IRES Documentation

Version 1.0

Groupe IRES

05 August 2015

Table des matières

CHAPITRE 1

Fonctions - `def`

Les fonctions sont utilisées lorsque des mêmes opérations doivent être réalisées plusieurs fois. Elles aident aussi, pour améliorer la lisibilité du code, à structurer un programme.

1.1 Qu'est-ce qu'une fonction ?

Typiquement, une fonction **retourne** un résultat (grâce à l'instruction `return` de python).

Par exemple la fonction `max(x,y)` retourne le maximum des variables `x` et `y`.

Une fonction peut avoir zéro, un ou plusieurs **paramètres** (aussi appelé **arguments**) ; dans le cas de la fonction `max`, elle *attend* deux paramètres (ici `x` et `y` lui ont été *passés* en paramètres).

Pour utiliser une fonction, il n'est pas nécessaire de connaître exactement son implémentation. Il suffit de savoir quels sont ses arguments (par exemple deux entiers) et quel résultat elle calcule (par exemple, le maximum de ces deux entiers).

En informatique, une fonction peut aussi interagir avec son environnement (modifier le contenu de la mémoire ou afficher un message). On appelle cela un **effet de bord**.

1.2 Et en python ?

Une fonction est définie par le mot clé `def`. On indique ensuite, entre parenthèses, ses arguments, puis on termine par `:`.

Le code de la fonction est ensuite écrit de façon indentée.

L'instruction `return` retourne le résultat calculé (et la fonction s'arête) :

```
def maximum(a,b):
    if a > b:
        return a
    else:
        return b
```

Pour utiliser cette fonction

```
>>> n = maximum(6,9) # ici a=6 et b=9
>>> print(n*10)
90
```

1.3 Fonctions récursives

Une fonction peut être **récursive**, c'est-à-dire s'appeler soi-même !

```
def factorielle(n): # 1*2*3*...*n
    if n == 0:
        return 1
    else:
        return factorielle(n-1) * n
```

Attention, il faut qu'à un moment cette récursion se termine. En fait, c'est comme une récurrence, il faut un cas de base.

CHAPITRE 2

Chaînes de caractères - `str`

Les chaînes de caractères - type `str` pour *strings* - servent à représenter les textes.

— Une chaîne se reconnaît par l'utilisation de **délimiteurs** :
  — guillemets simples : `'permet les guillemets "doubles"'` ou
  — guillemets doubles : `"permet les guillemets 'simples'"`.
— Chaque caractère d'une chaîne est numéroté par un entier (positif ou négatif) appelé **position** ou index :
  — Le premier caractère a la position 0, le second la position 1, etc.
  — Le dernier caractère a la position -1, l'avant dernier -2, etc.
— La **longueur** d'une chaîne est le nombre de caractères qu'elle contient.
— **Caractères «spéciaux»** : Saut de ligne - `\n` ; tabulation (touche Tab) - `\t` ; insérer un backslash `\\` ; ou un délimiteur - `\'` ou `\"`.
— **Insérer un caractère non accessible au clavier** : on insère son «point de code» unicode via `\u<code>` où `<code>` doit être remplacé par la valeur adéquate (voir ce site). Par exemple, `\u265E` devrait donner un cavalier : .
— Obtenir la documentation en ligne : `help(str)` (taper «q» pour finir)

2.1 Opérations de base

— **Créer** une chaîne littéralement - `'...'` ou `"..."` :

```
>>> ch = "bonjour \u262F \n tout l'monde!"
>>> ch # contenu de ch, notez les guillemets.
"bonjour ☯ \n tout l'monde!"
>>> print(ch) # affichage à l'écran de ch1, différence ?
bonjour ☯
 tout l'monde!
>>> autreDelim = 'bonjour \u262F \n tout l\'monde!'
>>> ch == autreDelim
True
```

— Connaître le **nombre de caractères** qu'elle contient - `len(chaine)` :

```
>>> long = len(ch)
>>> long
24
```

— **Savoir si** une sous-chaîne ou un caractère apparaît dans la chaîne - `ss_chaine in chaine` :

```
>>> ch
"bonjour ☯ \n tout l'monde!"
>>> " " in ch # l'espace est un caractère comme un autre ...
True
```

```
>>> "z" in ch
False
>>> "Jour" in ch # attention à la «casse» (majuscule/minuscule)
False
>>> "jour" in ch
True
```

— **Accéder** à un caractère - `chaine[pos]` :

```
>>> ch
"bonjour  \n tout l'monde!"
>>> ch[0]
'b'
>>> # Quelle est la position positive du symbole   ?
>>> ch[-1]
'!'
>>> # Quelle est sa position négative ?
```

— **Extraire** une sous-chaîne - `chaine[pos1:pos2]` ou `chaine[pos1:pos2:pas]` :

```
>>> ch
"bonjour  \n tout l'monde!"
>>> ch[1:4] # Attention [pos1:pos2] = de pos1 inclus jusqu'à pos2 exclus !
'onj'
>>> ch[4:] # pos2 omis = jusqu'à la fin
"our  \n tout l'monde!"
>>> ch[:10] # pos1 omis = depuis le début
'bonjour  '
>>> ch[:10:2] # extraire pos 0, 2, 4, 6, 8 (10 exclus)
'bnor'
```

— **Concaténer** deux chaînes ou plus - `ch1 + ch2 + ...` :

```
>>> "ab" + 'cd'
'abcd'
>>> ch = "Spaghetti"
>>> ch = ch + " carbonara" + '.'
>>> ch
'spaghetti carbonara.'
```

— **Découper** une chaîne relativement à un caractère de séparation - `str.split([sep])` :

```
>>> "un deux    trois\nquatre  ".split() # si sep est omis, le découpage se fait sur les espace
['un', 'deux', 'trois', 'quatre']
>>> ch = 'un,2,,3, quatre'
>>> ch.split(',') # Notez les petites différences dans le cas où sep est précisé
['un', '2', '', '3', ' quatre']
```

2.2 Parcourt d'une chaîne

— **Direct** - `for car in chaine:` :

```
>>> ch = "huit"
>>> for c in ch:
...     print(c)
...
h
u
i
```

```
t
>>> res = ''
>>> for c in ch:
...     print("res='" + res + "' et c='" + c + "' donc res=c+res ???")
...     res = c + res
...
res='' et c='h' donc res=c+res ???
res='h' et c='u' donc res=c+res ???
res='uh' et c='i' donc res=c+res ???
res='iuh' et c='t' donc res=c+res ???
>>> res
'tuih'
```

— **Par énumération** - `for pos, car in enumerate(chaine):` :

```
>>> ch = "du feu"
>>> for p, c in enumerate(ch):
...     print("ch[" + str(p) + "]=" + c)
...
ch[0]=d
ch[1]=u
ch[2]=
ch[3]=f
ch[4]=e
ch[5]=u
```

— **Indirect** : par les positions dans la chaîne - `for pos in range(len(chaine)):` :

```
>>> ch = "du feu"
>>> str(5) # conversion d'un entier en chaîne
'5'
>>> # Note: range(nb) -> 0, 1, 2, 3, ..., nb - 1
>>> # or pos dans chaîne -> 0, 1, 2, ..., len(ch) - 1 !!!
>>> # donc range(len(ch)) -> positions possibles dans chaîne
>>> for i in range(len(ch)):
...     print("ch[" + str(i) + "]=" + ch[i])
...
ch[0]=d
ch[1]=u
ch[2]=
ch[3]=f
ch[4]=e
ch[5]=u
```

— **à l'envers** - `for car in reversed(chaine):` :

```
>>> ch = "bonjour"
>>> for c in reversed(ch):
...     print(c, end="")
...
ruojnob
```

2.3 Autres opérations utiles

— Majuscule/minuscule - `str.upper()`, `.lower()`, `.swapcase()` et `.capitalize()` :

```
>>> "Bonjour".upper()
'BONJOUR'
>>> "PaS PossiBLE".lower()
'pas possible'
>>> 'PaS PossiBLE'.swapcase()
'pAs pOSSIble'
>>> 'auReVoir'.capitalize()
'Aurevoir'
```

