INITIATION À PYTHON

Published : 2013-05-08

INTRODUCTION

Les langages informatiques sont nombreux, très nombreux. Un jour les ordinateurs parleront peut-être même plus de langues que les humains, pardon les humains pourront parler en plus de langues avec les ordinateurs qu'entre eux-mêmes.

Si vous vous lancez dans une première expérience de programmation remarquez que l'on parle de langage et non pas de langues, mais en fait c'est bien le même effort qui est à l'oeuvre : communiquer. Dans un cas, vous communiquez avec un congénère, dans l'autre avec un ordinateur.

L'avantage de communiquer avec un congénère c'est la ressemblance qui vous unit à lui et qui fait que lorsque vous souriez vos, vos semblables comprennent que c'est normalement positif (ça dépend du type de sourire mais la variation entre honnête et narquois ne trompe en général pas longtemps).

L'avantage de communiquer avec un ordinateur, c'est qu'il connaît déjà toute sa langue sur le bout des doigts. Rien à lui expliquer au niveau dictionnaire, grammaire. En revanche cet ami à l'inconvénient d'être entêté et de ne tolérer aucune erreur. Sans quoi, il ne comprend pas. l'intelligence de l'ordinateur s'arrête aux règles qu'on lui a inculquées et c'est bien pour cela que l'on programme, pas pour jacasser autour d'un verre, mais bien lui inculquer de nouvelles tâches qu'il réalisera à notre place par la suite. L'ordinateur est un robot et le langage de programmation le moyen de lui parler.

Au moins la relation est claire :

- vous pourrez être très directif avec l'ordinateur « fais-ci », « fais-ça » et il en vous crachera pas à la figure
- vous pourrez lui faire faire ce que vous voulez « mets du rouge, du vert, du bleu et multiplie par 12 si la couleur est inférieure à celle de mon smoking »

En revanche, il vous dira explicitement quand vos indications sont trop floues, peu précises, mal coordonnées...Vous récoltez la responsabilité de votre pouvoir.

Python, comme son nom l'indique, est un langage malléable qui repose sur une base puissante. Il est considéré comme un langage facile à apprendre parce qu'il y a une grande clarté dans l'expression en python (comparé à d'autres évidemment). Python est à la croisée de plusieurs besoin, à mi-chemin entre différentes technologies, s'avère prêt à être employé dans de nombreux contextes avec une grande rapidité d'adaptation. Python est un compromis parfait pour ceux qui ne veulent pas multiplier les langages.

QUELS SONT LES AUTRES LANGAGES ?

On regroupe les langages selon plusieurs catégories qui sont mobiles selon les besoins. Vous en connaissez déjà certainement, au moins de nom : HTML, CSS, Java (-script), C, C++, Qt, GTK, etc*.

les langages interprétés / compilés

L'une des plus grande différence entre les langages tient dans leur fonctionnement au moment de l'exécution. Les langages compilés sont autonome et peuvent fonctionner d'eux-mêmes, mais seulement sur les environnements pour lesquels ils sont conçus. Les langages interprétés ont toujours besoin d'un interpréteur pour fonctionner, ils sont cependant en général plus portable. Les défenseurs du compilé mettent en général en avance les performances (rapidité d'exécution...) et ceux de l'interprété l'efficience (rapidité de création...).

les langages déclaratifs / structurés

Les langages déclaratifs sont parmi les plus utilisés par le grand public : on y comptera le HTML et le CSS ou des langages de ce type. Dans ce type de langage, toutes les actions sont effectuées quoi qu'il arrive, sans ouvrir de possibilité selon les contextes. Les langages structurés au contraire peuvent laisser plus de possibilité lors de l'exécution mais en son d'autant plus compliqués à utiliser.

Python est un langage structuré, de nature interprété même si des procédés permettent de le compiler. Nous partirons du principe que vous souhaitez réaliser des projets de plus ou moins grande envergure et que python est peut être le langage qui vous y aidera.

POUR QUOI FAIRE SI COMPLIQUÉ ?

C'est souvent la remarque des débutants en programmation, c'est peut-être déjà la vôtre, et par expérience elle ressort à chaque difficulté. Je préfère que l'on prenne les choses de l'autre côté. Les langages informatiques répondent à des besoins, et ces besoins peuvent s'exprimer de façon différentes. Après tout, pourquoi ne parle-ton pas tous anglais ou chinois, ce serait plus simple.

La seconde chose, ce n'est pas la programmation qui est compliqué, c'est ce que l'on souhaite faire ou dire. Vous limitez vous souvent à des phrases sujet-verbe-complément ? Certainement pas. En programmation, c'est identique : il y a des règles simples, mais en général, on en veut plus. En clarifiant bien les objectifs et en préparant bien la façon d'y arriver, cela rend en général les choses beaucoup plus faciles.

* (désolé etc n'est pas un langage, pas encore, un jour peut-être)

A PROPOS DE CE LIVRE

Ce livre a été débuté en janvier 2013 par Cédric Gémy, graphiste et formateur au sein d'Activdesign. Membre actif de FlossManuals, il a déjà mis à disposition plusieurs livres sur Gimp dont l'Initiation à Gimp de FlossManuals, mais a aussi participé à Fontes Libres, Créer un Epub, Scribus, et Cinelerra.

Membre de l'équipe de Scribus, il utilise régulièrement python pour créer des extensions au logiciel.

Elisa de Castro Guerra, a participé à la relecture attentive de ce livre.

Vous pouvez aussi contribuer à cet ouvrage ou à tout autre documentation libre sur le site de FlossManuals.

NOTIONS ESSENTIELLES

1. PREMIER PROGRAMME

Lancer l'interpréteur python. En ligne de commande, cela est très facile. Ouvrez un terminal et écrivez python. Ensuite, à l'invite python commençant par >>>, écrivez

```
1+1
```

Programmer c'est comme utiliser une calculatrice, pas plus compliqué. Ce qui est compliqué, c'est ce qu'on veut faire et savoir comment le faire. Le reste, c'est python qui le fera. Mais pour cela, vous devrez bien définir que vous souhaitez obtenir.

soustraction : -

multiplication : *

division : /

division résultat entier : /

modulo : %

parenthèses : ()

2. CONSERVER LE PROGRAMME

Lancer un éditeur de texte ou encore un éditeur python. Écrivez alors les lignes suivantes :

```
#!/usr/bin/python
1+1
```

Enregistrez le fichier sous le nom ex1.py
Exécuter en ligne de commande dans un terminal avec

```
python ex1.py
```

L'enregistrement des programmes python dans un fichier n'est pas nécessaire mais c'est cependant le seul moyen fiable de préserver les lignes écrites pour pouvoir les réutiliser. En effet, si dans le cadre de ce cours il suffira bien souvent d'utiliser l'historique de l'interpréteur python, dans le cadre de projet plus complets, il sera plus pratique de tout enregistrer. Cela facilitera en particulier le débogage.

À l'heure actuelle, pour ceux qui utilisent des systèmes récents et python 3, il est conseillé d'écrire dans un logiciel compatible utf-8 ce qui permet de ne pas voir de messages d'erreur apparaître dès qu'un accent est trouvé dans le fichier. Dans ce cas, c'est une bonne idée d'en informé python en rajoutant une ligne spécifique en début de fichier :

```
-*- coding: utf-8 -*-
```

ce qui donne au complet :

```
#!/usr/bin/python
1+1
```

Évidemment, ces quelques lignes ne sont pas très sexy et ne réalisent pas grand chose d'extraordinaire, mais en programmation, tout projet commence avec des choses simples. Le vrai problème de ce programme et de ne laisser aucune variation dans les données : il ne peut servir qu'à faire ce calcul particulier soit 1+1, ce qui est un peu limitatif. Un programme devrait pouvoir s'adapter à différentes situations.

3. DÉCOUVRIR LES VARIABLES

La variable est un élément d'un programme dont la valeur pourra changer en fonction des besoins. Il s'agit d'une sorte de boite de stockage (tiroir) dont le contenu pourra varier.

Ainsi au lieu d'écrire :

```
1+1=2
```

on peut écrire :

```
a+a=b
```

Le calcul sera toujours valable même si on remplace 1 par 2. Seul le résultat sera adapté.

```
2+2=?
```

La variable sera donc bien souvent une simple lettre ou un mot qui identifie les éléments à traiter ou à calculer. Cela va rappeler quelques heures douloureuses des maths au collèges, mais tout va bien se passer.

Il faudra assigner (affecter) la valeur a avant que le calcul soit possible en python. Il s'agit simplement d'informer des chiffres qui doivent être pris en compte dans le calcul et intervenir en remplacement des variables (ici les lettres).

L'affectation se fait à l'aide de l'opérateur =

```
a=2
b=a+a
b
```

affiche

```
4
```

Le nom des variables doit être le plus parlant possible. Dans notre cas, a ou b ont l'avantage d'être courts, mais n'expriment pas grand chose. On pourrait utiliser des noms de variables comme age, largeur, hauteur...

Le nom des variables est libre. Cependant, il ne peut contenir que des lettres, chiffres et underscore. Enfin il ne peut commencer par un chiffre et ne doit pas être identique à un mot réservé du langage python. Cette contrainte est importante pour éviter toute confusion.

Dernier point important qui exigera un peu de rigueur : python est sensible à la casse. Essayez :

```
a=2
b=A+100
NameError: name 'A' is not defined
```

est le message renvoyé par l'interpréteur pour vous spécifier qu'il n'a pas compris votre instruction. Ici il vous qu'il ne connaît aucune variable nommée 'A'.

Comme on peut le voir dans les exemples précédents, l'affection peut être simple ou composée. Elle peut soit attribuer une valeur fixe a=2, soit attribuer un calcul b=a+a. Dans ce dernier cas, b sera toujours le résultat de ce calcul quelque soit la valeur donnée à a. Cependant, cette valeur est calculée au moment ou l'affectation est effectuée.

```
a=2
b=a+a
b
```

affiche

```
4
```

mais si l'on continue

```
a=3
b
```

affiche

```
4
```

Il faudra alors respécifier l'affectation pour reforcer le calcul :

```
a=3
b=a+a
b
```

affiche

```
6
```

Ainsi, une affectation est valable tant qu'elle n'a pas été remplacée par une nouvelle et le résultat d'une opération tant que celle-ci n'a pas été reconduite.

4. LES TYPES SIMPLES : STR, INT, FLOAT

Les variables sont typées. Cela signifie qu'elles acceptent un certain type de contenu, pas d'autre, un peu comme une voiture qui ne fonctionne pas avec tous les types de carburant.

Les deux types principaux et les plus simples sont les types permettant l'interprétation des nombres entiers (pour faire des calculs si besoin) et du texte (pour évaluer par exemple ce qu'un utilisateur a saisi comme mot de passe ou autre).

