Python en Action

Nicolas Paris

September 20, 2011

http://www.nicosphere.net

nicolas.caen@gmail.com

Table des matières

Préface

Ce document est issue principalement de billets pris sur mon blog, revu pour l'occasion. Aillant depuis quelques temps cumulé des suites de petits tutoriels un peu sous forme de recettes, je pense qu'il pouvait être intéressant de les réunir sous un format PDF. Ce format s'y prêtait plutôt bien, du fait d'avoir traité quelques thèmes sous différents aspects, notamment avec la partie Ncurses et Tests Unitaires. Le style est certainement plus celui utilisé pour un blog, bien que j'en ai apporté quelques modifications pour garder une cohérence.

Le document est sous licence libre *Créative Commun CC-BY-SA* [1] et puisqu'un PDF ne serait pas vraiment considéré comme libre sans les sources, elles peuvent être téléchargées sur Github [2]. Vous avez le droit de le télécharger, le redistribuer, le modifier et même de l'imprimer pour le lire dans les toilettes. Merci de copier ce document ! Je demande juste d'être cité, Nicolas Paris, et de faire un lien vers mon blog, http ://www.nicosphere.net.

Vous pouvez me contacter via email : nicolas.caen@gmail.com

La version de Python utilisé est généralement la 2.7, parfois la 3.2. J'ai généralement fait mon possible pour que ce soit clair. Une précision tout de même, étant sous Arch Linux, Python par défaut est le 3.2.x, mais j'ai essayé autant que possible de garder dans le texte Python 2.7 comme celui de défaut, pour éviter les confusions de ceux ne connaissant pas cette spécificité d'Arch Linux.

J'espère que ce document trouvera lecteurs, bien que ce soit mon premier "ebook".

– Nicolas Paris

1. http ://creativecommons.org/licenses/by-sa/2.0/fr/
2. https ://github.com/Nic0/python_en_action

CHAPITRE 1

Utilisation de Modules

1.1 Consulter la documentation en local

L'idée est de lire la documentation de tout les modules dans un navigateur, qu'ils soient natif à Python ou installée implicitement, le tout en local. Plusieurs avantages à ça :
– Avoir toujours la documentation correspondante à la version installée
– Naviguer rapidement entre les modules tiers, installé par l'utilisateur
C'est une façon assez classique de procéder, qu'on retrouve dans certains langages. J'en écrivais un billet pour ruby [1] de ce même procédé. On retrouve également pour Perl, Perldoc-server [2].

Cependant, je le trouve vraiment moins bien fait que celui de Ruby, qui a pour avantage d'être bien plus facilement navigable, la possibilité d'afficher le code correspondant (vraiment utile). Sans parler de l'esthétisme global bien meilleur. Il est dommage qu'il ne soit pas un peu plus aboutis que ça.

Le fonctionnement est très simple, on lance le serveur comme suit :

```
$ pydoc -p 1337
Server ready at http://localhost:1337/
Server commands: [b]rowser, [q]uit
server>
```

Il vous suffit à présent de se rendre sur http ://localhost :1337 et de naviguer entre les différents modules affichés.

Le choix du port est arbitraire.

1. http ://www.nicosphere.net/lire-la-documentation-rdoc-des-gem-avec-un-navigateur-2239/
2. http ://search.cpan.org/dist/Perldoc-Server/

1.2 Piles et Files avec Deque

1.2.1 Introduction

En programmation, les piles [3] et files [4], aka LIFO (last in, first out) et FIFO (first in, first out) sont des incontournables. Il s'agit de stocker des données, et surtout de l'ordre dans lequel on récupère celle-ci. Pour information, j'avais écrit Exemple de file avec deux pointeurs en langage C [5] qui est un exemple de liste chaînée utilisée comme une file en C.

Comme Python est un langage de haut niveau, il offre une façon simple de mettre en place l'un ou l'autre. On va voir qu'il correspond à une classe du module *collections*.

Pourquoi ne pas utiliser simplement une liste ?

Il est possible d'obtenir le même comportement avec une liste, surtout qu'elle offre un peu près les mêmes méthodes (pop), cependant, *deque* est spécialement prévu pour cette effet, et est donc optimisé dans ce sens. Il est donc préférable d'utiliser *deque* si l'on sait qu'on l'utilise comme fifo ou lifo.

1.2.2 Utilisation

Rien de bien compliqué, regardons cette suite de commande effectuée directement avec l'interpréteur.

– Fonctionnement comme pile (lifo)

```
>>> from collections import deque
>>> pile = deque([1, 2, 3, 4, 5])
>>> from collections import deque
>>> pile = deque([1, 2, 3, 4, 5])
>>> pile.pop()
5
>>> pile.pop()
4
>>> pile.append(6)
>>> pile
deque([1, 2, 3, 6])
```

– Fonctionnement comme file (fifo)

```
>>> from collections import deque
>>> pile = deque([1, 2, 3, 4, 5])
>>> pile.popleft()
1
>>> pile.popleft()
2
>>> pile.append(6)
```

3. http ://fr.wikipedia.org/wiki/Pile_(informatique)
4. http ://fr.wikipedia.org/wiki/File_(structure_de_données)
5. http ://www.nicosphere.net/exemple-de-file-avec-deux-pointeurs-en-langage-c-1752/

```
>>> pile
deque([3, 4, 5, 6])
```

Et après ?

Le module vient avec d'autres méthodes pouvant être utile dans ce genre de cas tel que par exemple :
– appendleft()
– clear()
– extend()
– reverse()
– rotate(n)
Pour plus de détails sur les possibilités, lisez la documentation officiel sur ce module [6].

Ce module est assez simple à prendre en main, mais toujours utile si on sait qu'on veut mettre en place rapidement une fifo ou lifo. Ce n'est qu'une méthode parmi d'autre pour obtenir ce que l'on souhaite, mai l'avantage de *deque* est d'être très simple à l'emploi, tout en étant plus performant. Éventuellement, pour les plus curieux, l'utilisation de *queue.LifoQueue* pourrait intéresser, mais l'idée ici était de garder une utilisation simplifié au maximum.

1.3 Tenir un fichier de log

Il est souvent utile de tenir un fichier de log dans plusieurs situations, comme archiver différentes erreurs, ou même pour les débugs. Python fournit nativement un module [7] très bien fait et complet pour faire cela, avec une redirection de l'information soit en console ou dans un fichier, et permettant plusieurs niveaux d'importance de logs, du message de débug aux erreurs critiques.

Ce module permet d'aller bien plus loin de ce que j'en aurai besoin, comme un serveur de log pour centraliser les logs de clients, des logs tournantes ou encore l'envoi de journaux par SMTP (mail). Il n'est pas question ici de tout voir. Une petite introduction qui permet de tenir simplement un fichier de log, pour une prise main rapide.

1.3.1 Les niveaux de messages

Le module défini plusieurs niveau d'importance dans les messages, qui sont dans l'ordre croissant :
– Debug
– Info
– Warning
– Error
– Critical

Note : Dans cet exemple, je me suis servis de Python3, je n'ai pas vérifié la compatibilité avec Python2, mais son usage doit être très similaire.

6. http ://docs.python.org/py3k/library/collections.html#collections.deque
7. http ://docs.python.org/library/logging.html

1.3.2 Premier essai

Essayons un script rapide, qui n'a pas vraiment d'intérêt si ce n'est de voir ce qu'il s'y passe.

```
import logging

logging.warning('coin')
logging.debug('pan!')
```

On execute le programme, et on obtient la sortie suivante :

```
$ ./logger.py
WARNING:root:coin
```

Plusieurs remarques :
- Seul le warning s'est affiché, s'expliquant par la configuration de défaut n'affichant uniquement les messages à partir du seuil warning. On verra comment le modifier par la suite.
- La sortie contient le niveau (WARNING), un 'root' un peu obscure, et le message, la sortie n'est pas très jolie ni très verbeuse, mais fait ce qu'on lui demande.
- Le message est envoyé en console et non pas dans un fichier comme nous le souhaitons.

1.3.3 Fichier et niveau de log

Regardons pour enregistrer ce message dans un fichier maintenant. Cette redirection peut être utile par exemple si le programme est écrit en ncurses ou qu'on ne possède pas vraiment de retour visuel. Mais également pour avoir des informations sur une plus longue période.

Dans le script précédent, on rajoute avant les messages de warning et debug la ligne suivante :

```
logging.basicConfig(filename="prog.log",level=logging.DEBUG)
```

On exécute et regarde le fichier *prog.log*, on obtient exactement à quoi l'on s'attendait, les deux messages. Par simplicité, le fichier est renseigner par un chemin relatif, et se retrouve donc à l'endroit de l'execution du programme. On pourrait mettre un chemin absolu, en concaténant des variables d'environnement.

1.3.4 Logger les variables

Comme il peut être utile de rajouter quelques variables, afin d'avoir un aperçu au moment du log, on peut le faire en les rajoutant comme argument, comme on le ferait pour une *string*, comme dans l'exemple suivant :

```
import logging

a = 'coin'
```

```
b = 'pan!'
logging.warning('a:%s b:%s', a, b)
```

La sortie sera la suivante :

```
$ ./logger.py
WARNING:root:a:coin b:pan!
```

1.3.5 Ajouter l'heure aux messages

Dans un fichier de log, on souhaite généralement avoir à chaque message, une indication de l'heure, et éventuellement de la date, il est possible de le faire, une fois pour toute, de la façon suivante :

```
import logging
import time

logging.basicConfig(
    filename='prog.log',
    level=logging.INFO,
    format='%(asctime)s %(levelname)s - %(message)s',
    datefmt='%d/%m/%Y %H:%M:%S',
    )

logging.info('Mon programme démarre')
time.sleep(2)
logging.warning('Oh, quelque chose ce passe!')
logging.info('fin du prog')
```

Nous obtenons dans le fichier de log les informations suivantes :

```
16/07/2011 13:33:26 INFO - Mon programme démarre
16/07/2011 13:33:28 WARNING - Oh, quelque chose ce passe!
16/07/2011 13:33:28 INFO - fin du prog
```

La sortie est plus lisible et exploitable.

1.3.6 Fichier de configuration

Pour avoir la possibilité de séparer la configuration du logger et le code en général, il est possible de tenir un fichier spécifique à la configuration, il supporte deux syntaxes différentes, basé sur ConfigParser ou YAML.

Comme ce module permet un niveau de personnalisation assez avancé, il est normal que cela se ressente sur la verbosité du fichier de configuration. Pour ma part, je ne pense pas en avoir besoin dans l'immédiat, donc je vous renvoi sur la documentation officiel, et notamment ce how-to [8] qui est un bon endroit pour commencer, et dont ce billet s'en inspire fortement.

8. http ://docs.python.org/py3k/howto/logging.html

1.4 Script concret pour Urllib

Je l'accorde que ce script à été fait un peu à La Rache [9] un samedi soir et sur un coup de tête, le titre original était d'ailleurs *Le script idiot du dimanche*, titre honteusement repompé de PC INpact (les liens idiots du dimanche) mais qui était de circonstance. Je pense qu'il peut faire une petite introduction à Urllib, tout en restant amusant, chose importante lorsqu'on apprends, et surtout de façon auto-didacte.

Mais la question qu'on peut se poser est :

Mais que fait ce script ? !

Vous connaissez peut être le site downforeveryoneorjustme.com ? Il permet de s'assurer que pour un site qui vous est inaccessible, qu'il ne s'agisse pas d'un problème venant de chez vous. L'intérêt est de passer par un autre site, afin de s'assurer que le problème rencontrer n'est pas entre le premier site et soi. Ce script est tout simplement un front-end de ce site, dont l'usage en console est on ne peu plus simple :

```
$ ./downforeveryoneorjustme http://www.nicosphere.net
It's just you. http://www.nicosphere.net is up.
```

De plus, l'affichage se fait en couleur, rouge si c'est down, et vert si le site est fonctionnel (j'ai hésité longuement sur le choix des couleurs).

Vous pouvez cloner le dépôt git fait à cette occasion. Si si, j'ai fais un dépôt rien que pour ça...

Note : Je n'ai testé le script qu'avec **Python 2.7**, et il faudra certainement adapter la partie avec urllib, comme il y a quelques changement entre les deux versions majeurs de Python.

```
git clone git://github.com/Nic0/DownForEveryoneOrJustMe.git
```

Mais jetons un œil au script tout de même.

```
import re
import sys
import urllib

class DownForEveryone(object):

    base_url = 'http://www.downforeveryoneorjustme.com/'

    def __init__(self):
        self.parse_url()
        self.is_url()
        self.get_answer(
            urllib.urlopen(self.base_url+self.url).read()
        )

    def parse_url(self):
```

9. http ://www.nicosphere.net/utilisez-la-methode-la-rache-pour-vos-projets-1363/

```
        try:
            self.url = sys.argv[1]
        except IndexError:
            self.usage()

    def is_url(self):
        url_regex =
        'http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\(\),]|(?:%[0-9a-fA-F][0-9a-fA-F]))+'
        resp = re.findall(url_regex, self.url)
        if len(resp) != 1:
            print 'The argument does not seems to be an URL'
            self.usage()

    def get_answer(self, response):
        up = 'It\'s just you.'
        down ='It\'s not just you!'
        if up in response:
            print '\033[1;32m{} {} {}\033[1;m'.format(up, self.url, 'is up.')
        elif down in response:
            print '\033[1;31m{} {} {}\033[1;m'.format(
                down, self.url, 'looks down from here.')
        else:
            print 'Error to find the answer'

    def usage(self):
        print 'Usage:'
        print ' {} http://www.exemple.org'.format(sys.argv[0])
        sys.exit(0)

if __name__ == '__main__':
    DownForEveryone()
```

Il n'y a pas beaucoup à dire sur le fonctionnement simpliste du script. L'argument passé en commande est vérifié, voir s'il existe, que c'est bien un lien, puis on récupère la page demandé, sur lequel on vérifie simplement la présence d'une petite phrase clé pour l'un ou l'autre des évènements. Puis on affiche le résultat avec de belles couleurs. Et c'est tout !

Le script méritait donc bien le titre de ce billet.

Il y aurait bien sûr beaucoup d'amélioration à effectuer ou même des simplification, mais je le laisse tel quel pour montrer qu'en vraiment peut de temps, et sans connaitre très bien Python, on peut arriver à un résultat amusant et concret.

1.5 Parser un fichier de configuration

Il y a de multiples raisons de mettre en place un fichier de configuration pour une application. Nous allons voir comment faire cela grâce à un module de python appelé ConfigParser [10] qui sert... à parser une configuration de façon très simple. Il est dans l'esprit de ce langage, de ne

10. http ://docs.python.org/library/configparser.html

pas recréer un module afin de prendre en compte un fichier de configuration, fonctionnalité très courante.

1.5.1 Pourquoi un fichier de configuration ?

Utiliser un fichier de configuration permet d'obtenir une application plus souple, si vous faites un choix arbitraire dans votre programme, demandez vous s'il ne serait pas plus efficace de l'extraire, et de le mettre en tant que paramètre, modifiable par l'utilisateur. D'une part il est plus agréable de les manipuler de façon séparé et regrouper, et surtout cela évite qu'un utilisateur voulant changer une variable le fasse en fouillant dans le code, au risque de changer un mauvais paramètre par mégarde dans votre code source.