— **Formatage** - `str.format()` :

```
>>> # Les «{}» sont remplacés par les valeurs correspondantes
>>> ville = 'Bruxelle'
>>> '{} est la capitale de la {}.'.format(ville, 'Belgique')
'Bruxelle est la capitale de la Belgique.'
>>> piece, pos = "cavalier", (3, 5)
>>> "La position du {a} est ligne {b[0]} colonne {b[1]}.".format(a=piece, b=pos)
'La position du cavalier est ligne 3 colonne 5.'
>>> conv = "En binaire {a}={a:b} et en hexadécimal {a}={a:x}."
>>> conv = conv.format(a=43)
>>> print(conv)
En binaire 43=101011 et en hexadécimal 43=2b.
```

— Chaînes **multilignes** - `'''...'''` ou `"""...."""` :

```
>>> discours = '''Bonjour chers amis,
...
...     Je tenais tout particulièrement à
... vous remercier pour blah blah blah ...
...
... Sincèrement ...'''
>>> discours
'Bonjour chers amis,\n\n    Je tenais tout particulièrement à\nvous remercier pour blah blah bla
>>> print(discours)
Bonjour chers amis,

    Je tenais tout particulièrement à
vous remercier pour blah blah blah ...

Sincèrement ...
```

— **Joindre** les chaînes d'une «séquence» - `str.join(seq)` :

```
>>> '; '.join(['a', 'b'])
'a; b'
>>> l = ["un", "deux", "trois"]
>>> sep = ' puis '
>>> sep.join(l)
'un puis deux puis trois'
```

— **Encoder** pour communiquer - `str.encode()` et `bytes.decode()` :
Python3 représente chaque caractère d'une chaîne par son identifiant unicode. Cela permet, virtuellement, de représenter toutes les langues du monde (ou presque). Pour connaître cet identifiant, utiliser `ord(car)`. Inversement, pour trouver un caractère d'identifiant *id*, utiliser `chr(id)`.
Lorsqu'on veut, par exemple, envoyer un message comme `'bonjour'` sur un réseau, il est en pratique nécessaire d'encoder le message (par défaut en Utf-8) de manière à le représenter (en interne) comme une chaîne d'octets ou *bytes* (regroupement de 8 bits - 0 ou 1). Pour en savoir plus ....

```
>>> ch = 'aïe' # chaîne de caractères
>>> # encodage en un bytes (chaîne d'octets) via Utf-8
>>> chEnc = mess.encode('utf-8')
```

```
>>> type(chEnc) # chaîne d'octet
<class 'bytes'>
>>> chEnc # le préfixe «b» précise qu'il s'agit d'un bytes
b'a\xc3\xafe'
>>> for car in ch: # parcourt de la chaîne de caractères
...     print(ord(car), end=' ') # ord(caractère): identifiant unicode (en décimal)
...
97 239 101
>>> for octet in chEnc: # parcourt de la chaîne d'octets
...     print(octet, end=' ') # chaque octet correspond à un entier de [0,256[
97 195 175 101
>>> # notez que le 'ï' est codé sur 2 octets en Utf-8 !
>>> # pour décoder un bytes c'est à dire retrouver la chaîne de caractères correspondante
>>> message = chEnc.decode('utf-8')
>>> message
'aïe'
```

— Récupérer la liste des lignes - `str.splitlines()` :

```
>>> texte = "un\ndeux\ntrois"
>>> print(texte)
un
deux
trois
>>> lignes = texte.splitlines()
>>> lignes
['un', 'deux', 'trois']
```

CHAPITRE 3

Listes - `list`

Une **liste** - type `list` - est un moyen de regrouper des éléments d'information ou données :

- Les éléments d'une liste sont **ordonnés** ;
- Chaque élément d'une liste est *numéroté* par un entier (positif ou négatif) appelé **position** ou index ;
  - Le premier élément a la position 0, le second la position 1, etc.
  - Le dernier élément a la position -1, l'avant dernier la position -2, etc.
- La **longueur** d'une liste est le nombre d'éléments qu'elle contient ;

3.1 Opérations de base

- **Créer** une liste «littéralement» :

```
>>> l = [5, "liste", -2.3, (1,3)]
>>> l
[5, "liste", -2.3, (1,3)]
```

- Connaître sa «longeur» ou le nombre de ses éléments - `len(list)` :

```
>>> len(l)
4
```

- Savoir si *elt* **appartient ou non** à une liste - `elt in list` :

```
>>> l
[5, "liste", -2.3, (1,3)]
>>> -2.3 in l
True
>>> 1.3 in l
False
```

- **Récupérer** l'élément de position *pos* - `l[pos]` :

```
>>> l[2] # l[0] premier elt, l[1] deuxième, ...
-2.3
>>> l[-1] # l[-1] dernier elt, l[-2] avant dernier, ...
(1, 3)
```

- **Modifier** l'élément de position *pos* - `l[pos] = nouvel_elt` :

```
>>> l
[5, "liste", -2.3, (1,3)]
>>> l[-1] = 0
>>> l
[5, "liste", -2.3, 0]
```

— **Ajouter** *elt* **à la fin** d'une liste - `list.append(elt)` :

```
>>> l.append(["b", "a", "c"])
>>> l
[5, "liste", -2.3, 0, ["b", "a", "c"]]
>>> len(l)
5
```

— **Insérer** *elt* à la postion *pos* - `list.insert(pos, elt)`

```
>>> l
[5, "liste", -2.3, 0, ["b", "a", "c"]]
>>> l.insert(2, "truc")
>>> l
[5, "liste", "truc", -2.3, 0, ["b", "a", "c"]]
```

— **Récupérer et supprimer** le dernier élément (resp. celui de position *pos*) - `list.pop([pos])` :

```
>>> x = l.pop() # si pos est omis la suppression concerne le dernier élément
>>> x
["b", "a", "c"]
>>> l
[5, "liste", "truc", -2.3, 0]
>>> l.pop(2)
"truc"
>>> l
[5, "liste", -2.3, 0]
```

— **Supprimer** l'élément de position *pos* - `del l[pos]` :

```
>>> del l[0]
>>> l
["liste", -2.3, 0]
```

3.2 Parcourt d'une liste

— **Direct** - chaque élément est récupéré successivement - `for elt in list:` :

```
>>> l = [3, "truc", -2.5]
>>> for elt in l:
...     print(elt)
...
3
truc
-2.5
```

— **Par énumération** - `for pos, elt in enumerate(list):` :

```
>>> l = [3, "truc", -2.5]
>>> # Rappels:
>>> #   str(truc): convertit «truc» en chaîne de caractères
>>> #   concaténation: "l[" + "3" + "]=" + "erreur" donne "l[3]=erreur"
>>> for p, v in enumerate(l):
...     print("l[" + str(p) + "]=«" + str(v) + "»")
...
l[0]=«3»
l[1]=«truc»
l[2]=«-2.5»
```

— **Indirect** - en utilisant les positions des éléments dans la liste :

```
>>> # Rappels:
>>> #   range(nb): intervalle d'entiers [0,nb[ (nb exclus),
>>> #   len(l): nombre d'éléments de l,
>>> #     donc range(len(l)) représente toutes les positions possibles
>>> #
>>> for pos in range(len(l)):
...     elt = l[pos]
...     print("Pos. de «", elt,"»:", pos)
...
Pos. de « 3 »: 0
Pos. de « truc »: 1
Pos. de « -2.5 »: 2
```

Note : Même si cette façon de parcourir une liste semble plus compliquée, la connaissance dans la boucle de la position de l'élément peut être déterminante dans certain problème.