Le typage, c'est-à-dire l'attribution d'un type, se fait principalement au moment de la déclaration :

```
a=2
```

signifie implicitement que la valeur de a est de type entier car 2 est bien un nombre entier.

```
nom="cedric"
```

signifie implicitement que la valeur de cedric est de type texte (nous dirons "chaîne", comme une suite de lettre).

Il existe de nombreux types, mais entier et chaîne sont les plus simples et fréquents. Nous pouvons y ajouter dès maintenant le type float qui est prévu pour les nombres décimaux. Nous obtenons donc l'association suivante :
a=2 integrer nombre entier
a="texte" string chaîne de caractère
a=2.0 float nombre décimal
Là où il va être important de donner une bonne attention aux types tient au fait qu'il est impossible de combiner des données de types différents dans un calcul :

```
a=2
nom="cedric"
b=nom+a
TypeError:cannot concatenate 'str' and 'int' objects
```

Python nous informe qu'on ne peut ajouter des chiffres et des lettres. Jusque là tout est normal, même si dans certains cas, comme nous le verrons, cela devra être contourné. Vous devez vous dire "Mais qu'est ce qu'il est bête !". C'est un peu vrai mais vous conviendrez que les cas de devoir effectuer des calculs sur des mots sont assez rares ! Et puis regardez maintenant :

```
a=2
b=1.5
c=a+b
c
```

affiche

```
3.5
```

Il est donc possible de faire des calculs entre des entiers et des flottants, bref des chiffres, tout est donc normal. Remarquez simplement la façon d'écrire les chiffres qui n'utilise pas une virgule mais un point (à l'anglaise).

5. AFFICHER DES VALEURS ET RÉSULTATS

Dans le cadre de l'écriture de nos programmes, nous avons souvent besoin d'afficher des résultats. Vous avez certainement remarqués que Python ne fait pas cela immédiatement :

```
a=2
b=3
c=a+b
```

il n'affiche pas le résultat de calcul pour c. Il faut donc lui dire. jusqu'à présent, nous avons simplement provoqué cet affichage en écrivant simplement le nom de la variable.

```
c
```

affiche

```
5
```

mais comment faire si nous souhaitons afficher plusieurs informations. Par exemple :

```
a b c
```

```
Affiche :SyntaxError: invalid SyntaxError
```

Il ne comprend pas ce que nous souhaitons faire et nous informe que la syntaxe de notre ligne est mal faite. Mettre à la suite les noms de variables n'est pas assez clair : doit-il les calculer ? dans ce cas comment ? les afficher ?

Il existe de nombreuses façons d'afficher du contenu, mais pour l'instant, l'utilisation de la fonction print sera amplement suffisant :

```
age=100
print age
```

affiche

100

```
print "j'ai ", age," ans."
```

affiche

j'ai 100 ans.

Remarquez ici que les mots qui doivent apparaître tels quels sont écrits entre guillemets et que les variables n'en ont pas besoin. Les virgules entre chaque parties de la phrase sont absolument nécessaires pour éviter toute erreur. Elles servent à Python pour comprendre que le texte se poursuit, finalement comme des parties d'une phrase. Il s'agit d'une concaténation : ajout d'informations à la suite les unes des autres.

6. INTERAGIR AVEC L'UTILISATEUR

Lorsque nous souhaitons réutiliser nos portions de codes, nous les enregistrons dans des fichiers de manière à ce qu'ils constituent de véritables petits programmes. Le problème que nous rencontrons dans ce cas est de pouvoir transmettre des informations au programme sans avoir modifier le code systématiquement. En particulier, si nous souhaitons calculer le périmètre d'une zone, le calcul étant systématiquement identique, il nous faudra cependant modifier les variables.
Pour :

perimetre=(largeur+hauteur)*2

notre programme serait :

```
largeur = 100
hauteur = 200
perimetre=(largeur+hauteur)*2
print perimetre
```

Pour éviter d'éditer notre fichier, ce qui est long et risque d'y introduire des erreurs par mégarde, nous pouvons mentionner à notre programme de demander à l'utilisateur de saisir les informations quand elles sont nécessaires.

```
largeur = input()
hauteur = input()
perimetre = (largeur + hauteur) * 2
print perimetre
```

Enregistrer votre programme sous le nom perimetre.py et exécutez-le dans une console/terminal :

```
python perimetre.py
```

Dans le cas où la valeur a récupéré est une chaîne de caractère, Python renverra une erreur. Mieux vaut utiliser raw_input.

```
nom_de_forme = raw_input()
```

De nombreuses personnes conseillent d'utiliser raw_input dans tous les cas de figure, mais il faudra alors transtyper la variable.

7. CONNAÎTRE ET MODIFIER UN TYPE

Pour connaître le type d'une variable, il existe deux façons principales qui donnent des résultats sensiblement différents.

TYPE()

Pour connaître le type d'une variable, il suffit d'utiliser une fonction interne à python qui s'écrit invariablement sous la forme :

```
type(nom_de_variable)
```

par exemple :

```
a=100
type(a)
```

qui renvoie

```
<type 'int'>
```

Cela est en particulier utile pour connaître le type qui résulte d'une opération

```
a=100
b=0.5
c=a*b
type(c)
```

qui renvoie le résultat

```
<type 'float'>
```

Python a attribué tout seul le type.

Pour modifier le type de la variable, il faudra utiliser des fonctions spécifiques à chaque type de destination :

```
int() pour convertir en entier
str() pour convertir en chaîne de caractère
```

Pour informer Python de la variable à retyper, il suffit de donner son nom entre les parenthèses.

#reprendre programme avec raw_input >

```
largeur = int(raw_input())
hauteur = int(raw_input())
perimetre = (largeur + hauteur) * 2
```

print perimetre

exo : améliorer ce script pour qu'il soit plus lisible en rajoutant des phrases d'explications à l'utilisateur.

8. COMPARER DES DONNÉES

Nous avons déjà effectué des petits programmes. Mais ils ne suffiront certainement pas à la plupart des situations. C'est une étape importante que de programmer en prenant en compte de critères que nous, développeurs, n'aurions pas besoin nécessairement d'utiliser à un moment donné. La variable était déjà une façon de prendre en compte la diversité des valeurs possibles pour une information. Maintenant, il existera des cas dans lesquels le calcul ne peut être effectué ou ne doit l'être. Pour détecter ces cas, il suffira de comparer les données fournies avec des valeurs spécifiées pour s'assurer que certaines conditions sont remplies. Par exemple, gardons notre programme calculant le périmètre, mais incluons de quoi calculer les dimensions en fonction d'une unité fournie. Pour cela, nous demanderons à l'utilisateur de choisir entre deux possibilités, nous testerons ce qu'il a saisi et afficheront le résultat dans l'unité de son choix.

La comparaison la plus simple se fait à l'aide de if...else... s'écrivant sous une forme bien particulière :

```
if comparaison :
    action à effectuer si la comparaison est valable
else :
    action à effectuer si elle ne l'est pas
```

Dans cette structure de base, sont absolument nécessaires :
les : à la fin de la ligne if et else
la tabulation en début des lignes d'actions, nommées en général bloc d'instruction
Note : le concept d'indentation (tabulation en début de ligne) est un concept fondamental en Python.

Les retraits permettent à Python de comprendre la suite et l'imbrication des actions et d'éviter au programme de se tromper de direction.

Revenons maintenant à notre exemple.

Pour demander les possibilités, nous utiliserons pour l'instant une simple solution : nous écrirons les unités possibles et attendrons que l'utilisateur saisisse la valeur. Nous utiliseront donc print et raw_input().

```
print "votre unité de mesure finale voulue : mètre ou centimètre ?"
unite = raw_input()
```

Il suffit alors de reprendre le programme tel que nous l'avions défini et de nous placer à la dernière ligne.

```
if unite == "mètre" :
    print perimetre, " mètre" #nous considérons qu'il les a écrit en
mètre
else :
    print perimetre*100, " cm"
```

Si le nombre de conditions est supérieur à 2 la structure if...else... ne sera pas suffisante. Dans ce cas, nous pourrons avoir recours à sa variante : if...elif...elif...else...

Elle s'écrira plus précisément sous la forme :

```
if unite == "mètre" :
    print perimètre, " mètre"
elif unité == "centimètre" :
    print perimètre*100, " centimètre"
elif unité == "millimètre" :
    print perimètre*1000, " millimètre"
elif unité == "kilomètre" :
    print perimètre/1000, " kilomètre"
```

Une version plus courte d'écrire les tests simples est souvent utilisée. Il s'agit de l'opérateur dit ternaire.

Alors qu'une condition occupe 4 ligne avec la syntaxe de base, avec l'opérateur ternaire, tout sera sur une seule ligne. Cela peut être pratique quand les lignes se multiplient et les niveaux d'indentation aussi. La syntaxe est la suivante :

```
x = true_value if condition else false_value
```

La forme de cette expression peut sembler un peu étrange du fait qu'elle est inversée et fonctionne directement en mode affectation. Au lieu d'exprimer, si une condition est vraie ou fait ceci, sinon on fait cela, on exprime on fait cela si une condition est vraie sinon va voir ailleurs. Il s'agit d'une petite chose mais qui marche parfaitement pour les petits besoins. Évidemment, le rôle de l'opérateur ternaire n'est pas de prendre la place de blocs d'instructions complexes.

9. COMPARAISONS DE GRANDEUR

Dans la partie Comparer des données, nous avons tester si la valeur saisie par l'utilisateur correspondait exactement à celle que nous cherchions. Mais comment cela se passe-t-il si nous cherchons un ordre de granduer, comme par exemple savoir si un chiffre est plus grand qu'un autre ?

Pour cela, nous utiliserons des opérateurs de comparaison. Ces opérateurs, vous les connaissez déjà certainement et les utilisez presque tous les jours. Commençons par celui que nous avons déjà vu et passons aux plus courants :

```
==  égal à

<   plus petit (inférieur) et strictement plus petit

<=  plus petit (inférieur) ou égal

>   plus grand (supérieur) et strictement plus grand

=>  plus grand (supérieur) ou égal

Il en existe un autre moins évident et pourtant pratique :

!=  différent de

qui sert justement de contraire à la simple comparaison d'égalité.
```

Les cas d'utilisation de ces opérateurs sont nombreux. Demander son âge à un utilisateur d'un site web pour lui donner accès. Valider un code postal ou un numéro de téléphone. Donner accès à des messages ou un contenu dans une langue particulière... Bref, les applications ne manquent pas.

