Rappels sur le langage Python

Sommaire

- Introduction au langage de programmation Python
- Principalement, les aspects procéduraux du langage
- Très brève allusion aux aspects orienté objet du langage

Ce chapitre a pour mission d'introduire le langage de programmation Python, langage qui sous-tend tout le projet Django. Les utilisateurs de Django doivent posséder une maîtrise minimale du langage pour la programmation, pour l'essentiel des aspects « contrôle » de leur projet. Les règles syntaxiques brièvement décrites ici doivent suffire à la compréhension des différentes applications de Django qui suivent dans les chapitres.

Pourquoi Python

Il n'est pas question dans ce chapitre de faire de vous, lecteurs, des experts en programmation Python. Les quelques pages qui vont suivre ni suffiraient en rien. Il existe de multiples ouvrages enseignant la programmation à différents niveaux et dans les nombreux langages de programmation qui existent à ce jour. Ne sera couvert ici que le BABA de la programmation: les variables, les typages de base, les instructions d'affectation et de contrôle, le découpage des programmes par l'utilisation de fonctions et une rapide introduction aux aspects orienté objet. Le but est de vous permettre de comprendre tous les développements et utilisations de Django basés sur le langage Python qui vont suivre.

En tant que langage de programmation, Python présente énormément de qualité liées pour l'essentiel à sa facilité d'accès, de téléchargement, de mise en œuvre, l'aisance de sa syntaxe, son côté « open source », le fait qu'il soit interprété plutôt que compilé puis exécuté, rendant possible de voir directement les résultats des instructions s'afficher sans étape préalable de

compilation. Python est un langage ayant visé dès son origine une grande simplicité d'écriture, tout en conservant tous les mécanismes de programmation objet de haut niveau. Il cherche à soulager au maximum le programmeur de problèmes syntaxiques non essentiels aux fonctionnalités clés du programme. Les informaticiens parlent souvent à son compte d'un excellent langage de prototypage qu'il faut remplacer par un langage plus « solide », plus « robuste », tel Java, les langages .Net ou C++, lorsqu'on arrive aux termes de l'application. Autre avantage certain, tout ce qui est nécessaire à l'utilisation du langage et à sa compréhension se trouve réuni sur un site unique: www.python.org. Par sa simplicité d'usage, Python, à la différence d'autres langages tels Java ou C++, est un choix qui s'impose assez naturellement, par exemple, lorsqu'on démarre la programmation. De nombreuses universités font ce choix ces jours-ci, rangeant les manuels de Java juste à côté de ceux anciennement entreposé de C++, dans les rayonnages poussiéreux de bibliothèques où plus personne ne va jamais.

APARTÉ Python et son papa Guido Van Rossum

Python est très souvent plébiscité pour la simplicité de son fonctionnement, ce qui en fait un langage de choix pour l'apprentissage de la programmation. La programmation est, de fait, très interactive (vous tapez « 1+1 » à l'écran et comme par magie « 2 » apparaît). On arrive par des écritures plus simples et plus intuitives, donc plus rapidement, au résultat escompté. Si, nous l'avons déjà dit, `print ("Hello World")` est incontestablement plus simple à écrire que `public static void main (String[] args) {System.out.println "Hello World"}`, en Java, nous restons pourtant un peu sceptiques quant à l'extension de cette même simplicité à la totalité de la syntaxe. Python reste un langage puissant car il est à la fois OO et procédural. Pour l'essentiel, il n'a rien à envier à des langages comme Java ou C++ et exige donc de maîtriser, comme pour ceux-ci, toutes les bases logiques de la programmation afin de parvenir à des codes d'une certaine sophistication. D'où notre réserve. Il est incontestable que sa popularité n'a cessé de croître au cours des ans, et que de nombreuses solutions logicielles dans le monde Linux ou Microsoft y ont de plus en plus souvent recours. Un Python.Net est sur le point d'éclore.

Ce langage de programmation veut préserver la simplicité qui caractérise les langages de script interprétés plutôt que compilés (les exécutions sont traduites dans un bytecode intermédiaire et s'exécutent au fur et à mesure qu'elles sont rencontrées) grâce à son aspect multiplateformes, à l'absence de typage statique ou à la présence de structures de données très simples d'emploi (listes, dictionnaires et chaînes de caractères). Il souhaite en outre préserver toutes les fonctionnalités qui caractérisent les langages puissants (OO, ramasse-miettes, héritage multiple et librairies d'utilitaires très fournies, comme nous le verrons dans les chapitres qui suivent).

Une telle hybridation, pour autant qu'elle soit réussie, en ferait un langage de tout premier choix pour le prototypage de projet, quitte à revenir par la suite à des langages plus robustes pour la phase finale du projet (comme C++ et Java, avec lesquels, d'ailleurs, Python se couple parfaitement : il peut hériter ou spécialiser des classes Java ou C++). Une telle hybridation est-elle possible ? En quoi cette motivation serait-elle innovante par rapport à celle qui a présidé à la création de Java et C# ? Ce mélange idéal entre puissance fonctionnelle et simplicité d'usage n'est-il pas le Graal dont tous les langages de programmation d'aujourd'hui sont en quête ? Qui sera la suivant sur la liste ?

Python, écrit en C et exécutable sur toutes les plates-formes, est développé par les Python Labs de Zope Corporation qui comprennent une demi-douzaine de développeurs. Ce noyau est dirigé par Guido Van Rossum, inventeur et « superviseur » du langage, qui se targue de porter le titre très ambigu de Dictateur Bénévole à Vie. Toute proposition de modification du langage est débattue par le noyau et soumis à la communauté Python, mais la prise en compte de celle-ci dans le langage reste la prérogative de Guido, qui conserve le dernier mot (d'où son titre : gentil et à l'écoute… mais dictateur tout de même, ces modifications n'étant pas soumises à un vote majoritaire). Guido est hollandais d'origine, détenteur de maîtrises en mathématiques et en informatique de l'université libre d'Amsterdam. Il participa, à la fin des années 1980, à un groupe de développeurs dont le but était la mise au point d'un langage de programmation abordable par des non-experts, d'où son nom « ABC ». Dès 1991, il s'attaque à Python.

> (Ce nom ne doit rien à l'horrible reptile mais se réfère aux extraordinaires humoristes anglais que sont les Monthy Python, qui ont révolutionné dans les années 1970 et 1980 et l'humour et la télévision, et dont l'humour nous manque tellement aujourd'hui, d'où la nécessité de les remplacer par un jumeau ouvert sur le Web et la programmation.) Guido Van Rossum travaille alors au CWI (Centrum voor Wiskunde en Informatica). En 1995, il prend la direction des États-Unis, comme tout bon informaticien qui se respecte et qui veut percer, et travaille pour le CNRI (Corporation for National Research Initiatives) jusqu'en 2000. À cette époque, il publie Computer Programming for Everybody, sa profession de foi pour l'enseignement de la programmation. C'est également à cette époque qu'il est nommé directeur des Python Labs de Zope Corporation. Depuis 2005, il travaille pour Google, entreprise qu'il n'est plus vraiment besoin de présenter, et qui semble investir beaucoup dans le langage Python. Il y divise son temps de travail entre le projet Open Source Python et les développements de Google utilisant Python.
>
> La communauté Python reste très structurée autour d'un seul site Web : http://www.python.org, ce qui s'avère un atout considérable, surtout lors de l'apprentissage et de la découverte du langage. À quelques subtilités près, Python est dans le prolongement de l'aventure Open Source, dont le représentant le plus emblématique reste Linux. Il pourrait devenir le langage de programmation phare de l'Open Source, tout comme Linux est celui de l'OS. Toutes les sources restent accessibles et disponibles, monsieur-tout-le-monde peut participer à l'évolution du produit, mais la prise en charge officielle de ces évolutions reste sous la responsabilité de quelques-uns (ici, un seul). Il semble que les éditeurs informatiques, tant Sun que Microsoft, aient décidé d'évoluer pour leur tendre bébé, vers ce même modèle de développement. Java est en effet devenu Open Source et le Projet Mono poursuit en Open Source l'aventure .Net.

