d'erreur signale la dernière instruction dont on a demandé l'exécution et qui a provoqué l'erreur.

Voici maintenant un fichier à exécuter ; lui aussi est syntaxiquement correct, et la détection d'erreur avec **`Check module`** ne donne rien.



A l'exécution du fichier on a le message suivant dans une fenêtre IDLE :

```
>>>
Traceback (most recent call last):
  File "D:\python\script\atelier4errexec.py", line 3, in <module>
    print (1 / 0)
ZeroDivisionError: int division or modulo by zero
>>>
```

L'erreur a un identificateur : **`ZeroDivisionError`**. Les identificateurs prédéfinis de Python sont répertoriés dans l'aide (rubrique : Python Library Reference).

Les erreurs d'exécution peuvent être dues à une mauvaise conception du programme : c'est une affaire de programmation ! Il vaut mieux avoir un programme correct, qui ne plante pas. Mais certaines erreurs sont imprévisibles : si l'on demande de rentrer un entier par la fonction **`input()`** et que le client confond la lettre O et le zéro, une erreur se produit et plante le programme ; on aimerait dans ce cas avertir le client qu'il y a une erreur et lui redemander de recommencer par exemple, sans que le programme s'interrompe. On appelle cela **capturer** l'erreur. En programmation une erreur crée un événement spécifique, appelé exception. On dira donc plus correctement que l'on, **capture l'exception** quand l'erreur est détectée et qu'au lieu de planter l'exécution, la machine détourne le programme vers un bloc qui comporte le traitement de cette erreur.

La capture d'erreur est un mode de gestion du déroulement d'un programme, tout comme les clauses **`if, else elif, while`**...

7. capture d'une exception.

7.1. Bases de la capture d'exception.

* le premier élément à définir est le bloc à protéger du plantage. La clause « **`try`** » précédant un bloc indique que ce bloc peut donner lieu à exception. En cas d'erreur, l'exécution du bloc est arrêtée sur l'instruction fautive, mais le programme ne s'arrête pas et donne la main à un bloc de gestion d'erreur. Le programme ne s'arrête que s'il ne trouve pas de bloc qui gère l'exception.

* un bloc de gestion d'erreur est introduit par la clause **`except`** suivie de l'identificateur du type d'erreur qu'il est destiné à traiter. Il peut donc y avoir plusieurs blocs avec une clause **`except`** pour gérer les exceptions potentielles d'un seul bloc « **try** ». Si le mot-clef identifiant un type d'exception, n'est pas présent, le bloc récupère toutes les exceptions non encore récupérées. Ce n'est qu'un pis aller !

* il se peut que le bloc protégé ait mobilisé des ressources (fichiers, bases de données, mémoire…), et qui ne servent plus à rien : soit parce que tout s'est bien passé, soit parce qu'une exception s'est produite et que le traitement projeté n'a pas été mené à son terme. Il faut donc qu'il y ait un bloc qui s'exécute, avant de passer à autre chose, qu'il y ait eu exception ou pas. Ce bloc nettoyeur est introduit pas la clause **`finally`** ; il est facultatif. La clause **`finally`** est incompatible avec **`except`** ; même si l'exception n'est pas capturée, il faut gérer les ressources. Le fonctionnement de cette clause est différente de la clause du même nom dans Pascal, C++ ou Java.

* il se peut également que le bloc protégé soit suivi d'un bloc qui ne s'exécute que s'il n'y pas d'exception. Ce bloc alternatif aux blocs **`except`** est introduit par la clause **`else`**.

Ceci n'est qu'un premier regard sur les exceptions : par exemple on peut placer plusieurs identificateurs dans une même clause **`except`**, l'identificateur peut être accompagné de paramètres, on peut reporter une gestion d'exception sur une clause **`try`** extérieure ; on peut aussi définir des exceptions crées par l'utilisateur. Mais ceci correspond à des niveaux de programmation avancés.

7.2. Un exemple classique : protection d'une entrée clavier.

Le problème consiste à entrer un entier en se protégeant des erreurs de frappe. L'erreur de cast d'une chaîne en entier est **`ValueError`** (les identificateurs d'exception se terminent traditionnellement par **`Error`**).

atelier4valueerror.py - D:\python\script\atelier4valueerror.py

File Edit Format Run Options Windows Help

```
# protection d'une entrée clavier
#
boucle = True
while (boucle):
    try:
        valeur = input('tapez un entier : ')
        unEntier = int(valeur)
    except ValueError :
        print ('vous avez tapé "',valeur,'"',sep='')
        print ('ce n\'est pas un nombre entier ; recommencez \n')
    else:
        print ('"',unEntier,'" est un nombre entier', sep='')
        boucle = False
print('\n fin du programme')
```

Confusion de la lettre O et du zéro :

```
>>>
tapez un entier : 109
vous avez tapé "109"
ce n'est pas un nombre entier ; recommencez

tapez un entier : 109
"109" est un nombre entier

 fin du programme
>>>
```

7.3. Détection d'erreurs de programmation.

L'exemple qui suit n'est pas un cas qui pourrait se retrouver en production. Mais il arrive qu'un programme complexe présente, au cours de sa mise au point, l'une des erreurs classiques suivantes que l'on n'arrive pas à identifier : une variable n'a pas été initialisée, ou une variable a un type incompatible avec ce que l'on veut en faire. Ce qui suit est donc à prendre comme un cas d'école, puisque si on exécutait le programme sans intercepter les exceptions, le programme, en plantant, donnerait les mêmes renseignements.

On affiche donc la même chose que dans une exécution interactive, c'est à dire la trace de la pile d'exécution, avec indication de la ligne de détection d'erreur, nature de l'erreur etc. Le programme n'est pas arrêté.

atelier4miseaupoint.py - D:\python\script\atelier4miseaupoint.py

File Edit Format Run Options Windows Help

```
# exceptions de mis au point
#
import sys, traceback

variable = input ('tapez un entier : ')
try:
    print (2+ variable)
except:
    print("-"*70)
    traceback.print_exc(file=sys.stdout)
    print("-"*70)
#
#
try:
    print(3 * varable)
except:
    print("-"*70)
    traceback.print_exc(file=sys.stdout)
    print("-"*70)
#
#
print ("\nfin de programme")
```

Il y a deux erreurs dans ce programme. On a oublié de caster **`variable`** en entier ; on a donc une erreur de type dans l'addition "**`2 + variable`**" et plus loin, on a mal orthographié "**`variable`**", ce qui conduit à un défaut d'initialisation.

A l'exécution, voici ce que l'on obtient :

```
>>>
tapez un entier : 53
------------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\python\script\atelier4miseaupoint.py", line 7, in <module>
    print (2+ variable)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
------------------------------------------------------------------------
------------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\python\script\atelier4miseaupoint.py", line 15, in <module>
    print(3 * varable)
NameError: name 'varable' is not defined
------------------------------------------------------------------------

fin de programme
>>>
```

On note les **identificateurs d'exception** : **`TypeError`** et **`NameError`**.

8. Un mode de programmation.

Les débuts de la programmation par langages (Fortran, Cobol, Algol) peuvent être caractérisés, du point de vue des erreurs, par un soucis croissant de **sécuriser** les programmes. Les contrôles du code sont de plus en plus pointilleux : contrôle automatique de la syntaxe, contrôle des structures à la programmation, apparition des mots réservés, typage des variables, déclaration préalable des variables avec leur type, disparition du cast automatique etc.

Les contraintes dans l'écriture de programmes est maximale avec Ada (vers1980). La multiplication des contraintes a pour effet de rendre le code très lourd et rigide, ce qui rend les langages qui évoluent dans cette lignée (Pascal, C++, Java) mal adaptés à l'écriture de scripts ou à l'expérimentation. La maintenance n'est pas facilitée non plus.

Dans les langages de script modernes (Javascript, PHP, Ruby, Python, Pearl) le souci de la sécurisation demeure. Mais ces langages s'allègent d'un certain nombre de contraintes : les variables n'ont pas un type fixe et prédéfini, le nombre de types diminue, le cast automatique se développe quand il est prévisible, comme après un **`if`** ou un **`while`** (on attend un booléen).

La place de l'erreur dans la programmation évolue ; d'exceptionnelle parce qu'inévitable (aphorisme : "***l'erreur est humaine***") elle devient un élément presque normal de la programmation. L'auteur de Python développe l'aphorisme suivant : "***il vaut mieux demander pardon quelquefois que de demander constamment la permission***".

Par exemple, dans un langage traditionnel, avant de caster une chaîne en nombre, on examine si la transformation est licite ; dans un langage comme Python, on recommande d'essayer d'abord, et de voir s'il y a erreur, avec bien entendu la gestion *ad hoc* de l'erreur le cas échéant. Il est en effet probable dans un programme correctement écrit, que si on transforme une chaîne en nombre, c'est que cette chaîne est bien l'écriture d'un nombre ! Tout en ayant conscience qu'il peut y avoir des cas présentant une difficulté : une erreur de saisie clavier, une valeur trop grande, une confusion dans les symboles de l'écriture décimale (point ou virgule ?), etc. Comme en principe ce sont des événements rares, le programme gagne à ce que la vérification *a priori* soit remplacée par une gestion d'erreur un peu plus substantielle peut-être, mais destinée à être peu utilisée. La sécurité est ainsi conservée et le code plus léger.

fiche 4 : variable et mémoire

1. mémoire adressable.

1.1. des octets.

La mémoire "vive" de l'ordinateur (RAM) peut se représenter comme un tableau à une seule dimension : chaque case est un octet, et cet octet a un numéro appelé son adresse.

Un objet informatique "simple" est un objet qui peut être représenté par un octet ou une succession d'octets. L'exemple la plus élémentaire est le "petit entier" (*byte* dans certains langages), nombre compris entre 0 et 255, qui ne nécessite qu'un octet pour être représenté. Les "entiers relatifs" (*integer*) nécessitent plusieurs octets, et selon le langage, ce peut être 2 octets, 4, 8, 16, 32 octets ou davantage ; ainsi dans un langage qui code sur 4 octets, on dispose des entiers de -2147483648 à 2147483648 (2 à la puissance 4x8 = 32). Les nombres "flottants" peuvent prendre 6 octets… Quant aux chaînes de caractères chaque caractère est chiffré suivant un code (ISO par exemple) sur un octet ; en C classique, les caractères se suivent dans l'ordre de la chaîne ; un caractère de fin de chaîne la termine, de code 0 qui ne correspond à rien (en ISO). Il existe d'autres codage des chaînes, certains utilisant 2 octets (unicode UTF-16), ou un nombre variable d'octets (unicode UTF-8) pour chaque caractère.

On a symbolisé les adresses par les nombres des cases vertes et les octets par les cases grises.

Dans la réalité, la capacité mémoire s'exprime en MégaOctets, voire en GigaOctets

00	01	02	03	04	05	06	07
08	09	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

** Le tableau est rectangulaire par commodité.*

1.2. Adressage d'un objet.

Lorsque l'on veut accéder à un objet simple, il faut connaître deux choses : son adresse (celle du premier octet qui le constitue) et son encombrement (ou longueur). Pour l'encombrement, on va se centrer sur le cas où l'encombrement est connu car il est consigné dans une table. C'est le cas des langages interprétés comme Python.

Note : Dans le cas des langages compilés, la table d'encombrement est établie au niveau de la compilation, et les accès sont réglés en fonction des données de cette table qui n'est donc plus nécessaire lors de l'exécution.

1.2. Notion de variable.

Une variable est un identificateur qui est associé à un objet. Un identificateur est une suite de caractères alphabétiques, numériques et plus quelques symboles (comme le tiret bas, ou *underscore*). En général, majuscules et minuscules sont distingués, et les chiffres ne peuvent être en premier caractère. Voici quelques exemples :

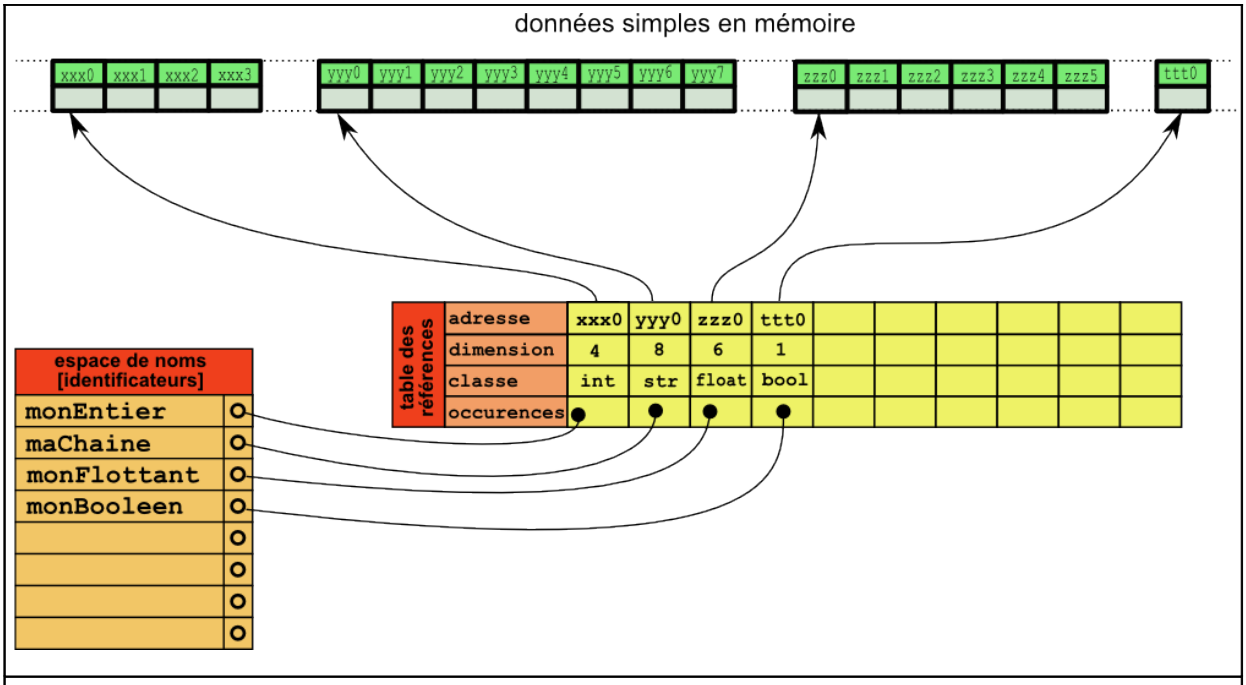
```
A,  a,  Abracadabra,  abracadaabra,  _avecUndescore,  a5_python,  _524
```

Les chaînes suivantes ne sont pas des identificateurs dans de nombreux langages. En effet, on y trouve un espace, un chiffre en tête, des caractères interdits comme les guillemets, les caractères accentué, l'apostrophe…:

```
program files,  8abc,  fenêtre,  trait-d'union,  "mot",  aujourd'hui
```

note : en Python, les caractères accentués sont admis ; mais ils sont déconseillés.

Il existe dans les interpréteurs une table qui à la manière d'un dictionnaire associe chaque identificateur à une adresse, elle-même associée à une longueur (encombrement) et d'autres renseignements encore. Les schémas utilisés dans ce chapitre sont d'abord explicatifs ; ils ne représentent la réalité que dans la mesure où ils renseignent le programmeur sur les mécanismes logiques mis en œuvre.



Les cellules de mémoire ont été représentées par des rectangles grisés. La table des références comporte une ligne où est repérable le nombre des "occurrences" d'utilisation de la donnée en mémoire. Une case vide signifie que la zone de donnée n'est pas accessible par le programme. Un lien unit l'identificateur et la référence associée ; un pointeur (un référence) permet d'accéder au contenu de la mémoire des données à partir du contenu de la table de référence.

2. Affectation.

2.1. Qu'est-ce qu'une affectation ?

* **Une affectation est d'abord une mise en relation d'un identificateur et d'un objet (qui peut être un objet "calculé")**. Les langages usuels connaissent tous l'affectation :

langage C, Python, Javascript, Java, php : `monId = 4 * 5 + 6`

Pascal, Delphi : `monId := 4 * 5 + 6`

Peu importe la syntaxe exacte : il y a un membre de gauche qui est un identificateur de variable ; et un membre de droite, en général une valeur ou une expression calculable, en tout état de cause, un objet (nombre entier, nombre flottant, chaîne de caractères etc).

L'identificateur se substitue alors à l'objet dans les usages qui en sont faits.

Par exemple : `monId = monId + 100`

Le membre de droite est d'abord évalué, la liaison est faite ensuite entre identificateur et objet généré.

* Mais au-delà de cette apparente simplicité se cachent d'autres comportements qui de plus différent d'un langage à l'autre :

- en C, Pascal, Java, on ne peut faire une affectation que si on a déclaré l'identificateur, ainsi que son type (un entier, un flottant, une chaîne de caractères, un booléen…). Comme chaque

type a un encombrement défini, il n'y a pas de table d'encombrement lors de l'exécution ; et si une affectation se fait avec un objet d'un mauvais type, il y a erreur de compilation.

- en C et en php, une affectation est une expression. Ce qui veut dire que la relation d'affectation **retourne** une valeur qui peut ensuite être un composant d'une nouvelle expression. Ainsi, on peut avoir une affectation dans une conditionnelle :

```
if (monId = tonId) { suite du programme ….
```

Ceci est très dangereux car peu intuitif : on ne teste pas si `moniD` et `tonId` sont égaux ! mais on affecte la valeur de `tonId` à `monId` et on examine si la variable `monId` est nulle ou pas !

- en Python et dans d'autres langages interprétés comme Javascript, il n'y a pas de déclaration préalable de variable. Cela oblige donc à avoir une table d'encombrement des variables lors de l'exécution. Mais une variable n'a pas de type fixe. Le type de la variable est celui de l'objet qui lui est affecté (on appelle cela le **typage dynamique**).

- Mais il y a plus important : **une affectation crée la variable, si elle n'a pas été créée auparavant (en principe par une autre affectation)**. Ainsi, si une variable est créée dans un corps de fonction et qu'il existe un paramètre de la fonction de même identificateur, celui-ci est occulté une fois l'affectation faite. Ce comportement est facile à comprendre mais ce n'est pas celui qui a cours en C, en Pascal, en Java... Méfiance donc si on a déjà travaillé avec ces langages.

2.2. le mécanisme en Python.

Ce point est extrêmement important et on va le voir en détail. Le premier cas à examiner est l'affectation du genre, à partir de la situation précédente :

```
monNouveau = 79 // 5
```

* le membre de droite est évalué ; on y trouve un quotient de deux entiers. Le résultat est de type entier ; la valeur de l'expression est 15, qui est enregistré sur 4 octets (au moins, c'est ce qu'on dit à ce stade).

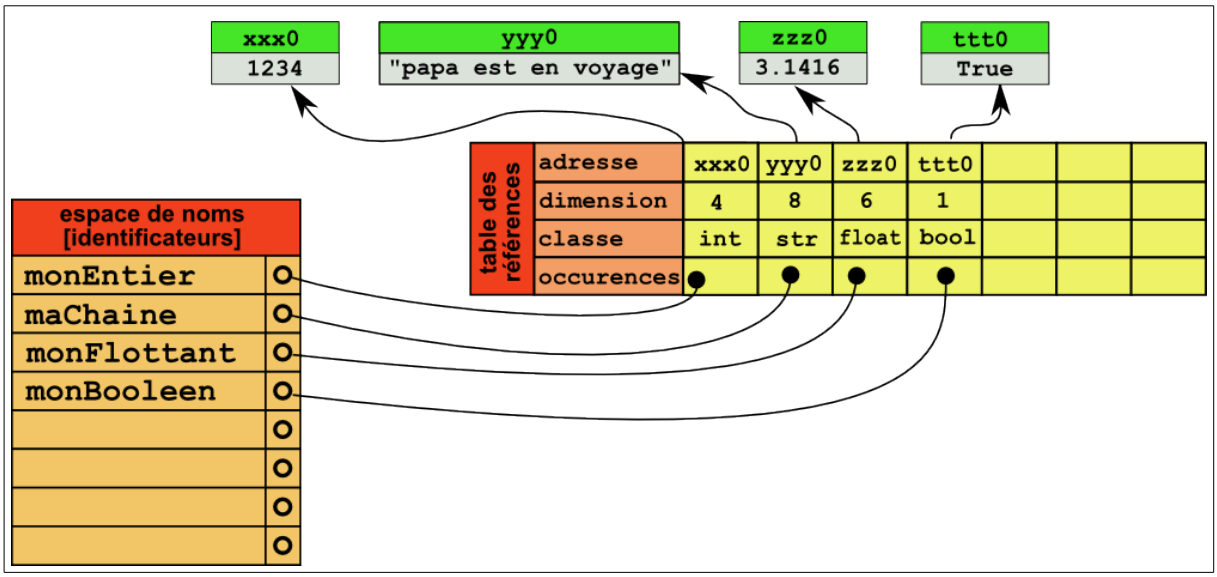
* une nouvelle occurrence est créée

* le membre de gauche est examiné : il ne se trouve pas dans la table des identificateurs. Il est donc créé.

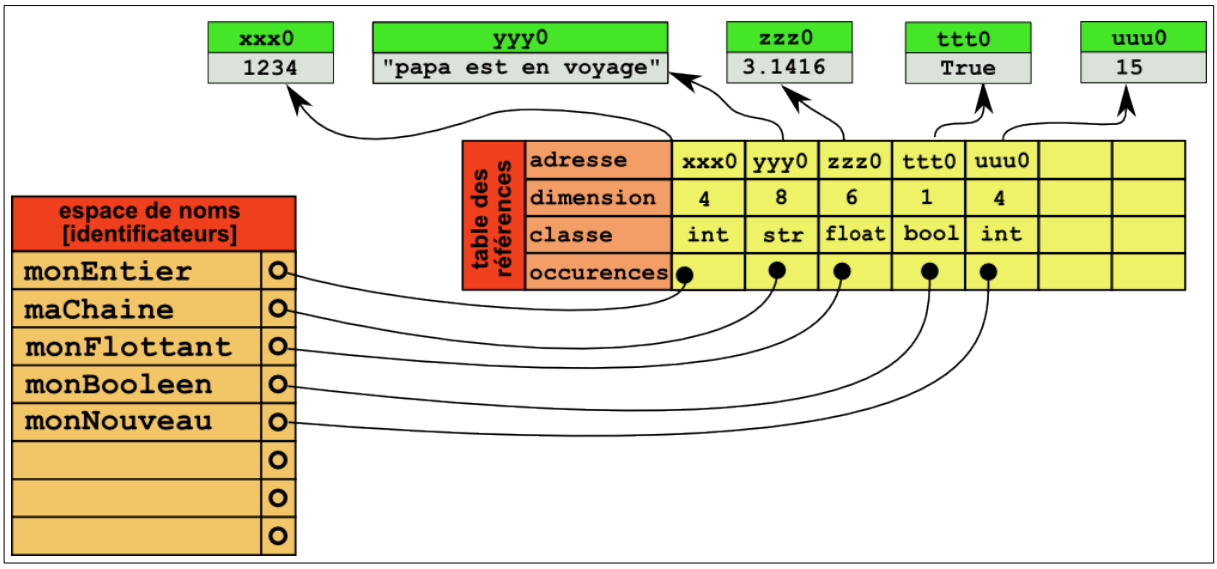
* le lien entre les tables est établi.

note sur la schématisation :

Pour simplifier les schéma et faciliter le repérage, on va désormais représenter une donnée simple en mémoire par un petit tableau, avec en haut l'adresse de la donnée et en bas la "valeur" codée en mémoire. On a donc, avant l'affectation :



et après l'affectation :



2.3. Notion d'alias.

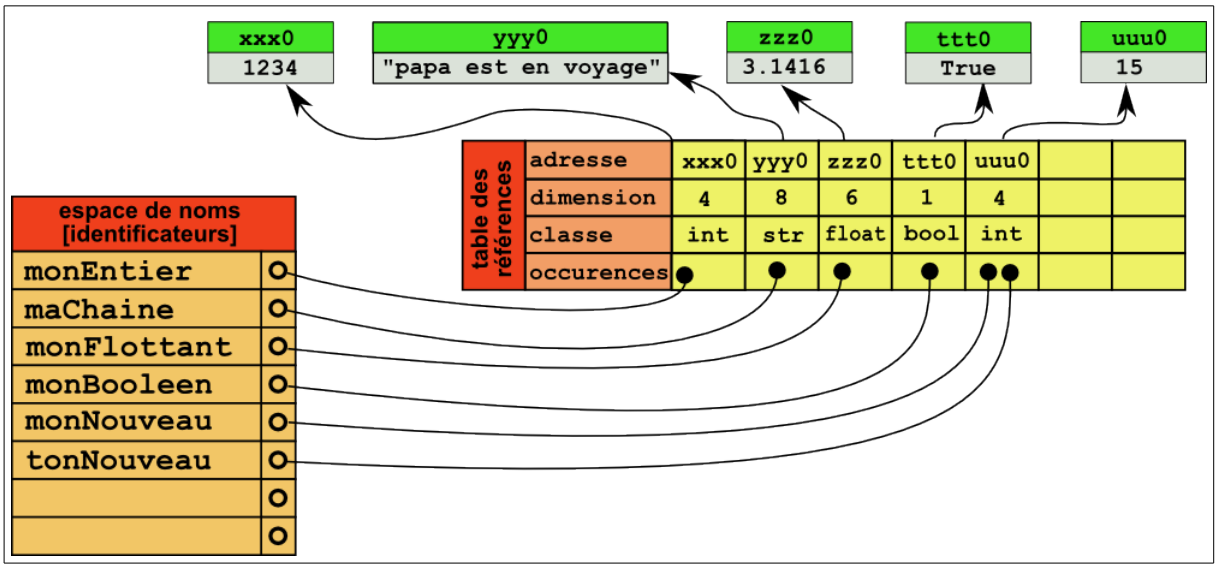
Comment interpréter la séquence qui suit, à partir de la situation précédente :

```
tonNouveau = monNouveau
```

Il y a au moins deux façons de comprendre cette séquence de deux expressions.

- à la façon du C ou du Pascal. Les deux identificateurs **monNouveau** et **tonNouveau** ont été déclarés : ce qui signifie que pour chacun, un espace mémoire a été réservé pour la valeur. Dans l'affectation, une copie de la valeur **de monNouveau** est réalisée dans l'espace mémoire de la second variable.

- dans les langages objets, il existe un objet **int** créé dans la mémoire ; il est affecté à la variable **monNouveau**. Mais l'affectation de **tonNouveau** se fait sur ce même objet. Les deux identificateurs sont donc deux noms pour ce même objet. On dit que **tonNouveau** est un alias de **monNouveau** (et réciproquement). Il y a deux occurrences sur la valeur de monNouveau.

