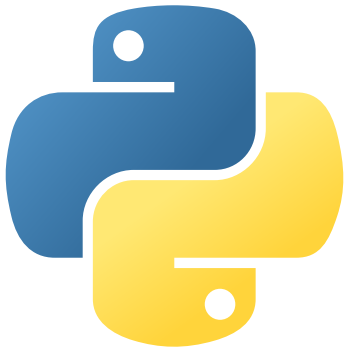
Python
niveau avancé

Vincent LE GALLIC

Séminaire Technique du Cr@ns

4 février 2014



Ce séminaire fait suite au séminaire Python niveau débutant

Fonctions mathématiques

```
import math
```

- `math.log(x [,base]),math.exp(x)`
- `math.sqrt(x)`
- `math.cos(x),math.sin(x),math.tan(x),`
  `math.acos(x),math.asin(x),math.atan(x)`
- Les mêmes avec un `h` à la fin pour les hyperboliques.
- `math.degrees(r),math.radians(d)` : convertit des radians en degrés et inversement
- Deux constantes : `math.pi`, `math.e`
- Doc complète : `http://docs.python.org/2/library/math.html`

Aléatoire

```
import random
```

- `random.random()` : renvoie un `float` aléatoire dans $[0, 1[$
- `random.randint(a ,b)` : renvoie un entier aléatoire dans $[\![a, b]\!]$
- `random.choice(l)` : choisit un élément aléatoire dans l'itérable.
- `random.randrange(start, stop, step)` équivalent à `random.choice(random.randrange(start, stop, step))` (`start` et `step` sont facultatifs)
- `random.shuffle(l)` : mélange l *in place*, `random.sample(l, k)` : donne `k` éléments choisis aléatoirement dans `l` (sans remise)
- `random.uniform(a, b)` : retourne un `float` tiré de manière uniforme dans $[a, b]$
- D'autres lois de répartition : `expovariate, gammavariate, gauss, normalvariate...`
- Doc complète : `http://docs.python.org/2/library/random.html`

Pseudo-Aléatoire ?

- `random.seed([x])` : permet d'initialiser le générateur aléatoire. Si `x` n'est pas fourni, le temps système sera utilisé. (Ce seed est effectué à l'import du module.)
- `random.getstate()`, `random.setstate(state)` : récupère/force l'état interne du générateur aléatoire. Utile pour rejouer les mêmes séquences.
- Tout ceci est pseudo-aléatoire, avec une seed. On peut générer de l'aléatoire en utilisant `random.SystemRandom`, qui utilisera `os.urandom`, qui se sert des sources d'entropie du système d'exploitation, si disponibles.

Représentation du temps

```
import time
```

- `time.time()` : date actuelle, en secondes depuis l'Epoch [1].
- `time.localtime([seconds])` : renvoie (année, mois, jour, heure, minute, seconde, `wday`, `yday`, `isdst`) de la timezone locale où `wday` est le numéro du jour dans la semaine (lundi = 0), `yday` le numéro du jour dans l'année (de 1 à 366) et `isdst` le flag Daylight Savings Time. Si `seconds` est fournit, convertit cette durée depuis l'Epoch en 9-tuple.
- `time.gmtime([seconds])` : même chose mais en temps UTC.
- `time.mktime(9-tuple)` : convertit un 9-tuple en secondes depuis l'Epoch.
- `time.clock()` : temps CPU depuis le démarrage de l'interpréteur.