1.5.2 Présentation du module ConfigParser

Puisqu'un module existe pour Python, autant l'utiliser. Un des avantages est d'avoir la possibilité de commenter vos configuration en commencent par *# (dièse)* ou *; (point-virgule)* une ligne, celle-ci sera ignoré. Le fichier suit une structure assez simple, qu'on retrouve régulièrement, basé sur un système de section, et d'argument/valeurs. Pour résumer dans un schéma :

```
[my_section]
coin = pan
answer = 42
```

On peut de la sorte avoir plusieurs noms de section et ce n'est là que pour donner une idée basique du module.

1.5.3 Lire dans un fichier de configuration

Premier exemple tout simple, nous allons lire dans un fichier quelques arguments, pour les afficher en retour dans la console. Par commodité, on va placer le fichier de configuration à l'endroit où sera exécuté le programme. Le nom du fichier serra ici *conf.rc* mais aucune importance, du moment qu'on s'y retrouve.

conf.rc :

```
[user]
nom = Dupond
age = 42
```

Il nous reste plus qu'à écrire le bout de code en python pour interpréter ça.

config.py :

```
import ConfigParser

def main ():

    config = ConfigParser.RawConfigParser()
```

```
    config.read('conf.rc')

    nom = config.get('user', 'nom')
    age = config.get('user', 'age')

    print 'Bonjour M. %s, vous avez %s ans' % (nom, age)

if __name__ == '__main__':
    main()
```

On le rend exécutable :

```
chmod +x config.py
```

Puis on le lance :

```
$ ./config.py
Bonjour M. Dupond, vous avez 42 ans
```

Ça fait ce qu'on voulais, mais voici quelques explications :

```
import ConfigParser
```

On importe le module, rien de compliqué ici.

```
config = ConfigParser.RawConfigParser()
```

Sûrement la ligne la plus obscure du code, on instancie simplement un objet (config) de la classe RawConfigParser venant du module ConfigParser. Et c'est grâce à cet objet que tout va se jouer.

```
config.read('conf.rc')
```

Nous indiquons ici quel est le fichier de configuration que nous utilisons. Pour simplifier l'affaire, le fichier est en relatif (par rapport à l'endroit d'où est exécuté le programme), et surtout, il n'y a pas de vérification que le fichier existe, ni qu'il est lu correctement. Je suis sûr que vous trouverez comment le faire de vous même.

```
nom = config.get('user', 'nom')
```

C'est là qu'on lit vraiment les variables, un détail à noté, tout comme on peut le deviner par la suite, la variable age est considéré comme une chaine de caractère, et non comme un nombre à part entière, dans l'état il ne serait pas possible d'effectuer des calcules dessus, mais pouvant être convertie facilement de la sorte :

```
age = int(config.get('user', 'age'))
```

De la sorte, age est maintenant effectivement un entier, et nous pourrions en faire toute sorte de calculs afin de trouver l'âge du capitaine.

Je ne pense pas que le reste ait besoin de plus d'explication, plusieurs remarques cependant.

– Le plus important, on remarque qu'il est simple d'interpréter de la sorte un fichier de configuration, le tout en une poignée de lignes seulement.
– Il n'y a aucune gestion d'erreur ici, s'il manque un argument, ou même la section, le programme va planter lamentablement... Mais nous allons voir ça maintenant.

1.5.4 Gestion d'erreurs

Nous allons voir deux cas ici.

try/except block

```
import sys
import ConfigParser

def main ():

    config = ConfigParser.RawConfigParser()
    config.read('conf.rc')
    try:
        nom = config.get('user', 'nom')
        age = config.get('user', 'age')
    except ConfigParser.Error, err:
        print 'Oops, une erreur dans votre fichier de conf (%s)' % err
        sys.exit(1)

    print 'Bonjour M. %s, vous avez %s ans' % (nom, age)

if __name__ == '__main__':
    main()
```

Et on essaye avec un fichier de config erroné suivant :

```
[user]
nom = Dupond
```

On exécute, et regarde la sortie :

```
$ ./config.py
```

Oops, une erreur dans votre fichier de conf (No option 'age' in section : 'user')

has_section, has_option

Le module viens avec deux méthodes permettant de vérifier la présence de section ou d'option, on peut donc s'en servir, avec quelques choses ressemblant à ça par :

```
if config.has_option('user', 'nom'):
    nom = config.get('user', 'nom')
else:
    nom = 'Default_name'
if config.has_option('user', 'age'):
    age = config.get('user', 'age')
else:
    age = '42'
```

On affecte également des valeurs par défaut si une option n'est pas trouvé, on peut noté également, que la gestion d'erreur sur la section n'est pas faite ici, uniquement les options.

1.5.5 Écrire dans un fichier de configuration

Jusqu'ici, nous avons vu comment lire les données d'un fichier de configuration. Pendant qu'on y est, autant jeter un œil sur la façon d'écrire, et donc sauvegarder, une configuration. Dans la lignée de ce qui à déjà était fait, même nom de fichier, même section et options.

Dans un premier temps, on supprime le fichier de configuration *conf.rc* si vous l'aviez gardé depuis l'exercice plus haut, et on écrit dans *config.py* le code suivant :

```
import ConfigParser

def main ():

    config = ConfigParser.RawConfigParser()

    config.add_section('user')
    config.set('user', 'nom', 'Dupond')
    config.set('user', 'age', '42')

    with open('conf.rc', 'wb') as conf_file:
        config.write(conf_file)

if __name__ == '__main__':
    main()
```

On rend exécutable avec *chmod +x config.py*, puis on exécute le script, et on observe le résultat en ouvrant le fichier conf.rc, il contient exactement ce qu'on attendait.

Pour les explications, plus courte cette fois ci. On vois qu'on rajoute la section avec la méthode *add_section*, pour lequel on affecte les options avec la méthode *set* qui prends trois arguments :

```
config.set(section, options, valeur)
```

L'utilisation de *with open* pour la lecture ou l'écriture de fichier à l'avantage de ne pas avoir besoin de le refermer, et celà quoi qu'il arrive, même si l'écriture est defectueuse. Cette façon de procédé est privilégié.

1.5.6 Sauvegarder un dictionnaire comme configuration

Imaginons que nous voulons sauvegarder un dictionnaire qui nous sert de configuration dans un fichier, on peut donc effectuer de la sorte :

```
import ConfigParser

def main ():

    config = ConfigParser.RawConfigParser()
    params = {
            'Linux': 'Torvalds',
            'GNU': 'RMS',
            'answer': '42',
        }
    config.add_section('params')
    for arg in params:
        config.set('params', arg, params[arg])

    with open('conf.rc', 'wb') as conf_file:
        config.write(conf_file)

if __name__ == '__main__':
    main()
```

On exécute, et regarde le résultat obtenu, et c'est ce que nous voulions, un fichier contenant ce dictionnaire et sous forme *option* = *valeur*.

Voilà, cette introduction au module ConfigParser [1] touche à sa fin, C'est un module qui n'est pas compliqué à prendre en main, il est conseillé de lire la documentation fournis pour plus amples détails. En espérant motiver certain à utiliser un fichier de configuration plutôt que d'écrire « en dur » les variables directement dans le fichier source.

CHAPITRE 2

Python et Ncurses

2.1 Transparence avec Ncurses

J'ai recherché sur le net comment avoir la transparence avec Python et le module curse, mais l'information n'était pas très clair, ni dans la documentation officiel, ni sur le web. C'est en lisant le code des autres que j'ai trouvé.

Donc oui, il est possible d'avoir la transparence avec le module curses de défaut de Python. Il fonctionne avec les versions 2.7.x et 3.2.x.

Attention de ne pas appeler votre fichier curses.py, comme lorsqu'on fait des essais, on a tendance à appeler le fichier avec un nom parlant, mais cela interférerai avec l'import du module curses.

Cet exemple est également un bon point de départ pour afficher un mot avec ncurses.

2.1.1 Le code

Un exemple complet, qui affiche un gros `Hello World` en rouge, et avec fond transparent :

```
import curses

def main():

    scr = curses.initscr()
    curses.start_color()
    if curses.can_change_color():
        curses.use_default_colors()
        background = -1
    else:
        background = curses.COLOR_BLACK

    curses.init_pair(1, curses.COLOR_RED, background)
    scr.addstr(2, 2, 'Hello World', curses.color_pair(1))
    scr.getch()
```

```
if __name__ == '__main__':
    main()
```

2.1.2 Quelques explications

```
if curses.can_change_color():
    curses.use_default_colors()
```

La première ligne s'assure que le terminal est capable d'afficher la transparence, et la seconde l'active, en assignant la valeur `-1` à la variable background. Si le terminal ne supporte pas la transparence, on remplace par un fond noir.

2.2 Déplacement de fenêtre

Petit exercice, pour une approche de la bibliothèque curses en douceur. Le but, tout simple, est de créer une petite fenêtre à l'intérieur de la console, cette fenêtre est visualisable par un carré (4x6), et de la déplacer dans sa console à l'aide des flèches directionnel. Même si l'émerveillement de voir apparaître une fenêtre reste limité, ce n'est pas moins un bon exercice afin de voir quelques fonctions propre à ncurses.

L'approche est organisé étape par étape, avec en fin de chapitre le code complet de l'application.

2.2.1 Obtenir une petite fenêtre

Dans un premier temps, on va afficher une petite fenêtre et... c'est tout.

```
import curses
import curses.wrapper

def main(scr):
    curses.use_default_colors()
    curses.curs_set(0)
    win = scr.subwin(4, 6, 3, 3)
    win.border(0)
    win.refresh()
    win.getch()

if __name__ == '__main__':
    curses.wrapper(main)
```

On rend le tout exécutable avec un chmod +x, puis on regarde le résultat. Une fenêtre s'affiche, nous sommes émerveillé, on appuis sur une touche, le programme quitte. Mais déjà quelques explications s'impose, en commençant par l'appel du `main()` qui ne se fait pas comme on a l'habitude de voir.

```
if __name__ == '__main__':
    curses.wrapper(main)
```

`curses.wrapper` est une fonctionnalité de python, permettant d'obtenir l'initialisation rapide de curses, elle prend en argument une fonction, alors on a l'habitude de mettre des variables en argument, mais ce n'est pas grave, on met la fonction `main` en argument, il faut noter qu'il n'y a pas besoin des parenthèses dans ce cas, puisqu'on passe la référence de cette fonction, et non un message d'exécution de celle ci.

L'autre important avantage d'utiliser le wrapper, c'est de vous laisser la console dans un état *normal* lorsque vous quittez ou que le programme plante, sinon, il vous le laisse dans un état qui vous obligerez quasiment à fermer votre console, car l'affichage y serait très pénible.

La fonction qui est appelé reçois en argument l'écran initialisé, d'où le `scr` un peu inattendu dans la fonction de main :

```
def main(scr):
```

Passé ce détail, le reste est assez intuitif.

On initialise la transparence, comme je l'expliquais dans le chapitre précédent, et on désactive le curseur, car pour dessiner une petite fenêtre, nous n'en avons pas vraiment besoin.

```
curses.use_default_colors()
    curses.curs_set(0)
```

Vient ensuite la création de la fenêtre avec une méthode `subwin` attaché à l'écran `scr` :

```
src.submin(taille_y, taille_x, position_y, position_x)
win = scr.subwin(4, 6, 3, 3)
# On y affecte des bordures pour bien voir la fenêtre
win.border(0)
# On rafraîchis pour l'affichage.
win.refresh()
# On demande de saisir une touche, histoire d'avoir le temps
# de voir le résultat.
win.getch()
```

2.2.2 Faire bouger la fenêtre

Comme le but étant de faire bouger un peu la fenêtre, voyons comment faire évoluer le code pour cela. Par commodité, je vais bouger le `main` dans sa propre classe. Jetons directement un œil au code.

```
import curses
import curses.wrapper

class MovingWindow(object):

    def __init__(self, scr):
```

```
        self.scr = scr
        self.pos = [3, 3]
        self.size = [4, 6]
        self.init_curses_mode()
        self.draw_window()
        self.handle_key_stroke()

    def init_curses_mode(self):
        curses.use_default_colors()
        curses.noecho()
        curses.curs_set(0)

    def draw_window(self):
        self.scr.erase()
        self.win = self.scr.subwin(self.size[0], self.size[1],
                                   self.pos[0], self.pos[1])
        self.win.border(0)
        self.win.refresh

    def handle_key_stroke(self):
        while True:
            ch = self.scr.getch()
            if ch == curses.KEY_DOWN:
                self.pos[0] += 1
            elif ch == curses.KEY_UP:
                self.pos[0] -= 1
            elif ch == curses.KEY_LEFT:
                self.pos[1] -= 1
            elif ch == curses.KEY_RIGHT:
                self.pos[1] += 1
            elif ch == ord('q'):
                break
            self.draw_window()

if __name__ == '__main__':
    curses.wrapper(MovingWindow)
```

2.2.3 Explications

Dans un premier temps, nous n'appelons plus la fonction `main`, mais nous initialisons un objet de la classe `MovingWindow`.

```
curses.wrapper(MovingWindow)
```

Nous créons des attributs, pour avoir la taille (facultatif), mais surtout la position courante de la fenêtre à afficher, ce qui correspond dans le `__init__` aux lignes suivantes :

```
self.pos = [3, 3]
self.size = [4, 6]
```

Les trois lignes suivantes ne sont que des appels à d'autre méthode de la classe.

On initialise quelques éléments de ncurses :

```
def init_curses_mode(self):
    # Toujours les couleurs transarante
    curses.use_default_colors()
    # On s'assure de ne rien afficher si on écrit
    curses.noecho()
    # On désactive le curseur
    curses.curs_set(0)
```

La méthode permettant d'afficher la fenêtre n'est pas bien plus compliqué.

```
def draw_window(self):
    # On efface ce qu'on avait
    self.scr.erase()
    # On créer une nouvelle fenêtre, avec la position et taille
    # indiqué par les attributs
    self.win = self.scr.subwin(self.size[0], self.size[1], self.pos[0], self.pos[1])
    # On remets une bordure
    self.win.border(0)
    # Enfin, on affiche le résultat
    self.win.refresh
```

La dernière méthode `handle_key_stroke` gère les touches, et son fonctionnement est plutôt simple, `curses.KEY_UP` par exemple désigne la touche du haut. Lorsqu'une des flèches est appuyé, on change les attributs de position en fonction. En fin de boucle, on affiche le résultat obtenu.

Il est a noter, la ligne suivante :

```
elif ch == ord('q'):
```

On devine facilement qu'il sert à quitter l'application, mais le `ord` est utile pour convertir la lettre en son équivalant numérique, car les touches saisis sont des chars.

On lance le programme, on joue un peu avec, la fenêtre ce déplace, on est content... jusqu'à ce que... la fenêtre sort de la console, en faisant planter le programme. Nous savons ce qu'il nous reste à faire alors, nous assuré que cette fenêtre ne sorte pas de la console.