Revers de la médaille, beaucoup de critiques du langage utiliseront (c'est d'un attendu !) la simplicité de sa syntaxe, l'absence de typage explicite et de compilation, et les retournerons contre le langage (le Python se mangeant la queue), en prétextant de la difficulté d'aborder des projets complexes dans un idiome aussi primitif (comment écrire Hamlet en Esperanto). Tout langage de programmation est toujours à la recherche d'un graal sans doute introuvable (vu le nombre et la diversité des programmeurs) au carrefour de l'efficience machine, de la simplicité d'usage et d'écriture ainsi que de la robustesse. Il en va des guerres de langages comme de celles des religions : il est plutôt difficile de rationaliser ses préférences. Python est une étape non négligeable dans cette quête céleste jamais aboutie. Nous ne nous positionnerons pas dans ce débat. Ce langage nous est de toute façon imposé par Django, comme l'est le Java par la technologie JSP.

Le Python que nous abordons ici est la version 2.7 car c'est la seule que Django tolère à ce jour. Si le langage en est pour l'instant dans sa troisième version, l'introduction que nous en faisons ne devrait pas souffrir de l'éventuelle migration que vous seriez amenés à faire dans les années à venir (lorsque Django s'adaptera aux nouvelles versions de Python).

Les bases de la programmation Python

Variables et mémoire centrale

Une variable est un nom qui permet de repérer un emplacement donné précis de la mémoire centrale: l'adresse de cet emplacement. La manière la plus simple de comprendre une variable est de l'assimiler à une boîte aux lettres qui porterait et serait accessible par le nom de cette

variable. Elle fait donc le lien entre le nom de la variable que l'on manipule dans le programme et une adresse où est physiquement sauvegardée notre donnée. Cette notion, simple en apparence, facilite donc grandement la réalisation des programmes. Les données manipulées par un programme sont stockées le temps de son exécution en mémoire centrale. Les variables permettent ainsi de manipuler ces données sans avoir à se préoccuper de l'adresse explicite qu'elles occuperont effectivement en mémoire. Pour cela, il suffit de leur choisir un nom. Bien entendu, ceci n'est possible que parce qu'il existe un programme de traduction (l'interpréteur) du programme qui, à l'insu du programmeur mais pour son plus grand confort, s'occupe d'attribuer une adresse à chaque variable et de superviser tous les changements d'adresse qui pourraient s'en suivre.

Par exemple, a=5 signifie « à l'adresse mémoire référencée par "a" (dans la boîte aux lettres dénommée a), se trouve la valeur 5 » ; mais nous utilisons plus fréquemment l'abus de langage « la variable a vaut 5 ». Etant donné la liberté autorisée dans le choix du nom des variables et pour de simples raisons mnémotechniques, il est préférable, afin d'également améliorer la lisibilité des programmes, de choisir des noms de variables en fonction de ce qu'elles représentent réellement. Ainsi, préférerez par exemple des dénominations telles « montant », « cote », « tailleDesSeins » ou « prix » à x, y et z.

En Python, la simple déclaration d'une variable ne suffit pas à la créer. Après avoir choisi son nom, il est nécessaire d'affecter une valeur initiale à cette variable. Nous aurons donc par exemple pour les trois possibles variables qui suivent :

Déclaration et initialisation de variables

```
monChiffre=5
petitePhrase= " Quoi de neuf ? "
pi=3.14159
```

Comme le montre cet exemple, différents type d'information (nombres, chaînes de caractères…) peuvent être placés dans une boîte aux lettres, et il est capital de connaître comment la valeur qui s'y trouve a été codée. Cette distinction correspond à la notion de type, fondamentale en informatique car elle débouche, in fine, sur le codage binaire de la variable (par exemple, un entier numérique sur 32 bits et un caractère sur 16 bits) ainsi que sur les usages qui peuvent en être fait (on ne peut pas multiplier un caractère par deux et avoir pour prénom « R2D2 » sauf dans les films de science-fiction avec Robots). Ainsi, nous dirons qu'une variable recevant une valeur numérique entière est du type « int », une variable recevant un réel sera du type « float » et une variable recevant un ou un ensemble de littéraux sera du type « string ». Il s'agit là des trois types dits simples qui nous occuperont pour l'essentiel dans la suite.

L'instruction d'affectation réalise donc la création et la mémorisation de la variable, l'attribution d'un type à la variable, la création et la mémorisation d'une valeur et finalement l'établissement d'un lien entre le nom de la variable et l'emplacement mémoire. Ainsi, les deux instructions suivantes :

Modifier une variable

```
maVariable=5
maVariable = 2.567
```

permettent de simplement modifier le contenu de la boîte aux lettres `maVariable`. À la première ligne, on indique qu'à l'adresse `maVariable` est stockée la valeur 5 et, à la seconde, on remplace la valeur stockée à cet emplacement par 2,567. Après exécution des deux lignes, la valeur sauvegardée en mémoire à l'adresse `maVariable` est donc 2,567. Notons au passage que le type de `maVariable` a changé : il est passé d'`int` à `float`. Cette spécificité est une facilité essentielle de plus permise par le langage Python. A la différence d'autres langages de programmation plus contraignants, Python est dit typé dynamiquement et non statiquement. Cela signifie qu'au cours de l'écriture du code, il n'est plus nécessaire de préciser à l'avance le type des informations exploitées dans le programme, celui-ci devenant « implicite ». Python devine le type par la valeur que nous installons dans la boîte aux lettres – et ce type peut changer au fil des valeurs que la boîte va contenir. Le choix entre typage statique et dynamique est de ces controverses qui ont amusé, distrait même déchiré la communauté informatique depuis la nuit des temps (les années 60 en informatique).

Nous pouvons également prendre le contenu de la boîte aux lettres `maVariable` et le copier dans la boîte aux lettres `taVariable` par la simple instruction:

taVariable=maVariable

Le contenu de `maVariable` reste inchangé car il s'agit d'un recopiage et non pas d'un transport physique à proprement parler (ici la métaphore de la boîte aux lettres montre ses limites). En fait, il s'agit d'une duplication de l'information d'un lieu de la mémoire vers un autre. Si l'on comprend que la valeur affectée à une variable peut évoluer dans le temps, on comprend tout autant qu'il est possible de réaliser toutes sortes d'opérations sur ces variables, autorisées par leur type. Nous détaillerons pratiquement certaines de ces opérations par la suite. Nous allons voir qu'outre l'abstraction offerte par les variables, le langage nous offre toute une librairie d'utilitaires. Citons d'ores et déjà deux fonctions prédéfinies : type (`nomDeLaVariable`) qui affiche le type de la variable mise entre parenthèses et s'avère très utile pour vérifier ce qui y a été installé jusqu'ici, et `print nomDeLaVariable` qui affiche à l'écran la valeur stockée à l'adresse désignée par la variable `nomDeLaVariable`.

