Python Initiation

Table des matières

Présentation de Python

Python est un langage interprété de haut niveau, pour lequel la lisibilité du code est un concept de base.

Il utilise un mécanisme de typage dynamique, et est intrinsèquement orienté objet.

Sa large bibliothèque standard en fait un langage utilisable dans de nombreux contextes, du script d'administration à l'application lourde, en passant par les frameworks de développement web.

Plusieurs implémentations du langage existent, la principale étant CPython (implémenté en C).

On peut également citer :

- Jython (java)
- IronPython (.NET)
- PyPy (python)

1. Historique

Guido van Rossum commence le développement de Python en 1989, comme successeur du langage ABC.

Python 2.0 est publié en 2000, sa licence devient libre (GPL). Cette version fournit un ***garbage collector*** et le support d'***unicode***.

Python 3.0 est publié fin 2008. Cette version majeure casse la compatibilité avec les versions précédentes, afin de corriger certains problèmes présents depuis longtemps.

A l'heure actuelle les versions 2 et 3 de Python cohabitent.

2. Comparaison avec d'autres langages

Python allie les avantages de nombreux languages :

- simplicité de développement (comme de nombreux languages de scripts tel que PHP ou Shell)
- propose des outils avancés grâce au typage des données (comme on peut le trouver avec C++ ou Java)
- présente une base très étendue de modules additionnels pour étendre sa bibliothèque standard et des outils pour faciliter le déploiement (comme CPAN avec Perl ou Gem avec Ruby)

3. Installation simple

Le plus simple : les binaires

C'est un élément de base dans toutes les distributions Linux, on le trouve donc par défaut et packagé sur toutes les distributions.

On trouve également des binaires pour Windows, Mac OSX...

Sources pour la cohabitation

Il est toujours possible de compiler les sources sous Linux et d'avoir plusieurs versions de python en parallèle :

```
$ tar -jxvf Python-X.Y.Z.tar.bz2
$ cd Python-X.Y.Z
$ ./configure
# make altinstall
```

La syntaxe de base

Scripts python

Les scripts python dont de simples fichiers texte, utilisant une extension ***.py***.

Un ensemble de fonctionnalités peut être regroupé dans un ***module*** (fichier .py à importer) ou un ***package*** (dossier).

Clarté

La syntaxe de Python se veut claire et explicite. De ce fait, de nombreux outils trouvés sur nombre d'autres langages n'existent pas en Python (pas de until ni de switch/case par exemple).

Le nombre de mots clés du langage est par ce fait restreint.

Blocs

La gestion des blocs en python est basée sur l'indentation, et non sur des caractères ou mots clé spécifiques :

```
if a == 1 :
  # un bloc
  print ("OK")
else :
  # un nouveau bloc
  print ("pas OK")
```

L'indentation peut être faite par des tabulations ou des espaces.

Une mauvaise indentation peut générer une erreur à l'exécution, ou un comportement incorrect !

Variables

Les noms de variables peuvent contenir des lettres (majuscules et minuscules), des chiffres et le caractère souligné (underscore). Ils ne peuvent pas commencer par un chiffre.

Il n'y a pas de caractère d'accès au contenu des variables (à l'inverse du perl ou du php) :

```
# commentaire
a = 8
b = a + 1
_data = "ma chaine"
__version__ = "3.0"
```

Typage des variables

Le typage des variables est dynamique. Le type d'une variable est défini lors de son affectation :

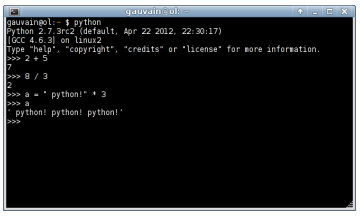
```
>>> a = 8
>>> type(a)
<type 'int'>
>>> a = "chaine"
>>> type(a)
<type 'str'>
```

Une variable doit être définie (affectée) avant d'être utilisée :

```
>>> print(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
>>> b=3 ; print(b)
3
```

1. L'interpréteur

L'interpréteur python permet la manipulation du langage de manière interactive.



Interpréteur Python

L'interpréteur est un outil de test et débuggage essentiel. Il permet de rapidement importer des modules et d'accéder à leur documentation.

La documentation des modules est généralement incluse dans le code lui même (sous forme de ***docstrings***), permettant ainsi d'y accéder facilement depuis la commande help :

```
>>> import os
>>> help(os.system)
```

2. Les différents nombres et leur manipulation

Les entiers

Python 2 possède deux types entiers : ***int*** et ***long***. L'utilisation de l'un ou l'autre est transparente.

Cette distinction n'existe plus pour Python 3, seul le type ***int*** est utilisé.

Les opérateurs de calcul sur les entiers sont ceux habituels :

Opérateurs arithmétiques	`+ - * / % **`
Opérateurs de bit	`& | ^ >> <<`

Attention

En Python 3, la division de 2 entiers génère un flottant.

L'opérateur ***//*** est à utiliser pour la division entière.

```
>>> 8/3 # python 2
2
>>> 8/3 # python 3
2.6666666666666665
>>> 8//3
2
```

Conversion

Il est possible de convertir une donnée en une autre grâce à la fonction ***int()***.

```
>>> b = 3.5
>>> int(b)
3
>>> b = "56"
>>> int(b)
56
```

Cette conversion explicite est obligatoire lors d'opérations sur des types différents :

```
>>> a = 5
>>> b = "3"
>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and
'str'
>>> a + int(b)
8
```

Les flottants (nombres réels)

Le type ***float*** est utilisé pour représenté les nombres à virgule.

Ce type possède les mêmes opérateurs arithmétiques que les entiers.

La fonction de conversion vers un flottant est ***float()***.

Approximation des flottants

Le type flottant ne représente pas les nombres réels de manière exacte, mais par des approximations (mécanique héritée du C et du fonctionnement des processeurs).

```
>>> 1.9*3
5.699999999999999
```

Le module ***decimal*** permet de manipuler les réels de manière exacte.