Complétons notre exemple pour qu'il prenne en compte ces critères : afficher les chiffres en kilomètres dès qu'ils sont supérieurs à 1000, en répondre les messages dans en anglais si la personne n'est pas française. Les modifications seront alors assez importante. Le chiffre renvoyé ne sera pas le fruit de la saisie d'unité de l'utilisateur. Cette question ne sera donc plus nécessaire. En revanche, nous devrons demaner la langue. Voilà ce que cela pourrait donner :

```
print "Quelle est votre langue ? "
```

```
langue = raw_input()
if langue != "fr" :
    message = "hello, the length is "
else :
    message = "Bonjour, la longueur est "
unite = "mètre"
if perimetre > 1000 :
    if langue != "fr":br>
        unite = "kilometer"
    else :
        unite = "kilomètre"
    print message, perimetre/1000, unite
else
    print message, perimetre, unite
```

10. ASSOCIER DES COMPARAISONS

Il est parfois nécessaire d'effectuer plusieurs comparaisons successives sur un même élément voire entre plusieurs éléments. La forme d'écriture la plus simple va demander d'écrire de nombreuses lignes sans pour autant augmenter la lisibilité :

```
n = input()

if n < 10 :

    if n > 3 :

        print "n est compris entre 3 et 10"
```

De la même façon, la ligne qui suit teste si le chiffre est bien dans la plage souhaitée :

```
n = input()

if n < 10 and n > 3 :

    print "n est compris entre 3 et 10"
```

L'opérateur *and* permet ainsi d'associer les deux comparaisons en une seule. Dans ce cas, la condition est rempli si le chiffre est compris dans les deux expressions cumulées.

Nous pourrions souhaiter que seulement l'une des expression soit validée. Dans ce cas, le recours à *or* sera plus judicieux :

```
if n < 10 or n > 20 :

    print "ok"
```

sera valable si le chiffre saisi est inférieur à 10 mais aussi s'il est supérieur à 20.

Enfin, l'opérateur *not* peut être utiliser pour exclure une possibilité :

```
if n not 10 :

    print "ok"
```

Dans ce cas, il agira un peu comme *!=*. L'avantage de ces opérateurs est cependant aussi de pouvoir s'appliquer à un type spécial, dit booléen et qui n'accepte que deux possibilités : *True* ou *False*

Ce type d'opérateur est utile pour mémoriser la réussite d'un test, sans pour autant que sa valeur soit préservée (par exemple, lorsque vous êtes connecté à un compte web, on se souvient que vous êtes validés quand vous changez de page, mais il n'est pas nécessaire de renvoyer vos données), ou encore lorsque python renvoie ce type comme résultat. Ainsi, on peut remplacer :

```
if n < 10 or n > 20 :

    print "ok"
```

par

```
if n < 10 or n > 20 :

    n_ok = True
```

et réutiliser ensuite ce résultat :

```
if n_ok == True :

    print "ok"

>> ok
```

ou

```
if n_ok is True :

   print "ok"

>> ok
```

STRUCTURER DES PROGRAMMES

11. DONNÉES STRUCTURÉES FONDAMENTALES
12. PLUS SUR LE TYPE CHAÎNE (STR)
13. LISTES SIMPLES ET TUPLES
14. LES DICTIONNAIRES
15. UTILISATION DES TYPES
16. SÉLECTIONNER DES ÉLÉMENTS DANS LES STRUCTURES
17. SÉLECTION MULTIPLE ET LONGUEUR DANS LES STRUCTURES
18. PARCOURIR UNE STRUCTURE DE DONNÉE
19. AJOUT ET CONCATÉNATION D'ÉLÉMENTS
20. SUPPRESSION D'ÉLÉMENTS
21. AFFECTATION ET REMPLACEMENT DE VALEUR
22. TRIER DES LISTES

11. DONNÉES STRUCTURÉES FONDAMENTALES

Jusqu'ici, nous avons essentiellement travaillé avec des chiffres qui sont des types de données relativement simples. Nous sommes parfois passé par le texte, mais sans rentrer dans le détail de sa manipulation. Tout cela est un peu limitatif et il sera souvent nécessaire de pouvoir associer plusieurs données et de garder leur relation. Certains types de données permettent de décrire de réelles structures et d'associer ainsi les différents éléments qui la compose de façon permanente ou temporaire. Parmi celle-ci, notons les chaînes de caractère, justement mais aussi les listes et les tuples qui sont un cas particulier de liste, ou encore les dictionnaires qui en sont une complexification. Enfin, les classes seront une variante étendu de ce principe, mais nous ne les verrons pas encore.

Pour connaître le type d'une donnée, vous pouvez à tout moment utiliser type(). Ainsi si vous avez une variable i avec la valeur 12, vous pourrez savoir si elle est de type str ou int :

```
i = 12
type(i)
```

affiche

```
<type 'int'>
```

alors que

```
i = "12"
type (i)
```

affiche

```
<type 'str'>
```

Les chaînes de caractères sont des suites de lettres. Les mots sont le type de chaîne de caractère le plus naturel. On peut donc considérer qu'avec python, les mots sont de simples suites de lettres analysées et rendues une à une. Le mot n'est que cet ensemble de lettres mises les unes à la suite des autres comme les maillons d'une chaîne.
Voici un exemple de chaîne

```
phrase = "je suis une chaîne de caractère"
```

globalement, tout valeur affectée à une variable entre guillement est compris par Python comme une chaîne de caractère, même si ce sont des chiffres qui sont entre les guillemets.
Pour changer un type il faudra utiliser une déclaration associée, déjà mentionnée :
str() pour l'utiliser en chaîne et int() pour l'utiliser comme un entier.
Ainsi si

```
i = 12
type(i)

> <type 'int'>
```

en faisant

```
str(i)
type(i)

> <type 'int'>
```

On voit bien que i n'a pas changé de type, alors que

```
n = str(i)
type(n)

> <type 'str'>
```

affecte le type à une nouvelle variable est évite alors toute confusion.

12. PLUS SUR LE TYPE CHAÎNE (STR)

Nous nous attarderons un peu sur le type string. En effet, la manipulation de chiffre avec les langages de programmation ne pose en général pas d'autres difficultés que les compétences mathématiques de l'auteur. Le texte, en comparaison, peut sembler d'un fonctionnement un peu étrange.

Le terme chaîne déjà peut surprendre et pourtant il exprime bien la réalité des choses : il s'agit d'une suite de lettres comme des maillons sont les éléments d'une chaîne métallique. Cela induit plusieurs choses qui rendent le texte «informatique» un peu différent de celui du quotidien :

- lorsque nous parlons, notre expression est fluide et continue, nous ne parlons pas lettre par lettre
- lorsque nous parlons, nous utilisons des structures grammaticales (sujet, verbes compléments...) dont aucun logiciel n'a nativement connaissance. Le mot lui-même est quelque chose de difficilement repérable.

Tout cela induit des difficultés qui ne sont pas insurmontables mais qui demandent un peu de démonstration parce que le texte est souvent une élément très important d'un projet informatique.

Allons donc petit à petit dans les détails :

```
texte = "Mon texte pour commencer"

print texte

> Mon texte pour commencer
```

ÉCRIRE AVEC DES GUILLEMETS

Remarquez à nouveau qu'une chaîne se définit entre guillemets. Si vous omettez les guillemets voici ce qui se passe :

```
texte = Mon texte pour commencer
  File "<stdin>", line 1
    texte = Mon texte pour commencer
```

```
SyntaxError: invalid syntax
```

Bref, cela ne marche pas. En revanche, si votre texte doit lui-même contenir des guillemets, un autre problème va se poser :

texte = "L'ouvreuse m'a dit : "Donnez-moi votre ticket." Je le lui ai donné."

```
  File "<stdin>", line 1
    texte = "L'ouvreuse m'a dit : "Donnez-moi votre ticket." Je le lui ai donné."
                                  ^
SyntaxError: invalid syntax
```

Même erreur. Dans le premier cas, le guillemet manquait ici il est en trop. En fait, python interprète le guillemet avant Donnez comme le guillemet de fermeture de la chaîne et ne comprend pas ce qui se passe par la suite. Il existe plusieurs solutions pour contourner cette difficulté :

Utiliser des guillemets simples

La solution la plus simple consiste à utiliser des guillemets simples (apostrophe) en remplacement de guillemets doubles :

```
 texte = 'Mon texte "essai" pour commencer'
>>> print texte
Mon texte "essai" pour commencer
```

Cela marche bien ici, mais posera un problème dans cet autre exemple :

```
texte = 'L'ouvreuse m'a dit : "Donnez-moi votre ticket." Je le lui ai donné.'
  File "<stdin>", line 1
    texte = 'L'ouvreuse m'a dit : "Donnez-moi votre ticket." Je le lui ai donné.'
                        ^
SyntaxError: invalid syntax
```

Comme des apostrophes sont présentes dans le texte, il y a ici un nouveau conflit. Dans des cas comme ceux-ci, un peu complexe, il faudra trouver une autre solution.

Échapper les caractères avec \

L'échappement de caractère consiste à différencier au sein du texte es caractères a priori identiques mais qui doivent être traité de façon différente par python. Il s'agit de dire : ceci est du code python, et ceci fait parti du texte à afficher.

Le principe est simple : il suffit d'ajouter un \ devant les guillemets ou caractères qui appartiennent au texte à afficher :

```
 texte = "L'ouvreuse m'a dit : \"Donnez-moi votre ticket.\" Je le lui
ai donné."
>>> print texte
L'ouvreuse m'a dit : "Donnez-moi votre ticket." Je le lui ai donné.
```

Le problème est souvent de savoir ce qu'il faut échapper. Généralement, il s'agit de tout caractère syntaxique du langage python qui peut entrer en conflit avec un caractère du texte. Par chance, python a limité au maximum sa syntaxe pour remplacer l'utilisation de caractères par l'indentation, donc cela arrive ici moins souvent que d'autres langages.

Globalement, il s'agit de :

" a remplacer par \"

' a remplacer par \'

et de \ a remplacer par \\

Le caractère d'échappement est aussi utilisé dans un contexte légèrement différent lorsqu'il s'agit d'utiliser des caractères invisibles.

LES CARACTÈRES SPÉCIAUX

Il s'agit toujours de quelque chose de spécial, ces caractères spéciaux. Tout est de avoir ce qui est spécial ou pas. Pour un anglais, un é est un caractère spécial, pas pour nous. il pourra donc sembler étrange à certains.

La méthode traditionnelle pour écrire des caractères est d'utiliser leur valeur dans la table des caractères ASCII avec chr() :

```
print chr(102)
```

```
> f
```

le problème est d'avoir accès à de vrais caractères étranges :

```
print chr(255)
```

```
> ?
print chr(196)
> ?
```

Ici, ces caractères sont simplement introuvables d'où le résultat identique. Il faudra utiliser unichr() en remplacement :

```
print unichr(255)
> ÿ
print unichr(196)
> Ä
```

Reste donc à connaître la correspondance entre ces chiffres et le caractère recherché. Un petit tour sur http://fr.wikipedia.org/wiki/Table_des_caractères_Unicode_(0000-0FFF) peut aider.