Les types simples

Le type int

Le code suivant illustre des opérations réalisées dans la console interactive de Python, sur des variables de type entier (int) (les lignes débutant par « # » sont des commentaires en Python – donc non exécutés - et seront utilisées ici pour éclairer le comportement des instructions). Les « >>> » que vous verrez apparaître par la suite sont produits par l'environnement de développement et sont une « invite » à taper vos instructions Python.

Opérations sur des variables de type int

```
>>> maVariable = 3
>>> print maVariable
3
>>> type(maVariable)
<type 'int'>
>>> taVariable=4
>>> maVariable=taVariable
>>> print maVariable
4
>>> maVariable = saVariable
#Une erreur apparaît car “saVariable” n’existe pas
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    maVariable = saVariable
NameError: name 'saVariable' is not defined
>>> maVariable = maVariable + 3
>>> maVariable
7
>>> maVariable = maVariable/2
>>> maVariable
3
#Python ne conserve que la valeur entière au vu du type de «maVariable »
>>> maVariable + 3 = 5
SyntaxError: can't assign to operator
#On ne peut pas écrire n’importe quoi et il faut respecter un minimum de
#syntaxe python.
>>> maVariable+=4 #Raccourci d’écriture !!!
>>> maVariable
7
>>>
```

En plus de ce que nous avions déjà dit sur l’affectation de variables, il est possible de réaliser des opérations mathématiques sur les valeurs contenues dans les variables. Nous avons vu dans le chapitre précédent que les opérations mathématiques font partie des premières instructions élémentaires comprises par tout processeur. Ainsi, `maVariable=taVariable+1` signifie que la valeur contenue dans la variable `maVariable` va prendre celle contenue dans `taVariable` augmentée de 1 (ici on est plutôt en présence de deux classes d'instruction élémentaire: une addition et un déplacement de valeur d'une variable à l'autre). Nous pouvons aussi constater que l’expression `maVariable=maVariable+6` signifie que l’on écrase la valeur de `maVariable` par sa propre valeur augmentée de 6. Autrement dit, on peut comprendre cette expression en disant: « la nouvelle valeur de `maVariable` est égale a l’ancienne augmentée de 6 ». On peut d’ailleurs en raccourcir l’écriture : `maVariable+=6`.

Au rayon des erreurs, on peut d’ores et déjà en répertorier deux. La première consiste à effectuer des opérations sur une variable qui n’a pas été initialisée au préalable. Dans l’exemple, on ne peut en effet assigner la valeur de `saVariable` à `maVariable` pour la bonne et simple raison que l’on n’a pas assigné avant cela de valeur à `saVariable`. La dernière ligne, quant à elle, est une simple erreur de syntaxe, montrant que la liberté d’écriture n’est pas totale. Il y a certaines règles de syntaxe et conventions à respecter et nous en découvrirons d’autres par la suite. Pour

les langages de programmation de type "compilé", c'est le compilateur qui s'occupe en général de repérer ces erreurs de syntaxe, alors que, dans Python, tout se passe à l'exécution. De nombreux informaticiens jugent cela un peu tard pour s'apercevoir d'une erreur, d'où la préférence qu'ils expriment pour les autres langages requérant l'étape de compilation (et le typage statique que cette compilation requiert). C'est en cela qu'ils jugent ces concurrents plus robustes : de nombreuses « idioties » sont détectées dès la compilation, vous évitant l'humiliation en public d'un plantage lors de l'exécution de votre programme.

Le type float

Le code suivant illustre des opérations élémentaires sur des réels.

Opérations sur des variables de type float

```
>>> taVariable=3.0
>>> maVariable=taVariable/2
>>> maVariable
1.5
>>> type(maVariable)
<type 'float'>
>>> print maVariable
1.5
>>> print mavariable

Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    print mavariable
NameError: name 'mavariable' is not defined
```

Côté erreur, on peut rajouter que Python est sensible au majuscule et minuscule, `maVariable` et `mavariable` devenant deux variables différentes. On peut également remarquer qu'une opération de division non entière sur un entier naturel donne lieu à une troncature du résultat, alors que la même opération réalisée sur un `float` (un réel) donne lieu au résultat correct. Retenons donc que 3 est `int` alors que 3.0 est `float`.

Le type string

Pour en terminer avec ces types simples, le code suivant illustre des exemples concernant des variables de type `string` (des chaînes de caractères).

Opérations sur des variables de type float

```
>>> phrase1="le framework"
>>> phrase2=" Django"
>>> phrase1+=phrase2
>>> phrase1
'le framework Django'
>>> phrase2*=3
>>> phrase2
' Django Django Django'
>>> print phrase2[1] #On saute le caractère blanc du début de ligne
```

```
D
>>> phrase2 += 5

Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    phrase2 += 5
TypeError: cannot concatenate 'str' and 'int' objects
>>> phrase2="phrase1"
>>> phrase2
'phrase1'
```

On constate ici que des opérations sur les littéraux sont possibles : le + en présence de deux strings est réinterprété et permet de concaténer ces strings entre eux, alors que le * sert à répéter un littéral. Remarquons également que la chaîne de caractères se présente comme une collection de caractères indexés ; le premier élément de la chaîne étant repéré par la valeur 0, le second la valeur 1, etc. Dans la plupart des langages de programmation, le premier élément d'une collection est toujours indexé par 0, ce qui entraîne le dernier à se trouver indexé par la longueur de la collection moins un. Commencer un tableau à « un » plutôt qu'à « zéro » est une source d'erreurs classique en programmation, et qui fait beaucoup rire dans les chaumières d'informaticiens. Enfin, remarquons que, nonobstant l'absence de typage explicite, il est interdit de concaténer deux éléments de type différents comme par exemple un string et un nombre naturel. A nouveau, l'erreur se produira à l'exécution vu l'absence de typage statique et d'une étape préalable de compilation. Le compilateur, le chien de garde des programmeurs aurait pu vous éviter tel affront devant vos pairs. Les programmeurs Python, tout aventuriers qu'ils sont, acceptent cette prise de risque et se satisfont de déférer cette responsabilité à la phase d'exécution. Les goûts et les couleurs …

Une chaîne de caractère est distinguable d'autre type de données par la présence des apostrophes (Python ne fait pas la différence entre simple et double apostrophe). En l'absence de celles-ci, la signification de l'expression peut être toute autre comme en témoigne les deux dernières expressions. Dans le premier cas, c'est la variable phrase1 qui est prise en compte, alors que dans le deuxième, c'est le mot « phrase1 ».

Les types composites

Nous avons jusqu'ici vu des données de type simple à savoir les `int`, les `float` et les `strings`. En marge de ces types dits simples, se trouvent les types composites. Il s'agit de types qui regroupent en leur sein un ensemble d'entités de type simple. En clair, il s'agit de collection de variable des trois types vus jusqu'ici. Nous en avons déjà vu un cas particulier. En effet, les chaînes de caractères regroupent un ensemble de caractères, chacun de type string. Ici toutes les entités sont du même type. Les autres données composites qui nous intéresseront par la suite sont les listes et les dictionnaires. L'utilité de ces collections est de permettre un même traitement répété sur l'ensemble des éléments composant la collection. Par exemple, si l'on souhaite mettre toutes les lettres d'un mot en majuscule, il serait stupide de réécrire cette opération autant de fois qu'il n'y a de lettres dans le mot. Autant ne l'écrire qu'une fois, et placer

cette opération dans une boucle qui balaiera tout le mot. Collections et instructions de boucle font souvent bon ménage comme nous allons le voir.