3. Les chaînes de caractères et leur manipulation

Les chaînes de caractères sont définies par plusieurs moyens :

- entre apostrophes (simples quotes) : `'une chaîne'`
- entre guillemets (double quotes) : `"une chaîne"`
- entre triple apostrophes : `'''une chaîne avec "guillemets".'''`
- entre triple guillemets : `"""une 'autre chaîne'."""`

Le type associé est ***str***, et la fonction de conversion ***str()***.

Les chaînes de caractères en python sont des objets non modifiables. Les outils de manipulation des chaînes retournent systématiquement une nouvelle chaîne de caractères.

Les chaînes sont des séquences, et peuvent être manipulées comme telles (voir la section sur les séquences).

Représentation des chaînes

Les chaînes de caractères représentent une chaîne d'octets, qui doivent être interprétés lors de leur utilisation.

Afin de manipuler les caractères de différents langages écrits, il peut s'avérer nécessaire d'utiliser un encodage supportant tous les caractères : ***unicode***.

Ainsi 2 types de chaînes existent : ***str*** et ***unicode*** :

```
>>> unicode("foo")u'foo'
>>> str("foo")'foo'
>>> unicode("Stéphane")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
```

Le é de Stéphane n'est pas compris, il faudra le traduire :

```
>>> "Stéphane".decode('utf-8')
u'St\xe9phane'
>>> u'St\xe9phane'.encode('utf-8')
'St\xc3\xa9phane'
```

Exemples de manipulation

```
>>> a = "hello"
>>> b = "world"
>>> a + " " + b
'hello world'
```

```
>>> 8*"5"
'55555555'
>>> str(8) + "5"
'85'
```

Substitution de variable

La syntaxe de python ne permet pas la substitution de variable dans une chaîne de caractère. Il est nécessaire de passer par un formatage :

```
>>> s = "%d %s en stock (prix unitaire : %.2f€)\n =>
%.2f€" % (34, "livres", 12.5, 34*12.5)
>>> print(s)
34 livres en stock (prix unitaire : 12.50€)
 => 425.00€
```

La méthode ***format()*** offre une autre méthode de substitution de variable. Cette méthode est préférée pour python 3 :

```
>>> "{0} + {1} = {2}".format(8, 5, 8+5)
'8 + 5 = 13'
>>> "{hello} {world}".format(hello="bonjour",
world="monde")
```

Méthodes du type str

Le type ***str*** fournit un ensemble de méthodes permettant la manipulation des chaînes de caractères. L'ensemble de ces méthodes est disponible via l'aide :

```
>>> help(str)
```

Quelques exemples

```
>>> s = "une ligne"
>>> s.upper() # majuscules
'UNE LIGNE'
>>> s.lower() # minuscules
'une ligne'
>>> s.replace("ligne", "poule") # remplacement
'une poule'
>>> s.find("li") # présence d'une sous chaine
4
>>> s.find("LI") # Python est sensible à la casse
-1
```

4. Comparaisons et type booléen

Python implémente un type ***bool*** (booléen), avec 2 valeurs : ***True*** et ***False***.

Ce type est utilisé dans toutes les structures de test (***if***, ***while***)

Les types primitifs peuvent être utilisés comme booléens :

```
>>> bool(3)
True
>>> bool(0)
False
>>> bool("") # chaine vide
False
>>> bool("foo")
True
```

Opérateurs de comparaison

Les opérateurs de comparaison sont :

`>`, `>=`, `<`, `<=`, `==`, `!=`

Les opérateurs logiques :

`and`, `or`, `not`

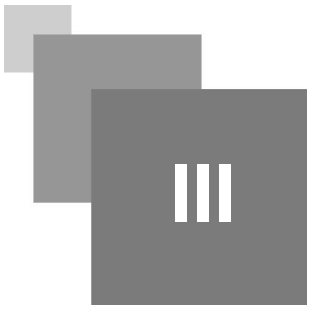
```
>>> a = 1
>>> b = 2
>>> (a == 1) and (b == 2)
True
```

5. Raccourcis

Python fourni quelques opérateurs permettant d'alléger la syntaxe : `+=, -=, *=, /=, %=.`

Ces opérateurs permettent de simplifier la modification d'un variable :

```
a = 4
a += 1 # équivaut à a = a + 1
b = 8.0
b /= a # équivaut à b = b / a
# même principe avec des chaînes de caractères :
s = "hello "
s += "world"
```



Les structures de contrôle

1. La condition if / else

L'exécution conditionnelle d'instructions est faite grâce au mot clé ***if***. Ce mot clé est suivi d'une expression évaluée en tant que booléen.

Si l'expression est vraie, le bloc est exécuté, sinon il ne l'est pas

Le mot clé ***else*** permet de définir un bloc à exécuter si l'expression fournie pas ***if*** est fausse.

```
if True :
  print("OK")
else :
  print("pas OK")
```

Le caractère ***:*** est indispensable après l'expression à tester.

On peut trouver des parenthèses autour du test, mais elles ne sont pas obligatoires.

Exemple avec saisie utilisateur

```
choix = raw_input("Oui ou non (o/n)? ")
if choix == "o" or choix == "O" :
  print("C'est Oui !")
else :
  print("Tant pis...")
```

raw_input va retourner une chaine de caractère contenant le texte tapé par l'utilisateur. Par opposition input va retourner une variable typée par l'élément renseigné par l'utilisateur.

Attention

En Python 3, ***raw_ input*** est renommé ***input***, alors que ***input*** a disparut.

Le mot clé elif permet d'enchaîner plusieurs tests.

Dès qu'un test est vrai, la boucle associée est exécutée, et le reste de la structure de test ignoré.

```
if a == 1 :
  print("Premier")
elif a == 2 :
  print("Second")
elif a == 3 :
  print("Troisième")
else :
  print("Perdu")
```

2. La boucle while

La boucle ***while*** permet d'exécuter un bloc de code tant que la condition spécifiée est vraie :

```
i = 0
while i < 10 :
  i += 1
  print i
print("Bye !")
```

Lorsque i vaut 10, la condition devient fausse, le programme poursuit son exécution.

