Bien démarrer avec python

...un échantillon de ce qui peut servir à l'IPSL dans un python standard ☺

J-Y Peterschmitt / LSCE

http://www.lsce.ipsl.fr/Phocea/Cours/index.php?uid=jean-yves.peterschmitt

Un exemple!

Commentaire (*note : un script exécutable doit commence par '**#!**'*)

Chargement du ***module*** de gestion du temps (*un module ajoute de nouvelles fonctionnalités au python de base…*)

```
#!/usr/bin/env python

import time

t = time.localtime()

print '(La variable t vaut', t, ')'

print "Bonjour, nous sommes le",

print time.asctime(t)

# Fin
```

Création d'une ***variable*** contenant l'heure et la date
(pas besoin de définir la variable avant de lui donner une valeur!)

Affichage de la valeur *brute* de **t**

Affichage d'un message
…sans aller à la ligne

Affichage de la date sous une forme *compréhensible*

```
(La variable t vaut time.struct_time(tm_year=2013, tm_mon=10, tm_mday=10,
tm_hour=16, tm_min=9, tm_sec=45, tm_wday=3, tm_yday=283, tm_isdst=1) )
Bonjour, nous sommes le Thu Oct 10 16:09:45 2013
```

Pourquoi utiliser python?

- Pratique pour la communauté scientifique (climat... et autre)
  - disponible sous linux et mac (et installable sur PC)
    → possibilité de faire des scripts portables
  - prise en main rapide
    → le mode ligne de commande permet d'expérimenter facilement
  - alternative moderne et puissante aux *scripts shell (sh, csh, ...)*
    - script = fichier texte contenant une suite de commandes à exécuter
    - l'automatisation des tâches répétitives avec un *script* diminue les risques d'erreur!
- Langage généraliste très complet, extensible en fonction des besoins (avec ***`import nom_de_module`***)
- Nombreuses ressources sur le web
- Langage GRATUIT!

Lancer (un script) python (1/2)

- lancement de l'interpréteur python :
  `python [options]` ou **`ipython [options]`**
  - historique des commandes : flèches haut/bas
  - mêmes raccourcis qu'**`emacs`** et **`tcsh/bash`**:
    `^A (CTRL-A) / ^E` → début/fin de ligne
    `^K` → effacer jusqu'à la fin de la ligne
  - **`^D`** → **`quitter`** l'interpréteur!
- exécution d'un script **`script.py`**
  - si on veut rendre le script *exécutable*
    - le script **doit** commencer par
      `#!/usr/bin/env python`
    - **`chmod +x script.py`**
      puis **`./script.py [paramètres]`**
  - si le script n'est pas exécutable
    `python script.py [paramètres]`
    ou **`python -i script.py [paramètres]`**
    → exécution et on reste dans l'interpréteur à la fin du script (très utile pour la mise au point!)

Langage *orienté objet*?? (en bref)

- Python dispose de plein d'*objets* prédéfinis faciles à utiliser!
- Des *méthodes* et des *attributs* sont associés aux objets
  → permettent de faire facilement des opérations complexes sur les objets
  - **`objet.nom_de_methode(paramètres)`**
  - **`objet.nom_attribut`**
- Les objets s'utilisent de manière intuitive!
  - création :
    - **`c = 'python'`** → objet *chaîne de caractères*
    - **`f = open('exemple.txt')`** → objet *fichier*
  - utilisation d'une méthode associée à un objet:
    - **`c.center(20,'=')`** → **`'=======python======='`**
    - **`f.close()`** → fermeture du fichier
  - valeur d'un attribut :
    - **`f.closed`** → **`True`**
  - destruction d'un objet :
    - python s'en charge normalement automatiquement, si l'objet *ne sert plus*…
    - possibilité de détruite explicitement un objet (pour libérer de la mémoire…)
      - **`del(c)`**
      - **`del(ma_matrice_temporaire_enorme)`**
  - type d'un objet :
    - **`type(c)`** → **`<type 'str'>`**
    - **`type(f)`** → **`<type 'file'>`**

Objets et aide en ligne

- Très utile si on n'a pas la documentation sous la main! ☺
- Méthodes et attributs associés à un objet
  - **`dir(f)` → `[…, 'close', 'closed', …]`**
  - note : autres usages de **`dir`**
    - **`dir()`** et **`locals()`** → liste des modules chargés et des variables déjà définies (localement)

      ```
      >>> dir()
      […, 'c', 'f', …]
      ```

    - **`dir(nom_de_module)`** → donne la liste des fonctions et des constantes définies dans un module

      ```
      >>> dir(time)
      […, 'asctime', …, 'tzname']
      ```

- Mode d'emploi d'une méthode
  - **`help(c.center)`**

    ```
    S.center(width[, fillchar]) → string
        Return S centered in a string of length width.
           Padding is done using the specified fill character
           (default is a space)
    ```

Les nombres sont des objets!