Les listes

Comme nous l'avons vu dans le cas des chaînes de caractères, chaque élément de la chaîne est indexé par une valeur entière numérotée à partir de 0. Les listes n'en deviennent donc qu'une généralisation à d'autres types simples des chaînes de caractères. Le code suivant clarifie leur utilisation.

Opérations sur des listes

```
>>> liste=["journal",9,2.7134,"pi"]
>>> liste[0]
'journal'
>>> print liste[3]
pi
>>> type(liste[1])
<type 'int'>
>>> liste[4]

Traceback (most recent call last):
  File "<pyshell#59>", line 1, in <module>
    liste[4]
IndexError: list index out of range
#Erreur car on a dépassé la capacité de la liste !!!
>>> liste.append("bazar")
>>> liste
['journal', 9, 2.7134, 'pi', 'bazar']
>>> liste.insert(4,"truc")
>>> liste
['journal', 9, 2.7134, 'pi', 'truc', 'bazar']
>>> len(liste) #donne la longueur de la liste
6
```

On voit clairement qu'une liste est une simple série d'entités de type simple, chacune étant accessible par l'intermédiaire de son index: 0,1,2… Attention à ne pas dépasser la longueur initiale de la liste, déterminée lors de son initialisation. Pour ajouter un élément en fin de liste, il suffit d'utiliser l'instruction `append`, alors que pour insérer un nouvel élément dans la liste en une position précise, il faut utiliser l'instruction `insert`.

Les indexations des éléments de la liste peuvent également s'effectuer en se référant à l'exemple suivant :

Opérations sur des listes

```
>>> monTexte = "hello" + " world"
>>> monTexte[2]
'l'
>>> monTexte[0:2]
'he'
>>> monTexte[:4]
'hell'
```

```
>>> monTexte[6:]
'world'
>>> monTexte[-1]
'd'
```

Les dictionnaires

Les dictionnaires, derniers types composites, sont quant à eux une généralisation des listes. Chaque élément est indexé non plus pas sa position dans la liste, mais par un élément choisi parmi les trois types simples. Il s'agit donc d'une sorte de matrice 2*n d'éléments simples dont ceux de la première colonne réalisent l'index ou la clé (key) d'accès au second.

L'exemple traité dans le code clarifiera les idées.

Opérations sur des dictionnaires

```
>>> dico={"computer":"ordinateur"}
>>> dico["mouse"]="souris"
>>> dico["mouse"]
'souris'
>>> dico[2]="two"
>>> dico
{2: 'two', 'computer': 'ordinateur', 'mouse': 'souris'}
>>> dico.keys()
[2, 'computer', 'mouse']
>>> del dico[2]
>>> dico
{'computer': 'ordinateur', 'mouse': 'souris'}
```

On peut ainsi remarquer que, pareillement aux listes, les dictionnaires sont un type dynamique. On peut rajouter dynamiquement, c'est-à-dire même après une première initialisation, des éléments au dictionnaire. Ainsi, après la création de celui-ci, nous l'avons enrichi de "souris" et "two" indexé respectivement par "mouse" et 2. Les éléments de la première colonne servent à accéder à leur pendant dans la seconde. L'ordre d'apparition des éléments n'a plus aucune importance dans les dictionnaires car l'indexation de chaque élément ne se fait plus par sa position.

Les instructions de contrôle

Jusqu'ici, nous avons a vu comment l'interpréteur Python parcourait le code de manière séquentielle. Ainsi, lorsqu'une séquence d'inscriptions est lue, l'interpréteur les exécute les unes à la suite des autres dans l'ordre où elles se trouvent écrites et où il les découvre. Cette exécution systématique présente des limitations dans de nombreux cas. Des instructions de rupture de séquence apparaissent donc nécessaires. Certaines instructions élémentaires des processeurs ont la charge de modifier la séquence naturelle d'exécution du programme en bifurquant vers une nouvelle instruction qui n'est plus la suivante. En substance, il existe deux types d'instruction qui permettent de rompre avec la séquence en cours : la forme conditionnelle et la forme répétitive.

Les instructions conditionnelles

Ce type d'instruction permet au code de suivre différents chemins suivant les circonstances. Il s'agit, en quelque sorte, d'un aiguillage dans le programme. Son fonctionnement est le suivant. Cet aiguillage teste une condition et décide en fonction de la validité ou non de celle-ci, du chemin à suivre dans le programme. La syntaxe est comme suit :

Syntaxe du « if »

```
if condition:
  bloc1
else:
  bloc2
bloc3
```

Ainsi, le **bloc1**, un ensemble d'instructions, ne sera exécuté que si la condition est vérifiée. Le résultat de l'évaluation de condition est donc vrai ou faux – on dit de cette expression qu'elle possède une valeur booléenne. Si le résultat est faux, c'est le **bloc2** qui sera exécuté. Dans un cas comme dans l'autre, après son exécution, c'est le **bloc3** qui sera entrepris.

Remarquons la présence essentielle des deux points et du retrait. Les deux points marquent la fin de la condition. Tout bloc d'instructions devant s'exécuter, en effet, « comme un seul bloc », doit faire l'objet d'un léger décalage mais indispensable vers la droite. On dit du code décalé vers la droite qu'il est « indenté ». En l'absence de cette indentation, l'interpréteur pourrait donner une erreur, car tout ce qui doit se réaliser, si une condition ou l'autre est vérifiée, doit respecter la même indentation.

L'indentation devient donc fondamentale ici. Elle fait intégralement partie de la syntaxe de Python. Ce qui était une bonne pratique d'écriture de code, ayant pour finalité d'en aérer la présentation, et généralement suivie par la majorité des programmeurs, tous langages confondus, s'est trouvée transformée en une règle d'écriture stricte.

Selon l'indentation choisie pour une instruction quelconque, le résultat à l'exécution pourrait s'avérer très différent, comme l'exemple de code suivant l'illustre quant à la position relative de l'instruction `print c'est faux`. Cette instruction ne s'exécutera pas dans le premier cas mais bien dans le deuxième, car elle n'est plus conditionnée par la deuxième condition portant sur le « a » (mais elle le reste par la première).

Exemple de « if »

```
>>> a=5
>>> if a<10:
      print "c'est vrai"
      if a>10:
         a=5
         print "c'est faux"


c'est vrai
>>> if a<10:
      print "c'est vrai"
```

```
      if a>10:
         a=5
      print "c'est faux"


c'est vrai
c'est faux
```

`Else` signifie « autre » en anglais et effectivement le `else` représente tous les cas non couverts par la condition du `if`. On dit aussi du `else` qu'il reprend toute la partie « autrement », car c'est là qu'on aboutit si aucune condition n'aura été validée avant cette partie. Le code suivant clarifiera son utilisation.

Exemple de « if »

```
>>> a=10
>>> if a<10:
      print "a inferieur a 10"
else:
      print "a superieur ou egal a 10"


a superieur ou egal a 10
```

Notez, là encore, la présence des deux points terminant le « else » et l'indentation. En théorie, et en l'absence des invites « >>> », le « else » se trouve comme il se doit en-dessous du « if ».