— **Inverser le sens de parcourt** `for elt in reversed(list):` :

```
>>> for elt in reversed(l):
...     print(elt)
...
-2.5
truc
3
>>> # range(n1,n2,pas) -> n1, n1+pas, n1+2*pas, ... sans dépasser n2
>>> for i in range(5,0,-1):
...     print(i, end=" ")
...
5 4 3 2 1
```

3.3 Autres opérations utiles

— **Extraire une sous-liste** d'éléments consécutifs - `list[pos1:pos2]` :

```
>>> l = [3, 0, -2, 5]
>>> # l[pos1:pos2] : positions récupérées = entiers de [pos1, pos2[ (pos2 exclus)
>>> l[1:3]
[0, -2]
>>> # l[pos1:] ->  de pos1 (inclus) jusqu'à la fin de la liste
>>> l[1:]
[0, -2, 5]
>>> # l[:pos2] -> du début de la liste jusqu'à pos2 (exclus)
>>> l[:2]
[3, 0]
```

— **Concaténer** deux listes - `list1 + list2` :

```
>>> l1 = [2, 5]
>>> l2 = [3, 0]
>>> l1 + l2
[2, 5, 3, 0]
```

— **Inverser** l'ordre des éléments - `list.reverse()` :

```
>>> l = [2, 5, 3, 0]
>>> l.reverse()
```

```
>>> l
[0, 3, 5, 2]
```

— **Trier** les éléments dans l'ordre croissant - `list.sort()` :

```
>>> l = [3, -1, 5, 0]
>>> l.sort()
>>> l
[-1, 0, 3, 5]
```

— **Compter** le nombre de fois où *elt* apparaît dans la liste - `list.count(elt)` :

```
>>> l = [0, 1, 0, 2, 0]
>>> l.count(0)
>>> 3
```

— Produire une liste qui contient *n* fois le même élément - `list * n` :

```
>>> [0] * 5
[0, 0, 0, 0, 0]
```

— **Convertir** un objet «composite» en liste - `list(obj_composite)` :

```
>>> list("abc")
["a", "b", "c"]
>>> list(range(4))
[0, 1, 2, 3]
```

— Construire une liste en «compréhension» :

```
>>> [x**2 for x in range(9)]
[0,1,4,9,16,25,36,49,64,81]
>>> [(x, y) for x in [-1,1] for y in [-1,1]]
[(-1, -1), (-1, 1), (1, -1), (1, 1)]
>>> [(x, y) for x in [-1,1] for y in [-1,1] if x != y]
[(-1, 1), (1, -1)]
```

CHAPITRE 4

Dictionnaires - `dict`

Un **dictionnaire** - type `dict` - est un moyen de mémoriser des *associations de la forme clé→valeur*.

— Littéralement, un **dictionnaire** est de la forme `{clé1: val1, clé2: val2, ...}`

— Les **clés** sont n'importe quelle valeur «primaire» ou *non modifiable* comme un entier, une chaîne, un tuple...

— Les **valeurs** correspondantes sont de type arbitraire.

— La **longeur** d'un dictionnaire est le nombre de couples clé→valeur qu'il contient

4.1 Opérations de base

— **Créer** un dictionnaire «littéralement» - `{clé1: val1, clé2: val2, ...}` :

```
>>> d = {"café": ":)", 0: ":(", "I": [1, 0]}
>>> d # Noter que l'ordre d'insertion n'est pas conservé
{0: ':(', 'I': [1, 0], 'café': ':)'}
```

— Connaître le nombre des couples clé→valeur contenu - `len(dict)` :

```
>>> len(d)
>>> 3
```

— Savoir si une **clé** appartient au dictionnaire - `cle in dict` :

```
>>> "I" in d
True
>>> "Café" in d
False
```

— Savoir si une **valeur** est associée à une clé d'un dictionnaire - `val in dict.values()` :

```
>>> d
{0: ':(', 'I': [1, 0], 'café': ':)'}
>>> ":-0" in d.values()
False
>>> ":)" in d.values()
True
```

— **Récupérer** la valeur associée à une clé - `dict[cle]` :

```
>>> x = d["I"]
>>> x
[1, 0]
>>> x[0]
1
>>> d["I"][0] # lire de droite à gauche: [0] premier élément de d["I"] c'est à dire [1, 0]
```

```
1
>>> d[0] # Une clé peut être un entier
':('
>>> d["Café"] # attention aux erreurs si la clé n'existe pas !
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
KeyError: 'Café'
```

— **Modifier** ou **ajouter** un couple clé→valeur - `dict[cle] = nouvelle_val` :

```
>>> d
{0: ':(', 'I': [1, 0], 'café': ':)'}
>>> cle = "café"
>>> d[cle] = "clop :-O" # la clé existe -> c'est une modification
>>> d
{0: ':(', 'I': [1, 0], 'café': 'clop :-O'}
>>> d["J"] = [0, 1] # la clé n'existe pas, c'est un ajout
>>> d # attention, les couples ne sont pas ordonnés!
{0: ':(', 'I': [1, 0], 'J': [0, 1], 'café': 'clop :-O'}
```

— **Supprimer** un couple clé→valeur - `del d[cle]` :

```
>>> d = {"Python": "de la balle !"}
>>> del d["Python"]
>>> d
{}
```

4.2 Parcourt d'un dictionnaire

— **Par les clés** - chaque clé est récupérée successivement - `cle in dict` :

```
>>> pts = {"A": [5, 3], "B": [-3, 5]}
>>> for c in pts:
...     print(c, "=>", pts[c])
...
A => [5, 3]
B => [-3, 5]
```

— **Intégrale** - Chaque couple est récupéré successivement - `cle, val in dict.items()` :

```
>>> pts = {"A": [5, 3], "B": [-3, 5]}
>>> for c, v in pts.items():
...     print("{}({};{})".format(c,v[0],v[1]))
...
A(5;3)
B(-3;5)
```

— **Par les valeurs** - chaque valeur est récupérée successivement - `val in dict.values()` :

```
>>> for coord in pts.values():
...     coord[1] -= 1
...
>>> pts
{'A': [5, 2], 'B': [-3, 4]}
```

4.3 Autres opérations utiles

— **Création** via `dict(...)`, en «compréhension» ou via `zip(list1,list2)` :

```
>>> # genre fonction
>>> d = dict(prenom="bob", nom="l'eponge", age=4)
>>> d
{'nom': "l'eponge", 'age': 4, 'prenom': 'bob'}
>>> # à partir d'une liste de tuple
>>> l = [("prenom", "bob"), ("nom", "l'eponge"), ("age", 4)]
>>> dict(l)
{'nom': "l'eponge", 'age': 4, 'prenom': 'bob'}
>>> # en «compréhension»
>>> {x: x**2 for x in range(10) if x not in (0,1,5,8)}
{2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 9: 81}
>>> # en zippant deux listes de même taille
>>> z = zip(("a", "b", "c"), (0, 1, 2))
>>> dict(z)
{'a': 0, 'c': 2, 'b': 1}
>>> dict(zip(list("abcdefghijklmnopqrstuvwxyz"), range(26)))
{'a': 0, 'c': 2, 'b': 1, 'e': 4, 'd': 3, 'g': 6, 'f': 5, 'i': 8, ...
```

— **Lecture** «sécurisée» - `dict.get(cle[, defaut])` :

```
>>> d = {"café": ":)", 0: ":(", "I": [1, 0]}
>>> d.get("Café") # si «defaut» n'est pas précisé, retourne None lorsque la clé n'existe pas.
>>> d.get("café")
':)'
>>> d.get("Café", 5) # si la clé n'est pas trouvée, retourne defaut=5
5
>>> d.get("café", 5) # sinon, retourne la valeur associée.
':)'
```

— **Écriture** «sécurisée» - `dict.setdefault(cle[, defaut])` :

```
>>> d.setdefault("café", ":(") # pas de modification, la clé existe !
':)'
>>> d.setdefault("Café") # la valeur par défaut est None
>>> d
{0: ':(', 'I': [1, 0], 'Café': None, 'café': ':)'}
>>> del d["Café"]
>>> d.setdefault("Café", ':]')
':]'
>>> d
{0: ':(', 'I': [1, 0], 'Café': ':]', 'café': ':)'}
```

— **Récupérer et supprimer** un couple - `dict.pop(cle[, defaut])` :

```
>>> cles = [0, 'café', 'i']
>>> for c in cles:
...     ret = d.pop(c, None) #  defaut=None -> valeur renvoyée si la clé n'existe pas
...     print(ret)
...
:(
:)
None
>>> d
{'I': [1, 0], 'Café': ':]'}
>>> d.pop('i') # si defaut est omis et que la clé n'existe pas -> erreur !
Traceback (most recent call last):
```

```
        File "<stdin>", line 1, in <module>
KeyError: 0
```