- **`>>> type(1), type(1L), type(1.), type(1j)`**
  `(<type 'int'>, <type 'long'>, <type 'float'>, <type 'complex'>)`
- Entiers (arbitrairement grands)
  - **`5`**, **`0xff (255)`**, **`100000L`**
  - **`2**100`** → **`1267650600228229401496703205376L`**
  - **Attention!** **`1/2`** → **`0`** **`1./2`** → **`0.5`**
- Réels
  - **`3.14`** (PI est défini dans **`math.pi`** et **`numpy.pi`**)
  - **`1./3`** → **`0.33333333333333331`**
  - pas de contrôle explicite de la précision…
    - Il peut y avoir des problèmes de chiffres significatifs…

      `10. ** 15 + 1` → **`1000000000000001.0`**
      `1e16 + 1` → **`10000000000000000.0`**
      `(1e16 + 1) – 1e16` → **`0.0`**
    - Utiliser le module **`numpy`** s'il est nécessaire de savoir précisément ce que l'on fait
- Complexes
  - **`1j**2`** → **`(-1+0j)`**

Interlude : exemple ‘interactif’

```
==> which python
/usr/bin/python

==> python
Python 2.6.6 (r266:84292, May 27 2013, 05:35:12)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.

>>> ipsl_labs =  ['LATMOS', 'LISA', 'LMD', 'LOCEAN', 'LPMAA', 'LSCE']

>>> LSCE in ipsl_labs
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'LSCE' is not defined

>>> 'LSCE' in ipsl_labs
True

>>> test_list = ['Dell', 'LMD', 'NASA']

>>> test_list
['Dell', 'LMD', 'NASA']

>>> for labo in test_list:
...     if labo in ipsl_labs:
...         print labo, 'est un labo de l\'IPSL'
...     else:
...         print labo, 'ne fait pas partie de l\'IPSL'
...
Dell ne fait pas partie de l'IPSL
LMD est un labo de l'IPSL
NASA ne fait pas partie de l'IPSL
>>>
```

Version de python

Aspect typique du message (*traceback*) affiché lorsqu'une erreur (ou *exception*, dans le jargon python) se produit

Il n’y a pas de variable s’appelant **'LSCE'**

Pas besoin de **print** en mode interactif

Une boucle et des tests en mode interactif ☺
On itère sur un objet qui comporte (ou peut générer à la volée) plusieurs éléments

Les blocs d’instructions sont définis par un même nombre d’indentations

CTRL-D pour quitter python !

Les chaînes de caractères … sont des objets (etc…)

- Syntaxe standard
  'abc', **"abc"**
  Utilisation classique de '\' :
  - **\'** et **\"** → **'j\'utilise python'**
  - ****, **\n** (retour chariot), **\t** (tabulation)
- Chaînes sur plusieurs lignes

```
s = """j'utilise
des chaines avec des " et des '
sur plusieurs lignes"""
```

- Conversion
  str(3.14) → **'3.14'**
  float('3.14') → **3.1400000000000001**
- Longueur
  len('abc') → 3
- Concaténation, répétition
  - **'a' + 'b'** → **'ab'**
  - **10 * ('a' + 'b')** → **'abababababababababab'**

Les caractères *exotiques*

- Si on utilise des minuscules accentuées sans précaution dans un script…

```
SyntaxError: Non-ASCII character '\xc3' in file test1.py on line
```

- La solution:
  1. Mettre en 1ère ou 2ème ligne du script une indication de codage *unicode* des caractères
  2. Ajouter un préfixe **u** devant les chaînes *exotiques*

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-

print u'utilisation de minuscules accentuées'
print 'Pas de caracteres exotiques ici'
print u'Affichage du caractère degré :  '
```

```
utilisation de minuscules accentuÃ©es
Pas de caracteres exotiques ici
Affichage du caractÃ re degrÃ© : Â
```