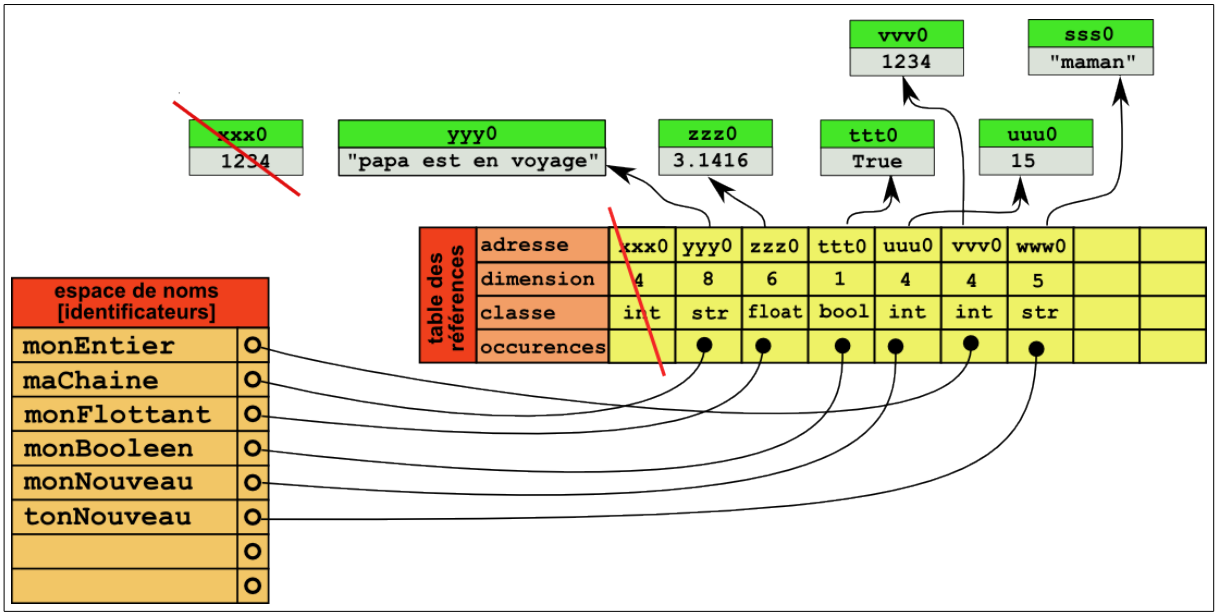


2.4. réaffectation.

Supposons que l'on ait les deux affectations suivantes :

```
monEntier = 1234
monNouveau= "maman"
```

Il faut appliquer scrupuleusement ce qui a été vu en 2.2. à cette différence près qu'il n'y a pas de nouvel identificateur dans la table des identificateurs. Attention, cela peut surprendre !



* un entier 1234 a été créé en mémoire à l'adresse **vvv0**.

* une chaîne "maman" également

* les nouvelles liaisons font que la référence d'adressage **uuu0** perd une occurrence. Mais également celle d'adresse xxx0 qui devient ainsi inaccessible, ainsi que la zone mémoire référencée (barré en rouge). Cette mémoire est perdue.

2.5. tableau résumé de l'affectation.

membre de gauche	membre de droite
l'identificateur existe dans la table des identificateurs : sa liaison est modifiée	le membre de droite est **un identificateur** : la liaison définit un alias
l'identificateur n'existe pas : il est créé ainsi qu'une liaison vers la table des références.	le membre de gauche est **une expression** (ou une constante littérale) : l'expression est évaluée et le résultat mis en mémoire. Une nouvelle référence apparaît dans la table des références.

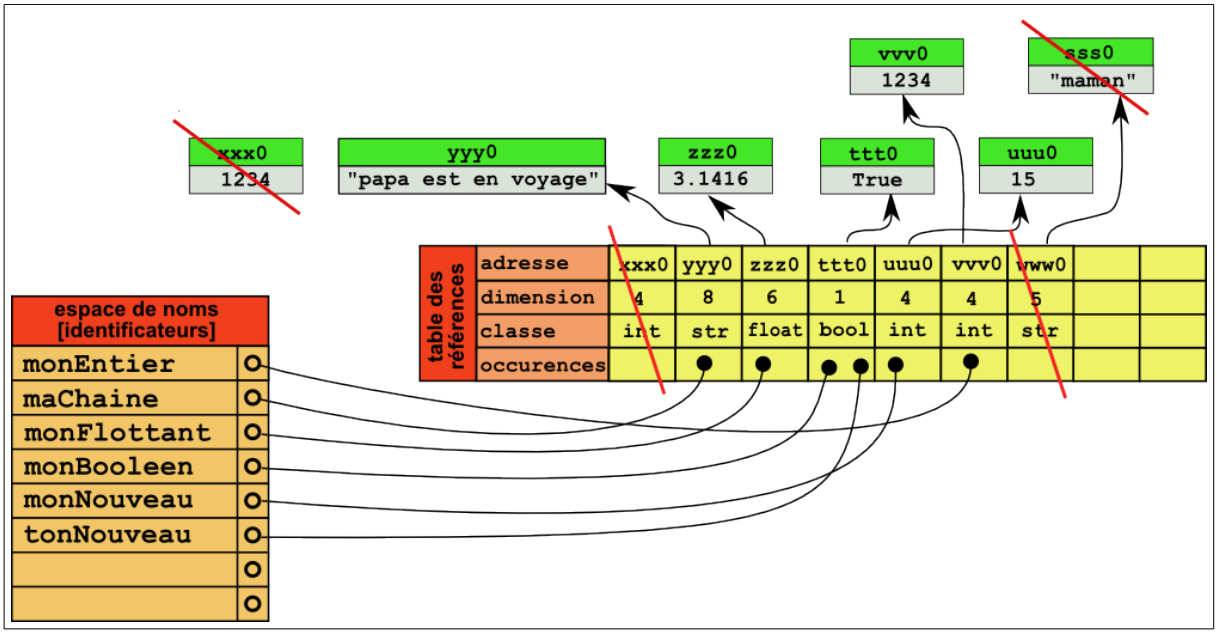
note : il y a 4 situations possibles.

Voici une illustration de la quatrième situation : l'identificateur existe et le membre de droite est un identificateur (alias) :

```
tonNouveau = maChaine
```

On remarque qu'une fois de plus, des plages mémoires sont rendues inaccessibles, et donc perdues. Cela n'est pas sans poser de problème, car comme on le sait, l'espace mémoire est une denrée rare.

On sent la nécessité de récupérer ces espaces perdus ; ceci est une autre affaire. Mais si avec les langages comme le C ou Pascal, ce problème doit être réglé par le programmeur -et c'est une source de difficulté majeure de ces langages- en Python il existe un mécanisme transparent pour le programmeur qui fait le travail. On l'appelle *ramasse-miettes* ou *garbage collector.*



Le mécanisme de l'affectation est fondamental dans la compréhension de Python. Il est différent de ce qui est habituellement décrit pour les autres langages : tout accès à la mémoire des données se fait à travers la table de références (unique). On ne saurait trop insister sur la nécessité de bien comprendre le processus avant d'aller plus loin.

fiche 5 : les listes

introduction.

On a rencontré jusqu'ici 4 types d'objets : entiers (**`int`**), flottants (**`float`**), chaînes (**`str`**) et booléens (**`bool`**).

```
>>> print (25,"type :",type(25))
25 type : <class 'int'>
>>> print (25.0,"type :",type(25.0))
25.0 type : <class 'float'>
>>> print ("toto","type :",type("toto"))
toto type : <class 'str'>
>>> print (False,"type :",type(False))
False type : <class 'bool'>
```

Tous ces types ont en commun d'avoir une représentation littérale et d'être définies dans le Python standard. Ils ne sont pas composés à partir d'autres objets (nous les avons qualifiés de "simples". Ils sont disponibles dans toutes les configurations.

Il existe des objets qui à la différence des précédents sont de objets complexes, c'est à dire fabriqués à partir d'objets préexistants, simples ou complexes. L'objet complexe le plus connu est le tableau indexé : il est composé d'objets de même type, de valeur variable, en nombre prédéfini non nul, et ordonnés (définition du Pascal). Lorsque les objets constituants ont une représentation littérale, l'objet composé peut avoir lui aussi une représentation littérale. Les tableaux indexés de nombres, de chaînes, de booléens ont une représentation littérale dans la plupart des langages où ils sont définis. En Python, il n'y a pas de tableau indexé du genre précisé précédemment ; les listes peuvent en tenir lieu. La structure de liste est plus riche que celle de tableau.

1. première approche des listes.

1.1. premières caractéristiques.

Une liste est un objets composé. Elle est **indexée à la manière d'un tableau**, à partir de zéro comme en Java, en C ou en PHP. Ses éléments peuvent être modifiés. Contrairement aux tableaux, le nombre d'éléments constitutifs peu varier, les éléments constitutifs ne sont pas nécessairement de même type et la liste peut être vide. De plus, à chaque indice valide correspond un objet, ce qui n'est pas toujours le cas des tableaux (on parle parfois de tableau creux pour désigner un tel tableau où à certains indices ne correspond aucune valeur). La représentation littérale se fait à l'aide de crochets. La désignation d'un élément ressemble à ce qui se fait pour les tableaux indexés.

1.2. Tableaux littéraux ; indexation.

écriture et affichage :

```
>>> listeUn = []
>>> listeDeux = [2, 3, 5, 7, 11, 13, 17, 19, 23]
>>> listeTrois = ["jambon", 2.5, True, "beurre", 2, False]
>>>
>>> print ("listeUn : ", listeUn)
listeUn :  []
>>> print ("listeDeux : ", listeDeux)
listeDeux :  [2, 3, 5, 7, 11, 13, 17, 19, 23]
>>> print ("listeTrois : ", listeTrois)
listeTrois :  ['jambon', 2.5, True, 'beurre', 2, False]
>>>
```

longueur d'une liste :

```
>>> # longueur des listes
>>> print ("nombre d'éléments : listeUn ",len(listeUn)
       ," listeDeux : ",len(listeDeux)
       ," listeTrois : ",len(listeTrois))
nombre d'éléments : listeUn  0  listeDeux :  9  listeTrois :  6
>>>
```

type des éléments :

```
>>>
>>> # type des éléments
>>> print("listeTrois, indice 3 : ",listeTrois[3],
          "   type(listeTrois[3] : ",type(listeTrois[3]))
listeTrois, indice 3 :  beurre    type(listeTrois[3] :  <class 'str'>
>>>
>>> print("listeTrois, indice 4 : ",listeTrois[4],
          "   type(listeTrois[4] : ",type(listeTrois[4]))
listeTrois, indice 4 :  2    type(listeTrois[4] :  <class 'int'>
>>>
>>> print("listeTrois, indice 5 : ",listeTrois[5],
          "   type(listeTrois[5] : ",type(listeTrois[5]))
listeTrois, indice 5 :  False    type(listeTrois[5] :  <class 'bool'>
>>>
```

1.3. variables.

```
>>> listeInser = ["un", "deux", "trois"]
>>> print (listeInser)
['un', 'deux', 'trois']
>>>
>>> listeEnglob = ["jambon", listeInsert, "beurre"]
>>> print (listeEnglob)
['jambon', ['un', 'deux', 'trois'], 'beurre']
>>> print ("type de l'élément d'indice 1 : ", type(listeEnglob[1]))
type de l'élément d'indice 1 :  <class 'list'>
>>>
```

* On peut déclarer une variable comme élément d'une liste ; même une variable qui représente une liste ! C'est **la valeur de la variable** lors de la définition qui est prise en compte. On peut, par exemple changer la valeur de **`listeInsert`** après la définition de **`listeEnglob`** ; cela est sans effet sur **`listeEnglob`** !

2. Travail sur les listes.

2.1. Lecture d'un élément d'une liste.

On a rencontré la lecture d'un élément d'une liste ; cela ressemble assez à la méthode utilisée pour les tableaux. Mais si on peut indexer à partir de 0 les éléments dans l'ordre naturel, on peut aussi les indexer à partir du dernier, en décroissant : le dernier élément a l'indice -1 et on compte ensuite en décroissant :

```
>>> lst = ["jambon", 2.5, True, "beurre", 3, False]
>>> print(lst)
['jambon', 2.5, True, 'beurre', 3, False]
>>>
>>> print (lst[0], lst[1], lst[2], lst[-3], lst[-2], lst[-1])
jambon 2.5 True beurre 3 False
>>>
```

2.2. Extraction d'une sous-liste.

```
>>>
>>> # sous-listes
>>> print (lst [1:-2])
[2.5, True, 'beurre']
>>>
>>> print (lst[:-2])
['jambon', 2.5, True, 'beurre']
>>>
>>> print (lst[-5:4])
[2.5, True, 'beurre']
>>>
>>>  print (lst[4:-5])

SyntaxError: unexpected indent (<pyshell#13>, line 1)
>>>
```

* l'indice 0 est posé par défaut en première valeur(deuxième exemple).

* si on peut indifféremment utiliser les indices positifs ou négatifs, il faut qu'ils correspondent effectivement à une sous-liste !

2.3. Changement d'un élément dans une liste.

```
>>> lst = [2, 3, 5, 9, 11, 13 ]
>>> print (lst)
[2, 3, 5, 9, 11, 13]
>>> lst[3] = 7
>>> print (lst)
[2, 3, 5, 7, 11, 13]
>>>
```

* **note :** ni le type de l'élément remplacé, ni celui du remplaçant n'ont pas à être pris en compte.

2.4. ajouter ou retrancher des éléments par "tranches".

```
>>>
>>> lst =["beurre", "jambon", "fromage"]
>>> print (lst)
['beurre', 'jambon', 'fromage']
>>> lst[1:1] = ["sausisson"]
>>> print (lst)
['beurre', 'sausisson', 'jambon', 'fromage']
>>> lst[3:3] = ["mayonnaise", "ketchup"]
>>> print (lst)
['beurre', 'sausisson', 'jambon', 'mayonnaise', 'ketchup', 'fromage']
>>> # suppression
>>> lst[3:4]=[]
>>> print (lst)
['beurre', 'sausisson', 'jambon', 'ketchup', 'fromage']
>>> lst[3:3]=[]
>>> print (lst)
['beurre', 'sausisson', 'jambon', 'ketchup', 'fromage']
>>> lst[3:5] = []
>>> print (lst)
['beurre', 'sausisson', 'jambon']
>>> lst[0:0] = lst
>>> print (lst)
['beurre', 'sausisson', 'jambon', 'beurre', 'sausisson', 'jambon']
>>>
```

* la substitution de tranche comporte deux indices :

- le premier indique où se place le premier élément de la tranche dans la liste d'accueil.

- le second indique **le premier élément de la liste d'accueil conservé** ; si le second élément est la longueur de la liste d'accueil, il ne correspond pas à un élément, mais la liste d'ajout se

met en queue de la liste d'accueil (il y a concaténation). Il existe une méthode plus naturelle de concaténation, l'opérateur binaire +

```
>>> accueil = ["beurre", "jambon", "sausisson"]
>>> ajout = ["mayonnaise", "ketchup"]
>>> acceuil = accueil + ajout
>>> print (accueil)
['beurre', 'jambon', 'sausisson']
>>> accueil = accueil + ajout
>>> print (accueil)
['beurre', 'jambon', 'sausisson', 'mayonnaise', 'ketchup']
>>> accueil[0:3]=[]
>>> print (accueil)
['mayonnaise', 'ketchup']
>>> accueil = accueil + accueil
>>> print (accueil)
['mayonnaise', 'ketchup', 'mayonnaise', 'ketchup']
>>>
```

Note.

Il existe une commande qui permet de réaliser l'équivalent de **`lst[3:4]=[]`** (substitution d'une liste vide à une sous-liste, c'est-à-dire suppression de cette sous-liste) : **`del lst[3:4]`**.

3. Une clause d'itération fonctionnant sur les listes : for

La clause **`for`** est une clause d'itération familière (Pascal, C, PHP, Java, Javascript). Mais elle a une programmation différente en Python : elle fonctionne par balayage de son argument (ici une liste). La variable de boucle (donnée commune avec les autres langages) prend successivement toutes les valeurs des éléments de l'argument donné après le **`in`** de la commande. Il vaut mieux travailler sur une liste qui n'évolue pas sur le corps de boucle ; ce n'est pas interdit, mais peut conduire à des résultats bizarres ou des exceptions, car **le nombre d'itérations est calculé sur la liste de départ**.

3.1. Exemple fondamental.

```
#
# balayage de liste
liste = ["jambon", "beurre", 1, 3.0, False, [9, 8, 7, 6, 5, 4, 3, 2, 1]]
#
for x in liste :
    print (x,type(x), end=" ")
    try :
        print ("longueur : ",len(x),end="")
    except :
        pass

    print ("",end="\n")

#
# fin de programme
```

* problème : une liste est donnée ; afficher chaque élément, son type et le cas échéant, sa longueur (nombre d'éléments pour une liste, de caractères pour une chaîne). Les nombres, les booléens n'ont pas de longueur, et l'application de la fonction **`len()`** à un nombre ou un booléen provoque une exception.

* on note une clause non encore vue, **`pass`**, qui est un bloc vide ; il n'y a en effet rien à faire si l'exception se produit. Mais comme la clause **`except`** est nécessairement suivie d'un bloc, il faut que celui-ci soit explicitement présent !

* **attention :** il ne peut y avoir de parenthèses après le mot-clef **`for`** alors que la condition du **`if, elif, while`** peut être parenthésée.

```
>>>
jambon <class 'str'> longueur :  6
beurre <class 'str'> longueur :  6
1 <class 'int'>
3.0 <class 'float'>
False <class 'bool'>
[9, 8, 7, 6, 5, 4, 3, 2, 1] <class 'list'> longueur :  9
>>>
```

3.2. Exercice sur les nombres premiers.

énoncé : afficher les 100 premiers nombres premiers sous forme de 10 sous-listes.

* **principe :** un nombre premier autre que 2 est un impair (**test**) non divisible par aucun nombre premier (**p**) inférieur à sa racine carrée.

* **initialisations :** la liste cherchée est la variable **liste**, dont le premier élément est 3 ; la valeur 2 est rajoutée en fin de parcours. On va tester les impairs à partir de 3 jusqu'à obtenir les 100 valeurs désirées (2 compris).

La longueur de la liste pourrait être recalculée avec la fonction **len(liste)** ; mais il est plus naturel d'utiliser un compteur **lgListe**, incrémenté à chaque fois que l'on trouve un nombre premier que l'on ajoute en queue de liste.

* **boucles** : La boucle while comporte deux instructions. La première incrémente **test** à chaque fois qu'on a un résultat sur la valeur précédente. La seconde est un test de primarité réalisé avec la boucle **for** portant sur l'état actuel de la liste des nombres premiers. Soit on trouve un diviseur premier de la variable **testée** et on arrête les frais (**break**). Soit on ne trouve pas de diviseur premier dont le carré soit inférieur au nombre testé : on en conclut que le nombre testé est premier et on l'ajoute en fin de liste.

```
#
# liste des premiers nombres premiers
#
liste = [3] # début de la liste des entiers > 2
test = 3    # début de la suite des nombres à tester
lgListe = 1 # longueur de la liste
#
while lgListe < 100 :
    test = test + 2
    for p in liste :
        if (test % p == 0) :  # attention
            break
        if (p*p > test) :
            lgListe = lgListe+1
            liste= liste + [test]
            break


# affichage par sous-listes de 10 éléments

liste[0:0] = [2] # rajouter 2 à la liste

for n in list(range(0,99,10)) :
  print (liste [n:n+10])

# fin de programme
```

* Pour l'affichage on utilise la fonction **range()** avec un paramétrage complet. Elle fournit une suite de nombres en progression arithmétique : le premier est fixé par le premier paramètre, la borne à ne pas atteindre par le second (ici, tout nombre entre 91 et 100 convient), et le pas de progression par le troisième. Il existe des paramétrages simplifiés avec un départ et un pas de progression facultatifs difficiles à utiliser, sauf la forme à un paramètre : **range(n)** est la suite des nombre de 0 à (n-1).

 range() ne retourne pas une liste mais un objet qui n'a pas de représentation littérale. Il faut donc caster le résultat avec `list()` pour obtenir une liste (comparer `print(range(10))` et `print(list(range(10)))` avec la version 3). Le comportement a évolué avec la version 3 de Python. Noter qu'ici, l'écriture `for x in range(0,99,10)` aurait cependant donné un résultat juste, car la boucle for fonctionne aussi avec l'objet range().

```
>>>
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
[31, 37, 41, 43, 47, 53, 59, 61, 67, 71]
[73, 79, 83, 89, 97, 101, 103, 107, 109, 113]
[127, 131, 137, 139, 149, 151, 157, 163, 167, 173]
[179, 181, 191, 193, 197, 199, 211, 223, 227, 229]
[233, 239, 241, 251, 257, 263, 269, 271, 277, 281]
[283, 293, 307, 311, 313, 317, 331, 337, 347, 349]
[353, 359, 367, 373, 379, 383, 389, 397, 401, 409]
[419, 421, 431, 433, 439, 443, 449, 457, 461, 463]
[467, 479, 487, 491, 499, 503, 509, 521, 523, 541]
>>>
```

Remarque : Le cast avec `list()` est réalisable aussi avec une chaîne :

```
>>>
>>> maListe = list ("abracadabra")
>>> print (maListe)
['a', 'b', 'r', 'a', 'c', 'a', 'd', 'a', 'b', 'r', 'a']
>>>
```

4. question de mémoire.

On ne comprend réellement le fonctionnement des listes sous Python que si on étudie comment elles sont représentées en mémoire.

4.1. Création d'une variable liste.

Supposons que l'on ait les trois affectations suivantes :

```
monEntier = 789
uneChaine = "abracadabbra"
laListe = [4562, uneChaine, 3.14]
```

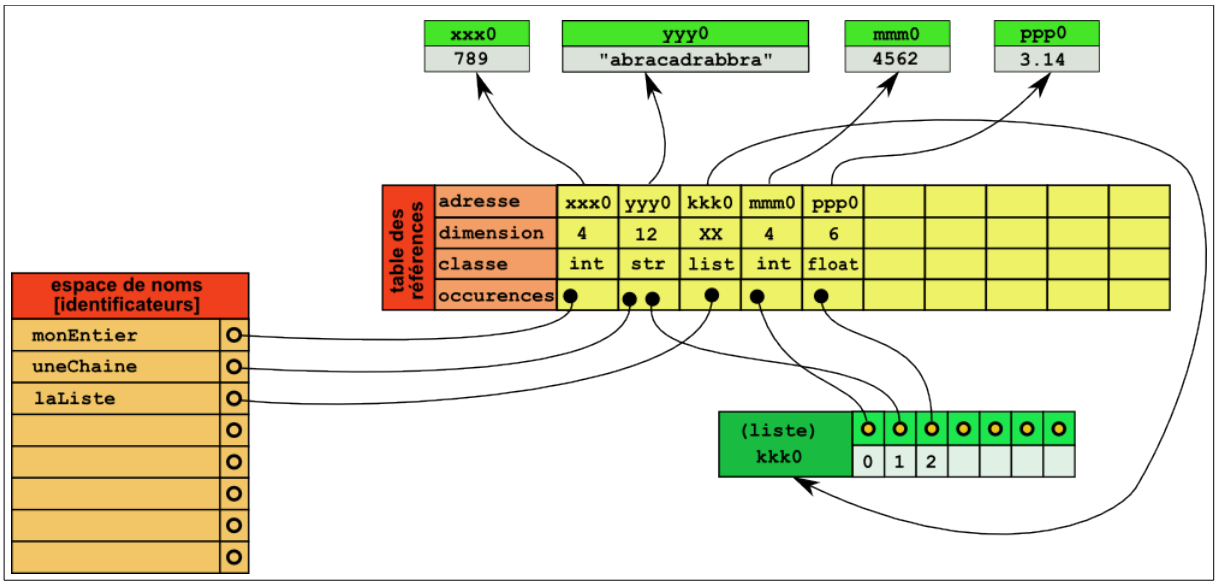
On a trois variables nouvelles : `monEntier, uneChaine` et `laListe.`

Il y a donc une triple création de réservation mémoire.

Celle pour la liste a une structuration particulière. Elle est codée dans un champ de la mémoire comme toutes les données, et ce champ est segmenté. Chaque segment peut représenter un item de la liste ; dans ce cas il est associé à une référence de la table de références. Chaque segment utilisé est numéroté (ce qui autorise l'accès à la propriété "nombre d'éléments de la liste"). La réservation en segments dépasse le nombre strictement nécessaire, permettant une manipulation directe de la liste (ajout d'éléments).

Chaque item de la liste fonctionne comme une variable ; à ceci près qu'il n'y a pas d'identificateur . l'accès aux éléments de la liste se fait sous la forme `laListe[0], laListe[1], laListe[2]`

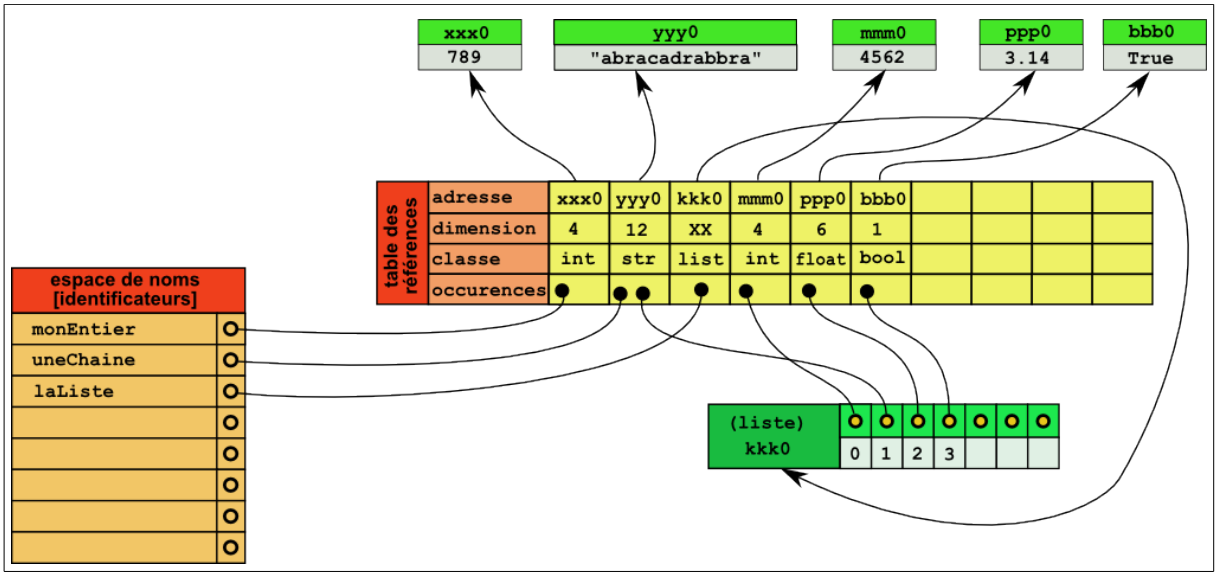
La structure globale présente une certaine complexité ; sa compréhension est absolument nécessaire pour la suite.



4.2. Ajout d'un élément simple à une liste.

Soit l'affectation suivante :

```
laListe[3:3] = [True]
```



Apparemment, il n'y a pas grand chose de neuf. Mais on a ici un processus qui n'a pas encore été rencontré : **la donnée "liste" en mémoire a été modifiée**. On dit que la liste est un type de données modifiable. On aurait pu de même changer un élément. On va le faire dans un cas assez complexe, celui où l'élément de substitution est lui-même une liste (littérale pour rendre le schéma lisible ; mais il serait intéressant de reprendre le schéma avec un liste comportant elle-même des variables).

