# CDAT et le langage python

I. Introduction à python

J-Y Peterschmitt / LSCE

**LSCE :** http://boulimix/~jypeter/CDAT/Doc/Tutorials/JYP/

# Un exemple!

(py_jyp_ex_01.py)

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 t
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*

```
(2007, 1, 15, 16, 49, 37, 0, 15, 0)
(La variable t vaut (2007, 1, 15, 16, 49, 37, 0, 15, 0) )
Bonjour, nous sommes le Mon Jan 15 16:49:37 2007
```

# Objectifs du cours

- Découvrir le langage python et…
  - … savoir ce que l'on peut facilement faire avec
    - → remplacer les anciens scripts sh/ksh !
      - manipulation des chaînes de caractères
      - gestion des fichiers
      - manipulation des nombres
  - … vous donner envie de l'utiliser tout de suite grâce à des exemples concrets!
  - … vous donner envie d'aller plus loin!
- Pour en savoir encore plus…
  - … n'oubliez pas de venir aux prochains cours ☺

# Ce cours n'est pas…

- Une présentation exhaustive du langage python
  - nombreux sites consacrés à python sur le web: cours, exemples, librairies de scripts
    - Voir la page liens utiles…
- Un cours (purement) sur CDAT
  - merci de revenir aux prochains cours ☺

# Qu'est-ce que python?

- Langage *interprété* de haut niveau, *orienté objet*
  - Patience pour les explications…
- Inventé fin 1989 par Guido van Rossum
- Python ← Monty Python!
- Et CDAT??
  - **C**limate **D**ata **A**nalysis **T**ools
  - CDAT rajoute des fonctionnalités au python standard
  - développé au PCMDI (USA) pour l'analyse des données AMIP, IPCC, …

    → bien adapté aux besoins de la communauté *climat*

# Pourquoi utiliser python?

- Pratique pour la communauté scientifique (climat… et autre)
  - disponible en standard sous linux

    …également sur les Mac et les PC!
  - prise en main rapide

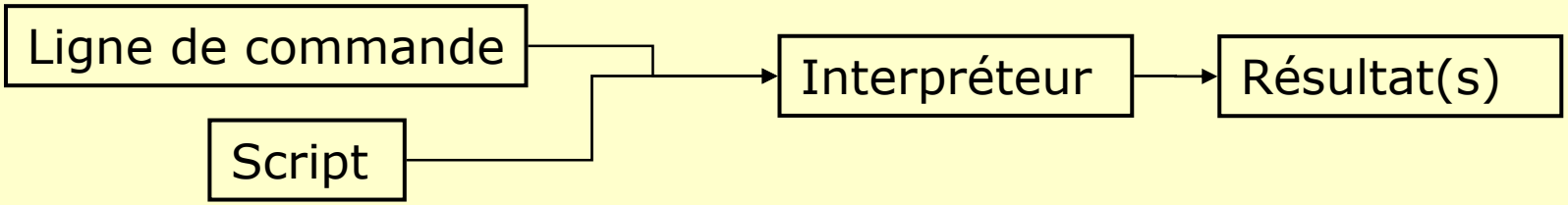
    → le mode ligne de commande permet d'expérimenter facilement
  - alternative moderne et puissante aux *scripts shell (sh, csh, …)*
    - script = fichier texte contenant une suite de commandes/programmes à exécuter
    - l'automatisation des tâches répétitives avec un *script* diminue les risques d'erreur!
- Langage extensible en fonction des besoins : ajout dynamique (sans relancer python) de nouvelles fonctionnalités (avec ***`import nom_de_module`***)
- Nombreuses ressources sur le web
- Langage GRATUIT!

# Langage *interprété* vs *compilé (1/2)*

- Langages interprétés
  - exemples : sh/ksh/csh, idl, ferret, perl, lisp… et python!
  - instructions en ligne de commande → facilité de mise au point

```
>>> import time
>>> time.tzname
('Paris, Madrid', 'Paris, Madrid')
>>> time.asctime()
'Tue Jan 16 15:35:12 2007'
```

  - plusieurs lignes d'instructions dans un fichier → '*script*'
  - inconvénients (éventuels) :
    - nécessité d'avoir l'interpréteur installé sur l'ordinateur…
      … mais un même script peut fonctionner sur plusieurs plateformes → portabilité
    - parfois des problèmes de performance…
      … mais on peut souvent contourner ces problèmes

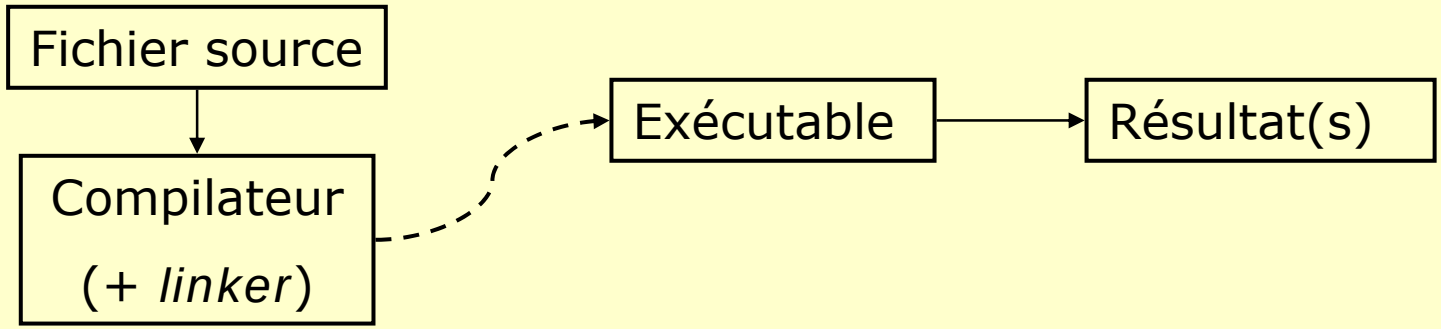


# Langage *interprété* vs *compilé (2/2)*

- Langages compilés
  - Fortran, C…
  - génération d'un programme exécutable
    - Pas de problème de performance
    - Pas vraiment portable

      le même exécutable ne peut pas fonctionner sous Windows/Mac/Linux…
    - Mise au point plus difficile

      il faut recompiler pour tester des nouvelles fonctions



# A quel python ai-je accès?

- Win : démarrer → programmes → python
  - Note : distribution recommandée sous Windows

    http://code.enthought.com/enthon/
- Linux : taper '**`which python`**'
  - **`/usr/bin/python`** → python standard
  - Utilisation du CDAT commun du LSCE :

    '**`source ~jypeter/CDAT/cdat.login`**'

    → '**`which python`**' → **`…/cdat/…/python`**
- Mac OS X :

  **`which python`** → **`/usr/bin/python`**

# Lancer (un script) python (1/2)

- Linux et Mac (OS X):
  - lancement de l'interpréteur :

    **> python**
    - historique des commandes : flèches haut/bas
    - mêmes raccourcis qu'**emacs** :

      **^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 optionnels]**
    - 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!)