```
utilisation de minuscules accentuées
Pas de caracteres exotiques ici
Affichage du caractère degré :
```

- Il peut y avoir des interactions inattendues avec l'éditeur de texte, le shell dans lequel on exécute le script…

- Plus de détails dans le *PEP* approprié

  PEP = Python Enhancement Proposals

  http://www.python.org/dev/peps/pep-0263/

Manipulation des chaînes (1/2)

- **`dir('')`** pour lister toutes les méthodes
- Exemples : **`s = '==[abc==abc]=='`**
  - suppression des *blancs* (ou autre) en début et en fin de chaîne
    - **`s.strip('=')`** → **`'[abc==abc]'`**

      Note : nettoyage d'une chaîne (on ôte les espaces, tabulations et passage a la ligne en début/fin)

      `'\tEssai  \n\r'.strip()` → **`'Essai'`**
    - Nettoyage à droite ou à gauche

      `s.rstrip()`, **`s.lstrip()`**
  - **`s.replace('abc','X')`** → **`'==[X==X]=='`**
  - **`s.upper()`** → **`'==[ABC==ABC]=='`**
  - **`s.count('abc')`** → **`2`**
  - **`s.find('ab')`** → **`3`**

    `s.find('abd')` → **`-1`**

Manipulation des chaînes (2/2)

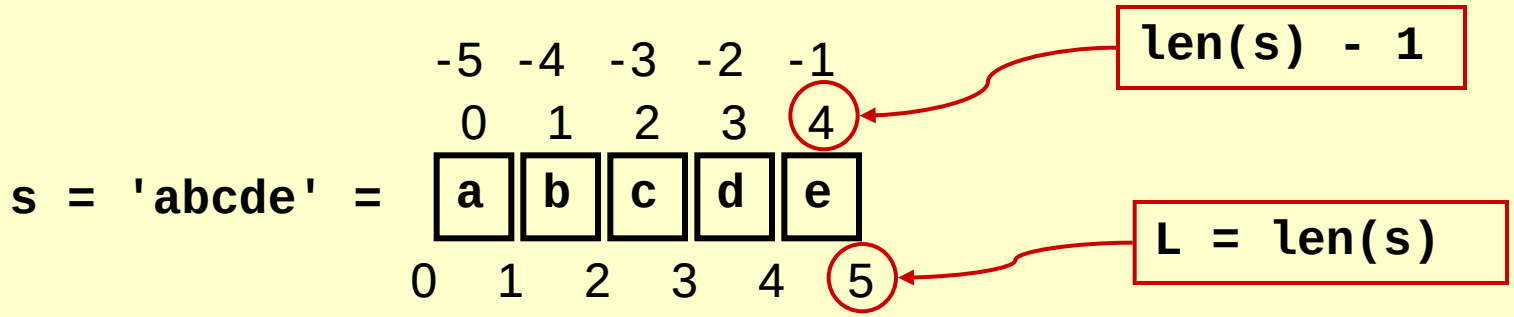
- `'abc;999;def'.split(';')`
  → `['abc', '999', 'def']`

  Note : données lues dans un fichier texte *csv* (comma separated values)...
  - `'-1;-0.5;0;0.5'.split(';')`
    → `['-1', '-0.5', '0', '0.5']`
  - `map(float, '-1;-0.5;0;0.5'.split(';'))`
    → `[-1.0, -0.5, 0.0, 0.5]`
- `' -- '.join(['abc', '999', 'def'])`
  → `'abc -- 999 -- def'`

- Combinaison d'opérations
  - `s.strip('=').count('=')` → `2` `('==[abc==abc]==')`
- Opérations plus complexes avec le module de traitement des *expressions régulières*
  - `import re`
    - voir la doc... ☺
  - exemples à méditer :