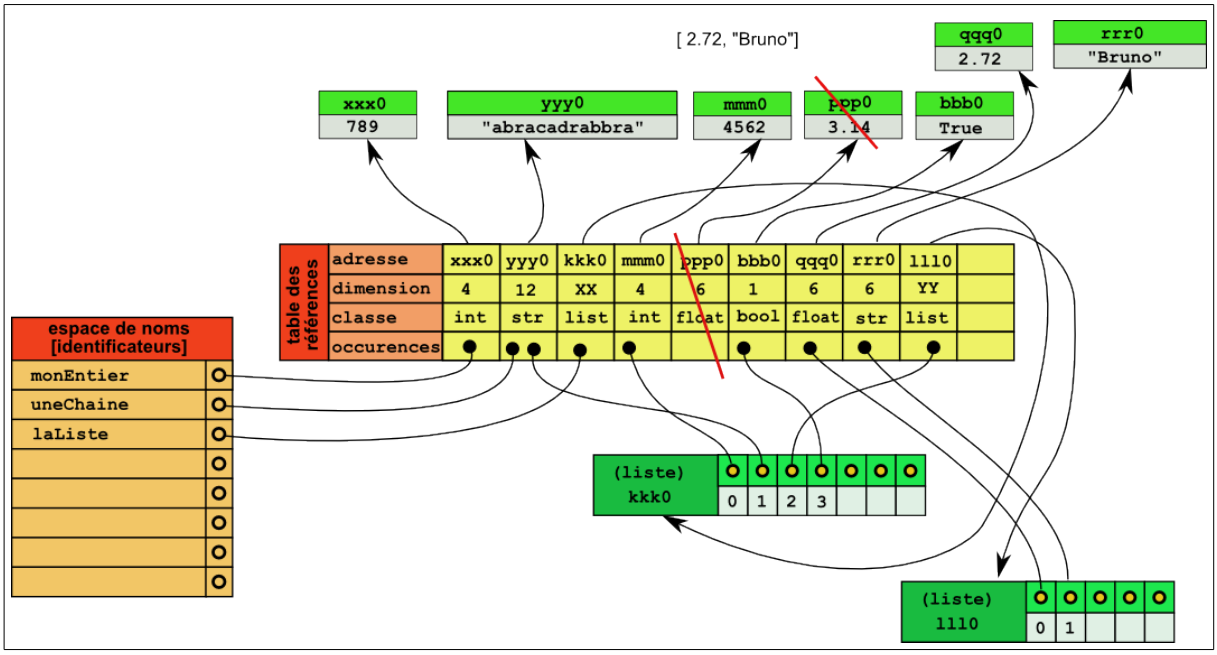
On aurait eu un résultat complètement différent si on avait donné l'instruction :

```
laListe = laListe + [True]
```

En effet, l'évaluation du membre de droite aurait donné lieu à la création d'une nouvelle liste. La réservation mémoire à l'adresse **`kkk0`** aurait été perdue dans l'affectation. Apparemment, le résultat aurait été similaire mais le processus complètement différent.

4.3. substitution d'un élément.

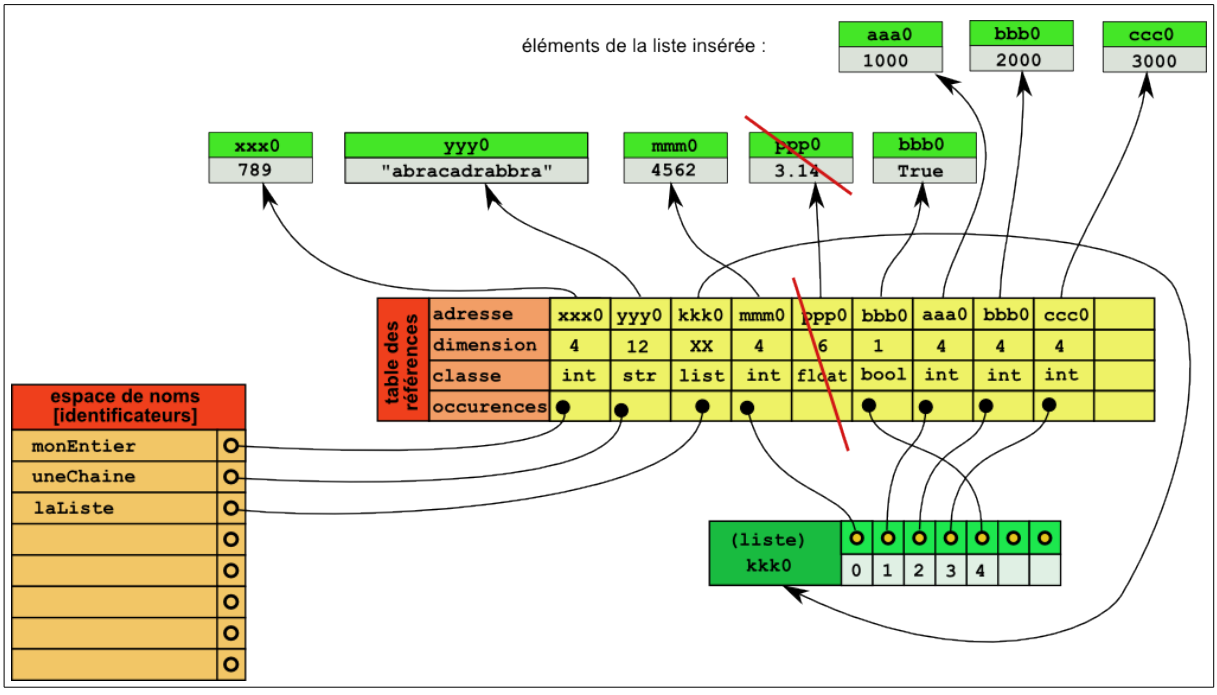
```
laListe[2] = [2.72, "Bruno"]
```



4.4. Substitution d'une sous-liste.

On revient à la situation du 4.2. où laListe a 4 éléments : `[789, "abracadabbra", 3,14]`.

```
laListe[1:2] = [1000, 2000, 3000]
```

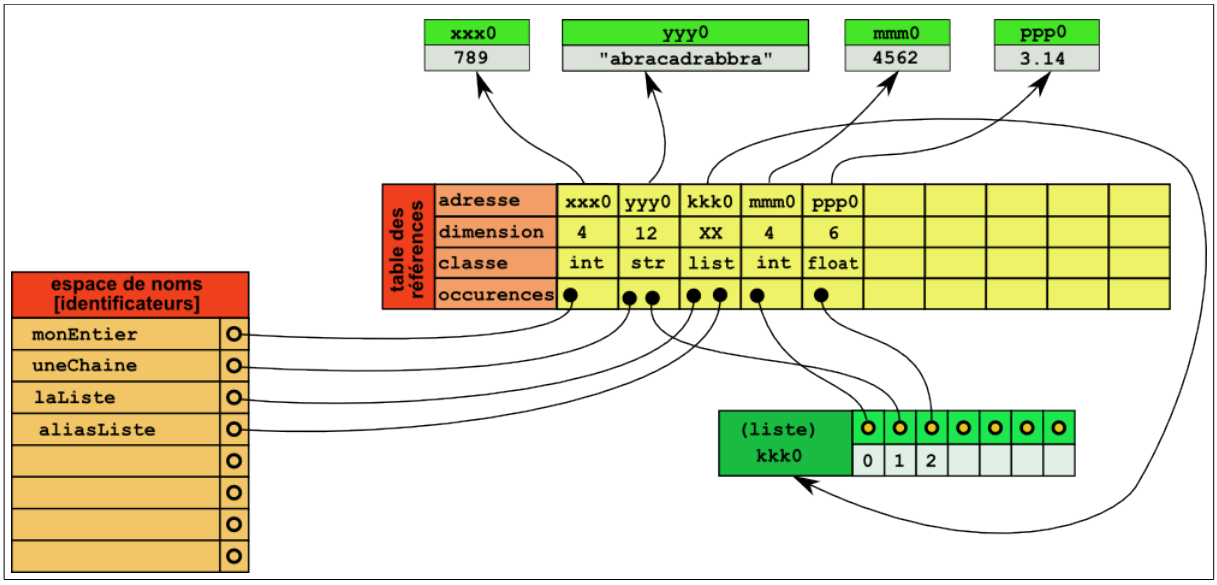


```
>>> monEntier = 789
>>> uneChaine= "abracadabbra"
>>> laListe  = [4562, uneChaine, 3.14]
>>> print (laListe)
[4562, 'abracadabbra', 3.1400000000000001]
>>> laListe[3:3] = [True]
>>> print (laListe)
[4562, 'abracadabbra', 3.1400000000000001, True]
>>> laListe[1:2] = [1000, 2000, 3000]
>>> print (laListe)
[4562, 1000, 2000, 3000, 3.1400000000000001, True]
>>>
```

4.5. exemple avec un alias.

```
monEntier = 789
uneChaine = "abracadabbra"
laListe = [4562, uneChaine, 3.14]
```

aliasListe = laListe



Soit maintenant l'instruction :

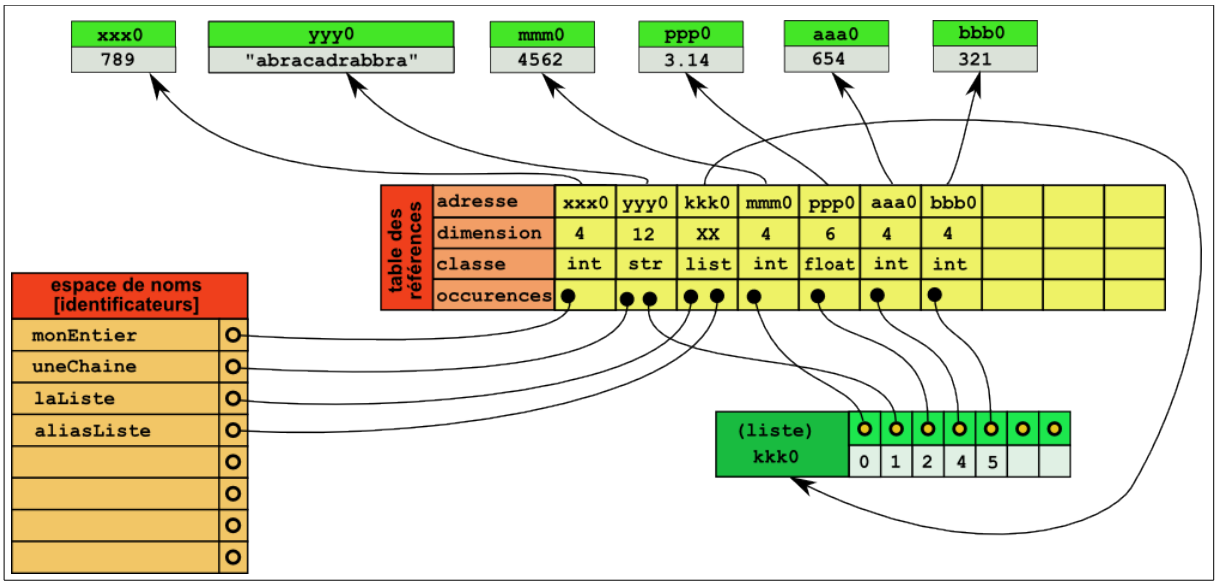
```
aliasListe[3:3] =[654, 321]
```

Ou encore l'instruction :

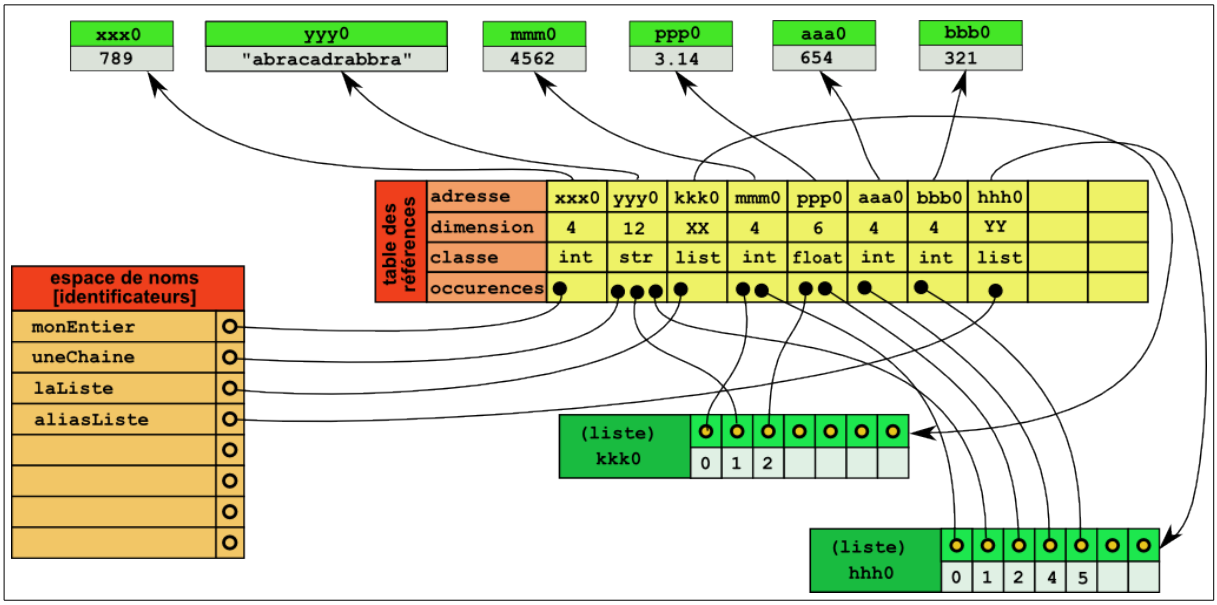
```
aliasListe = aliasListe +[654, 321]
```

Ces deux cas sont complètement différents : en effet, **`aliasListe`** prend la même valeur ; il n'en est pas de même de **`laListe`** !

premier cas :



deuxième cas :



fiche 6 : chaînes et caractères, n-uplets

On a déjà utilisé le type "chaînes de caractères", structure obligée de la programmation. La syntaxe et l'implémentation des chaînes sont très différentes d'un langage à l'autre. Dans Python elle est assez singulière.

1. Faire une chaîne de caractère.

1.1. Ecriture littérale.

Une chaîne de caractère est une suite (c'est une "séquence"), éventuellement vide, de caractères. Pour représenter littéralement une chaîne littérale, voici trois modes pratiques :

- soit on met la suite entre quotes simples ;
- soit on la met entre quotes doubles ;
- soit ont la met entre triple quotes doubles.

Une des difficultés qui se présente est évidemment la présence de quotes simples (apostrophes) ou de quotes doubles (guillemets) dans la chaîne elle-même. Dans le premier cas, les quotes doubles sont correctement interprétées ; dans le deuxième, ce sont les quotes simples, et dans les derniers cas, les deux (sauf la suite improbable d'au moins trois quotes doubles).

```
>>> 'un mot "réservé" ne peut être un identificateur'
'un mot "réservé" ne peut être un identificateur'
>>> "sur l'appareil, on troube le label NF"
"sur l'appareil, on troube le label NF"
>>> """sur l'appareil, le label "made in china" est possible"""
'sur l\'appareil, le label "made in china" est possible'
>>>
```

1.2. L'antislash.

Il arrive qu'en écrivant une instruction ou un programme Python, on n'ait pas assez de place sur la ligne pour écrire une instruction ; on peut aussi désirer l'écrire sur plusieurs lignes pour des raisons de commodité ou de clarté. Dans ce cas on utilise **l'antislash**, qui indique que la ligne Python se continue sur la ligne physique suivante.

```
>>> print ("Pierre",      # premier apôtre \
           "Jean",        # l'apôtre le plus jeune \
           "Juda",        # dit l'Iscariote \
           sep=" ... "  # le séparateur d'affichage \
          )
Pierre ... Jean ... Juda
>>>
```

Le même artifice peut être utilisé à l'intérieur d'une chaîne. L'antislash joue à la manière d'une espèce de trait d'union : la ligne Python s'arrête sur l'antislash et recommence sur le premier caractère de la ligne physique suivante :

```
>>> print ("la raison du plus fort \
est toujours la meilleure")
la raison du plus fort est toujours la meilleure
>>>
```

1.3. Echappements.

Il y a donc trois caractères litigieux à l'intérieur des chaînes de caractères : la quote simple, la quote

double et l'antislash. Une façon générale de les insérer dans une chaîne est de les échapper ; le caractère d'échappement est l'antislash.

```
>>>
>>> str = "les trois caractères \", \', \\ sont litigieux"
>>> print (str)
les trois caractères ", ', \ sont litigieux
>>>
```

On a déjà rencontré un quatrième cas d'échappement, le **\n**, qui désigne le caractère "saut de ligne". Il y a d'autres cas d'échappement, liés à la représentation binaire des caractères.

L'identificateur **str** n'est pas réservé en Python, comme aucun nom de type.

1.4. conversion entier/caractère.

Il existe une manière de récupérer une chaîne à un caractère à partir du code de ce caractère. La fonction chr() prend un entier et retourne une chaîne à un seul caractère dont le code (unicode, et donc iso) est passé en paramètre. Par exemple **chr(233)** donne **"é"**

1.5. opérateurs sur les chaînes.

Les chaînes peuvent être concaténées (opérateur +) ou itérées (opérateur *)

```
>>>
>>> ch1="abracadabra"
>>> ch2="<>"
>>> print (ch2+ch1+ch2)
<>abracadabra<>
>>> print (ch2*20)
<><><><><><><><><><><><><><><><><><><><><><><><><><><><>
>>>
```

2. Accès aux caractères d'une chaîne.

2.1. Extraction d'une sous-chaîne.

Une chaîne (non vide) est une séquence de caractères. Les caractères d'une chaîne sont donc indexés à partir de zéro. **Le caractère n'est cependant pas un objet en Python** ; si on veut lire un caractère dans une chaîne, ce qu'on lit, c'est une chaîne formée de seul caractère (et que l'on manipule comme chaîne). La technique est identique à celle des listes, qui sont aussi des séquences.

```
>>> chn = "abracadabra"
>>> print (chn[4], type(chn[4]))
c <class 'str'>
>>> print (chn[2:5])
rac
>>> print (chn[:5])
abrac
>>> print (chn[3:])
acadabra
>>> print (len(chn))
11
>>> print (chn[11])
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    print (chn[11])
IndexError: string index out of range
>>> print (chn[3:-4])
acad
>>>
```

2.2. Fixité des chaînes.

Les chaînes de caractères ne sont pas modifiables : on peut lire un caractère, mais ni le supprimer, ni le changer (plus de détails dans la fiche sur les méthodes). On ne peut que simuler la modification d'un chaîne en la remplaçant par une autre, en utilisant par exemple la concaténation et l'extraction de sous-chaînes :

```
>>> chn="abracadebra"
>>> chn = chn[:7]+'a'+chn[8:]
>>> print (chn)
abracadabra
>>>
```

Attention cependant à la représentation en mémoire !

3. Itération sur une chaîne.

Comme sur les listes, on peut utiliser la clause **`for`** en itérant sur les caractères d'une chaîne. Voici un exemple de codage appelé ROT13, qui consiste à décaler de 13 caractères les éléments d'une chaîne (ASCII). Si on applique deux fois le codage on retrouve la chaîne initiale. C'est un algorithme utilisé pour éviter les lectures (de mail par exemple) "par dessus l'épaule". Ceci est un exercice d'école ; en production, on utiliserait des primitives plus performantes.

```
#
# l'algorithme ROT13
#
alpha ='abcdefghijklmnopqrstuvwxyz'
print (alpha,"longueur : ",len(alpha),"\n")
#
chn ="les cons osent tout ; c'est à cela qu'on les reconnaît"
chn13=""
for x in chn :                          # itèration sur les caractères de chn
    c=0                                 # gestion de l'index sur rot
    for y in alpha :                    # itération sur les caractères de alpha
        if (x==y) :                     # caractère trouvé dans alpha
            sy = alpha[(c+13) % 26]     # décalage de 13
            chn13 = chn13 + sy
            break
        else :
            if c==25 :                  # caractère absent ; on recopie
              chn13 = chn13 + x
              break
        c = c + 1                       # fin de traitement d'un caractère

#
print (chn13,"\n")
# ###################### CONTRÔLE ############################
chn=""
for x in chn13 :
    c=0
    for y in alpha :
        if (x==y) :
            sy = alpha[(c+13) % 26]
            chn = chn + sy
            break
        else :
            if c==25 :
              chn = chn + x
              break
        c = c+1

#
print (chn)
```

les résultats (on applique deux fois l'algorithme)

```
>>>
abcdefghijklmnopqrstuvwxyz longueur :  26

yrf pbaf bfrag gbhg ; p'rfg à pryn dh'ba yrf erpbaanîg

les cons osent tout ; c'est à cela qu'on les reconnaît
>>>
```

4. Les n-uplets ou tuples.

4.1. Séquence.

On a pu constater les similarités entre les listes et les chaînes de caractères : similitude d'accès à un élément, ou une sous-structure, similitude de comportement dans la boucle **`for`**, ou une fonction en commun, **`len()`**. Les types qui possèdent ces propriétés se nomment des séquences. Les éléments d'une séquence peuvent être figés comme les chaînes de caractère, ou modifiables comme les listes.

Il existe un genre de séquence appelée n-uplet ou encore **`tuple`**, assez similaire aux listes, à la différence près que le n-uplet est non modifiable.

4.2. Créations de n-uplet et accès aux éléments constitutifs

Un n-uplet est une suite ordonnée d'objets ; les objets constitutifs sont séparés par des virgules. Les objets constitutifs peuvent être des n-uplets, des chaînes, des listes etc. En principe, les n-uplets littéraux (qui peuvent s'écrire) sont entourés de parenthèses, mais ce n'est pas une obligation s'il n'y a pas d'ambiguïté. Un n-uplet peut être vide (parenthèses obligatoires !).

Il peut être formé d'un élément (**singleton**) ; dans ce cas les parenthèses sont obligatoires, et même plus : **l'élément constitutif est suivi d'une virgule !**

```
>>> unTuple = 12345, 6789, "une chaîne",["du texte", 9876, True]
>>> print (unTuple)
(12345, 6789, 'une chaîne', ['du texte', 9876, True])
>>>
>>> print (unTuple [2], type(unTuple [2]))
une chaîne <class 'str'>
>>>
>>> sousTuple = unTuple [1:3]
>>> print (sousTuple)
(6789, 'une chaîne')
>>> sousTuple = unTuple [1:1]
>>> print (sousTuple)
()
>>> sousTuple = unTuple [3:4]
>>> print (sousTuple)
(['du texte', 9876, True],)
>>> unSingleton = (1357,)
>>> print (unSingleton [0])
1357
>>>
>>> s= (9753)
>>> print (s, type(s))
9753 <class 'int'>
>>>
```

* l'accès à un élément unique donne un élément du type de l'objet, comme pour les listes.

* attention aux singletons : il y a une virgule finale.

4.4. Emballage et déballage des n-uplets.

Un n-uplet est un container qui emballe une suite ordonnée de valeurs. On peut donc réaliser une affectation multiple avec un n-uplet ; on appelle cela **un déballage de tuple**. Il faut évidemment mettre le bon nombre de variables !

```
>>>
>>> tpl1 = 1, 2, 3
>>> tpl2 = "un", "deux", "trois"
>>> tpl = tpl1 + tpl2
>>> print (tpl)
(1, 2, 3, 'un', 'deux', 'trois')
>>> x, y, z, t, u, v = tpl  # déballage
>>> print("x=",x,"  y=", y,"  z=", z,"  t=", t, "  u=", u,"  v=", v,sep="")
x=1  y=2  z=3  t=un  u=deux  v=trois
>>>
```

4.5. cast entre liste et tuple.

```
>>> maListe = list ("abracadabra")
>>> maListe
['a', 'b', 'r', 'a', 'c', 'a', 'd', 'a', 'b', 'r', 'a']
>>> liste2tuple = tuple(maListe)
>>> liste2tuple
('a', 'b', 'r', 'a', 'c', 'a', 'd', 'a', 'b', 'r', 'a')
>>> tuple2liste = list ( (1, 2, 3) )
>>> tuple2liste
[1, 2, 3]
>>>
```

Les tuples ont la même représentation en mémoire que les listes La différence tient essentiellement à la modification : les listes sont souples, les tuples impossible à modifier une fois créés. Cependant, si un objet du tuple est modifiable (une liste, pour l'instant), il le reste. La fixité des tuple est en fin de compte assez relative, dès que l'on y insère des items modifiables.

fiche 7 : définition de fonctions

introduction.

Tous les langages utilisent des fonctions : une fonction est un bloc de code identifié (nom de la fonction). L'utilisation de l'identificateur dans un texte appelle l'exécution du bloc de code. On peut passer des paramètres (notion à préciser) au bloc de code. Les puristes distinguent deux types de "fonctions" : les procédures qui se réduisent à une exécution de code et sont en fait des instructions ; et les fonctions proprement dites dont l'exécution retourne une valeur au programme d'inclusion.

Cette valeur a pour vocation d'être utilisée à l'intérieur d'une instruction. La fonction **`input()`** de Python est une vraie fonction ; **`print()`** est une procédure. Mais l'usage a fait que la distinction a été tournée : le langage C par exemple, ou Java, PHP, Javascript, ne connaissent que les fonctions. Certaines ne retournent rien, et d'autres retournent un objet du langage, une valeur. Python ne distingue pas formellement fonction et procédures.

1. Définir une fonction.

1.1. mot clef, identificateur, paramétrage.

Le schéma de définition d'une fonction est :

```
def identificateur ( suite de paramètres formels ) :

     bloc de code (indenté!)
```

* **`def`** est un mot réservé de Python

* l'identificateur, les parenthèses, les deux-points sont obligatoires ; l'ensemble constitue une ligne Python. Si la ligne physique est trop petite on utilise l'antislash.

* Les paramètres formels sont facultatifs. S'il y en a plusieurs, ils sont séparés par des virgules.

* une fonction ne retourne rien si on ne spécifie pas de valeur après l'instruction **`return ; return`** arrête l'exécution du corps de fonction. Comme en C ou en Java, ce n'est pas tout à fait vrai, puisque toute fonction retourne une valeur qui est ici appelée **`None`** (**`void`** en C ou Java) et qui désigne un objet vide, avec lequel on ne peut rien faire !

1.2. Exemple de la suite de Fibonacci.

Une suite de Fibonacci est une suite de nombres où chacun est la somme des deux qui le précèdent. Le deux premiers nombre sont égaux à 1. La fonction à définir doit écrire les valeurs successives inférieures à une borne précisée à l'exécution.

```
#
# suite de Fibonacci / première version
#
def fibonacci (borne) :
    print ("la borne imposé est : ",borne, "\n")
    i, j = 0, 1
    while (j < borne) :
        print (j,end="  ")
        i, j = j, i+j
# fin de la définition de la fonction
#
# test
fibonacci (3000)

# fin de programme
```

* **i** et **j** identifient les deux nombres sur lesquels travailler ; le couple **i, j** et remplacé par le couple **j, i+j** à chaque exécution de la boucle **while**. Seul le nombre **j** est affiché tant que la condition d'arrêt de l'itération n'est pas satisfaite.

Le résultat :

```
>>>
la borne imposé est :  3000

1  1  2  3  5  8  13  21  34  55  89  144  233  377  610  987  1597  2584
>>>
```

1.3. Seconde version.

On impose cette fois que la fonction constitue une liste qu'elle retourne.

```
#
# suite de Fibonacci / version retour de liste
#
def fibonacci (borne) :
    print ("la borne imposée est : ",borne, "\n")
    liste =[]
    i, j = 0, 1
    while (j < borne) :
        liste = liste + [j]
        i, j = j, i+j
    else :
        return liste
# fin de la définition de la fonction
#
# test
print (fibonacci (3000))

# fin de programme
```

* la différence essentielle est le **return liste**. L'affichage est réalisé en utilisant le résultat retourné.

```
>>>
la borne imposée est :  3000

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584]
>>>
```

1.4. Chaîne de documentation.

La première ligne Python du bloc de définition d'une fonction peut être une chaîne de caractères. Cette chaîne sert à la documentation de la fonction ; il existe des outils de documentation et des librairies Python qui utilisent ces chaînes que l'on appelle aussi des **docstrings.**

1.5. Questions de variables.

Les questions suivantes sont récurrentes dans tous les langages :

- que désigne le paramètre formel ?
- quelle est la portée d'une variable définie dans une fonction ?
- les variables définies en dehors de la fonction sont-elles utilisables dans le corps de fonction ?

* au premier abord, le paramètre formel désigne une valeur ! **il ne peut pas désigner**, comme en Pascal, **une variable externe** dont la valeur pourrait être manipulée depuis la fonction. Pour être plus précis, on va montrer dans la suite le mécanisme qui opère dans Python.