Imaginons maintenant un aiguillage à plus de deux directions :

Aiguillage à plus de deux directions

```
if condition1:
  bloc1
elif condition2:
  bloc2
else:
  bloc3
```

`elif` est une concaténation de « else » et « if » et pourrait se traduire par « sinon si ». Les conditions doivent être disjointes. En effet, l'ordinateur ne s'accommodant pas d'incertitude, son comportement face à plusieurs conditions validées en même temps serait très ambigu. Dès lors, il faut s'arranger pour que les conditions soient disjointes. Il y a là deux sous-ensembles de situations disjointes, correspondant aux conditions 1 et 2, inclus dans un ensemble englobant correspondant au else (et reprenant toutes les autres situations). Remarquons encore que pour un aiguillage à plus de 3 voies, la syntaxe sera:

Aiguillage à plus de trois directions

```
If condition1:
  bloc1
elif condition2:
  bloc2
elif condition3:
  bloc3
```

```
else:
  bloc4
```

Un des avantages du « elif » est de pouvoir aligner toutes les conditions disjointes les unes en-dessous des autres, sans être contraint dès lors de décaler de plus en plus vers la droite à chaque condition (la largeur du papier n'y suffirait pas).

Les boucles

Ce type d'instruction permet au programme de répéter, de compter ou d'accumuler tout en permettant une économie d'écriture considérable. La syntaxe de cette instruction est:

Syntaxe d'une boucle while

```
while condition:
  bloc
```

Tout d'abord, on se trouve à nouveau en présence des deux points (qui termine la condition) et du bloc indenté. Tant que la condition est vérifiée, c'est l'ensemble du bloc, c'est-à-dire la suite d'instructions indentées de la même manière, qui va être exécutée. Dès que la condition s'arrête d'être vérifiée, le bloc est ignoré et l'instruction suivante du code est exécutée. On peut d'ores et déjà voir apparaître deux problèmes : l'initialisation des variables composant la condition avant de procéder à l'évaluation et le risque d'une boucle infinie en présence d'une condition toujours vérifiée. Il faut donc veiller à faire évoluer dans le bloc la valeur d'au moins une des variables intervenant dans la condition et qui permettra à la boucle de s'interrompre.

La boucle est un mode de programmation qui permet d'éviter de réécrire un bloc d'instructions autant de fois qu'on souhaite l'exécuter.

Syntaxe d'une boucle while

```
compteur=1
while compteur <= 2:
  bloc1
  compteur =compteur + 1
bloc2
```

Lors de l'exécution du code qui précède, le bloc1 va être répété deux fois: en effet, au démarrage, le compteur vaut 1. Ainsi, étant plus petit que 2, on va exécuter le bloc1 et incrémenter le compteur de 1. Sa valeur vaut donc 2 à présent. A la fin de l'exécution du bloc, la condition du « while » est à nouveau testée. Cette valeur étant toujours plus petite ou égale à 2, le bloc1 va s'exécuter encore une fois et le compteur passera à 3. A ce stade-ci, la condition n'est plus validée et les instructions subordonnées au « while » seront ignorées. Le bloc2 sera exécuté et le code ira de l'avant.

Évidemment, même si la répétition est ici limitée (seulement 2 fois), on imagine aisément l'économie d'écriture dans le cas où nous souhaiterions exécuter 350 fois le bloc1. Remarquons, pour étayer nos propos, que si le compteur n'était pas incrémenté de 1 à chaque tour, sa valeur restant à 1, la condition serait toujours vérifiée et le bloc1 se verrait exécuter à l'infini, jusqu'à épuisement de l'ordinateur, contraint et obligé de jeter son processeur aux pieds du

programmeur. Il faut donc toujours, c'est capital, veiller à ménager une porte de sortie pour toute boucle. Elle vous en sera toujours gré. Le code qui suit illustre la boucle.

Exemple de boucle while

```
>>> compteur = 0
>>> while compteur < 4:
      print compteur
      compteur += 1


0
1
2
3
```

L'instruction `for … in` pour sa part est une autre forme de boucle qui permet cette fois d'itérer sur une collection de données, telle une liste ou un dictionnaire. Le petit exemple ci-dessous l'illustre dans le cas d'une liste. L'avantage est double. Il n'est pas nécessaire de connaître la taille de la collection, et la variable « x » incluse dans le « for in » reprend l'une après l'autre la valeur de tous les éléments de la collection.

Exemple de boucle for

```
>>> liste=["Bob","Jean","Pierre","Alain","Yves"]
>>> for x in liste:
      print (x)


Bob
Jean
Pierre
Alain
Yves
```

Dans ce code, la variable "x" incluse dans le "for … in" reprend l'une après l'autre la valeur de tous les éléments de la collection. Muni de ces différents éléments, il est aisé de comprendre le code résolvant le petit problème suivant, impliquant un dictionnaire cette fois. Soit une collection de 10 étudiants, chacun avec sa cote obtenue à un examen. Le programme qui suit, et dans lequel de nombreuses lignes ne sont pas montrées, permet d'établir la moyenne à l'examen ainsi que le nombre d'étudiants ayant raté cet examen. Sachez juste que `for x,y in listeEtudiant.items ()` permet de boucler et sur la clé et sur les éléments indexés par celle-ci.

Exemple de boucle for

```
>>> Moyenne = 0
>>> #On crée un dictionnaire des cotes indexés par les étudiants
>>> listeEtudiant ["Pierre "] = 10
>>> …….
>>> NombreEtudiants=0
>>> NombreRates=0
>>> for x,y in listeEtudiant.items():
      NombreEtudiants+=1
```

```
      Moyenne+=y
      if y < 10:
         NombreRates += 1


>>> Moyenne
63
>>> Moyenne/=NombreEtudiants
>>> Moyenne
10.5
>>> NombreRates
3
```

Autre petit code distrayant, un "jeu à boire" bien connu des étudiants de l'Université Libre de Bruxelles, le jeu du "ding ding bottle". Le jeu est on ne peut plus simple, car il faut boire vite et beaucoup. Il consiste à énumérer les nombres à tout de rôle et, si le nombre est un multiple de 5, le remplacer par "ding ding" et par "bottle" s'il s'agit d'un multiple de 7. Chaque erreur est punie par l'ingurgitation cul sec d'un verre d'alcool fort. Vous imaginez le déroulement des évènements et la joie simple des participants. Plus on se trompe, plus on boit, plus on boit, plus on se trompe. Voici dans sa version Python, qui a pour lui d'être d'une sobriété sans égale, la version de ce petit jeu décapant.

Ding ding bottle en Python

```
>>> while compteur < 20:
      if (compteur % 5 == 0) and (compteur % 7 == 0):
         print ("ding ding bottle")
      elif (compteur % 5 == 0):
         print ("ding ding")
      elif (compteur % 7 == 0):
         print ("bottle")
      else:
         print (compteur)
      compteur += 1


ding ding bottle
1
2
3
4
ding ding
6
bottle
8
9
ding ding
11
12
13
bottle
ding ding
16
```

```
17
18
19
```

Les fonctions

Il est possible dans Python et dans tous les langages de programmation de manière générale de découper le programme en "blocs fonctionnels" appelés "fonction" ou "procédure" et appelables partout dans ce même programme. La programmation de ces fonctions permet à la fois une meilleure modularité du code et s'avère d'autant plus nécessaire que ce même bloc fonctionnel se trouve sollicité à différents endroits du code, évitant ainsi de le réécrire pour chaque appel. Le comportement et le résultat des fonctions dépendront, comme en mathématique lorsque l'on définit une fonction f(x), d'un ou plusieurs arguments reçus en entrée de la fonction. L'exemple simple illustré ci-dessous est la définition de la fonction "cube(x)" qui renvoie le cube de l'argument x. Après avoir défini la fonction, on l'appelle 4 fois à l'intérieur d'une boucle.