```
l = '1 un\t2 deux\t3 trois'
l.split()     → ['1', 'un', '2', 'deux', '3', 'trois']
l.split('\t') → ['1 un', '2 deux', '3 trois']
re.split('[0-9 \t]*', l)      → ['', 'un', 'deux', 'trois']
re.split('[a-z \t]*', l)      → ['1', '2', '3', '']
```

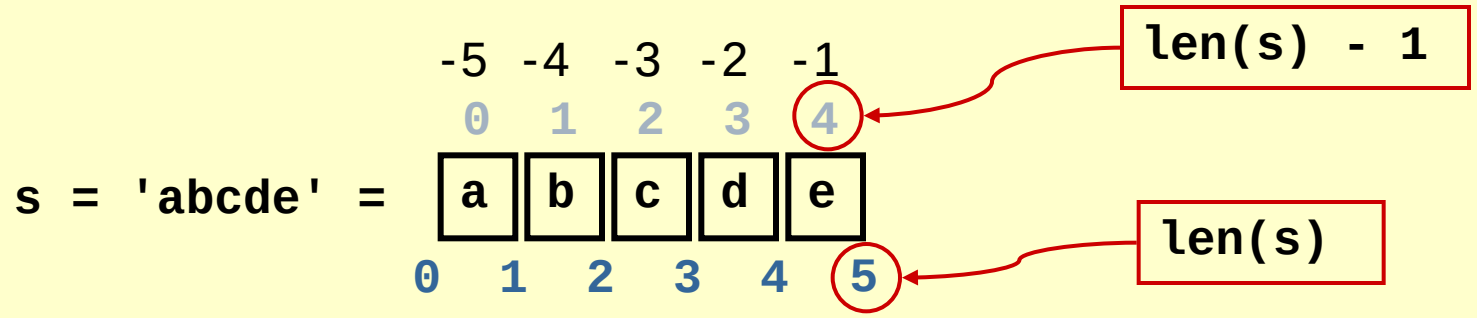
Le contenu d'une chaîne... (1/2)



- Il **faut** absolument comprendre le fonctionnement des indices en python!
  - les indices fonctionnent de la même façon pour tous les objets python qui sont ordonnés (chaînes, listes, tableaux, ...)
  - c'est plus facile à comprendre pour les chaînes de caractères

- Les indices commencent à ZERO!
  - le **n**ème élément d'un objet a donc l'indice ***n-1***!
  - le dernier élément d'un objet a l'indice ***-1***, l'avant dernier a l'indice ***-2***, etc...
  - les indices des éléments d'un objet de longueur ***L*** vont de
    0 à ***L-1***
    ou de ***–L*** à ***-1***

Le contenu d'une chaîne... (2/2)



- s[*indice*] → *indexing*

  s[0] → 'a' s[4] → 'e' s[-1] → 'e'

- **s[*debut*:*fin*:*pas*] → *slicing***

  Attention! Quand on spécifie un indice de fin, l'élément correspondant à cet indice est EXCLU!

  Astuce! On peut compter les *indices entre les cases*

```
s[1:4]              → 'bcd'
s[2:]               → 'cde'        s[:2] → 'ab'
s[0:4:2]            → 'ac'
s[::-1]             → 'edcba'
    Note : s[:]     → 'abcde'
```

(**:** ⇔ tous les élément de s ⇔ **copie** de s)

- Note : il y a des compléments sur l'affichage et le formatage des chaînes dans la suite du cours

Les listes et les *tuples*

- Ce sont des ensembles ordonnés de n'importe quels objets python.

- On accède donc aux éléments des listes/tuples par slicing/indexing (comme les chaînes de caractères)

  - Liste : **lst = [élément$_0$, …, élément$_{N-1}$]**
    - **lst = ['deux', 0, f, 1., s]**

```
>>> print lst
['deux', 0, <open file '…\test_02.py', mode 'r' at 0x00BE5800>, 1.0,
'==[abc==abc]==']
```

    - utile : génération d'une liste de nombres entiers

      range(*[début,]* fin *[,pas]*)

      range(5) → **[0, 1, 2, 3, 4]**

      range(5, 10, 2) → **[5, 7, 9]**

      Attention! Pour les boucles, utiliser plutôt **xrange**

  - *Tuple* : **tpl = (élément$_0$, …, élément$_{N-1}$)**
    - équivalent à une liste, la plupart du temps
    - attention! Un tuple à 1 seul élément se termine par une virgule!

      tpl = (élément$_0$,)

Listes : exemples d'utilisation

- Rappel: indice du 1er élément → **0**
  indice du dernier → **N-1** ou ***-1***
- Indexation :

```
lst = [-1, 'abc', [2, 4, 'def'], 10.]
   lst[ 0]            → -1
   lst[-2][ 1]        → 4
   lst[ 2][-1][1]     → 'e'
```

- Concaténation, répétition :