* la deuxième question a une réponse univoque : tout comme les paramètres formels, les variables définies dan le corps de la fonction (par une affectation) n'ont d'existence que dans le bloc de définition de la fonction.

* la réponse à la troisième question est plus délicate :

- une variable globale est lisible depuis le corps de la fonction, sauf si l'identificateur est identique à celui d'un paramètre ou d'une variable locale à la fonction.

- il n'y a pas d'effet de bord par défaut. C'est-à-dire qu'on ne peut modifier une variable globale, **sauf si on le déclare explicitement par l'instruction `global`**. Les effets de bord ne peuvent être accidentels ; il faut exprimer l'intention de les provoquer. De toutes façon, c'est une manière de faire qui est à éviter, sauf dans des cas bien particuliers, comme par exemple lorsqu'on définit une fonction d'initialisation de variables globales.

1.6. Les paramètres formels ne sont pas typés.

Contrairement à beaucoup de langages de haut niveau (C, Pascal, Ada, Java) il n'y a pas de type attaché aux paramètres formels. Ainsi, le même paramètre peut être utilisé avec des valeurs de type différent ; cela peut être utile … et dangereux.

```
#
# les variables ne sont pas typées
#
def sansType (param) :
        print ("-"*10,param,"-"*10,"\n")

sansType (1000)
sansType ("1000")
sansType ("mille")
sansType ([1, 3, 5, 7 ])
```

résultat :

```
>>>
---------- 1000 ----------

---------- 1000 ----------

---------- mille ----------

---------- [1, 3, 5, 7] ----------

>>>
```

2. paramètres à valeur par défaut ou paramètres clefs/valeur.

2.1. valeur par défaut si le paramètre n'est pas renseigné

On a déjà rencontré avec la fonction **`print()`** la notion de valeur par défaut, avec les paramètres **`sep`** et **`end`** : un paramètre formel peut avoir une valeur prédéfinie qui sera utilisée si, lors de l'appel de la fonction, **le paramètre formel n'est pas renseigné**. Pour la fonction **`print()`**, **`sep`** est la valeur du séparateur des valeurs affichée, par défaut un espace, et **`end`** la valeur de fin d'affichage, par défaut le passage à la ligne.

```
>>> print (1,2,3,4,5)
1 2 3 4 5
>>> print (1,2,3,4,5, sep=" --- ",end="  fin\n")
1 --- 2 --- 3 --- 4 --- 5  fin
>>>
```

Les paramètres à valeur par défaut sont aussi appelés paramètres à clef, paramètres formels clef/valeur…

2.2. règles.

Voici les règles d'usage dans l'utilisation des paramètres formels à valeur par défaut :

- la valeur par défaut est définie une seule fois, lors de la création de la fonction.
- la valeur par défaut ne doit pas être modifiée au cours de l'exécution. On reprendra ce point très problématique ultérieurement.
- les paramètres formels à valeur par défaut sont définis en queue de la liste des paramètres formels.
- on peut par contre affecter une valeur par défaut différente lors de l'appel de la fonction. Ceci ne modifie pas la valeur par défaut pour des appels de fonction ultérieurs.
- on n'est pas obligé de renseigner un tel paramètre ; il peut ainsi y avoir moins de valeurs de paramètres dans un appel de fonction que de paramètres déclarés.

3. Particularité des appels de fonctions Python.

Les fonctions sous Python ont deux particularités moins importantes qui sont disponibles à ce stade de l'étude.

3.1. Arguments sous la forme clef/valeur.

Un paramètre formel est normalement remplacé par une valeur lors de l'appel. Mais on peut aussi utiliser un système clef/valeur sur le modèle suivant :

```
>>> def motClef (oiseau, couleur) :
        print ("le ",oiseau," est un oiseau de couleur ",couleur, sep="")

>>> motClef ("merle", "noire")
le merle est un oiseau de couleur noire
>>> motClef ("colombe", couleur = "blanche")
le colombe est un oiseau de couleur blanche
>>> motClef ( couleur = "jaune", oiseau="canari")
le canari est un oiseau de couleur jaune
>>> motClef (oiseau = "tupi", "rouge")
SyntaxError: non-keyword arg after keyword arg (<pyshell#12>, line 1)
>>>
```

Dans le cas de plusieurs arguments mis sous forme clef/valeur, l'ordre de ces arguments n'a plus d'importance, à condition toutefois que les variables à passage d'argument usuel soient en premier et dans le bon ordre !

3.2. Nombre indéfini d'arguments.

On peut avoir besoin d'une fonction où **le nombre des arguments n'est pas défini par avance**. La fonction **`print()`** en est un bon exemple. Dans ce cas, on met une astérisque devant le paramètre formel qui a un nombre d'arguments indéterminé ; on récupère les valeurs sous forme d'un n-uplet (tuple) dans le corps de la fonction. On traite ce n-uplet en le déballant ou à l'aide d'une fonction **`for`**. On rappelle que la fonction **`len()`** s'applique aux n-uplets. Si la liste d'argument est vide, on n'écrit rien à l'appel !

Plus sophistiqué encore, on peut avoir une suite indéfinie de clefs/valeur ; on précède le nom du paramètre de deux astériques : **`**args`**. Si on a par exemple

```
def maFonction (**args):
```

l'appel se fait par : **`maFonction( arg1="toto", arg2=45, arg3 = True)`** et le passage de paramètre se fait par le dictionnaire **`args`** avec **`arg = { "arg1" : "toto", "arg2 : 45, "arg3 : True }`**. On peut ainsi tester la présence de **`argx`** comme clef et en récupérer la valeur par **`args[argx]`**.

programme

```
#
# arguments multiples
#
def vosEnfants ( message, *prenom ) :
    nombre = len (prenom)
    date =()
    if nombre :
        for x in prenom :
            naissance = input (x + " est né(e) le : ")
            date = date + (naissance,)

        # affichage #
        print(message)
        for n in range(nombre) :
            print ("---", prenom [n], "est né(e) le",date [n], sep=" ")

    else :
        print (message + " il n'y a pas de fratrie")

# fin de définition de la fonction

# cas usuel
vosEnfants ("\npremier cas\nvoici les référence des enfants", \
            "Jean",\
            "Claude",\
            "Marie")

# cas de l'argument absent
vosEnfants ("\n\ndeuxième cas\nAucune référence trouvée : ",)
```

* la liste des prénoms est transmise par un tuple ; en cas d'absence de l'argument, on recueille un tuple vide, de longueur nulle. Dans ce cas, le cast de la longueur se fait en False.

* on itère sur le tuple de **`prenom`** par une boucle **`for`**. On appelle cela "déplier" le tuple.

* on peut avoir des paramètres à une seule valeur en même temps que des paramètres formels indéterminés.

résultat des deux appels

```
>>>
Jean est né(e) le : 04/10/1937
Claude est né(e) le : 29/11/1939
Marie est né(e) le : 16/04/1932

premier cas
voici les référence des enfants
--- Jean est né(e) le 04/10/1937
--- Claude est né(e) le 29/11/1939
--- Marie est né(e) le 16/04/1932


deuxième cas
Aucune référence trouvée :  il n'y a pas de fratrie
>>>
```

4 La mémoire.

On va trouver ici une particularité de Python assez déroutante : la gestion des paramètres en mémoire (variables locales et paramètres). Elle est très différente de ce qui se fait dans d'autres langages et partant un peu délicate à comprendre. Quelques règles simples permettent une utilisation non ambiguë des variables dans une fonction.

4.1. variable locale obtenue par un paramètre.

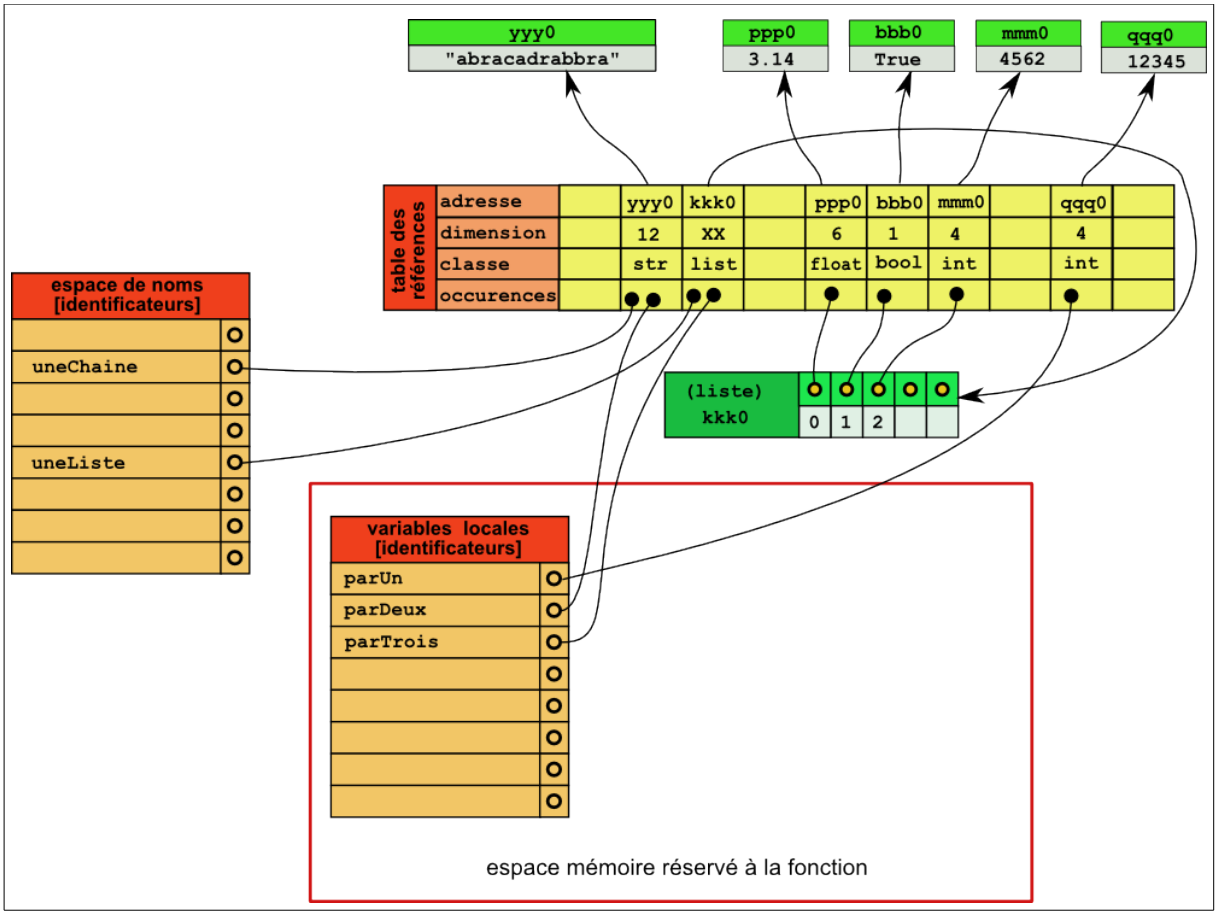
On examine le cas d'une fonction définie par :

```
def maFonction (parUn, parDeux, parTrois) :
```

et appelée par une instruction comportant : `maFonction (12345, uneChaine, uneListe)`

où `uneChaine` est une variable de type `str` et `uneListe`, une variable de type `list`.

Règle : la fonction crée à l'appel, un espace de noms pour les identificateurs locaux ; les paramètres formels sont des identificateurs locaux. Cet espace de nom disparaît lorsque la fonction se termine.



L'initialisation des variables locales issues d'un paramètre formel se fait de façon différente selon que le paramètre réel est une expression ou un identificateur. La valeur est liée comme pour une affectation. Ainsi, le paramètre réel 12345 est une expression (un littéral) ; la valeur d'évaluation de cette expression est stockée en mémoire, une nouvelle référence est créée et le lien est posé. Par contre le second paramètre réel est l'identificateur `unChaine` ; c'est une forme d'alias qui est créé. Même chose pour la liste.

Attention : la liste étant un type de variable modifiable, si on modifie la variable `parTrois`, on modifie dans la foulée `uneListe`. Et cette modification subsiste lorsque la fonction se termine.

4.2. ajout où substitution de variable locale.

Règle : il n'y a qu'un seul espace de noms pour les paramètres formels et le variables crées par affectation.

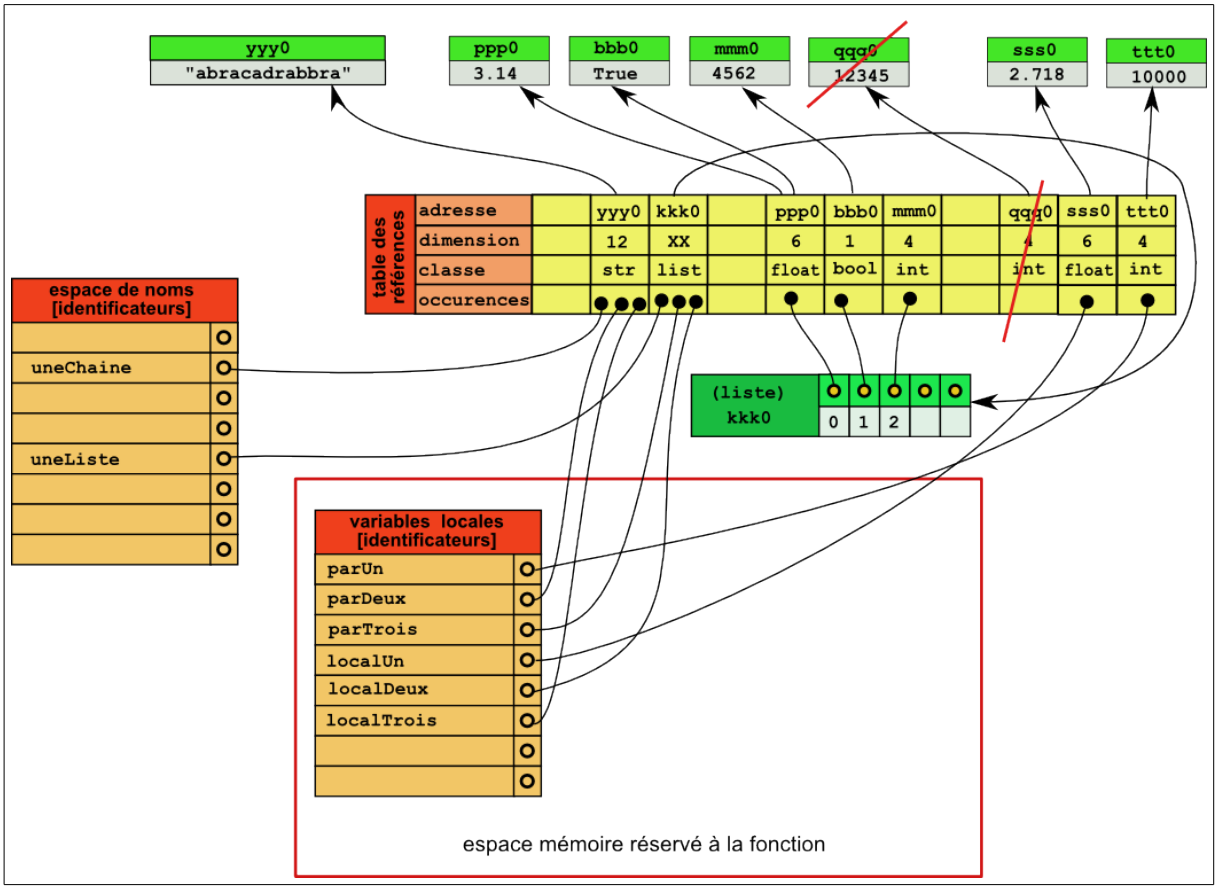
L'affectation au sein de la fonction fonctionne de la façon habituelle. On rappelle les quatre cas :

- identifcateur existant ; valeur expression : nouvelle référence, changement de lien.

- identificateur existant ; valeur variable : alias.
- identificateur nouveau ; valeur expression : nouvelle référence, nouvel identificateur.
- identificateur nouveau ; valeur variable : alias.

En guise d'exemples on va supposer l'existence des affectations suivantes dans la fonction :

```
localUn = 2.718
localDeux = parTrois
localTrois = parDeux
parUn = 10000
```



4.3. paramètre clef/valeur.

Règles : La table des paramètres clef/valeur est créée une fois pour toutes. Cette table est lue après celle des variables locales.

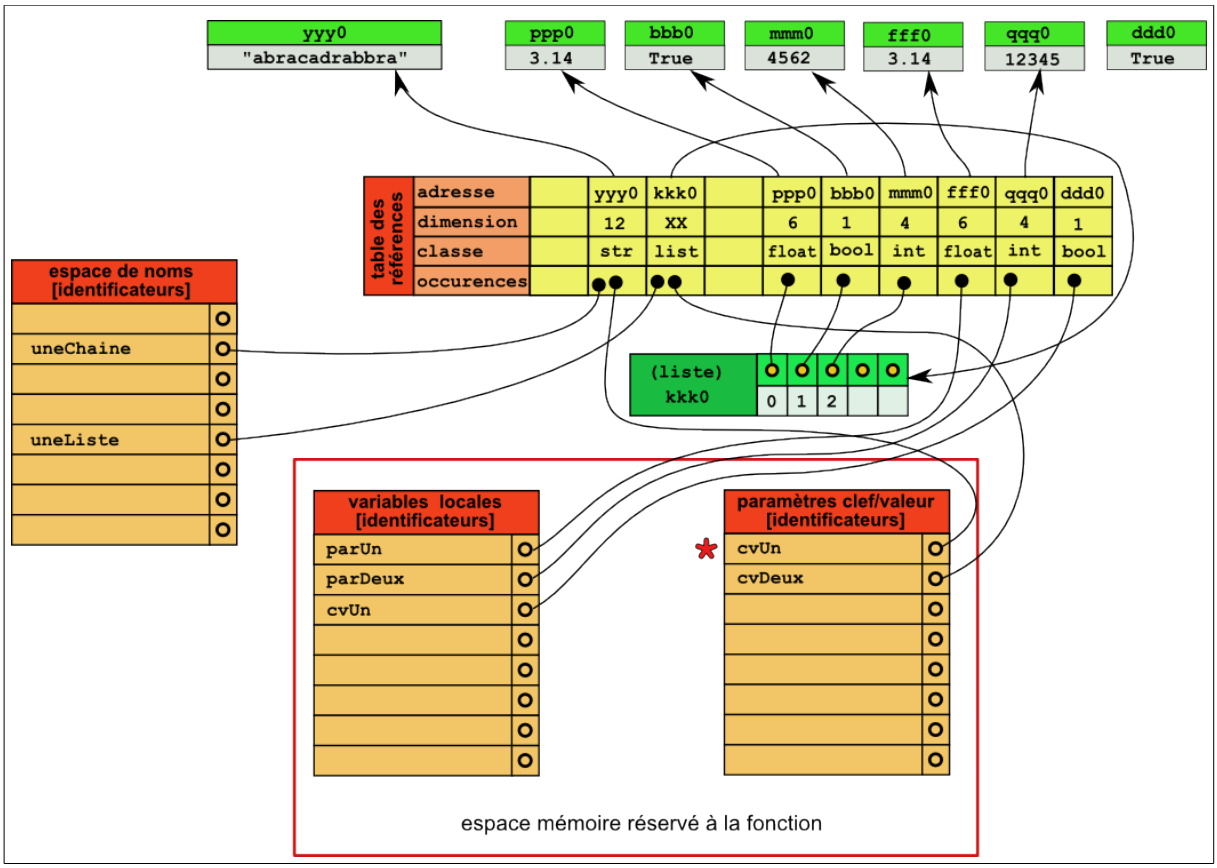
En cas d'affectation sur un identificateur présent dans la table des paramètre clef/valeur, que ce soit dans l'appel, que ce soit dans le corps de fonction, une nouvelle variable locale est créée et le paramètre clef/valeur est occulté.

On examine le cas d'une fonction définie par :

```
def maFonction (parUn, parDeux,cvUn = uneChaine, cvDeux = uneListe) :
```

et appelée par une instruction comportant : `maFonction (3.14, 12345, cvUn = True)`

On la configuration qui suit ; l'astérisque signale une variable que l'on ne peut atteindre :



variables modifiables.

la variable **`cvDeux`**, tout comme **`uneListe,`** sont modifiables ; que se passe-t-il si on effectue une telle modification ? Il suffit de suivre le schéma pour voir que cette modification est définitive. Il y a là un piège provoquant une erreur qui peut s'avérer difficile à détecter. Mais aussi une opportunité dans certaines circonstances : comptage des appels de la fonction, initialisation par une fonction ad hoc...

En tout état de cause, sauf on est dans ces cas spéciaux, il vaut mieux considérer les paramètres clef/valeur comme en lecture seulement, ou alors créer une copie du paramètre dans l'espace des variables locales. Par exemple pour une liste, on aurait localUn = cvDeux + [] . Bien mettre une expression comme membre de droite, pas un identificateur qui donnerait un alias !

fiche 8 : dictionnaires

introduction.

Les entiers, les flottants, les chaînes, les listes, les tuples admettent une représentation littérale. On va voir un nouveau type d'objets ayant une représentation littérale : le dictionnaire. En PHP, ont connaît les tableaux associatifs (par opposition aux tableau indexés de C, Pascal, Java…). Le dictionnaire est semblable au tableau associatif : c'est un tableau de valeurs, mais au lieu d'un numéro, chaque valeur est repérée par une clef, souvent une chaîne de caractères, mais ce peut être d'autres objets.

Le tableau associatif est extensible (on peut lui ajouter des couples clef/valeur), modifiable (on peut changer la valeur associée à une clef) et peut être parcouru (mais la notion d'ordre des composants n'a pas de sens pour un dictionnaire).

1. La notion de clef/valeur en Python.

1.1. La clef.

Une clef est un objet Python non modifiable (ce concept sera formalisé ultérieurement). En pratique, nombre, chaîne de caractères, et tuple formé d'objets non modifiables. Une liste ne peut être une clef ; ni un tuple dont l'un des éléments est une liste. Techniquement, il faut que la clefs soit un objet "hachable", en pratique, non modifiable, ni directement, ni indirectement.

1.2. Le couple clef/valeur.

La valeur peut être n'importe quel objet, et si l'on peut avoir une représentation littérale, le couple clef/valeur s'écrit : clef / deux-points / valeur.Exemple : "Jean":1937

1.3. Écriture littérale d'un dictionnaire.

Les objets "valeur" ne sont pas nécessairement des littéraux ; si elle est posible, l'écriture littérale ressemble à un tuple dont les éléments sont des couples clef/valeur, et les caractères englobants, des accolades au lieu de parenthèses.

1.4. Accès aux éléments.

```
>>>
>>> monDict = { "jean" : 1937, "marie" : 1932, "claude" : 1939 }
>>> print (monDict)
{'claude': 1939, 'jean': 1937, 'marie': 1932}
>>>
>>> for x in monDict :
        print (x)


claude
jean
marie
>>> for x in monDict :
        print (x,"--->",monDict [x])


claude ---> 1939
jean ---> 1937
marie ---> 1932
>>> print (monDict['jean'])
1937
>>> print(monDict['toto'])
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    print(monDict['toto'])
KeyError: 'toto'
>>>
```

1.5. ajout d'une clef/valeur.

Il suffit de créer une nouvelle clef/valeur : affecter une valeur à une clef non existante ; l'affectation à une clef valide modifie la valeur associée.

```
>>>
>>> leDico = {}
>>> leDico ["computer"] = "ordinateur"
>>> leDico ["mouse"] = "souris"
>>> leDico ["RAM"] = "Mémoire Vive"
>>> leDico ["ROM"] = "Mémoire De Masse"
>>> print (leDico)
{'RAM': 'Mémoire Vive', 'computer': 'ordinateur', 'mouse': 'souris', 'ROM': 'Mémoire De Masse'}
>>> leDico ["RAM"] = "Mémoire interne"
>>>
>>> print (leDico)
{'RAM': 'Mémoire interne', 'computer': 'ordinateur', 'mouse': 'souris', 'ROM': 'Mémoire De Masse'}
>>>
```

2. Autres modes de création.

2.1. A partir d'une liste de paires (2-uplets) : la fonction dict().

```
>>> listeDict = [("jean", 1937), ("claude", 1939), ("marie anne", 1946)]
>>> monDict = dict (listeDict)
>>> for x in monDict :
        print (x, "===>>",monDict[x])


claude ===>> 1939
jean ===>> 1937
marie anne ===>> 1946
>>>
>>> print (monDict)
{'claude': 1939, 'jean': 1937, 'marie anne': 1946}
>>>
```

2.2. La fonction dict() avec des arguments à mots-clefs.

Note importante : ceci ne fonctionne que si les clefs du dictionnaire peuvent être des identificateurs :

```
>>>
>>> monDico = dict (jean=71.60, claude=69.40, marie_anne=40.)
>>> print (monDico)
{'claude': 69.400000000000006, 'marie_anne': 40.0, 'jean': 71.599999999999994}
>>>
```

Notes.

* le dictionnaire n'est pas séquencé (on ne peut pas parler d'ordre de ses éléments, contrairement à ce qui se fait en PHP où l'ordre est une donnée fondamentale).

La clause **`for`** permet cependant de le balayer ; sa variable de contrôle prend successivement la valeur des **clefs** du dictionnaire, dans un ordre imprévisible.

* l'opérateur + n'est pas applicable aux dictionnaire.

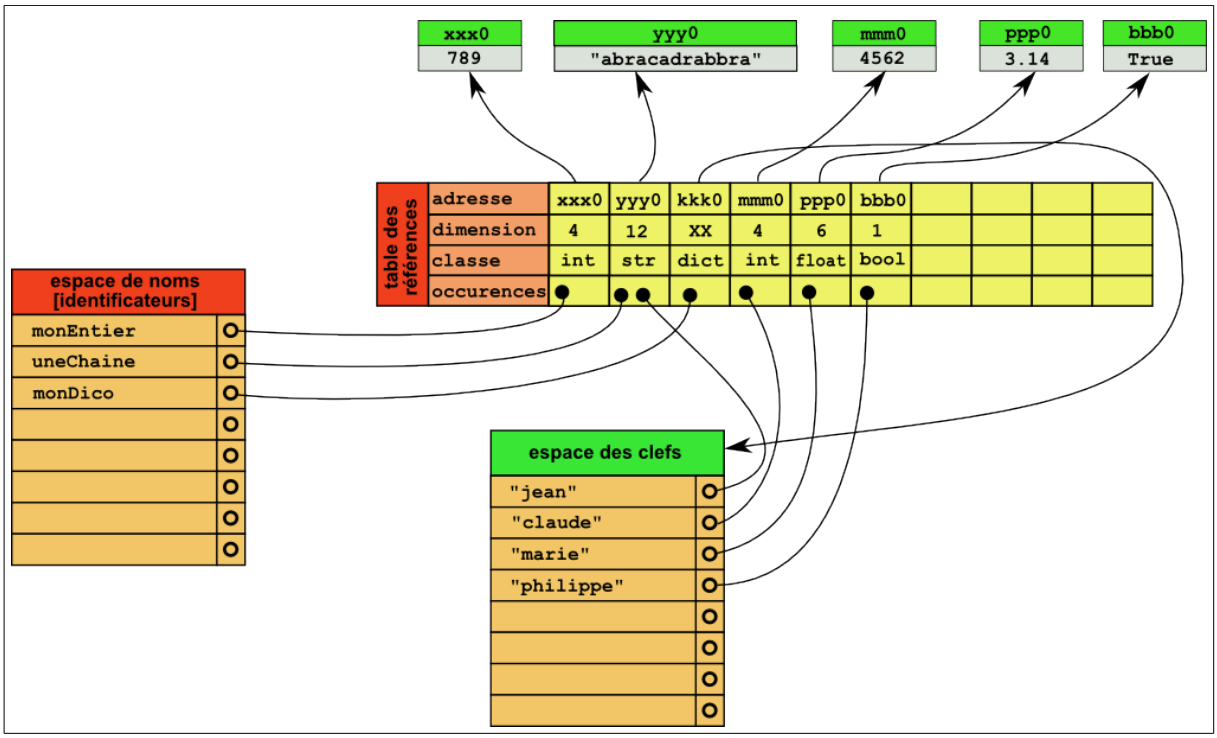
* pour supprimer un élément du dictionnaire, il suffit d'effacer la clef par la fonction **`del()`** :

`del(monDico["claude"])` supprime la référence de clef **`"claude"`** du dictionnaire **`monDico`**.