Exemple de fonction

```
>>> def cube(x):
      return x*x*x

>>> compteur = 1
>>> while compteur < 5:
      cube(compteur)
      compteur+=1


1
8
27
64
```

La présence du mot clef "return" indique la fin de la fonction, en renvoyant le résultat qui suit le "return". En présence d'un « return », l'exécution de la fonction se termine en renvoyant le contenu du « return » au code appelant. On pourra bien sûr appeler la fonction cube partout où cela s'avère nécessaire. L'appel de la fonction à l'intérieur d'un code rompt avec la séquence en cours (afin de déclencher celui repris dans la fonction) par l'intermédiaire du mécanisme dit d'interruption. Ce que le code faisait jusqu'à l'appel de la fonction est sauvegardé dans une zone mémoire dédiée, de sorte qu'une fois la fonction achevée, le code puisse reprendre son cours là où il l'avait laissé.

Comme autre exemple, la fonction "inverse" décrite ci-dessous permet d'inverser une chaîne de caractères passée comme argument. Elle est ensuite exécutée. A nouveau, il est possible d'appliquer la fonction "inverse", comme la suite du code l'illustre, sur tous les éléments d'une liste afin de remplacer chaque élément de la liste par son inverse.

Exemple de fonction

```
>>> def inverse(x):
```

```
      i=""
      n=0
      for a in x:
         n+=1
      compteur = n
      while compteur > 0:
         i+=x[compteur-1]
         compteur-=1
      return i

>>> inverse("Bersini")
'inisreB'
>>> liste=["Jean","Paul","Jacques"]
>>> n=0
>>> for x in liste:
      liste[n]=inverse(x)
      n+=1


>>> liste
['naeJ', 'luaP', 'seuqcaJ']
>>>
```

Lorsqu'on passe un argument dans une fonction, il est important de comprendre qu'il s'agit là d'une variable locale (à la différence de "globale") qui n'aura d'existence que le temps d'exécution de la fonction. La fonction terminée, toutes les variables passées en argument disparaissent d'une zone de mémoire locale associée à la seule fonction. Le petit programme qui suit devrait vous permettre de mieux saisir ce principe.

Exemple de fonction

```
>>> def test(x):
      x+=1
      print (x)


>>> x=5
>>> test(x)
6
>>> x
5
>>>
```

On y réalise qu'alors que la variable x comprend la valeur 5 au départ de l'exécution de la fonction, le fait qu'elle soit incrémentée de 1 à l'intérieur de la fonction ne modifie en rien sa valeur originale, car cette incrémentation n'affecte qu'une copie locale de la variable qui disparaîtra de la mémoire aussitôt l'exécution de la fonction terminée. La variable originale, elle, sera préservée dans une zone mémoire dédiée et sera récupérée intacte à l'issue de l'appel de la fonction. On découvre également deux variables qui bien qu'elles soient nommées pareille ne sont en rien confondues par le programme.

La programmation objet

Classe et objet

La programmation objet répartit l'effort de programmation sur un ensemble de classes associées aux acteurs de la réalité que le programme se doit d'affronter. Ainsi une application bancaire reprendra comme classe : « les clients », « les comptes en banque », « les opérations », « les emprunts », etc. Chaque classe est à la source d'un ensemble d'instances appelées objets et qui ne se différencient entre eux que par la valeur prise par leurs attributs.

Une classe se définit par ses attributs et ses méthodes. Les attributs décrivent ce que la classe est, les méthodes ce que la classe fait. Chaque objet possède à un moment donné pour ses attributs des valeurs qui lui sont propres et sur lesquels les méthodes agissent. Les méthodes sont déclarées exactement comme des fonctions mais ont ceci de singulier qu'elles ne portent que sur les seuls attributs de la classe. L'état de l'objet évolue dans le temps au fur et à mesure de l'exécution des méthodes sur cet objet. Décortiquons le code suivant.

Exemple de programmation orientée objet en Python

```
    class Client:

    #Le constructeur définit et initialise les attributs de l'objet
      def __init__(self,nomInit,prenomInit,adresseInit,ageInit):
        self.__nom=nomInit #par la présence des « __ » les attributs
                           #deviennent privés.
        self.__prenom=prenomInit
        self.__age=ageInit
        self.__adresse=adresseInit

      def donneNom(self):
        return self.__nom

      def donneAdresse(self):
        return self.__adresse

      def demenage(self,newAdresse):
        self.__adresse=newAdresse

      def donneAge(self):
        return "age: " + str(self.__age)

      def vieillit(self):
        self.__age+=1

      def __str__(self):
        return "client: " + self.__prenom + " " + self.__nom


#creation d'un premier client
client1 = Client("Bersini","Hugues","rue Louise",20)

#creation d'un deuxieme client
```

```
client2 = Client("Lenders","Pascale","rue Juliette",30)

#exécution des méthodes sur les objets clients
#d'abord la méthode __str__ appelée de manière implicite.
print (client1)
print (client2)
client1.__nom="Dupont"
#on ne peut pas accéder à l'attribut privé __nom
print (client1)
print (client1.donneAdresse())
#la seule manière de modifier l'adresse
client1.demenage("rue Juliette")
print (client1.donneAdresse())
#la seule manière de modifier l'âge
client2.vieillit()
print (client2.donneAge())
```

Résultat de l'exécution

```
>>>
client: Hugues Bersini
client: Pascale Lenders
client: Hugues Bersini
rue Louise
rue Juliette
age: 31
>>>
```

La classe client possède quatre attributs privés, c'est-à-dire inaccessibles de l'extérieur de la classe. C'est la présence de « __ », précédent l'attribut qui leur donne ce caractère « privé ». Ainsi, lorsque plus bas dans le code, on essaie de modifier le nom « client1.__nom = « Dupont » », le changement ne s'effectue pas, où plutôt s'effectue sur une autre variable créée à la volée, mais pas l'attribut qui nous intéresse ici. L'orienté objet, pour des raisons de stabilisation de code, favorise l'encapsulation de la plupart des caractéristiques de la classe (les attributs comme la plupart des méthodes). Tout ce qui est encapsulé ou privé peut se trouver modifier sans affecter en rien les autres parties du code qui accèderont à la classe client (accès restreint à ses seules parties publiques). L'encapsulation est le garant d'une stabilité dans le temps car elle permet à de larges parties du code de subir des modifications sans que d'autres parties ne doivent être réécrites.