```
[1]+[2]+[[96,72]] → [1,2,[96,72]]
3 * ['a', 1]   → ['a', 1, 'a', 1, 'a', 1]
```

- Rappel sur le *tuple* à 1 élément

```
4 * (1)        → 4
4 * (1,)       → (1, 1, 1, 1)
```

-

```
valeurs = [0., 10., -1.]
min(valeurs)        → -1.0
len(valeurs)        → 3
10. in valeurs      → True
```

Listes : quelques méthodes utiles

Opération	*Valeur de la liste*
a = []	**[]** (*liste vide*)
a = a + [10]	**[10]**
a += [-1.]	**[10, -1.0]**
a += valeur ⇔ a = a + valeur	
a.extend([0,4])	**[10, -1.0, 0, 4]**
Concaténation de **a** et du paramètre de **extend**	
a.append([0,4])	**[10, -1.0, 0, 4, [0,4]]**
Insertion du paramètre de **append** a la fin de **a**	
a.remove([0,4])	**[10, -1.0, 0, 4]**
Note : l'opération supprime juste la *première occurrence*!	
a.reverse()	**[4, 0, -1.0, 10]**
Attention! L'inversion se fait *in-place* : le contenu de a est mis à jour, et il n'y a pas création d'une nouvelle liste…	
a.sort()	**[-1.0, 0, 4, 10]**
Attention! Opération *in-place*, comme pour **reverse**	
Utiliser **sorted(a)** pour récupérer une copie triée de a	
a.index(*4*) →2	**[-1.0, 0, *4*, 10]**
a.pop(2) →4	**[-1.0, 0, 10]**

python : les tests et les booléens

- Comparaison :

  `x<y`, `<=`, `>`, `>=`, `==`, `!=` ou `<>`

  - **attention** à bien utiliser **`==`** pour tester l'égalité!
  - écriture compacte de plusieurs comparaisons :

    `1<2<2<3` → **`False`** **`1<2<=2<3`** → **`True`**

- Appartenance : x in s, x not in s

  Très utile pour remplacer une suite de tests!

  `'blanc' not in ['rouge', 'vert', 'bleu']` → **`True`**

- **`and`**, **`or`**, **`not`**
- Ce qui est ***`False`***

  `None`, **`0`**, **`0.`**

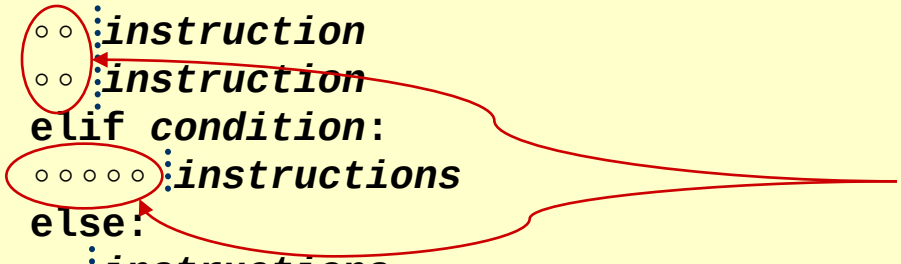
  `''`, **`[]`**, **`()`**, **`{}`** ← chaîne, liste, tuple et dictionnaire **vides**
- Ce qui est ***`True`***

  Tout le reste!

python : structures de contrôle

- Des instructions appartiennent à un même bloc si elles sont indentées de la même façon! Pas besoin de **endif**, **enddo**, **begin-end**, **{** et **},** …
- Tests :

```
if condition:
    instruction
    instruction
elif condition:
    instructions
else:
    instructions
```



Le nombre d'espaces pour l'indentation n'a pas d'importance (≥1), mais il doit être le même pour toutes les lignes d'un même bloc d'instructions!

- Boucles :

```
for élément in séquence:
    instructions
```

Note : sortie de boucle possible avec **break** et **continue**

Attention! Pour faire une boucle sur un grand nombre d'entiers, utiliser ***xrange***, plutot que **range**! (**range** commence par créer une liste avec tous les entiers en mémoire…)

- Il y a aussi :

```
while condition:
    instructions
else:
    instructions
```

On mélange un peu tout...

Traduction des noms abrégés de la liste des composants d'un modèle couplé