— **Récupérer et supprimer** un couple choisi «au hasard» - `dict.popitem()` :

```
>>> # Utile pour parcourir «destructivement» un dictionnaire
>>> d = {0: ':(', 'I': [1, 0], 'café': ':)'}
>>> while len(d): # rappel: 0 -> False, tout autre entier -> True
...   print("len(d) =", len(d))
...   cle, val = d.popitem()
...   print(cle, "=>", val, "et len(d) =", len(d))
...
0 => :( et len(d) = 2
I => [1, 0] et len(d) = 1
café => :) et len(d) = 0
>>> d # le dictionnaire est vide !
{}
```

— **Mettre à jour** un dictionnaire à partir d'un autre - `dict.update()` :

```
>>> d1 = {"A": (1,2), "B": (5, 3)}
>>> majd = {"O": (0, 0), "B": (-5, -3)}
>>> d1.update(majd)
>>> d1
{'A': (1, 2), 'B': (-5, -3), 'O': (0, 0)}
```

CHAPITRE 5

Fichiers - open

Pouvoir lire et/ou écrire dans un fichier est indispensable lorsqu'on souhaite, par exemple, récupérer/sauvegarder des informations entre deux exécutions d'un même programme. On parle alors de **persistance** de l'information.

5.1 Lire/écrire du texte dans un fichier

— **Créer** un nouveau fichier et y **écrire** - `fich = open(<nom_fichier>, 'w')` et `fich.write(chaine)` :

```
>>> # Dans quel dossier suis-je ?
>>> from os import getcwd
>>> getcwd()
'/home/etienne/Python/Fichiers'
>>> # votre fichier sera créé dans ce répertoire.
>>> f = open('test.txt', 'w') # 'w' pour write -> écrire.
>>> # f est un ojet qui représente le fichier créé.
>>> f.write('Je découvre la gestion des fichiers.')
36
>>> f.write('Ça paraît assez simple ...')
26
>>> # C'est fini ? fermer votre fichier!
>>> f.close()
```

— **Lire** un fichier existant - `fich = open(<nom_fichier>)` et `fich.read()` :

```
>>> f = open('test') # oups ...
Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'test'
>>> f = open('test.txt') # ou open('test.txt', 'r')
>>> contenu = f.read()
>>> contenu
'Je découvre la gestion des fichiers.Ça paraît assez simple ...'
>>> f.read()
''
>>> # car vous avez déjà tout lu ... il est temps de fermer !
>>> f.close()
```

— **Ajouter** du texte à un fichier existant - `fich = open(<nom_fichiers>, 'a')` :

```
>>> f = open('test.txt', 'a') # 'a' pour append -> ajouter
>>> f.write('\nligne2\nligne3\n\nblah blah') # rappel \n symbolise le caractère saut de ligne.
25
```

```
>>> f.close()
>>> f = open('test.txt')
>>> print(f.read())
Je découvre la gestion des fichiers.Ça paraît assez simple ...
ligne2
ligne3

blah blah
>>> f.close()
```

— Récupérer la liste des lignes d'un fichier - `fich.readlines()` :

```
>>> f = open('test.txt')
>>> lignes = f.readlines()
>>> lignes # noter que les caractères de saut de lignes - \n - sont conservés.
['Je découvre la gestion des fichiers.Ça paraît assez simple ...\n', 'ligne2\n', 'ligne3\n', '\
>>> f.close()
```

— **Parcourir** les lignes d'un fichier - `for ligne in fich:`

```
>>> copie = open('copie', 'w')
>>> orig = open('test.txt')
>>> i = 0
>>> for lgn in orig:
...     i += 1
...     lgn = str(i) + ": " + lgn
...     copie.write(lgn)
...
>>> orig.close()
>>> copie.close()
```

— Options courantes d'ouverture d'un fichier - `open(<nom_fichier>, option)` :

Option	Signification
`'r'`	ouverture en lecture (par défaut)
`'w'`	ouverture en écriture (un fichier existant est perdu)
`'x'`	ouverture en écriture sécurisée (erreur si le fichier existe)
`'a'`	ouverture en ajout à la fin du fichier s'il existe
`'b'`	mode binaire - on lit ou écrit via une chaîne d'octes `bytes`
`'t'`	mode texte (par défaut)

`'wt'` est équivalent à `'w'`. Pour voir un fichier comme une chaîne d'octets, utiliser `'rb'`, `'wb'` , etc.

5.2 Sauvegarder/restaurer des «objets»

`pickle` est un module qui facilite considérablement la sauvegarde/récupération de données Python par l'intermédiaire de fichiers.

Avertissement : Les fichiers doivent être ouverts en **mode binaire** - `'b'`.

— **Sauvegarder** une ou des données dans un fichier - `pickle.dump(donne, fichier)` :

```
>>> import pickle
>>> donnees = {"nom": 'Dupond', "Prénom": 'andré', "age": 32}
>>> fsauv = open('donnees.pickle', 'wb') # ouverture en mode binaire !!
>>> pickle.dump(donnees, fsauv)
>>> fsauv.close()
```

— **Restaurer** une donnée sauvée via pickle - `pickle.load(fichier)` :

```
>>> import pickle
>>> f = open('donnees.pickle', 'rb') # mode lecture binaire !!
>>> restaure = pickle.load(f)
>>> restaure
{'nom': 'Dupond', 'age': 32, 'Prénom': 'andré'}
>>> f.close()
```

5.3 Notion de chemin

Pour ouvrir un fichier qui ne se trouve pas dans le répertoire courant, il faut être capable d'indiquer où il se trouve dans l'arborescence du disque c'est à dire son **chemin**.

Voici un exemple (volontairement très simple !) d'organisation d'un disque :

```
dossier1/
     fichier1.txt
     ...
     ss_dossier/
             fichier2.ppm
             ...
fichier3.py
...
```

Les noms complets des fichiers dépendent d'un dossier de référence :

- **Chemin absolu** - depuis la «racine» du disque notée `/` :
  - de *fichier1.txt* : `/dossier1/fichier1.txt`
  - de *fichier2.ppm* : `/dossier1/ss_dossier/fichier2.ppm`
- **chemin relatif** - à partir d'un dossier particulier (souvent le dossier courant) :
  - de *fichier1.txt* à partir de **dossier1** : `fichier1.txt`
  - de *fichier3.py* à partir de **dossier1** : `../fichier3.py` (`..` ~ dossier parent)
  - de *fichier3.py* à partir de **ss_dossier** : `../../fichier3.py`

Note : Un programme possède toujours un dossier de référence appelé **répertoire courant**. C'est normalement le dossier qui contient le fichier du programme. Pour s'en assurer :

```
import os
# ...
rep_courant = os.getcwd() # cwd pour current working directory
print(rep_courant)
```

CHAPITRE 6

Modules - `import`

6.1 Module

Un **module** est basiquement un fichier `nom_module.py` ordinaire. Les variables, fonctions, classes qui y sont définies peuvent être **importées** afin d'être réutilisées.

Python est accompagné de nombreux modules ; ils forment sa librairie standard. On y trouve par exemple les modules *random*, *math*, *tkinter* et beaucoup d'autres.

