Cours programmation - Marc Jachym
Licence professionnelle – Métrologie dimensionnelle et qualité
IUT de St Denis, Université Paris 13

Cours de Programmation avec le langage Python

Niveau débutant en programmation

Table des matières

Partie III – Notions avancées

III.1 Retour sur les collections

Nous avons vu dans la partie 1 deux types de « collections » ou « séquences » : les **chaines de caractères** et les **listes**. Dans cette partie nous allons voir deux autres types de séquences offerts par le langage Python : les **tuples** et les **dictionnaires.** Nous reviendrons ensuite sur les opérateurs et fonctions associés à ces types en présentant ce qu'ils ont en commun et ce qui est particulier à chacun des types.

Voici déjà trois choses que tous les types séquences ont en commun :

1/ On peut accéder à leurs éléments par l'utilisation d'un indice placé entre crochets, de la manière suivante :

```
i = <une certaine valeur>
elem = maCollection[i]
```

2/ Ils renvoient tous leur nombre d'éléments grâce à la fonction len() :

```
nbElem = len(maCollection)
```

3/ La structure **for** permet de parcourir tous les éléments qu'ils contiennent :

```
for elem in maCollection :
        …. utiliser l'élément courant elem ...
```

[Sequence types : http://docs.python.org/2.7/library/stdtypes.html#sequence-types-str-unicode-list-tuple-bytearray-buffer-xrange]

III.1.1 Les tuples (ou appelés aussi n-uplets)

Les tuples sont des ensembles figés de valeurs. Les valeurs sont séparées par des virgules. On peut mélanger les types pour les éléments constituant un tuple.

Ex :

```
t1 = (1, 123, 7,9)   # un tuple homogène
t2 = (1,2,3, "ab", (1,2,3))  # un tuple hétérogène, ou imbriqué
```

[Tuples and sequences : http://docs.python.org/2.7/tutorial/datastructures.html#tuples-and-sequences]

III.1.2 Les dictionnaires

Les dictionnaires sont un type de données dynamiques très pratique offert par le langage Python. Par données « dynamique », on entend un type collection dont le nombre d'éléments n'est pas figé à l'avance et dans lequel ont peut à tout moment ajouter de nouvelles valeurs. C'est l'interpréteur Python qui s'occupe de réserver (et au final de libérer) la mémoire nécessaire au stockage. Ce point est également vrai avec les listes.

A la différence des listes, dont les valeurs sont en quelque sorte "ordonnées" grâce à leur index numérique, **les valeurs des dictionnaires sont associées à des clés** qui sont en général des entiers ou des chaines de caractères.

Un dictionnaire s'initialise grâce à une notation utilisant des accolades.
Initialisation d'un dictionnaire vide :

```
MonDict = {}
```

Initialisation d'un dictionnaire avec un certain nombre de couples « clé valeur »

```
UnePersonne = { "Prénom" : "Luc", "Nom" : "Martin", "Age" : "22", "Ville" : "Paris"}
```

L'accès aux éléments se fait grâce à la **notation à crochets,** comme pour les listes, mais **avec la clé de l'élément** à la place d'un indice.

```
leNom = UnePersonne["Nom"] # accès en lecture à un élement
UnePersonne["Ville"] = "Marseille" # accès en écriture à l'élément
UnePersonne["Age"] = 23 # accès en écriture à l'élément
```

Si on utilise une clé qui n'existe pas encore pour écrire dans un dictionnaire, alors l'élément clé : valeur est ajouté.

```
UnePersonne[CodePostal] = 75000 # ajout d'un nouvel élément dans le dictionnaire
print UnePersonne # affiche dans la console le contenu du dictionnaire
```

Remarque : ce dernier point est très surprenant par rapport aux langages de programmation plus anciens et plus proches de la "structure de la machine" comme le langage C. Il n'est pas anodin d'écrire en mémoire à une adresse qui n'a pas été préalablement réservée ! En C ce serait une erreur grave qui conduirait à un comportement indéterminé et très rapidement à un arrêt du programme.