2.2.4 Script au complet

```
import curses
import curses.wrapper

class MovingWindow(object):

    def __init__(self, scr):
        self.scr = scr
        self.pos = [3, 3]
```

```
        self.size = [4, 6]
        self.maxyx = []
        self.init_curses_mode()
        self.draw_window()
        self.handle_key_stroke()

    def init_curses_mode(self):
        curses.use_default_colors()
        curses.noecho()
        curses.curs_set(0)
        self.maxyx = self.scr.getmaxyx()

    def draw_window(self):
        self.scr.erase()
        self.win = self.scr.subwin(self.size[0], self.size[1],
                                   self.pos[0], self.pos[1])
        self.win.border(0)
        self.win.refresh

    def move_down(self):
        if self.pos[0] + self.size[0] < self.maxyx[0]:
            self.pos[0] += 1

    def move_up(self):
        if self.pos[0] > 0:
            self.pos[0] -= 1

    def move_left(self):
        if self.pos[1] > 0:
            self.pos[1] -= 1

    def move_right(self):
        if self.pos[1] + self.size[1] < self.maxyx[1]:
            self.pos[1] += 1

    def handle_key_stroke(self):
        while True:
            ch = self.scr.getch()
            if ch == curses.KEY_DOWN:
                self.move_down()
            elif ch == curses.KEY_UP:
                self.move_up()
            elif ch == curses.KEY_LEFT:
                self.move_left()
            elif ch == curses.KEY_RIGHT:
                self.move_right()
            elif ch == ord('q'):
                break
            self.draw_window()

if __name__ == '__main__':
```

```
    curses.wrapper(MovingWindow)
```

Il n'y a pas énormément de changement, et corresponde à la gestion de la taille maximal de l'écran. On remarque dans un premier temps, que j'en ai profiter pour créer autant de méthodes que de mouvement, permettant de gagner en lisibilité un peu.

La ligne suivante, va retourner la taille de la console dans une turple :

```
self.maxyx = self.scr.getmaxyx()
```

Tout le reste n'est qu'un petit peu de calcul et de logique pour s'assurer que la fenêtre ne sorte pas.

On pourrait très bien essayer quatre touches qui aurait pour effet d'agrandir la fenêtre par l'un des côtés, toujours en s'assurant de l'espace disponible.

2.3 Menu navigable

2.3.1 Ce que l'on veut

Deux scripts sont proposé dans ce chapitre, le second en est une extention.
- Un menu complet, dans lequel nous pouvons nous déplacer à l'aide de flèche, et sélectionner l'un des éléments. L'élément du menu sera mis en surbrillance en choisissant une couleur différente (ici, il sera rouge). Il faudra s'assurer qu'on ne dépasse pas les limites du menu.
- Au lieu d'afficher tout le menu, seul quelques éléments seront affiché, mais la liste doit être navigable entièrement, on s'attend donc à voir apparaitre les éléments qui n'était pas affiché tout en se déplacent dans la liste. La navigation serra naturel.

2.3.2 Menu entièrement affiché

```
import curses
import curses.wrapper

class Menu(object):

    menu = [
        'item 0', 'item 1', 'item 2', 'item 3',
        'item 4', 'item 5', 'item 6', 'item 7',
    ]

    def __init__(self, scr):
        self.scr = scr
        self.init_curses()
        self.item = { 'current': 0 }
        self.draw_menu()
        self.handle_keybinding()
```

```
    def init_curses(self):
        curses.curs_set(0)
        curses.use_default_colors()
        curses.init_pair(1, curses.COLOR_RED, -1)

    def draw_menu(self):
        i = 2
        for element in self.menu:
            if element == self.get_current_item():
                self.scr.addstr(i, 2, element, curses.color_pair(1))
            else:
                self.scr.addstr(i, 2, element)
            i += 1
        self.scr.refresh()

    def get_current_item(self):
        return self.menu[self.item['current']]

    def navigate_up(self):
        if self.item['current'] > 0:
            self.item['current'] -= 1

    def navigate_down(self):
        if self.item['current'] < len(self.menu) -1:
            self.item['current'] += 1

    def show_result(self):
        win = self.scr.subwin(5, 40, 2, 10)
        win.border(0)
        win.addstr(2, 2, 'Vous avez selectionné %s'
                   % self.menu[self.item['current']])
        win.refresh()
        win.getch()
        win.erase()

    def handle_keybinding(self):
        while True:
            ch = self.scr.getch()
            if ch == curses.KEY_UP:
                self.navigate_up()
            elif ch == curses.KEY_DOWN:
                self.navigate_down()
            elif ch == curses.KEY_RIGHT:
                self.show_result()
            elif ch == ord('q'):
                break
            self.draw_menu()

if __name__ == '__main__':
    curses.wrapper(Menu)
```

Si vous avez bien suivis le précédant chapitre, celui-ci doit aller tout seul. Puisqu'il n'y a pas beaucoup de nouveautés.

Le choix de *self.item = { 'current' : 0 }* est surtout du à la seconde partie qu'on va voir après, il n'est effectivement pas très justifié ici de prendre un dictionnaire pour un seul élément.

Ce que l'on souhaite surtout, c'est garder une indication, un pointeur, sur l'élément courrant du menu, pour savoir où on en est dans le menu. Il faut également s'assurer qu'on ait pas dépasser le début ou la fin du menu, dans tel cas il faut empècher d'aller plus loin.

2.3.3 Menu déroulant

```
import curses
import curses.wrapper

class Menu(object):

    menu = [
        'item 0', 'item 1', 'item 2', 'item 3',
        'item 4', 'item 5', 'item 6', 'item 7',
    ]

    def __init__(self, scr):
        self.scr = scr
        self.init_curses()
        self.item = {
                'current': 0,
                'first':   0,
                'show':    5,
              }
        self.draw_menu()
        self.handle_keybinding()

    def init_curses(self):
        curses.curs_set(0)
        curses.use_default_colors()
        curses.init_pair(1, curses.COLOR_RED, -1)

    def draw_menu(self):

        first = self.item['first']
        last = self.item['first'] + self.item['show']
        menu = self.menu[first:last]

        i = 2
        for element in menu:
            if element == self.get_current_item():
                self.scr.addstr(i, 2, element, curses.color_pair(1))
            else:
                self.scr.addstr(i, 2, element)
```

```
            i += 1
        self.scr.refresh()

    def get_current_item(self):
        return self.menu[self.item['current']]

    def navigate_up(self):
        if self.item['current'] > 0:
            self.item['current'] -= 1
            if self.item['current'] < self.item['first']:
                self.item['first'] -= 1

    def navigate_down(self):
        if self.item['current'] < len(self.menu) -1:
            self.item['current'] += 1
            if self.item['current'] >= self.item['show'] + self.item['first']:
                self.item['first'] += 1

    def show_result(self):
        win = self.scr.subwin(5, 40, 2, 10)
        win.border(0)
        win.addstr(2, 2, 'Vous avez selectionné %s'
                   % self.menu[self.item['current']])
        win.refresh()
        win.getch()
        win.erase()

    def handle_keybinding(self):
        while True:
            ch = self.scr.getch()
            if ch == curses.KEY_UP:
                self.navigate_up()
            elif ch == curses.KEY_DOWN:
                self.navigate_down()
            elif ch == curses.KEY_RIGHT:
                self.show_result()
            elif ch == ord('q'):
                break
            self.draw_menu()

if __name__ == '__main__':
    curses.wrapper(Menu)
```

La particularité est surtout de ne pas afficher tout le menu, il est ainsi découpé avant d'être affiché, ce n'était pas la seul possibilité d'aborder le problème.

```
first = self.item['first']
last = self.item['first'] + self.item['show']
menu = self.menu[first:last]
```

L'autre point à faire attention, c'est de bien s'assurer que les méthodes de navigation fassent

ce qu'on attend. Par commodité, j'ai choisis un petit dictionnaire `item` pour repérer certains emplacement clé du menu, comme le nombre d'élément affiché (show).

2.4 Visualiser et scroller un fichier

On va essayer de reproduire sommairement le comportement de la commande **less**, qui permet de naviguer en scrollant dans un fichier, et de quitter, même si la commande permet plus que ça, c'est du moins la fonctionnalité qu'on cherche ici à reproduire. En bonus, nous afficherons le numéro de ligne.

L'utilité de ce script en est réduit à l'intérêt d'apprendre à le faire, ni plus, ni moins. Le fichier qu'on cherche à visualisé sera renseigné en ligne de commande, préparez un fichier de quelques centaines de lignes qui servira de fichier test.

Les touches sont : 'j' et 'k' pour ce déplacer dans le fichier, et 'q' pour quitter.

Note : J'ai utilisé ici **Python 3** pour écrire le script, bien que ça ne fasse que peu de différence, il est utile de changer quelques détails, pour Python2. Il est cependant intéressant ici de diversifier un peu les exemples.

2.4.1 Idée générale

Nous allons utiliser deux fenêtre, ou cadre pour faire cela.
- *scr* : la console, utilisé de façon classique comme dans les derniers billet, mais qui n'aura que peu d'utilité ici.
- *pad* : une fenêtre de taille supérieur à la console, contenant la totalité du fichier texte à afficher et scroller.

Pour comprendre le fonctionnement du `pad`, imaginons que j'ai un petit cadre (la console) derrière lequel je place une grande feuille (pad) bien plus grand que le cadre, si je déplace la feuille, tout en ne regardant que le cadre, j'aurai l'impression que le contenu dans le cadre est *scrollé*, et c'est un peu ce comportement qui est reproduit ici. Le texte reste fixe par rapport au *pad*, mais l'impression lorsqu'on bouge le *pad* par rapport à la console (window), c'est d'avoir un texte qui défile.

Voici le script entier dans un premiers temps, sur lequel j'ai placé des commentaires tout du long pour la compréhension, je reviens tout de même sur le *pad* à la suite de ce billet.

2.4.2 Le script complet

```
import sys
import curses
import curses.wrapper

class Scroll(object):
```

```
    def __init__(self, scr, filename):
        self.scr = scr
        self.filename = filename
        # On garde en mémoire la première ligne à affiché
        self.first_line = 0

        self.run_scroller()

    def run_scroller(self):
        self.init_curses()
        self.init_file()
        self.init_pad()
        self.key_handler()

    def init_curses(self):
        '''Quelques banalités pour le fonctionnement de curses
        Voir les billets précédent s'il y a des doutes dessus
        '''
        curses.use_default_colors()
        curses.curs_set(0)
        self.maxyx = self.scr.getmaxyx()

    def init_file(self):
        '''On ouvre le fichie, en gardant son contenu, sous forme de tableau
        avec une ligne par entrée, on compte également le nomble de ligne
        afin de savoir la hauteur utile pour le "pad"
        '''
        f = open(self.filename, 'r')
        self.content = f.readlines()
        self.count = len(self.content)+1
        f.close()

    def init_pad(self):
        '''On créer le pad, dans lequel on affiche ligne par ligne le contenu
        du fichier, avec son numéro de ligne. Le pad est finalement affiché
        avec l'appel à la méthode refresh_pad qu'on va voir en dessous.
        '''
        self.pad = curses.newpad(self.count, self.maxyx[1])

        for i, line in enumerate(self.content):
            self.pad.addstr(i, 0, '{0:3} {1}'.format(i+1, line))

        self.refresh_pad()

    def refresh_pad(self):
        '''Le plus gros du concept est ici, voir le billet pour plus
        d'explication
        '''
        self.pad.refresh(self.first_line, 0, 0, 0,
                self.maxyx[0]-1, self.maxyx[1]-1)
```

```
    def key_handler(self):
        '''Une boucle classique, afin d'interpréter les touches'''
        while True:
            ch = self.pad.getch()
            if ch == ord('j'):
                self.scroll_down()
            elif ch == ord('k'):
                self.scroll_up()
            elif ch == ord('q'):
                break
            self.refresh_pad()

    def scroll_down(self):
        '''On scroll, tout en s'assurant de l'affichage restant'''
        if self.maxyx[0] + self.first_line < self.count:
            self.first_line += 1

    def scroll_up(self):
        '''On scroll, en s'assurant qu'on est pas déjà en début
        de fichier.'''
        if self.first_line > 0:
            self.first_line -= 1

if __name__ == '__main__':

    try:
        # On essaie de lire l'argument fournis en console
        # correspondant au nom du fichier
        filename = sys.argv[1]
    except IndexError as e:
        # Si aucun argument est trouvé, on affiche l'usage
        print('Erreur: {}'.format(e))
        print('Usage:  {} filename'.format(sys.argv[0]))
        sys.exit(0)
    # On appelle la classe avec le wrapper curses.
    curses.wrapper(Scroll, filename)
```

2.4.3 Quelques informations supplémentaires

Comme promis, je reviens sur la méthode suivante :

```
def refresh_pad(self):
    self.pad.refresh(self.first_line, 0, 0, 0,
            self.maxyx[0]-1, self.maxyx[1]-1)
```

Pour la signification des arguments, en les prenant deux par deux :

– coordonnées y, x de départ sur le pad. Pour reprendre l'exemple plus haut de la feuille, ces coordonnées précise à quel point je vais commencer à afficher le contenu. Si j'ai par exemple 2, 10, ça signifie que je commence à afficher à la ligne 2 et à partir du 10 ème caractère, Je

ne sais pas encore où je vais afficher, ni quel quantité, mais je connais le commencement de mon contenu. Dans l'exemple, on veux bien sûr afficher dès le 1er caractère et commencer à la première ligne, pour scroller, il suffira d'incrémenter la première ligne.
– coordonnées y, x de départ sur la console. Puisque je n'affiche qu'une partie, je décide donc de commencer l'affichage tout en haut à gauche de ma console, c'est à dire 0, 0.
– coordonnées y, x de fin sur la console. Pour avoir notre cadre, il faut le début (voir ci-dessus) et la fin, correspondant au coin inférieur droit de la console, obtenu grâce à la fonction `scr.getmaxyx()`.

Je pense qu'avec les explications, le système de scroll est plus clair, on peut noter que le script est assez simplifié, qu'il manque par exemple la gestion de redimension de la console.

2.5 Les scripts de démo fournis par Python

Les sources de Python sont fournis avec une poignée de scripts pouvant servir à apprendre quelques bases. Dans ce billet, nous allons voir ceux prévu pour le module curses de Python2.7. Le choix de Python2.7, et non pas 3.2, est du au fait qu'il y a plus d'exemples pour curses dans celui-ci.

Sans plus attendre, on prend les sources de Python, avec quelques commandes comme suit :

```
$ wget http://www.python.org/ftp/python/2.7.2/Python-2.7.2.tar.bz2
$ tar xf Python-2.7.2.tar.bz2
$ cd Python-2.7.2/Demo/curses
$ ls
life.py*  ncurses.py  rain.py  README  repeat.py*  tclock.py  xmas.py
```

Ok, on voit quelques scripts disponible dont la curiosité nous pousse à les essayer tous.

Petite précision, si vous prenez Python3.2, les scripts ne se trouvent plus au même endroit, mais dans *Python-3.2/Tools/demo/*, et on voit qu'il y en a nettement moins, sauf s'ils sont cachés ailleurs.

Jetons un œil sur le contenu de chacun.

2.5.1 README

Fournis un petit récapitulatif, moins complet que dans ce billet cela dit, et quelques crédits.

2.5.2 ncurses.py

– 273 lignes

Un bon exemple pour montrer l'intérêt du module *curses.panel*, qui est une forme de fenêtre un peu particulière, car elle permet de gérer le super-positionnement les unes par rapport aux autres. Le programme positionne simplement des panels, et passe à un positionnement prédéfini à chaque frappe de touche.

Une petite remarque sur le script, on y voit deux constantes. Cependant, elles ne figurent pas sur la documentation, ce qui m'a plutôt surpris.
– ncurses.COLS
– ncurses.LINES

2.5.3 rain.py

– 93 lignes

Un script un peu plus simple dont le but est d'afficher des dessins asciiart, dont la suite représente une goutte s'éclatant. Idéal pour les premiers pas avec Python et ncurses.