Les instructions ***break*** et ***continue*** permettent de modifier le comportement de la boucle :

```
i = 0
while i < 1000 :
  i += 1
  if (i % 2) == 1 :
    continue
  print i
  if i == 9 :
    break
```

continue interrompt l'exécution du bloc et passe à l'itération suivante.

break interrompt également l'exécution du bloc, mais sort complétement de la boucle.

3. La boucle for

La boucle *for* exécute un bloc de code pour un ensemble d'éléments donnés.

Les éléments sont fournis sous forme de séquence (liste, tuple, ...), type Python abordé au chapitre suivant.

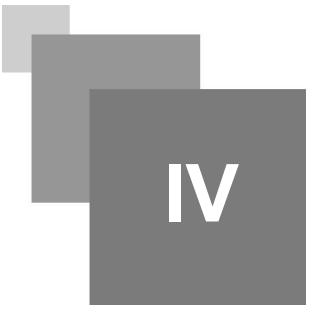
Les chaînes de caractères étant des séquences, il est possible de les parcourir via la boucle *for* :

```
for lettre in "une phrase" :
  print lettre
```

La fonction *range()* permet de générer une séquence d'entiers :

```
for i in range(9):
  print i # de 0 à 8 inclus
for i in range(1, 9):
  print i # de 1 à 8 inclus
for i in range(0, 11, 2):
  print i # de 0 à 10 par pas de 2
```

Les structures de données



Python offre deux types de structures de données en plus des types primitifs :

- les ***séquences***, suite ordonnée d'éléments (pouvant être hétérogènes)
- les ***mappings***, table d'association clé => valeur

1. Les séquences

Les séquences font intervenir 2 notions fondamentales :

- un ordre
- et par conséquent un accès par index

3 types primitifs implémentent les mécanismes de séquence :

- les ***listes***
- les ***tuples***
- les ***chaînes de caractères***

1.1. Les listes

Une liste est définie comme un ensemble ordonné d'éléments. Ses éléments peuvent être de types différents.

Une liste vide peut être créée de 2 manières :

```
>>> l = []
>>> l = list()
```

Il est également possible de créer une liste contenant déjà des objets :

```
>>> l = [1, 3, "texte"]
```

Accès et modification des éléments

Un élément d'une liste peut être récupéré en utilisant son ***index***, le premier objet étant à l'index 0 :

```
>>> l = [1, 3, "texte"]
>>> l[2]
'texte'
>>> l[0]
1
>>> l[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

Une liste possède un nombre fini d'éléments, l'accès à un indice trop grand lèvera une exception.

L'affectation d'un élément se fait avec la même syntaxe :

```
>>> l[2] = "change moi"
>>> l
[1, 3, 'change moi']
```

Il n'est pas possible d'affecter un objet à un indice de liste inexistant :

```
>>> l[3] = 8
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

Il faudra utiliser la méthode ***append()*** de la liste pour la compléter.

Ajout d'éléments

La méthode ***append()*** permet l'ajout d'un élément à la fin de la liste :

```
>>> l.append(8)
>>> l
[1, 3, 'change moi', 8]
```

La méthode ***insert(i, o)*** permet l'insertion de l'objet `o` à l'index `i` :

```
>>> l.insert(1, 'tadaaa')
>>> l
[1, 'tadaaa', 3, 'change moi', 8]
```

Suppression d'éléments

La méthode ***remove(v)*** supprime la première occurrence de `v` dans une liste :

```
>>> l.remove(3)
>>> l
[1, 'tadaaa', 'change moi', 8]
```

La méthode ***pop()*** supprime l'objet à l'index indiqué (ou le dernier élément si rien n'est indiqué). L'objet supprimé est retourné :

```
>>> l.pop()
8
>>> l
[1, 'tadaaa', 'change moi']
>>> l.pop(0)
1
>>> l
['tadaaa', 'change moi']
```

Taille d'une liste

La fonction ***len()*** retourne la taille de la séquence passée en argument :

```
>>> len([1, 3, 8])
3
>>> len("compte moi")
10
```

Autres méthodes utiles

reverse()	inverse la liste (la liste elle même est modifiée !)
index(value)	retourne l'index auquel `value` est trouvé, ou lève une exception
count(value)	retourne le nombre d'occurences de `value` dans la liste

1.2. Les tuples

Un ***tuple*** est un cas particulier de liste.

Un tuple a la principale particularité d'être immuable. Une fois créé, il ne pourra plus être modifié.

Un tuple est représenté entouré de parenthèses, et non de crochets comme pour les listes :

```
>>> a=1 ; b=2 ; c=3
>>> t = (a, b, c)
>>> t
(1, 2, 3)
```

1.3. Les chaînes de caractères

Les chaînes de caractères sont des cas particuliers de tuples, ne contenant que des caractères. Les chaînes de caractères sont donc immuables.

1.4. Parcours d'une séquence

Quelle que soit la séquence, il est possible de la parcourir grâce à une boucle *for* :

```
for element in [1, "text", range(3)] :
  print element
```

L'opérateur *in* peut également être utilisé pour tester la présence d'une valeur dans une liste :

```
>>> l = ['a', 'bc', 'def']
>>> 'a' in l
True
>>> 'foo' in l
False
```

1.5. Les slices

Python fourni une syntaxe permettant l'extraction (et la copie) d'une partie de séquence (on parle de *slice*) :

```
l[idx_debut:idx_fin]
```

Les deux index peuvent être omis.

```
>>> l = [3, 2, 1]
>>> l[0:2]
[3, 2]
>>> l[:2] # depuis le début
[3, 2]
>>> l[2:] # jusqu'à la fin
[1]
>>> l[:] # copie de la liste
[3, 2, 1]
>>> s = "ligne\n"
>>> s[:-1] # du début à l'avant dernier élément
'ligne'
```

2. Les dictionnaires

Les dictionnaires permettent d'associer une clé à une valeur.