Parmi les caractères spéciaux, on retrouve les caractères invisibles du texte, en général ceux qui permettent de faire des retours à la ligne ou des tabulations. Dans ces cas très fréquents, python intègre des solutions standards utilisées dans de nombreux langages par des lettres de raccourci échappées \n pour une nouvelle ligne et \t pour une tabulation.

```
texte = "L'ouvreuse m'a dit : \n\t\"Donnez-moi votre ticket.\" \nJe le
lui ai donné."
print texte

>
L'ouvreuse m'a dit :
    "Donnez-moi votre ticket."
Je le lui ai donné.
```

AGIR SUR LE TEXTE

Modifier la casse

Passer le texte en majuscule ou en minuscule peut avoir plusieurs usages : en particulier, il permet de faciliter des recherches (voir ci-dessous) ou encore d'augmenter la lisibilité en différenciant les lettres dans le terminal sur lequel il n'est pas possible de faire de différenciation de style (gras, italique...).

Python possède 2 fonctions de changement de casse : mettre en majuscule avec upper(), mettre en minuscule avec lower().

```
cherche = texte.upper()
print cherche
> L'OUVREUSE M'A DIT : "DONNEZ-MOI VOTRE TICKET." JE LE LUI AI DONNé.
cherche = texte.lower()
```

```
print cherche
> l'ouvreuse m'a dit : "donnez-moi votre ticket." je le lui ai donné.
```

Chercher une lettre ou un mot

Dans les processus d'automatisation, il n'est pas rare d'avoir à comparer des données. Dans un texte, nous aurons parfois besoin de savoir si une portion de texte est incluse dans une autre. Cette opération s'effectue à l'aide de la méthode find.

```
texte = "L'ouvreuse m'a dit : \"Donnez-moi votre ticket.\" Je le lui ai
donné."
texte.find("ticket")
> 39
texte.find("billet")
> -1
```

Le résultat, s'il est valide, renvoie la position de l'occurence dans le texte. Ainsi si le résultat est inférieur à 0 c'est que le texte n'a pas été trouvé. Comme les lettres majuscules et minuscules représentent des caractères différents en ASCII ou unicode, il est compréhensible que python n'y soit pas indifférent et les différencie lors de la recherche.

```
texte.find("don")
> 61
texte.find("Don")
> 22
```

Pour faire des recherches un ignorant la casse, il faudra soit modifier la casse du texte avant recherche soit utiliser des expressions régulières (qu 'il n'est pas encore temps de voir).

Attention car find ne retourne que la première occurence trouvée, même s'il n'y en a plusieurs.

Remplacer une partie du texte

Il en est de même du remplacement. Il suffira d'utiliser la fonction replace() pour remplacer toutes les occurences d'un texte par autre.

```
print texte.replace("ticket","billet")
L'ouvreuse m'a dit : "Donnez-moi votre billet." Je le lui ai donné.
texte = "L'ouvreuse m'a dit : \"Donnez-moi votre ticket.\" Je le lui ai
donné ce ticket."
print texte.replace("ticket","billet")
> L'ouvreuse m'a dit : \"Donnez-moi votre billet.\" Je le lui ai donné
ce billet.
```

Découper le texte

Enfin, il est parfois nécessaire d'avoir de portions du texte. Cela est particulièrement vrai dans différents formats textes comme le CSV ou autre. Imaginons simplement des donnés utilisateurs contenant ces informations

```
M,Alphonse,1932
```

M pour le sexe, Alphonse pour le prénom, et 1932 pour la date de naissance. En tant que chaîne, chacune des informations n'a pas de réalité en propre. En utilisant split(), python aura connaissance que nous souhaitons les traiter différemment. Il suffira de placer dans les parenthèses le caractères présent dans le texte qui doit servir à faire le découpage :

```
texte = "M,Alphonse,1932"
print texte.split(",")
> ['M', 'Alphonse', '1932']
```

Le texte obtenu n'est plus une chaîne, mais une liste qui, elle, se manipule différemment, c'est ce qu'il est temps de voir.

13. LISTES SIMPLES ET TUPLES

Les listes et les tuples sont de simples suite d'éléments indexés. Dans les listes ou dans les tuples, les éléments qui se suivent ne sont pas nécessairement de même type. Alors que dans une chaîne de caractère, tout élément est compris comme une lettre, dans une liste, un élément chiffre (int) peut suivre un élément chaîne de caractère (str)...
Il existe 2 façons de déclarer une liste :
- soit en créant une liste d'emblée, vide ou non :

```
liste1 = []
liste2 = [1,2,3]
type(liste1)

> <type 'list'>

type(liste2)

> <type 'list'>
```

- soit en typant la variable :

```
liste3 = list()
type(liste3)
> <type 'list'>
```

Le tuple héritera des mêmes procédés :

```
tuple1 = ()
tuple2 = (1,2,3)
tuple3 = tuple()
type(tuple1)

> <type 'tuple'>

type(tuple2)

> <type 'tuple'>

type(tuple3)

> <type 'tuple'>
```

La seule différence visible entre un tuple et une liste consiste dans l'utilisation de crochets dans le cas de la liste et de la parenthèse dans le cas du tuple. Au niveau de l'usage, nous verrons bientôt que la liste est modifiable alors que le tuple est prévu comme un élément figé dont les valeurs ne peuvent être modifiées.

14. LES DICTIONNAIRES

Enfin, parmi les structures de données fondamentales en Python, on notera les dictionnaires : ce sont des listes un peu plus complexes en ce que chacun des éléments qui le compose est au moins composé de 2 membres, l'un étant la clé, l'autre a valeur le tout embrassé par des accolades { }. La clé est la valeur sont associées par un : et les membres se suivent, séparés par des virgules comme dans des listes. L'utilisation de dictionnaire est en particulier utile lorsque les clés sont des mots qui permettent ainsi d'avoir une approche sémantique des données :

```
dictionnaire1 = {}
dictionnaire2 = {'nom':'Descartes','prenom':'René'}
dictionnaire3 = dict()
type(dictionnaire1)

> <type 'dict'>

type(dictionnaire2)

> <type 'dict'>

type(dictionnaire3)

> <type 'dict'>
```

Ainsi, en imaginant 2 variables différentes elles peuvent être représentées sous les formes :
Chaînes de caractère

```
nom = "Descartes"
prenom = "René"
```

dans ce cas, rien n'associe informatiquement les 2 variables, seul le développeur connaît la relation implicite que le nom entretien avec le prénom.
liste ou tuples

```
personne = ["Descartes","René"]
```

dans ce cas, les 2 valeurs sont associées dans un ensemble qui peut être signifiant, mais la nature des éléments n'est pas mentionnée. Cela peut être trompeur dans des cas comme

```
personne = ["David","Vincent"]
```

Comment savoir lequel est le nom et lequel le prénom ?

```
personne1 = {'nom':'Descartes','prenom':'René'}
```

qui enlève toute confusion possible quel que soit les noms et prénoms. Les dictionnaires permettent de nombreux autres applications grâce à des spécificités que nous traiterons ultérieurement, comme les classes, lorsque nous aurons avancé sur notre connaissance et compréhension des mécanismes de Python.

15. UTILISATION DES TYPES

16. SÉLECTIONNER DES ÉLÉMENTS DANS LES STRUCTURES

Connaître l'existence de ces structures ne suffit pas, il faut aussi pouvoir les manipuler. Alors que dans le cas des entiers (int) la manipulation se concentrait sur des opérations mathématiques, ces structures nouvelles étant une façon d'associer des éléments par ailleurs séparés ou compris comme tel, la plupart des procédés seront relatifs à la manipulations des éléments. L'ajout de données est le plus simple.

La sélection des éléments est évidemment fondamentale, soit pour extraire la donnée est l'exploiter dans un nouveau contexte, soit pour la modifier.

Elle peut se faire de deux façons : soit par la position des éléments plutôt utilisé pour les chaînes et listes, soit par la valeur.

SÉLECTION PAR POSITION (INDEX)

Les chaînes, la liste et le tuple

L'accès aux éléments se gère en général par un chiffre qui représente la position de l'élément dans l'ensemble. Ce chiffre, nommé indice, est en général exprimé entre crochet quelque soit le type de structure utilisée :

```
phrase = "Je pense donc je suis."
print phrase
```

```
> Je pense donc je suis.
```

```
print phrase[1]
```

Cette dernière ligne permet d'extraire la lettre placée en position 1 de la chaîne "phrase" dont la valeur est "Je pense donc je suis." Le résultat attendu est donc ?

```
> e
```

Ici, Python renvoie "e". Vous direz oui mais la première lettre est le "J". Certes, mais en Python comme dans de nombreux cas en informatique, on pourra estimer que la numérotation commence à 0. Il nous faudra donc tout décaler en cherchant notre indice :

```
print phrase[0]
> J
```

En ce qui concerne la liste ce sera la même chose :

```
liste1 = ("René", "Descartes")
print liste1[1]

> Descartes
```

et le tuple

```
tuple1 = ["René","Descartes"]
print tuple1[1]

> Descartes
```

En cas de besoin, il est possible de retrouver l'index d'un élément par sa valeur :

```
liste1=("Descartes","Erasme","Montaigne","Montesquieu","Diderot","Rousse

liste1.index('Voltaire')

> 6
```

Le dictionnaire

En ce qui concerne le dictionnaire, on accédera à la valeur en nommant la clé :

```
dictionnaire1={'nom':'Descartes','prenom':'René'}
print dictionnaire1['nom']

> Descartes
```

Il est aussi possible d'utiliser

```
dictionnaire1.get('prenom')
```

On voit que dans ce cas, l'avantage du recours au tableau est qu'il évite le recours à la position de l'élément. Ainsi si les éléments ont changé de place suite à une manipulation, l'appel restera correcte grâce à la clé, inchangée.