— **Importer un module** et l'utiliser - `import <nom_module>` :

```
>>> import random # random -> aléatoire
>>> random.randint(1,6) # «dé electronique»
2
>>> def de(): # trop long à écrire ?
...     return random.randint(1,6)
...
>>> de()
3
>>> de()
1
>>> random.choice('abcdefghijklmnop') # caractère aléatoire de la chaîne
'h'
```

— Utiliser un **alias** - `import <nom_module> as <alias>` :

```
>>> import random as alea
>>> l = [1, 2, 3, 4]
>>> alea.shuffle(l)
>>> l
[1, 4, 2, 3]
```

— Importer une ou plusieurs fonctions d'un module - `from module import f1, f2, ...` :

```
>>> from math import sqrt as racine, exp, log as ln, e
>>> r = racine(5)
>>> r
2.23606797749979
>>> round(r,2)
2.24
>>> [(x, round(racine(x), 2) ) for x in [0,1,2,3,4,5]]
[(0, 0.0), (1, 1.0), (2, 1.41), (3, 1.73), (4, 2.0), (5, 2.24)]
>>> from random import uniform
>>> x = uniform(0, 100)
>>> ln(exp(x)) == x
```

```
True
>>> exp(ln(x)) == x
True
>>> e
2.718281828459045
>>> ln(e)
1.0
```

— Importer **tout** ce qu'un module définit - `from <nom_module> import *` :

```
>>> from tkinter import *
>>> fenetre = Tk() # une fenêtre devrait apparaître
>>> bouton = Button(fenetre, text="cliquez moi !", command=fenetre.destroy)
>>> # le bouton n'apparaît pas ??? normal, il faut encore le positionner
>>> bouton.pack()
```

Avertissement : Procéder de la sorte est **généralement déconseillé** car de nombreux «noms» sont alors introduis dans l'interpréteur Python ce qui peut être source de conflits (en cas de multiples imports par exemple). Vous pouvez toutefois utiliser cette facilité pour écrire de petits programmes de découvertes de tel ou tel module :

```
>>> # La fonction dir(truc) sert à connaître les noms définis dans le contexte de «truc»
>>> dir() # si «truc» est omis, c'est le contexte courant
['__builtins__', '__doc__', '__name__', '__package__']
>>> a = 5
>>> dir() # le nom «a» existe à présent
['__builtins__', '__doc__', '__name__', '__package__', 'a']
>>> import tkinter
>>> dir() # seul le nom «tkinter» (ou plutôt l'espace de nom) est importé
['__builtins__', '__doc__', '__name__', '__package__', 'a', 'tkinter']
>>> from tkinter import *
>>> dir() # et maintenant ? argh !
['ACTIVE', 'ALL', 'ANCHOR', 'ARC', 'At', 'AtEnd', 'AtInsert', 'AtSelFirst', 'AtSelLast', 'BASE
```

6.2 Mon module : exemple

— Créer un fichier d'extension `.py`. Dans cet exemple, il s'appelle `monModule.py` :

```
# définitions de monModule.py
CONST = 3.14
def truc():
    print("Salut cowboy.")
```

— Démarrer l'interpréteur *depuis le dossier qui contient le fichier* `monModule.py` :

```
>>> from monModule import *
>>> CONST
3.14
>>> truc()
Salut cowboy.
```

Note : Votre module sera chargé pourvu que Python puisse le trouver ! Pour savoir où Python cherche les modules :

```
>>> import sys
>>> sys.path # affiche la liste des dossiers de recherche des modules
```

— **Tester** son module - `if __name__ == '__main__':`

```
# ajouter les lignes suivantes à votre module puis exécuter le normalement

if __name__ == '__main__':
    # __name__ vaut '__main__'
    # seulement si Python est
    # directement appelé sur ce fichier
    # code pour tester le module:
    print(CONST)
    truc()
```

6.3 Notion de Paquet (avancé)

Un **paquet** - *package* - sert à regrouper logiquement plusieurs modules. En pratique, c'est un dossier caractérisé par la présence d'un fichier `__init__.py` (qui peut être vide). Outre ce fichier «spécial», on y trouve les modules et éventuellement d'autres paquets...

— Un paquet est donc un **dossier de la forme** :

```
paquet/
        __init__.py
        module1.py
        module2.py
        sousPaquet/
                __init__.py
                autreModule.py
                ...
        ...
```

— **Importer un paquet** (revient en fait à «charger» son `__init__.py`) - `import <paquet>` :

```
import paquet
# si son __init__.py définit la fonction «truc»
paquet.truc() # ok
truc() # pas ok !
from paquet import truc
truc() # là ok
```

— Importer un module contenu dans un paquet - `import <paquet>.<module>` ou aussi `from <paquet> import <module>` :

```
import paquet.module1
# si module1.py définit la fonction «bidulle»
paquet.module1.bidulle() # ok.

# vous pouvez utiliser un alias
import paquet.module1 as pm1
pm1.bidulle() # ok

# ou encore
from paquet import module1
module1.bidulle() # ok

# Enfin, pour importer «autreModule»
import paquet.sousPaquet.autreModule
```

— Importer un (ou plusieurs) «objets» définis dans un module d'un paquet - `from <paquet>.<module> import obj1, obj2, ...`.
— Sens particulier de `from <paquet> import *`.
On pourrait penser que ça charge tous les (sous)modules de <paquet> mais ce n'est en général pas le cas ; La convention est la suivante :

si le fichier `__init__.py` définit une liste nommée `__all__`, elle est utilisée comme la liste des noms de modules qui devraient être chargés si `from <paquet> import *` est utilisé.

CHAPITRE 7

Objets - `class`

Un **objet** est une entité informatique qui *possède* :
- des **attributs** : noms qui font référence à d'autres objets.
  si l'objet `o` possède l'attribut `x`, l'écriture `o.x` permet d'accéder à cet attribut et l'écriture `o.x = ...` permet de modifier cet attribut (ou de le créer s'il n'existait pas).
- des **méthodes** : fonctions internes qui détermine son «comportement» ;
  si l'objet `o` possède une méthode `f`, l'écriture `o.f(...)` sert à déclencher le comportement correspondant.

Chaque objet informatique est *caractérisé* par :
- une **identité** : son adresse mémoire - `id(obj)` ;
- un **type** : la *classe* à laquelle il appartient - `type(obj)` ;
- une **valeur**

Comme objets vous connaissez déjà les *int*, *float*, *str*, *list*, *dict* ... mais aussi les *fonctions*, *modules* ... ; Exemple :

```
>>> o = -5 # o est une référence vers un objet de type int
>>> # tout objet de type (classe) int possède un attribut «denominator»
>>> o.denominator
1
>>> # il possède aussi divers méthodes internes dont  __bool__ qui
>>> # donne la valeur de condition (dans un if par exemple) d'un tel objet
>>> o.__bool__()
True  # tout entier non nul est considéré comme vrai
>>> dir(o) # affiche la liste des attributs et méthodes de l'objet ...
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__divmod__',
```

7.1 Définir une classe d'objets - `class`

Pour construire des objets, on commence par définir une **classe** qui sert à préciser ce que les objets auront en commun. Voici la syntaxe d'une telle déclaration :

```
class NomClasse:
        """ documentation de
        cette classe ...
        """

        # déclaration des attributs et méthodes de cette classe
```

Cela fait, pour construire des objets de type `NomClasse`, on appelle la classe un peu comme une fonction :

```
>>> # définition d'une classe Test minimale
>>> class Test:
...     "blah blah"
...
>>> # Création d'un objet par l'intermédiaire de cette classe
>>> o1 = Test()
>>> o1
<__main__.Test object at 0x23f1690>
>>> # 0x23f... ? c'est l'adresse en mémoire vive de l'objet
>>> o2 = Test()
>>> o2
<__main__.Test object at 0x23f1690>
>>> # observez que o1 et o2 ont des adresses différences
>>> # donc ils sont distincts.
>>> type(o1)
<class '__main__.Test'>
>>> o1 == o2
False
```

Une classe est en quelque sorte un «moule à objet». Chaque objet créé à partir d'elle possède une **identité** (son adresse en mémoire vive) et un **type**, la classe elle-même.

L'opération qui consiste à créer un objet à partir de sa classe est appelé **instantiation**.

7.2 Attributs - `def __init__(self,...):`

Même si les objets que nous avons créés sont distincts puisqu'ils ont des adresses mémoires différentes, ils ont la même valeur (vide pour l'instant).