La présente fiche est très incomplète concernant les propriétés des dictionnaires ; on verra après l'étude circonstanciée de la notion d'objet d'autres propriétés liées aux dictionnaire.

3. Représentation mémoire.

Un dictionnaire est une "table de hachage" ; une telle table est organisée de façon à rendre très rapide l'accès à un couple clef/valeur à partir de la clef. Ce point est important en pratique : Python étant en interne organisé avec des dictionnaires qui peuvent avoir une taille fort importante, c'est une question d'efficacité. Mais pour le programmeur, le détail de l'implémentation est sans intérêt.. On se contentera donc d'une représentation simplifiée, mais qui couvre les cas qu'il peut rencontrer.



4. un exercice d'application.

4.1. codage et brouillage.

*** On propose un système de cryptage selon la grille de codage suivante :**

	1	2	3	4	5	6	7
1	h	y	p	z	n	d	ù
2	û	e	à	i	è	t	w
3	)	v	a	x	é	espace	j
4	l	/	!	ô	c	?	ï
5	'	k	"	s	î	â	o
6	q	b	ë	ç	u	f	;
7	g	(	r	-	.	,	m

Chaque caractère du tableau est codé par deux caractères majuscules : "y" est codé "12", "x" est code "34". Pour une chaîne données (sans majuscules ni chiffres) de longeur n, on a ainsi un codage sur 2n chiffres.

*** Pour un meilleur cryptage, on utilise le système un brouillage à mot secret.**

Un mot secret ne doit pas avoir de lettres en doubles ; seules les 26 lettres de l'alphabet sont admises. Il a un nombre pair de lettres. Supposons que le mot secret soit **`TURING`** ; il a 6 **caractères** qui mis dans l'ordre alphabétique donnent **`GINRTU.`** On construit un tableau à 6 colonnes de chapeau "TURING" et on distribue les chiffres du codage de gauche à droite et du haut vers le bas. On complète le tableau de façon à avoir un nombre pair de chiffres à l'aide du code 36 (code de l'espace).

* On brouille le tableau en mettant les colonnes dans l'ordre "alphabétique". Le cryptage définitif s'obtint en lisant le tableau de hau en bas, et de gauche à droite.

Exemple :

Soit la chaîne "**`papa part en voyage !`**" et le mot secret **`TURING`**

première étape : codage 133313333622542636221536325712337122371223643

les lignes : 133313 333622 542636 221536 325712 337122 3643

deuxième étape : mise en tableau / complétion du tableau

T	U	R	I	N	G
1	3	3	3	1	3
3	3	3	6	2	2
5	4	2	6	3	6
2	2	1	5	3	6
3	2	5	7	1	2
3	3	7	1	2	2
3	6	4	3	**3**	**6**

troisième étape : brouillage

G	I	N	R	T	U
3	3	1	3	1	3
2	6	2	3	3	3
6	6	3	2	5	4
6	5	3	1	2	2
2	7	1	5	3	2
2	1	2	7	3	3
6	3	**3**	4	3	6

quatrième étape : lecture

colonnes : 3266226 3665713 1233123 3332574 135333 334236

cryptage : 326622636657131233123 3332574135333334236

4.2. Le code Python d'encodage

*** encodage et décodage : une affaire de dictionnaires**

```
encode={ "h":"11","y":"12","p":"13","z":"14","n":"15","d":"16","ù":"17",
         "û":"21","e":"22","à":"23","i": "24","è":"25","t":"26","w":"27",
         ")":"31","v":"32","a":"33","x":"34","é":"35"," ":"36","j":"37",
         "l":"41","/":"42","!":"43","ô":"44","c":"45","?":"46","ï":"47",
         "'":"51","k":"52","\"":"53","s":"54","î":"55","â":"56","o":"57",
         "q":"61","b":"62","ë":"63","ç":"64","u":"65","f":"66",";":"67",
         "g":"71","(":"72","r":"73","-":"74",".":"75",",":"76","m":"77"
       }

decode = {}
for c in encode :
    v = encode [c]
    decode[v] = c
```

- on peut remarquer que toute chaîne peut être une clef !

*** on a besoin de l'ordre alphabétique des lettres du mot secret**

```
# recherche de l'ordre alphabétique
def ordreNaturel (pSecret) :
    lOrdre=[]
    for c in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" :
        x=-1
        for l in pSecret :
            x=x+1
            if c==l :
                lOrdre = lOrdre + [x]
                break
    return lOrdre
```

- la méthode est élémentaire ; il y a des façons plus naturelles de faire la même chose, mais qui sont encore non abordables à ce niveau. L'ordre est fourni par une liste de nombre.

*** première étape :**

```
# encodage
def etape1(pChn, pLgSecret) :
    lCodage = ""
    for car in pChn :
       lCodage = lCodage + encode[car]
    lCodage = lCodage + "36"*((pLgSecret-1) // 2) # espaces de commodité
    return lCodage
```

*** deuxième étape :**

```
# tableau de chaînes
def etape2(pCodage, pLgSecret, pNbLignes) :
    lTab = []
    for x in range(pLgSecret) :
```

```
        lTab= lTab + [""]
    for i in range(pLgSecret) :
        for j in range(pNbLignes) :
            lTab [i] = lTab [i] + pCodage[i + j*pLgSecret]
    return lTab
```

- le tableau est modélisé par une liste de chaînes ; c'est une des solutions possibles.

*** troisième étape :**

```
# brouillage
def etape3(pTableau, pSecret) :
    lOrdre= ordreNaturel (pSecret)
    lBrouillage = []
    for i in lOrdre :
        lBrouillage = lBrouillage + [pTableau[i]]
    return lBrouillage
```

*** quatrième étape :**

```
# extraction du cryptage
def etape4(pTableau, pLgSecret, pNbLignes) :
    lCryptage =""
    for i in range(pLgSecret) :
        for j in range(pNbLignes) :
            lCryptage= lCryptage + pTableau[i][j]
        lCryptage = lCryptage+"  "
    return lCryptage
```

*** l'algorithme principal :**

```
# routine principale
def encodage (pChn, pSecret) :
    lLgSecret = len (pSecret)
    lCodage = etape1 (pChn, lLgSecret)
    lNbLignes = len(lCodage) // lLgSecret
    lTableau = etape2(lCodage,lLgSecret, lNbLignes)
    lTableau = etape3(lTableau, pSecret)
    lCryptage = etape4 (lTableau, lLgSecret, lNbLignes)
    return lCryptage
```

* ********** **main** **********

```
# ******************** main *****************
chn = "papa est en voyage !"
secret = "TURING"
cryptage = encodage (chn, secret)
print (cryptage)
```

4.3. Le code complet :

```
# exercice de cryptage

encode={  "h":"11","y":"12","p":"13","z":"14","n":"15","d":"16","ù":"17",
          "û":"21","e":"22","à":"23","i": "24","è":"25","t":"26","w":"27",
          ")":"31","v":"32","a":"33","x":"34","é":"35"," ":"36","j":"37",
          "l":"41","/":"42","!":"43","ô":"44","c":"45","?":"46","ï":"47",
          "'":"51","k":"52","\"":"53","s":"54","î":"55","â":"56","o":"57",
          "q":"61","b":"62","ë":"63","ç":"64","u":"65","f":"66",";":"67",
          "g":"71","(":"72","r":"73","-":"74",".":"75",",":"76","m":"77"
        }

decode = {}
for c in encode :
    v = encode [c]
    decode[v] = c

# recherche de l'ordre alphabétique
def ordreNaturel (pSecret) :
    lOrdre=[]
    for c in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" :
        x=-1
        for l in pSecret :
            x=x+1
            if c==l :
                lOrdre = lOrdre + [x]
                break
    return lOrdre

# encodage
def etape1(pChn, pLgSecret) :
    lCodage = ""
    for car in pChn :
       lCodage = lCodage + encode[car]
    lCodage = lCodage + "36"*((pLgSecret-1) // 2) # espaces de commodité
    return lCodage

# tableau de chaînes
def etape2(pCodage, pLgSecret, pNbLignes) :
    lTab = []
    for x in range(pLgSecret) :
        lTab= lTab + [""]
    for i in range(pLgSecret) :
        for j in range(pNbLignes) :
            lTab [i] = lTab [i] + pCodage[i + j*pLgSecret]
    return lTab

# brouillage
def etape3(pTableau, pSecret) :
```

```
    lOrdre= ordreNaturel (pSecret)
    lBrouillage = []
    for i in lOrdre :
        lBrouillage = lBrouillage + [pTableau[i]]
    return lBrouillage

# extraction du cryptage
def etape4(pTableau, pLgSecret, pNbLignes) :
    lCryptage =""
    for i in range(pLgSecret) :
        for j in range(pNbLignes) :
            lCryptage= lCryptage + pTableau[i][j]
        lCryptage = lCryptage+""
    return lCryptage

# routine principale
def encodage (pChn, pSecret) :
    lLgSecret = len (pSecret)
    lCodage = etape1 (pChn, lLgSecret)
    lNbLignes = len(lCodage) // lLgSecret
    lTableau = etape2(lCodage,lLgSecret, lNbLignes)
    lTableau = etape3(lTableau, pSecret)
    lCryptage = etape4 (lTableau, lLgSecret, lNbLignes)
    return lCryptage

# décryptage : remise en tableau
def etape_1 (pChn, pLgSecret, pNbLignes):
    lTableau = []
    for i in range(pLgSecret) :
        lTableau = lTableau + [""]
    for i in range(pLgSecret):
        lTableau [i] = pChn [i*pNbLignes:(i+1)*pNbLignes]
    return lTableau

# décryptage : remise en ordre du tableau
def etape_2(pTableau, pSecret, pLgSecret) :
    lOrdre= ordreNaturel (pSecret)
    lTab = []
    for i in lOrdre :
        lTab = lTab+ [""]
    for i in range(pLgSecret) :
        c = lOrdre [i]
        lTab [c]= pTableau [i]
    return lTab

# remise en chaine
def etape_3 (pTableau, pLgSecret, pNbLignes) :
    lChn = ""
```

```
    for i in range(pNbLignes):
        for j in range(pLgSecret):
            lChn = lChn + pTableau[j][i]
    return lChn

# décryptage de la chaine
def etape_4 (pCodage):
    lChn = ""
    lLgCodage = len (pCodage)
    for r in range(0,lLgCodage, 2) :
        lClef = pCodage[r:r+2]
        lChn = lChn + decode[lClef]
    return lChn

def decodage (pChn, pSecret) :
    lLgSecret = len (pSecret)
    lNbLignes = len(pChn) // lLgSecret # cela doit tomber juste
    lTableau = etape_1 (pChn, lLgSecret, lNbLignes)
    lTableau = etape_2 (lTableau, pSecret, lLgSecret)
    lCodage = etape_3 (lTableau, lLgSecret, lNbLignes)
    lChn = etape_4 (lCodage)
    return lChn

# ******************** main *****************
chn = "papa est en voyage !"
secret = "TURING"
cryptage = encodage (chn, secret)
print (cryptage)
decryptage = decodage (cryptage, secret)
print (decryptage)
```

fiche 9 : ensembles

introduction

Une liste possède les propriétés suivantes : ses éléments sont indexé, elle est modifiable, un ou plusieurs composants peuvent apparaître plusieurs fois, on peut la balayer, dans l'ordre des éléments par une boucle for… Un ensemble ressemble à une liste à trois propriétés près : il n'est pas ordonné (éléments non indexés), un élément ne peut y apparaître qu'une fois, les éléments sont non modifiables. Comme éléments d'un ensemble, on ne peut mettre que des nombres, des chaînes, des n-uplets. On ne peut avoir un accès direct et en lecture seule aux éléments constitutifs que par un balayage. Un ensemble est modifiable : on peut lui ajouter ou retirer des éléments. Comme une liste, il peut être vide.

1. Définition d'un ensemble.

1.1. écriture littérale.

```
>>>
>>> ens1 = { 1, 2, 3, 4, 'toto', 'bibi', (1, 2)}
>>> print (ens1, "///", type (ens1))
{(1, 2), 'bibi', 2, 3, 4, 1, 'toto'} /// <class 'set'>
>>>
>>> ens2 = {2, 2, 2, "toto" }
>>> print (ens2, "///", type (ens2))
{2, 'toto'} /// <class 'set'>
>>>
```

1.2. définition à partir d'une liste.

```
>>>
>>> lst1 = [ 1, 2, 3, 4, 'une chaîne', ("chn1", "chn2") ]
>>> ens3 = set(lst1)
>>> print (ens3 , "///", type (ens2))
{1, 2, 3, 4, 'une chaîne', ('chn1', 'chn2')} /// <class 'set'>
>>>
```

1.3. définition à partir d'une chaîne de caractères.

```
>>>
>>> chn = "abracadabra"
>>> ens5 = set (chn)
>>> print (ens5)
{'a', 'r', 'b', 'c', 'd'}
>>>
```

1.4. définition à partir d'un n-uplet.

Il faut évidemment que le n-upleet (tuple) soit formé d'éléments non modifiables (pas de dictionnaire, de liste, d'ensemble).

```
>>>
>>> nup = ( 1, 1, 2, 2, 3, 3 )
>>> ens6 = set (nup)
>>> print (ens6)
{1, 2, 3}
>>>
```

1.5. ensemble vide.

L'utilisation conjointe des accolades pour les ensembles et les dictionnaires ne pose pas problème tans qu'il y a au moins un élément. L'ensemble vide se définit obligatoirement par un **`set()`**.

```
>>>
>>> quoi = {}
>>> print (type(quoi))
<class 'dict'>
>>>
>>> vide = set()
>>> print (type (vide))
<class 'set'>
>>>
```

```
>>>
>>> ens1
{'a', 'r', 'b', 'c', 'd'}
>>> ens2
{'p', 'r', 'z', 't'}
>>> print ( ens1 & ens2 )
{'r'}
>>> ens3
{'p', 'z', 't'}
>>> print ( ens1 & ens3)
set()
>>> print (ens1 & ens3 == {})   # l'erreur de type !
False
>>>
```

2. Appartenance et inclusion.

2.1. Appartenance d'un objet à un ensemble.

C'est l'opérateur **`in`** (ou **`not in`**) qui est utilisé.

```
>>>
>>> ensb = set ("abracadabra")
>>> print ("'a' est-il dans ensb ?",'a' in ensb)
'a' est-il dans ensb ? True
>>> print ("'z' est-il dans ensb ?",'z' in ensb)
'z' est-il dans ensb ? False
>>>
>>> print ("'a' n'est pas dans ensb ?",'a' not in ensb)
'a' n'est pas dans ensb ? False
>>>
```

2.2. Inclusion d'un ensemble dans un autre.

```
>>> ens1
{'a', 'r', 'b', 'c', 'd'}
>>> ens2
{'a', 'r', 'b', 'd'}
>>> ens3 = ens1
{'a', 'r', 'b', 'c', 'd'}
>>>
>>> print (ens1.isdisjoint (ens3))
False
>>> print("ens2 inclus dans ens1  ?",ens1 < ens2)
ens2 inclus dans ens1  ? False
>>> print("ens2 inclus dans ens1  ?",ens1 > ens2)
ens2 inclus dans ens1  ? True
>>>
>>> print ("le même ensemble ?", ens1==ens2)
le même ensemble ? False
>>> print ("le même ensemble ?", ens1==ens3)
le même ensemble ? True
>>>
```

Note :

Un certain nombre de propriétés classiques : test de disjonction, ajout d'un éléments, suppression d'un élément seront vus ultérieurement. Il font appel à des concepts non encore rencontrés. On peut simuler ces propriétés avec les opérateurs déjà vus : ensemble vide, réunion, différence etc. Mais il faut prendre garde que réunir un ensemble et un singleton crée une nouvel ensemble, alors que l'adjonction d'un élément se fait dans le même ensemble ; on a la même situation qu'avec les listes.

2. Les opérateurs sur les ensembles.

En mathématiques, les ensembles sont des éléments avec lesquels on peut opérer. Les opérations de base sont :

* **la réunion** de deux ensembles, ensemble formé des éléments qui appartiennent à l'un au moins des deux ensembles.

* **la différence** de deux ensembles, ensemble formé d'éléments qui appartiennent au premier ensemble, mais pas au deuxième.

* **l'intersection** de deux ensembles, ensemble formé d'éléments qui appartiennent à la fois aux deux ensembles.

* **la différence symétrique** de deux ensembles, ensemble formé d'éléments qui appartiennent à l'un des ensembles sans appartenir aux deux.

```
>>>
>>> chn = "abracadabra"
>>> ens5 = set (chn)
>>> print (ens5)
{'a', 'r', 'b', 'c', 'd'}
>>>
>>> nup = ( 1, 1, 2, 2, 3, 3 )
>>> ens6 = set (nup)
>>> print (ens6)
{1, 2, 3}
>>>
>>> chn1 = "abracadrabra"
>>> chn2 = "alacasazerie"
>>> ens1 = set (chn1)
>>> ens2 = set (chn2)
>>> print (ens1, ens2)
{'a', 'r', 'b', 'c', 'd'} {'a', 'c', 'e', 'i', 'l', 's', 'r', 'z'}
>>>
>>> # REUNION
>>> # opérateur |
>>> #
>>> print (ens1 | ens2)
{'a', 'c', 'b', 'e', 'd', 'i', 'l', 's', 'r', 'z'}
>>>
>>> # DIFFERENCE
>>> # opérateur -
>>> #
>>> print (ens2 - ens1)
{'i', 's', 'z', 'e', 'l'}
>>>
>>> # INTERSECTION
>>> # opérateur &
>>> #
>>> print (ens2 & ens1)
{'a', 'c', 'r'}
>>>
>>> # DIFFERENCE SYMETRIQUE
>>> # opérateur ^
>>> #
>>> print (ens1 ^ ens2)
{'b', 'e', 'd', 'i', 'l', 's', 'z'}
>>>
```

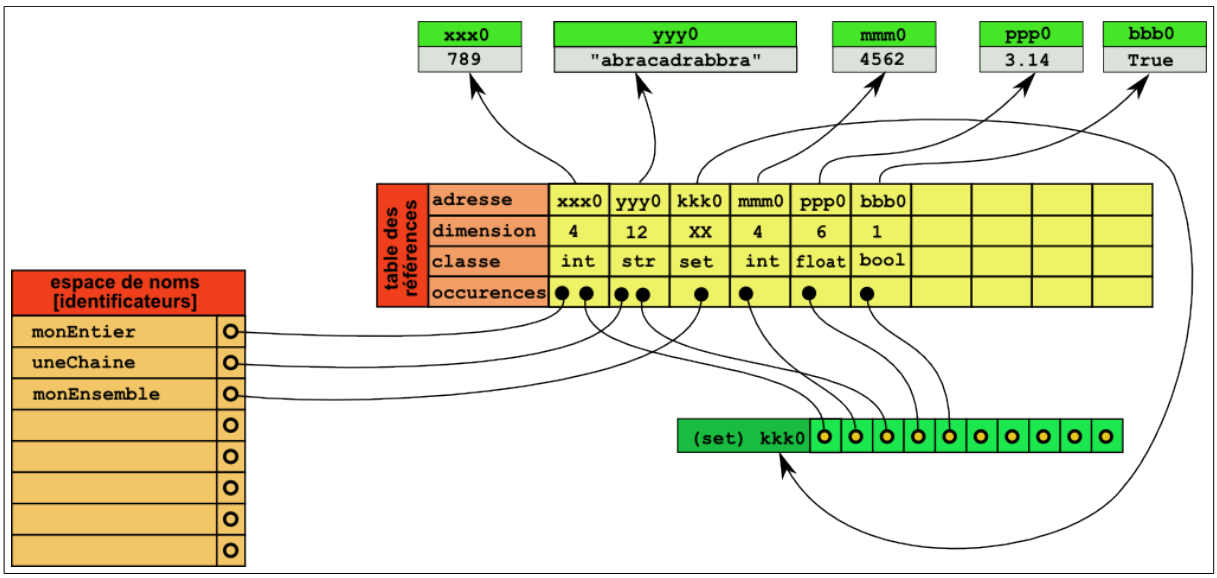
3. Balayage d'un ensemble.

```
>>>
>>> ens = set ("abracadabra")
>>> print (ens)
{'a', 'r', 'b', 'c', 'd'}
>>> for x in ens :
        print ("ceci est la lettre",x)


ceci est la lettre a
ceci est la lettre r
ceci est la lettre b
ceci est la lettre c
ceci est la lettre d
>>>
```

4. représentation en mémoire.

La représentation est réduite : pas d'index comme pour les listes ; pas de table de clefs comme pour les dictionnaires.

```
>>> monEntier = 789
>>> uneChaine = "abracadabbra"
>>> monEnsemble = {monEntier, 4562, uneChaine, 3.14, True }
>>> print (monEnsemble)
{True, 4562, 'abracadabbra', 789, 3.1400000000000001}
>>>
```



fiche 10 : classes et instances

introduction.

Python est un **langage orienté objets (OOL).** On a souvent évoqué la notion d'objet dans les fiches précédentes : un entier, un flottant, un booléen, une liste, un dictionnaire, une chaîne sont des objets. On a évoqué la notion de type, qui rassemble les caractéristiques communes à tous les objets qui lui appartiennent. Mais cela a été fait de façon informelle. En programmation orienté objet, on parle plutôt de classes d'objets que de type, et d'instances de ces classes plutôt que d'objets. C'est que l'on peut créer **des types d'objet formellement appelées classes, et des objets engendrés par ces classes, appelées instances de ces classes**.

Étant donné la complexité de la notion de classe, ce concept va être abordé par étapes, pour se trouver défini de manière à peu près complète à l'issue de la fiche 11 (héritage) et 12.

1. La classe : attributs et méthodes

1.1. Première approche.

Une classe est une encapsulation d'attributs "de classe" et de fonctions appelées méthodes. On peut trouver en plus du code pour initialiser la classe et exécuté lors de la création de la classe ; ce code est habituellement appelé le constructeur d'instances de la classe.

Un attribut est tout simplement une variable ; on ajoute "de classe", car on va rencontrer d'autre genres d'attribut. Une méthode est une fonction définie dans une classe. **L'encapsulation est un mode d'existence des attributs et méthodes : ils ne sont connus en tant que tel qu'à l'intérieur de la classe**. Pour y avoir accès, il faut d'abord se référer à la classe (à affiner ultérieurement) puis à la variable ou la méthode. On appelle ce mode de référencement la **qualification.**

Il faut rappeler que dans un corps de fonction on ne peut utiliser sans précaution que les paramètres et les variables locales ; les attributs de la classe doivent donc être qualifiés

1.2. Un exemple de référence.

On va définir une classe utilitaire que l'on va appeler Cercle, dans laquelle on va encapsuler l'attribut `pi` (le flottant 3.141598), et définir les méthodes `circonference()`, `surface()` et `arc()`.

Ces méthodes ont un paramètre formel `rayon`. La méthode `arc()` a en plus un paramètre formel `angle` qui est l'angle au centre, exprimé en radians. La première méthode retourne la circonférence d'un cercle sous la forme d'un flottant, la seconde sa surface, la troisième méthode retourne sous forme d'un 2-uplet la longueur de l'arc et la surface du secteur limité par l'arc.

l'exemple :

```
#
# définition de la classe
#
class Cercle :
    pi = 3.141598
    db_pi = 2 * pi
    c, s, a = "circonférence :", "aire :" ,"mesures d'arc :"

    def circonference (rayon) :
        return rayon * Cercle.db_pi

    def surface (rayon) :
        return rayon * rayon * Cercle.pi

    def arc (rayon, angle) :
        return ( rayon * angle, rayon * rayon * angle / 2 )

#
# programme test
#
r = 2.5 # le rayon
c = Cercle.pi / 3 # angle de 60 degrés
#
print (Cercle.c, Cercle.circonference (r))
print (Cercle.s, Cercle.surface (r))
print (Cercle.a, Cercle.arc (r, c))
#
# fin du test
```

* l'indentation montre bien ce qui est défini dans la classe Cercle.

* ne pas oublier le signe deux-point à chaque définition d'une classe ou d'une méthode.

* il n'y a pas de confusion avec la variable **`c`** : si on fait référence à la variable globale, c'est **`c`** seulement ; si c'est à l'attribut de Cercle, il faut qualifier : **`Cercle.c`** et **`arc(r, c)`**.

résultat :

```
>>>
circonférence : 15.70799
aire : 19.6349875
mesures d'arc : (2.6179983333333334, 3.2724979166666666)
>>>
```

2. Instanciation d'une classe.

2.1. instance.

Une classe sert à fabriquer des instances.

On a déjà rencontré des instances de classe : à chaque fois que l'on dit qu'une donnée (une chaîne, une liste, un tuple…) est d'un certain type (**`str, list, tuple`**…). Selon la terminologie actuelle, le type est une classe, la donnée est une instance de la classe. Mais on a, jusque maintenant, peu montré le passage de la classe à l'instance. Ce que nous avons appelé cast est en fait une instanciation cachée. Les types inclus dans Python cachent un peu le fonctionnement de l'instanciation ; en effet, écrire un littéral, c'est aussi créer une instance.

Dans l'exemple de la première partie, on a défini un utilitaire Cercle. Mais en pratique, on peut avoir besoin dans un programme de manipuler plusieurs cercles, caractérisés par les coordonnées de leur centre et leur rayon ; et par exemple avoir besoin de savoir si un point donné est intérieur ou extérieur à l'un ou l'autre de ces cercles.

Un cercle particulier va ainsi être caractérisé comme étant "**une instance de la classe Cercle**". C'est simple :

```
monCercle = Cercle ()
```

Adjoindre un attribut à une instance.

Mais en plus, des attributs qui lui sont propres : les coordonnées de son centre et son rayon. Ces attributs sont appelés "attributs de l'instance".

Pour adjoindre un attribut à une instance, il suffit de qualifier l'instance et de l'initialiser. Exemple :

```
monCercle.x = 4.1
monCercle.y = 8.0
monCercle.r = 2.5
```

On peut le faire dans le programme une fois créé **`monCercle`** ; ce n'est pas une bonne solution, puisqu'il faut recommencer pour chaque nouveau cercle que l'on va créer !

Et de plus, la classe ne comporte aucune méthode relative aux instances.

Attention : les instances ont accès en lecture seulement aux attributs de classe. Il faut en plus qu'un attribut d'instance n'ait pas le même nom que l'attribut de classe !

2.2. déclaration de méthode d'instance.

Note : lorsqu'une méthode est une méthode de classe on le dit ; comme on utilise plus souvent des méthode d'instance, on omet souvent le mot instance, étant sous-entendu que si l'on dit "méthode" sans préciser, il s'agit de méthode d'instance. Python ne fait pas la différence formelle entre les deux.

`self`. Le premier paramètre qui apparaît dans les méthodes d'instance désigne l'instance qui appelle la méthode. On utilise en général le mot **`self`** pour ce paramètre mais ce pourrait être n'importe quel identificateur. Il est sous-entendu lors de l'appel, ou plus précisément, **`self`** désigne **l'instance qui est qualifiée par la méthode au moment de l'appel**.