SÉLECTION PAR VALEURS

Cette méthode est plutôt utilisée dans des tests qui permettent de vérifier la présence d'un élément dans la séquence :

```
nom = "Descartes"
if "cartes" in nom:
    print "Descartes est de la famille des cartomanciens"
else:
    print "Descartes aurait du s'appeler Descartes"
```

Nous aurons bien sûr le même comportement pour une liste véritable :

```
liste1=("Descartes","Erasme","Montaigne","Montesquieu","Diderot","Rousse

if "Voltaire" in liste1:
    print "Voltaire est un philosophe"

> Voltaire est un philosophe
```

17. SÉLECTION MULTIPLE ET LONGUEUR DANS LES STRUCTURES

Dans le cas des chaînes surtout, mais parfois aussi des listes et tuples, il est parfois utile de sélectionner plusieurs éléments. Quand il s'agit de chaînes, cela est évident si on souhaite extraire plusieurs lettres faisant sens ensemble, comme un mot, par exemple. Dans le cas des listes et des tuples, cela se fera surtout sentir dans le cas de réaffectation, ou de suppression.
On pourra connaître la longueur de la structure en ayant recours à len().

```
phrase = "Je pense donc je suis"
len(phrase)

> 21
liste1("René","Descartes")
len(liste1)
> 2
```

Cela fonctionne aussi, de la même façon avec les tuples et les dictionnaires.
Ensuite et dans tous les cas, la longueur d'une sélection sera définie par les indices de début et fin de la sélection, séparé par un : . Ainsi :

```
phrase = "Je pense donc je suis"
print phrase[3:8]

> affiche
> pense
```

N'oubliez pas que les indices commencent à 0 !!
Lorsque l'on veut sélectionner à partir du début on écrira simplement :

```
print phrase[:8]

> Je pense
```

Lorsque l'on souhaite sélectionner les derniers éléments, les choses se compliquent un peu puisqu'il faut connaître la quantité d'élément. Ainsi on pourrait faire :

```
liste4=("René","Descartes","Méditations Métaphysiques")
```

```
longueur = len(liste4)
print liste4[1:longueur]

> ('Descartes', 'Méditations métaphysiques')
```

Remarquez ici 2 choses. Premièrement la spécificité du traitement des accents, et ensuite que le résultat est rendu sous forme de liste.
On pourra abréger en notant :

```
liste4 = ("René","Descartes","Méditations métaphysiques")
print liste4[1:len(liste4)]
```

18. PARCOURIR UNE STRUCTURE DE DONNÉE

L'objectif des structures de données est d'organiser le contenu pour y avoir ensuite accès de façon facilitée, et avec des outils qui permettent de les exploiter simplement selon des critères spécifiques. Il est en particulier régulièrement nécessaire de parcourir les éléments de la structure pour en extraire les valeurs ou faire une recherche.

Ici rien de bien nouveau, nous nous contenterons de montrer comment les boucles déà vues s'adaptent à ces formes nouvelles.

WHILE

```
phrase = "Je pense donc je suis"
i=0
while i < len(phrase):
    print (phrase[i])
    i+=1
```

on utilise ici la valeur incrémentée de i comme indice pour extraire la valeur. On retrouvera le même principe dans :

```
liste4 = ("René","Descartes","Méditations métaphysiques")
i=0
while i < len(liste4):
    print (liste4[i])
    i+=1
```

FOR

Avec for nous pourrons éviter le recours à l'incrémentation d'un chiffre et parcourant la liste en elle-même :

```
for element in liste4:
    print(element)
```

```
> René
> Descartes
> Méditations Métaphysiques
```

Ainsi s'il s'agit de parcourir toutes les données, ce procédé pourra se révéler plus astucieux. While permettra d'avoir un meilleur contrôle des limites mais aussi de faire des boucles non pas élément après élément, mais tous les 2, 3 ou 4... éléments :

```
i=0
while i < len(phrase):
     print(phrase[i])
     i+=2

>
 J

 e
 s

 o
 c
 j

 u
 s
```

RANGE()

Si vous préférez for quoi qu'il arrive, vous pouvez le combiner à range pour obtenir des résultats du même type, voire même parfois plus succint et clair :

```
liste1=("Descartes","Erasme","Montaigne","Montesquieu","Diderot","Rousse

for n in (range(3,6)):
    print liste1[n]

>
Montesquieu
Diderot
Rousseau
```

Range() accepte plusieurs paramètres qui sont interprétés de la façon suivante :

- 1 seul paramètre, range(3) : indique un quantité de 3 ou encore range(len(liste1)) indique le nombre d'élément de liste1 parcouru comme plage
- 2 paramètres, range(3,6) : indique une plage, ici du 4° au sixiéme
- 3 paramètres, range(0,7,2) : indique une plage ainsi qu'un rythme de sélection, ici tous les deux éléments. Nous avons dans ce cas :

```
for n in range(0,7,2):
     print liste1[n]

>
```

```
Descartes
Montaigne
Diderot
Voltaire
```

PARCOURS DE DICTIONNAIRES

La spécificité du parcours d'un dictionnaire tient dans le fait que le dictionnaire est composé de deux informations pour chaque élément : la clé et la valeur. Le parcours du dictionnaire peut se faire avec une boucle for mais si rien n'est spécifié, seul la clé est récupérée. Pour tout récupérer on utilisera une boucle du type :

```
dictionnaire1={'nom':'Descartes','prenom':'René','ecrit':'Meditations
Métaphysiques'}
for cle, valeur in dictionnaire1.items():
    print cle, " ", valeur

>
ecrit   Meditations Métaphysiques
nom   Descartes
prenom   René
```

Les informations sont bien écrites. Remarquez simplement qu'elles sont parcourues à partir de la fin du dictionnaire.
Pour ne récupérer que la cle, utilisez

```
for cle in dictionnaire1.keys()
```

Pour ne récupérer que la valeur, utilisez

```
for valeur in dictionnaire1.values()
```

19. AJOUT ET CONCATÉNATION D'ÉLÉMENTS

Pour ajouter des éléments à un semble structuré, les procédés pourront varier. En ce qui concerne les tuples, rien ne sera possible puisqu'ils sont immuables.
Pour ajouter un élément dans une liste, la méthode employée sera dépendante de la position souhaite. Pour positionner le nouvel élément en fin de liste, on utilisera la méthode append alors que pour insérer à un emplacement particulier, insert sera préféré.

```
liste1=['Rene','Descartes']
liste1.append('Philosophe')
print liste1

> ['Rene', 'Descartes', 'Philosophe']

liste1.insert(2,'France')
print liste1

['Rene', 'Descartes', 'France', 'Philosophe']
```

En ce qui concerne les dictionnaire, il suffira d'affecter une nouvelle clé et une nouvelle valeur en utilisant la clé comme indice :

```
dictionnaire1['pays']="France"
```

ou par la position

```
dictionnaire1[2]="France"
```

Ensuite, il arrive régulièrement qu'il y ait plusieurs éléments à ajouter et que ces éléments proviennent éventuellement d'une autre variable de même type ou non. Si les types sont différents, une boucle pourra permettre l'opération mais dans le cas ou les types sont identiques une simple concaténation va suffire. La concaténation sur une simple opération de mise à la suite de deux éléments. La concaténation a déjà été entrevue avec la fonction print et les chaînes de caractères :

```
citation1 = "je pense"
citation2 = "je suis"
print citation1 + ", donc " + citation2

> je pense, donc je suis
```

Dans le cas des listes, il est possible d'utiliser la même opération :

```
liste1 = ['Rene', 'Descartes', 'France', 'Philosophe']
liste2 = ['Discours de la Methode','Principes de la philosophie']
liste1 + liste2

> ['Rene', 'Descartes', 'France', 'Philosophe', 'Discours de la
Methode', 'Principes de la philosophie']
```

Pour concaténer les deux sans passer par une opération, la méthode extend fera parfaitement l'affaire :

```
liste1 = ['Rene', 'Descartes', 'France', 'Philosophe']
liste2 = ['Discours de la Methode','Principes de la philosophie']
liste1.extend(liste2)
print liste1

> ['Rene', 'Descartes', 'France', 'Philosophe', 'Discours de la
Methode', 'Principes de la philosophie']
```

20. SUPPRESSION D'ÉLÉMENTS

La suppression d'élément n'est pas l'opération la plus courante, l'ajout et l'affectation de nouvelles valeurs seront bien plus utilisées. Il n'est cependant pas inutile de savoir comment faire pour limiter la recherche, et éventuellement accélérer les manipulations, boucles...
En ce qui concerne les listes, la suppression se fera soit par la position de l'élément, son indice :

```
print liste1

> ['Rene', 'Descartes', 'France', 'Philosophe', 'Discours de la
Methode', 'Principes de la philosophie']

del liste1[2]
print liste1

> ['Rene', 'Descartes', 'Philosophe', 'Discours de la Methode',
'Principes de la philosophie']
```

Il est possible de supprimer plusieurs éléments en une seule opération à l'aide 'une plage de sélection :

```
del liste1[2:4]
```

soit par sa valeur avec la méthode remove à laquelle on passera la valeur à supprimer :

```
liste1.remove('Descartes')
print liste1

> ['Rene', 'Philosophe', 'Discours de la Methode', 'Principes de la
philosophie']
```

Dans le cas des dictionnaires, en plus de pouvoir réutiliser del comme pour les listes, on retrouve la sempiternelle attention aux clés et aux valeurs :
clear() videra complètement le dictionnaire
pop(cle) supprime la clé mentionnée
popitem() supprimera la paire clé:valeur

```
dictionnaire1={"nom":"Descartes","Prenom":"Rene","Pays":"France"}
del dictionnaire1['nom']
print dictionnaire1

> {'Prenom': 'Rene', 'Pays': 'France'}
```

Ici l'ensemble cle:valeur est purement et simplement supprimé. Ce ne sera pas tout à fait le cas avec pop qui renverra la valeur associée à la clé au moment de la suppression.

```
dictionnaire1={"nom":"Descartes","Prenom":"Rene","Pays":"France"}
dictionnaire1.pop('nom')

> 'Descartes'

print dictionnaire1
{'Prenom': 'Rene', 'Pays': 'France'}
```

Ces renvois pourront être utile dans certains cas, associés à des fonctions que nous verrons prochainement.
Enfin popitem renvoie la paire cle:valeur sous forme de tuple au moment de la sélection. La paire supprimée est la première du dictionnaire. Cette forme de suppression est surtout utile dans les boucles, évitera les doublons, mais ouvre le risque des pertes.