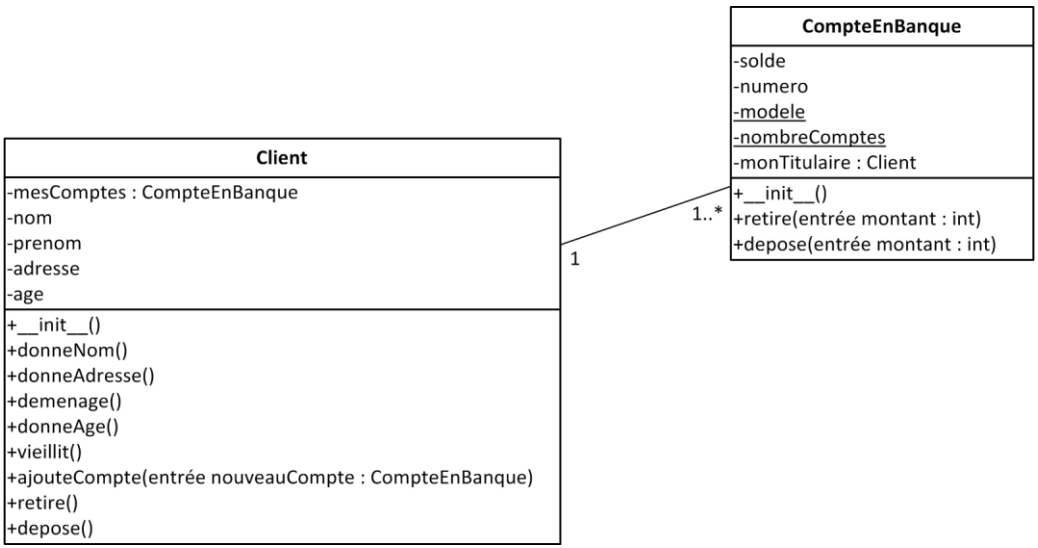
C'est le constructeur « __init__ » qui se charge d'initialiser les attributs privés. Il est appelé lors de la création de l'objet, comme lors de « client1=Client(« Bersini », « Hugues », « rue Louise », 20) ». Dans le code, deux objets clients sont créés référés par « client1 » et « client2 ». La méthode « __str__ » est héritée de la super classe object (nous introduirons l'héritage plus tard), elle est redéfinie dans la classe Client (nous expliquerons la « redéfinition » plus tard également) et appelée de manière implicite à chaque fois que l'on affiche l'objet (en fait à chaque fois que l'on veut associer une chaîne de caractère à l'objet). L'adresse et l'âge étant deux attributs privés, la seule manière d'y accéder est par l'entremise de méthodes publiques (aucun « __ » ne précède leur déclaration). Ainsi, seule la méthode « demenage » pourra modifier l'adresse et la méthode « vieillir » l'âge du client. Le mot clé « self » est indispensable

dès qu'il s'agit de faire référence à l'objet lui-même. De fait, toutes les méthodes portant sur l'objet se doivent de recevoir cet objet, c-à-d « self » en paramètre, car elles agissent bien sur le « self » de l'objet.

Association entre classes

La pratique de l'orienté objet consiste d'abord et avant tout en l'éclatement de l'application logicielle entre les classes constitutives du projet. Ainsi, si dans un logiciel bancaire, le client veut retirer 1000 euro de son compte en banque, c'est l'objet client qui sollicitera la méthode « retrait(1000) » exécutée sur le compte en question. Ce déclenchement de la méthode « retrait » sur le compte, provoqué par le client, est appelé dans le jargon objet « un envoi de message ». Tous les ingrédients d'un message s'y retrouvent, l'expéditeur : « l'objet client », le destinataire : « l'objet compte en banque » et le contenu du message : « retrait (1000) ».

Dans le diagramme de classe UML représenté ci-dessous, chaque objet client est associé à un ensemble d'objets compte en banque et, en retour, chaque objet compte en banque est associé à un et un seul client. Cette association « 1-n » est bidirectionnelle. Le code qui suit réalise cette association entre les deux classes. Son exécution se limite à créer un client, un compte en banque, et à permettre au client de déposer un certain montant d'argent sur son compte. Pour ce faire, le client va envoyer le message « depose(1000) » sur le compte qu'il aura sélectionné. Notez également, l'apparition d'attributs statiques dans la classe compte en banque, le modèle utile au numéro de compte et le nombre de comptes créés (qui seront tout deux utilisés pour attribuer le numéro de compte). Les attributs statiques sont associés à la classe plutôt qu'aux objets (ils sont soulignés dans le diagramme de classe). Ils n'ont besoin que de la classe pour exister et être utilisés. Ils le sont d'ailleurs en faisant toujours allusion à leur classe (comme dans « CompteEnBanque.nombreComptes »). Leur utilisation ne nécessite donc pas de recourir au mot clé « self ». Tout ce qui est statique peut exister sans objet.



(3.1 - Diagramme de classes 1.png)

Clients et comptes

Implémentation du diagramme en Python

```
class CompteEnBanque:
    __modele="210-" #attribut statique
    __nombreComptes=0 #attribut statique

    def __init__(self):
        self.__solde=0
        self.__titulaire=""
        self.__numero=CompteEnBanque.__modele + \ #a la ligne
            str(CompteEnBanque.__nombreComptes)
        CompteEnBanque.__nombreComptes+=1

    def assigneTitulaire(self,titulaire):
        self.__monTitulaire = titulaire

    def donneNumero(self):
        return self.__numero

    def retire(self,montant):
        self.__solde-=montant

    def depose(self,montant):
        self.__solde+=montant

    def __str__(self):
        return "le solde du compte: " + self.__numero + \
               " est: " + str(self.__solde)



class Client:
    #Le constructeur initialise les attributs de l'objet
    def __init__(self,nomInit,prenomInit,adresseInit,ageInit):
        self.__nom=nomInit
        self.__prenom=prenomInit
        self.__age=ageInit
        self.__adresse=adresseInit
        self.__mesComptes=[]

    def donneNom(self):
        return self.__nom

    def donneAdresse(self):
        return self.__adresse

    def demenage(self,newAdresse):
        self.__adresse=newAdresse

    def donneAge(self):
```

```
        return "age: " + str(self.__age)

    def vieillit(self):
        self.__age+=1

    def __str__(self):
        return "client: " + self.__prenom + \
            " " + self.__nom

    #cette methode rajoute un compte dans la liste du client
    #elle s'assure egalement que le compte ait bien son client

    def ajouteCompte(self,compte):
        self.__mesComptes.append(compte)
        compte.assigneTitulaire(self)

    def identifieCompte(self):
        print ("Sur quel compte ?")
        x=input("?") #cette instruction permet de lire un string
                     #a l'ecran en promptant avec un "?"
        for c in self.__mesComptes:
            if c.donneNumero() == x:
                res=c
                break
        return c

    def retire(self):
        c=self.identifieCompte()
        print ("quel montant ?")
        x=int(input("?"))
        c.retire(x)

    def depose(self):
        c=self.identifieCompte()
        print ("quel montant ?")
        x=int(input("?"))
        #le string lu a l'ecrant devra etre modifie en int
        c.depose(x)

#creation d'un premier client
client1 = Client("Bersini","Hugues","rue Louise",20)
#creation d'un premier compte
compte1= CompteEnBanque()
print (compte1)
#ajout du compte dans la liste des comptes du client
client1.ajouteCompte(compte1)
#le client depose de l'argent sur son compte
client1.depose()
print (compte1)
```