Attention : appeler une méthode de classe depuis une instance ou le contraire suppose une maîtrise parfaite des paramètres. Sauf cas exceptionnels, c'est à éviter (voir exemple 3.2.) !

Voici un programme dont l'intérêt est de montrer le mécanisme, mais qui est mal venu du point de vue programmation : en effet, il ne lève pas l'hypothèque liée à la déclaration d'attributs.

```
#
# programme pour montrer le self
#
class Cercle :
    pi = 3.141598
    db_pi = 2 * pi

    def circonference (self) :
        return self.rayon * Cercle.db_pi

#
# programme test
#
cercle_A = Cercle()
cercle_A.rayon = 2.5
cercle_B = Cercle()
cercle_B.rayon = 6.3
cercle_C = Cercle()
print ("longueur de la circonférence des cercles :")
print("  cercle A :", cercle_A.circonference())
print("  cercle B :", cercle_B.circonference())
print("  type du cercle C :", type(cercle_C))
print("  cercle C :", cercle_C.circonference())
#
# fin du test
```

* **`circonférence()`** est maintenant une méthode d'instance. Elle utilise un attribut, **`rayon`**, de l'instance depuis laquelle la méthode est appelée.

* Le programme définit trois cercles ; pour les deux premiers, l'attribut rayon est explicité, mais pas

pour le troisième ! Pour ce troisième cercle, afficher son type (sa classe) est licite, pas afficher sa circonférence.

* **`circonférence()`** qualifie une instance et c'est le rayon de cette instance qui est pris en compte !

résultat :

```
>>>
longueur de la circonférence des cercles :
  cercle A : 15.70799
  cercle B : 39.5841348
  type du cercle C : <class '__main__.Cercle'>
Traceback (most recent call last):
  File "D:\python\script\atelierAprg2.py", line 23, in <module>
    print("  cercle C :", cercle_C.circonference())
  File "D:\python\script\atelierAprg2.py", line 9, in circonference
    return self.rayon * Cercle.db_pi
AttributeError: 'Cercle' object has no attribute 'rayon'
>>>
```

* Le type de **`cercle_C`** est bien affiché ; on verra plus tard le sens du **`__main__`**

* Mais évidemment il y a erreur sur la ligne suivante :

l'objet 'Cercle' dont on évalue la circonférence n'a pas l'attribut 'rayon'

2.3. Définir les attributs d'instance lors de l'instanciation.

Il existe une méthode spécifique, appelée le constructeur, qui est appelée lors de l'instanciation. Cette méthode se définit comme une méthode d'instance et elle s'appelle toujours **`__init__()`** (double souligné !!!). Elle permet de déclarer et ainsi initialiser par défaut les attributs d'instance.

Voici le programme précédent, mais cette fois avec la définition de l'attribut rayon lors de l'instanciation. On ne peut que donner une valeur par défaut à l'attribut de l'instance. C'est cette valeur qui est prise en compte si cet attribut n'est pas "revalorisé" avant l'appel de la méthode.

Cette solution évite les exceptions. Il serait plus satisfaisant d'avoir une instanciation paramétrée, de façon à ne pas avoir d'oublis dans l'initialisation ! C'est l'objet du paragraphe suivant.

```
#
# programme : constructeur
#
class Cercle :
    pi = 3.141598
    db_pi = 2 * pi

    def __init__(self) :
        self.rayon = 0

    def circonference (self) :
        return self.rayon * Cercle.db_pi

#
# programme test
#
cercle_B = Cercle()
cercle_B.rayon = 6.3
cercle_C = Cercle()
print ("longueur de la circonférence des cercles :")
print("  cercle B :", cercle_B.circonference())
print("  cercle C :", cercle_C.circonference())
#
# fin du test
```

résultat :

```
>>>
longueur de la circonférence des cercles :
  cercle B : 39.5841348
  cercle C : 0.0
>>>
```

2.4. passage de paramètres lors de l'instanciation

On peut paramétrer le nom de la classe qui sert lors de l'instanciation, comme si celle-ci était un appel de fonction ; si on a créé des paramètres formels dans le constructeur **`__init__()`**, les valeurs d'appel sont passées dans l'ordre aux paramètres du constructeur.

On n'est pas obligé d'avoir le même nombre de valeurs que de paramètres, mais le risque apparaît de voir des paramètres non initialisé. Aussi est-il prudent de placer des paramètres déjà initialisés. On n'évite pas complètement la situation évoquée au paragraphe précédent, mais on minimise les risques.

```
#
# programme : constructeur initialisé
#
class Cercle :
    pi = 3.141598
    db_pi = 2 * pi

    def __init__(self, ray=0) :
        self.rayon = ray

    def circonference (self) :
        return self.rayon * Cercle.db_pi

#
# programme test
#
cercle_B = Cercle(6.3)
cercle_C = Cercle()
print ("longueur de la circonférence des cercles :")
print("  cercle B :", cercle_B.circonference())
print("  cercle C :", cercle_C.circonference())
#
# fin du test
```

On obtient évidemment la même chose que précédemment :

résultat :

```
>>>
longueur de la circonférence des cercles :
  cercle B : 39.5841348
  cercle C : 0.0
>>>
```

3. Exercices d'école.

3.1. Passage de paramètre normal.

Cahier des charges. Dans un repère orthonormé, un cercle est défini par son centre (deux coordonnées) et son rayon. On demande de définir une classe **`Cercle`** dotée des méthodes suivantes : donner la valeur de pi utilisée ; donner la circonférence d'un cercle donné (instance) ; donner son aire ; pour un point donné par ses coordonnées, dire s'il est dans le cercle ou extérieur au cercle.

```
# programme : exercice d'école
#
class Cercle :
    pi = 3.141598
    db_pi = 2 * pi

    def __init__(self, cx=0, cy=0, ray=0) :
        self.centreX = cx
        self.centreY = cy
        self.rayon = ray

    def getPi(self) :
        return Cercle.pi

    def circonference (self) :
        return self.rayon * Cercle.db_pi

    def surface (self) :
        return self.rayon**2 * self.getPi()

    def getPosition (self, x, y) :
        dx = x - self.centreX
        dy = y - self.centreY
        d = dx**2 + dy**2
        return d <= self.rayon**2

#
# programme test
#
monCercle = Cercle(1.0, 5.3, 6.3)
print ("valeur de pi : ", monCercle.getPi())
print ("longueur de la circonférence du cercle :", \
       monCercle.circonference())
print ("surface du cercle :",monCercle.surface())
x = float(input ("abscissee du point étudié : "))
y = float(input ("ordonnée du point étudié : "))
sur = monCercle.getPosition (x, y)
if sur :
    print ("le point de coordonnées", (x, y), \
           "est sur le disque")
else :
    print ("le point de coordonnées", (x, y), \
           "est extérieur au disque")

#
# fin du test
```

* le test d'inégalité sur les flottants est en général à éviter, à cause des approximations liées aux calculs. Dans les cas limites, il peut donner un résultat faux. Mais ce n'est pas le lieu d'en discuter ici.

* Les méthodes de type **`getQQCH()`** sont recommandées en POO. La manipulation directe des attributs (classes ou instances) conduit facilement à des erreurs. avec les méthodes de type **`setQQCH()`** (= poser) et **`getQQCH()`** (= fournir), on sait qu'il faut faire l'effort de programmer minutieusement la classe. Mais ensuite, on ne risque pas de voir une valeur changer par erreur (souvent une homonymie), puisque les deux méthodes sont le seules qui manipulent les attributs. Certains modules de Java obligent à programmer ainsi. C'est aussi une bonne façon de programmer en Python.

* on a utilisé l'opérateur "puissance" formé de 2 étoiles.

résultats sur deux essais :

```
>>>
valeur de pi :  3.141598
longueur de la circonférence du cercle : 39.5841348
surface du cercle : 124.69002462
abscissee du point étudié : 10
ordonnée du point étudié : 10
le point de coordonnées (10.0, 10.0) est extérieur au disque
>>> ================================ RESTART ================================
>>>
valeur de pi :  3.141598
longueur de la circonférence du cercle : 39.5841348
surface du cercle : 124.69002462
abscissee du point étudié : -1
ordonnée du point étudié : 5
le point de coordonnées (-1.0, 5.0) est sur le disque
>>>
```

3.2. Paramètres, méthodes de classe et méthodes d'instance.

déclaration d'une classe :

```
#
# programme : conflit de paramètre
#
class Cercle :
    pi = 3.141598
    db_pi = 2 * pi

    def __init__(self, cx=0, cy=0, ray=0) :
        self.centreX = cx
        self.centreY = cy
        self.rayon = ray

    def getDeuxPi() :
        return Cercle.db_pi


    def getPi(self) :
        return Cercle.pi


    def getParam (self) :
       return (self.centreX, self.centreY, self.rayon)
```

* la méthode **`getDeuxPi()`** est clairement une méthode de classe, n'ayant pas de paramètre.
* la méthode **`getPi()`** a un paramètre, mais il ne sert pas dans le corps de la méthode.
* la méthode **`getParam()`** est utilisable comme méthode d'instance, comme **`getParam()`**.
* **`pi`** et **`db_pi`** sont des variables de classe.

le corps du programme :

```
#
# programme test
#
# création d'une instance de Cercle
monCercle = Cercle(1.0, 5.3, 6.3)

# méthode d'instance appelées normalement
print ("valeur de pi : ", monCercle.getPi())
print ("liste des paramètres du cercle : ",list(monCercle.getParam()),"\n")
#
# méthode de classe appelée normalement
print ("valeur de 2 pi :", Cercle.getDeuxPi(),"\n")

# variable et méthode de classe appelée depuis une instance
print ("variable de classe appelée par une instance : monCercle.db_pi","\n")
try :
    print ("monCercle.getDeuxPi() : ",monCercle.getDeuxPi())
except :
    print ("monCercle.getDeuxPi() n'est pas une instruction valide","\n")

# méthode d'instance appelée depuis une classe
try :
    print (Cercle.getPi())
except :
    print ("Cercle.getPi() n'est pas une instruction valide","\n")

print (" un appel correct possible : Cercle.getPi(0)", Cercle.getPi(0))

print (" un appel correct et valable : Cercle.getParam(monCercle) :   \n",\
       Cercle.getParam(monCercle))
#
# fin du test
```

Les méthodes sont appelées soit depuis la classe, soit depuis une instance. Les parties litigieuses sont protégées (capture d'exception).

le résultat :

```
>>>
valeur de pi :  3.141598
liste des paramètres du cercle :  [1.0, 5.2999999999999998, 6.2999999999999998]

valeur de 2 pi : 6.283196

variable de classe appelée par une instance : monCercle.db_pi

monCercle.getDeuxPi() n'est pas une instruction valide

Cercle.getPi() n'est pas une instruction valide

 un appel correct possible : Cercle.getPi(0) 3.141598
 un appel correct et valable : Cercle.getParam(monCercle) :
 (1.0, 5.2999999999999998, 6.2999999999999998)
>>>
```

* **première remarque** : il y a un conflit de paramétrage avec **`monCercle.getDeuxPi()`**, puisque l'appel envoie la valeur de paramètre **`monCercle`** et que **`getDeuxPi()`** n'a pas de paramètre.

* **seconde remarque :** l'appel depuis une classe, **`Cercle.getPi()`** n'envoie pas de paramètre alors que **`getPi()`** en requiert un. Pour cet exemple, on peut mettre un paramètre quelconque, et l'appel est alors valide. Même chose avec **`Cercle.getParam()`**, mais l'instance est passée explicitement en valeur de paramètre ; **l'appel fonctionne alors comme méthode de classe**, avec comme astuce de lui passer l'instance en valeur de paramètre. Cette manière de faire n'est pas à recommander, sauf dans certains cas d'espèce qu'on verra à la fiche suivante.

fiche 11 : héritage et importation

introduction

Une classe peut être déclarée comme "fille" d'une autre classe. Cela signifie qu'elle hérite de ses variables et de ses méthodes. La règle est la suivante :

> **Lorsque l'on utilise (classe ou instance) dans la fille un attribut ou une méthode dont le nom, défini dans la fille, est aussi reconnu dans l'ascendante, c'est l'attribut ou la méthode de la fille qui est utilisée. Sinon, c'est l'attribut ou la méthode de l'ascendante qui est prise en considération comme si elle était déclarée dans la fille. Cette propriété se propage d'ascendante en ascendante le cas échéant. Il convient évidemment invoquer le constructeur de l'ascendant dans le constructeur de la classe "fille".**

attention : les attributs ou méthodes d'une classe sont connus une fois la classe déclarée ; on ne peut donc, dans la définition d'une classe fille, utiliser un attribut hérité qu'en spécifiant la classe ascendante.

1. Déclaration d'héritage.

1.1. Le formalisme.

L'exemple suivant considère une première classe, **`Figures`**, dont hérite la deuxième classe **`Figure2D`**, dont hérite la classe **`Cercle`**. On a une classe ascendante pour **`Figure2D`** et **`Cercle`** ; on pourrait en avoir plusieurs, car Python connaît également d'héritage multiple (exclus de la présente étude). Par contre, il est habituel qu'une classe ait plusieurs classes filles.

les classes :

```
#
# classe de départ
#
class Figure :
    dimensions = ["1D", "2D", "3D"]

    def getGenre () :
        return "Figure"

#
# une classe fille de Figure
#
class Figure2D (Figure) :

    def getDimension() :
        return Figure.dimensions [1]

    def getFigReg() :
        return False

#
# deux classes filles de Figure2D
#
class Cercle (Figure2D) :

    def getFigReg() :
        return True

class Brisee (Figure2D) :
    pass
```

* la grand-mère **`Figure`**, la mère **`Figure2D`**, les filles **`Cercle`** et **`Brisee`**.

* la méthode **`getFigReg()`** est **surchargée** dans **`Cercle`**, pas dans **`Brisee`**.

le programme de tests :

* mise à part la fonction **`type()`**, les autres méthodes utilisée sont des méthodes de classe. Pour "suivre" l'héritage et voir le choix de la méthode effectivement utilisée, il est plus simple pour l'instant

d'utiliser les méthodes de classe.

* en ce qui concerne les attributs, voir la dernière ligne du programme : l'attribut est hérité de la grand-mère !

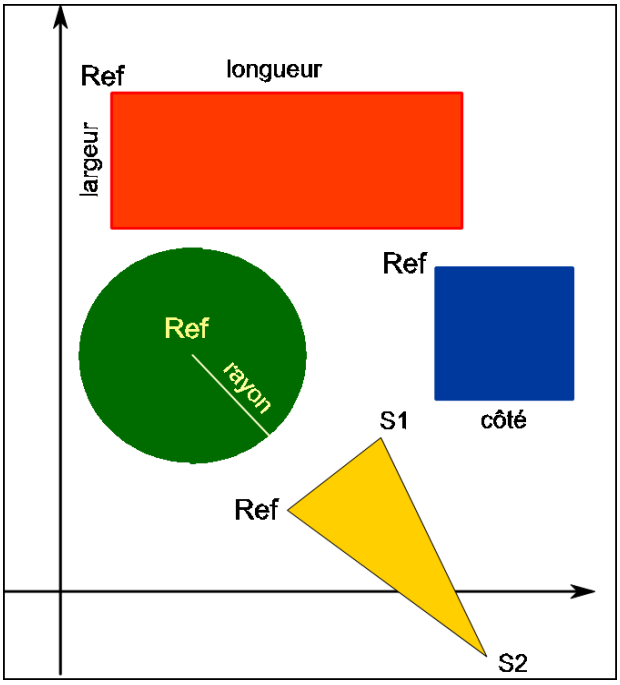
```
#
# programme de tests
#
monCercle = Cercle() # une instance de Cercle
print ("questions d'héritage","\n")
#
print ("  genre d'objet :", Cercle.getGenre())
print ("  les dimensions possibles :",Cercle.dimensions)
print ("  dimension :",Cercle.getDimension())
print ("  classe de monCercle :",type(monCercle),"\n")
#
print ("  Cercle / figure régulière :", Cercle.getFigReg())
print ("  Brisee / figure régulière :", Brisee.getFigReg(), "\n")
print ("  liste des figures :", Brisee.dimensions)

# fin de programme de test
```

le résultat :

```
>>>
questions d'héritage

  genre d'objet : Figure
  les dimensions possibles : ['1D', '2D', '3D']
  dimension : 2D
  classe de monCercle : <class '__main__.Cercle'>

  Cercle / figure régulière : True
  Brisee / figure régulière : False

  liste des figures : ['1D', '2D', '3D']
>>>
```

1.2. Initialisation et héritage.

Pour illustrer les particularités liées à l'héritage, on propose l'exercice d'école suivant :



Dans un plan muni d'un repère euclidien, une Figure2D est un dessin, avec sa couleur, et un point référent qui donne sa place dans le repère ; pour le carré ou le rectangle, ce référent est le sommet en haut et à gauche, pour le cercle, c'est son centre, et pour un triangle, un de ses sommets. Un dessin de ce type peut être translaté : on applique nécessairement la translation à son point référent. Cela est suffisant pour le rectangle ou le cercle. Pas pour le triangle

Le carré, le rectangle, le cercle, le triangle sont définis comme héritant de Figure2D. Chaque classe construit son type d'objet.

les classes :

```
#
# la classe parent
#
class Figure2D :
    # variables privées,
    __listeCouleur = ["noir", "rouge", "vert", "bleu", \
                      "cyan", "magenta", "jaune", "blanc"]
    __listeNature = ["cercle", "rectangle", "carré", "triangle"]

    def getCouleur (i) :
         return Figure2D.__listeCouleur [i]

    def getNature (i) :
         return Figure2D.__listeNature [i]

    def __init__(self, couleur=0, refX = 0, refY = 0) :
        """ constructeur de Figure2D """

        self.nature = ""
        self.couleur = Figure2D.getCouleur(couleur)
        self.topX = refX
        self.topY = refY

    def translater (self, transX, transY) :
        self.topX += transX
        self.topY += transY

    def afficher (self) :
        print ("\ncaractéristiques de l'objet Figure 2D :")
        print ("    nature :", self.nature)
        print ("    couleur :",self.couleur)
        print ("    référence : abscisse", self.topX)
        print ("                ordonnée", self.topY)
```

* **__listeCouleur** : un nom de variable de classe ou de méthode qui commence par un double souligné et ne se termine pas par un double souligné est dite "privée" ; ce nom est connu dans la classe, ignoré en dehors. Ceci permet d'avoir des variables qu'on ne peut changer de l'extérieur de la classe. L'accès aux variables privées ne peut se faire que par des méthodes de la classe (les méthodes **getQQCH()** et **setQQCH()** trouvent ici une pleine utilité).

* C'est dans la classe parent que sont définis les attributs que possèdent toutes les instances des filles : couleur, point de référence. Il en est de même des méthodes communes : translater, afficher.

```
#
#  la classe enfant Cercle
#
class Cercle (Figure2D) :

    def __init__(self, couleur=0, refX = 0, refY = 0, ray=0) :
        Figure2D.__init__(self, couleur, refX, refY )
                        self.nature = Figure2D.getNature (0)
        self.rayon = ray

    def afficher (self) :
        Figure2D.afficher (self)
        print ("    rayon :", self.rayon)
```

* La classe **Cercle** est fille de **Figure2D**. Elle a un attribut de plus : le rayon du cercle ; le constructeur appelle donc le constructeur de la classe mère, comme si c'était une méthode de classe et lui passe explicitement le paramètre self.

* La méthode **afficher()** écrase la méthode de même nom de la classe mère ; il faut donc appeler la méthode de la classe mère comme si c'était une méthode de classe et lui passer explicitement le paramètre self.

* La définition d'instance comporte donc 4 paramètres numériques: numéro de la couleur, coordonnées du centre et rayon.

* On aurait construit la classe **Carre** sur le même modèle.

```
#
#  la classe enfant Rectangle
#
class Rectangle (Figure2D) :

    def __init__(self, couleur=0, refX = 0, refY = 0, long=0, large=0) :
        Figure2D.__init__(self, couleur, refX, refY )
        self.nature = Figure2D.getNature (1)
        self.longueur = long
        self.largeur = large

    def afficher (self) :
        Figure2D.afficher (self)
        print ("    longueur :", self.longueur)
        print ("    largeur :", self.largeur)
```

* La démarche est la même que pour le cercle. Sauf qu'il y a deux attributs spécifiques pour le rectangle, la longueur et la largeur.

```
#
#  la classe enfant triangle
#
class Triangle (Figure2D) :

    def __init__(self, couleur=0, refX = 0, refY = 0,\
                 r1X=0, r1Y=0, r2X=0, r2Y=0) :
        Figure2D.__init__(self, couleur, refX, refY )
        self.nature = Figure2D.getNature (3)
        self.r1X = r1X
        self.r1Y = r1Y
        self.r2X = r2X
        self.r2Y = r2Y


    def afficher (self) :
        Figure2D.afficher (self)
        print ("    deuxième sommet :", (self.r1X, self.r1Y) )
        print ("    troisième sommet :", (self.r2X, self.r2Y) )

    def translater (self, transX, transY) :
        self.topX, self.topY = self.topX+transX, self.topY+transY
        self.r1X, self.r1Y = self.r1X+transX, self.r1Y+transY
        self.r2X, self.r2Y = self.r2X+transX, self.r2Y+transY
```

* le triangle est défini par ses trois sommets. On a déjà le sommet référence ; il reste à adjoindre les deux autres sommets, ce qui donne 4 attributs supplémentaires.

* Mais en plus, le triangle est connu par ses trois sommets, qui évoluent dans la translation : il faut donc gérer de façon spécifique les deux sommets supplémentaires. La méthode **`translater()`** est donc redéfinie ; il est plus simple de le faire complètement. On a utiliser ici l'affectation multiple pour des raisons de lisibilité.

```
#
# programme proprement dit
#
monCercle = Cercle (2, 5.0, 6.0, 2.0)
monCercle.afficher ()
monCercle.translater (-1, -2)
print ("\n","-"*15, "translation du cercle (-1, -2)","-"*15)
monCercle.afficher ()
print ("/ "*30)
#
monRectangle = Rectangle (3, 2.0, 9.0, 7.5, 6.5)
monRectangle.afficher ()
monRectangle.translater (2, 1)
print ("\n","-"*15, "translation du rectangle (2, 1)","-"*15)
monRectangle.afficher ()
print ("/ "*30)
```

```
#
monTriangle = Triangle (4, 1.0, 3.0, 2.5, -3.5, 4.5, 6)
monTriangle.afficher ()
monTriangle.translater (-5, -12)
print ("\n","-"*15, "translation du triangle (-5, -12)","-"*15)
monTriangle.afficher ()
print ("/ "*30)
```

* pour chaque classe, on définit une instance, on affiche se caractéristiques, puis on translate et on affiche les nouvelles caractéristiques.

le résultat :

```
caractéristiques de l'objet Figure 2D :
    nature : cercle
    couleur : vert
    référence : abscisse 5.0
                ordonnée 6.0
    rayon : 2.0

 --------------- translation du cercle (-1, -2) ---------------

caractéristiques de l'objet Figure 2D :
    nature : cercle
    couleur : vert
    référence : abscisse 4.0
                ordonnée 4.0
    rayon : 2.0
/ / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / /

caractéristiques de l'objet Figure 2D :
    nature : rectangle
    couleur : bleu
    référence : abscisse 2.0
                ordonnée 9.0
    longueur : 7.5
    largeur : 6.5

 --------------- translation du rectangle (2, 1) ---------------

caractéristiques de l'objet Figure 2D :
    nature : rectangle
    couleur : bleu
    référence : abscisse 4.0
                ordonnée 10.0
    longueur : 7.5
    largeur : 6.5
/ / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / /
```

```
caractéristiques de l'objet Figure 2D :
    nature : triangle
    couleur : cyan
    référence : abscisse 1.0
                ordonnée 3.0
    deuxième sommet : (2.5, -3.5)
    troisième sommet : (4.5, 6)

 --------------- translation du triangle (-5, -12) ---------------

caractéristiques de l'objet Figure 2D :
    nature : triangle
    couleur : cyan
    référence : abscisse -4.0
                ordonnée -9.0
    deuxième sommet : (-2.5, -15.5)
    troisième sommet : (-0.5, -6)
/ / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / /
```

1.3. la méthode __init__()

Que se passe-t-il en cas d'héritage en ce qui concerne la méthode __init__() ? La méthode **`__init__()`** respecte la règle de la surcharge : surchargée dans la fille, elle occulte la déclaration dans le parent.

- cas 1 : la méthode `__init__()` n'existe ni dans la mère, ni dans la fille.

Il n'y a aucune obligation à déclarer une méthode __init__() ; rien de particulier dans ce cas.

- cas 2 : la méthode `__init__()` est déclarée dans la fille et pas dans la mère.

Il n'y a rien de particulier non plus dans ce cas : il suffit de créer les instances de la fille en paramétrant correctement, c'est-à-dire en créant l'instance avec le bon nombre de paramètres et du bon type.

- cas 3 : la méthode `__init__()` est déclarée dans la mère mais pas dans la fille.

Il n'y a pas de surcharge. La fonction d'initialisation de la mère est appelée ; il faut donc paramétrer correctement la création de l'instance !

- cas 4 : la méthode `__init__()` est déclarée dans la mère et aussi dans la fille.

Seule ma méthode de la fille est appelée. Mais en général, la méthode de la mère n'est pas innocente puisque par exemple, il peut y avoir des attributs de l'instance définis dans la mère. Il faut donc dans ce cas réaliser un appel à la méthode de la mère comme méthode de classe, c'est-à-dire sous la forme : **`Parent.__init__(self, <paramètres d'initialisation de la mère>`**).

Ces divers cas se rencontreront ultérieurement et seront illustrés dans les études qui suivent.

2. Notion de module.

2.1. formalisme.

La notion de module est assez complexe en Python. On ne voit ici que quelques aspects pratiques de l'importation d'un ou plusieurs modules.

Un module est un regroupement de classes, de fonctions et de code "programme" (instructions) qui sert à l'initialisation du module. Ce code est exécuté une fois, au chargement du module.

Le module est sauvegardé dans un fichier d'extension **`.py`** ou **`.pyc`** et le nom du module est celui du fichier. On recommande de n'écrire les noms de module qu'en minuscules (avec chiffres et souligné) pour éviter un traitement différent sous Linux et Windows (Windows, dans ses noms de fichier, ne fait

pas la différence de casse alors que Linux la fait).

Pour utiliser un module, on peut l'importer, avec l'instruction **`import nom_de_module.`** On peut importer plusieurs modules en séparant les noms par une virgule. Le module principal s'appelle __main__ et il ne peut être un fichier importé ; c'est lui l'importateur principal (les modules importés peuvent aussi importer des modules). Une collection de modules est une librairie. On a vu que quand on demandait le type d'une instance on avait un résultat du genre : **`__main__.Cercle`** ; on a un nom du module (**`__main__`**) qualifié (**`Cercle`**). Il y a un problème d'accès lorsque les modules ne sont pas dans le répertoire courant. Il y a plusieurs façon de régler le problème : soit on définit la variable d'environnement PATH de façon à ce que l'accès puisse être recherché ailleurs, soit on définit une variable d'environnement PYTHONPATH à la façon de la variable PATH, avec priorité à celle-ci…

2.2. la clause "import".