Il faut bien comprendre qu'avec les langages interprétés modernes comme Php ou Python, il y a comme une "machine virtuelle" entre les instructions du programme et le microprocesseur. Cette machine virtuelle effectue de nombreuses actions pour prendre en charge les ordres du programme, comme réserver de la mémoire et mettre à jour en mémoire les objets évolués manipulés par le programmeur, par exemple les listes et les dictionnaires.

[Doctionnaries : http://docs.python.org/2.7/tutorial/datastructures.html#dictionaries]

III.1.3 Les opérateurs et fonctions applicables aux séquences

Nous connaissons trois types de séquences, deux fixes, les chaines de caractères et les tuples et le troisième modifiable, les listes.

Voici les opérateurs que ces 3 types ont en commun :

- **le test d'appartenance** :

X **in** S, X **not in** S

par exemple :
'n' in 'lune'
renvoie la valeur True

- **la boucle for**

for X in S :

- **la concaténation** (mise bout à bout)

S + S

- **l'accès par indice**

S[i]

- **l'accès par plage d'indices**

S[i:j]
S[:j]
S[i:]

- **opérateur longueur**

len(S)

- **opérateurs min et max**

min(S)
max(S)

III.1.4. Les opérateurs et fonctions applicables aux dictionnaires

- **len(d)** retourne le nombre d'élément du dictionnaire d
- d[cle] accès à l'élément repéré par *cle*
- d[cle] = valeur ajout d'une entrée au dictionnaire
- del d[cle] enlève l'élément du dictionnaire
- cle in d renvoie vrai si une clé *cle* est définie dans le dictionnaire d
- d.items() retourne le contenu du dictionnaire sous la forme d'une liste de paires (cle, valeur)
- d.keys() retourne l'ensemble des clés du dictionnaire dans une liste
- d.values() retourne l'ensemble des valeurs du dictionnaire dans une liste

[Dictionnaires ou 'Mapping types' : http://docs.python.org/2.7/library/stdtypes.html#mapping-types-dict]

III.2 Travail sur les fichiers

Un fichier est une suite de données enregistrées, sous un certain nom, sur le disque dur d'un ordinateur ou sur un support amovible. Au niveau le plus élémentaire toute donnée est représente par un certain nombre d'octets. Les supports contiennent les fichiers dans des structures appelés « systèmes de fichiers », elles-même contenues dans des partitions. Il y a différents types de systèmes de fichiers, par exemple FAT (16 ou 32), NTFS, EXT3, …
Le système de fichier permet de gérer des répertoires et des fichiers repérés par des noms obéissant à certaines règles : nombre de caractères maximal, caractères autorisés et interdits. De plus le système de fichier garde un certain nombre de d'information sur chaque fichier et répertoire : le propriétaire, les droits du propriétaire et des autres utilisateurs, les dates de création, de dernière modification, de dernière accès en lecture, etc.

Il y a deux niveau de gestion des fichiers dans un programme. Et donc deux catégories de fonctions correspondantes :

1/ La gestion des systèmes de fichiers : trouver des répertoires et des fichiers et gérer les fichiers et répertoires dans leur ensemble : renommer, effacer, déplacer et obtenir les informations concernant les utilisateurs et les dates.

2/ La gestion du contenu d'un fichier donné : créer un fichier, ouvrir un fichier existant, lire les données dans le fichier, écrire dans le fichier.

III.2.1 Lire / Écrire le contenu des fichiers

Tout commence par l'utilisation d'une fonction de la bibliothèque standard (Built-in) : open() dont voici la signature habituelle :

open(*name*, *mode*)

name est une chaine de caractères indiquant le nom du fichier à ouvrir. Ce nom peut contenir l'indication du chemin de répertoires à suivre pour trouver le fichier. La référence (le point de départ) est le répertoire courant de travail.