Contrairement aux séquences, il n'est pas possible de parcourir un dictionnaire de manière ordonnée.

L'accès aux éléments se fait par l'intermédiaire des clés.

Exemple

```
>>> d = {"toto" : 12, "tutu" : 18} # création
>>> d["toto"] # accès à un élément
12
>>> d["titi"] = 14 # création d'un nouvel élément
>>> len(d) # taille
3
>>> d.pop("toto") # suppression d'un élément
12
```

Parcours d'un dictionnaire

Le parcours d'un dictionnaire peut se faire sur ses clés, ses valeurs, ou sur les deux en même temps (via un tuple) :

- ***d.keys()*** retourne la liste des clés
- ***d.values()*** retourne la liste des valeurs
- ***d.items()*** retourne une liste de tuples (clé, valeur)

L'ordre dans lequel les éléments apparaissent dans une des listes générées est arbitraire.

Mais l'ordre est respecté pour les clés et les valeurs. Ce qui signifie que :

```
d[d.keys()[x]] == d.values()[x]
```

Attention

En python 3, les méthodes précédentes retournent des vues sur les différents éléments.

Ces vues évoluent avec le dictionnaire, contrairement aux séquences retournées en Python 2.

Exemple en Python 3 :

```
>>> d = {"toto" : 12, "tutu" : 18}
>>> v = d.keys()
>>> len(v)
2
>>> d["titi"] = 14
>>> len(v)
3
```

Organisation du code Python

1. Les fonctions

Les fonctions ont 2 rôles essentiels :

- éviter la duplication de code
- modulariser l'application

Une fonction peut recevoir des arguments, et retourner une valeur.

Il est également possible de ne rien recevoir et/ou de ne rien renvoyer.

Définition d'une fonction

Le mot clé ***def*** permet de spécifier la définition d'une fonction.

Suivent le nom de la fonction, ses éventuels arguments entre parenthèses, le caractère **:**, puis le bloc de code de la fonction.

Exemple :

```
def plus_5 (i):
  j = i +5
  return j
```

La fonction s'appelle ***plus_5***, elle attend un argument, et retourne une valeur.

Le mot clé ***return*** est utilisé pour retourner cette valeur.

Une fois cette instruction appelée, le reste du bloc de la fonction n'est pas exécuté.

Appeler une fonction

L'appel d'une fonction est fait en spécifiant son nom suivi des arguments à passer entre parenthèses :

```
>>> a = 8
>>> b = plus_5(a)
>>> b
13
```

Valeur par défaut des arguments

Il est possible de définir des valeurs par défaut pour tout ou partie des arguments.

Les arguments sans valeur par défaut doivent être déclarés avant les autres.

```
def ajoute(a, nb=1) :
  return a + nb
>>> ajoute (8, 2)
10
>>> ajoute (8)
9
```

Si plusieurs arguments ont des valeurs par défaut, il est possible de spécifier explicitement quels arguments sont définis au moment de l'appel :

```
def ajoute(a, nb=1, verbose=False) :
  if verbose :
    print("j'ajoute %d à %d" % (nb, a))
  return a + nb
>>> ajoute (8, 2)
10
>>> ajoute (8, verbose=True)
j'ajoute 1 à 8
9
```

Arguments variables

Il est enfin possible de spécifier plusieurs arguments, dont le nombre et le nom sont inconnus au moment de l'écriture de la fonction.

Ces arguments pourront être nommés, ou non.

2 variables permettront de recevoir les informations :

- un tuple recevra les arguments anonymes
- un dictionnaire recevra les arguments nommés

```
def dummy(a, *args, **kargs) :
  print a
  print args
  print kargs
>>> dummy (1, 4, 5, 6, foo="FOO", bar="BAR")
1
(4, 5, 6)
{'foo': 'FOO', 'bar': 'BAR'}
```

2. Les modules

Les fonctions ayant trait à un même mécanisme sont généralement regroupées en ***modules***.

Ces modules permettent :

- la mise en place d'***espaces de nommage***
- de ne charger en mémoire que les fonctionnalités nécessaires
- de mieux organiser son code

Pour utiliser une fonction définie dans un module, il est donc nécessaire de charger tout ou partie du module.

2.1. Import

Le mot clé *import* permet l'import d'un module :

```
>>> sys.version
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sys' is not defined
>>> import sys
>>> sys.version
'2.7.3rc2 (default, Apr 22 2012, 22:30:17) \n[GCC 4.6.3]'
```

Tous les attributs du module sont alors disponibles dans son espace de nom (*sys* dans l'exemple précédent).

La fonction *dir()* permet de lister l'ensemble des éléments importés avec le module passé en argument.

Le mot clé *from* associé à *import* permet de n'importer qu'une partie d'un module, et de raccourcir la syntaxe.

```
>>> import os
>>> os.path.sep
'/'
>>> from os.path import sep
>>> sep
'/'
>>> from sys import *
>>> version
'2.7.3rc2 (default, Apr 22 2012, 22:30:17) \n[GCC 4.6.3]'
```

Il est également possible de définir des synonymes aux modules afin d'alléger l'écriture :

```
>>> import sqlalchemy as SA
>>> SA.__version__
'0.7.8'
```

2.2. Création d'un module

Un module peut avoir 2 formes :

- un fichier avec extension *.py*
- un dossier contenant un fichier *__init__.py*

Dans les deux cas, l'import pourra se faire si l'élément est présent dans le chemin de python (*sys.path*), ou dans le dossier courant.

Programmation Orientée Objet

1. Les objets

Tous les éléments manipulés par le langage python sont des objets.

Même si l'utilisation de la ***Programmation Orientée Objet*** n'est pas une obligation en Python, cette technique est systématiquement présente.