On reprend le programme précédent et on le découpe en trois modules : le module **`test_parent`**, le module **`test_filles`** et le modules principal dans le fichier **`test_module`** (tous trois avec l'extention **`.py`**). Pour faire court, les textes sont tronqués !

fichier : test_parent.py

```
#
# la classe parent
#
class Figure2D :
    # variables privées,
    __listeCouleur = ["noir", "rouge", "vert", "bleu", \

------------------------------------------------------------

        print ("    référence : abscisse", self.topX)
        print ("                ordonnée", self.topY)
```

fichier : test_filles.py

```
import test_parent
#
#  la classe enfant Cercle
#
class Cercle (test_parent.Figure2D) :

    def __init__(self, couleur=0, refX = 0, refY = 0, ray=0) :
        test_parent.Figure2D.__init__(self, couleur, refX, refY )
        self.nature = test_parent.Figure2D.getNature (0)
        self.rayon = ray

    def afficher (self) :
        test_parent.Figure2D.afficher (self)
        print ("    rayon :", self.rayon)
#
#  la classe enfant Rectangle
#
```

* les référence du module importé qualifient le nom du module d'origine (**test_parent**).

exemple : **test_parent.Figure2D.afficher (self)**

fichier : test_module.py /// module __main__

```
import test_filles

#
# programme proprement dit
#
monCercle = test_filles.Cercle (2, 5.0, 6.0, 2.0)
monCercle.afficher ()
monCercle.translater (-1, -2)
print ("\n","-"*15, "translation du cercle (-1, -2)","-"*15)
monCercle.afficher ()
print ("/ "*30)
#
monRectangle = test_filles.Rectangle (3, 2.0, 9.0, 7.5, 6.5)
------------------------------------------------------------------------
#
print ("type de monCercle :",type(monCercle))
```

* la dernière ligne donne :

```
type de monCercle : <class 'test_filles.Cercle'>
```

2.3. la clause from…import…

Dans l'exemple précédent, on a utilisé la clause **import.** Cette clause ne fait que donner les indications nécessaires pour pouvoir utiliser le contenu de la classe importée ; si on ne peut atteindre le bon fichier, il y a erreur. Comme on l'a vu, il faut ensuite qualifier le nom du module pour avoir accès aux objets du module.

La clause **from <modules> import <classes>** présente trois particularités :

* on peut importer uniquement les classes dont on a besoin. Si on veut importer toutes les classes, on écrit **import ***

* cette clause ressemble davantage à l'importation en C ou Pascal, ou à la clause include du PHP. Les objets importés sont connus de la classe importatrice : la qualification n'est plus nécessaire s'il n'y a pas de conflit de noms.

* Dans le cas d'un from… import…, le nom de la classe d'importation n'est pas visible dans le module importateur, alors que dans le cas d'un import…, seul le nom du module est visible. On peut utiliser les deux clauses en même temps : la première autorise la qualification si on veut éviter un conflit de nom, la seconde sert à inclure des objets importés.

Voici l'exemple étudié précédemment.

On a changé le nom du module **test_filles** en **test_filles_** (on a rajouté un souligné).

module principal. Le module principal a été modifié pour présenter un conflit de noms avec :

Rectangle = "comment éviter un conflit de noms"

```
monRectangle = test_filles_.Rectangle (3, 2.0, 9.0, 7.5, 6.5)
```

module principal /// fichier : test_module_.py

```
import test_filles_
from test_filles_ import Cercle, Triangle

#
# programme proprement dit
#
Rectangle = "comment éviter un conflit de noms"

monCercle = Cercle (2, 5.0, 6.0, 2.0)
monCercle.afficher ()
monCercle.translater (-1, -2)
print ("\n","-"*15, "translation du cercle (-1, -2)","-"*15)
monCercle.afficher ()
print ("/ "*30)
#
monRectangle = test_filles_.Rectangle (3, 2.0, 9.0, 7.5, 6.5)
monRectangle.afficher ()
monRectangle.translater (2, 1)
print ("\n","-"*15, "translation du rectangle (2, 1)","-"*15)
monRectangle.afficher ()
print ("/ "*30)
#
monTriangle = Triangle (4, 1.0, 3.0, 2.5, -3.5, 4.5, 6)
monTriangle.afficher ()
monTriangle.translater (-5, -12)
print ("\n","-"*15, "translation du triangle (-5, -12)","-"*15)
monTriangle.afficher ()
print ("/ "*30)
#
print (Rectangle)
#
```

module fille

module enfant /// fichier: test_filles_.py

```
from test_parent import Figure2D
#
#  la classe enfant Cercle
#
class Cercle (Figure2D) :

    def __init__(self, couleur=0, refX = 0, refY = 0, ray=0) :
        Figure2D.__init__(self, couleur, refX, refY )
        self.nature = Figure2D.getNature (0)
```

```
        self.rayon = ray

    def afficher (self) :
        Figure2D.afficher (self)
        print ("    rayon :", self.rayon)

#
#  la classe enfant Rectangle
#
-------------------------------------------------------------------
```

Le module **`test parent`** n'a pas été modifié. Le résultat est le même, et le conflit de nom évité ; la dernière ligne est bien :

```
 --------------- translation du triangle (-5, -12)

caractéristiques de l'objet Figure 2D :
    nature : triangle
    couleur : cyan
    référence : abscisse -4.0
                ordonnée -9.0
    deuxième sommet : (-2.5, -15.5)
    troisième sommet : (-0.5, -6)
/ / / / / / / / / / / / / / / / / / / / / / / / / / / / /
comment éviter un conflit de noms
>>>
```

Remarque.

Les fichiers importés ont été compilés lors de l'importation (fichiers **`.pyc`**) de façon automatique. La compilation se fait en pseudo-code, optimisé et d'interprétation plus rapide que le texte du fichier **`.py`**. Les commentaires sont supprimés et le code est optimisé, ce qui est important pour un langage interprété.

fiche 12 : espaces de noms

1. Espace de noms.

1.1. Un genre de dictionnaire.

Rappel : Un **identificateur** est un mot comportant des majuscules, des minuscules, des chiffres, et le caractère souligné (underscore). Il ne commence pas par un chiffre. Quelques mots sont réservés (**`class, def, import, global, return`**…) et ne peuvent être utilisés comme identificateurs. Majuscules et minuscules sont différentiés : **`monObjet`** et **`MonObjet`** sont deux identificateurs distincts.

Python autorise les caractères accentués, mais il vaut mieux les éviter. On travaille ainsi en ASCII pur, où le codage, qu'ils soit ISO ou UTF-8 est le même. Pour les identificateurs de module il vaut mieux en plus ne pas mélanger minuscules et majuscules (on recommande de tout mettre en minuscules dans Python 3…). Avec ces précautions, on évite les problèmes de noms de fichiers quand on change de système d'exploitation.

Lors de la création d'un code global de module ou de classe, lors de la création ou l'exécution d'une fonction ou d'une méthode, lors de l'instanciation d'une classe, Python associe une table, une espèce de dictionnaire dont les clefs sont les identificateurs reconnus dans le bloc de programme, et la valeur, la valeur associée à chacun des identificateurs. Peu importe quand ce dictionnaire est établi ; ce qui importe, c'est **quelles sont les clefs reconnues dans un bloc de code donné**. Ces clefs peuvent appartenir à plusieurs dictionnaires comme dans les fonctions. Les entrées de ces dictionnaires qui sont examinés dans un ordre prédéterminé, constituent **l'espace des clefs** du bloc de programme.

1.2. identificateurs provenant du système.

Le système Python comporte un espace de noms qu'il vaut mieux ne pas redéfinir et qui est accessible de partout : noms de types comme **`int, float, str, list, tuple, set`**… noms de fonction comme **`len(), type()`**…

1.3. espace des noms d'un module.

Un module peut comporter :

* **des instructions d'importation** : les identificateurs des modules importés par **`import`** (pas **`import ... from`**) appartiennent à l'espace des noms. On peut ainsi avoir accès aux identificateurs de l'espace des noms d'un module importé par la qualification (voir la fiche sur héritage et modules).

Les importations avec **`from`** …**`import`**… ajoutent les objets importés au module importateur, et ces objets sont considérés comme lui appartenant ; dans ce cas, l'identificateur du module d'importation n'appartient pas à l'espace des noms et en pratique, ne peut apparaître dans une référence qualifiée. Il peut alors y avoir conflit de noms (on a recommandé dans ce cas, d'importer selon les deux modes, **`import`** et **`import ... from`** ; on peut aussi créer des alias).

* **des définitions de fonctions** : les identificateurs de fonction appartiennent à l'espace des noms du module.

* **des définitions de classes** : les noms de classe appartiennent à l'espace des noms du module. Ainsi les variables de classe et les méthodes de classe sont accessible par qualification de la classe.

* **des variables définies dans le code du module** : elles aussi sont incluses dans l'espace des noms. C'est le cas des instances de classes, et l'accès aux champs et méthodes d'instances se fait par qualification de l'identificateur de l'instance.

Les conflits d'identificateur sont évidemment à éviter, et restent insidieux car ils ne conduisent pas toujours à une erreur trouvée par Python ! Il est de nombreux cas où une variable en occulte une autre de même nom, comme par exemple une variable locale de fonction occulte une globale de même nom.

1.4. espace des noms d'une fonction ou d'une méthode.

Le premier "dictionnaire" est formé des variables locales de la fonction (paramètres formels sans initialisation et variables définies dans son bloc de code par une affectation) ; le deuxième, celui des paramètres initialisés).

On rappelle que dans une déclaration de fonction ou de méthode, les paramètres non initialisées doivent précéder ceux qui le sont ; ceux-ci ne sont pas nécessairement renseignées lors de l'appel. Le passage de paramètre a été étudié dans la section sur les fonctions. Attention, il est assz spécifique à Python. On rappelle l'impossibilité de remplacer le paramètre rééel passé lors de l'appel, mais que dans le cas des objets modifiables, rien n'empêche cette modification !

Attention !

Il y a deux comportements **assez peu intuitifs** qu'il faut signaler :

* l'accès aux identificateurs globaux depuis un corps de fonction est en lecture seule ; si on veut accéder aux variables en écriture (effet de bord), il faut le signaler par une commande **`global`** dans la fonction ou méthode !

* plus déroutant encore, **les variables de classes ne sont pas globales pour les méthodes de la classe**. Cela a pour conséquence pratique qu'on n'y accède qu'à travers la qualification de la classe.

1.5. espace des noms d'une classe.

Une classe est faite de code d'initialisation de la classe, où l'on trouve les affectations définissant les variables de classe, et de définition de méthodes. L'espace des noms de la classe comporte les identificateurs des variables, les fonctions définies dans la classe et l'espace de nom global ; l'accès à une variable globale respecte les mêmes règles que pour les fonctions : en lecture seulement, sauf si la variable est déclarée **`global`** . En cas d'héritage, il faut ajouter la classe parente.

Si la classe est fille d'une autre classe, il est indispensable d'invoquer le constructeur (**`__init__()`**) de la classe parent dans le constructeur de son enfant ; on dispose alors des variables d'instance de la classe parent. L'espace des noms des variables d'instance et des méthodes de la classe parente est alors accessible et il est examiné si la recherche de la clef dans l'espace des noms de la classe enfant a échoué : c'est le principe même de l'héritage ; ainsi si une méthode ou une variable de même nom est définie dans le parent et la fille, c'est la méthode ou la variable de la fille qui est reconnue, celle du parent étant occultée. La recherche peut ainsi se propager dans toute l'ascendance.

2. Alias.

Un "alias" est un identificateur ; il peut recevoir comme valeur la référence, même obtenue par qualification, d'une fonction, d'une méthode d'une classe... et pas seulement de variables. Une méthode devient une fonction dans un cadre global (cf. exemple ci-dessous). En termes plus techniques, un alias est un "pointeur" sur la donnée aliasée ; si on a une affectation à deux identificateurs **`al_objet = val_objet`**, alosrs **`al_objet`** et **`val_objet`** "pointent" la même chose, variable, fonction, classe, méthode.... **Si le contenu (= objet pointé) de `val_objet` évolue, le contenu de al_objet évolue de même. Si le contenu de `val_objet` est changé (nouvelle affectation), al_objet reste inchangé et pointe toujours sur l'ancienne valeur**.

```
#
# exemples d'alias
#
def funAlias() :
    print ("message associé à funAlias")

class Aliasing :
    chn1 = "étude de quelques aliasing"

    def affAlias (self) :
        print("message de la méthode affAlias()")

var = "une variable"
#
varAlias = var
print ("var :",var,"///   varAlias :", varAlias)
varAlias ="autre chose"
print ("var :",var,"///   varAlias :", varAlias)
#
alias_funAlias = funAlias
print ("\n   fonction funAlias : ",end="")
funAlias()
print ("   alias alias_funAlias : ",end="")
alias_funAlias()
#
fun = Aliasing.affAlias
monMessage = Aliasing ()
print ("\n   la méthode donne : ",end="")
monMessage.affAlias()
print ("   l'alias, comme fonction, donne : ",end="")
fun (monMessage)
```

le résultat :

```
>>>
var : une variable ///   varAlias : une variable
var : une variable ///   varAlias : autre chose

   fonction funAlias : message associé à funAlias
   alias alias_funAlias : message associé à funAlias

   la méthode donne : message de la méthode affAlias()
   l'alias, comme fonction, donne : message de la méthode affAlias()
>>>
```

3. Variables privées.

3.1. formalisme.

Un identificateur qui **commence par un double souligné et ne se termine pas par un double souligné** est une variable privée. Si une variable privée est globale, elle ne peut être importée. Si elle est une variable de classe, son accès dans la classe est possible, mais pas à l'extérieur de la classe.

3.2. exemple.

```
#
#  variable privée dans une classe
#
class Secret :
    __maPrivee ="ceci est une variable privée"
    print("dans la classe : ",__maPrivee)

    def getMaPrivee () :
        print ("méthode  getMaPrivee() : ",Secret.__maPrivee)
#
# global
#
Secret.getMaPrivee ()
print ("appel direct de la variable privée :")
print (Secret.__maPrivee)
```

```
>>> ================================ RESTART ================================
>>>
dans la classe :  ceci est une variable privée
méthode  getMaPrivee() :  ceci est une variable privée
appel direct de la variable privée :
Traceback (most recent call last):
  File "D:/python/script/atelierCprg2.py", line 15, in <module>
    print (Secret.__maPrivee)
AttributeError: type object 'Secret' has no attribute '__maPrivee'
>>>
```

fiche 13 : complément sur les types de base

introduction.

On s'intéresse dans cette fiche au types suivants, qui ont une représentation littérale : nombres (entiers et flottants) chaîne de caractère (**`str`**), listes (**`list`**), ensembles (**`set`**), n-uplets (**`tuple`**), dictionnaires (**`dict`**) ; en effet, comme le type de base **`range`** (sans représentation littérale), ces types sont des classes, à la différence près qu'**ils ne peuvent être parents d'une classe fille** (en Java, on les dirait d'attribut "final") . Mais ils ont des instances et des méthodes d'instance.

1. Types modifiables et types non modifiables.

1.1. le concept.

Les types évoqués précédemment sont de types figés : comme on ne peut les prendre comme parents d'une nouvelle classe, on ne peut enrichir l'arsenal des méthodes qui opèrent sur eux. Ainsi, une classe qui ne comporte pas de méthodes de modification (ou leur "raccourcis" ; cf. § 1.2. ss.) a des instances non modifiables.

Le seul recours dont on dispose pour **simuler partiellement** la modification est de remplacer une instance par une autre dans une même variable.

Exemple :

```
ch1 = "tout flatteur vit aux dépens "
ch2 = "de celui qui l'écoute"
ch1 = ch1 + ch2
```

La variable **`ch1`** est un pointeur sur la chaîne **`"tout flatteur vit aux dépends "`** et **`ch2`** sur la chaîne **`"de celui qui l'écoute".`** Les deux chaînes peuvent continuer d'exister après la concaténation, qui n'a fait que créer une troisième chaîne sur laquelle pointe **`ch1`** en ligne 3. On aurait pu espérer une méthode **`concat()`** appelée selon la modalité suivante : **`ch1.concat(ch2)`**. Elle n'existe pas et ne peut être créée car le type **`str`** n'a pas de descendant. La chaîne est un objet non modifiable. Il en est de même des nombres (heureusement !) et du n-uplet.

1.2. objets modifiables.

Les listes, les ensembles, les dictionnaires sont modifiables. On peut leur adjoindre un élément par une méthode d'instance ; on peut supprimer ou remplacer un élément ; on peut adjoindre une liste à une autre :

* **listes** : la méthode **`append()`** permet d'ajouter un élément, **`extend()`** une liste :

```
>>>
>>> lst1 = ["abc", "def"]
>>> lst4 = lst1
>>> lst1.append ("xyz")
>>> print (lst4, lst1, sep ="\n")
['abc', 'def', 'xyz']
['abc', 'def', 'xyz']
>>> lst5 = lst1
>>> lstA = ["tuv"]
>>> lst1 = lst1 + lstA
>>> print (lst5, lst1, sep ="\n")
['abc', 'def', 'xyz']
['abc', 'def', 'xyz', 'tuv']
>>>
>>> lst4 = lst5
>>> lst4.extend(lstA)
>>> print (lst4, lst5, sep ="\n")
['abc', 'def', 'xyz', 'tuv', 'tuv']
['abc', 'def', 'xyz', 'tuv', 'tuv']
>>>
```

On note la différence importante entre le traitement avec concaténation et les traitements par une méthode : avec la méthode **`append()`** et la méthode **`extend()`**, la liste est modifiée et **`lst4`** change de valeur (pointée) en même temps que **`lst1`**.

* **ensembles** : il s'agit de la méthode add()
On peut ajouter *ou retirer* un élément à un ensemble :

```
>>>
>>> ensemble = set ("abracadabra")
>>> print (ensemble)
{'a', 'r', 'b', 'c', 'd'}
>>> bis = ensemble
>>> ensemble.add("xyz")
>>> ensemble.remove ('a')
>>> print (ensemble, bis, sep = "\n")
{'c', 'b', 'd', 'xyz', 'r'}
{'c', 'b', 'd', 'xyz', 'r'}
>>>
```

* **dictionnaires** : il n'y a pas que des **méthodes** pour modifier les objets systèmes ; voici un exemple pour les dictionnaires (il y a des "raccourcis" du même genre pour listes et ensembles avec l'écriture à crochets : [ ])

```
>>>
>>> monDico = { "pierre":1234, "paul" :6789, "jean":1000 }
>>> leBis = monDico
>>> monDico["paul"] = -1
>>> monDico["marie"]=3
>>> del monDico[pierre]
>>> print (monDico, leBis, sep="\n")
{'jean': 1000, 'paul': -1, 'marie': 3}
{'jean': 1000, 'paul': -1, 'marie': 3}
>>>
```

1.3. le piège.

```
# question de listes

# premier mode
composant = [0, 1, 2]
listePremier = []
for i in range (3) :
    listePremier.append (composant)

# deuxième mode
listeDeuxieme = []
for i in range (3) :
    composant = [0, 1, 2] # variable locale
    listeDeuxieme.append (composant)

# affichages
print ("listePremier", listePremier)
print ("listeDeuxieme", listeDeuxieme)

# modification
listePremier[1][2] = 999
listeDeuxieme[1][2] = 999

# affichages après modification
print ("listePremier modifiée", listePremier)
print ("listeDeuxieme modifiée", listeDeuxieme)
```

On a ci-dessus un programme ; il simule un tableau d'entiers à deux dimensions et initialisé. Voici le résultat de ce programme. Expliquer ce qui se passe !

```
>>>
listePremier [[0, 1, 2], [0, 1, 2], [0, 1, 2]]
listeDeuxieme [[0, 1, 2], [0, 1, 2], [0, 1, 2]]
listePremier modifiée [[0, 1, 999], [0, 1, 999], [0, 1, 999]]
listeDeuxieme modifiée [[0, 1, 2], [0, 1, 999], [0, 1, 2]]
>>>
```

2. Question de méthodes.

Le présent paragraphe illustre quelques méthodes relatives aux chaînes, listes, ensembles et dictionnaires. Il est avant tout une incitation à consulter la bibliothèque standard de Python et le Tutorial Python (en particulier le paragraphe Data Structure) dans la documentation officielle.

2.1. Nombres.

* **ce que dit la documentation officielle.**

float.hex()

Return a representation of a floating-point number as a hexadecimal string. For finite floating-point numbers, this representation will always include a leading 0x and a trailing p and exponent.

float.fromhex(s)¶

Class method to return the float represented by a hexadecimal string s. The string s may have leading and trailing whitespace.

* **exemple** :

```
>>>
>>> nombre = 3.1416
>>> hexa = nombre.hex()
>>> print (hexa)
0x1.921ff2e48e8a7p+1
>>>
>>> bis = float.fromhex("0x4.5f3p+5")
>>> print (bis)
139.8984375
>>>
```

2.2. Chaînes.

* **ce que dit la documentation officielle.**

str.join(*seq*)

Return a string which is the concatenation of the strings in the sequence *seq*. The separator between elements is the string providing this method.

str.lower()

Return a copy of the string converted to lowercase.

* **exemple** :

```
>>>
>>> ch1 = "La cigale ayant chanté tout l'été"
>>> ch2 = "Se trouva fort dépourvue"
>>> ch3 = "Quand la bise fut venue"
>>> concat = "\n".join ([ch1, ch2, ch3])
>>> print (concat)
La cigale ayant chanté tout l'été
Se trouva fort dépourvue
Quand la bise fut venue
>>> minus = concat.lower()
>>> print (minus)
la cigale ayant chanté tout l'été
se trouva fort dépourvue
quand la bise fut venue
>>> print (concat.upper())
LA CIGALE AYANT CHANTÉ TOUT L'ÉTÉ
SE TROUVA FORT DÉPOURVUE
QUAND LA BISE FUT VENUE
>>>
```

2.3. Listes.

* **ce que dit la documentation officielle.**

list.append(*x*)

Add an item to the end of the list; equivalent to a[len(a):] = [x].

list.extend(*L*)

Extend the list by appending all the items in the given list; equivalent to a[len(a):] = L.

list.insert(*i*, *x*)

Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).

list.pop([*i*])

Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position.)

list.count(*x*)

Return the number of times *x* appears in the list.

list.sort()

Sort the items of the list, in place.

list.reverse()

Reverse the elements of the list, in place.

* **exemple** :

```
>>>
>>> maListe = ["xxx", "ppp", "qqq"]
>>> maListe.insert ( 1, "kkk" )
>>> print (maListe)
['xxx', 'kkk', 'ppp', 'qqq']
>>>
>>> print (maListe.pop ( 2 ))
ppp
>>> print (maListe)
['xxx', 'kkk', 'qqq']
>>>
>>> print (maListe.count( "qqq"))
1
>>>
>>> maListe.sort()
>>> print (maListe)
['kkk', 'qqq', 'xxx']
>>>
>>> maListe.reverse()
>>> print (maListe)
['xxx', 'qqq', 'kkk']
>>>
```

2.4. ensembles.

* **ce que dit la documentation officielle**.

isdisjoint(*other*)

Return True if the set has no elements in common with *other*. Sets are disjoint if and only if their intersection is the empty set.

issubset(*other*) , issuperset(*other*),

intersection(*other*, ...), union(*other*, ...), difference(*other*, ...)

copy()

Return a new set with a shallow copy of *s*.

* **exemple** :

```
>>>
>>> setUn = set ("abarcadabra")
>>> setDeux = set ("abricot")
>>> print (setUn, setDeux)
{'a', 'r', 'b', 'c', 'd'} {'a', 'c', 'b', 'i', 'o', 'r', 't'}
>>> print (setUn.isdisjoint(setDeux))
False
>>> print (setUn.issubset(setDeux))
False
>>> print (setUn.difference (setDeux))
{'d'}
>>> print (set().union (setUn, setDeux) )
{'a', 'c', 'b', 'd', 'i', 'o', 'r', 't'}
>>>
>>> setTrois = setUn.copy ()
>>> setQuatre = setUn
>>> setUn.add("z")
>>> print (setUn, setQuatre, setTrois, sep="\n")
{'a', 'c', 'b', 'd', 'r', 'z'}
{'a', 'c', 'b', 'd', 'r', 'z'}
{'a', 'c', 'r', 'b', 'd'}
>>>
```

2.5. Dictionnaires.

* **ce que dit la documentation officielle.**

fromkeys(*seq*[, *value*])

Create a new dictionary with keys from *seq* and values set to *value*.

fromkeys() is a class method that returns a new dictionary. *value* defaults to None.

get(*key*[, *default*])

Return the value for *key* if *key* is in the dictionary, else *default*. If *default* is not given, it defaults to None, so that this method never raises a KeyError.

items()

Return a new view of the dictionary's items ((key, value) pairs). See below for documentation of view objects.

keys()

Return a new view of the dictionary's keys. See below for documentation of view objects.

* **exemple** :

```
>>>
>>> dict.fromkeys ( ("un", "deux", "trois"), 1)
{'un': 1, 'trois': 1, 'deux': 1}
>>> dico = dict.fromkeys ( ("un", "deux", "trois"), 1)
>>>
>>> dico = dict.fromkeys ( ("un", "deux", "trois"), 1)
>>> print (dico)
{'un': 1, 'trois': 1, 'deux': 1}
>>> dico ["deux"] = 2
>>> dico ["trois"] = 3
>>> print ( dico.keys())
<dict_keys object at 0x00E63E10>
>>> print ( list(dico.keys()))
['un', 'trois', 'deux']
>>>
>>> print (list (dico.items()))
[('un', 1), ('trois', 3), ('deux', 2)]
>>> print (dico.get("trois"))
3
>>>
```

3. Paramètre des fonctions (et méthodes) et "objets".

3.1. Mécanisme d'affectation (rappel).

* **Lors de la création d'une fonction**, Python crée une table d'identificateurs comportant tous les paramètres initialisés, appelés aussi paramètres à mots-clefs. Il le fait une fois pour toutes et cette table est intangible.

* **Lors de l'appel d'une fonction**, Python crée une autre table d'identificateurs, comportant en clef les paramètres formels non initialisés avec la valeur d'appel et tous les paramètres initialisés explicités dans l'appel.

Par exemple supposons la définition de fonction suivante :

```
def fnct (par1, \
          par2, \
          par3, \
```

```
            par4="une chaîne", \
            par5=3.1416, \
            par6 = ['abc', 2, 4.5])
```

…..

appelée de la façon suivante :

```
          fnct ("toto", \
                69, \
                0.5,\
                par5=3.141598)
```

Les deux tables suivantes sont disponibles (attention à **par5**, présent deux fois) :

table des paramètres formels **créée lors de l'appel**	
par1	➔ **"toto"**
par2	➔ **69**
par3	➔ **0.5**
par5	➔ **3.141598**

table des paramètres clef/valeurs **établie lors de la création de fnct()**	
par4	➔ **"une chaîne"**
par5	➔ **3.1416**
par6	➔ **['abc', 2, 4.5]**

Les objets appartiennent à une classe (entier, flottants, chaînes, listes etc.) ; **le paramètre ne contient pas la copie de l'objet, mais un lien** (on dit aussi une adresse, un pointeur) qui permet d'accéder à la donnée.

* **Lors d'une affectation dans le corps de la fonction**, comment réagit Python ? Le problème a déjà été abordé de fait : une variable Python n'est pas déclarée ; elle est crée dynamiquement par une affectation **si elle n'est pas déjà existante**. Ceci présente une difficulté lorsque l'on est dans un corps de fonction. **Où la machine Python cherche-t-elle la variable affectée ?**

  - si la variable affectée **n'existe que dans l'espace global, et qu'elle a été déclarée `global`**, elle est trouvée dans cet espace (note : on ne peut déclarer **`global`** une variable dont l'identificateur est celui d'un paramètre formel de la fonction).

  - si la variable existe dans la première table, il n'y a pas de difficulté particulière : elle prend la valeur de l'expression affectée.

  - si les deux recherches précédentes on échoué, **une nouvelle variable (locale) est créée dans la première table** et elle est affectée avec la valeur de l'expression d'affectation. Si un paramètre de même nom existe dans la deuxième table, il est occulté. Même chose pour une variable de même nom dans l'espace global.