# Lancer (un script) python (2/2)

- Win :
  - lancement de l'interpréteur (**IDLE**):

    **Démarrer → Tous les programmes → python 2.x → IDLE**
    - historique des commandes : **Alt-P** / **Alt-N**
    - mêmes raccourcis qu'**emacs** :

      **^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 dans l'interpréteur **IDLE**
    - chargement du script : **File → Open**…
    - lancement du script : **F5** ou **Run → Run module**
  - autres méthodes?
    - ???
    - passage de paramètres à un script?

# Langage *orienté objet*?? (en bref)

- Python est plein d'objets!
  objet = *quelque chose que l'on crée et que l'on utilise/modifie facilement dans python*
- 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_01.py')`** → objet *fichier*
  - utilisation d'une méthode :
    - **`c.center(20,'=')`** → **`'=======python======='`**
    - **`f.close()`** → fermeture du fichier
  - valeur d'un attribut :
    - **`f.closed`** → **`True`**
  - destruction d'un objet :
    - pas très utilisé, car python s'en charge plus ou moins 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 (pas très utilisé…):
    - **`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()`** → donne la liste de tous les modules chargés et les 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

- Entiers (arbitrairement grands)
  - `5, 0xff (255), 100000L`
  - `2**100` → `1267650600228229401496703205376L`
  - Attention! `1/2` → `0` `1./2` → `0.5`
- Réels
  - `3.14`
  - `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`

      `1e17 + 1` → `1e+017`
    - Utiliser le module **`Numeric`** s'il est nécessaire de savoir précisément ce que l'on fait
- Complexes
  - `1j**2` → `(-1+0j)`

# Python… comme une calculatrice!

- '**python**' pour démarrer le mode interactif
  - **^D** (**ctrl-D**) pour quitter l'interpréteur!
- Opérations de base :

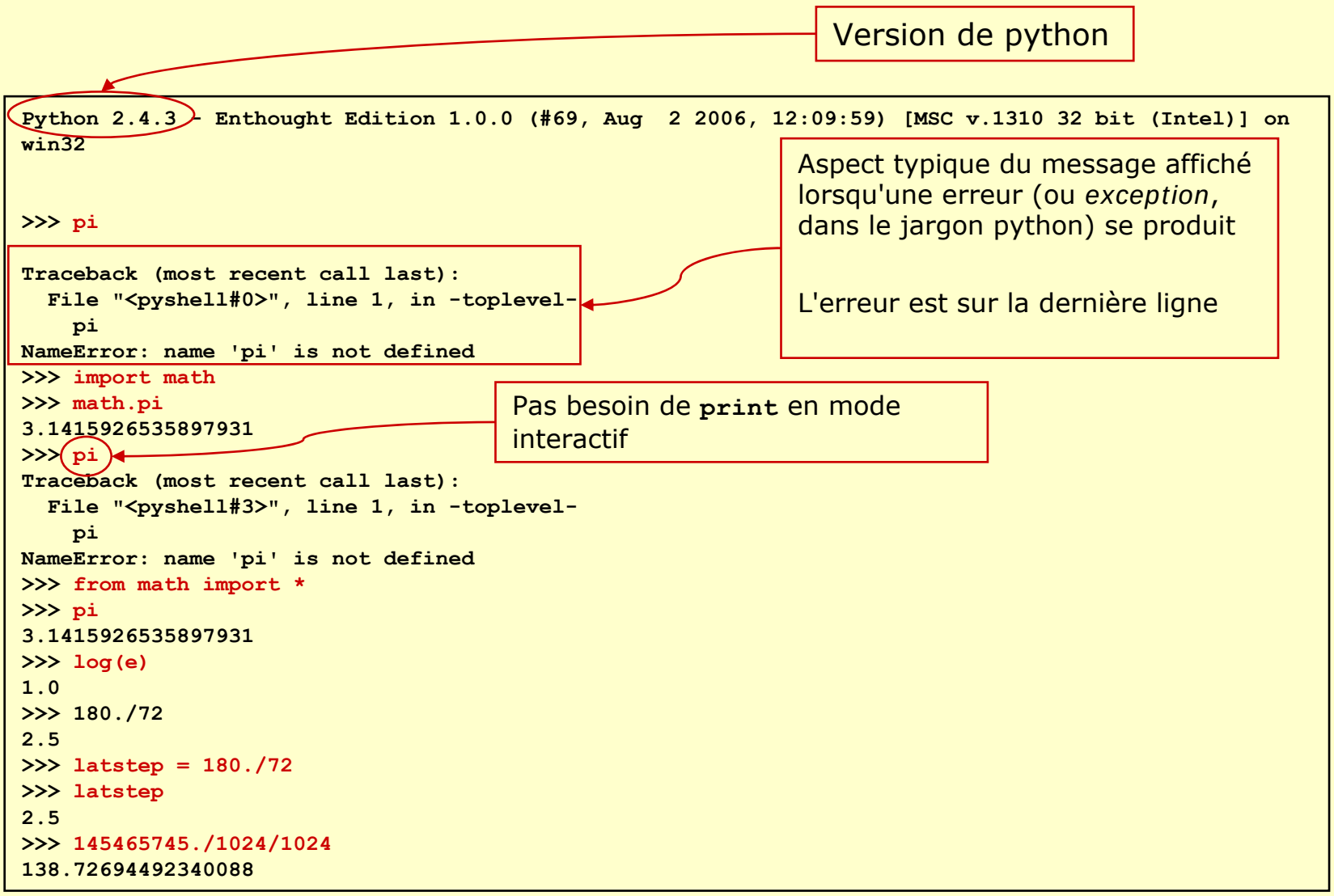
  +, -, *, ** (puissance), /, % (modulo)

  abs(), int(), float()  …et str()
- Pour faire plus de maths… il faut le module **math**! ☺
  - **import math** → **math.cos(math.pi)**