Pour les différencier, nous allons leur donner des **attributs** que nous initialiserons avec des valeurs différentes. Pour cela, on définit une **méthode spéciale** `__init__()` qui sera appelée automatiquement au moment de l'instantiation :

```
class Individu:
    " ici la documentation ... "

    def __init__(self, nom, prenom, age):
        self.nom = nom.upper()
        self.prenom = prenom.capitalize()
        self.age = int(age)

# Créons quelques «Individus»
un_individu = Individu('durand', 'john', 16)
autre_individu = Individu('dupont', 'alexandre', 36.7)
print("{i.prenom} {i.nom} a {i.age} ans".format(i=un_individu))
print("{i.prenom} {i.nom} a {i.age} ans".format(i=autre_individu))
```

L'exécution de se code donne

```
John DURAND a 16 ans
Alexandre DUPONT a 36 ans
```

Voilà ce qui se passe lors de l'instantiation `Individu('durand', 'john', 16)` :

1. Python crée un objet *o* (vide pour l'instant) de type `Individu` ;
2. puis, il appelle `__init__` en lui fournissant :

— automatiquement **cet objet** *o* comme premier argument ; donc, dans le code, `self` représente l'objet *o* fraîchement créé !

— les arguments fournis lors de l'instantiation pour les paramètres `nom`, `prenom`, `age`.

3. `self.nom = ...` signifie créer l'attribut `nom` pour **cet objet** (désigné par `self`) et lui donner la valeur indiquée ;
4. les autres lignes définissent de la même façon les attributs `prenom` et `age` pour cet objet.

Important : Retenir : dans le code d'une classe, le mot `self` représente toujours l'objet individuel sur lequel on est en train d'agir. Ainsi, l'écriture `self.attribut` désigne la valeur particulière de l'attribut de **cet objet**.

7.3 Méthodes - `def agir(self,...):`

Ajoutons deux méthodes `est_majeur` et `vieillir` à nos objets de type `Individu` :

```
class Individu:
    # ...

    def est_majeur(self):
        if self.age >= 18:
            return True
        else:
            return False

    def vieillir(self, ans=1):
        self.age = self.age + ans

# ...
john = un_individu
print(john.est_majeur()) # -> False
john.vieillir()
print(john.age) # -> 17
john.vieillir(2)
print(john.age + " donc majeur: " + john.est_majeur()) # -> 19 donc majeur: True
autre_individu.vieillir(5)
if autre_individu.est_majeur():
    print(autre_individu.prenom + " a " + autre_individu.age + "ans !")
# -> Alexandre a 41 ans !
```

7.4 Méthodes spéciales - `__meth__(self,...)`

Ces méthodes ont la particularité d'être **appelées automatiquement** par Python dans *certaines circonstances*. Par exemple, la méthode spéciale `__init__` est appelée automatiquement lors de l'**instantiation** d'une classe (création d'un objet).

À finir ...

CHAPITRE 8

Réseau - `socket`

Pour faire **communiquer** deux programmes (ou plus) sur le réseau, on utilise des *canaux de communication* appelés **sockets**. Voici le minimum à avoir à l'esprit pour pouvoir s'en servir effectivement :

- Un tel programme commence par **importer le module socket** - `import socket`.
- Pour joindre un programme «prog» qui tourne sur une machine A, on a besoin :
  - de l'**adresse IP de la machine A** qui est de la forme `xxx.xxx.xxx.xxx` (ex : 192.168.1.1)
  - d'un numéro, qui sert à identifier «prog» sur cette machine, appelé **port**.
- Il faut distinguer deux types de *canaux de communication* ou *sockets* :
  - **socket serveur** : elles écoutent le réseau en vue d'initier la communication - penser à l'opérateur téléphonique.
  - **socket client** : elles servent à transmettre et recevoir effectivement les messages échangés - penser au téléphone lui-même.
- Les messages émis sur le réseau doivent-être des chaînes d'octets - de type *bytes* :
  - **encoder** : action de convertir une chaîne de caractères - *str* - en une chaîne d'octets - *bytes* - `str.encode('utf-8')`.
  - **décoder** : action de convertir une chaîne d'octets - *bytes* - en une chaîne de caractères - *str* - `bytes.decode('utf-8')`.

8.1 Création du serveur

8.1.1 Structure du programme

1. **Création d'un canal de communication** - `serveur = socket.socket()`

```
import socket
serveur = socket.socket() # création du «canal de communication»
```

2. Choix du port d'**écoute** (supérieur à 1000, ici 6789) - `serveur.bind(adresse)` et `.listen(1)`

```
ip = '' # chaîne vide pour adresse locale; sinon mettre l'ip de la machine
port = 6789 # choix du port (doit-être supérieur à 1000)
adresseServeur = ip, port # adresse complète de ce programme ...
serveur.bind(addresseServeur) # ... qu'on associe au canal de communication
serveur.listen(1) # puis on lance l'écoute de ce canal.
```

3. **Attente bloquante d'une connexion** d'un nouveau client - `serveur.accept() -> client, adresseClient`

   Suite à cette instruction, la variable *client* est un canal qui permettra de gérer la communication avec le nouveau client ; la variable *adresseClient* contient son adresse IP et son port.

```
client, adresseClient = serveur.accept() # «bloquant» ...
# ... = le code situé après la ligne précédente
# ne sera exécuté que lorsqu'un client se sera effectivement connecté.
```

4. **Envoi** d'un message (*bytes*) vers le client - `client.send(message)`

```
mess = 'Salut !' # chaîne de caractères -> str
messEnc = mess.encode('utf-8') # encodage: conversion str vers bytes
client.send(messEnc) # envoie du message
# en une seule ligne ? client.send('Salut'.encode('utf-8'))
```

5. **Réception** bloquante d'un message (*bytes*) envoyé par le client (1024 octets maximum) - `recuEnc = client.recv(1024)`

```
recuEnc = client.recv(1024) # bloque jusqu'à réception
recu = recuEnc.decode('utf-8') # on a recu une chaîne d'octets : il faut la «décoder» (bytes ->
# en une seule ligne ? recu = client.recv(1024).decode('utf-8')
```

6. **Déconnexion et arrêt** du serveur - `.close()`

```
client.close()
serveur.close()
```

8.1.2 Exemple complet

```
import socket

# Identification réseau de ce programme
IP = ''
PORT = 6789
ADRESSE = IP, PORT

# création d'un canal de communication - socket - de type serveur
serveur = socket.socket() # création
serveur.bind(ADRESSE) # association à l'adresse du programme ...
serveur.listen(1) # écoute du réseau

# on attend une connexion entrante
client, adresseClient = serveur.accept()
print('Connexion de', adresseClient)

# Boucle de dialogue (ici de type «perroquet»)
while True:
    recu = client.recv(1024)
    if len(recu) == 0:
        print('Erreur de réception.')
        break
    else:
        recu = recu.decode('utf-8') # décodage du message reçu
        print('Réception de:', recu)
        reponse = recu.upper() # «perroquet»
        print('Envoi de :', reponse)
        reponse = reponse.encode('utf-8') # encodage du message à émettre
        n = client.send(reponse)
        if n != len(reponse):
            print('Erreur envoi.')
            break
```

```
        else:
            print('Envoi ok.')

# si on est là c'est que la connexion est rompue ;
# il faut alors fermer les canaux de communication
print('Fermeture de la connexion avec le client.')
client.close()
print('On se débranche')
serveur.close()
```

8.2 Création du client

8.2.1 Structure du programme

1. **Création d'un canal** pour gérer la communication - `client = socket.socket()`

```
import socket
client = socket.socket()
```

2. **Connexion au serveur** en utilisant son adresse et son port - `client.connect(adresseServeur)`

```
adrServ = '', 6789 # mettre la véritable ip du serveur à joindre à la place de ''
client.connect(adrServ)
```

3. **Envoi** d'un message vers le serveur - `client.send(message)`

```
mess = 'Bonjour'
messEnc = mess.encode('utf-8')
client.send(messEnc)
```

4. **Réception** bloquante d'un message du serveur (1024 octets maximum) - `client.recv(1024)`

```
recuEnc = client.recv(1024)
recu = recuEnc.decode('utf-8')
```

5. **Déconnexion** - `client.close()`

```
client.close()
```

8.2.2 Exemple complet

```
import socket

IPSERVEUR = '' # pour test en local; sinon mettre la vraie ip
PORT = 6789

client = socket.socket()
client.connect((HOST, PORT))
print('Connexion vers ' + HOST + ':' + str(PORT) + ' reussie.')

while True:
    message = input('>>> ')
    print('Envoi de :', message)
    message = message.encode('utf-8')
    n = client.send(message)
```

```
    if n != len(message):
        print('Erreur envoi.')
        break
   else:
        print('Envoi ok.')
        print('Reception...')
        recu = client.recv(1024)
        recu = recu.decode('utf-8')
        print('Recu :', recu)

print('Déconnexion.')
client.close()
```

Pour en apprendre plus sur le sujet voir, par exemple, le cours de Swinnen.