En cas d'erreur, en particulier si le fichier indiqué n'est pas trouvé, une erreur se produit. En langage Python, **les erreurs sont gérées grâce à un mécanisme appelé exceptions**.
Ce mécanisme offre au programmeur la possibilité de prévoir des traitements spéciaux à effectuer quand une erreur particulière, prévue à l'avance, se produit. Par exemple, on pourrait imaginer, si un fichier à ouvrir n'a pas été trouvé, de demander à l'utilisateur de saisir un autre nom pour le fichier.
Exemple :

```
>>> open("rep\monfichier.txt", "r")
Traceback (most recent call last):
 File "<pyshell#0>", line 1, in <module>
   open("monfichier.txt", "r")
IOError: [Errno 2] No such file or directory: 'rep\monfichier.txt'
>>>
```

Ici la fonction est appelée sur un **chemin** (répertoire + nom de fichier) qui n'est pas trouvé. L'interpréteur Python déclenche une exception IOError (erreur d'Entrée/Sortie). Cette exception n'étant pas gérée par le programme, le programme s'arrête et affiche le message associé à l'exception.

Le second argument de la fonction *open* est **mode**. C'est une chaine de caractère indiquant dans quel mode on souhaite ouvrir le fichier. Les modes possibles sont : lecture, écriture d'un nouveau fichier, écriture avec ajout à la fin d'un fichier existant,

lecture et écrite. De plus un fichier peut être ouvert en **mode texte** ou en **mode binaire**.

Que sont les fichiers texte et binaire ?

Un **fichier texte** est vu comme un ensemble de lignes terminées par un séparateur. Le séparateur n'est pas le même sur les différentes plate-formes (= systèmes d'exploitations). Sous Unix/Linux, les fins de lignes des fichiers textes sont indiquées par le caractère «new line», noté \n (de valeur ASCII 10). Sous Macintosh, la fin de ligne est indiquée par «carriage return», noté \r (de valeur ASCII 13). Sous Windows, ce sont les deux caractères qui sont utilisés, le marqueur de fin de ligne étant \r\n.

(voir en annexe la description des tables ASCII)
Python, qui sait sur quelle plate-forme il est lancé, normalise les fins de ligne des fichiers texte par la notation unique \n , quel que soit le système. Les lignes sont lues dans les fichiers texte par Python et sont considérées comme des chaines de caractères.

Dans le cas des **fichiers binaires**, les octets contenus dans le fichier sont lus tels quels, sans une interprétation de la part de Python. Ce mode de travail est adapté pour les fichiers de données non textuelles, comme les fichiers images par exemple.

[Reading and writing files : http://docs.python.org/2.7/tutorial/inputoutput.html#reading-and-writing-files]

III.2.2 Les opérations offertes par l'objet *file*

En cas de succès, la fonction *open* renvoie un objet de type fichier
Voici quelques méthodes des objets de type *file* **:**

file.read(nbre)	size est une valeur entière. la méthode read lit (au plus) *nbre* octets dans le fichier et retourne ces octets sous la forme d'une chaine de caractères. S'il reste moins de *nbre* caractères à lire jusqu'à la fin du fichier, tous ces caractères sont retournés. Si *nbre* est nul ou négatif, tous des caractères jusqu'à la fin du fichier sont retournés. On voit donc qu'il est possible de lire en seule une fois tout le contenu d'un fichier, contenu qui sera placé dans une chaine de caractères que Python se charge de placer en mémoire. C'est là une spécificité des langages dynamiques comme Python.

file.write(str)	écrit le contenu de la chaine *str* dans le fichier. Cette fonction ne retourne rien.
file.seek(pos)	place le curseur (= la position courante) dans le fichier à la position *pos*
file.tell()	renvoie la position courante de lecture/écriture dans le fichier. Cette fonction, ainsi que la précédente, est utile quand on travaille sur des fichiers structurés où l'on parcourt des enregistrements de taille fixe.
file.readline(nbre)	lit une ligne complète ou au maximum *nbre* caractères si une fin de ligne n'esty pas rencontrée. Renvoie une chaine de caractères.
file.readlines()	lit depuis la position courante dans le fichier jusqu'à la fin du fichier. Retourne, sous la forme d'une liste de chaines de caractères, l'ensemble des lignes lues dans le fichier.
file.close()	Ferme le fichier. Cette opération est importante sans quoi les données pourraient ne pas être effectivement écrite dans le fichier.