    **from math import *** → **cos(pi)**
  - **dir(math)** ← dir() donne le contenu d'un module

```
['acos', 'asin', 'atan', 'atan2', 'ceil',
   'cos', 'cosh', 'degrees', 'e', 'exp',
   'fabs', 'floor', 'fmod', 'frexp',
   'hypot', 'ldexp', 'log', 'log10',
   'modf', 'pi', 'pow', 'radians', 'sin',
   'sinh', 'sqrt', 'tan', 'tanh']
```

# La calculatrice en action!



# 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!
- Note : test du type d'un objet avec **`isinstance`**
  (juste pour savoir que c'est possible de le faire ☺)
  **`isinstance(10, int)`** → **`True`**

# Les chaînes de caractères

- 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'**

# Traitement des chaînes (1/2)

- dir('') pour avoir 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'`**
    - **`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`**

# Traitement des chaînes (2/2)

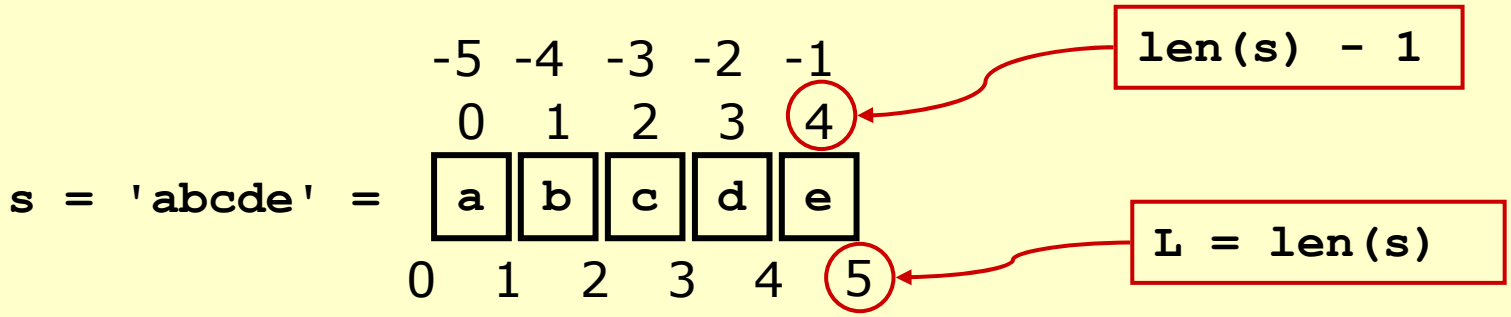
- `'abc;999;def'.split(';')`
  → `['abc', '999', 'def']`

  Note : données lues dans un fichier texte…
  - `'-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`
- 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', '']
```

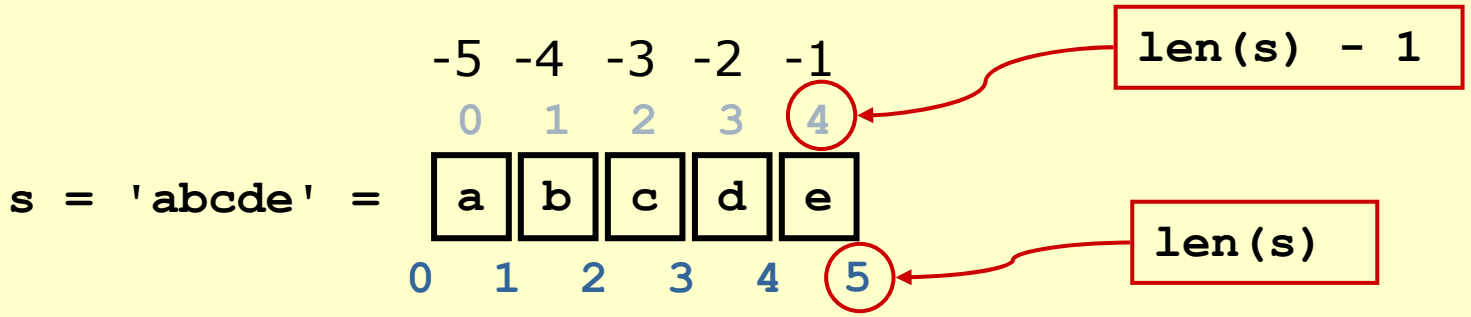
# Le contenu d'une chaîne… (1/2)




- Il **faut** absolument comprendre le fonctionnement des indices!
  - les indices fonctionnent de la même façon pour tous les objets python qui sont ordonnés (chaînes, listes, tableaux, …)
  - c'est plus facile de comprendre avec 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 à 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***, et ***–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 2ème partie du cours

# Les listes et les *tuples*

Ce sont des ensembles ordonnés (on peut faire du *slicing/indexing*) de n'importe quels objets python

- 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$_0$,)***

- Accès aux éléments par slicing/indexing

  → cf. chaînes de caractères

  **lst[1], tpl[3:], …**

# 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

| *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` | |
| `a.index(4)` →2 | `[-1.0, 0, 4, 10]` |
| `a.pop(2)` →4 | `[-1.0, 0, 10]` |

# 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…

(py_jyp_ex_02.py)

*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

- 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 = {`$clé_0$:$val_0$, …, $clé_{N-1}$:$val_{N-1}$`}`
- 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

(py_jyp_ex_03.py)

*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 : `'rU'`=lecture, `'w'`=écriture, `'a'`=ajout
- Fermeture : **`fic.close()`**
- Lecture :
  - ligne par lignes :

    **`ligne = fic.readline()`**

    **`len(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!
    - '**`\n`**', si le fichier a été ouvert en mode '**`rU`**' (*Universal*)

      **`'ligne 0\n'`**

      …

      **`'ligne N-1\n'`**
    - marque dépendant du système où a été créé le fichier, si ouverture du fichier en mode '**`r`**'
      - 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
```