2.5.4 tclock.py

– 148 lignes

De conception plus avancé, ce script reproduit une horloge à aiguille, avec même une trotteuse représenté par le point rouge. Assez amusant.

2.5.5 xmas.py

– 906 lignes

Ce script correspond à une animation d'asciiart, d'où le nombre important de ligne, il est amusant de le regarder, mais n'apporte peut être pas énormément pour l'usage ncurses je crois.

2.5.6 life.py

– 216 lignes

Une jolie adaptation du jeu de la vie de Conway en 200 lignes, un grand classique de la programmation et de la communauté de hacker, dont j'en ai fait une présentation sur [ce billet][2]. Ce script peut intéresser pour le côté curses, mais également pour lire une implémentation simple de ce jeu.

2.5.7 repeat.py

– 58 lignes

Le but est de passer en argument une commande, dont l'affichage sera rafraîchis toute les secondes. On peut essayer par exemple :

```
repeat.py uptime
```

Voilà un petit tour d'horizon des quelques programmes de démonstration que fournis Python, ils peuvent être utile pour se faire la main. Le but du billet n'était pas d'en faire une description détaillé, mais surtout de motiver à aller voir de plus près le code. À vos éditeurs !

CHAPITRE 3

Utilisation d'API

3.1 Générateur de documentation Sphinx

Lorsqu'on utilise un projet ou une bibliothèque, il est souvent agréable de générer la documentation localement, plusieurs raisons à ça.
– Temps de réaction aux chargement de pages plus rapide.
– Pas besoin de connexion Internet pour consulter la documentation, pouvant être utile à certain (train, déplacement, Internet coupé par Hadœpi...)
– Pas de « mince le site est down... »
– Pour les petit projet, la documentation sur le site ne correspond pas forcément à l'actuel car le développeur ne prend pas forcément le temps de mettre à jour. Et je pense qu'on est plus d'un à avoir le reflex de comparer le numéro de version entre le site et l'api réellement utilisé.

Par chance, les projets basé sur Python utilises presque tous un même outil pour générer la documentation, Sphinx [1]. Consistant à parser des fichiers en ReStructuredText (extention .rst). Spinx étant lui même écrit en Python, et s'appuyant sur pigments pour la coloration syntaxique.

Pour Arch Linux, Sphinx est disponible dans [community], il est également disponible pour Ubuntu et Debian, et très certainement pour toutes autres distributions tant ce projet est un classique. Il est par ailleurs utilisé pour générer la documentation en ligne officielle de Python.

```
pacman -S python-sphinx
apt-get install python-sphinx
```

Le fonctionnement de sphinx est simple, un utilitaire règle les détails en écrivant un fichier `Makefile`, pour lequel il suffira d'utiliser avec un make. Le fichier `Makefile` peut être générer avec la commande `sphinx-quickstart` par exemple.

Par exemple, prenons Lettuce comme projet, un projet pour les tests unitaires, qui sera vu par la suite. C'est un projet permettant de gérer les tests, qui sera vu un peu plus en détail dans la partie tests unitaires.

```
$ git clone git://github.com/gabrielfalcao/lettuce.git
$ cd lettuce/docs
```

1. http ://sphinx.pocoo.org/

```
$ make html
rm -rf _build/*
sphinx-build -b html -d _build/doctrees   . _build/html
Making output directory...
Running Sphinx v1.0.7
loading pickled environment... not yet created
building [html]: targets for 17 source files that are out of date
updating environment: 17 added, 0 changed, 0 removed
reading sources... [100%] tutorial/tables
looking for now-outdated files... none found
pickling environment... done
checking consistency... /home/nicolas/test/lettuce/docs/tutorial/django.rst::
WARNING: document isn't included in any toctree
done
preparing documents... done
writing output... [100%] tutorial/tables
writing additional files... genindex search
copying images... [100%] tutorial/flow.png
copying static files... done
dumping search index... done
dumping object inventory... done
build succeeded, 1 warning.

Build finished. The HTML pages are in _build/html.
```

Comme indiqué, le résultat se trouve dans `_build/html` dont il suffit de lancer `index.html`.

Il est donc possible de générer rapidement la documentation avec un simple :

```
make html
```

D'autre format sont supporté, notamment le PDF. Cette opération passe par la génération d'un document LaTeX.

```
make latex
make latexpdf
```

Pour anecdote, le présent document est généré en utilisant Sphinx et le make latexpdf.

Pour générer le fichier Makefile, Sphinx fournit un utilitaire en ligne de commande, permettant, une fois quelques questions basique répondu, de générer le Makefile. Heureusement les questions sont assez simple, et les valeurs par défaut sont souvent suffisante.

```
sphinx-quickstart
```

Ce billet est une très courte introduction de ce que Sphinx peut faire. Cet outil vaut à être connu.

3.2 Client Twitter et Identi.ca

On va voir ici, comment envoyer et lire des tweets depuis la console vers twitter ou Identi.ca, en passant par le système Oauth pour les deux cas. Les étapes sont expliqué du début à la fin, et ne requière pas spécialement de connaissances importantes en Python.

Pour arriver à nos fins, nous allons utiliser une API, python-twitter. Ce billet est plus une introduction à cette API que d'avoir un intérêt réel dans l'application. Surtout dans le cas de identi.ca où il serait bien plus simple d'utiliser une authentification http basique au lieu du système Oauth. Il reste cependant intéressant de voir que Identi.ca propose un comportement similaire à Twitter dans son API, permettant d'intégrer deux services avec un quasiment le même code, évitant une réecriture pour s'adapter à l'un ou l'autre des services.

3.2.1 Enregistrer son application

Pour utiliser Oauth, il faut enregistrer son application, aussi bien pour Twitter que pour Identi.ca. Rien de bien méchant mais indispensable pour obtenir *les clés* de votre application.

Twitter

Une fois identifié, on se rend sur la page de gestion d'application : https ://dev.twitter.com/apps
On click sur « Register a new app », possible qu'il soit écrit en français chez vous.

Je vous épargne le screenshot, nous remplissont le formulaire suivant :
– Application name : Un nom d'application
– Description : Faut vraiment expliqué ? :p
– Application website : C'est le lien qui vient lorsqu'on a la source d'un client. Pas super utile si c'est juste un client fait à La Rache mais bon.
– Application type : Important, faut choisir « client ».
– Default access type : C'est ce que twitter à changé récemment, il faut au minimum « read & write », mais le « read, write & direct message » peut être utile pour avoir accès au messages direct.
– Application icon : bah... un avatar n'a jamais tué personne.
On remplit le captcha, on valide leurs conditions d'utilisation, et nous voilà avec une application fraîchement crée. Vous obtenez quelques renseignements et clés, un peu obscure au premier abord, vous ne relevez uniquement les deux informations suivantes :
– Consumer key (Clé de l'utilisateur)
– Consumer secret (Secret de l'utilisateur)

Identi.ca

Le principe est similaire, on s'identifie à son compte Identi.ca, puis on enregistre son application sur le lien suivant : http ://identi.ca/settings/oauthapps

On click sur *Register a new application*, puis on remplit également le petit formulaire, qui est similaire à celui de twitter, pour lequel je ne reprendrai que les détails qui peuvent laisser un doute :

– Callback url : Laissez vide,
– Navigateur/Bureau : Vous choisissez Bureau.
– Lecture-écriture : vous souhaitez pouvoir écrire.
Une fois validé, il vous affiche une liste de vos applications enregistré, clickez sur le nom de celle que vous venez de valider pour obtenir les éléments recherché.
– Clé de l'utilisateur
– Secret de l'utilisateur

3.2.2 Authentification du client

Dans la première étape, nous avons obtenu les clés spécifique à l'application, maintenant, il faut obtenir les clés pour un utilisateur, lié à cette application, et l'autoriser à accéder à son compte.

Comme nous voulons aller au plus simple, et qu'il est pratique courante de réutiliser un code de quelqu'un d'autre lorsque celui-ci est libre. On utilise l'utilitaire fournis avec l'API python-twitter.

On télécharge la version de python-twitter (0.8.2, la dernière), on décompresse l'archive, et rentre dans le répertoire :

```
wget http://python-twitter.googlecode.com/files/python-twitter-0.8.2.tar.gz
tar xvf python-twitter-0.8.2.tar.gz
cd python-twitter-0.8.2
```

Le fichier important à ce niveau est `get_access_token.py` puisqu'il permet d'obtenir les clés. Je ne copie-colle pas tout le code, mais vous devez rechercher les deux paramètres suivant à la ligne 34 et 35 :

```
consumer_key    = None
consumer_secret = None
```

En le remplacant par les valeurs obtenu plus haut, par exemple :

```
consumer_key    = 'vIxy85s7r2jOBmr7m7bQ'
consumer_secret = 'PyLzYa3WLMqv6xziFAiOqMlQlSxP9vXyXsTemqyB7c'
```

L'application vous donne un lien à visiter, un code à rentrer, puis vous fournit vos deux codes.

```
$ python2.7 get_access_token.py
Requesting temp token from Twitter

Please visit this Twitter page and retrieve the pincode to be used
in the next step to obtaining an Authentication Token:

https://api.twitter.com/oauth/authorize?oauth_token=c5X[...]31Y

Pincode? 4242424

Generating and signing request for an access token
```

```
Your Twitter Access Token key: 1232[...]qH0y43
          Access Token secret: HHzs[...]IyoJ
```

Un jeu d'enfant, vous notez vos token obtenu.

Identi.ca

Pour adapter à Identi.ca, il n'y a besoin que de peu de changement, l'idée et de remplacer `https://api.twitter.com/` par `https://identi.ca/api/` ou par toute autre URL répondant à vos besoins.

hint : `:%s/api.twitter.com/identi.ca\/api/g`

Attention Il y a une différence notable entre les deux, sur le `REQUEST_TOKEN_URL`, il faut rajouter à la fin `?oauth_callback=oob`. Je ne suis plus certain d'où vient cette information, mais il ne fonctionnera pas sans, alors que c'est sans problème pour Twitter.

Pour obtenir :

```
REQUEST_TOKEN_URL =
    'https://identi.ca/api/oauth/request_token?oauth_callback=oob'
ACCESS_TOKEN_URL  = 'https://identi.ca/api/oauth/access_token'
AUTHORIZATION_URL = 'https://identi.ca/api/oauth/authorize'
SIGNIN_URL        = 'https://identi.ca/api/oauth/authenticate'
```

N'oubliez pas de remplir `consumer_key` et `consumer_secret` comme pour Twitter. La démarche y est similaire.

3.2.3 L'application

La partie fastidieuse étant passé, les clés pour le bon déroulement obtenu, place enfin à ce qui nous intéresse, le code !

On veut quoi ?
- Appeller mon application, avec un argument, on va mettre -s comme send, puis le tweet à envoyer directement dans la ligne de commande.
- L'application doit être correctement authentifier au service voulu.
- Utiliser python-twitter comme API pour gérer l'authentification et l'envoi.
- Un nom pour l'application, va pour... Clitter... Bon un peu pourris, mais je vais faire avec :þ (Clitter pour CLI, Commande Line Interface, je préfère préciser, on sait jamais)

Dans un premier temps, on veut passer correctement l'argument -s. J'utilise dans l'exemple getopt, qui est un parser ressemblant à celui de C, mais on peut utiliser également argparse, qui est moins verbeux à mettre en place.

Je reprends un peu l'exemple pris sur la page de documentation de getopt [2], on fait un premier essai, voir si l'argument est bien pris en compte.

clitter.py :

2. http ://docs.python.org/library/getopt.html

```
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "s:", ["send="])
    except getopt.GetoptError, err:
        print str(err) # will print something like "option -a not recognized"
        sys.exit(2)
    for o, a in opts:
        if o in ("-s", "--send"):
            tweet = a
            print a
        else:
            assert False, "unhandled option"

if __name__ == "__main__":
    main()
```

Ce qu'on veut ici, c'est uniquement avoir en sortie le tweet entrée, on essaye :

```
python2.7 clitter.py -s 'mon tout premier test'
```

On a ce quoi l'on s'attendait en retour en console, le « mon tout premier test », très bien, passons aux choses sérieuses.

Il nous reste plus qu'a rajouter l'authentification, et l'envoi sur twitter, ce n'est pas bien compliqué maintenant.

On va créer un répertoire dans lequel on place : - `__init__.py` avec rien dedans - `clitter.py` correspondant à notre script - `twitter.py` l'api qu'on a téléchargé tout à l'heure, là où se trouvait

get_access_token.py, il nous suffit de le déplacer dans le répertoire.

Voici le code permettant d'envoyer un tweet :

```
import getopt, sys
import twitter

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "s:", ["send="])
    except getopt.GetoptError, err:
        print str(err) # will print something like "option -a not recognized"
        sys.exit(2)
    for o, a in opts:
        if o in ("-s", "--send"):
            tweet = a
        else:
            assert False, "unhandled option"

    consumer_key = 'vIxy85s[...]7m7bQ'
    consumer_secret = 'PyLzYa[...]9vXyXsTemqyB7c'
```

```
    oauth_token = 'zCNVC[...]hxgI5m5'
    oauth_token_secret = '3Q4tL3U[...]FyHvWCh'

    api = twitter.Api(consumer_key, consumer_secret,
                      oauth_token, oauth_token_secret)

    api.PostUpdate(tweet)

if __name__ == "__main__":
    main()
```

La première chose, c'est d'importer le module twitter :

```
import twitter
```

Puis on renseigne les quatres clées pour s'identifier, que j'ai un peu raccourcis ici.

Note : Lorsqu'on distribue une application, on fournit la clé *consumer_secret*, c'est un peu déroutant au début, on se dit, hmm, si c'est secret, pourquoi le diffuser, bref, peut-être que le nom est trompeur, mais par exemple, pour le programme *hotot*, elles sont dans le fichier *config.py* à la ligne 76/77, et on retrouve bien également la *consumer_secret*, c'est normal, rien d'inquiétant.

Vient ensuite l'authentification proprement dite, avec un double rôle, instancier l'objet de l'api, et s'identifier à celle-ci :

```
api = twitter.Api(consumer_key, consumer_secret,
                  oauth_token, oauth_token_secret)
```

Enfin, on envoie le tweet grâce à cette api tout simplement :

```
api.PostUpdate(tweet)
```

Et c'est aussi simple que ça !

Identi.ca

Comme je disais en introduction, l'intérêt d'utiliser Oauth et python-twitter pour identi.ca, c'est de pouvoir utiliser un même code sans quasiment rien changer. La seul différence sera d'indiqué l'URL lors de l'authetification, comme dans cette exemple :

```
url = "https://identi.ca/api"
api = twitter.Api(consumer_key, consumer_secret,
                  oauth_token, oauth_token_secret,
                  base_url=url)
```

Et c'est tout, pour envoyer un tweet, le code est exactement le même.

Bonus, lire ses tweets

Comme il est assez facile d'envoyer un tweet, qu'on a finalement pas vu grand chose de python-twitter, compliquons un peu les choses, en essayant de lire ses tweets en console. Une bonne occasion de réorganiser un peu le code dans la foulée.

La méthode utilisé pour lire ses tweets est :

```
api.GetFriendsTimeline(retweets=True)
```

On récupère la timeline de nos « amis », et on active les retweets. On veut appeler cette fonction lorsqu'on utilisera l'option `-r` pour read, en console.