CHAPITRE 9

Aléatoire - `random`

Prérequis :
- les types de nombres et de variables
- les listes

Le module qui gère l'aléatoire en python est le module `random` :

```
from random import *
```

9.1 Les nombres au hasard

1. Nombre flottant (réel) entre 0 et 1 :

```
>>> random()
0.49852220170348827
```

2. Nombre flottant entre deux bornes.
   Pour tirer un nombre au hasard entre 10 et 12.5 :

```
y = uniform(10, 12.5)
```

3. Nombre entier entre deux bornes :

```
>>> randint(0, 20)
20
```

   Les deux bornes sont incluses dans les cas possibles.

4. Générer une probabilité p
   32 % de chance de gagner à ce jeu...

```
if random() <= 0.32:
    print("gagné")
else:
    print("perdu")
```

9.2 Les listes aléatoires

1. Créer une liste aléatoire de 1000 nombres entiers entre 0 et 100 :

```
liste = []
for i in range(1000):
    liste.append( randint(0, 100) )
```

2. Mélanger une liste :

```
shuffle(liste)
```

3. Choisir au hasard un élément d'une liste :

```
N = choice(liste)
```

3. Extraire au hasard k éléments d'une liste
   Extrait aléatoirement trois éléments de la liste :

```
jeu = sample(liste, 3)
```

> **Avertissement :** La liste n'est pas modifiée (les 3 éléments choisis sont encore présents dans la liste).

9.3 Exemple complet

```
from random import *

# un jeu de carte
couleur = ["pique", "coeur", "carreau", "trèfle"]
hauteur = ["As", "Roi", "Dame", "Valet", "Dix", "Neuf", "Huit", "Sept"]

jeu = []
for c in couleur:
    for h in hauteur:
        jeu.append(h+" de "+c)

print("Le jeu neuf :")
print(jeu)

# mélanger
shuffle(jeu)
print("Le jeu mélangé")
print(jeu)

# Choisir au hasard le nombre de carte à donner
N = randint(3, 10)
print("Je donne "+str(N)+" cartes")

# Donner N cartes
donne = sample(jeu, N)
print("Les voilà :")
print(donne)

# Attention les cartes données sont encore dans le jeu
print("Le jeu est-il complet ?")
print(len(jeu))
```

CHAPITRE 10

Bases de données - `sqlite3`

Les bases de données peuvent être un moyen efficace de mémoriser des données, surtout si elles se structurent naturellement sous forme de table.

La base de données se présentera physiquement sous la forme d'un fichier de type *sq3* sur le disque dur.

Il sera possible de compléter, de modifier et bien sûr de lire le contenu de la base de données.

Une base de donnée peut contenir plusieurs tables.

Chaque table est composée de multiple lignes ayant la même structure.

Prérequis :
- les chaînes de caractères
- les listes

10.1 Importation du module sqlite

10.2 Création d'une base de donnée

1. **Connexion** à la base de données :

```
connexion = sqlite3.connect("bd-celebrites.sq3")
```

Cette instruction crée la base si elle n'existe pas encore (le fichier est créé dans le répértoire courant).

2. **Création d'un curseur** sur la base :

```
curseur = connexion.cursor()
```

Le curseur servira ensuite à manipuler la base de données.

Dans toute la suite, on exécute des commandes SQL (données sous forme de chaînes de caractères).

3. **Création d'une table** dans la base :

```
curseur.execute("CREATE TABLE IF NOT EXISTS celebrites (nom TEXT, prenom TEXT, annee INTEGER)")
```

La commande crée la table "celebrite" si elle n'existe pas encore. On définit sa structure au moment de sa création : chaque ligne de la table est constituée d'un nom, d'un prénom et d'une année.

4. **Ajout de données** à la table :

```
    curseur.execute("INSERT INTO celebrites(nom, prenom, annee) VALUES('Turing','Alan', 1912)")
    curseur.execute("INSERT INTO celebrites(nom, prenom, annee) VALUES('Lovelace','Ada', 1815)")
    curseur.execute("INSERT INTO celebrites(nom, prenom, annee) VALUES('Shannon','Claude', 1916)")
    curseur.execute("INSERT INTO celebrites(nom, prenom, annee) VALUES('Hooper','Grace', 1906)")
```

5. **Valider** l'enregistrement dans la base :

```
    connexion.commit()
```

Avertissement : Sans cette instruction rien ne sera réellement enregistré dans la base de données.

6. **Fermer** la base :

```
    connexion.close()
```

10.3 Lecture de la base de données

```
connexion = sqlite3.connect("bd-celebrites.sq3")
curseur   = connexion.cursor()

curseur.execute("SELECT * FROM celebrites")
resultat  = curseur.fetchall()
```

La liste `resultat` contient alors tous les enregistrements.

10.4 Modifier un enregistrement

```
connexion = sqlite3.connect("bd-celebrites.sq3")
curseur   = connexion.cursor()

curseur.execute("UPDATE celebrites SET prenom='Alan Mathison' WHERE nom='Turing'")
connexion.commit()
```

10.5 Pour aller un peu plus loin

1. Une requête de recherche ciblée :

```
    curseur.execute("SELECT * FROM celebrites WHERE nom = 'Turing'")
    resultat = list(curseur)
    print(resultat)
```

La requête recherche et extrait seulement les lignes de la table dont l'entrée [nom] est 'Turing'. On transforme (transtype) le curseur en liste avant de l'afficher en tant que résultat.

2. Utiliser une variable dans une requete :

```
    qui = "Shannon"
    curseur.execute("SELECT * FROM celebrites WHERE nom = '" + qui + "'")
    quand = 1515
    curseur.execute("SELECT * FROM celebrites WHERE annee >= " + str(quand))
```

10.6 Exemple complet

```
import sqlite3
connexion = sqlite3.connect("bd-celebrites.sq3")
curseur = connexion.cursor()

# creation d'une table
curseur.execute("CREATE TABLE IF NOT EXISTS celebrites (nom TEXT, prenom TEXT, annee INTEGER)")

# ajout de données à la base
curseur.execute("INSERT INTO celebrites(nom, prenom) VALUES('Turing','Alan', ???)")
curseur.execute("INSERT INTO celebrites(nom, prenom) VALUES('Lovelace','Ada')")
curseur.execute("INSERT INTO celebrites(nom, prenom) VALUES('Shannon','Claude')")
curseur.execute("INSERT INTO celebrites(nom, prenom) VALUES('Hooper','Grace')")

# valider l'enregistrement dans la base
connexion.commit()

# charger toutes les données de la base dans un tableau
curseur.execute("SELECT * FROM celebrites")
resultat = curseur.fetchall()

# affichage en console du résultat
print(resultat)
for r in resultat:
    print(r[0],r[1],r[2])

# Modifier un enregistrement
curseur.execute("UPDATE celebrites SET prenom='Alan Mathison' WHERE nom='Turing'")
connexion.commit()

# Accèder à l'enregistrement
curseur.execute("SELECT * FROM celebrites WHERE nom = 'Turing'")
resultat = list(curseur)
print(resultat)

# fermer la base
connexion.close()
```

CHAPITRE 11

Persistance des données

La **sérialisation** (en anglais *serialization* ou *marshalling*) est le processus par lequel on transforme des données présentes en mémoire en une suite de codes qui pourra :

— être enregistrée dans un fichier. Cela permet de **rendre persistant** un « objet informatique » (il survivra à un redémarrage du système, au terme duquel on pourra le reconstituer à l'identique) ;
— être transportée sur un réseau. Cela permet de **transférer à un ordinateur distant** un « objet informatique » (par exemple *via* un réseau).

L'opération réciproque (le décodage des informations pour créer une copie conforme à l'original), s'appelle la **désérialisation** (*deserialization* ou *unmarshalling*).