1. 1er Janvier 1970 00h, UTC

- `time.strftime(format[, 9-tuple])` : Renvoie la date courante (ou celle du tuple) formatée selon `format` : par exemple `"%Y-%m-%d_%H:%M:%S"` (cf `man date`).
- `time.strptime(format, string)` : Convertit la chaîne en 9-tuple en utilisant le format.

```
import datetime
```

- `datetime.date(year, month, day)` : Créer un objet date.
- `datetime(year, month, day[, hour[, minute [, second[, microsecond[, tzinfo]]]]])` : Créer un objet timestamp.
- `datetime.timedelta` : Type de la différence entre deux dates.
- On peut faire la différence entre deux `datetime`, on obtient un `timedelta`, type qu'on peut ajouter à un `datetime`.
- Un objet `datetime` a une méthode `.strftime(format)`

Intéraction avec le système

```
import os
```

- `os.rename(old, new)`,`os.renames(old, new)` : déplace le fichier/dossier. (`renames` crée et supprime les dossiers nécessaires.)
- `os.remove(path)`,`os.rmdir(path)` : supprime le fichier/le dossier si il est vide.
- `os.chdir(path)` : change le répertoire courant.
- `os.umask(mask)` : changel l'umask en octal (et renvoie le précédent).
- `os.chmod(path, mode)`,`os.chown(path, uid, gid)`
- `os.mkdir(path[, mode])`, `os.makedirs(path[, mode])` : crée le/les répertoires.
- `os.environ` : l'environnement du shell, sous forme de dictionnaire.
- `os.getcwd()` : renvoie le chemin absolu du répertoire courant.

- `os.getenv(key[, default])`, `os.putenv(key, value)`, `os.unsetenv(key)` : récupère une variable d'environnement / définit ou en modifie une / en supprime une.
- `os.geteuid()`, `os.getgid()`, `os.getgroups()` : récupère l'uid, le gid principal ou tous les gid de tous les groupe de l'utilisateur courant.
- `os.getpid()` : obtenir le PID du processus actuel
- `os.fork()` : dupliquer le processus. Dans le processus fils, la valeur retournée est 0, dans le processus parent, la valeur est le PID du fils.
- `os.linesep` : contient le séparateur de ligne (`"\n "` sous Linux, `"\r \n "` sous Windows...), utile pour la portabilité.
- `os.symlink(src, dest)`, `os.readlink(path)` : crée un lien symbolique / lit la destination d'un lien symbolique.

```
import sys
```

- `sys.argv` : La liste des paramètres fournis sur la ligne de commande (avec la commande en `[0]`).
- `sys.exit([status])` : Quitter (avec éventuellement un code de retour, sinon 0)
- `sys.getfilesystemencoding()` : Renvoie l'encodage utilisé par le système de fichiers.
- `sys.getrecursionlimit()`, `sys.setrecursionlimit(n)` : donne/modifie la profondeur d'appel maximale.
- `sys.getrefcount(objet)` : donne le nombre de référence vers cet objet python (+1 parce que la fonction elle-même y fait référence).
- `sys.maxint`, `sys.maxunicode` : contiennent le plus grand `int` et le dernier point de code unicode.

- `sys.path` : la liste des dossiers dans lesquels l'interpréteur va chercher (dans l'ordre) les modules lorsqu'on fait un **`import`**. (ex : `sys.path.insert(0, "/meslib/prioritaires/")`, `sys.path.append("/usr/scripts/")` [2])
- `sys.stin`, `sys.stdout`, `sys.stderr` : Les fichiers d'entrée, de sortie et d'erreur standards liés au tty en cours.

2. Au Cr@ns, on préfèrera utiliser `#!/bin/bash /usr/scripts/python.sh` qui fait cet ajout une fois pour toutes.

La Programmation Orientée Objet : principe

- Une **classe** est un *moule* qui permet de créer plusieurs objets (des **instances**) de même type. La classe `Animal` me permet de créer l'instance `pet` :
  `pet = Animal(<paramètres d'initialisation>)`
- Les objets ont des **attributs** : `pet.taille_queue`, `pet.cri`. Ils peuvent différer d'une instance à l'autre, et être modifiés (normalement, pas depuis l'extérieur de l'objet).
- Ils ont également des **méthodes**, fonctions internes à l'objet : `pet.crier()`, `pet.manger(foodsize)`.

La Programmation Orientée Objet : pourquoi ?