Voici le code final de notre petite application (à adapté comme vu plus haut pour identi.ca)

```
import getopt, sys
import twitter

def authentication():

    consumer_key = '2jO[...]Bm'
    consumer_secret = 'PyLzYa[...]3WLM'
    oauth_token = 'tsr[...]ds5'
    oauth_token_secret = 'PSd3[...]tSt'

    return twitter.Api(consumer_key, consumer_secret,
                       oauth_token, oauth_token_secret)

def send(tweet):
    api = authentication()
    api.postUpdate(tweet)

def read():
    api = authentication()
    statuses = api.GetFriendsTimeline(retweets=True)

    for status in statuses:
        print "%s:\t%s" % (status.user.screen_name, status.text)

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "rs:", ["send="])
    except getopt.GetoptError, err:
        print str(err) # will print something like "option -a not recognized"
        sys.exit(2)
    for o, a in opts:
        if o in ("-s", "--send"):
            tweet = a
            send(tweet)
        elif o in ("-r", "--read"):
            read()
        else:
```

```
            assert False, "unhandled option"

if __name__ == "__main__":
    main()
```

Pas besoin de plus de code que ça ! Quelques explications encore. On retrouve nos tweets avec le *GetFriendsTimeline*, auquel on rajoute les retweets dans la timeline, sinon vous ne verriez pas les retweets de vos follower, ce qui est généralement le comportement par défaut des applications twitter.

```
statuses = api.GetFriendsTimeline(retweets=True)
```

On obtient une liste, sur lequel on va lire le pseudo et le tweet en accédant directement au attribut de l'object Status.

```
for status in statuses:
    print "%s:\t%s" % (status.user.screen_name, status.text)
```

Et on appele le script :

```
python2.7 clitter.py -r
```

Et voilà !

3.2.4 Conclusion

Grâce à ce script d'une cinquantaine de lignes tout mouillé, vous pouvez lire et envoyer des tweets directement en console. C'est l'avantage de s'appuyer sur une bibliothèque, même si au début cela peut faire peur de s'y plonger. Il ne faut pas hésiter à ouvrir le code, lire la documentation tout au long de l'API pour en apprendre plus.

3.3 Api goo.gl et Oauth

Désirant essayer un peu le service de Google pour raccourcir les url goo.gl, j'ai voulu l'intégrer à mon petit client Twitter Tyrs [3], et surtout intégrer l'identification Oauth. Nous allons voir ici comment utiliser l'API pour raccourcir un lien, et la documentation pour aller plus loin.

Pas de débats sur Google, mais simplement une bonne occasion d'essayer un nouveau jouet et de se plonger un peu dans du code.

3. http ://tyrs.nicosphere.net Tyrs est un client twitter en console se basant sur ncurses et sous licence GPL. J'en ai commencer l'écriture début mai (2011), il est fonctionnel et je m'en sers encore au quotidien.

3.3.1 Enregistrer son application

Comme toujours avec Oauth, il faut dans un premier temps enregistrer votre application, en allant sur le lien suivant [4], comme l'est expliqué ici [5]. Vous remplissez les quelques cases, attention, il faut choisir ici *Installed application*, correspondant à un usage Desktop, et non un service sur un site web. Vous devez également activer le support pour l'urlshortener, comme le montre l'image suivante, disponible toujours au même endroit :

Note : Pour suivre ce tutoriel, vous n'êtes pas obligé d'enregistrer une application. Si vous le faite, vous avez obtenu un classique couple de clée client id / client secret. Un client secret, qui comme je l'expliquais dans le chapitre précédant, n'a rien de secret du tout, et peut être partagé par tous. L'exemple serra pris avec les clés préparé et fonctionnant, vous pouvez donc les utiliser.

3.3.2 Les dépendances

Une seule dépendance est requise, mais une autre est optionnel on verra par la suite pourquoi.
– **google-api-python-client**
Cette bibliothèque est disponible à plusieurs endroits, dont le dépôt officiel [6] et sur pypi [7], service connu des pythonistes.

Il peut être utile de regarder dans les dépôts de votre distribution avant, pour Arch Linux, j'ai regardé, et il est disponible avec un :

```
yaourt -S python2-google-api-python-client
```

– **python-gflags**

Pas utile si vous suivez _strictement_ le tuto, mais si vous prenez l'autre code exemple que je vais fournir en lien, vous en aurez besoin. La page du projet [8], et très certainement disponible sur pypi également.

Pour Arch Linux :

```
yaourt -S python-gflags
```

4. https ://code.google.com/apis/console/
5. http ://code.google.com/apis/accounts/docs/OAuth2.html#Registering
6. http ://code.google.com/p/google-api-python-client/
7. http ://pypi.python.org/pypi/google-api-python-client/1.0beta2
8. http ://code.google.com/p/python-gflags/

3.3.3 Le code

Le code suivant, est une version simplifié du code exemple pouvant être trouvé sur le lien suivant [9], on peut noté qu'on trouve également des exemples pour d'autres langages tel que Ruby, Java, PHP, .NET...

La version que je fournis plus bas, basé sur leur exemple est allégé du système de logging et de leur système de commande, qui nécessite également la dépendance *python-gflags* qui n'apporte pas un plus ici (surtout que leur lien à raccourcir est écrit en dure dans le code)

L'usage voulu ici est par exemple `./googl.py http://www.nicosphere.net`, et ainsi obtenir en retour le lien raccourcis.

```
import httplib2
import sys

from apiclient.discovery import build
from oauth2client.file import Storage
from oauth2client.client import AccessTokenRefreshError
from oauth2client.client import OAuth2WebServerFlow
from oauth2client.tools import run

FLOW = OAuth2WebServerFlow(
    client_id='382344260739.apps.googleusercontent.com',
    client_secret='fJwAFxKWyW4rBmzzm6V3TVsZ',
    scope='https://www.googleapis.com/auth/urlshortener',
    user_agent='urlshortener-tyrs/1.0')

def main(argv):

  # On essaye d'obtenir le lien fournis en ligne de commande
  try:
    long_url = argv[1]
  except IndexError:
    print 'Il faut fournir une URL'
    return

  # Une fonctionnalité de l'API est associé au fichier googl.tok
  storage = Storage('googl.tok')
  credentials = storage.get()
  # Si le fichier n'existe pas, ou n'est pas valide, on demande une
  # authorisation, le fonctionnement est directement dans l'API de google.
  if credentials is None or credentials.invalid:
    credentials = run(FLOW, storage)

  # La requete http est préparé.
  http = httplib2.Http()
  http = credentials.authorize(http)
  service = build("urlshortener", "v1", http=http)
```

9. http ://code.google.com/p/google-api-python-client/source/browse/samples/urlshortener/urlshortener.py

```
  try:

    url = service.url()

    body = {"longUrl": long_url }
    # On envoie le tout
    resp = url.insert(body=body).execute()

    short_url = resp['id']
    # On imprime le résultat
    print short_url

  except AccessTokenRefreshError:
    print ("The credentials have been revoked or expired, please re-run"
      "the application to re-authorize")

if __name__ == '__main__':
  main(sys.argv)
```

Warning : Python 2.x est utilisé ici, et ne fonctionnera pas avec Python3

3.3.4 Authentification à votre compte

À la première utilisation, Il vous serra demandé de suivre un lien, afin d'autoriser l'application à s'associer avec votre compte, vous devez être identifié à votre compte Google. Exactement comme lorsque vous utilisez un nouveau client Twitter avec un compte, sauf que ici, pas besoin de PIN code de validation.

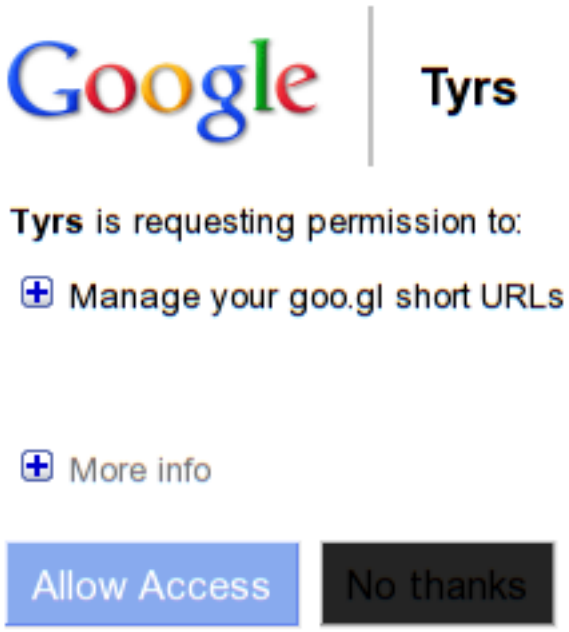
Pour que l'application soit utilisable une fois sur l'autre sans avoir à valider, les accès sont enregistré dans un fichier, à côté de votre executable, dans cet exemple : *googl.tok*

La sortie console ressemble à celà :

```
$ python2.7 googl.py http://www.nicosphere.net
Go to the following link in your browser:
https://accounts.google.com/o/oauth2/auth?scope=https%3A%2F%2Fwww.googleapis.com%2Fauth%2Furlsho

You have successfully authenticated.
```

Du côté validation dans le navigateur web, cela ressemble à ça :



Note : Les clées sont celles générées pour Tyrs [1], c'est pourquoi il apparaît dans le screenshot, si vous avez suivis la première étape, vous devez avoir le nom de votre application apparaître.

Finallement, vous devez voir dans la console le lien raccourcis correspondant, si l'opération est renouvelé il n'y a pas besoin de s'authentifier comme la première fois.

3.3.5 Conclusion

Ici on ne fait que retourner le lien racourcis de Google, l'API permet d'en faire bien plus encore, pour cela, il faut consulter le guide de démarrage [10] ou la référence de l'API [11]. Mais ce petit morceau de code permet d'avoir une idée, et montre que l'utilisation Oauth / API Google est plus accessible qu'on pourrait le croire au première abord.

3.4 Request, HTTP pour humains

Urllib est le module par défaut pour gérer les requetes HTTP avec Python. Le cookbook [12] du site officiel donne un petit aperçu de l'usage d'urllib2. Certains détails peuvent surprendre aux premières utilisations. Ce qu'on aime avec Python, c'est avant tout de pouvoir penser à l'application, plutôt qu'au code lui même.

Requests [13] se veux être un petit module répondant à cette problématique, « HTTP for humains » comme titre le site. On va voir quelques exemples, qui seront certainement plus marquant lorsque les besoins se complexifie.

10. http ://code.google.com/apis/urlshortener/v1/getting_started.html
11. http ://code.google.com/apis/urlshortener/v1/reference.html
12. http ://docs.python.org/howto/urllib2.html
13. http ://docs.python-requests.org/en/latest/index.html

Warning : Requests n'existe pour le moment que pour Python 2.x, il n'a pas encore été porté pour la version 3. C'est donc un choix à faire.

L'installation est on ne peut plus classique avec pip ou autre installeur de Python.

```
sudo pip install requests
```

3.4.1 Lire une page

Le premier exemple du how-to officiel est :

```
import urllib2

req = urllib2.Request('http://www.voidspace.org.uk')
response = urllib2.urlopen(req)
page = response.read()
```

Il pourrait être écrit avec Requests comme suit :

```
import requests

req = requests.get('http://www.nicosphere.net')
page = req.content
```

3.4.2 Remplir un formulaire

Un autre exemple tiré du site officiel, dont le but ici est de remplir un formulaire.

```
import urllib
import urllib2

url = 'http://www.someserver.com/cgi-bin/register.cgi'
values = {'name' : 'Michael Foord',
          'location' : 'Northampton',
          'language' : 'Python' }

data = urllib.urlencode(values)
req = urllib2.Request(url, data)
response = urllib2.urlopen(req)
page = response.read()
```

Et l'équivalant avec Requests :

```
import requests

url= 'http://wathever.url'
values = {'name': 'Nicolas',
          'location': 'Somewhere',
```

```
          'language': 'Python'}

req = requests.post(url, values)
page = req.content
```

Quelques remarques qu'on peut déjà faire.
– Pas besoin d'importer urllib _et_ urllib2.
– Pas besoin d'encoder les données avant, requests le fait pour vous.
– On reconnait get, post, mais d'autre méthode *RESTful* sont disponible tel que put et delete.
– Moins à écrire, et un code plus instinctif.

3.4.3 Authentification basique

Le dernier exemple, gérant l'authentification :

```
# create a password manager
password_mgr = urllib2.HTTPPasswordMgrWithDefaultRealm()

# Add the username and password.
# If we knew the realm, we could use it instead of None.
top_level_url = "http://example.com/foo/"
password_mgr.add_password(None, top_level_url, username, password)

handler = urllib2.HTTPBasicAuthHandler(password_mgr)

# create "opener" (OpenerDirector instance)
opener = urllib2.build_opener(handler)

# use the opener to fetch a URL
opener.open(a_url)

# Install the opener.
# Now all calls to urllib2.urlopen use our opener.
urllib2.install_opener(opener)
```

Un peu verbeux pour une simple authentification basique... Mais c'est l'exemple officiel encore une fois, j'ai laissé les commentaires du code pour plus de « lisibilité ». Maintenant, ce même code écrit avec Requests, exemple pris sur leur site.

```
>>> r = requests.get('https://api.github.com', auth=('user', 'pass'))
>>> r.status_code
200
```

Comme qui dirait... « y a pas photo ! »

Il est intéressant de regarder la documentation, par exemple de 'get', afin de voir ce qui est supporté, et là on voit que la bibliothèque se charge de redirection, des sessions/cookies (CookieJar), timeout, proxies, et tout ce dont on a normalement besoin.

```
requests.get(url, params=None, headers=None, cookies=None,
             auth=None, timeout=None, proxies=None)
```

3.4.4 Conclusion

Voilà un petit survole de Requests, et bien que je n'ai pas encore eu tellement l'occasion de l'utiliser, c'est très certainement un module que je vais garder sous le coude. Pour des scripts jetables, ou des petites applications personnel, il me semble évident que ça peut être un gain de temps et de confort. Pour ce qui est de son utilisation pour une application redistribué, je comprends qu'on puisse préférer l'utilisation d'un module *core* tel que urllib, cependant, avec un usage de setup.py pour redistribuer, les dépendances sont installé très facilement sans actions supplémentaires de l'utilisateur, pourquoi pas utiliser Requests donc.

3.5 Scrapy, Crawler web

Les forums (oui, je n'utiliserai pas fora), sont un bon moyen de visualiser l'état de santé d'une communauté. Si on me présente un projet comme révolutionnaire et incontournable, et que le forum contient trois sujets et deux membres, j'aurais comme des doutes.

Dans le cas d'archlinux.fr, il pourrait être intéressant de visualiser le nombre d'inscriptions en fonction du temps, d'en visualiser une tendance générale. Et en prime, comparer les résultats avec le forum anglophone d'archlinux.org. C'est exactement ce qui va être fait dans ce billet.

Les forums fournissent généralement une liste de membre, avec la date d'inscription, le plus souvent accessible uniquement aux membres déjà inscrit (et ça tombe bien, ça rend le jeu plus intéressant). L'idée est simplement de récupérer toutes les dates d'inscriptions, d'en faire un cumule pour chaque mois, et de faire un petit graphique pour présenter et visualiser le résultat. Évidemment, comme c'est un blog aussi sur la programmation, on va voir comment obtenir se graphique ici.