  - ecriture ligne par ligne

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

  - ecriture de plusieurs lignes d'un coup

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

# Exemple de lecture d'un fichier

(py_jyp_ex_04.py)

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

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

nomfic = 'py_jyp_ex_04_data.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

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

**py_jyp_ex_04_data.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}' py_jyp_ex_04_data.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]
```

# Liens utiles

- Site web officiel de python
  http://www.python.org/
- Python eggs : plein de liens
  http://www.python-eggs.org/
- Vaults of Parnassus : d'autres liens
  http://www.vex.net/parnassus/

# Documentations recommandées

- **Python tutorial** : *Le Mode d'Emploi de Base*
  → très bien écrit et facile à lire!

  http://docs.python.org/tut/tut.html
  **`tut.pdf`** dans http://www.python.org/ftp/python/doc/2.5/pdf-a4-2.5.zip
  *Version française* (pas à jour…): http://olivierberger.org/python/tut.pdf

- **Python 2.5 Quick Reference** : *Le Résumé*
  → tout sous la main en un minimum (55…) de pages!

  http://rgruet.free.fr/PQR25/PQR2.5.html
  http://rgruet.free.fr/PQR25/PQR2.5_modern_a4.pdf

- **Python Library Reference** : *La Référence*
  → toutes les fonctions et les modules disponibles en standard
  (très utile, avec la fonction 'rechercher' d'Acrobat ☺ )

  http://docs.python.org/lib/lib.html
  **`lib.pdf`** dans http://www.python.org/ftp/python/doc/2.5/pdf-a4-2.5.zip

- *Les docs (plus ou moins à jour) sont aussi disponibles, au LSCE, dans*
  *http://boulimix/~jypeter/CDAT/Doc/*
  *ou /home/users/jypeter/CDAT/Doc*

# Des questions?

- …

# CDAT et le langage python

II. Plus loin avec python…

J-Y Peterschmitt / LSCE

# Affichage avec **print**

- Les arguments de **print**, séparés par une virgule, sont automatiquement convertis en chaîne de caractère (si nécessaire), et affichés séparés par un espace

```
c, t = 'XXX', (0, 'a')
print '->', c, 0, t, 1, '<-'
```

→ `->◦XXX◦0◦(0, 'a')◦1◦<-`

- Pas de retour à la ligne s'il y a une virgule (',') à la fin du **print**

```
print 'un',
print 'deux'
```

→ `un◦deux`

- Si on ne veut pas d'espaces entre les arguments :
  - transformer les arguments en chaînes de caractères et les concaténer

```
print '->' + c + str(0) + str(t) + str(1) + '<-'
```

→ `->XXX0(0, 'a')1<-`

  - utiliser l'*affichage formaté*

# Chaînes formatées (1/2)

- L'utilisation d'une chaîne formatée permet d'avoir un contrôle précis du contenu de la chaîne (et de ce qui est affiché, si on utilise un **`print`**)

- **`c = 'format' % (paramètres)`**

  Ex: **`'cours no %03i/%03i' % (2, 50)`** → **`'cours no 002/050'`**

  Les éléments ***`%fmt`*** de la chaîne **`'format'`** sont remplacés par le contenu du paramètre correspondant du *tuple* **`(paramètres)`**.

  | | |
  |---|---|
  | **`%i`** | affichage d'un nombre entier |
  | **`%Ni`** | entier sur une largeur de N caractères |
  | **`%-Ni`** | entier sur une largeur de N caractères, justifié à gauche |
  | **`%0Ni`** | entier sur N car., précédé par des 0 |
  | | → très utile pour générer des noms de fichiers! |
  | | **`'an_%03i'`** % val → **`'fic_000'`** … **`'fic_049'`** … |
  | ***`%*i`*** | entier sur un nombre de caractères spécifié en argument |
  | ***`%%`*** | affichage du caractère %... |

  - **`d = 10`**
  - **`'[%i]' % (d,)`** → **`'[10]'`**
    - ***`'[' + str(d) + ']'`***
  - **`'[%5i]' % (d,)`** → **`'[◦◦◦10]'`**
    - ***`'[' + str(d).rjust(5) + ']'`***
  - **`'[%-5i]' % (d,)`** → **`'[10◦◦◦]'`**
  - **`'[%05i]' % (d,)`** → **`'[00010]'`**
  - **`'[%0*i]' % (5, d)`** → **`'[00010]'`**

# Chaînes formatées (2/2)

- Autres types de formats :
  - **`%f (et %e, %g)`** : nombres réels
    - spécification du nombre **`P`** de chiffres après la virgule avec : **`%N.Pf`**
  - **`%s`** : chaînes et autres types de données (automatiquement convertis avec **`str(…)`**)

- Exemple : **`lst = [-0.5, 5., 0.]`**

```
'max(%s) = %.2f (%i elements)' %
          (lst, max(lst), len(lst))
```

→ **`'max([-0.5, 5.0, 0]) = 5.00 (3 elements)'`**

- Les éléments à formater peuvent être fournis dans un dictionnaire!

→ Pas besoin de se soucier de l'ordre des paramètres, et un paramètre peut servir plusieurs fois

```
'The RGB components of <%(eng)s> are: %(RVB)s' %couleurs['vert']
```

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

→ **`'The RGB components of <green> are: [0, 100, 0]'`**

# Les fonctions (bref aperçu)

- **`def nom_fonc(`*`paramètres`*`,`**
  **`                `*`paramètres_optionnels`*`):`**
  - **`"""ma documentation"""`**
  - **`instructions`**
  - **`help(nom_fonc)`** → **`'ma documentation'`**

    Si la première ligne d'une fonction est une chaîne de caractères, cette chaîne servira de documentation à la fonction → *doc string*

- Les types des paramètres d'une fonction ne sont pas définis/contrôlés (prudence…)
- Une fonction peut avoir des paramètres optionnels, initialisés avec une valeur par défaut (**$arg_0$=$val_0$, …, $arg_{N-1}$=$val_{N-1}$**)
- Le(s) résultat(s) d'une procédure sont renvoyés avec l'instruction **`return`** (sous la forme d'un *tuple*, si la fonction produit plusieurs résultats)

  **`return resultat`**

  **`return`** **$resultat_0$, …, $resultat_{N-1}$**

- Lire le tutorial de python (**`tut.pdf`**) pour aller plus loin… ☺

# Exemple de fonction