- **Encapsulation** : la définition et le comportement d'un objet sont intérieurs à la classe, le reste du monde n'a à se soucier que de son interface.
- **Réutilisation du code** : un objet défini une fois peut très bien resservir tel quel dans un autre projet/script/...
- **DRY** [3] : permet d'écrire à un seul endroit des méthodes au lieu de répéter plusieurs fois des fonctions à peine différentes.
- **Clarté** : **Cet** objet a **ce** comportement, **ces** méthodes. Contrairement à si je trouve une fonction seule, pour laquelle je dois me demander quels objets elle peut/doit prendre en paramètres. (adéquation sémantique/syntaxe)

3. "Don't Repeat Yourself", remember ? Mais vous me faites me répéter...

Définition d'une classe

```
>>> class Chien:
        def __init__(self, poids):
            """Initialiseur"""
            # on stocke le paramètre fourni à l'initialisation dans
            # un attribut de l'objet
            self.poids = poids

        def crier(self):
            print "Ouaf !"

        def manger(self, foodsize):
            self.poids += foodsize

>>> medor = Chien(3)
>>> medor.poids
3
>>> medor.manger(5)
>>> medor.poids
8
>>> medor.crier()
Ouaf !
```

- Par convention, on nomme les classes en CamlUpperCase et les méthodes en lower_case.
- Les méthodes seront toujours appelées avec comme premier paramètre l'objet lui-même. Par convention, on utilise le mot "`self`", mais on pourrait utiliser autre chose (contrairement au "this" Java ou C++).
- Chaque méthode peut (et devrait) avoir une docstring. La docstring de la méthode `__init__` écrasera celle de la classe.
- On ne devrait pas toucher aux attributs nous-même (par exemple, faire `pet.poids = 4`), explication slide suivante.

Respect du privé

```
>>> import math
>>> class Vecteur:
        def __init__(self, x, y):
            self.x, self.y = x, y
            self._update_norme()

        def __calculer_norme(self):
            return math.sqrt(self.x**2 + self.y**2)

        def _update_norme(self):
            self.norme = self.__calculer_norme()

        def setcoords(self, newx, newy):
            self.x, self.y = newx, newy
            self._update_norme()

>>> v = Vecteur(3, 4)
>>> v.norme
5.0
>>> v.x = 10
>>> v.norme
5.0
>>> v.setcoords(6, 8)
>>> v.norme
10.0
```

Plus ou moins privé

- Les méthodes qui commencent par _ sont par convention privées. Mais Python permet tout de même d'y accéder :

```
>>> v.x, v.y = 9, 12
>>> v._update_norme()
>>> v.norme
15.0
```

- Les méthodes commençant par __ mais qui ne finissent pas par __ sont réellement privées et donc inaccessibles :

```
>>> v.__calculer_norme()
AttributeError: Vecteur instance has no attribute '__calculer_norme'
>>> v._Vecteur__calculer_norme()
15.0
```

En réalité, la méthode est renommée en ajoutant `_ClassName` au début [4], elle reste *quand même* accessible, mais c'est sale, ne faites pas ça.

4. On appelle ce procédé le **mangling**

Un peu de cosmétique

- Par défaut, c'est un peu moche :

```
>>> v
<__main__.Vecteur instance at 0x7fc1e42ec5a8>
>>> print v
<__main__.Vecteur instance at 0x7fc1e42ec5a8>
```

- On peut **surcharger** les **fonctions spéciales** qui sont appelées dans ces cas-là :

```
>>> class Vecteur:
        [...]
        def __str__(self):
            return "Vecteur de coordonnées (%s, %s)" % (self.x, self.y)

        def __repr__(self):
            return "(%s, %s)" % (self.x, self.y)
>>> v = Vecteur(3, 4)
>>> v
(3, 4)
>>> str(v)
'Vecteur de coordonn\xc3\xa9es (3, 4)'
>>> print v
Vecteur de coordonnées (3, 4)
```

En règle générale, il est déconseillé de surcharger `repr`, et fortement déconseillé de lui faire sortir du non-ASCII.

