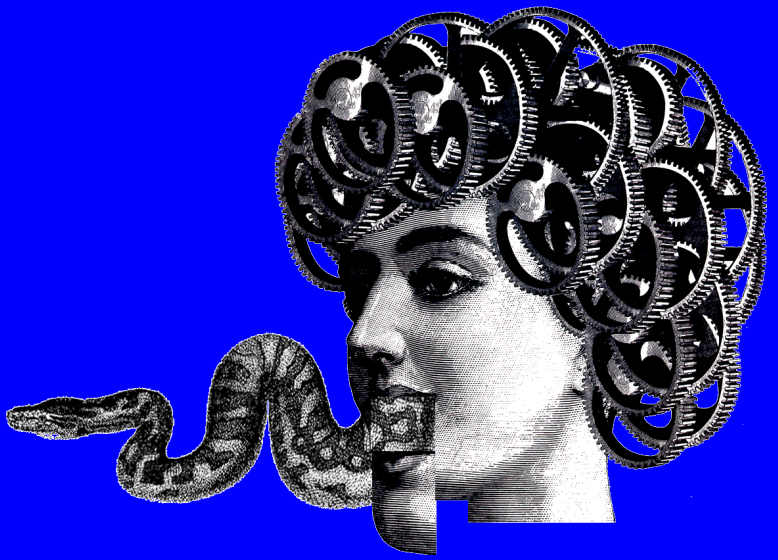
Université de Limoges
FACULTÉ DES SCIENCES ET TECHNIQUES



Mon'tit Python

P-F. Bonnefoi

Version du 30 août 2011

Table des matières

Il est :

⋆ **portable**, disponible sous toutes les plate-formes (de Unix à Windows) ;

⋆ **simple**, avec une syntaxe claire, privilégiant la lisibilité, libérée de celle de C/C++ ;

⋆ **riche**. Il incorpore de nombreuses possibilités de langage

- ◇ tiré de la **programmation impérative** : *structure de contrôle, manipulation de nombres comme les flottants, doubles, complexe, de structures complexes comme les tableaux, les dictionnaires, etc.*
- ◇ *tiré des langages de script : accès au système, manipulation de processus, de l'arborescence fichier, d'expressions rationnelles, etc.*
- ◇ tiré de la **programmation fonctionnelle** : *les fonctions sont dites « fonction de première classe », car elles peuvent être fournies comme argument d'une autre fonction, il dispose aussi de lambda expression, de générateur etc*.
- ◇ tiré de la **programmation orienté objet** : *définition de classe, héritage multiple, introspection (consultation du type, des méthodes proposées), ajout/retrait dynamique de classes, de méthode, compilation dynamique de code, délégation ("duck typing"), passivation/activation, surcharge d'opérateurs, etc*.

Il est :

- *dynamique* : il n'est pas nécessaire de déclarer le type d'une variable dans le source. Le type est associé lors de l'exécution du programme ;
- *fortement typé* : les types sont toujours appliqués (un entier ne peut être considéré comme une chaîne sans conversion explicite, une variable possède un type lors de son affectation).
- compilé/interprété à la manière de Java. Le source est compilé en bytecode (pouvant être sauvegardé) puis exécuté sur une machine virtuelle.

Il dispose d'une gestion automatique de la mémoire ("garbage collector").

Il dispose de nombreuses bibliothèques : interface graphique (`TkInter`), développement Web (le serveur d'application ZOPE, gestion de document avec Plone par exemple), inter-opérabilité avec des BDs, des middlewares ou intergiciels objets(SOAP/COM/CORBA/.NET), d'analyse réseau (SCAPY), manipulation d'XML, *etc*.
Il existe même des compilateurs vers C, CPython, vers la machine virtuelle Java (Jython), vers .NET (IronPython) !

Il est utilisé comme langage de script dans PaintShopPro, Blender3d, Autocad, Labview, *etc*.

Disponibilité et documentation

Sur tout Unix, Python est intégré.

Sous la ligne de commande (*shell*), il suffit de lancer la commande "python" pour passer en mode interactif : on peut entrer du code et en demander l'exécution, utiliser les fonctions intégrées (*builtins*), charger des bibliothèques *etc*

Sous ce mode interactif, il est possible d'obtenir de la documentation en appelant la fonction `help()`, puis en entrant l'identifiant de la fonction ou de la méthode.

La documentation complète du langage est disponible sur le réseau à `http://docs.python.org/`.

Ecriture de code et exécution

L'extension par défaut d'un source Python est ".py". Pour exécuter un source python (compilation et exécution sont simultanées), il existe deux méthodes :

- en appelant l'interprète de l'extérieur : `python mon_source.py`
- en appelant l'interprète de l'intérieur :
  - ◇ on rend le source exécutable, comme un script :`chmod +x mon_source.py`
  - ◇ on met **en première ligne du source** la ligne `#!/usr/bin/python`
  - ◇ on lance directement : `mon_source.py`

Les instructions

Les commentaires vont du caractère # jusqu'à la fin de la ligne.

Il n'existe pas de commentaire en bloc comme en C (/ ... */).*

Chaque instruction s'écrit sur un ligne, il n'y a pas de séparateur d'instruction. *Si une ligne est trop grande, le caractère \permet de passer à la ligne suivante*.

Les blocs d'instructions

Les blocs d'instruction sont matérialisés par des indentations (plus de { et } !).

```
1  #!/usr/bin/python
2  # coding= latin1
3  # les modules utilises
4  import sys, socket
5  # le source utilisateur
6  if (a == 1) :
7     # sous bloc
8     # indente (1 ou 4 espaces)
```

Le caractère : sert à introduire les blocs.

La syntaxe est allégée, facile à lire et agréable (*si si !*).

La ligne 2, `# coding= latin1`*, permet d'utiliser des accents dans le source Python*.

Une variable doit exister avant d'être référencée dans le programme.

Il faut « l'instancier » avant de s'en servir, sinon il y aura une erreur (*une **exception** est levée*).

```
1 print a # provoque une erreur car a n'existe pas
```

```
1 a = 'bonjour'
2 print a # fonctionne car a est définie
```

La variable est une référence vers une entité du langage

```
1 a = 'entite chaine de caracteres'
2 b = a
```

les variables a *et* b *font références à la même chaîne de caractères.*

Une variable ne référençant rien, a pour valeur `None`.

Il n'existe pas de constante en Python (*pour signifier une constante, on utilise un nom tout en majuscule*).

Choix du nom des variables

- Python est **sensible à la casse**, il fait la différence entre minuscules et majuscules.
- Les noms des variables doivent être différents des mots réservés du langage.

Les mots réservés « *Less is more !* »

and	del	for	is	raise
assert	elif	from	lambda	return
break	else	global	not	try
class	except	if	or	while
continue	exec	import	pass	
def	finally	in	print	

5 Les valeurs et types de base

Il existe des valeurs prédéfinies :

`True`	valeur booléenne vraie
`False`	valeur booléenne vide
`None`	objet vide retourné par certaines méthodes

Python interprète tout ce qui **n'est pas faux à vrai**.

Est considéré comme faux :

`0 ou 0.0`	la valeur 0
`''`	chaîne vide
`""`	chaîne vide
`()`	*séquence* ou *tuple* vide
`[]`	*séquence* ou liste vide
`{}`	dictionnaire vide

Et les pointeurs ?

Il n'existe pas de pointeur en Python : tous les éléments étant manipulés par référence, il n'y a donc pas besoin de pointeurs explicites !

- ◇ Quand deux variables référencent la même donnée, on parle « d'alias ».
- ◇ On peut obtenir l'adresse d'une donnée (par exemple pour comparaison) avec la fonction `id()`.

Les séquences d'instructions

Une ligne contient une seule instruction. Mais il est possible de mettre plusieurs instructions sur une même ligne en les séparant par des ; (syntaxe déconseillée).

```
1  a = 1; b = 2; c = a*b
```

Les conditions

Faire toujours attention aux tabulations !

```
1  if <test1> :
2     <instructions1>
3  elif <test2>:
4     <instructions2>
5  else:
6     <instructions3>
```

Lorsqu'une seule instruction compose la condition, il est possible de l'écrire en une seule ligne :

```
1  if a > 3: b = 3 * a
```

Les itérations

La boucle `while` dépend d'une condition.

```
while <test>:
    <instructions1>
else :
    <instructions2>
```

Les ruptures de contrôle

`continue`	continue directement à la prochaine itération de la boucle
`break`	sort de la boucle courante (la plus imbriquée)
`pass`	instruction vide (*ne rien faire*)

Le `else` de la structure de contrôle n'est exécuté que si la boucle n'a pas été interrompue par un `break`.

Boucle infinie

Il est souvent pratique d'utiliser une boucle `while` *infinie* (dont la condition est toujours vraie), et d'utiliser les ruptures de séquences.

```
while 1:
    if <condition> : break
```

7 Les opérateurs

Logique

`or`	OU logique
`and`	ET logique
`not`	négation logique

Binaires (bit à bit)

`|`	OU bits à bits
`^`	OU exclusif
`&`	ET
`«`	décalage à gauche
`»`	décalage à droite

Comparaison

`<,>,<=,>=,==,!=` *inférieur, sup., inférieur ou égale, sup. ou égale, égale, différent*

`is, is not` *comparaison d'identité (même objet en mémoire)*

```
1 c1= 'toto'
2 c2 = 'toto'
3 print c1 is c2, c1 == c2 # teste l'identité et teste le contenu
```

Arithmétique

`+, -, *, /,//,%` *addition, soustraction, multiplication, division, division entière, modulo*

`+=,-=,...` *opération + affectation de la valeur modifiée*

Python v3

L'opérateur <> est remplacé définitivement par `!=`.

L'opérateur `/` retourne **toujours un flottant**, et `//` est utilisé pour la division entière.

8 La gestion des caractères

Il n'existe pas de type caractère mais seulement des chaînes contenant un caractère unique.