```
def c1c2(c1='X', c2='Y'): # 2 paramètres, avec des valeurs par défaut
        "Concatene deux chaines de caracteres"
        c = str(c1) + str(c2) # Les paramètres sont convertis en chaines…
        return c
```

- `help(c1c2)`

```
    Help on function c1c2 …:
    c1c2(c1='X', c2='Y')
        Concatene deux chaines de caracteres
```

- Différentes façons de passer des paramètres…
  - `c1c2()`

```
    'XY'
```

  - `c1c2(1)`

```
    '1Y'
```

  - `c1c2(c1=1)`

```
    '1Y'
```

  - `c1c2(1, 'a')`

```
    '1a'
```

  - `c1c2(c2='a')`

```
    'Xa'
```

# Les modules

- Permettent de regrouper, réutiliser et partager des fonctions
- Un module peut être un simple script, ou un ensemble de scripts, éventuellement *imbriqués* (ex : **os.path.*xx***)
- Chemin de recherche des modules
  - modification du chemin, pour tous les programme python, en tcsh (dans le .cshrc) :
    **setenv PYTHONPATH ${PYTHONPATH}:/chemin_modules1:/chemin_modules2**
  - ajout d'un repertoire au chemin de recherche par défaut, directement dans un script python
    **sys.path.append('/chemin_modules1')**
    ou **sys.path.extend(['/chemin_modules1', '/chemin_modules2'])**
- Chargement d'un module (nombreuses variantes)
  - **import os**
  - **from os import path**
  - **from os import path as p**
  - **from os.path import exists, isdir**
  - **from os.path import ***
- Info : un script est *compilé* (*byte-compiled*) lors de son chargement, pour accélérer les chargement ultérieurs
  **mon_module.py + import** → création de **mon_module.pyc** au même endroit
- Prise en compte de modifications dans un module en cours d'utilisation (sans relancer le script qui utilise le module) :
  - **import mon_module** → rien ne se passe… ☹
  - **reload(mon_module)** → OK!

- Plus de détails dans **tut.pdf**! ☺
- Liste et mode d'emploi des modules disponibles de base dans **lib.pdf**…

# Les erreurs et leur gestion

- Une erreur est appelée une '*exception*'
  Lorsqu'une erreur est générée (*raised*), elle peut être :
  - prise en compte (*caught/handled*) : le script peut éventuellement continuer
  - ignorée : fin de l'exécution du script, et affichage de la *pile* des erreurs (traceback) → message d'erreur sur la dernière ligne
- Exemple : division par zéro sans gestion d'erreur

```
def abdiv(a, b):
        return float(a)/float(b)

abdiv(1, 0) →     Traceback (most recent call last):
                    …, line 2, in abdiv
                      return float(a)/float(b)
                  ZeroDivisionError: float division
```

- Exemple avec gestion explicite d'erreurs :

```
def abdiv(a, b):
        try:
                c = float(a)/float(b)
        except ZeroDivisionError:
                c = 'Division par zero'
        except StandardError:
                c = 'Erreur inconnue'
        return c
```

Tous les (autres) types d'erreurs

  - `abdiv('1', 2)` → `0.5`
  - `abdiv(1, 0)` → `'Division par zero'`
  - `abdiv('a', 0)` → `'Erreur inconnue'`

# Manipulation de chemins d'accès

**Rappel : on veut pouvoir faire au moins les mêmes opérations qu'avec des scripts *shell*! ☺**

- Les fonctions standard de manipulation des chemins permettent d'utiliser les mêmes programmes sur différents systèmes!
  - Linux : `/chemin/fichier.ext`
  - Win : `D:\chemin\fichier.ext`

    Attention! Dans une chaîne, un `\` doit s'écrire `\\` pour le distinguer de celui qui précède les caractères spéciaux (`\t`, `\n`, ...)

    `D:\chemin\fichier.ext` → `p = 'D:\\chemin\\fichier.ext'`
  - Mac : ?
- `from os import path`

  Cela permet de n'avoir à taper que `path.fonction(…)` au lieu de `os.path.fonction(…)` dans ce qui suit!
- Extraction chemin/fichier

```
p = 'E:\\Users\\jypeter\\exemple_01.py'
path.dirname(p)     → 'E:\\Users\\jypeter'
path.basename(p)    → 'exemple_01.py'
```

- Récuperation de l'extension

```
path.splitext(p)
→ ('E:\\Users\\jypeter\\exemple_01', '.py')
```

- Création d'un chemin d'accès

```
path.join('Users','jypeter') → 'Users\\jypeter'
```

# Opérations de base sur les fichiers

- Détermination du répertoire courant

  **`os.getcwd()`**
- Changement de répertoire/dossier

  **`os.chdir(rep)`**
- Création d'un répertoire/dossier (et des répertoires intermédiaires!)

  **`os.makedirs(rep, 0755)`**
- Effacement d'un fichier/répertoire
  - **`os.remove(fic)`**
  - **`os.rmdir(rep)`**

    **`os.removedirs(rep)`** (Efface les répertoires intermédiaires…)
- Renommage/Déplacement

  **`shutil.move(fic_rep, destination)`**
- Copie

  **`shutil.copy(fic, destination)`**

# Informations sur des fichiers

- Contenu d'un répertoire/dossier

  **`os.listdir(rep)`**

  **`glob.glob(rep_pattern)`**
  - **`glob.glob('*.py')`**
- Taille d'un fichier

  **`path.getsize(fic)`**
- Date de modification

  **`time.asctime(time.localtime(path.getmtime(fic_rep)))`**
- Changement des droits d'accès

  **`os.chmod(fic_rep, droits)`**
- Existence et type des fichiers/répertoires
  - **`path.exists(fic_rep)`**
  - **`path.isfile(fic)`**, **`path.isdir(rep)`**, **`path.islink(fic_rep)`**
- Vérification du droit d'écrire dans un fichier/répertoire

  **`os.access(fic_rep, os.W_OK)`**

# Exemple : renommage de fichiers (py_jyp_ex_05.py)

*Changement des fichiers ***.ps** d'un répertoire en ***.eps***

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

from os import path
import glob, re, shutil

nomrep = 'py_jyp_ex_05_dir'
scanfic = '*.ps'
nouv_ext = 'eps'

ficlist = glob.glob(path.join(nomrep, scanfic))

nb_ren = 0
for f in ficlist:
    f_rep, f_nom = path.split(f)
    f_nom_noext = path.splitext(f_nom)[0]
    f_nouveau_nom = '%s.%s' % (f_nom_noext, nouv_ext)
    nouveau_f = path.join(f_rep, f_nouveau_nom)
    try:
        shutil.move(f, nouveau_f)
        print '[%20s] --> [%20s]' % (f_nom, f_nouveau_nom)
        nb_ren += 1
    except:
        print '** Impossible de renommer %s en %s' % (f_nom, f_nouveau_nom)

if len(ficlist) == 0:
    print 'Pas de fichiers de type \'%s\' dans %s' % (scanfic, nomrep)
else:
    print '%i/%i fichiers ont ete renommes' % (nb_ren, len(ficlist))

# Fin
```