Conséquences : les variables globales et les paramètres clef/valeur ne sont accessibles qu'en lecture seulement et restent donc inchangées durant l'exécution de la fonction. Mais attention ! Comme les objets "pointés", ce qui reste inchangé, **c'est l'adresse, pas le contenu effectif**. Evidemment, cela n'a aucune importance pour les objets comme les nombres, les chaînes où les n-uplets pour lesquels Python ne fournit aucune méthode de modification. Dans ce cas, il n'y a pas de question particulière à se poser. Tout se passe comme si le passage se faisait par valeur immédiate (comme en Pascal ou en C, ou pour les types de base Java).

Mais que se passe-t-il pour les instances de classes modifiables, comme les listes, les ensembles ou les dictionnaires ? Ou les classes créées par l'utilisateur ?

*** Lors de l'utilisation d'un paramètre comme instance qualifiée** (ex. **`titi.x`**), que se passe-t-il ? La variable/paramètre est recherchée dans la première table **puis la seconde**, puis l'espace global (inutile de déclarer **`global`** dans ce cas, puisqu'il ne s'agit que d'une lecture !). Si elle n'est pas

trouvée, il y a erreur. sinon la méthode est exécutée. Pas de problème tant que la méthode ne modifie pas la valeur pointée ! Mais dans le cas de la deuxième table, s'il y a modification de la valeur pointée, cette modification devient définitive. Ce qui veut dire qu'on a changé la valeur "par défaut", ce qui peut être surprenant ! Lors d'un nouvel appel de la fonction, c'est la nouvelle valeur qui est prise en compte.

On a la même chose avec un paramètre qui a été affecté avec une variable référant un objet modifiable. C'est d'ailleurs cette possibilité qui exploitée lorsque l'on définit une méthode d'instance avec le paramètre **`self`**. L'utilisation de variables qualifiées rend plus naturelle cette possibilité de modifier, puisque c'est la même notation qu'en C++, Pascal, Java etc.

3.2. Modification dans l'espace global.

Pour une liste, la méthode append() ajoute un élément.

```
>>> maListe = ["abc", "def", "ghi"]
>>> print (maListe)
['abc', 'def', 'ghi']
>>> def varGlobale () :
        maListe.append("xyz")


>>> varGlobale()
>>> print (maListe)
['abc', 'def', 'ghi', 'xyz']
>>>
```

3.3. Passage d'un objet modifiable en paramètre.

```
>>> maListe = ["abc", "def", "ghi"]
>>> def parGlobal (par1) :
        par1.append("xyz")
        print (par1)


>>> parGlobal(maListe)
['abc', 'def', 'ghi', 'xyz']
>>> print (maListe)
['abc', 'def', 'ghi', 'xyz']
>>>
```

3.4. Appel avec modification d'un paramètre initialisé

```
>>> maListe = ["abc", "def", "ghi"]
>>> def paramInit (par1, parInit = maListe) :
        parInit.append(par1)
        return parInit

>>> print (paramInit ("xyz", parInit = ["mnp"]))
['mnp', 'xyz']
>>> print (maListe)
['abc', 'def', 'ghi']
>>>
```

3.5. Création d'une variable occultante.

```
>>> maListe = ["abc", "def", "ghi"]
>>>
>>> def paramInit (par1, parInit = maListe) :
        parInit = parInit + [par1]
        return parInit

>>> print (paramInit ("xyz"))
['abc', 'def', 'ghi', 'xyz']
>>> print (maListe)
['abc', 'def', 'ghi']
>>> print (paramInit ("xyz"))
['abc', 'def', 'ghi', 'xyz']
>>>
```

Une nouvelle variable **`parInit`** est créée, qui occulte le paramètre initialisé. Il n'y a pas d'effet de l'affectation sur le paramètre ni la valeur d'appel.

3.6. Un effet surprenant !

```
>>>
>>> def paramInit (par1, parInit = ["abc", "def", "ghi"]) :
        parInit.append(par1)
        return parInit

>>> print (paramInit ("xyz"))
['abc', 'def', 'ghi', 'xyz']
>>> print (paramInit ("xyz"))
['abc', 'def', 'ghi', 'xyz', 'xyz']
>>> print (paramInit ("rst"))
['abc', 'def', 'ghi', 'xyz', 'xyz', 'rst']
>>>
```

La variable initialisée **`parInit`** est modifiée à chaque appel ! **On ne saurait trop insister sur une étude de l'utilisation de la mémoire telle qu'elle a été vue dans le chapitre sur les fonctions.**

fiche 14 : fichiers en Python

introduction.

Le travail sur fichier comporte quatre niveaux d'actions, quel que soit le langage ; seule diffère la syntaxe et les primitives disponibles pour les réaliser :

* **assignation d'un chemin de fichier (nom de fichier, path…) à une variable de fichier**. Si le fichier est sur un support de stockage (mémoire flash, disque dur, CD, DVD…), le chemin respecte les règles du système d'exploitation dans sa gestion du système de fichier : chemin de répertoire + nom propre au fichier. Par exemple **`d:\python\script\monfichier.txt`** est un chemin valide sous Windows ; en Linux, on aurait quelque chose comme **`/home/python/script/monfichier.txt`**. Le fichier peut être un périphérique de sortie (écran, imprimante…) ou d'entrée (clavier…). Sous Windows, certaines librairies acceptent la notation Unix ; mais la cohérence n'est malheureusement pas de règle.

  Remarque : Les principaux systèmes d'exploitation actuels utilisent le "/" comme séparateur. Si l'on donne à Python un chemin utilisant le séparateur standard "/" Python saura l'interpréter correctement : le chemin sera valide et fonctionnera sous Windows. Avec Python sous Windows, nous aurions donc pu, écrire **`d:/python/script/monfichier.txt`** mais pour montrer la particularité de Windows nous avons utilisé le séparateur "\" ce qui oblige à échapper ce caractère **`d:\\python\\script\\monfichier.txt`**. Les exemples donnés ci-dessous sont donc tous spécifiques à Windows.

* **ouverture du fichier**. Une ouverture de fichier est une mobilisation de ressources du système d'exploitation. L'ouverture renseigne celui-ci sur la nature des opérations qui lui sont dévolues : lecture ou écriture ou les deux ; mode du traitement de fichier (traiter en mode texte, en mode binaire, créer un nouveau fichier ou modifier le fichier existant…). Le système d'exploitation réagit en validant les données qui lui sont fournies (chemin de fichier, existence du fichier…) et en préparant les traitements (allocation de buffers…)

* **traitement proprement dit**. Le traitement se fait à travers des primitives adaptées au mode d'ouverture : lecture, écriture, ajout, remplacement partiel etc.

* **fermeture du fichier**. La fermeture d'un fichier est la restitution des ressources mobilisées par les système d'exploitation et utilisées pour réaliser les opérations sur le fichier. Cette fermeture est importante car la gestion d'un fichier demande de la mémoire vive qui en cas d'oubli de fermeture du fichier reste bloquée et indisponible. De plus, au cours de la fermeture, les buffers de fichiers sont vidés ; si on ne ferme pas explicitement un fichier, le fichier physique peut ne pas être mis à jour.

1. Fichier de texte.

1.1. Caractères, chaînes et lignes.

Les langages classiques (C, Pascal, PHP, Java) on leurs "fichiers de textes", mais la réalité recouverte n'est pas toujours exactement la même. En Python, **un fichier de texte est un fichier qui a pour données une chaîne de caractère (on insiste : une chaîne)**. Cette chaîne peut être scandée par le caractère de fin de ligne, codée avec le caractère d'échappement **`\n`**. On peut alors parler de lignes. Les lignes structurent le fichier, ce qui est intéressant dans le cas de gros fichiers : il existe des primitives de gestion des lignes et on peut alors travailler sur des unités plus petites, ce qui ménage les ressources du système d'exploitation. Une ligne, sauf éventuellement la dernière, comporte le caractère de fin de ligne. Il appartient donc au programme de gérer ces fins de lignes.

1.2. L'exemple type (sous Windows).

```
fichier : ficheEprg2.py
# ********************
```

```
# fichiers de texte *
# ******************

chemin = "d:\\python\\script\\essai.txt"
# création
# ########
objetFichier = open (chemin, "w")
objetFichier.write("La cigale ayant chanté\nTout l'été\n")
objetFichier.write("Se trouva fort dépourvue\n")
objetFichier.write("Quand la bise fut venue:\n")
objetFichier.write("Elle alla crier famine\n")
objetFichier.write("Chez la fourmi")
objetFichier.write(" sa voisine\n")
objetFichier.close()

# lecture globale
# ###############
objetFichier = open (chemin, "r")
texte = objetFichier.read()
objetFichier.close()
print ("le début du texte : lecture par read()\n")
print (texte)

# **********************************

fin = """La Fourmi n'est pas préteuse :\nC'est là son moindre défaut.\n\
Que faisiez-vous au temps chaud ?\nDit-elle à cette emprunteuse.\n\
- Nuit et jour à tout venant\nJe chantais, ne vous déplaise.\n\
-Vous chantiez ? j'en suis fort aise.\nEh bien! dansez maintenant."""
# **********************************

# ajout en écriture
# #################
objetFichier = open (chemin, "a")
objetFichier.write("La priant de lui prêter\n")
objetFichier.write("Quelque grain pour subsister\n")
objetFichier.write("Jusqu'à la saison nouvelle.\n")
objetFichier.write("\n")
objetFichier.write("\"Je vous paierai, lui dit-elle,\n\
Avant l'Oût, foi d'animal,\n")
objetFichier.write("Intérêt et principal.\"\n")
objetFichier.write (fin)
objetFichier.close()

# lecture dans une liste
# ######################
objetFichier = open (chemin, "r")
texteListe = objetFichier.readlines()
objetFichier.close()
```

```
print ("\n le texte complet par liste : \n")
for x in texteListe :
    print (x, end="")

# lecture  ligne à ligne
# ######################
print ("\n\n le texte complet ligne à ligne : \n")
objetFichier = open (chemin, "r")
while True :
    ligne = objetFichier.readline()
    if ligne=="":
        break
    print (ligne, end="")
objetFichier.close()
```

* on trouve dans ce script les trois modes d'ouverture d'un fichier de textes : **"w"** (création/écriture), **"a"** (ajout au fichier existant) et **"r"** (lecture)

* on a varié les manières d'inscrire les lignes (littéraux, variables…). Bien noter les fins de lignes ! En l'absence de fin de ligne, les chaînes partielles se concatènent.

* une ligne vide dans le texte comporte uniquement un **"\n"**. La fin de fichier se détecte en lecture par un retour de chaîne vide.

* **read()** lit tout le fichier en une seule chaîne ; **realines()** crée une liste des lignes ; **readline()** lit une ligne à la fois en passant à la ligne suivante à chaque lecture.

1.3. Le résultat.

```
>>>
le début du texte : lecture par read()

La cigale ayant chanté
Tout l'été
Se trouva fort dépourvue
Quand la bise fut venue:
Elle alla crier famine
Chez la fourmi sa voisine

 le texte complet par liste :

La cigale ayant chanté
Tout l'été
Se trouva fort dépourvue
Quand la bise fut venue:
Elle alla crier famine
Chez la fourmi sa voisine
La priant de lui prêter
Quelque grain pour subsister
Jusqu'à la saison nouvelle.

"Je vous paierai, lui dit-elle,
Avant l'Oût, foi d'animal,
```

```
Intérêt et principal."
La Fourmi n'est pas prêteuse :
C'est là son moindre défaut.
Que faisiez-vous au temps chaud ?
Dit-elle à cette emprunteuse.
- Nuit et jour à tout venant
Je chantais, ne vous déplaise.
-Vous chantiez ? j'en suis fort aise.
Eh bien! dansez maintenant.
 le texte complet ligne à ligne :
La cigale ayant chanté
Tout l'été
Se trouva fort dépourvue
Quand la bise fut venue:
Elle alla crier famine
Chez la fourmi sa voisine
La priant de lui prêter
Quelque grain pour subsister
Jusqu'à la saison nouvelle.

"Je vous paierai, lui dit-elle,
Avant l'Oût, foi d'animal,
Intérêt et principal."
La Fourmi n'est pas prêteuse :
C'est là son moindre défaut.
Que faisiez-vous au temps chaud ?
Dit-elle à cette emprunteuse.
- Nuit et jour à tout venant
Je chantais, ne vous déplaise.
-Vous chantiez ? j'en suis fort aise.
Eh bien! dansez maintenant.
>>>
```

1.3 le mode.

```
'r' open for reading (default)
'w' open for writing, truncating the file first
'a' open for writing, appending to the end of the file if it exists
'b' binary mode
't' text mode (default)
'+' open a disk file for updating (reading and writing)
The default mode is 'rt' (open for reading text).
```

1.4. Problème de codage.

encoding **is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent, but any encoding supported by Python can be passed. See the codecs module for the list of supported encodings.**

On peut cependant se poser la question de la façon dont se réalise le codage du fichier. En version 3, le codage interne est l'Unicode ; il n'y a pas de problème pour l'utilisation de la fonction **`chr()`** par exemple dans les cas classiques puisque l'ISO-latin1 et l'Unicode ont pratiquement le même codage

pour les langues européennes. Il n'en est pas de même lorsque l'on passe au mode fichier : même si on évite l'Unicode16 (codage uniforme sur deux octets), il reste que le codage en ISO se fait de toutes façon sur un octet, alors que l'UTF8 est de règle dans le monde Unix et utilise un codage avec un nombre d'octets variable.

En pratique, il faut spécifier le mode d'encodage lors de l'ouverture du fichier :

```
f= open (chemin, encoding="latin_1" , mode= "w")
```

Le fichier est codé en **latin-1** appelé aussi ISO-8859-1; noter que le **mode** est explicité complètement si on le place après **encoding** (l'ordre naturel est chemin, mode, encodage...). On aussi **iso8859_15**, **utf_8** et toutes les autres façons connues d'encoder.

2. Fichiers de nombres, de booléens.

2.1. le problème.

Peut-on créer de la même façon des fichiers d'entiers, de flottants, de booléens ? Il y a deux réponses possibles : soit on travaille sur le cast, et on enregistre les données en mode texte. La solution est simple à mettre en œuvre et ne met en jeu que le Python natif. Il existe une seconde solution, mais elle exige l'appel à un module externe (voir la section suivante).

2.2. L'exemple type.

le fichier : atelierEprg3.py

```
import random

# ****************************
# fichiers de types "simples" *
# ****************************

# fichier d'entiers
# #################

nomFichier = "d:\\python\\script\\entiers.txt"

# création
# --------

fichierEntiers = open (nomFichier, "w")
for x in range (1, 21, 2) :
    fichierEntiers.write (str (x)+"\n")
fichierEntiers.close()

# lecture
# -------
x=0
fichierEntiers = open (nomFichier, "r")
while True:
    x = fichierEntiers.readline ()
    if x=="":
        break
    x = x[0:(len(x)-1)]
```

```
    x = int (x)
    print (x, type(x))
fichierEntiers.close()
print()
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#
# fichier de flottants
# ####################
nomFichier = "d:\\python\\script\\flottants.txt"

# création
# --------

fichierFlottants = open (nomFichier, "w")
for x in range (10) :
    fichierFlottants.write (str (random.random())+"\n")
fichierFlottants.close()

# lecture
# --------
r=0
fichierFlottants = open (nomFichier, "r")
while True:
    r = fichierFlottants.readline ()
    if r=="":
        break
    r = r[0:(len(r)-1)]
    r = float (r)
    print (r, type(r))
fichierFlottants.close()
print()
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#
# fichier de booléens
# ###################
nomFichier = "d:\\python\\script\\booleens.txt"

# création
# --------
b, bl = True, True

fichierBooleens = open (nomFichier, "w")
for x in range (10) :
    b = random.random() > 0.5
    fichierBooleens.write (str(b)+"\n")
fichierBooleens.close()

# lecture
# -------
```

```
fichierBooleens = open (nomFichier, "r")
while True:
    b = fichierBooleens.readline ()
    if b=="":
        break
    bl = b=="True\n" # attention bool("False") est True !
    print (bl, type(bl))
fichierBooleens.close()
```

* **random.random()** : Le premier **random** est le nom du le module, et **random()** nomme la fonction qui retourne un flottant aléatoire entre 0 et 1

* **bl = b=="True\n"** : il y a un piège dans le cast avec le type **bool** ; en effet, il faut ici appliquer la règle du cast automatique des booléens. Les équivalents de False sont **None, ""** et **0** ; tout le reste est **True**, en particulier la chaîne **"False"** ! Ce qui fait que le cast booléen pour **False** est l'une des formes suivantes : **bool(), bool(""), bool(''), bool(0), bool(None)**.

2.3. Résultats.

```
>>>
1 <class 'int'>
3 <class 'int'>
5 <class 'int'>
7 <class 'int'>
9 <class 'int'>
11 <class 'int'>
13 <class 'int'>
15 <class 'int'>
17 <class 'int'>
19 <class 'int'>

0.711775469314 <class 'float'>
0.677964106018 <class 'float'>
0.198143231183 <class 'float'>
0.0482113064804 <class 'float'>
0.0614051375575 <class 'float'>
0.122782162711 <class 'float'>
0.653097030615 <class 'float'>
0.668816294634 <class 'float'>
0.0687990102897 <class 'float'>
0.412693984559 <class 'float'>

True <class 'bool'>
True <class 'bool'>
True <class 'bool'>
True <class 'bool'>
False <class 'bool'>
False <class 'bool'>
False <class 'bool'>
True <class 'bool'>
```

```
False <class 'bool'>
False <class 'bool'>
>>>
```

3. Fichiers d'objets.

3.1. Le problèmes des objets.

La solution donnée dans la section 2 est limitée. Que faire si on a un mélange de types où le cast à partir d'une chaîne est difficile ou impossible (ex : range) ? Il existe une solution qui suivant les contextes est appelée pickling (conservation), mashallisation (rassemblement/triage), sérialisation (adaptation en suites, écrit aussi serialization). Python parle pickling là où Java et la littérature sur les langages parle plutôt de sérialisation. La sérialisation consiste à donner une représentation d'un objet(une instance) sous forme d'une suite binaire, voire un texte, de telle façon que l'objet puisse être reconstitué ensuite. Entre codage et décodage, on a en général soit le stockage sur fichiers, soit la transmission par un média de type réseau.

3.2. le module pickle.

Ce module fait partie du système Python. Il n'est pas le seul permettant la sérialisation, mais c'est le module standard. Il comporte deux fonctions, **`dump()`** et **`load()`** : la première écrit un objet dans un fichier binaire en le sérialisant, et la seconde lit séquentiellement un fichier binaire, objet par objet, en restituant ces objets dans des variables. En guise d'exemple type, on a fait un fichier d'instances objets de base de Python ayant (chaînes, listes…) ou pas (range) une représentation littérale. Mais à peu près tous les objets peuvent être sérialisés.

3.3. exemple de base.

fichier : atelierEprg4.py

```
import pickle
# ************************************
# fichiers d'objets : le module pickle *
# ************************************
a = 5
b = 2.654321
c = True
d = [a,b,c]
e = (a,b,c)
f = "abc"
g = { a,b,c,f }
h = { "a":a, "b":b, "c":c }
i = range (5, 100, 10)
#
print ("******** valeurs à sérialiser par dump() *************")
print ("",a,type(a),"\n",b,type(b),"\n",c,type(c),"\n",d,type(d),"\n",\
       e,type(e),"\n",f,type(f),"\n",g,type(g),"\n",\
       h,type(h),"\n",i,type(i),"\n")
#
nomFichier = "d:\\python\\script\\objets.obj"

# création
# ########
fichier = open (nomFichier, "wb")
```

```
pickle.dump (a, fichier)
pickle.dump (b, fichier)
pickle.dump (c, fichier)
pickle.dump (d, fichier)
pickle.dump (e, fichier)
pickle.dump (f, fichier)
pickle.dump (g, fichier)
pickle.dump (h, fichier)
pickle.dump (i, fichier)
fichier.close()

# lecture
# #######
fichier = open (nomFichier, "rb")
al = pickle.load (fichier)
bl = pickle.load (fichier)
cl = pickle.load (fichier)
dl = pickle.load (fichier)
el = pickle.load (fichier)
fl = pickle.load (fichier)
gl = pickle.load (fichier)
hl = pickle.load (fichier)
il = pickle.load (fichier)
fichier.close()
#
# contrôles
#
print ("******** valeurs lues par load() *************")
print ("",al,type(al),"\n",bl,type(bl),"\n",cl,type(cl),"\n",dl,type(dl),\
       "\n",el,type(el),"\n",fl,type(fl),"\n",gl,type(gl),\
       "\n",hl,type(hl),"\n",il,type(il),"\n")
#
print("\n************ contrôle de la liste ***********")
for x in dl:
    print (x, type(x))
#
print("\n********** contrôle de l'objet range ***********")
for x in il:
    print (x, end="  ")
print()
```

* **`import pickle`** : module importé ; il faut qualifier le nom de module pour les fonctions **`dump()`** et **`load()`**.

* **`"wb", "rb"`** : sous Windows il faut déclarer les fichiers comme binaires.

3.4. résultats.

```
>>>
******** valeurs à sérialiser par dump() *************
```

```
 5 <class 'int'>
 2.654321 <class 'float'>
 True <class 'bool'>
 [5, 2.6543209999999999, True] <class 'list'>
 (5, 2.6543209999999999, True) <class 'tuple'>

 abc <class 'str'>
 {2.6543209999999999, True, 'abc', 5} <class 'set'>
 {'a': 5, 'c': True, 'b': 2.6543209999999999} <class 'dict'>
 range(5, 100, 10) <class 'range'>

******** valeurs lues par load() *************
 5 <class 'int'>
 2.654321 <class 'float'>
 True <class 'bool'>
 [5, 2.6543209999999999, True] <class 'list'>
 (5, 2.6543209999999999, True) <class 'tuple'>
 abc <class 'str'>
 {2.6543209999999999, True, 'abc', 5} <class 'set'>
 {'a': 5, 'c': True, 'b': 2.6543209999999999} <class 'dict'>
 range(5, 100, 10) <class 'range'>


************ contrôle de la liste ***********
5 <class 'int'>
2.654321 <class 'float'>
True <class 'bool'>

********** contrôle de l'objet range ***********
5  15  25  35  45  55  65  75  85  95
>>>
```

4. complément sur le module OS.

4.1. Système d'exploitation.

On ne va voir que quelques utilisations, relatives aux fichiers, de ce module qui couvre en fait les commandes de l'O.S. utilisé.

4.2. Illustration de quelques commandes de fichier.

fichier : atelierEprg1.py

```
import glob

# lister un répertoire
print ("*** lister un répertoire ***")
listeRep = os.listdir('D:\\python\\script\\atelier14')
print (listeRep, "\n\n")

# lister le répertoire avec les chemins
```

```
print ("*** lister un répertoire avec les chemins ***")
listePath = glob.glob('D:\\python\\script\\atelier14\\*')
for x in listePath :
    print (x)
print ()

# lister récursivement les fichiers
print ("*** lister récursivement un répertoire ***")
def listeRepertoire (path):
    l = glob.glob(path+'\\*')
    listeFichiers =[path]
    for i in l:
        if os.path.isdir(i):
            listeFichiers.extend(listeRepertoire(i))
        else:
            listeFichiers.append(i)
    return listeFichiers

# s.extend(x)     same as s[len(s):len(s)] = x

chemin = 'D:\\python\\script\\atelier14'
listeProfonde = listeRepertoire (chemin)

def slashes (chn) :
    start = 0
    accu = 0
    while True :
        start = chn.find ("\\",start)
        if start == -1 :
            return accu
        else :
            start = start + 1
            accu = accu + 1

def printListeProfonde() :
    zero = slashes(listeProfonde [0])
    retrait = ""
    for x in listeProfonde :
        retrait ="    "*(slashes(x)-zero)
        print (retrait, x, sep="")

# str.find(sub[, start[, end]])¶
#    Return the lowest index in the string where substring sub is found,
#    such that sub is contained in the range [start, end].
#    Return -1 if sub is not found.

printListeProfonde()

# Comment savoir si un chemin représente un fichier ?
```

```
print ()
print ("*** savoir si un chemin représente un fichier ***")
def fichier (p) :
    if os.path.isfile(p) :
        t = os.path.getsize(p)  # taille
        print (p, "est un fichier de",t,"ko")
    elif  os.path.isdir(p) :
        print (p, "est un répertoire")
    else :
        print (p, "ne désigne rien")

fichier ("d:\\python")
fichier ("d:\\python\\tkinter\\a1_lotus.gif")
fichier ("d:\\pytho")
```

* il est inutile d'importer le module **os** (invocations du système d'exploitation) car il fait partie du Python actif. Par contre le module complémentaire glob doit l'être.

* Le système et ici Windows ; pour Linux, il faut adapter l'écriture des chemins.

*Les exemples donnés ici ne fonctionnent que sous Windows.

Pour avoir des exemples portable sous tous les systèmes actuel, il faut et il suffit d'utiliser l'écriture standardisé des chemins avec le séparateur « / » ce qui n'empêchera pas Windows de renvoyer des « \ » quand on lui demandera un chemin. Mais les chemins que nous écrivons avec des « / » seront traduit par Python pour fonctionner sous Windows comme sous les autres systèmes actuels.

* la fonction **slashes()** : elle sert à compter les séparateurs (ici, "anti slashes"). Leur nombre caractérise la profondeur du répertoire mis en jeu, et donc l'identation de l'affichage.

* la recherche de répertoires et de leur contenu est récursive, et assez délicate à tracer. Une programmation purement itérative oblige à créer une pile, qui ici pourrait être une liste avec les méthodes **append()** (empilage) et **pop()** (dépilage).

4.3. résultat.

```
>>>
*** lister un répertoire ***
['document', 'images', 'scripts', 'test_filles.py', 'test_filles.pyc',
'test_module.py', 'test_parent.py', 'test_parent.pyc']

*** lister un répertoire avec les chemins ***
D:\python\script\atelier14\document
D:\python\script\atelier14\images
D:\python\script\atelier14\scripts
D:\python\script\atelier14\test_filles.py
D:\python\script\atelier14\test_filles.pyc
D:\python\script\atelier14\test_module.py
D:\python\script\atelier14\test_parent.py
D:\python\script\atelier14\test_parent.pyc

*** lister récursivement un répertoire ***
D:\python\script\atelier14
    D:\python\script\atelier14\document
```

```
        D:\python\script\atelier14\document\atelier1_idle.doc
        D:\python\script\atelier14\document\atelier2_bloc.doc
        D:\python\script\atelier14\document\atelier3_ctrl.doc
    D:\python\script\atelier14\images
        D:\python\script\atelier14\images\png
            D:\python\script\atelier14\images\png\atelier6_imgb.png
            D:\python\script\atelier14\images\png\atelier6_imgc.png
            D:\python\script\atelier14\images\png\atelier6_imgd.png
            D:\python\script\atelier14\images\png\atelier7_img1.png
        D:\python\script\atelier14\images\wmf
            D:\python\script\atelier14\images\wmf\atelierBfigures.wmf
            D:\python\script\atelier14\images\wmf\atelierC_pointeur.emf
    D:\python\script\atelier14\scripts
        D:\python\script\atelier14\scripts\atelierAprg5.py
        D:\python\script\atelier14\scripts\atelierAprg6.py
        D:\python\script\atelier14\scripts\atelierAprg66.py
        D:\python\script\atelier14\scripts\atelierBprg1.py
    D:\python\script\atelier14\test_filles.py
    D:\python\script\atelier14\test_filles.pyc
    D:\python\script\atelier14\test_module.py
    D:\python\script\atelier14\test_parent.py
    D:\python\script\atelier14\test_parent.pyc

*** savoir si un chemin représente un fichier ***
d:\python est un répertoire
d:\python\tkinter\a1_lotus.gif est un fichier de 35826 ko
d:\pytho ne désigne rien
>>>
```