Une chaîne est délimitée par des ' ou des " ce qui permet d'en utiliser dans une chaîne :

```
1 le_caractere = 'c'
2 a = "une chaine avec des 'quotes'" # ou 'une chaine avec des "doubles quotes"'
3 print len(a) # retourne 28
```

La fonction `len()` permet d'obtenir la longueur d'une chaîne. Il est possible d'écrire une chaîne contenant plusieurs lignes sans utiliser le caractère `'\n'`, en l'entourant de 3 guillemets :

```
1 texte ="""   premiere ligne
2       deuxieme ligne"""
```

Pour pouvoir utiliser le caractère d'échappement dans une chaîne il faut la faire précéder de `r` (pour *raw*) :

```
1 une_chaine = r'pour passer à la ligne il faut utiliser \n dans une chaine'
```

En particulier, ce sera important lors de l'entrée d'expressions régulières.

Concaténation

Il est possible de concaténer deux chaines de caractères avec l'opérateur + :

```
1 a = "ma chaine"+" complete"
```

9 Les chaînes de caractères

Il est possible d'insérer le contenu d'une variable dans une chaîne de car. à l'aide du %.

```
1  a = 120
2  b = 'La valeur est %d' % a      # b contient la chaîne 'La valeur est 120'
```

Les caractères de formatage sont :

- `%s` chaîne de caractères, en fait récupère le résultat de la commande `str()`
- `%f` valeur flottante, par ex. `%.2f` pour indiquer 2 chiffres après la virgule
- `%d` un entier
- `%x` entier sous forme hexadécimal

Les chaînes sont des objets

Elles proposent différentes méthodes :

- `rstrip` supprime les caractères en fin de chaîne (par ex. le retour à la ligne)
  Exemple : `chaine.rstrip('\n ')`
- `upper` passe en majuscule
  Exemple : `chaine.upper()`
- `splitlines` décompose une chaîne suivant les lignes et retourne une liste de « lignes », sous forme d'une liste de chaîne de caractères.
- *etc.*

Ces listes peuvent contenir n'importe quel type de données.

Il existe deux types de listes :

- celles qui ne peuvent être modifiées, appelées *tuples* ;
- les autres, qui sont modifiables, appelées simplement liste !

Les tuples

Il sont notés sous forme d'éléments entre parenthèses séparés par des virgules.

```
1 a = ('un', 2, 'trois')
```

Une liste d'un seul élément correspond à l'élément lui-même

La fonction `len()` renvoie le nombre d'éléments de la liste.

Les listes modifiables

Elles sont notées sous forme d'éléments entre crochets séparés par des virgules.

Elles correspondent à des *objets* contrairement aux tuples.

```
1 a = [10, 'trois', 40]
```

Les méthodes sont :

`append(e)`	ajoute un élément e	`pop()`	enlève le dernier élément
`pop(i)`	retire le $i^{ème}$ élément	`extend`	concaténe deux listes
`sort`	trie les éléments	`reverse`	inverse l'ordre des éléments
`index(e)`	retourne la position de l'élément e		

10.1 Les listes — Exemples d'utilisation

```
1  a = [1, 'deux', 3]
2  a.append('quatre')
3  print a
```

```
[1, 'deux', 3, 'quatre']
```

```
4  element = a.pop()
5  print element, a
```

```
quatre [1, 'deux', 3]
```

```
6  a.sort()
7  print a
```

```
[1, 3, 'deux']
```

```
8  a.reverse()
9  print a
```

```
['deux', 3, 1]
```

```
10 print a.pop(0)
```

```
deux
```

10.2 Les listes — Utilisation comme « pile » et « file »

Pour une approche « algorithmique » de la programmation, il est intéressant de pouvoir disposer des structures particulières que sont les **piles** et **files**.

La pile

empiler	`ma_pile.append(element)`
dépiler	`element = ma_pile.pop()`

```
>> ma_pile = []
>> ma_pile.append('sommet')
>> ma_pile
['sommet']
>> element = ma_pile.pop()
>> element
'sommet'
```

La file

enfiler	`ma_file.append(element)`
defiler	`element = ma_file.pop(0)`

```
>> ma_file = []
>> ma_file.append('premier')
>> ma_file.append('second')
>> element = ma_file.pop(0)
>> element
'premier'
```

Attention

Si « element » est une liste, alors il ne faut pas utiliser la méthode `append` mais `extend`.

Affectation multiples

Il est possible d'affecter à une liste de variables, une liste de valeurs :

```
1 (a, b, c) = (10, 20, 30)
2 print a, b, c
```

Les parenthèses ne sont pas nécessaires s'il n'y a pas d'ambiguïté.

```
1 a, b, c = 10, 20, 30
2 print a, b, c
```

```
10 20 30
```

En particulier, se sera utile pour les fonctions retournant plusieurs valeurs.

Opérateur d'appartenance

L'opérateur `in` permet de savoir si un élément est présent dans une liste.

```
1 'a' in ['a', 'b', 'c']
```

```
True
```

10.4 L'accès aux éléments d'une liste ou d'un tuple

Il est possible de parcourir les éléments d'une liste à l'aide de `for` :

```
1 for un_element in une_sequence:
2     print un_element
```

Il est possible de les considérer comme des vecteurs :

⋆ on peut y accéder à l'aide d'un indice positif à partir de zéro ou bien *négatif* (pour partir de la fin)

```
1 ma_sequence[0] # le premier element
2 ma_sequence[-2] # l'avant dernier element
```

Il est possible d'extraire une « sous-liste » :

⋆ une tranche qui permet de récupérer une sous-liste

```
1 ma_sequence[1:4]   # du deuxieme element au 4ieme element
2                    # ou de l'indice 1 au 4 (4 non compris)
3 ma_sequence[3:]      # de l'indice 3 à la fin de la séquence
```

Il est possible de créer une liste contenant des entiers d'un intervalle :

```
1 l = range(1,4)    # de 1 à 4 non compris
2 m = range(0,10,2) # de 0 à 10 non compris par pas de 2
3 print l,m
```

```
[1, 2, 3] [0, 2, 4, 6, 8]
```

D'où le *fameux accès indicé*, commun au C, C++ ou Java :

```
1 for valeur in range(0,5) :
2     print vecteur[valeur]
```

10.5 Des listes comme tableau à 2 dimensions

Il n'existe pas de tableau à deux dimensions en Python comme dans d'autres langages de programmation.

— Un tableau à une dimension correspond à une liste.

— Un tableau à deux dimensions correspond à une liste de liste.

Création et utilisation du tableau à deux dimensions

```
1  nb_lignes = 5
2  nb_colonnes = 4
3  tableau2D = []
4  for i in xrange(0, nb_lignes):
5      tableau2D.append([])
6      for j in xrange(0, nb_colonnes):
7          tableau2D[i].append(0)
8  tableau2D[2][3] = 'a'
9  tableau2D[4][3] = 'b'
10 print tableau2D
```

◇ *La fonction* `xrange()` *fonctionne comme la fonction* `range()` *mais elle est plus efficace dans ce contexte d'utilisation.*

◇ *L'accès aux différentes cases du tableau se fait suivant chaque dimension à partir de la position 0, suivant la notation :* `tableau2D[ligne][colonne]`

Ce qui donne :

```
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 'a'], [0, 0, 0, 0], [0, 0, 0, 'b']]
```

Il est possible de généraliser à des tableaux de dimensions supérieures.

10.6 Un accès par indice aux éléments d'une chaîne

Et le rapport entre une liste et une chaîne de caractères ?

Elles bénéficient de l'accès par indice, par tranche et du parcours avec `for` :

```
1 a = 'une_chaine'
2 b = a[4:7] # b reçoit 'cha'
```

Pour pouvoir modifier une chaîne de caractère, il n'est pas possible d'utiliser l'accès par indice :

```
1 a = 'le voiture'
2 a[1] = 'a'
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Il faut d'abord convertir la chaîne de caractères en liste, avec la fonction `list()` :

```
1 a = list('le voiture')
2 a[1] = 'a'
```

Puis, recomposer la chaîne à partir de la liste de ses caractères :

```
1 b = ''.join(a)
2 print b
```

```
la voiture
```

11 Les dictionnaires

Ces objets permettent de conserver une **association** entre une clé et une valeur.

Ce sont des *tables de hachage* pour un accès rapide aux données :

- ◇ *La clé comme la valeur peuvent être de n'importe quel type **non modifiable**.*
- ◇ *La fonction `len()` retourne le nombre d'associations du dictionnaire.*

Liste des opérations du dictionnaire

Initialisation	`dico = {}`
définition	`dico = {'un': 1, 'deux' : 2}`
accès	`b = dico['un'] # recupere 1`
interrogation	`if dico.has_key('trois') :`
	`if 'trois' in dict:`
ajout/modification	`dico['un'] = 1.0`
suppression	`del dico['deux']`
récupère la liste des clés	`les_cles = dico.keys()`
récupère la liste des valeurs	`les_valeurs = dico.values()`

Afficher le contenu d'un dictionnaire

```
1  for cle in mon_dico.keys():
2      print "Association ", cle, " avec ", mon_dico[cle]
```

Un module regroupe un ensemble cohérent de fonctions, classes objets, variables globales (pour définir par exemple des constantes).

Chaque module est nommé. Ce nom définit un *espace de nom*.

En effet, pour éviter des collisions dans le choix des noms utilisés dans un module avec ceux des autres modules, on utilise un accès préfixé par le nom du module :

```
1   nom_module.element_defini_dans_le_module
```

Il existe de nombreux modules pour Python capable de lui donner des possibilités très étendues.

Accès à un module dans un autre

Il se fait grâce à la commande `import`.

```
1   import os # pour accéder aux appels systèmes
2   import sys # pour la gestion du processus
3   import socket # pour la programmation socket


5   os.exit() # terminaison du processus
6   socket.SOCK_STREAM # une constante pour la programmation réseaux
```

13 Les sorties écran

La fonction `print` permet d'afficher de manière *générique* tout élément, que ce soit un objet, une chaîne de caractères, une valeur numérique *etc*.

Par défaut, elle ajoute un retour à la ligne après.

Le passage d'une liste permet de coller les affichages sur la même ligne.