Il y a plusieurs façon de sérialiser des données :

— à l'aide de *Module json* pour des « données simples » ;
— à l'aide de *Module pickle* pour des « objets Python » plus complexes ;
— à l'aide de XML (proche de JSON, mais plus verbeux, plus riche et plus complexe) ;
— il arrive souvent que les données soient en outre compressées, afin d'économiser la place sur le disque dur et/ou la bande passante sur les réseaux. Voir *Compresser avec gzip*.

11.1 Module `json`

JSON (**J**ava**S**cript **O**bject **N**otation) est un format de données textuelles permettant de représenter des informations structurées. JSON est utilisable par de nombreux langages de programmation. C'est une alternative moins verbeuse (mais également moins « parlante ») à XML.

En Python, la structure de base sera une liste, ou un dictionnaire dont les clés devront impérativement être des chaînes de caractères. Les valeurs seront uniquement des chaînes, des nombres, les valeurs `True`, `False` ou `None` (traduit en `null` en terminologie JSON), ou d'autres listes ou dictionnaires respectant les mêmes contraintes. Le résultat après conversion sera une chaîne de caractères représentant cette structure au format JSON. Cette chaîne pourra être enregistrée dans un fichier ou transmise à un autre ordinateur via le réseau (c'est surtout dans ce dernier contexte que JSON est utile, cf. AJAJ versus AJAX).

Remarque 1 : le stockage JSON n'est possible que pour certains objets Python. Voir par ici pour les plus curieux...

Remarque 2 : JSON est incapable de figer l'ordre des données (dictionnaire Python). Si c'est un point important (utilisation d'OrderedDict), il faut impérativement recourir à Pickle...

11.1.1 `json.dump` et `json.dumps` : sérialisation avec JSON

Conversion en chaîne de caractères pour envoi via réseau

```
>>> personne = { "id" : 1, "nom" : "DUPONT", "prénom" : "Jean", "âge" : 25, "marié" : False, "conjoi
>>> import json
>>> json.dumps(personne)
'{"nom": "DUPONT", "\\u00e2ge": 25, "mari\\u00e9": false, "conjoint": null, "pr\\u00e9nom": "Jean", '
```

Enregistrement dans un fichier

```
>>> with open("test.js","w") as f:
...     json.dump(personne,f)
...
```

11.1.2 `json.load` et `json.loads` : désérialisation avec JSON

Conversion d'une chaîne de caractères contenant du code JSON en structure Python :

```
>>> import json
>>> json.loads('[ 1, 2, { "a":1, "b":[1,2], "c":true, "d":null}]')
[1, 2, {'a': 1, 'b': [1, 2], 'c': True, 'd': None}]
```

Chargement depuis un fichier :

```
>>> with open("test.js","r") as f:
...     d = json.load(f)
...
>>> print(d)
{'conjoint': None, 'id': 1, 'marié': False, 'nom': 'DUPONT', 'prénom': 'Jean', 'âge': 25}
```

11.1.3 Tableau récapitulatif et astuce mnémotechnique

Fonction	Travaille avec des chaînes	Travaille avec des fichiers
Lit	`json.loads`	`json.load`
Écrit	`json.dumps`	`json.dump`

Astuce : le **s** dans `json.loads` et `json.dumps` fait référence aux chaînes de caractères (**s**tring en anglais).

>> *Tout savoir sur la bibliothèque Python consacrée à JSON*.

11.2 Module `pickle`

Le module `pickle` de Python 3 implémente une meilleure sérialisation que ce que l'on peut obtenir avec *Module json* : il permet de transformer en suite de bits des « objets » Python complexes. Seul Python peut désérialiser le résultat, Pickle n'est donc **pas** un bon moyen d'échanger des données entre des programmes écrits dans différents langages.

11.2.1 `pickle.dump` et `pickle.dumps` : sérialisation avec Pickle

Conversion en « chaîne de caractères **binaires** » (type `bytes`) pour envoi via réseau :

```
>>> personne = { "id" : 1, "nom" : "DUPONT", "prénom" : "Jean", "âge" : 25, "marié" : False, "conjoi
>>> import pickle
>>> pickle.dumps(personne)
b'\x80\x03}q\x00(X\x03\x00\x00\x00nomq\x01X\x06\x00\x00\x00DUPONTq\x02X\x04\x00\x00\x00\xc3\xa2geq\x0
\x00\x00\x00mari\xc3\xa9q\x04\x89X\x08\x00\x00\x00conjointq\x05NX\x07\x00\x00\x00pr\xc3\xa9nomq\x06X
\x00Jeanq\x07X\x02\x00\x00\x00idq\x08K\x01u.'
```

Enregistrement dans un fichier :

```
>>> with open("test.pkl", "wb") as f:
... pickle.dump(personne, f)
...
```

Remarque : l'option `b` *d'ouverture de fichier en* **mode binaire** *est importante !*

11.2.2 `pickle.load` et `pickle.loads` : désérialisation avec Pickle

Conversion d'un objet `bytes` (« chaîne de caractères **binaires** ») en structure Python :

```
>>> pickle.loads(b'\x80\x03]q\x00(K\x01K\x02}q\x01(X\x01\x00\x00\x00aq\x02K\x01X\x01\x00\x00\x00cq\x0
\x01\x00\x00\x00bq\x04]q\x05(K\x01K\x02eX\x01\x00\x00\x00dq\x06Nue.')
[1, 2, {'a': 1, 'b': [1, 2], 'c': True, 'd': None}]
```

Chargement depuis un fichier :

```
>>> with open("test.pkl","rb") as f:
... d = pickle.load(f)
...
>>> print(d)
{'conjoint': None, 'id': 1, 'marié': False, 'nom': 'DUPONT', 'prénom': 'Jean',
'âge': 25}
```

Remarque : l'option `b` *d'ouverture de fichier en* **mode binaire** *est importante !*

11.2.3 Tableau récapitulatif et astuce mnémotechnique

Fonction	Travaille avec des chaînes **binaires** (`bytes`)	Travaille avec des fichiers
Lit	`pickle.loads`	`pickle.load`
Écrit	`pickle.dumps`	`pickle.dump`

Astuce : le **s** dans `pickle.loads` et `pickle.dumps` fait référence aux chaînes de caractères (**s**tring en anglais).

>> *Tout savoir sur la bibliothèque Python consacrée à Pickle.*

11.3 Compresser avec `gzip`

Il peut être utile de compresser certains « objets » Python volumineux (structures textuelles de grande taille, comme par exemple une base de données cartographiques libres issue du projet OpenStreeMap - la plus grosse dépasse les 30 Go !). Plusieurs bibliothèques permettent cela, entre autres `zipfile` et `gzip`. La 1ère est plus complète et gère de nombreux algorithmes de compression, mais son emploi est bien plus complexe que la seconde. C'est donc elle (`gzip`) qu'on illustrera dans la suite.

Note : cette fiche suppose que vous savez à quoi sert le module `pickle`. Voir *Module pickle*.

11.3.1 Utilisation conjointe de `gzip.open` et `pickle.dumps` pour compresser des données sérialisées

```
>>> import gzip, pickle
>>> d={}
>>> for i in range(100000):
...     d[i]=chr(i % 26 + 65)*100
# NE SURTOUT PAS faire print(d), ce serait trèèèèèès long !
>>> with gzip.open("test.pkz", "wb") as f:
...     f.write(pickle.dumps(d))
```

L'effet sur la taille des données est drastique (division par 10 !) :

```
>>> import sys
>>> sys.getsizeof(d)
6291736
>>> d2=open("test.pkz", "rb").read()
>>> sys.getsizeof(d2)
605077
```

11.3.2 Utilisation conjointe de `gzip.open` et `pickle.loads` pour décompresser des données sérialisées

```
>>> with gzip.open("test.pkz","rb") as f:
...
datas = pickle.loads(f.read())
>>> datas == d
True
```

11.3.3 Ressources sur l'utilisation du module `zipfile`

Note : les liens donnés ci-dessous concernent Python 2, il faudra donc adapter le contenu à Python 3 !

- http://pymotw.com/2/zipfile/
- http://effbot.org/librarybook/zipfile.htm