1.1. Classes, attributs, méthodes

En POO, un objet ne peut être créé que si sa définition a été faite d'abord (de même qu'une fonction ne peut pas être appelée si elle n'a pas été définie).

La définition d'un objet porte le nom de ***classe***. Un objet est une ***instance de classe***.

Chaque instance d'une même classe est un objet unique, mais toutes les instances d'une classe partagent les mêmes caractéristiques.

2 types d'éléments définissent un objet :

- ses ***attributs*** (variables qui lui sont associées, données)
- ses ***méthodes*** (fonctions qui font "travailler" l'objet)

Déclaration d'une classe

Le mot clé ***class*** permet de définir une classe.

Voici un exemple de déclaration simple :

```
class Point(object):
  def set_x(self, x) :
    self.x = x
  def set_y(self, y) :
    self.y = y
  def print(self) :
    print ("%d, %d" % (self.x, self.y))
```

Cette classe définit que toutes ses instances :

- posséderont 2 attributs (***x*** et ***y***)
- posséderont 3 méthodes (***set_x***, ***set_y*** et ***print***)

Le mot clé ***self*** représente l'objet sur lequel les méthodes vont travailler.

Instanciation d'un objet

L'instanciation d'un objet se fait par cette syntaxe :

```
>>> p = Point()
```

Il est alors possible d'accéder aux attributs et méthodes de l'objet :

```
>>> p.set_x(4)
>>> p.set_y(-3)
>>> p.print()
4, -3
```

1.2. Constructeur et destructeur

__init__()

Un constructeur est une méthode qui permet de définir un ensemble d'informations concernant l'objet au moment de son instanciation.

Le constructeur de Python est une méthode nommée ***__init__()***.

L'exemple précédent aurait pu (et dû) être construit comme suit :

```
class Point(object) :
  def __init__(self, x, y) :
    self.x = x
    self.y = y
  def print(self) :
    print ("%d, %d" % (self.x, self.y))
```

__del__()

La méthode ***__del__()*** est le destructeur, permettant de "nettoyer" l'objet avant sa suppression de la mémoire de Python :

```
def __del__(self) :
  self.un_fichier.close()
  self.db.close()
```

1.3. Autres méthodes spéciales

De nombreuses méthodes spéciales peuvent être définies afin d'intégrer au mieux les objets dans le langage.

On peut citer parmi ces méthodes :

- ***__str__()*** pour l'affichage avec print()
- ***__len__()*** pour la taille d'un objet
- ***__add__()*** pour l'ajout d'objets du même type
- ***__contains__()*** pour la syntaxe 'x in y'
- ...

2. Les propriétés

Privé vs Public

La majorité des langages objets classiques proposent différentes portées pour les variables.

Python n'y échappe pas et propose public et privée. Sauf que cette séparation est purement une convention syntaxique, en définissant les attributs et variables privées en les faisant commencer par le caractère _

Principe

Il existe cependant un moyen pour s'assurer qu'un attribut sera utilisé de la manière dont l'auteur de la classe utilisée le désire : les ***propriétés*** (ou ***property***).

Syntaxe

Une propriété sera créée avec la fonction ***property*** à partir d'un attribut, en lui définissant au moins un accesseur privé (_get) et un mutateur privé (_set), qui accèdent eux au réel attribut privé.

```
class Point1D(object) :
  def __init__(self, x) :
    self.x = x
  def _get_x(self) :
    return self._x
  def _set_x(self, x) :
    self._x=x
  x=property(_get_x, _set_x)
```

3. L'héritage

Python supporte l'héritage (y compris l'héritage multiple).

Créer une classe dérivée

La création d'une classe dérivée se fait en spécifiant la classe mère dans la définition :

```
class Point3D(Point) :
  pass
```

La classe `Point3D` hérite ici des attributs et méthodes de la classe `Point`.

Surcharge des méthodes

Les classes dérivées peuvent surcharger les méthodes de leur classe parente :

```
class Point3D(Point) :
  def __init__(self, x, y, z) :
    self.x = x
    self.y = y
    self.z = z
  def print(self) :
    print ("%d, %d, %d" % (self.x, self.y, self.z))
```

Appels des méthodes des classes parents

L'exemple précédent réimplémente entièrement les méthodes.

Si la classe parente est modifiée, la classe dérivée doit l'être aussi.

On perd alors l'intérêt de l'héritage.

Il serait plus judicieux d'appeler le constructeur de la classe parente :

```
class Point3D(Point) :
  def __init__(self, x, y, z) :
    Point.__init__(self, x, y)
    self.z = z
```

Une modification de la classe parente n'aura alors plus d'impact sur la classe dérivée.

4. Conventions

Les développeurs python proposent de respecter certaines règles de syntaxe afin d'augmenter encore plus la lisibilité du code.

PEP8 définit ces règles : *http://www.python.org/dev/peps/pep-0008*

Pour la programmation orientée objet on pourra noter :

- `CamelCase` pour les noms de classes
- `do_something` pour les fonctions (mots séparés par le caractère souligné)
- noms de variables en minuscules
- constantes en majuscules
- `self` représente un objet, `cls` une classe
- les noms sont préfixés par le caractère souligné pour les objets privés (à usage interne)

Les exceptions

Les *exceptions* sont un moyen de contrôler les erreurs pouvant survenir lors de l'exécution d'un programme.

Lorsqu'une fonction ou méthode ne peut fonctionner correctement, elle *lève une exception*.

Lorsqu'une exception est levée, le comportement par défaut de Python est le déclenchement de l'arrêt du programme.

Il faudra donc *attraper les exceptions* pour poursuivre l'exécution du programme, en gérant l'erreur générée.

1. Attraper les exceptions

Une section critique (pouvant générer une exception) est définie dans un bloc *try...except*.