Surcharge des opérateurs

- ```
  >>> v + v
  TypeError: unsupported operand type(s) for +: 'instance' and 'instance'
  ```
- Une autre fonction spéciale particulièrement utile à surcharger : `__add__`

```
>>> Class Vecteur:
        [...]
        def __add__(self, v2):
            return Vecteur(self.x + v2.x, self.y + v2.y)

>>> u, v = Vecteur(1,2), Vecteur(3,4)
>>> print u + v
Vecteur de coordonnées (4, 6)
```

- Le principe est toujours le même : la fonction spéciale (correspondant à l'opérateur) de l'objet de gauche est appelée avec pour paramètre l'objet de droite.
- On peut prévoir de prendre en paramètre un objet d'un autre type (comme `datetime.datetime` et `datetime.timedelta`).
```

- Autre exemple, avec `__eq__` :

```
>>> class NonCaseSensitiveString:
        def __init__(self, s):
            self.s = s
        def __str__(self):
            return self.s.__str__()
        def __eq__(self, s2):
            return self.s.lower() == s2.s.lower()
>>> s = NonCaseSensitiveString("CouCou")
>>> t = NonCaseSensitiveString("coUcOu")
>>> s == t
True
```

- Autres fonctions spéciales utiles à connaître :

- `__sub__` : -
- `__div__` : /
- `__mod__` : %
- `__mul__` : *
- `__pow__` : **
- `__neg__` : opposé
- `__cmp__` : renvoie -1 si <, 0 si ==, 1 si >
- `__nonzero__` : résultat de `x != 0`, utilisé pour la conversion en booléen
- `__float__` : `float(x)`
- `__int__` : `int(x)`
- `__and__`, `__or__` : opérations logiques

DRY++ : L'héritage

- On peut factoriser le code en définissant la même méthode pour plusieurs classes.
-
```
>>> class Animal:
        def __init__(self, poids):
            """Initialiseur"""
            self.poids = poids
        def crier(self):
            print "%s !" % (self.cri)
        def manger(self, foodsize):
            self.poids += foodsize

>>> class Chien(Animal):
        def __init__(self, poids):
            Animal.__init__(self, poids)
            self.cri = "Ouaf"

>>> class Chat(Animal):
        def __init__(self, poids):
            Animal.__init__(self, poids)
            self.cri = "Miaou"

>>> snoopy, felix = Chien(5), Chat(4)
>>> snoopy.manger(2)
>>> felix.manger(2)
>>> snoopy.poids, felix.poids
(7, 6)
>>> snoopy.crier(); felix.crier()
Ouaf !
Miaou !
```

DRY++ : L'héritage

- On peut factoriser le code en définissant la même méthode pour plusieurs classes.
- Appeler l'initialiseur du parent n'est pas automatique (et donc pas obligatoire).

```
>>> class Animal:
        def __init__(self, poids):
            """Initialiseur"""
            self.poids = poids
        def crier(self):
            print "%s !" % (self.cri)
        def manger(self, foodsize):
            self.poids += foodsize

>>> class Chien(Animal):
        def __init__(self, poids):
            Animal.__init__(self, poids)
            self.cri = "Ouaf"

>>> class Chat(Animal):
        def __init__(self, poids):
            Animal.__init__(self, poids)
            self.cri = "Miaou"

>>> snoopy, felix = Chien(5), Chat(4)
>>> snoopy.manger(2)
>>> felix.manger(2)
>>> snoopy.poids, felix.poids
(7, 6)
>>> snoopy.crier(); felix.crier()
Ouaf !
Miaou !
```

DRY++ : L'héritage

- On peut factoriser le code en définissant la même méthode pour plusieurs classes.
- Appeler l'initialiseur du parent n'est pas automatique (et donc pas obligatoire).
- On peut écraser une méthode :

```
>>> class Lapin(Animal):
        [...]
        def crier(self):
            raise NotImplementedError
```

-
```
>>> class Animal:
        def __init__(self, poids):
            """Initialiseur"""
            self.poids = poids
        def crier(self):
            print "%s !" % (self.cri)
        def manger(self, foodsize):
            self.poids += foodsize