21. AFFECTATION ET REMPLACEMENT DE VALEUR

L'affectation est une opération souvent effectuée en création. Les modes d'affectation par défaut ont été vus en début de section sur les types structurés. La plupart du temps, il suffira de donner l'indice et de mentionner la nouvelle valeur à lui donner :

```
liste1=['Rene', 'Philosophe', 'Discours de la Methode', 'Principes de
la philosophie']
liste1[2]="Discours de la methode"
print liste1

> ['Rene', 'Philosophe', 'Discours de la methode', 'Principes de la
philosophie']
```

Au sujet des dictionnaire, il suffit de sélectionner par la clé est de modifier la valeur :

```
dictionnaire1={"nom":"Descartes","Prenom":"Rene","Pays":"France"}
dictionnaire1['Pays']='Suede'
print dictionnaire1

> {'nom': 'Descartes', 'Prenom': 'Rene', 'Pays': 'Suede'}
```

Enfin, en ce qui concerne les chaînes, il faudra effectuer un remplacement d'une portion de la chaîne par une autre en réaffectant ou en l'enregistrant dans une nouvelle chaîne :

```
chaine1 = "Bonjour"
chaine1.replace("jour", "soir")
print chaine1

> Bonsoir
```

22. TRIER DES LISTES

Il est parfois important de changer l'ordre des éléments. Cela peut être pratique pour des classements ou mettre en relation deux séquences. sort() va faire ce travail très simplement :

```
liste1 = [5,10,3,7]
liste1.sort()
print liste1

> [3, 5, 7, 10]
```

Cela marche aussi avec les éléments textuels de la liste :

```
liste1 =
["Descartes","Erasme","Montaigne","Montesquieu","Diderot","Rousseau","Vo

liste1.sort()
print liste1

> ['Descartes', 'Diderot', 'Erasme', 'Montaigne', 'Montesquieu',
'Rousseau', 'Voltaire']
```

En revanche, une fois encore mais de façon plus étonnante, cela ne pourra s'appliquer aux tuples, ni aux chaînes d'ailleurs :

```
nom = "cedric"
nom.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  AttributeError: 'str' object has no attribute 'sort'
```

Si la liste contient des éléments de type int ou float et des éléments str, alors les éléments de type str sont placés à la fin alors que la suite est bien ordonnée en ce qui concerne les nombres qu'est-ce que soit leur type.

```
liste1 =
["Descartes","Erasme","Montaigne","Montesquieu","Diderot","Rousseau","Vo

liste2 = [5,10,7,3]
liste1 = liste1+liste2
liste1.sort()
print liste1

> [3, 5, 7, 10, 'Descartes', 'Descartes', 'Diderot', 'Erasme',
'Montaigne', 'Montesquieu', 'Rousseau', 'Voltaire']

liste3 = [1.5, 3.5, 6.2, 10.5]
liste1 = liste1+liste3
```

```
liste1.sort()
print liste1

> [1.5, 3, 3.5, 5, 6.2, 7, 10, 10.5, 'Descartes', 'Descartes',
'Diderot', 'Erasme', 'Montaigne', 'Montesquieu', 'Rousseau', 'Voltaire']
```

Avec les dictionnaires, les choses se gâtent un peu. Que trier ? les clés, les valeurs ? En utilisant la fonction sorted, on obtient in premier résultat.

```
dictionnaire2 = {'nom':'Descartes','prenom':'René','age':'?'}
dictionnaire2

> {'nom': 'Descartes', 'age': '?', 'prenom': 'Ren\xc3\xa9'}

sorted(dictionnaire2)

> ['age', 'nom', 'prenom']
```

J'ai souvent vu des personnes effectuant un tri pour rapprocher les éléments similaires et éventuellement trouver des doublons, comme on le ferait en cliquant sur les colonnes dans un navigateur de fichiers. Pour trouver des éléments similaires, il y a bien plus simple, rapide et fiable count().

```
nom = "cedric"
nom.count('c')
> 2
```

ou sur des listes :

```
liste1=["Descartes","Erasme","Montaigne","Montesquieu","Diderot","Rousse

liste1.count('Descartes')

> 1

liste1.append('Descartes')
liste1.count('Descartes')

> 2
```

On tri aussi souvent connaître la plus petite et la plus grande valeur. Il est alors possible d'utiliser les fonctions de bases min() et max().

```
liste2 = [5,10,7,3]
min(liste2)

> 3

max(liste2)

> 10
```

et même d'en effectuer la somme avec sum() ce qui évite parfois d'avoir recours à des procédés plus longs et complexes :

```
liste2 = [5,10,7,3]
sum(liste2)

> 25
```

min() et max() fonctionnent aussi sur les chaînes et les éléments textuels :

```
nom = "cedric"
min(nom)

> 'c'

max(nom)

> 'r'

liste1=["Descartes","Erasme","Montaigne","Montesquieu","Diderot","Rousse

max(liste1)

> 'Voltaire'
```

MODULARISER

23. REGROUPER DANS DES FONCTIONS

A l'heure actuelle, nos programmes sont composés de petites suite opérations qui s'enchaînent les unes à la suite des autres dans le même ordre. Nous avons vu que pour répéter une opération nous pouvions utiliser des boucles. Mais cela ne change rien au fait que l'enchaînement est parfaitement continu. Or, il est souvent nécessaire de répondre à une action de l'utilisateur effectuée à un moment donnée, parmi d'autres. Chaque action faisant appel à un bloc d'instruction, chaque action pouvant être déclenchée par l'utilisateur à certains moments ou à tous moments, il est important d'avoir une conception des instructions qui ne soient pas complètement linéaire. Pour cela, nous utilisons des fonctions : ce sont des blocs d'instructions auxquels nous avons attribué un nom de façon à pouvoir les appeler, déclencher le moment opportun. Cela prend la forme suivante :

```
def nomdelafonction():
    bloc d'instruction
```

Il s'agit là de la portion de Python qui définit, ou déclare la fonction, un peu comme on déclare une variable. Elle commence par le mot clé def suivi du nom de la fonction (selon les mêmes règles de nommage que les variables environ), suivi de parenthèses qui permettent de la différencier de simples variables (et nous verrons qu'elles pourront être autrement utiles) et d'un : qui mentionne le début d'un bloc. Le bloc qui suit doit être indenté et toutes les lignes qui la composent indentées de façon similaire, comme dans le cas des boucles.
Pour appeler la fonction, nous noterons simplement son nom sous la forme :

```
nomdelafonction()
```

un appel de fonction peut être réalisé à n'importe quel moment du programme, y compris à partir d'une boucle, voire même d'une autre fonction.

24. RENDRE LES FONCTIONS PLUS UNIVERSELLES AVEC DES PARAMÈTRES

L'avantage d'une fonction est de pouvoir être réutilisée dans différents contextes. Mais en imaginant une fonction qui affiche un texte du type :

```
def affiche():
    print "Bonjour"
```

que faire si nous souhaitons aussi afficher "bonsoir" ? Faut-il créer une autre fonction ? et ainsi les cumuler au point de s'y perdre. Il semble évident que non. Python, comme tous les langages modernes, permet l'utilisation de paramètres. Ces paramètres sont des valeurs associées à des variables qui seront transmises directement à la fonction.

Les paramètres sont en général déclarés dans la déclaration de définition, et remplacent les valeurs constantes dans le bloc. On obtient ainsi :

```
def affiche(mot):
    print mot
```

et pour appeler la fonction :

```
affiche("bonjour")
> bonjour
affiche("bonsoir")
> bonsoir
```

Si plusieurs paramètres sont nécessaires, il suffit de les séparer par des virgules dans la déclaration et dans l'appel :

```
def affiche(mot, nom):
    print mot + ', ' +nom

affiche("Bonjour", "Richard")
```

ce qui est encore plus intéressant en jouant avec des valeurs inconnues saisies par l'utilisateur, récupérées dans des sources de données ou autre :

```
def affiche(mot,nom):
```

```
    print mot+', '+nom

print "Quel mot afficher ? "
mot = raw_input()
print "pour qui ?"
nom = raw_input()
affiche(mot,nom)
```

Enregistré dans un fichier et lancé :

```
> Quel mot afficher ?
Bonjour
> pour qui ?
Richard

> Bonjour, Richard
```

Attention cependant c'est l'ordre des paramètres qui compte, pas leur nombre :

```
affiche(nom,mot)
> Richard, Bonjour
```

25. PARAMÈTRES INCONNUS

Lorsque le nombre des paramètres est inconnu, par exemple lorsqu'ils sont issus d'une liste, nous sommes face à un problème : il n'est pas toujours possible de les déclarer ou de les appeler.

De fait, nous avons deux possibilités qui répondent à des besoins différents. Si nous connaissons le nombre de paramètres mais ne savons pas s'ils sera toujours possible de leur affecter une valeur ou si nous ignorons complètement la nature et le nombre des paramètres.

Dans le premier cas nous pourrons attribuer des valeurs par défaut :

```
def (nom, mot="Bonjour"):
    print mot+', '+nom

affiche("Richard")
> Bonjour, Richard

affiche("Richard","Bonsoir")
> Bonsoir, Richard
```

voire de façon plus explicite pour éviter tout risque de confusion en particulier si le nombre de paramètres :

```
affiche("Richard", mot="Bonsoir")
```

Dans le second cas, nous placerons une étoile (liste) ou deux (dictionnaires) devant un paramètre référent :

```
liste1 = [5,10,3,7]
def moyenne(*num):
    print "La moyenne est de "+sum(*num)/len(*num)

moyenne(liste2)
> La moyenne est de 6
```

26. RÉUTILISER LE RÉSULTAT D'UNE FONCTION

Le bloc d'instruction va effectuer une ou plusieurs actions. le résultat des ses actions peut être utilisé seulement en interne ou avoir besoin de resservir dans une autre partie du programme. Imaginons par exemple un jeu dans lequel nous devons mémoriser un score suite à des choix opportuns ou non. Chaque choix déclenche l'ajout ou la suppression de point, et ce total doit être réutilisé dans les actions suivantes. Nous pourrions dans ce cas avoir une fonction par action et une fonction pour le calcul des points. Les actions envoient leur résultat à la fonction calculant les points pour effectuer le calcul adéquat. Le total peut lui-même être utilisé pour donner des bonus ou faire des changements de niveau.

Chaque total peut donc être enregistrer dans une variable globale ou renvoyée à la fonction qui l'appelle afin qu'elle en tienne compte dans le déroulé des actions du jeu.

Un cas simple : écrire un résultat provenant d'un calcul utilisé dans une fonction.
L'exemple de base pourrait être :

```
def calcul(param1):
    param2=10
    print "le résultat est "+ str(param1+param2)

calcul(5)
> le résultat est 15
```

on pourrait remplacer cet exemple par

```
def calcul():
    return param1+param2

print "le résultat est "+ str(calcul(5))
> le résultat est 15
```

L'avantage de cette méthode est d'offrir une meilleure réutilisation de la fonction. dans le premier cas, la fonction calcule et affiche le calcul elle-même ce qui la rend inexploitable dans le cas où le calcul seul doit être effectué.

Dans le second cas, la fonction ne fait que le calcul, charge au contexte de savoir qu'en faire.

27. FACILITER LA RÉUTILISATION AVEC LES CLASSES

De nombreuses personnes utilisent python sans utiliser de classes et pourtant l'un des premier argument que l'on avance en présentant python est son orientation 100% objet. Il pourra peut-être vous sembler étrange de se passer d'une telle fonction du langage si elle est si centrale. Voire même, comment certains font pour s'en passer.

C'est une question que je me pose régulièrement : pourquoi se passer de la programmation objet alors qu'elle est si simple, si pratique et offre finalement et sans effort de nombreuses possibilités.

Moi qui écrit ces lignes et qui ne suis que simple graphiste ou artiste et loin d'être un programmeur hors-pair comme d'autres peuvent s'en vanter, voilà pourtant que, si je devais faire des statistiques de mon code produit, l'utilisation de la programmation objet représenterait au moins 90% des lignes écrites et surtout conservées. J'espère que vous n'en concluerez pas que comme je ne suis pas un expert, je me trompe. Si vous avez des doutes, il vous suffira de faire une petite recherche sur internet et vous vous rendrez rapidement compte que les discours sont en faveur de la programmation objet.