Le bloc *try* est exécuté. Si une exception est levée, le programme passe à l'exécution du bloc *except* :

```
def divise(a, b) :
  return a / b

try :
  c = divise(4, 0)
  print(c)
except :
  print "Division impossible !"
```

Les exceptions sont typées. En effet, plusieurs sources d'erreurs sont envisageables dans l'exemple précédant :

- division par zéro
- erreur de type (division d'un entier par une chaîne de caractères)

Il est donc possible de spécialiser le bloc *except*, en gérant plusieurs types d'erreurs :

```
try :
  c = divise(a, b)
  print c
except ZeroDivisionError :
  print "Division par 0 !"
except TypeError, e :
  print "Erreur de type : %s" % e.message
```

e est un objet de classe TypeError.

Attention

La syntaxe a changé avec Python 3 :

```
except TypeError as e :
  print e.message
```

else et finally

Le mot clé ***else*** permet de définir un bloc exécuté si ***aucune exception n'a été levée*** dans le bloc try.

Le mot clé ***finally*** permet de définir un bloc ***exécuté systématiquement*** après un bloc try...except, même si une exception a été levée.

```
try :
  f = open(fichier)
  travaille(f)
except :
  gere_erreur()
finally :
  f.close()
```

2. Générer ses propres exceptions

Créer un type d'exception consiste à créer une classe dérivée de la classe de base ***Exception***.

Cette classe pourra contenir des informations additionnelles sur l'erreur générée.

Le mot clé ***raise*** permettra de lever une exception :

```
raise TypeError("message à affichier")
```

Exemple

```
class DummyError(Exception) :
  pass

raise DummyError("mon erreur")
```

La bibliothèque standard

VIII

Le bibliothèque standard fournit de nombreux modules dans des domaines d'application divers (système, réseau, expressions régulières, sérialisation, ...).

1. os et sys

Les modules ***os*** et ***sys*** permettent une interaction avec le système d'exploitation de manière portable.

os

Le module ***os*** fournit l'accès aux principaux appels système permettant la gestion des processus et fichiers.

Liste (non exhaustive) de méthodes :

`chdir(), access(), fork(), dup2(), la famille exec(), getuid(), getgid(), mkdir(), rename() , ...`

Le module ***os.path*** est spécialisé sur la gestion des chemins. Il offre plusieurs méthodes de test et de manipulation :

`basename(), join(), exists(), walk(), ...`

```
>>> os.path.isfile("/etc/passwd")
True
>>> os.path.isdir("/etc")
True
>>> os.path.join("foo", "bar", "baz")
'foo/bar/baz' # ou 'foo\bar\baz' sous Windows
```

os.path.walk permet de parcourir un dossier récursivement :

```
def f(arg, dirname, fnames) :
  for file in fnames :
    print "%s/%s" % (dirname, file)

os.path.walk("/etc", f, None)
```

sys

Le module sys donne accès à des méthodes liées au système Python :

- `sys.argv` : liste des arguments passés au script
- `sys.path` : liste des dossiers contenant des modules Python
- `sys.exit()` : sortie du script
- `sys.stdin, sys.stdout, sys.stderr` : objets de type `file` ouverts sur l'entrée et les sorties standards
- ...

2. Gestion des fichiers

La gestion des fichiers en Python passe par la manipulation d'objets de type ***file***.

Ces objets sont créés grâce à la fonction ***open*** :

```
>>> f = open("/etc/passwd")
>>> type(f)
<type 'file'>
```

Le premier argument est le fichier à ouvrir. Un second argument (optionnel) définit le mode d'ouverture :

"r"	ouverture en lecture
"w"	ouverture en écriture
"a"	ouverture en ajout
"rb"	ouverture en mode binaire
"w+"	ouverture en lecture/écriture

Attention, un fichier ouvert en écriture ("w") est ***réinitialisé systématiquement !***

Lecture/écriture

Un ensemble de méthodes permet la lecture/écriture dans le fichier :

`read()`	lit le contenu du fichier, ou le nombre d'octets spécifié en argument
`readline()`	lit une ligne du fichier
`readlines()`	stocke les lignes du fichier dans une liste
`write(data)`	écrit `data` dans le fichier
`writelines(liste)`	écrit la liste de lignes dans le fichier

Fermeture

Il est important de fermer un fichier lorsqu'il n'est plus utilisé afin de libérer des ressources :

```
f.close()
```

3. Sérialisation

Les modules ***Pickle*** et ***cPickle*** (plus efficace) permettent de sérialiser les objets Python.

La sérialisation permet de facilement stocker et restaurer des données, sans passer par un formatage intermédiaire.

Exemple d'utilisation

```
import cPickle
o = MonObjet()
o.do_something()
s = cPickle.dumps(o, protocol=2) # format binaire
save_in_db(s)
r = load_from_db()
oo = cPickle.loads(r)
```

4. Expressions régulières

Le module ***re*** permet la gestion des expressions régulières étendues.

Ce module permet la compilation d'expressions régulières, afin d'optimiser les recherches.

Captures et remplacements sont supportés.

Compilation d'une expression régulière

La compilation d'une E.R. permet une réutilisation efficace.

```
>>> import re
>>> regexp = re.compile(r"^message")
>>> type(regexp)
<type '_sre.SRE_Pattern'>
```

La chaîne de caractère spécifiant l'expression régulière est préfixée par ***r*** pour indiquer à Python qu'il s'agit d'une chaîne brute (*raw*). Les caractères spéciaux perdent leur traitement particulier.

Options de compilation

Plusieurs attributs permettent de modifier le comportement de l'expression régulière.

- ***re.I*** : permet d'ignorer la casse
- ***re.M*** : permet de gérer les lignes mutiples

```
regexp = re.compile(r"^message", re.I|re.M)
```

Recherche

La correspondance peut se faire grâce à deux méthodes :

- ***`match()`*** cherche une correspondance *en début de chaîne*
- ***`search()`*** cherche une correspondance *dans toute la chaîne*

```
>>> regexp = re.compile("message")
>>> regexp.match("le message d'accueil") # pas de match,
None
>>> regexp.search("le message d'accueil") # match
<_sre.SRE_Match object at 0x1ed3b28>
```

Si la correspondance échoue, `None` est retourné. Sinon un objet de type `SRE_Match` est retourné.