Liste des fichiers à traiter

Détermination du nouveau nom du fichier :
**chemin/nom.extension → chemin/nom.eps**

Renommage, avec gestion d'erreur

```
[         fichier1.ps] --> [        fichier1.eps]
[         fichier2.ps] --> [        fichier2.eps]
[         fichier5.ps] --> [        fichier5.eps]
3/3 fichiers ont ete renommes
```

```
Pas de fichiers de type '*.ps' dans py_jyp_ex_05_dir
```

# Ex : autre renommage de fichiers

(py_jyp_ex_06.py)

*Changement de l'index numérique dans le nom des fichiers d'un répertoire*

```
...
nomrep = 'py_jyp_ex_06_dir'
scanfic = '*.dat'
l_index = 5
ficlist = glob.glob(path.join(nomrep, scanfic))

nb_ren = 0
for f in ficlist:
    f_rep, f_nom = path.split(f)
    re_res = re.search('[0-9]+', f_nom)
    if not re_res:
        print '** %s ne contient pas d\'entier!' % (f_nom,)
        continue
    f_index = f_nom[re_res.start():re_res.end()]
    if len(f_index) >= l_index:
        print '** Longueur de l\'index de %20s >= longueur desiree (%2i)' % \
              (f_nom, l_index)
        continue
    val_index = int(f_index)
    f_nouveau_nom = f_nom.replace(f_index, '_%0*i' % (l_index, val_index,), 1)
    nouveau_f = path.join(f_rep, f_nouveau_nom)
    try:
        shutil.move(f, nouveau_f)
        print '[%20s] --> [%20s]' % (f_nom, f_nouveau_nom)
        nb_ren += 1
    except:
        print '** Impossible de renommer %s en %s' % (f_nom, f_nouveau_nom)

if len(ficlist) == 0:
    print 'Pas de fichiers de type \'%s\' dans %s' % (scanfic, nomrep)
else:
    print '%i/%i fichiers ont ete renommes' % (nb_ren, len(ficlist))
```

Utilisation d'une *expression régulière* pour chercher une suite de 1 ou plusieurs chiffres

Récupération de la suite de chiffres (le numéro **f_index** du fichier)

Remplacement de l'index numérique **f_index** par un index sur **l_index** chiffres (on ne remplace que la 1ère occurrence de **f_index)**

```
[           data1.dat] --> [      data_00001.dat]
[           data5.dat] --> [      data_00005.dat]
[          data11.dat] --> [      data_00011.dat]
[          data12.dat] --> [      data_00012.dat]
4/4 fichiers ont ete renommes
```

# Récupération des arguments d'un script

- Les arguments d'un script sont contenus dans la liste **`sys.argv`**

  **`python -i prog.py fic1.txt fic2.txt`**

  - **`sys.argv = ['prog.py', 'fic1.txt', 'fic2.txt']`**
  - Nombre d'arguments : **`len(sys.argv) = 3`**

    (le nom du script fait partie des arguments!)
  - Nom du script : **`sys.argv[0]`**
  - Paramètres : **`sys.argv[1:]`**

    Nom du 2ème fichier : **`fic2 = sys.argv[2]`**
- Gestion plus souple : module **`getopt`**

  **`getopt`** permet de gérer des arguments précédés par des noms d'options. Les arguments peuvent donc être présents ou non, et leur ordre n'a pas d'importance.

  Les 2 exemples suivants sont équivalents avec **`getopt`**!

  - **`mon_script.py -v --option1 val1 fic1 -d 1 fic2`**
  - **`mon_script.py --option1=val1 -d 1 fic1 fic2 -v`**

# Exemple d'utilisation de `getopt` (1/2)

```
~jypeter/CDAT/Progs/Devel/cgm2eps --landscape -d 210 plot1.cgm plot2.cgm
```

```
import os, sys, getopt
```

Paramètres par défaut

```
density = 72
pageorient = 'l'
```

Fonction qui sera utilisée pour afficher le mode d'emploi, si l'utilisateur ne fournit pas les bons paramètres…

```
def usage():
```

Chaîne sur plusieurs lignes, en utilisant les triples quotes. Les `%X` seront remplacés par leur valeur effective grâce à un affichage formaté

```
    man = """
Usage: %s [options] file1.cgm [file2.cgm ... fileN.cgm]
-d DDD, --density DDD    : density when converting from postscript
                           (default = %i)
-l, --landscape          : landscape mode (default)
-p, --portrait           : portrait mode
…
"""
```

```
    print man % (os.path.basename(sys.argv[0]), density)
```

Nom effectif du script (sans le chemin d'accès) → ***cgm2eps***

# Exemple d'utilisation de `getopt` (2/2)

*--landscape -d 210 plot1.cgm plot2.cgm*

```
try:
    myopts, myargs = getopt.getopt(sys.argv[1:], 'd:lpv',
                                   ['density=','landscape', …])
except getopt.GetoptError, cmdle:
    print 'Command line error :', cmdle
    usage()
    sys.exit(1)

for myopt, myval in myopts:
    if myopt in ['-d', '--density']:
        density = int(myval)
    elif myopt in ['-l', '--landscape']:
        pageorient = 'l'
    elif …

if len(myargs) == 0:
    print '\nNo input file was supplied on the command line'
    usage()
    sys.exit(1)
else:
    inputfiles = myargs
```