```
1  a = 'bonjour'
2  c = 12
3  d = open('fichier.txt')
4  print a
5  print c,d, 'La valeur est %d' % c
```

On obtient :

```
bonjour
12 <open file 'fichier.txt', mode 'r' at 0x63c20> La valeur est 12
```

`Print` affiche le contenu « affichable » de l'objet.

Il est également possible d'utiliser `stdout` ou `stderr` :

```
1  import sys
2  sys.stdout.write('Hello\n')
```

La fonction `input()` permet de saisir des valeurs au clavier.

Cette fonction retourne les données saisies comme si elles avaient été **entrées dans le source Python**.

En fait, `input` permet d'utiliser l'interprète ou parser Python dans un programme (combiné à la fonction `eval`, elle permet de rentrer et d'évaluer du code dans un programme qui s'exécute !).

```
1 a = input() # ici on entre [10,'bonjour',30]
2 print a
```

On obtient :

```
[10, 'bonjour', 30]
```

Une **exception** *est levée si les données entrées ne sont pas correctes en Python.*

Ce que nous utiliserons

Pour saisir des données en tant que chaîne de caractères uniquement, il faut utiliser la fonction `raw_input()` qui retourne un chaîne de caractères, que l'on convertira au besoin.

```
1 saisie = raw_input("Entrer ce que vous voulez") # retourne toujours une chaîne
```

Python v3

La fonction `input()` effectue le travail de la fonction `raw_input()` qui disparait.

15 Les conversions

Il existe un certain nombre de fonctions permettant de convertir les données d'un type à l'autre.
La fonction `type()` permet de récupérer le type de la donnée sous forme d'une chaîne.

`fonction`	description	`exemple`
`ord`	retourne la valeur ASCII d'un caractère	`ord('A')`
`chr`	retourne le caractère à partir de sa valeur ASCII	`char(65)`
`str`	convertit en chaîne	`str(10), str([10,20])`
`int`	interprète la chaîne en entier	`int('45')`
`long`	interprète la chaîne en entier long	`long('56857657695476')`
`float`	interprète la chaîne en flottant	`float('23.56')`

Conversion binaire

Il est par exemple possible de convertir un nombre exprimé en format binaire dans une chaîne de caractères :

```
1 representation_binaire = bin(204) # à partir de Python v2.6, donne '0b11001100'
2 entier = int('11001100',2)   # on donne la base, ici 2, on obtient la valeur 204
3 entier = int('0b11001100',2) # donne le même résultat
```

INTERDIT : Opérateur d'affectation

L'opérateur d'affectation n'a pas de valeur de retour.

Il est interdit de faire :

```
1  if (a = ma_fonction()):
2     # opérations
```

INTERDIT : Opérateur d'incrémentation

L'opérateur ++ n'existe pas, mais il est possible de faire :

```
1  a += 1
```

Pour convertir un caractère en sa représentation binaire sur 8 bits

L'instruction `bin()` retourne une chaîne **sans les bits de gauche** égaux à zéro.

Exemple : `bin(5)` $\Longrightarrow$ `'0b101'`

```
1  representation_binaire = bin(ord(caractere))[2:] # en supprimant le '0b' du début
```

La séquence binaire retournée commence au premier bit à 1 en partant de la gauche.

Pour obtenir une représentation binaire sur 8bits, il faut la préfixer avec des `'0'` :

```
1  rep_binaire = '0'*(8-len(representation_binaire))+representation_binaire
```

Python utilise le mécanisme des exceptions : lorsqu'une opération ne se déroule pas correctement, une **exception est levée** ce qui interrompt le contexte d'exécution, pour revenir à un environnement d'exécution supérieur, jusqu'à celui gérant cette exception.

Par défaut, l'environnement supérieur est le shell de commande depuis lequel l'interprète Python a été lancé, et le comportement de gestion par défaut est d'afficher l'exception :

```
Traceback (most recent call last):
  File "test.py", line 1, in ?
    3/0
ZeroDivisionError: integer division or modulo by zero
```

Pour gérer l'exception, *et éviter la fin du programme*, il faut utiliser la structure `try` et `except` :

```
1  try:
2      #travail susceptible d'échouer
3  except:
4      #travail à faire en cas d'échec
```

Il est possible de générer des exceptions à l'aide de la commande `raise`.

Ces exceptions correspondent à des classes objet héritant de la classe racine `Exception`.

```
1 raise NameError('Oups une erreur !') #NameError indique que le nom n'existe pas
```

Il existe de nombreux types d'exception (différentes classes héritant de `Exception`).

Elles peuvent également transmettre des paramètres.

```
1 nombre = raw_input( "Entrer valeur: " )
2 try:
3       nombre = float( nombre )
4       resultat = 20.0 / nombre
5 except ValueError:
6       print "Vous devez entrer un nombre"
7 except ZeroDivisionError:
8       print "Essai de division par zéro"
9 print "%.3f / %.3f = %.3f" % ( 20.0, nombre, resultat )
```

Question : est-ce que toutes les exceptions sont gérées dans cette exemple ?

La définition de ses propres exceptions est en dehors du domaine d'application de ce cours.

Ouverture, création et ajout

La fonction "open" renvoie un objet de type `file` et sert à ouvrir les fichiers en :

"r"	lecture	
"w"	écriture	*le fichier est créé s'il n'existe pas, sinon il est écrasé*
"a"	ajout	*le fichier est créé s'il n'existe pas, sinon il est ouvert et l'écriture se fait à la fin*

Pour vérifier que l'ouverture du fichier se fait correctement, il faut **traiter une exception** de type `Exception` (elle peut fournir une description de l'erreur).

```
1 try:
2         fichier = open("lecture_fichier.pyt","r")
3 except Exception, message:
4         print message
```

Ce qui peut produire :

```
[Errno 2] No such file or directory: 'lecture_fichier.pyt'
```

Pour simplifier, on utilisera le type de la classe racine `Exception`, car on attend ici qu'une seule erreur.

Dans le cas où l'on veut gérer plusieurs exceptions de types différents, il faut indiquer leur type respectif.

Lecture d'un fichier

L'objet de type `file` peut être utilisé de différentes manières pour effectuer la lecture d'un fichier. C'est un objet qui peut se comporter comme une séquence, ce qui permet d'utiliser le `for` :

```
1 for une_ligne in fichier:
2     print une_ligne
```

Mais également comme un « itérateur » (qui lève une exception à la fin) :

```
1 while 1:
2     try:
3         une_ligne = fichier.next() #renvoie l'exception StopIteration en cas d'échec
4         print une_ligne
5     except: break
```

Ou bien simplement à travers la méthode `readline()` :

```
1 while 1:
2     une_ligne = fichier.readline() #renvoie une ligne avec le \n à la fin
3     if not une_ligne: break
4     print une_ligne
```

Lecture caractère par caractère

Sans argument, la méthode `readline` renvoie la prochaine ligne du fichier.

Avec l'argument `n`, cette méthode renvoie `n` caractères au plus (jusqu'à la fin de la ligne). *Pour lire exactement `n` caractères, il faut utiliser la méthode `read`.*

Avec un argument de 1, on peut lire un fichier caractère par caractère.

À la fin du fichier, elle renvoie une chaîne vide (pas d'exception).

```
1  while 1:
2     caractere = fichier.readline(1)
3     if not caractere : break
4  fichier.close() # ne pas oublier de fermer le fichier
```

Autres méthodes

`write(c)`	écrit la chaîne c dans le fichier
`fileno()`	retourne le descripteur de fichier numérique
`readlines()`	lit et renvoie toutes les lignes du fichier
`tell()`	renvoie la position courante, en octets depuis le début du fichier
`seek(déc,réf)`	positionne la position courante en décalage par rapport à la référence indiquée par 0 : début, 1 : relative, 2 : fin du fichier

18.3 Manipulation des données structurées

Les données structurées correspondent à une séquence d'octets. Cette séquence est composée de groupes d'octets correspondant chacun à un type. En C ou C++ :

```
1   struct exemple {
2      int un_entier;          # 4 octets
3      float un_flottant[2];   # 2*4 = 8 octets
4      char une_chaine[5];   # 5 octets
5      }                 # la structure complète fait 17 octets
```

En Python, les structures de cette forme **n'existent pas** et une chaîne de caractère sert à manipuler cette séquence d'octets.

Le module spécialisé `struct` permet de décomposer ou composer cette séquence d'octets suivant les types contenus.

Pour décrire la structure, on utilise une *chaîne de format* où des caractères spéciaux expriment les différents types.

Le module fournit 3 fonctions :

`calcsize(fmt)`	retourne la taille de la séquence complète
`pack(fmt,...)`	construit la séquence à partir de la chaîne de format et d'une liste de valeurs
`unpack(fmt,c)`	retourne une liste de valeurs en décomposant suivant la chaîne de format

La chaîne de format

Elle est composée d'une suite de caractères spéciaux :

	type C	Python		type C	Python
`x`	pad byte	pas de valeur			
`c`	char	chaîne de 1 car.	`s`	string	char[]
`b`	signed char	integer	`B`	unsigned char	integer
`h`	short	integer	`H`	unsigned short	integer
`i`	int	integer	`I`	unsigned int	long
`l`	long	integer	`L`	unsigned long	long
`f`	float	float	`d`	double	float

Un `!` devant le caractère permet l'interprétation dans le sens réseau (Big-Endian).

Pour répéter un caractère il suffit de le faire précéder du nombre d'occurrences (obligatoire pour le `s` où il indique le nombre de caractères de la chaîne).

Sur l'exemple précédent, la chaîne de format est : `iffccccc` ou `i2f5c'`.

```
1  import struct
2  sequence = struct.pack('iffccccc', 10, 45.67, 98.3, 'a', 'b', 'c', 'd', 'e')
3  liste_donnees = struct.unpack('i2f5c', sequence)
```

Une ER permet de faire de l'appariement de motif, *pattern matching* : il est possible de savoir si un motif est **présent** dans une chaîne, mais également **comment** il est présent dans la chaine (en mémorisant la séquence correspondante).

Une expression régulière est exprimée par une suite de *meta-caractères*, exprimant :

- une *position* pour le motif
  - `^` : début de chaîne
  - `$` : fin de chaîne
- un caractère
  - `.` : n'importe quel caractère
  - `[ ]` : un caractère parmi une liste
  - `[^ ]` : tous les caractères sauf...
  - `[a-zA-Z]` : toutes les lettres
- une alternative
  - `|` : *ceci* ou *cela*
- des quantificateurs, qui permettent de répéter le caractère qui les précédent :
  - `*` : zéro, une ou plusieurs fois
  - `+` : **une** ou plusieurs fois
  - `?` : zéro ou une fois
  - `{ n }` : *n* fois
  - `{ n, m }` : entre *n* et *m* fois
- des familles de caractères :
  - `\d` : un chiffre
  - `\D` : tout sauf un chiffre
  - `\s` : un espace
  - `\w` : un caractère alphanumérique
  - `\n` newline
  - `\r` retour-chariot

Le module `re` permet la gestion des expressions régulières :

- l'expression, avec des caractères spéciaux (comme `\d` pour *digit*) ;
- la compilation, pour rendre rapide le traitement ;
- la recherche dans une chaîne de caractères ;
- la récupération des séquences de caractères correspondantes dans la chaîne.