[Files objects : http://docs.python.org/2.7/library/stdtypes.html#file-objects]

III.3 Les objets en Python

III.3.1 La notion d'objets

Python est un l**angage à objets** ou **langage orienté objets**. C'est à dire qu'en plus de variables simples comme des entiers, des flottants, des caractères et des variables composées comme les tableaux et les structures du langage C, Python offre la possibilité de manipuler des variables de type beaucoup plus complexe.
En particulier, on appelle langage à objets, un langage qui permet d'**associer, au sein même des variables, les données avec leurs traitements**.

Nous avons déjà rencontré plusieurs types d'objets :

1/ les chaines de caractères. Ce ne sont pas de simples tableaux « passifs » de caractères. Au contraire, les variables de type « chaine de caractères » contiennent tout un ensemble de fonctions de manipulation à s'appliquer sur elles-mêmes.
Exemples :

str.capitalize()

str..isalnum()

str.count(sub[, start[, end]])

str.find(sub[, start[, end]])

2/ Les séquences et dictionnaires

3/ Les fichiers
La création d'un fichier se fait à l'aide de la fonction « traditionnelle » *open*, mais celle-ci, en cas de succès, **renvoie un objet de type *file*** qui contient toutes sortes de fonctions de manipulation du fichier qui vient d'être ouvert.

Dans tous ces exemples, on voit qu'un objet est une variable qui contient des données et des traitements sur ces données. Ces traitements sont des fonctions qui sont invoquées grâce à la notation « point ». Par exemple :

```
taille = 100
monfichier = open ("fichier.txt », "r")   # Création d'un objet file
buffer = monfichier.read(taille)   # lecture dans l'objet file grâce à sa méthode read
…
```

III.3.2 La programmation orientée objets

L'utilisation d'objets en programmation va au delà de l'utilisation des éléments et bibliothèques déjà prêts et offerts par le langage. Le programmeur peut lui-même créer et utiliser ses propres objets.

L'écriture d'un programme sous la forme d'interactions entre des objets permet de modéliser de manière plus intuitive le travail que doit réaliser un logiciel. Au lieu d'avoir de grands paquets d'instructions en blocs monolithiques, nous pouvons avoir des objets plus intuitifs qui offrent leurs services sous la forme de **méthodes**.

Ainsi la connaissance d'un objet disponible et la description de ses méthodes (ce qu'on appelle **l'interface de l'objet**) permet de l'utiliser sans avoir besoin d'aller regarder dans son code, c'est à dire sans besoin d'aller voir comment ses méthodes sont écrites (ce que l'on appelle la définition ou **l'implémentation de l'objet**).

III.3.2.1 Création d'un nouvel objet : définir une *classe*

Une **classe** est la définition d'un nouveau type d'objet.
En Python on utilise le mot clé *class*

La structure générale de définition d'une classe est la suivante :

class maClasse **:**

    def __init__(self, arg1, arg2, …**) :**
        … code du constructeur de la classe. ..

    def fonct1(**self,** args, …) **:**
        … code de la méthode 1 ….

    def fonction2 (**self,** arg2, …) **:**
        … code de la fonction 2 …

La variable **self** représente l'**instance** de la classe (l'objet courant représenté par cette classe lors de l'exécution du programme) à l'**intérieur** de la définition de celle-ci. C'est par cette variable self que l'on accède aux **données membres** et **fonctions membres** (= méthodes) de la classe dans sa définition.

Exemple :

On va créer une classe pour décrire un objet géométrique en deux dimensions.

Tout d'abord voici une classe permettant de décrire un point de l'écran et de lui affecter une couleur.

```
class Point :

    # constrcuteur. On enregistre les coordonnées dans des variables de classe
    def __init__(int x, int y) :
        self.x = x
        self.y = y
        self.color = {"red" : 255, "green" : 255, "blue" : 255  }
                                        # on utilise un dictionnaire
                                        # par défaut le point est noir

    # méthode pour afficher le point en couleur en donnant ses trois composantes
    # couleurs rouge, vert, bleu
```

```
    def setColor (self, red, green, blue) :
        # d'abord il faudrait tester que les trois arguments sont bien des valeurs
        # entières comprises entre 0 et 255
        self.color["red"] = red
        self.color["green"] = green
        self.color["blue"] = blue
```