>>> class Chien(Animal):
        def __init__(self, poids):
            Animal.__init__(self, poids)
            self.cri = "Ouaf"

>>> class Chat(Animal):
        def __init__(self, poids):
            Animal.__init__(self, poids)
            self.cri = "Miaou"

>>> snoopy, felix = Chien(5), Chat(4)
>>> snoopy.manger(2)
>>> felix.manger(2)
>>> snoopy.poids, felix.poids
(7, 6)
>>> snoopy.crier(); felix.crier()
Ouaf !
Miaou !
```

- Idéalement, on devrait faire hériter toutes les classes de `object`, afin qu'elles soient des new-style classes. Cela permet un certain nombre de fonctionnalités que n'ont pas les old-style classes :
  - les properties
  - les classmethod, staticmethod
  - le support des métaclasses [5]

  Au départ, ces new-style classes ont été créées pour unifier classes et types : `type(x) == x.__class__`, ce qui n'est pas garanti pour les old-style classes. Et accessoirement, en python3, il n'y a plus que les new.

5. pour tout ça, cf séminaire python hardcore

Lire/Écrire dans un fichier

- Pour écrire du texte dans un fichier :

```
>>> f = open("path/to/file", "w")
>>> f.write("Je mets des choses\ndans mon fichier")
>>> f.close()
```

- Attention, il faut écrire des `str`, donc il faut penser à encoder avant d'éventuels `unicode`.
- Le mode `"w"` signifie qu'on écrase le contenu du fichier. Le mode par défaut est `"r"` (read), il existe également le mode `"a"` (append) pour écrire à la suite du fichier sans écraser ce qui y est déjà. `"w"` et `"a"` créent le fichier si il n'existait pas.
- Pour lire :

```
>>> f = open("path/to/file", "r")
>>> f.read()
'Je mets des choses\ndans mon fichier'
>>> f = open("path/to/file", "r")
>>> f.readlines()
['Je mets des choses\n', 'dans mon fichier']
```

Stockage facile

- Certains objets peuvent être sérialisés, ce sont None, les strings, nombres et les listes et dictionnaires qui en contiennent.
- On peut les stocker dans un fichier avec des librairies prévues à cet effet.

```
>>> import json
>>> f = open("store.json", "w")
>>> json.dump({"a" : [3]}, f)
>>> f.close()
>>> f = open("store.json")
>>> json.load(f)
{u'a': [3]}
```

- Les chaînes sont restituées sous forme d'`unicode` (et devrait aussi être fournis ainsi).
- L'objet JSON dans le fichier est à peu près human-readable.
- `pickle` est un autre module qui s'utilise de la même façon. Le format de stockage n'est pas human-readable, cependant.

Listes en compréhension

- C'est un moyen très "python-esque" de créer des listes, qui est à peu près l'équivalent d'un `map`.

```
>>> l = [2, 2, 5, 10]
>>> l2 = [i**2 for i in l]
>>> l2
[4, 4, 25, 100]
```

- Il est possible d'imbriquer plusieurs boucles :

```
>>> [a + b for a in "cdf" for b in "aei"]
['ca', 'ce', 'ci', 'da', 'de', 'di', 'fa', 'fe', 'fi']
```

- On peut également ajouter des conditions (équivalent d'un `filter`) :

```
>>> [a + 1 for a in range(7) if a % 2 == 0]
[1, 3, 5, 7]
```

- Ça marche aussi avec les dictionnaires[6] :

```
>>> {i : str(i**2) for i in range(4)}
{0: '0', 1: '1', 2: '4', 3: '9'}
```

6. En 2.7, mais pas en 2.6

Parser les paramètres commandline

- Code :

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

import argparse

# Déclaration du parseur (noter l'absence de u"")
parser = argparse.ArgumentParser(description="Message d'aide")
parser.add_argument("-v", "--verbose", help="Afficher plus d'infos",
                    action="store_true")
parser.add_argument("-f", "--file", help="Fichier à traiter",
                    action="store", type=str, metavar="Fi")
parser.add_argument("-n", dest="num", help="Numéro de la ligne",
                    action="store", type=int)

# Parsing
print "before parsing"
args = parser.parse_args()
print "ater parsing"

# Utilisation des arguments parsés
print "Valeurs : %r, %r, %r" % (args.verbose, args.file, args.num)
```