Captures

L'objet retourné par `search()` ou `match()` contient les groupes capturés :

```
>>> regexp = re.compile("^(.*) (.*)$")
>>> m = regexp.search("Maurice Moss")
>>> m.groups()
('Maurice', 'Moss')
>>> m.group(0)
'Maurice Moss'
>>> m.group(2)
'Moss'
```

Remplacements

La méthode ***`sub()`*** permet le remplacement :

```
>>> regexp.sub(r"\2 or \1", "Maurice and Roy")
'Roy or Maurice'
```

La manipulation du CSV

Qu'est ce que le CSV

Un des formats les plus répandus pour gérer les données est le CSV (Comma Separated Values).
Il se caractérise par des fichiers textes où chaque ligne va être formé de la même sorte :

```
Nom, Prénom, Age
nom1, prenom1,age1
nom2, prenom2,age2
```

C'est un format qui est capable d'être géré par la plupart des tableurs et des bases de données.

Un module dédié

Un module python de la bibliothèque standard permet de manipuler du csv : le module ***csv***
Il faudra donc réaliser pour l'utiliser :

```
import csv
```

Dès lors nous pouvons utiliser les objets contenus à savoir writer et reader.

L'objet writer

C'est l'objet à utiliser lorsqu'on veut écrire un fichier csv. Sa syntaxe est :

```
obj = writer(fichier [, dialect='excel'], [options])
```

- fichier étant un handler de fichier (ouvert par file)
- dialect permettant de spécifier si on veut renforcer une comptatibilité avec un outil en particulier, par défaut excel
- options permet de spécifier de options dont :
    - delimiter, qui est le séparateur à utiliser entre les champs, par défaut ','
    - lineterminator, qui est le caractère utilisé en fin de ligne, par défaut '\r\n'

2 méthodes principales sont présentes dans l'objet writer :

- writerow(ligne)
- writerows(listelignes)

avec ligne qui sera une séquence contenant les éléments à écrire dans une ligne du fichier csv

```
import csv
champs=("Nom","Prenom","Age")
lecsv=csv.writer(file("fic.csv","w"),delimiter=";")
lecsv.writerow(champs)
```

L'objet reader

C'est l'objet à utiliser lorsqu'on veut lire un fichier csv. Sa syntaxe est :

```
obj = reader(fichier [, dialect='excel'], [options])
```

- fichier étant un handler de fichier (ouvert par file)
- dialect permettant de spécifier si on veut renforcer une comptatibilité avec un outil en particulier, par défaut excel
- options permet de spécifier de options dont :
    - delimiter, qui est le séparateur à utiliser entre les champs, par défaut ','
    - lineterminator, qui est le caractère utilisé en fin de ligne, par défaut '\r\n'

Dès lors il est possible d'itérer sur l'objet retourné par reader, ou d'utiliser la méthode next() pour passer à la ligne suivante.

```
import csv
lecsv=csv.reader(file("fic.csv","w"),delimiter=";")
for ligne in lecsv
  print ligne
```

Gestion du XML

Présentation de XML

Le XMl est un language structuré utilisé pour le stockage d'informations.

C'est un des formats de fichiers les plus utilisés car :

- portable et utilisable sur tous les systèmes
- lisible simplement (non binaire)

Exemple de fichier XML

```
<library>
  <book id="1">
    <author>Stephen King</author>
    <title>Carrie</title>
  </book>
  <book id="2">
    <author>J.K. Rowling</author>
    <title>Harry Potter</title>
  </book>
</library>
```

ElementTree

Python fournit en standard une méthode d'accès aux éléments XML nommée ***ElementTree***. Chaque élément de l'arbre XML est représenté par un objet python.

Le module python gérant cette mécanique est ***xml.etree.ElementTree***. Mais le project ***lxml*** réimplémente ce module en améliorant ses performances.

1. Lecture de fichier XML

La première étape pour la lecture d'un fichier XML est son ***parsing***. Le parser va traduire le fichier XML en objets python :

```
from lxml import etree
doc = etree.parse('file.xml')
```

La variable doc contient une représentation du ***DOM (Document Object Model)***.

Il devient alors possible de parcourir l'arbre, et d'y faire des recherches :

```
# récupération de l'élément racine
root = doc.getroot()
# Affichage du tag et des attributs de la racine
print root.tag, root.attrib
# accès au premier fils
child = root[0]
# affichage de son tag et de sa valeur
print child.tag, child.text
# recherches des tags 'book'
root.findall('book')
```

2. Ecriture/Modification d'un fichier

Un élément de l'arbre peut être modifié directement via l'objet ***Element*** qui le représente :

```
child = root[0]
author = child.find('author')
author.text = 'King Stephen'
print etree.tostring(doc)
```

Un élément peut également être créé de toute pièce :

```
book = etree.SubElement(root, 'book', {'id':'3'})
author = etree.SubElement(book, 'author')
author.text = 'Someone'
title = etree.SubElement(book, 'title')
title.text = 'some title'
print etree.tostring(doc)
```

La méthode ***write*** permet l'écriture dans un fichier :

```
outFile = open('file.xml', 'w')
doc.write(outFile)
```

Gestion du SQL

1. La spécification DB-API

L'accès à un moteur de base de données diffère selon le moteur utilisé.

Les APIs sont généralement différentes et complexifient le code nécessaire pour le support de plusieurs moteurs.

Les développeurs Python ont proposé de normaliser les APIs pour l'accès aux différents moteurs de base de données dans la spécification DB-API (PEP 249).

Cette spécification est accessible en ligne : *http://www.python.org/dev/peps/pep-0249/*

Cette spécification est respectée par de nombreux modules :

- SQLite
- MySQLDB
- PyGreSQL
- ...