Maintenant voici la définition de la classe Forme. Cette classe est basée sur un point de référence servant à positionner la forme quelque part dans le plan de l'écran. La classe Forme est une classe générique ou classe « de base », ne représentant en elle-même aucune forme véritable mais servant de point de départ à la création de formes réelles (rectangle, cercle, etc).

```
class Forme :

    # constructeur
    def __init__(self, pointRef) :
        self.pointRef = pointRef


    def translation(self, offsetX, offsetY) :
        self.pointRef.x += offsetX
        self.pointRef.y += offsetY

    def rotation(self, int angle) :
        pass  # ne fait rien sur cette classe de base

    def dessine(self,) :
        pass
```

|A first look at classes: http://docs.python.org/2.7/tutorial/classes.html#a-first-look-at-classes]

III.2.2.2 Comment découper un programme en classe de manière adéquate ?

Il n'y a pas de réponse simple et définitive à cette question, ni de méthode prédéterminée. La programmation objet vise à faciliter le travail de création logicielle en le structurant dès les premières étapes de la conception.
On peut imaginer des objets qui modélisent d'assez près un domaine de la vie courante que l'on veut gérer à l'aide d'un logiciel. Par exemple on peut imaginer des objets clients, produits, facture, etc...

La programmation d'interfaces graphiques est un autre domaine qui se prête bien à la découpe en objets. On peut imaginer des objets fenêtres ainsi que des widgets de différents types (bouton, cases à cocher, zone d'affichage de texte, zone de saisie, etc...).

Le fait d'imaginer des objets, indépendamment de savoir quel code informatique les mettra en œuvre, permet de clarifier la conception dès le dialogue avec les futurs utilisateurs.

Il est possible aussi d'imaginer des objets abstraits génériques que l'on pourra spécialiser par la suite.

Par exemple je peux imaginer un objet fenêtre général (lors de la conception d'une interface graphique). Une fenêtre a un titre, un style (par exemple épaisseur de bordure), une taille (largeur + hauteur). Les éléments de la fenêtre générique peuvent être mis à jour grâce aux méthodes de l'objet fenêtre comme setWidth(valeur), setHeight(valeur), setTitle(titre), resize (newWidth, newHeight), etc...

A partir de là, on peut imaginer des fenêtres plus spécialisées. Par exemple une fenêtre affichant un message avec un bouton Ok pour validation, une autre affichant une question et deux boutons « Oui » et « Non » pour saisir le choix de l'utilisateur, une autre encore permettant la saisie d'informations dans des zones de texte (boite de dialogue) avec un bouton « Ok » pour valider.

Il est facile d'imaginer à partir de là toute une hiérarchie d'objets.

III.2.2.3 Programmation orientée objet, classes dérivées, héritage.

Ce cours est un cours d'initiation, on n'étudiera pas tous les raffinements possibles de la programmation orientée objet mais il nous faut au moins signaler l’existence des classes dérivées.

Nous avons parlé dans l'exemple précédent sur les interfaces graphiques, d'une classe de fenêtre générique et de classes de fenêtres plus spécialisées. Or une fenêtre spécialisée et avant tout une fenêtre, avec un titre, un style, une largeur, une hauteur. Donc **la fenêtre spécialisée sera construite à partir de la fenêtre générique**. Si j'ai défini des attributs comme width et height et une méthode comme resize pour la fenêtre générique, je dois pouvoir récupérer ces attributs et cette méthode dans ma fenêtre particulière.
La fenêtre spécialisée sera **dérivée** de la fenêtre générique.

En langage Python, si *Base* est le nom d'une classe que l'on veut utiliser comme point de départ pour l'écriture d'une nouvelle classe plus spécialisée, voici la syntaxe à respecter :

```
class maClasse(Base) :

    __init__(self, …) :
        …. définition du constructeur.....

    uneMethode(self, …) :
        ….définition de la méthode....

    une AutreMethode(self, ….) :
```

C'est l'opération d'héritage : La nouvelle classe *maClasse* **hérite** de tous les attributs (c'est à dire toutes les variables membres) et de toutes les fonctions membres définis dans la classe de base.