Résultat de l'exécution

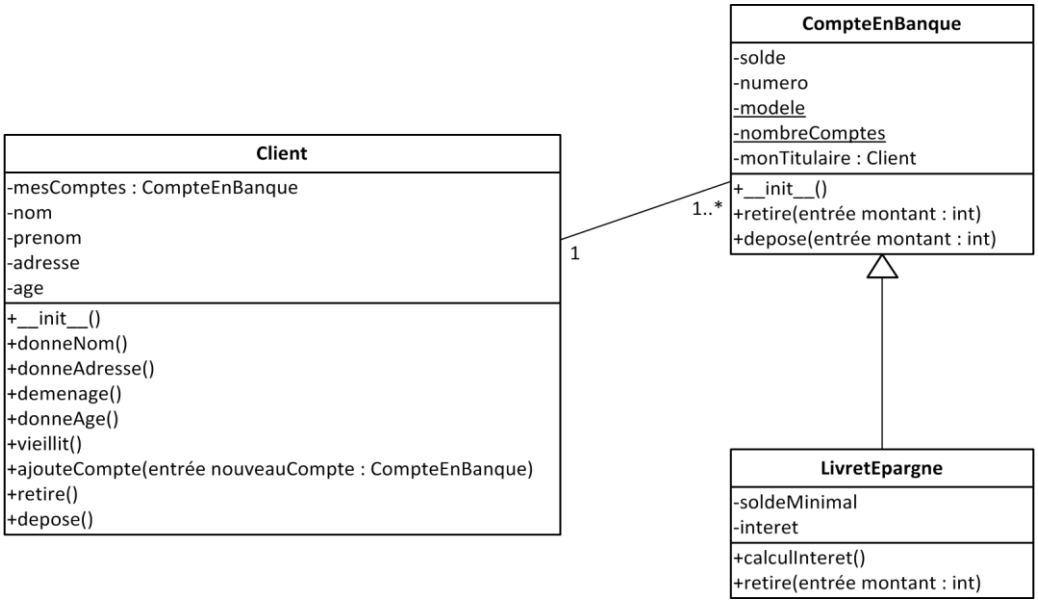
```
>>>
le solde du compte: 210-0 est: 0
Sur quel compte ?
```

```
?210-0
quel montant ?
?1000
le solde du compte: 210-0 est: 1000
>>>
```

Héritage et polymorphisme

En orienté objet, la deuxième manière de découper l'application en ses classes est de recourir au mécanisme d'héritage qui permet de considérer les classes à différents niveaux de généralité. On parlera ici de classes parents (ou super classes) pour les plus génériques et de classes filles (ou sous classes) pour celles qui en héritent. Les classes s'installent ainsi dans une taxonomie, des plus génériques aux plus spécifiques. Les classes filles héritent des attributs et des méthodes de leurs parents et peuvent se distinguer de trois manières : ajouter des nouveaux attributs qui leur seront propres, ajouter des nouvelles méthodes qui leur seront propres, redéfinir des méthodes existant déjà dans les classes parents. La redéfinition signifie de reprendre exactement la même signature de la méthode (nom et argument) mais lui associer un code différent.

Dans le code et le diagramme de classe mis à jour qui suivent, la classe « LivretEpargne » hérite de la classe « CompteEnBanque », via les parenthèses. Elle ne possède en propre que deux attributs statiques : le « soldeMinimal » en deçà duquel le client ne peut débiter son solde, et un taux intérêt. Elle rajoute une méthode particulière pour calculer son intérêt et elle redéfinit la méthode « retire » en prenant en considération la contrainte du solde minimal. Si nécessaire, elle peut rappeler la méthode prévue dans la classe mère (en la référant par « CompteEnBanque.retire() »). Par ailleurs, le solde étant privé dans la classe mère, il n'est accessible que par l'entremise de la méthode donneSolde(). Ici, il n'est pas nécessaire de désambiguïser l'appel en faisant explicitement allusion à la classe mère puisque la méthode n'est pas redéfinie.



(3.2 - Diagramme de classes 2.png)

Héritage de classe

Implémentation du diagramme en Python

```
class LivretEpargne(CompteEnBanque):
    __soldeMinimal=0 #attribut statique
    __interet = 0.1 #attribut statique

    def calculInteret(self):
        depose(donneSolde()*__interet)

    def retire(self,montant):
        if (self.donneSolde() - montant) >= \
        LivretEpargne.__soldeMinimal:
            CompteEnBanque.retire(montant)
            #appel de la methode de la classe mère
        else:
            print ("pas assez d'argent sur le compte")
```

Dans les instructions qui suivent, on voit que le client dépose d'abord 1000 euro sur son compte puis souhaite en retirer 2000. Comme le compte en question s'avère un livret d'épargne et non un compte en banque quelconque, c'est bien la version redéfinie de la méthode « retire » du livret d'épargne qui sera appelée. L'appel de la méthode redéfinie en fonction de la nature de la sous-classe sur laquelle cette méthode s'exerce est appelé « polymorphisme ». On en conçoit aisément l'avantage en présence d'un ensemble d'objets tous issus de classes filles différentes et toutes ayant leur propre implantation de la méthode « retire ». Envoyer le même message sur

tous ces objets aura des effets différents selon le type de classe fille dont est issu l'objet. Cela permettra à la classe expéditrice d'envoyer un seul et même message à plusieurs sous-classes de la classe « CompteEnBanque » sans se préoccuper plus que nécessaire de leur nature ultime. Chacune exécutera le message à sa manière et c'est tout ce qu'on lui demande.

Utilisation des classes définies

```
#creation d'un premier client
client1 = Client("Bersini","Hugues","rue Louise",20)

#creation d'un premier compte
compte1= LivretEpargne()
print (compte1)
#ajout du compte dans la liste des comptes du client
client1.ajouteCompte(compte1)
#le client depose de l'argent sur son compte
client1.depose()
print (compte1)
#le client retire de l'argent du compte
client1.retire()
```

Résultat de l'exécution

```
>>>
le solde du compte: 210-0 est: 0
Sur quel compte ?
?210-0
quel montant ?
?1000
le solde du compte: 210-0 est: 1000
Sur quel compte ?
?210-0
quel montant ?
?2000
pas assez d'argent sur le compte
>>>
```

Nous avons déjà rencontré un exemple de redéfinition et de polymorphisme en présence de la redéfinition de la méthode « __str__ » qui permet d'associer un string à un objet. Dans le même ordre d'idée, plus loin dans le livre, Django fera usage de la redéfinition de la méthode « __unicode__ », qui permet également une représentation des objets sous forme de caractère.

Import et from : Accès aux librairies Python

Comme dans tous les langages de programmation, le programmeur fait un très large usage de fonctionnalités déjà programmées et installées dans des modules de code couramment appelés « librairies ». Mais avant de pouvoir les utiliser, il faut évidemment les localiser, les retrouver et donc indiquer à l'exécution où se cachent ces librairies (ou, plus précisément, quel nom portent elles).

En python, cela se fait par l'entremise de l'instruction « import librairie » si l'on veut utiliser les fonctions de cette librairie (on écrit alors : « librairie.fonction()), ou plus finement : « from libraire import fonction » (pour simplement appeler « fonction() »).

L'exemple ci-dessous illustre les instructions « import » et « from ».

Exemple d'instructions import et from

```
>>> #Je souhaite obtenir un nombre aléatoire entre 1 et 10
>>> print (random.randint(1,10))

Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print (random.randint(1,10))
NameError: name 'random' is not defined
>>> import random   # j'importe la librairie random
#un nombre au hasard entre 1 et 10
>>> print (random.randint(1,10))
>>> liste=["a","b","c","d","e","f"]
>>> print liste
['a', 'b', 'c', 'd', 'e', 'f']
#J'utilise une autre fonction du module random
>>> random.shuffle(liste)
>>> liste
['c', 'b', 'e', 'a', 'f', 'd'
#Je veux avoir l'heure à ce moment précis
>>> clock()
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    clock()
NameError: name 'clock' is not defined
#j'importe la seule fonctionnalité qui me le permet: clock()
>>> from time import clock
>>> clock()
6.634921477450347e-06
#clock() me donne l'heure dans un format particulier.
```