Finalement, le point à retenir est que si je peux le faire, vous le pouvez, qui que vous soyez.
La programmation orientée objet est une forme supplémentaire d'organisation du code produit. Jusqu'ici, nous avons utilisé des blocs d'instructions, puis des fonctions en passant par des variables. Le plus évolué en terme de structuration est sans conteste jusqu'à présent l'utilisation de fonctions. Cependant, nous avons des petits soucis avec ces fonctions. Par exemple, si nous programmons un site internet, nous avons besoin d'une fonction qui vérifie si un utilisateur est connecté et une autre qui liste les fichiers d'un répertoire. A priori, ces deux fonctions n'ont rien à voir ensemble et elles seront cependant accessibles de la même façon, au même niveau. Un peu comme si vous rangiez les torchons avec les serviettes (je sais vous le faites peut-être et moi aussi, mais ce n'est pas une excuse).
En programmation objet, nous allons ajouter au-dessus de la fonction habituelle def un élément qui va permette de classer les fonctions selon leur contexte d'utilisation. Parti de ce principe simple, beaucoup de choses vont alors devenir possibles.

28. DÉFINIR DES CLASSES

Définir une classe se fait simplement en utilisant le mot réservé class suivi du nom personnalisé de la classe en tant que début de bloc d'instruction :

```
class MaClasse :
    #instructions
```

En ce qui concerne le nom de la classe, il est de coutume de respecter toujours la même règle, par simplicité. Nous utilisons ici un nom sans séparation entre les termes et ajout de majuscule au début de chacun d'entre eux.
Une fois la classe créée, il va être possible de l'instancier, c'est-à-dire de l'utiliser dans un cas concret que l'on pourra appeler objet, tout comme chaque table est une instance de ce que peut être une table. L'instanciation se fait simplement en faisant une affectation :

```
objet = MaClasse()
```

Malheureusement, si vous essayez ces quelques lignes dans l'interpréteur python, vous n'obtiendrez pas grand chose :

```
class MaClasse :
     #instructions

File "<stdin>", line 3
          ^
IndentationError: expected an indented block
```

Le message n'est pas très explicite, mais il mentionne bien qu'il manque quelque chose à cette classe : nous devrons donc lui ajouter des méthodes et des attributs.

29. AJOUTER DES ATTRIBUTS

A quoi sert de créer des classes si elles ne contiennent rien ? a priori à rien. La première tâche fréquente dans la définition des classes est souvent la définition de variables, comme pour toute autre structure. On appellera alors la variable "attribut" car elle devient une propriété liée en propre à l'objet et n'existe que dans son contexte.

```
class MaClasse :
     essai = "ok"
```

Notez au passage l'indentation car la classe agit comme un bloc d'instruction structuré. Il faudra donc être vigilant à ce niveau. Essayons maintenant de l'instancier.

```
objet = MaClasse()
print objet.essai
```

affiche

```
ok
```

Remarquez lors de l'instanciation, des parenthèses sont ajoutées au nom de la classe et aussi que pour faire référence à une propriétés, il faut que :

- celle-ci soit définie dans la classe
- l'objet qui la concerne soit instancié
- la rattacher à l'instance en plaçant un point entre les deux.

pour vérifier le type il est toujours possible d'utiliser la méthode type :

```
type(MaClasse)
<type 'classobj'>
```

Ainsi, l'attribut essai se retrouve associée à l'instance et il peut y avoir autant d'attributs du même type qu'il y a d'instance :

```
class MaClasse :
       essai = "ok"

objet = MaClasse()
print objet.essai
> ok

objet2 = MaClasse()
print objet2.essai
```

```
> ok

objet2.essai = "wow"
print objet2.essai,  objet.essai
> wow ok
```

Note : nous utiliserons régulièrement le signe > en début de ligne pour montrer ce que l'interpréteur renvoie. Cette pratique est inverse à l'interpréteur python mais nous permet, nous l'espérons, rendre plus claire les lignes de codes qui nous intéressent.

On voit bien dans cet exemple que la valeur de l'attribut a pu être modifié dans le bloc principal avec une simple affectation et que pourtant, l'affectation ayant été effectuée dans le contexte d'une instance, seule la valeur associée à l'attribut de cette instance a été réellement affectée, l'autre instance n'ayant pas été modifiée le moins du monde.

30. AJOUTER DES MÉTHODES

Il en est des fonctions comme des variables. Dans le cadre d'une classe, elles deviennent des "méthodes". Cette différence de terme peut sembler capricieuse, elle permet cependant de toujours bien savoir à quel niveau on parle.

DES MÉTHODES STANDARDS

Les méthodes vont permettre de définir des regroupements de fonctions propres à notre contexte de manière à mieux les exploiter, en respectant systématiquement ce contexte.

La définition d'une fonction interne à une classe est très simple puisqu'elle ne contient aucune différence avec la façon standard de la définir : on utilise le mot clé def suivi du nom de la fonction, un :, puis à la ligne le bloc d'instruction indenté.

```
class MaClasse:
     def affiche(self):
             essai="ok"
             return essai

objet = MaClasse()
print objet.affiche()
> ok
```

_INIT__ ET SELF, UNE MÉTHODE PARTICULIÈRE

Init est une méthode particulière qui est lancée automatiquement lors de la déclaration d'une instance. On l'appelle "constructeur" parce qu'elle aide à poser les bases de toute l'instance. On y déclare en général les principaux attributs et on peut éventuellement y commencer du traitement par appel de certaines méthodes.

Dans l'exemple qui précéde, l'utilisation du constructeur pourrait conduire à un programme comme celui-ci :

```
class MaClasse:

     def __init__(self):
             self.essai="ok"
```

```
     def affiche(self):
             return self.essai

objet = MaClasse()
print objet.affiche()
> ok
```

Le constructeur contient en général au minimal l'affectation de base des premières variables les plus importantes du programme de manière à s'assurer qu'elles sont bien déclarées et seront parfaitement partagées par la suite au sein des différentes méthodes.

Le mot-clé self permet quant à lui de spécifier que l'action s'effectue bien au niveau de l'instance elle-même. Nous aborderons plus abondamment ce sujet en parlant ultérieurement de leur opposé, les attributs de classe.

Enfin, si __init__ est un constructeur, __del__ est un destructeur. Plus rarement utilisé en python puisque l'interpréteur prend en charge la destruction des objets lorsqu'ils deviennent inutilisé, le destructeur permet cependant de mieux contrôler les événements et de déclencher éventuellement des actions à la destruction d'une instance, comme renvoyer un message, par exemple.

```
class MaClasse:
     def __init__(self):
             self.essai="ok"
     def affiche(self):
             return self.essai
     def __del__(self):
             print "objet détruit"

objet = MaClasse()
print objet.affiche()
> ok
objet.__del__()
> objet détruit
```

Ici, nous forçons la destruction de l'objet pour l'exemple, mais cette action est bien sûr réalisable dans tout projet, même si elle n'est pas nécessaire.

31. MANIPULATIONS D'ATTRIBUTS

MANIPULER LES ATTRIBUTS

Jusqu'ici, nous avons observé que les attributs auront des valeurs redéfinissables pour chaque objet. Mais qu'en est-il si l'on souhaite avoir une valeur partagée entre plusieurs instances ou à l'inverse gagner un peu en définissant spécifiquement les valeurs pour une instance.

ATTRIBUT DE CLASSE

Nous avons manqué un peu de précision en définissant précédemment les attributs. Nous avons présenté notre attribut essai de la même façon alors qu'il est apparu dans deux contextes différents : celui de la racine de la classe, et celui de la méthode. Nous avons mentionné que cet attribut était lié à l'instance, ce qui n'est pas pas tout à fait juste. Prenons le point suivi pour aborder le sujet : si à l'inverse, nous souhaitions que la même valeur soit partagée par plusieurs instances, la syntaxe de l'attribut devrait être un peu différente : on le définira en début de classe en dehors de toute méthode ou on le préfixera avec le nom de la classe de manière que l'attribut y soit explicitement associé, et non pas aux instances :

```
class MaClasse :
      essai = "ok"
      def affiche(self):
                print essai

objet = MaClasse()
objet2 = MaClasse()
objet.affiche()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in affiche
NameError: global name 'essai' is not defined
```

On voit dans cet exemple qu'un petit problème se pose : il n'est pas possible d'accéder à l'attribut essai à partir de la méthode affiche. La première solution qui pourrait nous venir à l'esprit serait d'utiliser le mot clé self pour spécifier l'appartenance de l'attribut, mais voici :

```
class MaClasse :
     essai = "ok"
     def affiche(self):
             print self.essai

objet = MaClasse()
objet.affiche()
> ok

objet2 = MaClasse()
objet2.affiche()
> ok

objet2.essai = "wow"
objet2.affiche()
> wow

objet.affiche()
> ok
```

Nous nous retrouvons alors dans la même configuration que précédemment et ne résolvons pas notre problème : l'attribut est associé à l'instance, et n'est pas partagé entre celle-ci.
La solution consiste à associé explicitement l'attribut à la classe en le pointant comme tel. Au lieu d'utiliser le mot-clé self, nous utiliserons alors le nom de la classe pour bien préciser que c'est à son niveau que les opérations vont s'effectuer.

```
>>> class MaClasse:
     def __init__(self):
             MaClasse.essai = "ok"

objet = MaClasse()
objet2 = MaClasse()
print objet.essai
> ok

print objet2.essai
> ok

print MaClasse.essai
> ok

MaClasse.essai = "wow"
print objet.essai
> wow

print objet2.essai
> wow
```

Ici, les deux instances partagent donc bien le même attribut et changer la valeur de cet attribut en la préfixant du nom de la classe, même au moment de la réaffectation, permet bien de spécifier que ce n'est pas au niveau de l'instance que les choses se jouent et que par conséquent, toutes les instances hériteront de cette nouvelle valeur.
Si les choses ne sont pas clairement définies, il est possible de se retrouver avec de petites surprises. Continuons dans la foulée des lignes précédentes :

```
class MaClasse :
     essai = "ok"
     def affiche(self):
             print self.essai

objet = MaClasse()
objet.affiche()
> ok

objet2 = MaClasse()
objet2.affiche()
> ok

objet2.essai = "wow"
objet2.affiche()
> wow

objet.affiche()
> ok

print objet2.essai
> wow

MaClasse.essai = "test"
objet.affiche()
> test

objet2.affiche()
> wow
```

A interpréter cet exemple, on voit que l'attribut de classe qui a été redéfini dans le cadre d'une instance, ne prend plus en compte l'affectation de valeur faitee à cet attribut de classe.
Ainsi il y a deux essais différents : celui de la classe créé en début de classe ou explicitement et l'attribut de la méthode. La méthode affiche fait référence à son propre essai qui hérite de celui de la classe au moment de la construction de l'instance, mais qui peut être redéfini séparément. Pour faire explicitement référence à l'attribut de classe, nous aurions du être print MaClasse.essai et non pas self.essai.