- Exécution :

```
vincent@eva $ ./test.py -h
before parsing
usage: test.py [-h] [-f Fi] [-n NUM]

Message d'aide

optional arguments:
  -h, --help        show this help message and exit
  -v, --verbose     Afficher plus d'infos
  -f Fi, --file Fi  Fichier à traiter
  -n NUM            Numéro de la ligne

vincent@eva $ ./test.py -v -f pif -n 6
before parsing
ater parsing
Valeurs : True, 'pif', 6

vincent@eva $ ./test.py -f pif
before parsing
ater parsing
Valeurs : False, 'pif', None

vincent@eva $ ./test.py -f pif -n
before parsing
usage: test.py [-h] [-v] [-f Fi] [-n NUM]
test.py: error: argument -n: expected one argument
```

Parser les paramètres commandline

- Regarder la doc et les docstrings d'argparse pour voir toutes les possibilités
- Notamment, `parser.add_mutually_exclusive_group`
- Il existe également le module `optparse`, moins bien.

Expressions régulières

- Notion venant de l'info théorique, très utile et utilisée.
- `perl`, `sed`, `python`, `grep` en ont, mais à chaque fois avec des différences de syntaxe et d'expressivité
- Un peu de syntaxe des regexp :
  - `'.'` : Tout sauf `'\n '` [7]
  - `'^'`/`'$'` : Début/Fin de chaîne [8]
  - `'(...)'` : Délimite un groupe. Il sera accessible après et référençable dans la regexp elle-même (`'\1'`, `'\2'`...) [9].
  - `'*'` : Le précédent groupe, 0 ou plus de fois.
  - `'+'` : Le précédent groupe, 1 ou plus de fois.
  - `'?'` : Le précédent groupe, 0 ou 1 fois.
  - `'{m}'`, `'{m,n}'`, `'{,n}'`, `'{m,}'` Le précédent groupe, respectivement `m` fois exactement, entre `m` et `n` fois, au plus `n` fois, au moins `m` fois.

7. Si DOTALL est activé, matche aussi les newline
8. Si MULTILINE, matche après/avant chaque newline
9. Attention, un groupe peut exister dans d'autres cas : `'a|b'`, `'[a-z]'`

Les regexp en python

- Le module : `re`
- Les choses qu'on peut faire avec une regexp :
  - `match` : vérifier que le motif matche le texte depuis son début
  - `search` : trouver la première occurrence du motif dans le texte
  - `findall` : trouver toutes les occurrences disjointes du motif dans le texte, renvoie une liste
  - `finditer` : presque [10] pareil, mais renvoie un itérateur
  - `sub` : trouver toutes les occurrences d'une regexp et les remplacer par quelque chose (éventuellement dépendant de ce qui a exactement été matché)
- Les deux façons de le faire :
  - `re.<nom_de_la_fonction>(motif, texte)`
  - `regexp = re.compile(motif)`
    `regexp.<nom_de_la_fonction>(texte)`

    Cette méthode permet de rajouter le paramètre `flags=re.UN_FLAG`

10. Renvoie des match objects et non pas le résultat de `.groups()`

Regexp : pratique

- Simple matching (vérifier si un truc matche une regexp) :

```
>>> import re
>>> re.match('ab+c', 'abbc')
<_sre.SRE_Match at 0x2cf9780>
>>> re.match('ab+c', 'abbcd')
<_sre.SRE_Match at 0x2cf9ac0>
>>> re.match('ab+c', 'acd')
```

- Accès aux groupes :

```
>>> match = re.match('(a{2,})((b+)k)(c?)', 'aaabkd')
>>> match.group()
'aaabk'
>>> match.groups()
('aaa', 'bk', 'b', '')
```