```
1  import re
2  une_chaine = '12 est un nombre'
3  re_nombre = re.compile(r"(\d+)") # on exprime, on compile l'expression régulière
4  resultat = re_nombre.search(une_chaine) #renvoie l'objet None en cas d'échec
5  if resultat :
6     print 'trouvé !'
7     print resultat
8     print resultat.group(1)
```

```
trouvé !
<_sre.SRE_Match object at 0x63de0>
12
```

L'ajout de parenthèses dans l'ER permet de mémoriser une partie du motif trouvé, accessible comme un groupe indicé de caractères (méthode `group(`*`indice`*`)`).

Différence majuscule/minuscule

Pour ne pas en tenir compte, il faut l'indiquer avec une constante de module.

```
1   re_mon_expression = re.compile(r"Valeur\s*=\s*(\d+)", re.I)
```

Ici, `re.I` est l'abréviation de `re.IGNORECASE`.

Motifs memorisés

Il est possible de récupérer la liste des motifs mémorisés :

```
1   import re
2   chaine = 'Les valeurs sont 10, 56 et enfin 38.'
3   re_mon_expression = re.compile(r"\D*(\d+)\D*(\d+)\D*(\d+)", re.I)
4   resultat = re_mon_expression.search(chaine)
5   liste = resultat.groups()
6   for une_valeur in liste:
7       print une_valeur
```

On obtient :

```
10
56
38
```

Décomposer une ligne

Il est possible "d'éclater" une ligne suivant l'ER représentant les séparateurs :

```
1  import re
2  chaine = 'Arthur:/::Bob:Alice/Oscar'
3  re_separateur = re.compile( r"[:/]+" )
4  liste = re_separateur.split(chaine)
5  print liste
```

```
['Arthur', 'Bob', 'Alice', 'Oscar']
```

Composer une ligne

Il est possible de composer une ligne en « joignant » les éléments d'une séquence à l'aide de la méthode `join` d'une chaîne de caractère :

```
1  sequence = ['Mon', 'chat',"s'appelle",'Neko']
2  print sequence
3  print "_".join(sequence)
```

```
['Mon', 'chat', "s'appelle", 'Neko']
Mon_chat_s'appelle_Neko
```

Une forme simple de *programmation par contrat* est introduite grâce à la fonction `assert` :

```
1  assert un_test, une_chaine_de_description
```

Si la condition n'est pas vérifiée alors le programme lève une exception.

```
1  try:
2          a = 10
3          assert a<5, "mauvaise valeur pour a"
4  except AssertionError,m:
5          print "Exception: ",m
```

```
Exception:  mauvaise valeur pour a
```

Ainsi, il est possible d'intégrer des tests pour être sûr des entrées d'une fonction par exemple et ce, afin de faire des programmables **maintenables** !

Exécuter une commande

Il est possible de lancer une commande shell pour en obtenir le résultat :

```
1  import commands
2  resultat_ls = commands.getoutput('ls *.py') # récupère la liste des fichiers
```

Se connecter à une commande

Il est possible de lancer une commande shell en multitâche et :

— de lui envoyer des lignes en entrée (sur le `stdin` de la commande) ;

— de récupérer des lignes en sortie (depuis le `stdout`).

```
1  import os
2  sortie_commande = os.popen('ma_commande')
3  while 1:
4      line = sortie_commande.readline()
```

équivalent à

```
2  sortie_commande = os.popen('ma_commande','r') # pour lire uniquement
```

Pour envoyer des lignes à la commande :

```
2  entree_commande = os.popen('ma_commande','w') # pour écrire uniquement
```

La définition d'une fonction se fait à l'aide de `def` :

```
1  def ma_fonction():
2      #instructions
```

Les paramètres de la fonction peuvent être nommés et recevoir des valeurs par défaut.

Ils peuvent ainsi être donné dans le *désordre* et/ou pas en totalité (très utile pour les objets d'interface comportant de nombreux paramètres dont seulement certains sont à changer par rapport à leur valeur par défaut).

```
1  def ma_fonction(nombre1 = 10, valeur = 2):
2      return nombre1 / valeur
3  print ma_fonction()
4  print ma_fonction(valeur = 3)
5  print ma_fonction(27.0, 4)
```

```
5.0
3.33333333333
6.75
```

Plusieurs valeurs de retour

```
1  def ma_fonction(x,y):
2      return x*y,x+y
3  # code principal
4  produit, somme = ma_fonction(2,3)
5  liste = ma_fonction(5,6)
6  print produit, somme
7  print liste
```

```
6 5
(30,11)
```

Une *lambda expression* ou un objet fonction

```
1  une_fonction = lambda x, y: x * y
2  print une_fonction(2,5)
```

```
10
```

Informations sur les fichiers

Pour calculer la taille d'un fichier, il est possible de l'ouvrir, de se placer en fin et d'obtenir la position par rapport au début (ce qui indique la taille) :

```
1  mon_fichier = open("chemin_fichier", "r")
2  mon_fichier.fseek(2,0)   #On se place en fin, soit à zéro en partant de la fin
3  taille = mon_fichier.ftell()
4  mon_fichier.fseek(0,0)   # Pour se mettre au début si on veut lire le contenu
```

Pour connaître la nature d'un fichier :

```
1  import os.path
2  if os.path.exists("chemin_fichier") : # etc
3  if os.path.isfile("chemin_fichier") : # etc
4  if os.path.isdir("chemin_fichier") : # etc
5  taille = os.path.getsize("chemin_fichier") # pour obtenir la taille d'un fichier
```

Lorsque l'on écrit un programme Python destiné à être utilisé en tant que « script système », c-à-d. comme une commande, il est important de soigner l'interface avec l'utilisateur, en lui proposant des *choix par défaut* lors de la saisie de paramètres :

```
1   #!/usr/bin/python
2   import sys
3   #configuration
4   nom_defaut = "document.txt"
5   #programme
6   saisie = raw_input("Entrer le nom du fichier [%s]" % nom_defaut)
7   nom_fichier = saisie or nom_defaut
8   try:
9      entree = open(nom_fichier, "r")
10  except Exception, message:
11     print message
12     sys.exit(1)
```

— en ligne 4, on définie une valeur par défaut pour le nom du fichier à ouvrir ;
— en ligne 6, on saisie le nom de fichier avec, entre crochets, le nom par défaut ;
— en ligne 7, si l'utilisateur tape « entrée », `saisie` est vide, c-à-d. *fausse* ;
résultat : l'opérateur « or » affecte la valeur par défaut, qui est *vraie*.

Il est possible d'obtenir une deuxième séquence à partir d'une première en appliquant sur chaque élément de la première une opération.
La deuxième séquence contient le résultat de l'opération pour chacun des éléments de la première séquence.

Il est possible d'obtenir le résultat en une instruction unique pour la construction de cette deuxième séquence.

Exemple : on cherche à obtenir la liste des lettres en majuscule à partir des valeurs ASCII de 65 ('A') à 91 ('B') :