32. PRIVATISER ATTRIBUTS ET MÉTHODES

Le but de la programmation objet est de mieux structurer le code pour mieux contrôler les contextes d'exécution. En effet, tout programme commence avec quelques petites lignes et finit souvent par plusieurs milliers et il devient alors plus difficile de retrouver ces marques et de bien continuer.
Les défauts de conception n'apparaissent en général pas au début des projets mais bien à la suite, lors des modifications successives, éventuellement réalisées par plusieurs personnes qui ont chacune leur vision des choses.
Différencier les attributs de classe et d'instance est une première approche améliorée, mais il est aussi possible de définir explicitement ce qui peut être de l'extérieur de l'objet. Jusqu'à présent, tout attribut et toute méthode peut être appelé du programme principal. Alors que dans la pratique certains d'entre eux seront certainement créé pour un usage interne en pré-traitement d'informations. Ces attributs n'ont pas besoin, et ne devraient donc pas laisser cette tentation d'être accessible de l'extérieur.
Un petit exemple pour voir de quoi il s'agit :

```
class MaClasse:
    def __init__(self):
            self.__essai="Bonjour "
            self.essai="le monde"
    def affiche(self):
            print self.__essai
    def montre(self):
            print self.__essai

objet=MaClasse()
print objet.essai
```

Le résultat sera bien

```
le monde
```

Maintenant essayons avec

```
print objet.__essai
File "private_public.py", line 15, in <module>
    print objet.__essai
    AttributeError: MaClasse instance has no attribute '__essai'
```

L'interpréteur nous informe que la classe n'a pas d'attribut de ce nom. Ce n'est évidemment pas exactement vrai, mais il garde le secret sur la possibilité de manière à ne pas nous tenter : __essai est privé et ne peut être utilisé que depuis une méthode de la classe :

```
objet.affiche()
> Bonjour
```

Comment avons nous fait pour rendre cette variable privée ? nous avons simplement rajouté deux _ en début de nom. Remarquez au passage que essai et __essai sont considérés comme étant deux attributs différents par l'interpréteur alors que pour le lecteur ils seront évidemment très similaires.
La même opération peut être réalisée avec les méthodes :

```
class MaClasse:
    def __init__(self):
           self.__essai="Bonjour "
    def affiche(self):
           print self.essai
    def __montre(self):
           print self.__essai, self.essai

objet=MaClasse()
objet.__montre()
File "private_public.py", line 15, in <module>

objet.__montre()
AttributeError: MaClasse instance has no attribute '__montre'
```

alors que :

```
class MaClasse:
   def __init__(self):
          self.__essai="Bonjour "
          self.essai="le monde"
   def affiche(self):
          self.__montre()
   def __montre(self):
          print self.__essai, self.essai

objet=MaClasse()
objet.affiche()
```

renverra :

```
Bonjour  le monde
```

Note : En fait, il existe plusieurs notations. On trouvera souvent référence à _variable comme étant une convention d'écriture. Cette convention était utilisée à titre mnémotechnique dans les moments où python ne gérait pas la privatisation des données. Une autre solutions existe aussi : __variable__. Dans ce cas, la variable est absolument privée et rappellera

les méthodes par défaut comme __init__, __del__... En fait, __variable ne produit une variable que partiellement variable qui est bien interdite d'accès hors du contexte mais qui peut toujours être appelée par le biais du chemin complet : instance._classe__variable.

33. PARAMÈTRES D'INITIALISATION D'INSTANCES

Il ne sera pas rare puisque c'est partiellement l'objet de la structuration en objet, d'avoir besoin de créer plusieurs instances de la même classe. Cela est réalisable très simplement, en passant des paramètres à notre constructeur :

```
class MaClasse:
    def __init__(self, nom):
            self.__essai="Bonjour"
            self.essai="le monde,"
            self.nom=nom
    def affiche(self):
            print self.__essai, self.essai, self.nom

objet=MaClasse("Cedric")
objet.affiche()
```

Nous avons déclaré le paramètre nom puis l'avons associé à self.nom pour le réutiliser plus librement.
Le résultat sera donc :

```
Bonjour le monde, Cedric
```

On passera en paramètre de constructeurs les éléments réellement indispensable à l'utilisation et à l'identification de cet objet dans le programme. Pour les paramètres moins fondamentaux, ils pourront être passés à l'appel d'une méthode. Il faudra bien sûr que celle-ci l'accepte au préalable :

```
class MaClasse:
    def __init__(self, nom):
            self.__essai="Bonjour"
            self.essai="le monde,"
            self.nom=nom
    def affiche(self, sexe):
            print self.__essai, self.essai, sexe, self.nom

objet=MaClasse("Cedric")
objet.affiche("Monsieur")

                                        > Bonjour le monde, Monsieur
Cedric
```

34. ETENDRE OU DÉRIVER UNE CLASSE

Un autre cas de réutilisation qui ne sera pas rare, sera souvent d'avoir besoin de créer une variante de l'objet, avec plus de méthodes, des attributs différents ou avec d'autres différences. Dans ce cas, vous avez deux solutions :

- soit utiliser la classe de base telle quelle et ne rien changer (cas lorsqu'on utilise des modules)
- soit modifier la classe d'origine pour définir les nouveaux besoins et les y intégrer
- soit créer une nouvelle classe qui sera basée sur la première mais qui prendra en compte spécifiquement ces différences.

Vous comprendrez à nous lire que cette dernière méthode est la plus fiable : elle permet de préserver une classe originale saine et exempte des cas particuliers qui sont alors simplement définies dans les classes filles.
Ce concept d'héritage est mis en oeuvre simplement
en passant le nom de la classe d'origine en paramètre de la classe fille :

```
class MaClasse:    def affiche(self):
              print self.nom


  objet=MaClasseFille("Cedric")
  objet.affiche()

    def __init__(self, nom):
             self.essai="Bonjour"
             self.nom=nom
    def affiche(self):
             print self.essai, self.nom

class MaClasseFille(MaClasse):
    def affiche(self):
             print self.nom

objet=MaClasseFille("Cedric")
objet.affiche()
```

ce qui affiche

```
Bonjour Cedric
```

La classe file peut donc posséder ses propres méthodes voire redéfinir les méthodes de sa classe mère :

```
class MaClasse:
    def __init__(self, nom):
            self.essai="Bonjour"
            self.nom=nom
    def affiche(self):
            print self.essai, self.nom

class MaClasseFille(MaClasse):
    def affiche(self):
            print self.nom

objet=MaClasseFille("Cedric")
objet.affiche()
```

Avec cet exemple, la classe fille n'affichera que le nom passé en paramètre lors de l'instanciation alors que dans la classe mère, le nom était concaténé à "Bonjour". Il sera ainsi possible de dériver énormément de chose sans toucher à l'original et ainsi diminuer le risque d'introduction de bugs dans d'autres applications qui utiliseraient aussi cette classe. Le code devient alors réellement plus simple à maintenir et chaque programme devient plus sûr. Les efforts peuvent être mutualisés sur plusieurs classes, voire plusieurs applications car toute modification effectuée sur la classe mère sera automatiquement reportées sur les classes filles qui en hérite, ce qui est fondamental lors d'un débogage.

35. LES MODULES : PARTAGER DES FONCTIONS ET OBJETS ENTRE PROGRAMMES

Un module est une classe ou ensemble de classe qui sera spécifiquement utilisé comme ressources dans d'autres programmes. Il a en général été conçu comme tel et s'avère très pratique lors de tâches répétitives.
La règle veut que les modules portent le même nom que la classe qu'ils contiennent. Ainsi MaClasse.py serait le nom de module courant pour le fichier contenant la classe MaClasse. Bien sûr, le module pouvant en contenir plusieurs, cette règle est adaptable.
Au niveau du fichier contenant les classes il est fréquent d'ajouter une condition de test d'exécution (en général en fin de fichier) :

```
if __name__ == "__main__"
    pass
    # autres instructions au lieu de pass
```

Le but de cette condition est de définir ce qui doit être exécuté si le module est exécuté comme corps principal du programme, sinon, cette partie est simplement ignorée.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

class MaClasse:
    def __init__(self, nom):
            self.essai="Bonjour"
            self.nom=nom
    def affiche(self):
            print self.essai, self.nom

class MaClasseFille(MaClasse):
    def affiche(self):
            print self.nom

if __name__ == "__main__":
    pass
```

Enregistrez ce fichier avec l'extension .py

UTILISATION DU MODULE

Une fois votre module créé, vous pourrez alors l'importer pour le réutiliser dans tous vos projets :

```
import MaClasse
objet = MaClasse.MaClasseFille("cedric")
objet.affiche()
```

Commentons un peu ce qui se passe ici :

- on commence par importer le module on utilisant son nom, sans l'extension. Par défaut, python cherche dans le PYTHONPATH et dans le répertoire courants
- on instancie les objets en utilisant le nom du module puis le nom de la classe (ce qui n'était pas nécessaire jusqu'ici
- enfin, l'instance peut alors jouer son rôle normalement.

La méthode import va charger l'ensemble du fichier en vue de son exploitation ultérieure.

```
from param_classe import MaClasseFille
objet = MaClasseFille("Cedric")
objet.affiche()
> Cedric
```

Si le gain ici n'est pas très important à cause de la simplicité de notre fichier module, remarquez qu'avec cette façon de faire, il n'est pas nécessaire nommer le module lors de l'instanciation. Si cela a l'avantage de faire saisir un peu moins de texte, si vous devez charger plusieurs modules, il faudra être certain qu'il n'y aura pas de conflit dans le nom des classes et méthodes utilisées, problème qui est résolu avec l'utilisation d'import seul et la préservation de son espace de nom.

ALLER PLUS LOIN AVEC PYTHON

36. TEST DE RÉUSSITE D'UNE FONCTION (TRY...ELSE)
37. GESTION DES FICHIERS ET CHEMIN AVEC OS
38. MANIPULATION AVANCÉE DE TEXTE AVEC RE
39. FACILITER LE DÉBOGAGE DES PROGRAMMES AVEC LES TESTS

36. TEST DE RÉUSSITE D'UNE FONCTION (TRY...ELSE)

37. GESTION DES FICHIERS ET CHEMIN AVEC OS

38. MANIPULATION AVANCÉE DE TEXTE AVEC RE

39. FACILITER LE DÉBOGAGE DES PROGRAMMES AVEC LES TESTS