Les escape sequences

Elles permettent de matcher plus de choses :

- `'\*'`, `'\?'`, `'\$'` . . . : pour matcher les caractères spéciaux (désactiver leur comportement)
- `'\1'`, . . ., `'\99'` : matche les groupes dans l'ordre
- `'\s '` : matche les whitespace, équivalent à `'[ \t \n \r \f \v ]'` si le flag UNICODE est inactif, si il est actif, matche tout caractère espace (notamment les espaces insécables)
- `'\S '` : matche tout non-whitespace
- `'\d '` : matche tout digit, équivalent à `'[0-9]'`, plus si UNICODE
- `'\D '` : matche tout non-digit
- . . .
- Voir la liste complète dans la doc

Les flags

On peut modifier le comportement avec un ou plusieurs flags :

```
>>> text = "a1a\na3"
>>> r = re.compile("a.")
>>> r.findall(text)
['a1', 'a3']
>>> r = re.compile("a.", flags=re.DOTALL)
>>> r.findall(text)
['a1', 'a\n', 'a3']
```

- DEBUG : affiche les informations sur la regexp au moment de sa compilation
- IGNORECASE : rend la regexp insensible à la casse
- MULTILINE : `'^'` et `'$'` matchent en début et fin de chaque ligne
- DOTALL : `'.'` matche aussi un `'\n'`
- UNICODE : change le comportement de certaines escape-sequences. Par exemple, `'é'` sera considéré comme un caractère alphanumérique et l'espace insécable sera matché par `'\s'`

Pour utiliser plusieurs flags, on les additionne.

Plus de syntaxe des regexp

- `'|'` : "Ou" entre deux groupes : `'(aa|bb)'` matche `'aa'` et `'bb'`. Attention, `'aa|bb'` matche `'aab'` et `'abb'`.
- `'[...]'` : Ensemble (set) de caractères
  - Un "ou++" : `'[abcd]'`
  - Range : `'[a-zA-Z]'` (`'\-'` ou en début/fin de set pour un vrai tiret)
  - Les caractères spéciaux perdent leur sens : `'[(+*)]'` matche un de ces caractères
  - On peut demander le complémentaire d'un set : `'[^a-z]'`
  - Pour inclure un `']'` dans un set, l'échapper à coup de backslash, ou le mettre au début : `'[a\]b]'`, `'[]ab]'`
- Groupe spéciaux
  - `'(?:...)'` : groupe non-capturant. Ne sera pas affiché par `.groups()` et ne peut pas être référencé par `'\1'`...
  - `'(?iLmsux)'` : activer un ou des flags. C'est une alternative à la méthode consistant à les déclarer dans le `re.compile`.
  - `'(?P<nom>...)'` : groupe capturant et nommé. On peut le référencer par son numéro ou par `'(?P=nom)'`. Apparaîtra dans `match.groupdict()`.

Quelques astuces supplémentaires

- Application, avec `urllib` en bonus :

```
>>> url = "https://wiki.crans.org/VieBde/PlanningSoirees/LeCalendrier"
>>> url += "?action=raw"
>>> texte = urllib.urlopen(url).read()
>>> regex = "(?:start|end):: (?P<date>[-0-9]+) (?P<heure>[0-9:]+)"
>>> [m.groupdict() for m in re.finditer(regex, texte)]
[{'date': '2015-03-28', 'heure': '00:00'},
 {'date': '2015-03-29', 'heure': '23:59'},
 ...
 {'date': '2008-03-06', 'heure': '14:00'}]
```

- Attention, les multiplicateurs sont par défaut gloutons (= matchent la plus longue chaîne possible).
  Alternative non-gloutonne : `'*?'`, `'+?'`, `'??'`, `'{m,n}?'`

```
>>> re.match('<.*>', '<h1>Title</h1>').group()
'<h1>Title</h1>'
>>> re.match('<.*?>', '<h1>Title</h1>').group()
'<h1>'
```