```
1   [chr(x) for x in range(65, 91)]
```

```
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',
 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
```

Type de données simple *vs* **Type de données complexe**

Toutes les données sont représentées dans un ordinateur sous forme de **séquences de bits**, 01001100110010110101110011011100, et c'est le **type** qui permet de les traiter comme un chaîne de caractères, un entier ou un réel.

Ce « type » permet de *qualifier* une donnée en l'associant à un ensemble de valeurs mais également à *exprimer* un ensemble d'opérations possibles sur ces données (les chaînes de caractères et la concaténation, les entier et les opérations arithmétiques).

Certains types de données font partie du langage de programmation, ce sont les « primitives » :

- les types de données **simple** : il correspond à des données exprimées sous la forme la plus simple et qui ne peuvent être décomposées en parties plus petites (les entiers et les réels qui sont exprimés directement en tant que valeur) ;
- les types de données **complexes** : ils sont constitués à partir de composants multiples, eux-mêmes de type simple ou complexe (les chaînes de caractères, les listes, les dictionnaires, les *objets*).

Pour résoudre des problèmes complexes et importants, il est nécessaire de pouvoir **définir ses propres types** en **combinant** les types fournis par le langage : *en adaptant les données au problème, on en simplifie la programmation.*

La notion d'abstraction

Une **abstraction** est un mécanisme qui permet de séparer les propriétés d'un *objet* et de restreindre l'*attention* du programmeur seulement sur celles nécessaires à la résolution du problème auquel il est confronté.

L'utilisateur de cette abstraction n'a pas besoin de comprendre tous les détails afin de l'utiliser, mais seulement les détails qui ont un rapport avec son problème.

- l'**abstraction procédurale** : elle correspond à utiliser une fonction ou une méthode suivant son résultat mais sans savoir comment elle l'obtient.

  La fonction `sqrt(x)`*, calcule la* racine carrée *d'un nombre, mais sait-on comment elle la calcule ? Est-ce qu'il est nécessaire de savoir comment elle est implémentée ?*

  N'est-ce pas suffisant de savoir comment l'utiliser ?
- l'**abstraction de donnée** : elle correspond à la séparation des propriétés d'un type de donnée, ses valeurs et ses opérations, de leur implémentation.

  Vous utilisez les chaînes de caractères dans Python couramment, mais savez vous comment sont-elles implémentées, comment les données sont organisées en interne et comment les opérations sont elles-mêmes implémentées ?

  Ces abstractions peuvent être organisées sous formes de **couches** successives :
  - ◇ la première couche est constituée de l'ordinateur lui-même, dans laquelle il existe peu d'abstraction : manipulation directe de bits, opérations logiques sur ces bits, *etc* ;
  - ◇ une couche supérieure correspond à la mise en œuvre d'opérations arithmétiques sur des entiers.

    Dans la plupart des langages, ces opérations ne fonctionnent que sur des données de taille binaire limitée, définissant les limites de l'intervalle de gestion des valeurs : pour un ordinateur fonctionnant avec des entiers signés sur 32 bits, les valeurs sont comprises entre -2^{31} *et* $2^{31} - 1$.
  - ◇ *Que se passe-t-il si on désire utiliser des entiers plus grands ?* On ajoute une nouvelle couche pour les mettre en œuvre, ce qui est le cas dans Python, où la taille des entiers n'est pas limitée.

Un **type abstrait de données** est un type de données défini par le programmeur qui définit :

— un ensemble de valeurs ;

— un ensemble d'opérations qui peuvent être réalisées sur ces valeurs.

*Cette définition est **indépendante** de son implémentation, ce qui permet de se focaliser sur son utilisation et non sur son implémentation.*

Mise en œuvre des types abstraits

Dans Python, on disposera de deux possibilités :

◇ une **version simplifiée**, basée sur une version dégradée du « modèle objet », introduit dans le transparent suivant. Elle ne permettra pas :

  ◇ de limiter les opérations réalisables sur le type abstrait (le programmeur peut faire ce qu'il veut sans garde-fou : une opération possible sur une liste en général n'a peut-être pas de sens sur une liste incluse dans le type abstrait que l'on a définit) ;

  ◇ d'organiser les abstractions en couches successives (on reste sur les types offerts par le langage Python) ;

  ◇ de s'interfacer avec des mécanismes présents dans Python visant à simplifier la mise en œuvre d'algorithmes de résolution de problèmes complexes (on verra dans les transparents portant sur la version *améliorée* ce qui manque).

Cette version simplifiée utilise une sorte de « fourre-tout » dans lequel on associera des données de types simples et complexes disponibles dans Python.

◇ une **version améliorée** où l'on pourra avoir un contrôle et une abstraction plus forte, tirant parti d'une **version réduite** du « modèle objet. »

La version améliorée peut-être proposée une fois la version simplifiée maîtrisée.

La notion d'abstraction : le « modèle objet »

Quelques remarques préliminaires

Python est, dès sa fondation, un langage « orienté objet ».

Il est difficile *d'échapper* à toute manipulation d'objet lorsque l'on programme Python :

- L'utilisation des chaînes de caractères en Python oblige l'utilisation de la *notation objet* : `"minuscule".upper()` qui renvoie `MINUSCULE`, correspond à l'utilisation de la chaîne en tant qu'*objet*.
- La gestion des listes, des dictionnaires, des fichiers, *etc.* se fait sous forme « *objet* ».

Le **but** :

- n'est pas d'introduire « l'approche objet » qui est un **modèle de programmation avancé**.
- est de fournir les **concepts de bases** et **les règles d'implémentation** suffisants :
  - ◇ pour exploiter au mieux le langage Python ;
  - ◇ pour bénéficier des avantages de cet approche objet dans la mise en œuvre des **types abstraits**.

Quelques concepts et notations

- Un « objet » est une entité logicielle qui stocke des données.
- Une « classe » est un modèle qui décrit les données stockées dans l'objet et qui définit les opérations qui peuvent être réalisées sur l'objet.
- Les objets sont créés ou « instanciés » à partir de classes, et chaque objet est une « instance » de la classe depuis laquelle il a été créé.
- Une « méthode » :
  - ◇ est une opération que l'on peut réaliser sur un objet (l'ensemble des méthodes est appelé « interface ») ;
  - ◇ est définie dans la classe et s'applique uniquement sur une instance ;
  - ◇ est similaire à une fonction qui reçoit **toujours en paramètre** l'instance sur laquelle elle s'applique sous l'identifiant « `self` » ;
  - ◇ utilise, lors de son appel, la notation `mon_objet.ma_methode(arguments)`.
    L'argument `self` *est ajouté automatiquement par Python et ne doit pas être fourni par le programmeur.*

La définition de classe

- ▷ Elle utilise le mot-clé « `class` » ;
- ▷ Elle contient la définition des méthodes de l'interface :
  - ◇ ces méthodes sont définies sous formes de fonction dans le corps de la classe (*attention à l'indentation correcte du source*) ;
- ▷ Elle implémente un *constructeur* chargé d'initialiser l'instance créée par l'appel de la classe :
  - ◇ elle définit les variables de l'instance, appelées « attributs », avec des valeurs éventuellement fournies par le programmeur.

Il n'y a pas de « destructeur », Python prenant soin de détruire les instances qui ne sont plus utilisées dans le programme.

```
1  class Point :
2      def __init__(self, x, y):
3          self.coordX = x
4          self.coordY = y
5      def obtenirX(self):
6          return self.coordX
7      def obtenirY(self):
8          return self.coordY
```

- La méthode constructeur est **toujours** appelée `__init__`.
- Les méthodes reçoivent **toujours** le paramètre `self` qui désigne l'instance sur laquelle on va travailler. *Ce qui est normal, car le code de la méthode est partagée par toutes les instances et doit donc se différencier par l'instance sur laquelle elle doit travailler.*
- on doit définir des méthodes pour récupérer/modifier les données, les « attributs », de l'objet.

```
1  pointA = Point(5,7)
2  pointB = Point(0, 0)
3  print "A(",pointA.obtenirX,";",
4          pointA.obtenirY,")"
```

On crée, ou « instancie », deux variables `pointA` et `pointB` en utilisant le nom de classe comme une fonction. Cela correspond à :

1. réserver de l'espace mémoire pour l'objet ;
2. appeler la fonction `__init__` sur cet objet.
3. on transmet éventuellement des arguments mais `self` est sous-entendu.

La définition de classe et l'organisation du code source

Dans Python, il est *conseillé* de mettre le code se rapportant à une classe dans un fichier indépendant.

Cela permet, entre autre, de masquer l'implémentation puisqu'elle n'est pas lisible directement dans le programme l'utilisant.

On utilisera un fichier « point.py » contenant la définition de la classe vu précédemment :

```
1  class Point :
2     def __init__(self, x, y):
3        self.coordX = x
4        self.coordY = y
5     def obtenirX(self):
6        return self.coordX
7     def obtenirY(self):
8        return self.coordY
9     ...
```

Puis dans le programme où l'on veut utiliser la classe :

```
1  from point import Point
2  ...
3  pointA = Point(5,7)
4  pointB = Point(0, 0)
5  print "A(",pointA.obtenirX,";",pointA.obtenirY,")"
```

La ligne 1 doit être mise avant toute utilisation de la classe Point (par exemple en début de programme).

24.2 Les types abstraits de données, version simplifiée

Dans Python, il est possible d'obtenir une type abstrait de données similaires à celles fournies par les langages Pascal ou C à l'aide de la définition suivante :

```
1  class ma_structure :
2      pass
```

En ligne 2, on « évite » de définir la classe ; elle devient, donc, vide.

On l'utilise en lui ajoutant des « champs » de la manière suivante :

```
1  ma_variable = ma_structure()
2  ma_variable.x = 15.2
3  ma_variable.y = 23.6
4  ma_variable.nom = "A"
5  print "Le point ", ma_variable.nom, " a pour coordonnees (", ma_variable.x, ";", ma_variable.y, ")"
```

En ligne 1, on fait appel à la classe vide définie précédemment pour obtenir un nouvel élément.

Avantages

Simplicité de mise en œuvre, *on évite l'approche « Objet » :*

- ◇ *on définit une classe vide.*
- ◇ *on appelle le « constructeur » de la classe (*notion abordée dans le transparent suivant*) ;*
- ◇ *on « peuple » la structure, en lui ajoutant des champs par affectation (ligne 2, 3 & 4).*

Inconvénients

Si une erreur est faite lors de l'écriture de l'accès à un champ, elle n'est pas détectable :

```
1  ma_variable.valeurdex = 5
2  ...
3  ma_variable.valeurdeX = 7
4  print ma_variable.valeurdex
```

Donne le résultat :