```
#!/usr/bin/env python

composants = ['atm', 'OCN', 'Veg', 'xxx']

separateur = '-'
for comp in composants:
    if comp == composants[-1]:
        separateur = ''
    comp = comp.lower()
    if comp == 'atm':
        nom = 'atmosphere'
    elif comp == 'ocn' or comp == 'ocean':
        nom = 'ocean'
    elif comp in ['veg', 'surf']:
        nom = 'vegetation'
    else:
        nom = 'other'

    print nom, separateur,

print
```

```
atmosphere - ocean - vegetation - other
```

Liste d'abréviations à traduire

Pas besoin d'afficher de séparateur après le dernier composant...

Détermination du nom du composant

Affichage du nom suivi du séparateur, sans aller à la ligne. Le séparateur n'est pas affiché (chaîne vide) après le dernier élément...

print à l'extérieur de la boucle, pour aller à la ligne

*Variantes : **1** - 2 – 3 - 4*

Les dictionnaires... sont très utiles!

- Un *dictionnaire* permet d'associer très facilement des *clés* (nombres, chaînes, ...) et des *valeurs*
  - création d'un dictionnaire vide et ajout de valeurs

    d = {}

    $d[clé_0]=val_0$ $d[clé_{N-1}]=Val_{N-1}$
  - création d'un dictionnaire pré-rempli

    d = {$clé_0$:val_0, ..., $clé_{N-1}$:val_{N-1}**}**
  - Attention! Les dictionnaires ne sont pas ordonnés, donc pas de slicing/indexing
- Exemple :

  d = {'vals':[0, 1], 1:'un', 'pi':3.14}
  - informations sur le contenu :

    len(d) → **3**

    d.keys() → **['vals', 1, 'pi']**

    d.has_key('vals') → **True**
  - accès au contenu :

    d['vals'] → **[0, 1]**
  - suppression/ajout :

    del d['vals'] → **d = {1:'un', 'pi':3.14}**

    d['one'] = 1 → **d = {1: 'un', 'pi': 3.14, 'one': 1}**

Pourquoi utiliser un dictionnaire?

- Une liste doit être utilisée lorsque l'on a besoin de disposer d'un ensemble ordonné d'éléments ET que l'on sait facilement déterminer l'indice des éléments auxquels ont doit accéder…
- Sinon… un dictionnaire est plus pratique!

```
couleurs = {'vert':  [[0, 100, 0], 'green'],
            'rouge': [[100, 0, 0], 'red']}
```

  - `couleurs['vert'][0]` → `[0, 100, 0]`
  - `couleurs['rouge'][1]` → `'red'`

- On peut évidemment imbriquer des dictionnaires! ☺

Exemple : on veut stocker et utiliser les composantes RVB et le nom anglais des couleurs

- initialisation :

```
couleurs = {}
couleurs['vert'] = {}
couleurs['vert']['RVB'] = [0, 100, 0]
couleurs['vert']['eng'] = 'green'
couleurs['rouge'] = {'RVB':[100,0,0], 'eng':'red'}
```

- utilisation :
  - `couleurs['vert']['RVB']` → `[0, 100, 0]`
  - `couleurs['rouge']['eng']` → `'red'`
  - `couleurs.has_key('noir')` → `False`

Exemple précédent, avec dico

Traduction des noms abrégés de la liste des composants d'un modèle couplé

```
#!/usr/bin/env python

noms_comp = {'surf': 'vegetation', 'veg':
'vegetation', 'atm': 'atmosphere', 'ocean':
'ocean', 'ocn': 'ocean'}

composants = ['atm', 'OCN', 'Veg', 'xxx']

separateur = '-'
for comp in composants:
    if comp == composants[-1]:
        separateur = ''
    comp = comp.lower()
    if noms_comp.has_key(comp):
        nom = noms_comp[comp]
    else:
        nom = 'other'