J'avoue, le but était bien plus d'essayer de nouveaux jouets, et de programmer un peu, que de vraiment connaître le résultat qui ne sont pas à prendre au pied de la lettre, d'autres solutions certainement plus simple existe, mais je voulais vraiment essayer Scrapy dans un exemple réel. Scrapy semble être un outil vraiment puissant et modulable, il est évidant que l'utiliser ici pour résoudre se problème si simple est utiliser une solution démesuré. Cependant, je pense que pour découvrir un outil, il est préférable de l'essayer avec un problème simple. C'est pourquoi je l'emploie ici.

3.5.1 Prérequis

Ce billet s'appuie sur Scrapy pour crawler le forum et Pylab pour obtenir le graphique, pylab est fournis par matplotlib. Il convient de l'installer avant avec `pip`, `easy_install` ou des gestionnaires de paquets comme `pacman` et `apt-get`. Dans le cas d'archlinux.org, il n'est accessible uniquement par du https, il convient d'installer pyopenssl

Arch Linux et Yaourt

```
yaourt -S scrapy python2-matplotlib pyopenssl
```

Pip

```
sudo pip install matplotlib scrapy pyopenssl
```

3.5.2 Scrapy, récupérer les données

La première chose à faire est de créer un nouveau projet, scrapy vient avec une suite de commande shell, dont startproject nous facilite la mise en place de notre nouveau projet.

Génération du projet

```
scrapy startproject archlinux
```

```
$ tree archlinux
archlinux
|-- archlinux
|   |-- __init__.py
|   |-- items.py
|   |-- pipelines.py
|   |-- settings.py
|    -- spiders
|        -- __init__.py
 -- scrapy.cfg

2 directories, 6 files
```

La structure peut dérouter au début, mais elle est logique est sa prise en main rapide. En gros, on va créer un *spider* dans le répertoire, qui récupèrera les données, et suivra les urls qu'on lui indiquera, il se chargera de s'identifier au début sur le forum, chaque données, ou items seront défini dans items.py, et traite dans pipelines, dans ce cas afin de les sauvegarder.

Les items

Commençons par le plus simple, définir les *items*, qui sera tout seul en fait dans ce cas, seul la date nous intéresse.

archlinux/items.py :

```
from scrapy.item import Item, Field

class ArchlinuxItem(Item):
    date = Field()
```

Comme le générateur de projet fournis un squelette bien avancé déjà, il nous est utiles que de rajouter la ligne *date = Field()* permettant de définir l'item date.

Le spider

Cette partie est la plus compliqué, je vais essayer de l'expliquer au mieux. Dans le fichier `archlinux/spiders/archlinux_spider.py`. On commence par les imports.

```
from scrapy.spider import BaseSpider
from archlinux.items import ArchlinuxItem
from scrapy.http import FormRequest, Request
from scrapy.selector import HtmlXPathSelector
from scrapy.contrib.linkextractors.sgml import SgmlLinkExtractor
```

Puis on va créer une classe héritant d'un spider de base, dans l'exemple, on prends en compte le forum d'archlinux.fr.

À propos de l'extracteur de lien `extractor`, C'est l'initialisation de ce qui extrait les liens à suivre, d'une part on le restreint à memberlist, mais l'idée est de toujours suivre le lien "suivant" un peu comme on cliquerai pour aller du début à la fin de la liste de membre, en appuyant frénétiquement sur Suivant. Le suivant est trouvé en restreignant au bon lien avec Xpath.

```
class ArchLinuxSpider(BaseSpider):

    name = "archlinux"
    allowed_domains = ["forums.archlinux.fr"]
    extractor = SgmlLinkExtractor(allow='memberlist',
                                  restrict_xpaths='//a[. = "Suivant"]')
```

`start_requests` est la première fonction appelé après l'initialisation du bot, je m'en sers ici pour demander une page de login, afin de la remplir avec mes identifiants, cette façon de procéder permet de m'assurer que les champs caché (token, csrf...) soit correctement remplis, ou encore de gérer les sessions, cookies... On retourne un Request, avec comme callback la fonction qui gèrera le login.

```
def start_requests(self):
    login_url ='http://forums.archlinux.fr/ucp.php?mode=login'
    return [Request(login_url, callback=self.login)]
```

La page de login est reçu, on traite ici en utilisant une classe un peu spéciale FormRequest et surtout avec la méthode from_response. On renseigne les éléments, et la réponse de cette demande de login sera géré par la méthode after_login (callback).

```
def login(self, response):
    return [FormRequest.from_response(response,
                    formdata={'username': 'Nic0',
                              'password': 'correcthorsebatterystaple'},
                    callback=self.after_login)]
```

En dernier, on gère les pages normalement, avec *parse* qui est la fonction par défaut pour faire le traitement des pages. On y gère la réponse, c'est a dire la page html downloadé, et on en extrait les liens a suivre, qui seront rajouté dans la queue avec yield Request, le callback se fera dans cette fonction.

À propos de `HtmlXPathSelector`, on cherche a trouver tout les éléments contenant la date, ils sont extrait en regardant le code source d'une page html et en adaptent l'Xpath encore une fois. Chaque élément trouvé est rajouté avec yield item, qui est en relation avec l'item du fichier défini plus haut.

```
def parse(self, response):
    links = self.extractor.extract_links(response)
    for url in links:
        yield Request(url.url, callback=self.parse)
    hxs = HtmlXPathSelector(response)
    dates = hxs.select('//td[contains(@class, "genmed")]/text()').extract()
    for date in dates:
        item = ArchlinuxItem()
        item['date'] = date
        yield item
```

Le Pipeline

Dans cette exemple, on va simplement rajouter chaque élément dans un fichier, et le traîter par un petit script python plus tard, il serait faisable de le faire en même temps ici.

archlinux/pipelines.py :

```
class ArchlinuxPipeline(object):

    def __init__(self):
        self.file = open('result.txt', 'wb')

    def spider_closed(self, spider):
        self.file.close()

    def process_item(self, item, spider):
        item['date'] = item['date'][1:-1].split(',')[0]
        item['date'] = item['date']
        self.file.write(item['date'].encode('utf-8')+'\n')
        return item
```

Ce fichier à moins besoin d'explication, il s'assure surtout de ne stocker que la date, dans un format correct.

Pour que ce fichier soit pris en compte, il faut le rajouter dans la configuration, c'est la dernière ligne qui nous intéresse ici :

archlinux/settings.py :

```
BOT_NAME = 'archlinux'
BOT_VERSION = '1.0'

SPIDER_MODULES = ['archlinux.spiders']
NEWSPIDER_MODULE = 'archlinux.spiders'
DEFAULT_ITEM_CLASS = 'archlinux.items.ArchlinuxItem'
USER_AGENT = '%s/%s' % (BOT_NAME, BOT_VERSION)
ITEM_PIPELINES = ['archlinux.pipelines.ArchlinuxPipeline']
```

Mise en route

La mise en fonctionnement du bot est très simple et se fait en ligne de commande. Une longue suite de ligne de débug apparaîtra en console, mais le plus import est de vérifier le résultat obtenu. Et après avoir parser une trentaine de pages, on obtient le fichier result.txt voulu.

```
$ scrapy crawl archlinux
```

Le résultat semble correct, et surtout :

```
$ cat result.txt | wc -l
3017
```

Un rapide coup d'œil au forum, qui indique le nombre d'inscrit, je tombe exactement sur le même chiffre, ce qui est rassurant. Nous voilà avec un grand fichier, avec tout plein de dates. Il nous faut maintenant trouver le moyen de traiter ces informations.

3.5.3 Traitement des données

Les dates sont sous la forme *jour mois année*, on souhaite cumuler le nombre de *mois année* identique, et l'afficher sous forme de graphique. Le script suivant répond à ce besoin.

```
import pylab

with open('result.txt', 'r') as f:
    dates = []
    values = []
    for line in f:
        line = line.strip().split(' ')[2:]
        line = ' '.join(line)
        try:
            if dates[-1] == line:
                values[-1] += 1
            else:
                dates.append(line)
                values.append(1)
        except IndexError:
            dates.append(line)
            values.append(1)
```

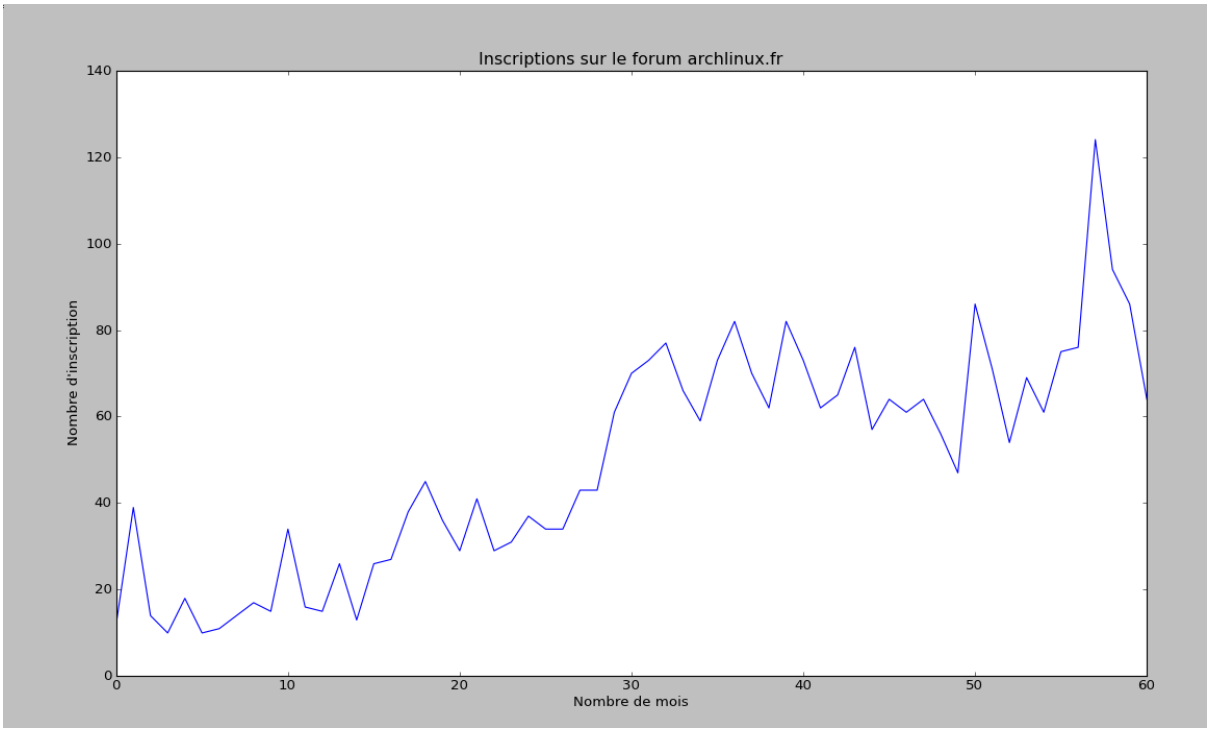
```
pylab.title('Inscriptions sur le forum archlinux.fr')
pylab.xlabel('Nombre de mois')
pylab.ylabel('Nombre d\'inscription')
pylab.plot(values)
pylab.show()
```

Le script fait ce qu'on demande, mais en y repensant, il y avait plus simple et plus élégant comme méthode, l'idée est de comparer la date avec le dernière élément et d'incrémenter ou de le rajouter selon le cas. Comme il existe une méthode permettant d'avoir le nombre d'occurrence d'un tableau, il aurait été préférable que je m'arrange de n'avoir que le *mois année* dans mon table et de traiter les occurences. Mais bon... Pylab gère également les abscisses avec les dates, je n'ai pas vu en détail ce fonctionnement bien qu'il aurait été pertinent de le faire.

3.5.4 Résultat

Il est temps de lancer le script, et de regarder les résultats obtenu, notons que le mois d'Août n'étant pas fini (au moment de la réduction du billet), il est normal de se retrouver avec une baisse pour le dernier mois.

3.5.5 Arch Linux Francophone



3.5.6 Arch Linux Anglophone

Le principe est le même, il faut simplement adapter certain détail pour le forum anglophone, qui n'utilise plus phpbb mais fluxbb, je place les codes ici, sans plus d'explications.

Il faut tout de même parser 760 pages pour obtenir les 34‘000 membres. Bien sûr, on retrouve dans notre fichier le nombre exacte de membres.

Tout de fois, un petit traitement du fichier en ligne de commande à été utile, d’une part avec vim (ou sed) car les inscription du jour et d’hier sont noté Yesterday et Today, au lieu de la date, ça pourrait fausser le résultat. D’autre part, pour que les dates soient dans l’ordre, un `sort` est requis. Si vraiment on y tient, il aurait été facile de le placer directement dans le script après.

archlinux/archlinux_spider.py :

```
from scrapy.spider import BaseSpider
from archlinux.items import ArchlinuxItem
from scrapy.http import FormRequest, Request
from scrapy.selector import HtmlXPathSelector
from scrapy.contrib.linkextractors.sgml import SgmlLinkExtractor

class ArchLinuxSpider(BaseSpider):

    name = "archlinux"
    allowed_domains = ["bbs.archlinux.org"]
    extractor = SgmlLinkExtractor(allow='userlist',
                                  restrict_xpaths='//a[. = "Next"]')

    def parse(self, response):
        links = self.extractor.extract_links(response)
        for url in links:
            yield Request(url.url, callback=self.parse)

        hxs = HtmlXPathSelector(response)
        dates = hxs.select('//td[contains(@class, "tcr")]/text()').extract()
        for date in dates:
            item = ArchlinuxItem()
            item['date'] = date
            yield item

    def start_requests(self):
        login_url ='https://bbs.archlinux.org/login.php'
        return [Request(login_url, callback=self.login)]

    def login(self, response):
        return [FormRequest.from_response(response,
                        formdata={'req_username': 'Nic0', 'req_password': 'my_password'},
                        callback=self.after_login)]

    def after_login(self, response):
        memberlist_url = 'https://bbs.archlinux.org/userlist.php'
        yield Request(memberlist_url, callback=self.parse)
```

Et maintenant le fichier *archlinux/pipelines.py* :

```
class ArchlinuxPipeline(object):
```

```
    def __init__(self):
        self.file = open('result.txt', 'wb')

    def spider_closed(self, spider):
        self.file.close()

    def process_item(self, item, spider):
        item['date'] = item['date'][:-3]
        self.file.write(item['date'].encode('utf-8')+'\n')
        return item
```