```
5
```

Et non la valeur 7 attendue.

On utilise *un peu* « l'approche objet », mais **sans en tirer parti.**

Retour sur la définition de type abstrait

Un **type abstrait de données** est un type de données défini par le programmeur qui définit :
- un ensemble de valeurs ;
- un ensemble d'opérations qui peuvent être réalisées sur ces données.

Apport du modèle objet

La séparation entre définition et implémentation est garantie par l'obligation d'utiliser un ensemble fini d'opérations, appelé **interface**, pour manipuler ces données. *On parle de « masquage » des détails de l'implémentation.*
Ces types abstraits de données peuvent être vus comme des « boîtes noires » (comme en électronique).
Le programme utilisateur interagit avec une *instance* d'un type abstrait en appelant une des différentes opérations définies dans son *interface*.

L'ensemble de ces opérations peuvent être regroupés en quatre catégories :
- les **constructeurs** : crée et initialise une nouvelle instance d'un ADT ;
- les **accesseurs** : retourne les valeurs contenues dans une instance sans les modifier ;
- les **modificateurs** : modifie les valeurs contenues dans une instance ;
- les **itérateurs** : traite les composants de manière individuelle et séquentielle.

Exemple : le type abstrait des chaînes de caractères utilise l'interface suivante :

Programme Utilisateur	Interface chaîne caractères	Implémentation cachée

`chaine = "texte".upper()`	`str()`
	`upper()`
	`lower()`
	`...`
		C'est l'intérieur de la boîte noire...

- ***On peut essayer de résoudre le problème à la main au lieu de rester bloquer sur l'implémentation.***
  Par exemple, imaginons que dans notre problème il soit nécessaire d'extraire les valeurs à traiter à partir d'un fichier sur disque et qu'il faille les stocker en mémoire dans le programme.
  Si l'on se focalise sur les détails d'implémentation, on va perdre beaucoup de temps sur la structure de stockage, comment l'utiliser et quel est le choix le plus efficace.

- ***On peut diminuer le nombre d'erreurs qui peuvent survenir du fait d'une mauvaise utilisation des structures de données en empêchant un accès direct à celles-ci.***
  On limite le nombre de « point d'accès » à la structure de donnée aux seules opérations de l'interface, et il est ainsi possible de découvrir les mauvais usages.

- ***On peut modifier l'implémentation du type abstrait sans avoir à modifier le programme qui l'utilise.***
  Il arrive que l'implémentation initiale du type abstrait ne soit pas la plus efficace, ou alors on a besoin que les données soient organisées d'une manière différente. En exigeant l'accès au type abstrait uniquement au travers de l'interface, il est possible de remplacer la « boîte noire » par une autre.

- ***Il est plus facile de décomposer de large programme en plusieurs morceaux plus petits sur lesquels peuvent travailler des personnes différentes.***
  À l'aide des types abstraits, il est possible de répartir le travail entre différentes personnes, une fois qu'elles se sont mises, ensembles, d'accord sur les différentes interfaces. Le travail de répartition, de coordination et de contrôle reste néanmoins difficile à réaliser.

Un type abstrait

Il est défini :
- en spécifiant le domaine des valeurs de chaque élément qui le compose ;
- l'ensemble des opérations qui peuvent réalisées sur ces domaines de valeurs.

La définition doit être **claire** et **exhaustive** : seules les opérations définies seront réalisables sur une *instance* du type abstrait.

Exemple de problème : calendrier et nombre de jours écoulés entre deux dates

- ◇ Le calendrier tel que l'on l'utilise actuellement, est le calendrier **grégorien** introduit le vendredi 15 octobre 1582.
- ◇ Il a été étendu aux dates antérieures à son introduction sous sa forme *proleptique*.
- ◇ Il a introduit la notion d'année bissextile afin d'ajouter un jour intercalaire, le 29 février, afin d'éviter de trop grandes dérives par rapport à la durée réelle d'une année définie par une révolution complète de la Terre autour du Soleil (on obtient une erreur d'un jour sur 3000 ans contre un jour tous les 129 ans dans le précédent calendrier, dit julien).

Dans notre problème, nous considérerons la date de référence du 1er janvier de l'an 1.

Pour gérer la notion de date, nous aurons besoin du type abstrait permettant les opérations suivantes :
- `Date(jour, mois, annee)` : crée une nouvelle *instance* de date.
  On souhaitera pouvoir exprimer le mois sous forme textuelle ou numérique.
- `jour()`, `mois()`, `annee()` : renvoi le jour, mois et année contenu dans la date ;
- `estUneDateValide()` : permet de vérifier que la date est valide (jour $\in [1, 31]$, mois $\in [1, 12]$ et annee> 0) ;
- `estBissextile(annee)` : renvoi un booléen indiquant si l'année donnée en paramètre est bissextile ;
- `nbJours()` : renvoi le nombre de jours écoulés depuis la date du 1/1/1.

Le but sera de pouvoir calculer des différences en nombre de jours entre deux dates.

Exemple de la date : la notion d'année bissextile

Exemple : Calcul du nombre de jours écoulés depuis le 1/1/1 et l'année précédente

D'après Wikipédia

Depuis l'instauration du calendrier grégorien, sont bissextiles les années :
- soit divisibles par 4 mais non divisibles par 100
- soit divisibles par 400.

Déterminer qu'une année est bissextile

```
1 def estBissextile(annee):
2     if (((annee % 4) == 0) and ((annee % 100) != 0)):
3         return True
4     if ((annee % 400) == 0):
5         return True
6     return False
```

On utilisera la possibilité de rupture de séquence offert par le retour de la fonction :

- ◇ *si le test de la ligne 2 réussie alors on quitte la fonction en ligne 3 ;*
- ◇ *Si l'on effectue le test de la ligne 4, alors cela veut dire que celui de la ligne 2 a échoué :* ***le « sinon » est sous-entendu.***
- ◇ *enfin, la ligne 6 sous-entend que les tests des lignes 2 et 4 ont échoué.*

Calcul du nombre de jours depuis le 1/1/1 et jusqu'à l'année précédente

On peut utiliser la formule suivante pour connaitre le nombre de jours écoulés depuis la date du 1/1/1 jusqu'à l'année de la date que l'on veut considérer :

```
1 jours = (annee - 1) * 365 + (annee - 1) / 4 + (annee - 1) / 400 - (annee - 1) / 100
```

On tient compte du jour à ajouter pour chaque année bissextile.

Exemple de la date : calcul du nombre de jours écoulés (version 1)

Calcul du nombre de jours écoulés depuis le début de l'année et jusqu'au mois précédent

On aura besoin du nombre de jours de chaque mois dans une année normale et bissextile.

Pour faciliter l'entrée des mois, on ajoutera également la possibilité d'utiliser le nom de chaque mois.

```
1  lesMois = ('janvier','fevrier','mars','avril','mai','juin','juillet','aout','septembre',
2             'octobre','novembre','decembre')
3  lesJours = ('lundi','mardi','mercredi','jeudi','vendredi','samedi','dimanche')
4  nbJoursAnnee = (31,28,31,30,31,30,31,31,30,31,30,31)
5  nbJoursAnneeBissextile = (31,29,31,30,31,30,31,31,30,31,30,31)
```

On utilise des listes pour contenir les différentes valeurs et on évite les lettres accentuées à moins que l'environnement d'exécution et d'édition des fichiers sources soient convenablement configurés.

Obtenir le nombre de jours du début de l'année au mois précédent (version avec « `for` »)

```
1  nbjours = 0
2  if (mois>=2) :
3     if (self.estBissextile(annee)):
4        mois_courant = 1
5        for duree in (nbJoursAnneeBissextile):
6           nbjours = nbjours + duree
7           mois_courant = mois_courant + 1
8           if (mois_courant == mois) :
9              break
10    else:
11       mois_courant = 1
12       for duree in (nbJoursAnnee):
13          nbjours = nbjours + duree
14          mois_courant = mois_courant + 1
15          if (mois_courant == mois):
16             break
```

- ◇ *On dispose de la liste du nombre de jours par mois.*
- ◇ *Python permet un **accès par position** aux éléments d'une liste, en commençant à la position 0.*

— *Ici se pose la question de savoir comment parcourir les éléments de la liste* `nbJoursAnneeBissextile` *:*

- ◇ *on connait le rang du mois courant exprimé comme une valeur entre* $[1, 12]$ *;*
- ◇ *on doit arrêter le parcours au dernier mois écoulé ;*
- ◇ *il faut un compteur numérique ;*
- ◇ *on a le choix entre :*
  - ⋆ *une boucle* `while` *avec une condition d'arrêt ;*
  - ⋆ *une boucle* `for` *avec une condition de rupture ;*

Exemple de la date : calcul du nombre de jours écoulés (version 2)

Obtenir le nombre de jours du début de l'année au mois précédent (version avec « `while` »)

```
1  nbjours = 0
2  if (mois>=2) :
3     if (self.estBissextile(annee)):
4        mois_courant = 1
5        while(mois_courant != mois) :
6           nbjours = nbjours + nbJoursAnneeBissextile[mois_courant - 1]
7           mois_courant = mois_courant + 1
8     else:
9        mois_courant = 1
10       while(mois_courant != mois) :
11          nbjours = nbjours + nbJoursAnnee[mois_courant - 1]
12          mois_courant = mois_courant + 1
```

▷ *dans cette version, on accède aux éléments par position :*
  ◇ *on doit débuter à 0, soit à la valeur « `mois_courant - 1` » ;*
  ◇ *on maintient le décalage avec « `mois_courant - 1` » ;*

▷ *la condition d'arrêt survient lorsque « le `mois_courant` prend la valeur du `mois` considéré. »*

Discussion : Quelle version choisir en boucle « `for` » et « `while` » ?

◇ la boucle `for` :
  ⋆ parcourt **explicitement** les éléments de la liste ;
  ⋆ ne fournit pas de position, il faut donc **maintenir** avec un compteur la position courante ;
  ⋆ utilise une instruction de **rupture**, `break`, pour *casser* la boucle ;
  ⋆ **Avantage** : le compteur permet de démarrer à la valeur 1 souhaitée ;
  ⋆ **Inconvénient** : utilisation d'une instruction de rupture ;

◇ la boucle `while` :
  ⋆ utilise **explicitement** la condition d'arrêt ;
  ⋆ **Avantage** : « plus proche » de l'esprit de programmation des autres langages ;
  ⋆ **Inconvénient** : requiert l'utilisation systématique d'un compteur de position et utilise l'accès par position en commençant à la position zéro.

*L'utilisation de la boucle `for` est plus appropriée si les **conditions d'arrêts sont multiples**.*

Exemple de la date : calcul du nombre de jours écoulés (version 3)

Obtenir le nombre de jours du début de l'année au mois précédent (version hybride)