2. Connexion et manipulation de bases de données

Création d'une connexion

```
import sqlite3 # ou autre module
conn = sqlite3.connect("storage.db")
```

Requête d'écriture

```
q = "CREATE TABLE log (timestamp TIMESTAMP DEFAULT
CURRENT_TIMESTAMP, log STRING)"
conn.execute(q)
q = "INSERT INTO log (log) VALUES ('message 1')"
conn.execute(q)
conn.commit()
```

Requête de recherche

Les requêtes de recherche font intervenir la notion de curseur :

```
q = "SELECT * FROM log WHERE log LIKE 'message %' LIMIT 2"
cursor = conn.execute(q)
rows = cursor.fetchall() # liste de tuples
print len(rows) # 2
```

L'accès aux colonnes par leur nom peut se faire en demandant au moteur d'utiliser un outil de parcours du curseur différent de celui par défaut :

```
conn.row_factory = sqlite3.Row
# requête faite ici
for row in rows :
  print ("%s : %s" % (row['timestamp'], row['log']))
```

Les requêtes paramétrées

Les requêtes paramétrées permettent :

- de sécuriser les données passées dans la requête (échappement)
- d'exécuter efficacement plusieurs requêtes du même format

```
l = [("log 1",), ("log 2",), ("log 3", )]
conn.executemany("INSERT INTO log (log) VALUES (?)", l)
conn.commit()
```

3. Présentation et Utilisation de l'ORM SQLAlchemy

La gestion des données en base de données peut s'avérer fastidieuse.

Un ***ORM*** (*Object-relational mapping*) permet d'établir un lien direct entre classes d'objets et tables de base de données.

Le module ***SQLAlchemy*** implémente cette mécanique pour Python.

Initialisation d'une connexion

SQLAlchemy nécessite l'initialisation de quelques éléments pour la gestion de la base de données :

- le moteur (*engine*) : la connexion à la base de données
- une session permettant l'ajout/modification d'objets
- une base relationnelle permettant la relation entre tables et objets

```
from sqlalchemy import create_engine
from sqlalchemy.orm import create_session
from sqlalchemy.ext.declarative import declarative_base

base = declarative_base()
engine = create_engine('sqlite:////tmp/test.db')
session = create_session(engine)
```

Définition des objets

Tous les objets héritant de **`base`** seront considérés comme associés à des tables :

```
from sqlalchemy import Column, String, Integer, ForeignKey

class User(base) :
  __tablename__ = "users"
  id = Column(Integer, primary_key=True)
  name = Column(String(50))

class Address(base) :
  __tablename__ = "addresses"
  email = Column(String(50), primary_key=True)
  user_id = Column(Integer, ForeignKey('users.id'))
```

Une fois les classes définies, il est possible de créer les tables et de commencer à les remplir :

```
base.metadata.create_all(engine)

u1 = User(name="Fred")
u2 = User(name="Barney")
session.add(u1)
session.add(u2)
session.flush()

a1 = Address(email="fred@flintstone.net", user_id=u1.id)
a2 = Address(email="barney@rubble.org", user_id=u2.id)
session.add_all([a1, a2])
session.flush()
```

Relations entre objets

La classe `User` peut être améliorée pour créer une référence vers les objets `Address` pour lesquels `user_id` correspond à l'`id` de l'utilisateur.

```
from sqlalchemy.orm import relationship, backref

class User(base) :
  # même déclaration que précédemment plus :
  addresses = relationship("Address", backref="user")
```

La création des utilisateurs/adresses en devient d'autant plus simple :

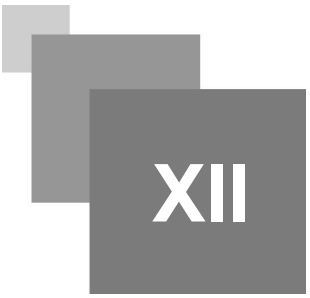
```
u1 = User(name="Fred")
u1.addresses.append(Address(email="fred@flintstone.net"))
session.add(u1)
session.flush()
```

Recherches

Les requêtes de recherche retournent des objets ou des listes d'objets, pour lesquels les relations avec d'autres tables/objets sont déjà établies :

```
u = session.query(User).filter_by(name="Fred").one()
print (u.addresses[0].email) # affiche
'fred@flintstone.net'
```

Bonnes pratiques



1. Validateurs de code

Plusieurs outils permettent de détecter certains potentiels problèmes dans le code, et dans la syntaxe :

- *pylint*
- *pychecker*
- *pep8*

pylint et pychecker sont plus orientés analyse du code que pep8.

2. Virtualenv

virtualenv permet de créer des environnements python isolés, permettant de facilement installer modules et packages dans des versions spécifiques.

Cette méthode permet d'éviter les conflits avec la version de python installée sur le système.

La création d'un environnement est faite grâce à la commande `virtualenv` :

```
mkdir env
virtualenv env/
```

Cet environnement peut être mise en place en sourçant un script shell :

```
cd env/
. bin/activate
python
```

L'environnement offert par défaut est minimal. Seul python et 2 outils sont disponibles :

- pip
- easy_install

Ces deux outils permettent d'installer des modules et packages :

```
easy_install pysmbc
easy_install django
```

3. Tests unitaires

Python fournit par défaut le module ***unittest*** permettant de simplifier l'écriture de tests unitaires.

Les classes de test sont définies en héritant de `unittest.TestCase` :

```
import unittest
class TestNumbers(unittest.TestCase):
  def test_addition(self):
    self.assertEqual(8, 6 + 2)
unittest.main()
```

Toutes les méthodes dont le nom commence par `test` sont exécutées et utilisent des variantes de la méthode `assert()` pour s'assurer que le test a fonctionné.

En cas d'erreur une exception est levée :

```
FAIL: test_addition (__main__.TestNumbers)
----------------------------------------
Traceback (most recent call last):
  File "test.py", line 23, in test_addition
    self.assertEqual(8, 6 + 3)
AssertionError: 8 != 9
```