Options que l'on va chercher sur la ligne de commande

Affichage du mode d'emploi en cas d'erreur

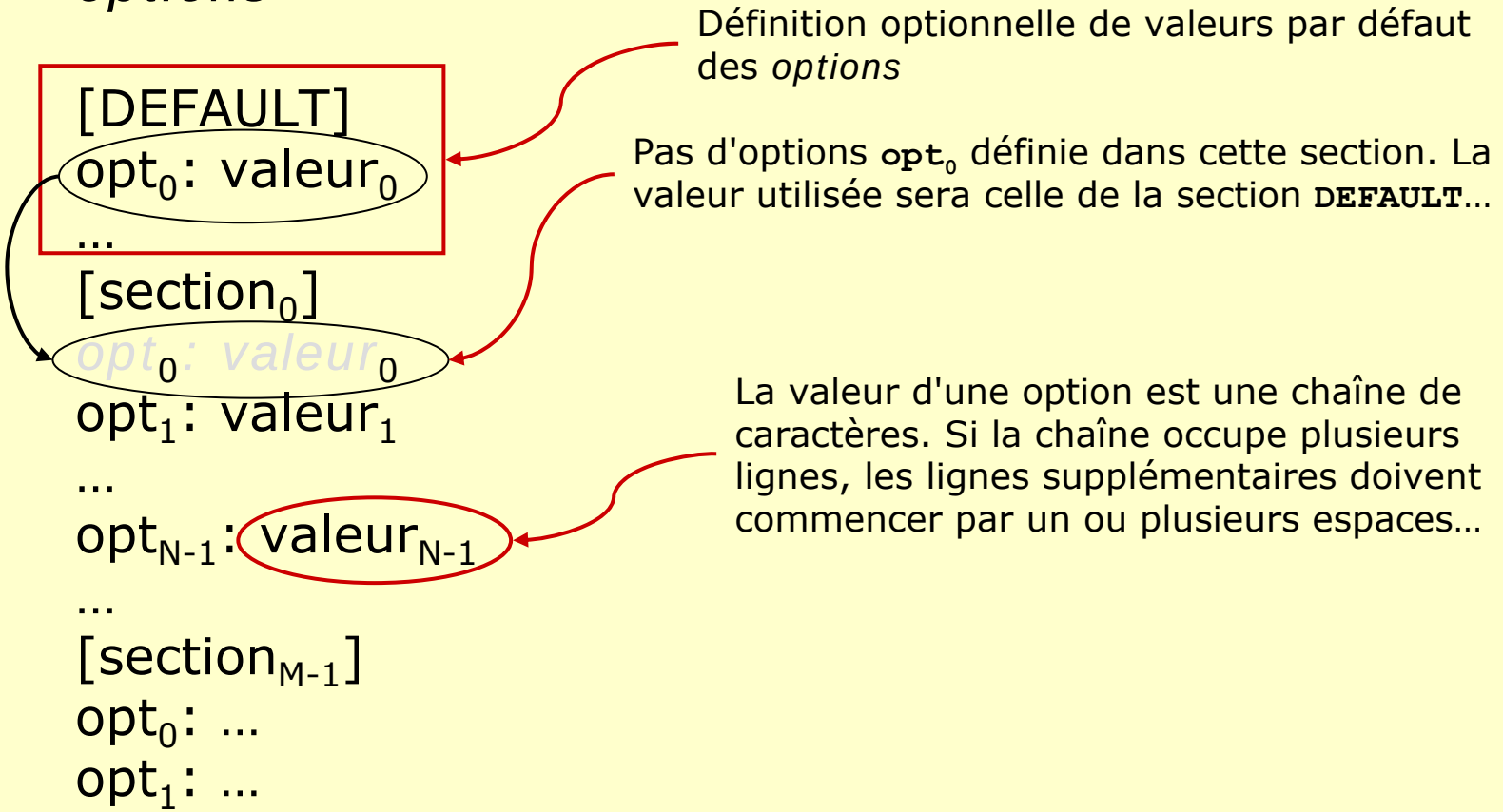
Analyse des options trouvées sur la ligne de commande : on fait une boucle sur la liste **`myopts`** de *tuples* à 2 éléments

→ ***[('–landscape', ''), ('-d', '210')]***

**`myargs`** contient les éléments de la ligne de commande qui n'ont pas été reconnus comme des options → ***['plot1.cgm', 'plot2.cgm']***

# Lecture de fichier de config : **`ConfigParser`**

- Un fichier de configuration est un fichier texte, constitué de différentes *sections* (dont une section contenant des valeurs par *défaut*) définissant des *options*



# Utilisation de `ConfigParser`

- Lecture de la configuration :

  **`config = ConfigParser.ConfigParser()`**

  **`config.read(`*`fichier_config`*`)`**

  - Attention! Si on change le contenu du fichier de configuration, il FAUT recréer un nouvel objet **`ConfigParser()`**. Il ne suffit pas de relire le contenu du fichier avec **`read(`*`fichier_config`*`)`**…
- Liste des sections :

  **`config.sections()`**
- Options disponibles dans une section (y compris les options définies dans la section **`DEFAULT`**) :

  **`config.options(`*`nom_section`*`)`**
- Valeur (chaîne) d'une option :

  **`config.get(`*`nom_section, nom_option`*`)`**
- Test de l'existence d'une section et de la disponibilité d'une option :
  - **`config.has_section(`*`nom_section`*`)`**
  - **`config.has_option(`*`nom_section, nom_option`*`)`**
- Autres méthodes de **`ConfigParser`**:

  Lire la doc dans **`lib.pdf`** ☺

# Exemples de fichiers de configuration

**py_jyp_ex_07_conf.txt**

Définition des options par défaut

```
[DEFAULT]
nom: ocean

[surf]
nom: vegetation

[veg]
nom: vegetation

[atm]
nom: atmosphere

[ocean]
nom: ocean

[ocn]
nom: ocean
```