```
1  nbjours = 0
2  if (mois>=2) :
3     if (self.estBissextile(annee)):
4     for i in (range(0,len(nbJoursAnneeBissextile))):
5           if (i == mois) :
6              break
7           nbjours = nbjours + nbJoursAnneeBissextile[i - 1]
8     else:
9     for i in (range(0,len(nbJoursAnnee))):
10          if (i == mois) :
11             break
12          nbjours = nbjours + nbJoursAnnee[mois_courant - 1]
```

Ici, on « simule » le fonctionnement de la boucle `for` *telle qu'elle est proposée dans les autres langages.*

- ▷ *on parcourt une « liste d'indices » fournie par la fonction* `range` *;*
- ▷ *on se sert de l'indice :*
  - ◇ *pour accéder par position à chaque élément à partir de la position 0 ;*
  - ◇ *pour déclencher une rupture de séquence avec* `break`*.*

Discussion : Que choisir entre parcours des éléments ou d'un ensemble de position à l'aide du `range` ?

- ◇ la boucle `for` sur `nbJoursAnnee` :
  - ⋆ parcourt **explicitement** les éléments de la liste ;
  - ⋆ ne fournit pas de position, il faut donc **maintenir** avec un compteur la position courante ;
  - ⋆ **Avantage** : ▷ le compteur permet de démarrer à la valeur 1 souhaitée ;
    ▷ c'est les éléments de la liste qui nous intéresse que l'on parcourt.
  - ⋆ **Inconvénient** : accès par position aux éléments, au lieu du parcours *générique* de Python.
- ◇ la boucle `for` sur `range(0,len(nbJoursAnnee))` :
  - ⋆ reproduit *artificiellement* le comportement d'autres langages en créant une liste d'indices que l'on parcourt ensuite.
  - ⋆ **Avantage** : « plus proche » de l'esprit de programmation des autres langages ;
  - ⋆ **Inconvénient** : plus éloigné de « l'esprit » de Python.

On privilégiera la version `for` *de parcours direct de la liste* `nbJoursAnnee` *ou bien la version avec le* `while`*.*

La programmation « par contrat » : post et pré-conditions

Un intérêt de l'utilisation de « l'approche objet » réside dans le fait que l'on doit utiliser des méthodes pour :

- **l'accès aux variables** d'une instance en consultation ou bien en modification ;
- **effectuer** des traitements pouvant modifier les valeurs des variables d'une instance.

*En langage « orienté objet », on parle **d'état** pour désigner l'ensemble des valeurs prises par l'ensemble des variables. Et on distingue les méthodes modifiant l'état de celles qui ne le font pas.*

Il est ainsi possible de spécifier des conditions sur chaque méthode :

- une **pré-condition** : qui doit être vraie pour que l'opération puisse se déclencher (conditions sur « l'état » courant de l'objet, condition sur les arguments donnés à l'appel de la méthode).

  Par exemple : est-ce que la valeur passée en argument est $\in [0, 10]$ comme requis par le traitement ?
- une **post-condition** : qui est vraie à l'issu de l'exécution de la méthode.

  Par exemple : si l'état courant de l'objet est correct il restera correct à l'issu de l'exécution de la méthode.

*On parle de programmation « par contrat » : si on **garantie** que les post-conditions sont vraies **si** les pré-conditions sont vraies.*

Si une **pré-condition n'est pas vraie**, alors on **doit générer une erreur** (cela signifie une mauvaise utilisation du programme avec des mauvaises entrées, un problème de programmation entre différentes parties du programme *etc*).

Mise en œuvre en Python

On utilise une « assertion » qui si elle n'est pas vraie lève une exception de type `AssertionError` :

```
1 assert self.estUneDateValide(j, m, a), "Date invalide"
```

Si l'exception se produit et qu'elle n'est pas traitée (mécanisme `try ... except`*) alors le programme d'arrête et affiche le texte indiqué, ici, « Date invalide ».*

Il est important d'en ajouter lors de l'écriture des méthodes d'un type abstrait pour détecter les erreurs de programmation et d'utilisation.

```
1  class Date:
2     lesMois = ('janvier','fevrier','mars','avril','mai','juin','juillet','aout','septembre','octobre','novembre','decembre')
3     lesJours = ('lundi','mardi','mercredi','jeudi','vendredi','samedi','dimanche')
4     nbJoursAnnee = (31,28,31,30,31,30,31,31,30,31,30,31)
5     nbJoursAnneeBissextile = (31,29,31,30,31,30,31,31,30,31,30,31)
6     def __init__( self, j, m, a):      # Cree une nouvelle instance et l'initialise
7        assert self.estUneDateValide(j, m, a), "Date invalide" # On verifie que les parametres sont acceptables
8        if (m in Date.lesMois) :      # On convertit eventuellement le mois de texte en rang
9           self.mois = Date.lesMois.index(m) + 1
10       else :
11          self.mois = m
12       self.jour = j
13       self.annee = a
14       self.nbjours = self.nbJours()      # on calcule le nombre de jours ecoules depuis le 1/1/1
15    def mois(self):      # Acces a la variable d'instance
16       return self.mois
17    def jour(self):
18       return self.jour
19    def annee(self):
20       return self.annee
21    def nombreJours(self):
22       return self.nbjours
23    def estUneDateValide(self, jour, mois, annee):
24       if not ((jour in Date.lesJours) or ((1 <= jour) and (jour <= 31))) :
25          return False
26       if not ((mois in Date.lesMois) or ((1 <= mois) and (mois <= 12))) :
27          return False
28       if not(annee >= 0) :
29          return False
30       return True
```

```
31      def estBissextile( self, annee):
32          if (((annee % 4) == 0) and ((annee % 100) != 0)):
33              return True
34          if ((annee % 400) == 0):
35              return True
36          return False

38      def nbJours(self):
39          tmp1 = (self.annee-1)*365+(self.annee-1)/4+(self.annee-1)/400-(self.annee-1)/100
40          tmp2 = 0
41          if (self.mois>=2) :
42              if (self.estBissextile(self.annee)):
43                  mois_courant = 1
44                  for duree in (Date.nbJoursAnneeBissextile):
45                      tmp2 = tmp2 + duree
46                      mois_courant = mois_courant + 1
47                      if (mois_courant == self.mois) :
48                          break
49              else:
50                  mois_courant = 1
51                  for duree in (Date.nbJoursAnnee):
52                      tmp2 = tmp2 + duree
53                      mois_courant = mois_courant + 1
54                      if (mois_courant == self.mois) :
55                          break
56          return tmp1 + tmp2 + (self.jour)
```

24.8 Exemple de la date : utilisation

Utilisation de la classe Date

```
darkstar:Cours Algorithmique pef$ python
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
Type "help", "copyright", "credits" or "license" for more information.
>> from date import Date
>> c=Date(27,8,2011)
>> u=Date(1,1,1970)
>> c.nbJours() - u.nbJours()
15213
>>
```

Pour aller plus loin dans l'interfaçage avec Python : la « surcharge d'opérateur »

On peut ajouter les méthodes suivantes à la classe Date :

- ◇ `__str__` : permet l'affichage direct à l'aide de `print`
- ◇ `__eq__` : permet d'utiliser l'opérateur d'égalité == sur notre type abstrait.

```
1  def __str__(self):
2      return "%d/%d/%d"%(self.jour,self.mois,self.annee)
3  def __eq__(self, autre_date):
4      return (self.nbjours == autre_date.nombreJours())
```

```
>> from date import Date
>> c = Date(27,8,2011)
>> u = Date(1,1,1970)
>> c == u
False
>> o = Date(27,"aout",2011)
>> print o
27/8/2011
>> c == o
True
>>
```

24.9 Un type abstrait de type container : le « sac »

C'est un container fonctionnant à la manière d'un panier de commission :
- un élément peut être ajouté ou retiré individuellement ;
- une opération permet de savoir si un élément est présent ou non ;
- une opération permet de parcourir la collection de tous les éléments.

On décide de l'interface suivante, en anglais pour utiliser les mécanismes génériques de Python :
- `Bag()` : créé un sac qui est initialement vide ;
- `length()` : retourne le nombre d'éléments présents.
  C'est cette opération qu'utilise Python lorsque l'on appelle la fonction générique « `len()` ».
- `contains(élément)` : retourne un booléen indiquant ou non la présence de l'élément.
  C'est cette opération qu'utilise Python avec l'opérateur « `in` » ;
- `add(élément)` : ajoute l'élément ;
- `remove(élément)` : retire l'élément et retourne l'élément lui-même.
  On « lèvera » une exception dans le cas où l'élément n'est pas présent ;
- `iterator()` : crée et retourne un itérateur qui peut être utilisé pour passer en revue chaque élément de la collection dans une boucle « `for` ».

Attention

Pour pouvoir « lier » le type abstrait aux opérations génériques de Python comme `len()`, `in` ou un comportement avec les itérateurs, on utilisera des noms d'opérations prédéfinis : `length()`, `contains()`, *etc.* ce qui se traduit par le choix naturel de l'anglais pour définir l'interface.

Les types abstraits contenant plusieurs éléments

Il existe différents termes qui sont utilisés de manière générale en Informatique et de manière particulière dans tel ou tel langage :

- une **collection** : un groupe de valeurs qui *n'impliquent pas d'organisation*, *ni de relation* entre les éléments.
  Parfois une collection peut ne contenir que des éléments de même type mais pas nécessairement (*une collection de nombres entiers ou réels*).

- un **container** : une structure de données ou un type abstrait qui stocke et organise une collection.
  Les valeurs considérées de manière individuelle sont appelées « éléments » et un container sans éléments est considéré comme « vide ».
  L'organisation d'un container ou l'arrangement des éléments entre eux peut varier d'un container à l'autre, de la même que la façon de passer d'un élément à l'autre.
  Python met à disposition , les chaînes de caractères, les tuples, les listes, les dictionnaires.

- une **séquence** : un container dans lequel les éléments sont rangés dans un ordre linéaire et où chaque élément est accessible par une position.
  Python fournit les séquences non modifiables, les chaînes de caractères et les tuples ainsi qu'une séquence modifiable : la liste.
  On remarquera qu'en Python, on de dispose pas de « tableau » mais de « liste ».

- une **séquence ordonnée** : une séquence dont les éléments sont ordonnés suivant une relation entre un élément et son successeur.
  Par exemple, une séquence ordonnée d'entier rangée par ordre croissant ou décroissant.

Exemple d'utilisation