La nouvelle classe peut ensuite définir de **nouveaux attributs et de nouvelles méthodes** (= fonctions membre) qui viendront s'ajouter aux attributs (=variables) et méthodes hérités. De plus, une fonction membre déjà présente dans la classe de base peut être **réécrite dans la classe dérivée** pour adapter son comportement aux besoins de la nouvelle classe.

Exemple :

Dans l'exemple vu plus haut sur la modélisation de formes géométriques à deux dimensions, nous avons créé la classe *Forme.* Mais celle-ci est bien trop limitée pour être directement utile. Elle fournit une base de travail pour déclarer et définir des formes géométriques réelles. La classe Forme n'est qu'une classe de base à partir de laquelle nous allons créer de nouvelles classes représentant des formes géométriques.

C'est ici que nous allons utiliser l'héritage, par exemple pour modéliser un rectangle. Pour simplifier on considérera que les cotés du rectangle doivent être parallèles aux bords de l'écran.

```
class Rectangle(Forme) :
```

```
    def __init__(self, topLeft, bottomRight) :
        self.poinRef = topLeft
        self.bottomRight = bottomRight

    def translation(self, offsetX, offsetY) :
        self.pointRef.x += offsetX
        self.pointRef.y += offsetY
        self.bottomRight.x += offsetX
        self.bottomRight.y += offsetY

    def rotation(self, angle) :
        …. programme pour la rotation du rectangle ...

    def dessine(self, ) :
        … programme pour l'affichage du rectangle …
```

Si l'on compare les deux premières instructions de la méthode *translation* de la classe dérivée *Rectangle* à celles de la classe de base *Forme*, on constate qu'elles sont identiques. Il est possible, dans les méthodes de la classe dérivée, d'utiliser les méthodes de la classe de base. Pour cela il faut préfixer la méthode avec le nom de la classe. Ici on veut réutiliser, dans la méthode *translation* de la classe dérivée, la méthode t*ranslation* de la classe de base.

```
# ---------- dans la définition de la classe Rectangle ----------------

def translation(self, offsetX, offsetY) :
    Forme.translation(self, offsetX, offsetY)
    self.bottomRight.x += offsetX
    self.bottomRight.y += offsetY
```

Cela permet d'effectuer la translation du point de référence du rectangle grâce à la fonction définie dans la classe de base.

[Inheritance: http://docs.python.org/2.7/tutorial/classes.html#inheritance]

III.2.2.4 Notion de polymorphisme

Le terme polymorphisme en programmation orientée objet correspond à la possibilité d'écrire un algorithme manipulant des objets « génériques », en se basant sur une définition basique de ces objets et de leurs méthodes, alors que, lors de l'exécution de

l'algorithme, ces objets pourront être plus évolués que le modèle de base utilisé pour écrire l'algorithme.

Pour mieux comprendre, reprenons l'exemple précédent des classes de formes géométriques.
Nous avons une classe de base *Forme* à partir de laquelle nous pouvons dériver de nouvelles classes comme *Rectangle*, *Triangle*, *Cercle*. Chacune de ces dernières classes redéfinira les méthodes *translation*, *rotation*, *dessine* pour les adapter à leurs particularités.

Imaginons maintenant que j'écrive un algorithme pour effectuer la translation d'un ensemble de formes et pour les afficher. L'ensemble de formes sera représenté par une liste.

```
x = …
y = ...
lesFormes = []
lesFormes = readFormes()  # lecture des formes, par exemple dans un fichier
for f in lesFormes :
        f.translation(x,y)
        f.dessine()
```

L'algorithme peut s'appliquer à toutes les formes issues de la classe de base *Forme*, chaque forme fournissant ses propres versions des méthodes *translation* et *dessine*. C'est cela qu'on appelle **polymorphisme**.

Cela paraît assez évident avec un langage dynamique comme Python où les variables ne sont pas déclarées avec un type à priori, mais c'est moins intuitif avec les langages à typage strict, comme le C, où le type des variables est vérifié lors de la compilation du programme. Là, il est moins intuitif d'imaginer que l'algorithme défini à partir du type de base d'une classe, s'appliquera en fait aussi à des types dérivés.