    print nom, separateur,

print
```

```
atmosphere - ocean - vegetation - other
```

Définition du dictionnaire de correspondance entre les abréviations et les noms complets

Utilisation d'un dictionnaire, plutôt qu'une série de tests **if**/**elif** !

*Variantes : 1 - **2** – 3 - 4*

Lecture/écriture de fichiers texte (1/2)

- Ouverture d'un fichier :
  `fic = open('nom_fichier', mode)`
  mode : **`'r'`**=lecture, **`'w'`**=écriture, **`'a'`**=ajout
- Fermeture : **`fic.close()`**
  Attention à ne pas oublier de fermer les fichiers! ☺
- Lecture :
  - ligne par ligne :
    `ligne = fic.readline()`
    l**`en(ligne)==0`** si on dépasse la fin du fichier
  - toutes les lignes d'un coup :
    `lignes = fic.readlines()`
  - attention! Les lignes lues contiennent la marque de fin de ligne!
    `'ligne 0\n'`
    `…`
    `'ligne N-1\n'`
    - La marque peut dépendre du système où a été créé le fichier
      - Linux → '**`\n`**'
      - Mac → '**`\r`**'
      - Win → '**`\r\n`**'

    Suppression de la fin de ligne avec : **`ligne = ligne.strip()`**

Lecture/écriture de fichiers texte (2/2)

- Parcourir les lignes d'un fichier :

```
for ligne in lignes:
    print ligne.strip()
```

  - pour parcourir toutes les lignes de façon efficace (rapidement, et sans charger tout le fichier en mémoire…) :

```
for ligne in fic:
    print ligne.strip()
```

- Ecriture :
  - ne pas oublier la marque de fin de ligne!
    - Linux = `'\n'`, Win = `'\r\n'`, Mac = `'\r'`
    - ou bien :

```
import os
fin_ligne = os.linesep
```

  - écriture ligne par ligne

```
fic.write('ma ligne' + fin_ligne)
```

  - écriture de toutes les lignes d'une liste de chaînes de caractères

```
lignes = []
lignes.append('ligne 0' + fin_ligne)
…
lignes.append('ligne N-1' + fin_ligne)
fic.writelines(lignes)
```

Exemple de lecture d'un fichier

Affichage d'une colonne d'un fichier (texte) de données

```
#!/usr/bin/env python

nomfic = 'fic_exemple.txt'
nb_entete = 2
separateur = ';'
num_colonne = 2

fic = open(nomfic, 'r')
for saute in range(nb_entete):
   fic.readline()

for ligne in fic:
    ligne = ligne.strip()
    colonnes = ligne.split(separateur)
    print colonnes[num_colonne]

fic.close()
```

Lecture de l'entête

(pour sauter les lignes d'entête)

Lecture des lignes de données, et affichage de la 3ème (indice = 2) colonne

fic_exemple.txt

```
# Exemple de fichier avec des colonnes
# choco;prix;quantite
Mars;0.5;10
M&M;0.5;0
Grany;0.5;15
Tagada;0.5;0
```

```
10
0
15
0
```

Linux : ***awk -F';' 'NR>2 {print $3}' fic_exemple.txt***

Attention! Erreurs à éviter (1/...)

- `==` et pas `=` pour tester l'égalité!
- Attention à l'indentation!

  Exemple : la position du **`print`**, juste à l'extérieur de la boucle sur **`j`**, ou a l'extérieur des deux boucles (sur **`i`** et **`j`**), donne des résultats assez différents!

```
for i in range(5):
    for j in range(2):
        print i*2 + j,
        # fin de la boucle sur j
    # print
    # fin de la boucle sur i
# print
```

```
0 1
2 3
4 5
6 7
8 9
```

```
0 1 2 3 4 5 6 7 8 9
```

- La division de nombres entiers renvoie... un entier : **`1/2`** → **`0`**

Attention! Erreurs à éviter (2/...)

- On récupère normalement une référence à un objet, alors que l'on croit en récupérer une copie...

```
a = [0, 1]
b = a                    →      b = [0, 1]
b.append(2)              →      a = [0, 1, 2]
```

Ce que l'on voulait vraiment :

```
a = [0, 1]
b = a[:]                 →      b = [0, 1]
b.append(2)              →      b = [0, 1, 2]
                                a = [0, 1]
```

Note : recopie d'un objet avec **`copy.deepcopy(objet)`**

- Ne pas oublier '**`()`**' lors de l'appel à une fonction/méthode sans paramètres!

```
>>> s = '    abc '
>>> s1 = s.strip()
>>> s2 = s.strip
>>> s1
'abc'
>>> s2
<built-in method strip of str object at 0x00BE2980>
```

- Il faut utiliser correctement les opérations *in-place* (**`sort, reverse`**)

```
>>> lst = [3, 1, 4, 2]
>>> lst2 = lst.sort()
    lst2 → None          lst → [1, 2, 3, 4]
```