```
1 mon_sac = Bag()
2 mon_sac.add(12)
3 mon_sac.add(23)
4 mon_sac.add(42)
5 valeur = int(input_raw("Deviner une valeur contenue dans le sac :"))
6 if valeur in mon_sac:
7    print("Le sac contient la valeur ", valeur)
8 else :
9    print("Le sac ne contient pas la valeur ", valeur)
```

Cet exemple d'utilisation permet de valider/définir l'interface, avant de procéder à l'implémentation du type abstrait.

Sélectionner la structure de données sous-jacente : un ensemble de critères à discuter

- Est-ce que la structure de donnée choisie correspond aux contraintes définies par le domaine de valeurs du type abstrait ? *Autrement dit, la structure de donnée doit être capable de stocker* ***toutes les valeurs possibles*** *que pourra prendre le type abstrait, avec les contrainte appliquées sur chaque élément.*
- Est-ce que la structure de donnée permet le niveau d'accès et de manipulation de données nécessaires au type abstrait ? *Il faut que l'implémentation permette les opérations définies dans l'interface sans avoir à* ***contourner*** *le principe d'abstraction (exemple : un accès par position est nécessaire dans l'interface et il doit être fourni directement par l'implémentation).*
- Est-ce que la structure de donnée permet une implémentation efficace des opérations de l'interface ? *Il faut sélectionner la structure de donnée la plus proche de celle demandée par le type abstrait.*

On voit qu'il est nécessaire de choisir la structure de donnée la plus adaptée à la mise en œuvre d'un type abstrait dans un contexte donné. Il arrive que pour s'adapter à différents contexte, un type abstrait soit implémenté de différentes manières que le programmeur peut ensuite choisir.

Ce choix peut être fait une fois le programme entier terminé, afin d'en optimiser l'efficacité.

Implémentation du sac : choix de la structure de donnée sous-jacente

Structures de données fournies par Python candidates :

- la **liste** : elle peut stocker tout type d'éléments, y compris des duplicata ;
- le **dictionnaire** : il utilise des paires « (clé/valeur) », où chaque clé doit être unique.
  On peut imaginer utiliser la clé pour stocker l'élément et la valeur pour stocker son nombre d'occurrence en cas de duplicata. Ainsi, lors de l'ajout d'un duplicata, on augmente le compteur si l'élément existe déjà. En cas de retrait et lorsque le compteur arrive à zéro, on supprime l'élément.

Choix de la « liste »

La « liste » plutôt que le « dictionnaire » : la possibilité de connaître le nombre d'occurrence d'un élément n'est pas nécessaire.

Une fois la structure de donnée choisie, on va étudier comment implémenter chaque opération de l'interface du sac :

- *une opération est identique à une opération fournie par la liste : un simple appel de fonction suffit ;*
- *une opération n'est pas fournie par la liste : il va falloir donner son implémentation.*

Analyse de l'interface pour son implémentation

- un sac vide : une liste vide ;
- la taille du sac : la taille de la liste ;
- le test d'appartenance d'un élément dans le sac : le même test fourni par la liste ;
- l'ajout d'un élément : il sera fait à la fin de la liste (l'ordre n'a pas d'importance) ;
- le retrait d'un élément : l'opération identique est fournie par la liste ;
- le parcours des éléments : utilisation d'un itérateur.

La liste est une structure de donnée adaptée. On pourra néanmoins, changer l'implémentation au besoin.

```
1  # Implementation du type abstrait Bag a partir d'une liste Python
2  class Bag :
3     # Construit un sac vide
4     def __init__( self ):
5        self._elements = list() # l'utilisation d'un _ en prefixe indique que la variable est privee
6     # Retourne le nombre d'elements
7     def __len__( self ):
8        return len( self._elements )
9     # Determine si un element est present
10    def __contains__( self, element ):
11       return element in self._elements
12    # Ajoute un element
13    def add( self, element ):
14       self._elements.append( element )
15    # Supprime et retourne un element du sac
16    def remove( self, element ):
17       assert element in self._elements, "L'element doit etre dans le sac."
18       position = self._elements.index( element )
19       return self._elements.pop( position )
20    # Retourne un iterateur pour parcourir le contenu
21    def __iter__( self ):
22       ...
```

L'utilisation du caractère « _ » en préfixe d'une variable d'instance (ou d'un attribut) ou d'une fonction indique que cette variable ou cette fonction ne devrait pas être utilisée en dehors de la classe.

Ce n'est qu'une convention d'écriture dans Python, mais elle permet de distinguer les méthodes d'interface à utiliser depuis l'extérieur de la classe et celles que l'on ne devrait pas utiliser depuis l'extérieur (par exemple depuis le programme principal).

Un itérateur

- permet le parcours du contenu d'un container ;
- est disponible pour les containers Python : chaîne de caractère, tuples, listes et dictionnaires ;
- est utilisé dans la structure de boucle « `for` » ;
- permet la recherche d'un élément particulier, l'impression de tous les éléments, *etc.*

*Il n'est pas obligatoire de fournir un **itérateur** pour un type abstrait de type container, mais en cas de besoin de pouvoir parcourir les éléments du container, c'est le **moyen adapté** pour ne pas avoir à contourner le principe d'abstraction.*

Un itérateur : implémentation

- ◇ C'est un **objet**, c-à-d une structure de donnée et un ensemble d'opération définie dans une **interface**.
- ◇ C'est un objet **indépendant** du container qu'il parcours.
- ◇ Cette **interface** permet à la structure « `for` » de pouvoir l'employer automatiquement.
- ◇ Cette **interface** est normalisée.
- ◇ « Créer son itérateur » revient à « appliquer » cette interface au type abstrait que l'on défini.

*Il est nécessaire **d'ajouter** un nouvel élément à notre type abstrait, un **objet** qui est une sorte de type abstrait plus proche des contraintes du langage que de celles du programmeur.*

Un itérateur : son interface

`__init__()` : sert à l'initialisation de l'itérateur, c-à-d avant le parcours des éléments ;
`__iter__()` : retourne l'itérateur lui-même ;
`next()` : retourne l'élément suivant, ou en fin de collection, lève l'exception `StopIteration`.

Il est **nécessaire** d'ajouter l'opération `__iter__()` au type abstrait, afin qu'il puisse retourner un itérateur.

Il **faut** établir un lien entre l'itérateur et le container qu'il parcourt.

On pourra alors écrire : « `for element in mon_sac : ...` »

L'implémentation

```
1  class _BagIterator :
2     def __init__( self, le_sac ):
3        self._les_elements = le_sac
4        self._element_courant = 0
5     def __iter__( self ):
6        return self
7     def next( self ):
8        if self._element_courant < len( self._les_elements ) :
9           element = self._les_elements[ self._element_courant
   ]
10          self._element_courant += 1
11          return element
12       else :
13          raise StopIteration
```

- ◇ l'opération `__init__` récupère le type abstrait pour établir le **lien** avec l'itérateur ;
- ◇ l'opération `__iter__` retourne l'itérateur lui-même *c'est toujours le cas* ;
- ◇ l'opération `next` réalise le travail de parcours à l'aide d'un compteur de position pour désigner chaque élément et le retourner. *Si la parcours est terminé, on lève l'exception* `StopIteration`.

On remarque que :

— *la seule variable associée à l'itérateur est celle du* ***compteur de position*** *;*

— *l'itérateur dispose d'un* ***accès direct*** *au contenu du type abstrait (il ne passe pas par l'interface de celui-ci pour accèder à ses éléments).*

— *l'utilisation d'un compteur est* ***adaptée au type abstrait*** *tel qu'il a été implémenté : une liste permet l'accès par position.*

Pour l'intégration dans le type abstrait, on rajoute l'opération suivante :

```
1     def __iter__( self ) :
2        return _BagIterator( self._elements )
```

L'opération crée un itérateur et lui fournie la liste contenant les éléments.

Utilisation d'un itérateur

```
1  for un_element in mon_sac :
2      print( element )
```

On utilise directement le type abstrait dans la structure « `for` » :

1. Python exécute automatiquement l'appel à `__iter__` sur le type abstrait « `mon_sac` » et récupère un objet `_BagIterator` ;
2. la boucle `for` appelle automatiquement l'opération `__next__` pour accéder à chaque élément (l'état de l'itérateur varie avec les valeurs prises successivement par sa variable `_element_courant` ;
3. la boucle `for` s'arrête lorsque l'exception `StopIteration` est levée.

On peut détailler le fonctionnement, en le reproduisant :

```
1  # Creation d'un objet BagIterator pour mon_sac
2  iterateur = mon_sac.__iter__()
3  # Repeter la boucle while jusqu'à l'appel à break
4  while True :
5      try:
6          # Obtenir l'element suivant.S'il n'y en a plus
7          # l'exception StopIteration survient
8          element = iterateur.next()
9          # Realiser le corps de la boucle
10         print( element )
11         # Traiter l'exception et casser la boucle
12     except StopIteration:
13         break
```

On pourrait se passer d'itérateur : utilisation d'une opération nouvelle pour parcourir les éléments, ***mais*** *:*

- *il faudrait l'adapter à différents traitements : une pour afficher chaque élément, une autre pour les modifier* etc.
- *il faudrait documenter ces différentes opérations ;*
- *on perdrait en* ***généricité*** *: on apprend une fois à faire un parcours et on généralise sur les différents containers qu'ils soient fournis par le langage ou un type abstrait ajouté.*

L'utilisation d'un itérateur combinée à la boucle `for` *permet de* ***simplifier considérablement*** *l'écriture d'un parcours des éléments contenu dans notre type abstrait.*

Combiner le « Bag » et la « Date »

Empiler les abstractions : une collection de date ?

```
1  from bag import Bag
2  from date import Date

4  c=Date(27,8,2011)
5  u=Date(1,1,1970)
6  o=Date(27,"aout",2011)
7  b=Bag()
8  b.add(c)
9  b.add(u)
10 b.add(o)
11 print "c in b ?", c in b
12 for d in b:
13     print d
```

Ce qui donne à l'exécution :

```
darkstar:Cours Algorithmique pef$ python demo.py
c in b ? True
27/8/2011
1/1/1970
27/8/2011
```

La surcharge d'opérateur

Il est possible d'utiliser les opérateurs de Python pour les définir sur nos propres types abstraits :

Opérateur	Méthode
`str( objet )`	`__str__( self )`
`len( objet )`	`__len__( self )`
`item in objet`	`__contains__( self, item )`
`y = objet[index]`	`__setitem__( self, index, valeur )`
`objet[index] = valeur`	`__getitem__( self, index )`
`objet == autre_objet`	`__eq__( self, autre_objet )`
`objet < autre_objet`	`__lt__( self, autre_objet )`
`objet <= autre_objet`	`__le__( self, autre_objet )`
`objet != autre_objet`	`__ne__( self, autre_objet )`
`objet > autre_objet`	`__gt__( self, autre_objet )`
`objet >= autre_objet`	`__ge__( self, autre_objet )`
`objet + autre_objet`	`__add__( self, autre_objet )`
`objet - autre_objet`	`__sub__( self, autre_objet )`
`objet * autre_objet`	`__mul__( self, autre_objet )`
`objet / autre_objet`	`__truediv__( self, autre_objet )`

Par exemple, sur notre type abstrait « bag » :

On rajoute la méthode :

```
1   def __getitem__(self,index):
2       return self._elements[index]
```

On exécute sur l'exemple précédent :

```
print "Seconde date ",b[1]
Seconde date  1/1/1970
```

En conclusion

L'utilisation d'une « approche objet » légère permet **d'améliorer la gestion des types abstraits en Python**.