**py_jyp_ex_08_conf.txt**

```
[atmosphere]
abrevs: atm

[ocean]
abrevs: ocean, ocn

[vegetation]
abrevs: surf, veg
```

# Exemple de *traduction* avec *py_jyp_ex_07_conf.txt* (py_jyp_ex_07.py)

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

import ConfigParser

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

cf = ConfigParser.ConfigParser()
cf.read('py_jyp_ex_07_conf.txt')

separateur = '-'
for comp in composants:
    if comp == composants[-1]:
        separateur = ''
    comp = comp.lower()
    if cf.has_section(comp):
        nom = cf.get(comp, 'nom')
    else:
        nom = 'other'

    print nom, separateur,

print
```

Importation du module

Lecture du fichier de config

```
[DEFAULT]
nom: ocean
[surf]
nom: vegetation
[veg]
nom: vegetation
[atm]
nom: atmosphere
[ocean]
nom: ocean
[ocn]
```

Utilisation du contenu du fichier de configuration pour *traduire* les abréviations

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

*Variantes : 1 - 2 – **3** - 4*

# Exemple de *traduction* avec *py_jyp_ex_08_conf.txt* *(py_jyp_ex_08.py)*

```
…
import ConfigParser

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

cf = ConfigParser.ConfigParser()
cf.read('py_jyp_ex_08_conf.txt')
noms_comp = {}
for nom in cf.sections():
    abrevs = cf.get(nom, 'abrevs')
    for abrev in abrevs.split(','):
        noms_comp[abrev.strip()] = nom

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,
…
```

```
[atmosphere]
abrevs: atm

[ocean]
abrevs: ocean, ocn

[vegetation]
abrevs: surf, veg
```

Lecture du fichier de config

Génération du dictionnaire de traduction à partir du contenu du fichier de configuration

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

Utilisation du contenu du dictionnaire pour *traduire* les abréviations

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

*Variantes : 1 - 2 – 3 - **4***

# Comparaison de la spécification de paramètres

| Méthode | Avantage(s) | Inconvénients |
|---|---|---|
| Variables (au début du script) | ☐ Le plus facile à programmer! | ☐ Modification du script pour modifier les paramètres<br>■ Droit d'écriture sur le script |
| Ligne de commande (**`sys.argv`**) | ☐ Facile à programmer | ☐ Pas très souple<br>■ L'ordre des arguments est imposé<br>■ Tous les arguments doivent être présents |
| Ligne de commande (**`getopt`**) | ☐ Très souple<br>■ Pas d'ordre des arguments<br>■ Possibilité d'arguments optionnels (valeurs par défaut) | ☐ Plus difficile à programmer<br>☐ Modification du script pour ajouter des nouveaux paramètres |
| Fichier de configuration (**`ConfigParser`**) | ☐ Facile à programmer<br>☐ Pratique si on relance souvent un script avec les mêmes paramètres<br>☐ Nombre de paramètres quelconque, gérable par le script | ☐ Il faut modifier un fichier pour modifier les paramètres |

# Appel de programmes externes

- **`os.system(`*`commande`*`)`**

  Lancement d'une commande et récupération du *statut* de l'opération (mais pas du résultat de la commande!)

  - `code_retour = os.system('date')`

- **`commands.getoutput(`*`commande`*`)`**

  Lancement d'une commande et récupération de la sortie de la commande (dans une chaîne)

  - `commands.getoutput('wc -l ~/.cshrc*')`
    `' 124 /home/users/jypeter/.cshrc\n 123 /home/users/jypeter/.cshrc~\n 247 total'`

- Pour des raisons de performance, toujours utiliser une fonction standard ou disponible dans un module plutôt qu'un appel système! ☺
  - ex : **`mkdir`**, **`rm`**, …

    → **`os.mkdirs`**, **`os.remove`**

# Trucs et astuces (1/…)

- Demande d'un paramètre à l'utilisateur
  - `raw_input('Appuyez sur <Return> pour continuer')`
  - `pi = raw_input('Valeur de PI?')`
    `pi = float(pi)`
- Variables d'environnement
  Disponibles dans le *dictionnaire* **`os.environ`**
  `os.environ['COMPUTERNAME']` → `'LSCE4032'`
- Quitter le programme en cours d'exécution
  - **`sys.exit(n)`** (**`n`** est un entier)
    Par convention, **`n==0`** si pas d'erreur, **`n<>0`** sinon

```
if not path.exists(nom_fic):
    print nom_fic, 'n\'existe pas!'
    sys.exit(1)
```

  - On peut aussi générer une *exception*!

```
if not path.exists(nom_fic):
    raise nom_fic, 'n\'existe pas!'
```

# Trucs et astuces (2/…)

- Appliquer une fonction à tous les éléments d'une liste
  - `slist = ['0.1', '1', '3.14']`

    `map(float, slist)` → `[0.100,1.0,3.14]`
  - `[c.center(3, '=') for c in ['a','b']]`

    → `['=a=', '=b=']` (*list comprehension*)
- Détermination de la date

```
t = time.localtime(time.time())
time.strftime('%d/%m/%Y', t)
  → '08/11/2005'
```

# Trucs et astuces (3/…)

- Gestion des minuscules accentuées
  - Si le script contient des minuscules accentuées (européennes), il est prudent de mettre au debut du programme (juste après le '#!...') la ligne:

```
# -*- coding: iso-8859-1 -*-
```

  - Suppression des minuscules accentuées d'une chaîne

```
s = 'Ma chaine a traduire : =é=ç=è=à='
→ 'Ma chaine a traduire : =\xe9=\xe7=\xe8=\xe0='
import string
transtab = string.maketrans('éçèà', 'ecea')
strans = s.translate(transtab)
→'Ma chaine a traduire : =e=c=e=a='
```

- Ecriture dans un fichier des commandes tapées dans l'interpréteur (pour les recopier dans un script ou les donner à quelqu'un) :

```
>>> import readline
>>> readline.write_history_file('chemin/fic_hist.py')
```

Attention! Le fichier créé est en lecture seule, donc il faut ensuite taper dans un shell :

```
chmod a+r chemin/fic_hist.py
```

# Attention! Erreurs à éviter (1/…)

- Il faut se méfier des 'import *' qui peuvent écraser des noms (de variables, fonctions, …) existants!

```
>>> e = 'abcd'
>>> import math
>>> print math.e
2.71828182846
>>> e
'abcd'
>>> from math import *
>>> e
2.718281828459045
```

# Des questions?

- …