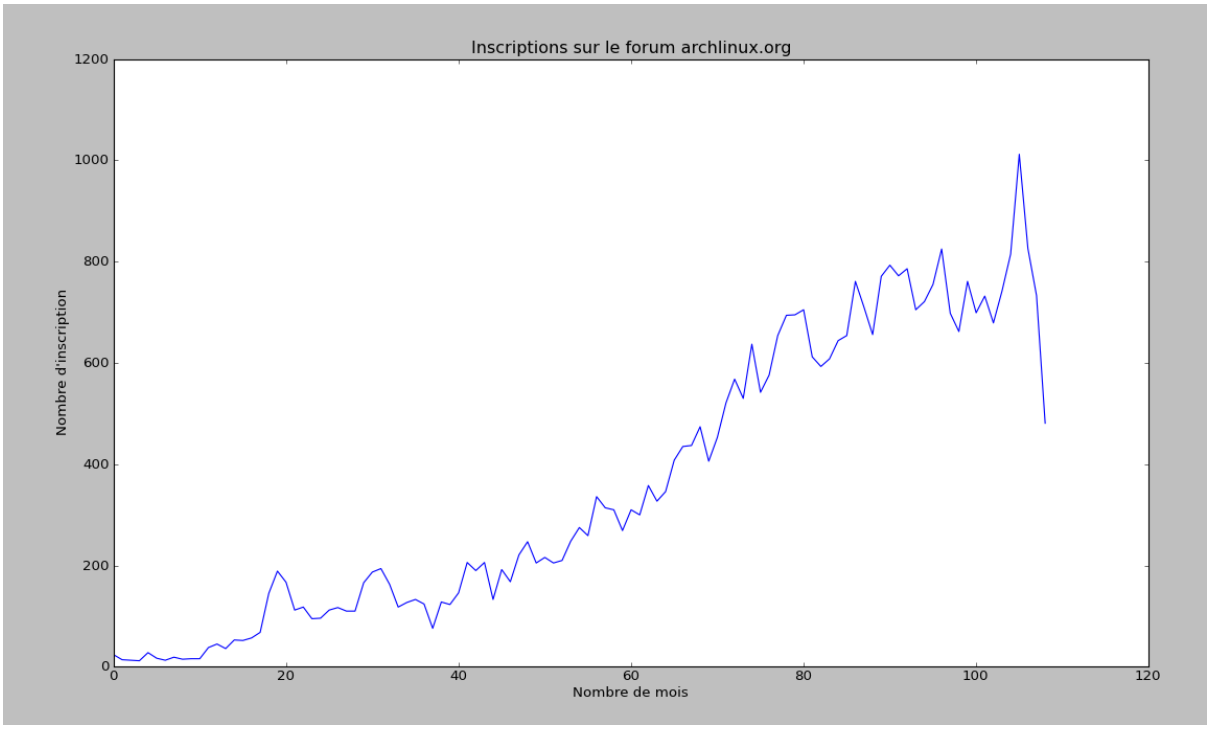
Et le script gérant pylab :

```
import pylab

with open('result.txt', 'r') as f:
    dates = []
    values = []
    for line in f:
        line = line.strip()
        try:
            if dates[-1] == line:
                values[-1] += 1
            else:
                dates.append(line)
                values.append(1)
        except IndexError:
            dates.append(line)
            values.append(1)

pylab.title('Inscriptions sur le forum archlinux.org')
pylab.xlabel('Nombre de mois')
pylab.ylabel('Nombre d\'inscription')
pylab.plot(values)
pylab.show()
```

Et le résultat en image :



3.5.7 Conclusion

Beaucoup de code pour pas grand chose pourrait on dire, cependant cela m'a été instructif sur bien des points.

Même si le titre du billet (originalement publié sous le titre *Petit bilan de santé d'Arch Linux*) n'est pas à prendre au pied de la lettre, ces deux graphiques donnent tout de même une petite indication sur l'état de santé des deux communautés d'Arch Linux. Chacun trouvera les interprétations à faire à partir des graphiques. Pour ma part, en conclusion, je dirai simplement :

Arch Linux se porte plutôt bien, et votre forum favori ?

CHAPITRE 4

Les Tests Unitaires

4.1 Unittest

4.1.1 Introduction

Quelques mots pour les personnes n'aillant jamais écrit de tests. Il y a beaucoup d'avantages à en écrire, par exemple être plus confient lors d'amélioration de code existant, être sûr de ne rien « casser ». Il s'agit bien souvent (pour les tests unitaires du moins) de tester une petite partie d'un code, afin de s'assurer qu'on obtient les valeurs auxquelles on s'attendait. Ce billet n'est pas là pour faire une introduction sur les avantages de tester son code, mais sachez que c'est une pratique indispensable et courante pour tout code.

Nous allons utiliser ici un module, unittest [1] qui est disponible directement avec Python. J'aurais pu commencer avec doctest [2], également un module natif à Python, permettant d'écrire les tests directement sous forme de commentaire, pour les personnes intéressées, la documentation officiel est certainement un bon endroit pour commencer. Ce billet n'est qu'un rapide aperçu de unittest, et ne se veux pas d'être complet.

4.1.2 Le code

Commençons par un exemple _très_ simple. En créant une fonction *add()* qui... additionne deux chiffres !

Créez un répertoire de test, dans lequel on crée la fonction suivante dans un fichier *chiffres.py*

```
def add (a, b):
    return a+b
```

On écrit maintenant le test correspondant

1. http ://docs.python.org/library/unittest.html
2. http ://docs.python.org/library/doctest.html

```
import unittest
from chiffres import add

class TestChiffres(unittest.TestCase):

    def test_addition(self):
        result = add(36, 6)
        self.assertEqual(result, 42)

if __name__ == '__main__':
    unittest.main()
```

4.1.3 Explications

On import le module pour unittest, ainsi que la fonction qu'on a crée :

```
import unittest
from chiffres import add
```

On crée une classe qui doit commencer par Test, et héritant de unittest.TestCase, correspondant à la ligne suivante :

```
class TestChiffres(unittest.TestCase):
```

La fonction suivante est celle qui est utilisé pour le test, et doit commencer par `test_` pour qu'elle soit pris en compte. Le plus important, c'est de vérifier la valeur que la fonction retourne avec celle auquel on s'attend, et c'est ce que fais la ligne suivante

```
self.assertEqual(result, 42)
```

La dernière partie, correspond à l'appel de la class par unittest :

```
if __name__ == '__main__':
    unittest.main()
```

Content de notre premier test, on fait un essai :

```
$ python test_chiffres.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Maintenant, on veut s'assurer que la fonction lève une exception si donne comme argument une chaine de caractères, on écrit donc le test :

```
import unittest
from chiffres import add

class TestChiffres(unittest.TestCase):

    def test_addition(self):
        result = add(36, 6)
        self.assertEqual(result, 42)

    def test_add_string(self):
        self.assertRaises(ValueError, add, 'coin', 'pan')

if __name__ == '__main__':
    unittest.main()
```

On utilise assertRaises, avec comme argument, l'exception attendu, la fonction et une liste d'argument fournis à la fonction. On essaye le test :

```
$ python test_chiffres.py
F.
======================================================================
FAIL: test_add_string (__main__.TestChiffres)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_chiffres.py", line 13, in test_add_string
    self.assertRaises(ValueError, add, 'coin', 'pan')
AssertionError: ValueError not raised by add

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Le test échoue, on s'y attendait un peu, puisque le résultat retourné est la concaténation des deux chaines, c'est à dire 'coinpan'.

On écrit un bout de code qui lèvera une erreur en cas de chaines passé en argument, comme suit :

```
def add (a, b):
    return int(a)+int(b)
```

On execute maintenant le test :

```
$ python test_chiffres.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Très bien, c'est ce qu'on voulait. Mais que ce passe-t-il si on envoie comme argument des floats (nombre à virgules) ?

Écrivont le test, qu'on rajoute à la suite, et regardons

```
def test_add_with_float(self):
    result = add(1.01, 2.001)
    self.assertEqual(result, 3.011)</code>

$ python test_chiffres.py
.F.
======================================================================
FAIL: test_add_with_float (__main__.TestChiffres)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_chiffres.py", line 17, in test_add_with_float
    self.assertEqual(result, 3.011)
AssertionError: 3 != 3.011

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)</code>
```

La ligne qui nous renseigne ici est surtout la suivante :

```
AssertionError: 3 != 3.011
```

Et effectivement int(a)+int(b) ne retournera pas de float, changeons le code pour que le test passe maintenant avec succès :

```
def add (a, b):
    if isinstance(a, basestring) or isinstance(b, basestring):
        raise ValueError
    return a+b</code>
```

Note : basestring, qui ici nous permet de comparer l'objet à une chaîne de caractère, est spécifique à Python 2, puisque dans la version 3, elles ne sont plus géré de la même manière.

On exécute une dernière fois le test :

```
$ python test_chiffres.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Voilà, grâce au test, on sait que la fonction à un comportement auquel on s'attend, on pourrait essayer de faire quelques modification, mais avec les tests, on s'assure que son comportement ne change pas de façon inattendu.

Note : Pour les floats et à cause du caractère parfois approximatif de leur résultat, il est sûrement préférable d'utiliser `assertAlmostEqual` au lieu de `assertEqual`

4.1.4 Précisions

Verbosité

On peut rajouter de la verbosité pour les tests, avec l'une des deux méthodes suivante :
– Directement en ligne de commande

:: $ python -m unittest -v test_chiffres.py test_add_string (test_chiffres.TestChiffres) ... ok test_add_with_float (test_chiffres.TestChiffres) ... ok test_addition (test_chiffres.TestChiffres) ... ok

Ran 3 tests in 0.001s

OK

– Directement dans le code

En remplacent le unittest.main() comme suit :

```
if __name__ == '__main__':
    suite = unittest.TestLoader().loadTestsFromTestCase(TestChiffres)
    unittest.TextTestRunner(verbosity=2).run(suite)</code>
```

Liste des asserts

On a vu jusqu'ici assertEqual et assertRaises, il en existe bien d'autre, dont voici la liste.
– assertEqual
– assertNotEqual
– assertTrue
– assertFalse
– assertIs
– assertIsNot
– assertTruetIsNone
– assertIsNotNone
– assertIsNottIn
– assertNotIn
– assertIsInstance
– assertNotIsInstance
– assertRaises
– assertRaisesRegex
– assertWarns
– assertWarnsRegex
Attention cependant à la compatibilité entre les versions de Python, pas mal de nouveautés on été introduite dans Python2.7 par exemple, ou 3.2. Pour aller plus loin, il est préférable de se référer à la documentation officiel.

Discover

Ce module vient maintenant avec une nouvelle option, ‘discover’, permettant d’automatiser la recherche des différents fichier de test, je ne m’étendrais pas sur ce détail, et on verra pourquoi dans le prochain billet.

```
$ python -m unittest discover
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

4.2 Introduction à Nose

Pour le moment, on a vu quelques bases du module Python unittest. Dans ce billet, on va voir comment faciliter l’exécution d’une suite de tests, même hétérogène, et plus encore.

Nose, s’intègre avec unittest et doctest, mais propose également sa propre syntaxe simplifié pour écrire des tests. Dans ce billet, on prendra la même fonction aillant servis au dernier billet, c’est à dire une fonction `add()`, additionnant simplement deux nombres entre eux.

4.2.1 Installation

Comme souvent, plusieurs façon d’installer Nose, pour Arch Linux par exemple, on peut installer l’un des deux :

```
yaourt -S python2-nose
yaourt -S python-nose
```

D’une façon plus générale, on peut utiliser l’outil `easy_install` ou `pip` :

```
sudo pip install nose
```

4.2.2 Utilisation de base

Reprenons où nous en étions la dernière fois, mais en séparant les fichiers sources et les fichiers tests, selon la structure suivante :

```
$ tree
.
|-- src
|    -- chiffres.py
 -- tests
     -- test_chiffres.py

2 directories, 2 files</code>
```

Avec le contenu des fichiers :

src/chiffres.py :

```
def add (a, b):
    if isinstance(a, basestring) or isinstance(b, basestring):
        raise ValueError
    return a+b
```

tests/test_chiffres.py :

```
import unittest
from chiffres import add

class TestChiffres(unittest.TestCase):

    def test_addition(self):
        result = add(36, 6)
        self.assertEqual(result, 42)

    def test_add_string(self):
        self.assertRaises(ValueError, add, 'coin', 'pan')

    def test_add_with_float(self):
        result = add(1.01, 2.001)
        self.assertEqual(result, 3.011)

if __name__ == '__main__':
    unittest.main()
```

Nose, contrairement à unittest, est prévu nativement pour parcourir les fichiers et arborescences afin de découvrir les différents tests. Le code est dans l'état laissé dans le dernier billet. Regardons le résultat de nose, placé vous dans la racine du projet, et entrez la commande :

```
$ nosetests
...
----------------------------------------------------------------------
Ran 3 tests in 0.016s

OK
```

On obtient ce qu'on avait avant.

4.2.3 Fichier de configuration

Un avantage de nose, c'est qu'il est possible de le faire fonctionner avec un fichier de configuration (`~/.noserc`) permettant d'enregistrer les préférences.

On peut ainsi rajouter de la verbosité comme paramètre de défaut, mais également inclure les doctests. De façon plus général, chaque argument dans la liste obtenu avec `nosetests -h` peut être rajouté dans votre .noserc. Prenons un exemple :

```
[nosetests]
verbosity=3
with-doctest=1
doctest-extention=txt
```

De cette façon, je n'ai plus à m'occuper de la verbosité, elle est automatiquement incluse, on va également rajouter un fichier .txt, contenant un test tout simple en doctest, pour voir qu'il est bien inclue directement.

tests/test_doctest_chiffres.txt :

```
>>> from chiffres import add
>>> add(2,5)
7
```

Puis, on exécute la suite de test avec la commande `nosetests` :

```
$ nosetests
test_add_string (test_chiffres.TestChiffres) ... ok
test_add_with_float (test_chiffres.TestChiffres) ... ok
test_addition (test_chiffres.TestChiffres) ... ok
Doctest: test_doctest_chiffres.txt ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.032s

OK
```

Les trois tests venant de unitest, auquel on rajoute doctest, le compte y est.

4.2.4 Framework de test spécifique à Nose

Bien que Nose soit compatible avec l'usage de unittest, il comporte également une version un peu allégé, mais pas moins complète de test. Incluant l'usage de `setUp()` et `tearDown()`, permettant de donner des directives avant et après le tests, souvent utile à la connexion de base de donnée par exemple. Les tests n'ont pas besoin d'hériter de unittest.Test.Case, et peut être de simples fonctions.

Dans cette exemple, on va reproduire dans un nouveau fichier, les tests déjà écrit avec unittest.

tests/test_nose_chiffres.py :

```
from chiffres import add
from nose.tools import raises

def test_add_int():
    assert add(3, 4) == 7

def test_add_float():
    assert add(2.01, 1.01) - 3.02 < 0.001
```

```
@raises(ValueError)
def test_add_chaine():
    add('coin', 'pan')
```

La première chose qu'on remarque, c'est la légèreté de la syntaxe, l'absence de classe (bien qu'on aurait pu) et l'usage *d'assert*.

Pour le test de float, je ne crois pas avoir trouvé un almostEqual, j'ai donc mis le code suivant qui revient un peu près au même donc :

```
assert add(2.01, 1.01) - 3.02 < 0.001
```

L'autre différence notable est pour la gestion de l'exception, qui se fait maintenant par un décorateur, et l'import correspondant (ne pas l'oublier) :

```
@raises(ValueError)
```

On relance nosetests comme suit :

```
$ nosetests
test_add_string (test_chiffres.TestChiffres) ... ok
test_add_with_float (test_chiffres.TestChiffres) ... ok
test_addition (test_chiffres.TestChiffres) ... ok
Doctest: test_doctest_chiffres.txt ... ok
test_nose_chiffres.test_add_int ... ok
test_nose_chiffres.test_add_float ... ok
test_nose_chiffres.test_add_chaine ... ok

----------------------------------------------------------------------
Ran 7 tests in 0.036s

OK
```

Comme prévu, nosetests à parcourus l'arborescence pour trouver les trois fichiers de tests (test_chiffres.py, test_doctest_chiffres.txt et test_nose_chiffres.py), dans ce cas les tests passent tous, mais on peut aisément à partir de là jouer un peu avec, pour voir son comportement. On note aussi que nosetests ne se soucis pas de passer d'un format (unittest, doctest) à son propre format de tests, le tout à la volée.

Comme indiqué plus haut, il est possible d'utiliser un setUp() et tearDown(), c'est à dire d'appeler une fonction avant et après l'exécution de chaque tests, utile lors de connexion à une base de donnée par exemple, le tout ce fait avec un décorateur (c'est pas la seul possibilité) voici l'exemple tiré de la documentation officiel.

```
@with_setup(setup, teardown)
def test_something():
    " ... "
```

Il faut bien entendu rajouter les deux fonctions correspondantes.

4.2.5 Pinocchio, plugin pour Nose

Pinocchio [3] est un plugin pour utiliser nose d'une façon plus proche de RSpec [4]. L'idée est de rendre les tests un peu plus parlant, un peu de la façon dont on procède pour le BDD (Behavior Driven Development), il est préférable ed se reporter à quelques documentations à ce sujet si vous n'êtes pas familier avec ce terme. Pinocchio ne semble pas fonctionner avec Python3, ou du moins, je n'ai pas réussi.

Installons le plugin :

```
sudo pip install pinocchio
```

Créons un troisième fichier de tests avec des tests plus « parlant » :

```
from chiffres import add
from nose.tools import raises

def test_should_add_two_integer():
    assert add(3, 4) == 7

def test_should_add_two_float():
    assert add(2.01, 1.01) - 3.02 < 0.001

@raises(ValueError)
def test_should_raise_an_exception_with_two_string():
    add('coin', 'pan')</code>
```

Deux possibilités, soit on utilise la commande suivante :

```
nosetests --with-spec --spec-color --spec-doctests
```

Ou plus simplement, on rajoute dans son fichier de configuration comme suit :

```
[nosetests]
verbosity=3
with-doctest=1
doctest-extension=txt
with-spec=1
spec-color=1
spec-doctests=1
```

On exécute maintenant (dans l'exemple, je force l'usage pour python 2.7, n'aillant pas réussi l'autre, et que Arch Linux fonctionne avec Python 3.x par défaut)

```
$ nosetests-2.7

Chiffres
- add string
- add with float
```

3. https ://github.com/infrared/pinocchio
4. http ://en.wikipedia.org/wiki/RSpec

```
- addition

test_doctest_chiffres.txt
- add(2,5) returns 7

Nose chiffres
- add int
- add float
- add chaine

Spec chiffres
- should add two integer
- should add two float
- should raise an exception with two string

----------------------------------------------------------------------
Ran 10 tests in 0.040s

OK
```

Bien qu'il n'apparaît pas ici, la sortie différencie les erreurs des bons tests avec les couleurs rouge/vert, et rajoutant un gros (ERROR) si besoin.

L'important à remarquer, c'est la lisibilité des tests pour spec, dont il est conseillé de faire des phrases qui ont un sens. L'exemple n'est certainement pas le plus parlant pour ce genre de tests, mais afin d'en garder une continuité, j'ai préféré garder toujours le même exemple.

Pour donner un exemple de sortie comportant une erreur :

```
$ nosetests-2.7

Chiffres
- add string
- add with float
- addition

test_doctest_chiffres.txt
- add(2,5) returns 7

Nose chiffres
- add int
- add float
- add chaine

Spec chiffres
- should add two integer
- should add two float
- should raise an exception with two string (ERROR)

======================================================================
ERROR: test_spec_chiffres.test_should_raise_an_exception_with_two_string
----------------------------------------------------------------------
```

```
Traceback (most recent call last):
  File "/usr/lib/python2.7/site-packages/nose-1.0.0-py2.7.egg/nose/case.py",
  line 187, in runTest
    self.test(*self.arg)
  File "/usr/lib/python2.7/site-packages/nose-1.0.0-py2.7.egg/nose/tools.py",
  line 80, in newfunc
    func(*arg, **kw)
  File "/home/nicolas/exo/chiffres/tests/test_spec_chiffres.py", line 15, in test_should_raise_a
    add('coin', 'pan')
  File "/home/nicolas/exo/chiffres/src/chiffres.py", line 5, in add
    raise ValueError
ValueError

----------------------------------------------------------------------
Ran 10 tests in 0.041s

FAILED (errors=1)
```

Pinocchio peut répondre à un besoin de faire des tests un peu différent, si le plugin fait ce qu'on lui demande, il ne semble pas être très activement maintenu. La raison est certainement que ce plugin ne correspond plus vraiment à un besoin, et que l'approche vu dans le prochain chapitre est plus pertinante pour ce type de développement. On y verra deux projets bien plus à jour et maintenu. Pour aller plus loin avec ce plugin, [une documentation][4] est à disposition.

4.2.6 Conclusion

Ce billet devrait faire découvrir un outil fort pratique pour effectuer des tests unitaires avec Python, et nous en avons vu qu'une petite partie de ses capacités. Il vient avec beaucoup de fonctionnalités et des plugins tiers. Les Plugins nativement prévu pour Nose sont compatible avec Python3, ce qui n'est pas toujours le cas avec les plugins tiers.

4.3 Behavior Driven Developpment avec Lettuce

Lettuce [5] est principallement un portage sur Python de Cucumber [6]. Cucumber a été écrit par la communauté de Ruby. Il permet une approche des tests et du développement de façon "BDD" Behavior Driven Developpement. Cet outil à été porté pour PHP (Behat [7]) et pour Python. C'est une pratique courante, comportant son lot d'adeptes et de récalcitrant..

Python dispose de deux outils semblable, Lettuce [3] et Freshen [8]. Les deux projets sont actif et fonctionnel. Lettuce est un standalone, tandis que Freshen est un plugin de Nose (voir mes deux précédents articles consacré à Nose). Bien que l'utilisation est peut être plus courante pour un développement web (Django par exemple) il est possible de s'en servir en tout autre contexte.

5. https ://github.com/gabrielfalcao/lettuce
6. http ://cukes.info/
7. http ://behat.org/
8. https ://github.com/rlisagor/freshen

Le but, est d'écrire un "scénario" compréhensible par n'importe qui, correspondant à un comportement voulu d'une fonctionnalité, de s'assurer que le test échoue, puis on écrit le code correspondant, et pour finir, on s'assure que le test passe avec succès. Ce billet n'est qu'une approche rapide, tiré principalement de l'exemple de la documentation.

4.3.1 Installation et doc de Lettuce

L'installation est toujours simplifié avec *pip* (pip-2.7 pour ArchLinux)

: : sudo pip install lettuce

La documentation du site n'étant pas des plus à jour, il est plus sage de la générer localement comme suit, comme vu dans la partie API/documentation avec Sphinx :

```
git clone https://github.com/gabrielfalcao/lettuce.git
cd lettuce/docs && make html
firefox _build/html/index.html
```

4.3.2 Exemple basique

L'exemple veut tester une fonction simple, factoriel.

Voici la structure avec lequel on part

```
.
|-- src
 -- tests
    -- features
```

Un peu déroutant à la première approche, mais les tests se range dans *features*.

L'important à comprendre, c'est qu'on va se retrouver avec deux types de fichiers.

1. *.feature : Sont les scénarios rédigé dans un anglais compréhensible par tous, décrivant une suite de comportement et de résultat attendu. 2. step.py : Le code servant à interpréter les scénarios.

Note : Il existe une coloration pour les features avec Vim, celui-ci le reconnais comme syntaxe de Cucumber. Emacs et tout autre éditeurs doivent certainement en faire de même.

Le code suivant, est là surtout pour donner une idée de la syntaxe utilisé pour lettuce, et du résultat obtenu. La documentation, que vous pouvez généré en local comme vu plus haut, donne un bon exemple de petit tutoriel reprenant un peu plus en détail.

Pour mieux visualisé, voici les répertoires et fichier après écriture du code :

```
$ tree
.
|-- __init__.py
|-- src
|   |-- fact.py
```

```
|    -- __init__.py
 -- tests
    |-- features
    |   |-- __init__.py
    |   |-- steps.py
    |    -- zero.feature
     -- __init__.py

3 directories, 7 files
```

tests/features/zero.feature :

```
Feature: Compute factorial
    In order to play with Lettuce
    As beginners
    We'll implement factorial

    Scenario: Factorial of 0
        Given I have the number 0
        When I compute its factorial
        Then I see the number 1

    Scenario: Factorial of 1
        Given I have the number 1
        When I compute its factorial
        Then I see the number 1

    Scenario: Factorial of 2
        Given I have the number 2
        When I compute its factorial
        Then I see the number 2

      Scenario: Factorial of 3
        Given I have the number 3
        When I compute its factorial
        Then I see the number 6

      Scenario: Factorial of 4
        Given I have the number 4
        When I compute its factorial
        Then I see the number 24
```

tests/features/steps.py :

```
from lettuce import *
import sys
sys.path.append('../src/')
from fact import factorial

@step('I have the number (\d+)')
def have_the_number(step, number):
```

```
    world.number = int(number)

@step('I compute its factorial')
def compute_its_factorial(step):
    world.number = factorial(world.number)

@step('I see the number (\d+)')
def check_number(step, expected):
    expected = int(expected)
    assert world.number == expected, \
        "Got %d" % world.number
```

src/fact.py :

```
def factorial(number):
    number = int(number)
    if (number == 0) or (number == 1):
        return 1
    else:
        return number*factorial(number-1)
```

Pour exécuter les tests, on se place dans tests, puis on exécute *lettuce*.

```
$ lettuce

Feature: Compute factorial       # features/zero.feature:1
  In order to play with Lettuce  # features/zero.feature:2
  As beginners                   # features/zero.feature:3
  We'll implement factorial      # features/zero.feature:4

  Scenario: Factorial of 0       # features/zero.feature:6
    Given I have the number 0    # features/steps.py:7
    When I compute its factorial # features/steps.py:11
    Then I see the number 1      # features/steps.py:15

  Scenario: Factorial of 1       # features/zero.feature:11
    Given I have the number 1    # features/steps.py:7
    When I compute its factorial # features/steps.py:11
    Then I see the number 1      # features/steps.py:15

  Scenario: Factorial of 2       # features/zero.feature:16
    Given I have the number 2    # features/steps.py:7
    When I compute its factorial # features/steps.py:11
    Then I see the number 2      # features/steps.py:15

  Scenario: Factorial of 3       # features/zero.feature:21
    Given I have the number 3    # features/steps.py:7
    When I compute its factorial # features/steps.py:11
    Then I see the number 6      # features/steps.py:15

  Scenario: Factorial of 4       # features/zero.feature:26
```

```
    Given I have the number 4    # features/steps.py:7
    When I compute its factorial # features/steps.py:11
    Then I see the number 24     # features/steps.py:15

1 feature (1 passed)
5 scenarios (5 passed)
15 steps (15 passed)
```

La sortie se fait en couleur bien entendu.

4.3.3 Conclusion

Encore une fois, l'important ici est d'écrire les tests avant d'en écrire le code, de s'assurer qu'il ne passe pas, et ensuite d'écrire le code correspondant afin que le test passe, principe du BDD.

Je suis aller assez vite sur le fonctionnement car je pense que si le rapide aperçu vous intrigue, vous aurez de toute façon bien assez envi d'attaquer leur tutoriel, et creuser les quelques pistes dont le billet était sujet, avant tout la présentation d'un outil, que l'explication de son fonctionnement.

4.4 Couverture de code

Nous allons voir cette fois ci une façon de visualiser, par console ou html, la quantité de code que couvre les tests. Lorsque le code et le nombre de tests correspondant grossi, il est utile de trouver un moyen d'en faire un bilan, il existe pour cela la couverture de code [9], sûrement plus connu sous son nom anglais de **code coverage**. C'est ce que l'on va regarder dans ce billet, à l'aide de Nose et de Coverage.

4.4.1 Installation

Si vous n'avez pas déjà installé Nose, faite le comme suit, ou consultez le chapitre sur nose pour plus de détails.

```
sudo pip install nose
```

Pour installer coverage, la procédure est similaire, on peut déjà noté que coverage n'est utilisé uniquement ici pour générer l'html.

```
sudo pip install coverage
```

Les utilisateurs d'Arch Linux pourront le retrouvé avec l'AUR

```
3 aur/python-coverage 3.4-1 (Out of Date) (36)
    A tool for measuring code coverage of Python programs.
```

9. http ://fr.wikipedia.org/wiki/Couverture_de_code

On note que le paquet est marqué périmé, cependant la version 3.5 n'est sortie que depuis un mois, et cette version doit certainement faire l'affaire.

4.4.2 Usage console

Reprenons l'exemple précédant, une fonction add, auquel on rajoute une fonction multiply, mais sans lui rajouter de tests. La première fonction est testé de diverses façons (doctest, unit-test, framework test de nose), alors que la seconde n'en comporte aucun.

Le fichier `src/chiffres.py` ressemble maintenant à ça :

```
def add (a, b):
    if isinstance(a, basestring) or isinstance(b, basestring):
        raise ValueError
    return a+b

def multiply(a, b):
    if isinstance(a, basestring) or isinstance(b, basestring):
        raise ValueError
    return a*b
```

On utilise l'argument `--with-coverage` de Nose, ou tout simplement rajouter `with-coverage=1` dans le `~/.noserc`.

On effectue les tests précédemment écrit, attention à basestring qui n'existe que pour Python2.x mais plus dans Python3 :

```
$ nosetests-2.7 --with-coverage
test_add_string (test_chiffres.TestChiffres) ... ok
[.....]
test_spec_chiffres.test_should_raise_an_exception_with_two_string ... ok

Name       Stmts   Miss  Cover   Missing
--------------------------------------
chiffres       8      3    63%   9-11
----------------------------------------------------------------------
Ran 10 tests in 0.046s

OK
```

L'important est de remarquer le pourcentage, mais également le *9-11* correspondant au lignes pour lequel il ne comporte pas de tests. L'exemple étant petit, on pouvait retrouver le résultat de tête facilement.

Une alternative est d'utiliser directement coverage comme suit :

```
$ coverage report
Name                      Stmts   Miss  Cover
-------------------------------------------
src/chiffres                  8      3    63%
tests/test_chiffres          13      1    92%
```

```
tests/test_nose_chiffres       8      0   100%
tests/test_spec_chiffres       8      0   100%
----------------------------------------------
TOTAL                         37      4    89%
```

Il manque cependant l'indication sur les lignes manquantes, sauf si je l'ai raté, mais il n'est peut être pas utile de toujours l'avoir.

4.4.3 Sortie HTML

Nose vient avec un argument `--cover-html`, cependant il est inutile de perdre du temps à essayer de le faire fonctionner, il comporte un bug depuis quelques années, le développeur le sait, et il est probable que cela reste comme ça encore longtemps, l'auteur encourage l'utilisation de *coverage* qui produit de toute façon un meilleur html (selon ses dires).

Utiliser `nosetests --with-coverage` avant coverage lui sert de hook.

> $ nosetests-2.7 –with-coverage [...] $ coverage html

Le résultat se trouvera dans le répertoire *htmlcov/* pour lequel il suffit par exemple de faire :

```
$ firefox htmlcov/index.html
```

On obtiendra un résultat similaire à l'image suivante :

Un exemple plus réel et navigable est visitable sur ce lien [10].

Voilà pour une présentation rapide d'une utilisation un peu spécifique de Nose, mais pouvant être utile.

10. http ://nedbatchelder.com/code/coverage/sample_html/

4.5 Autotest avec Watchr

4.5.1 Introduction

Watchr [1]_est un petit utilitaire pouvant être très pratique, il permet d'automatiser des tests à la sauvegarde de fichier. Bien qu'il soit écrit en Ruby, il peut être utilisé avec n'importe quel langage, je m'en suis servis par exemple pour Symfony2 (avec PHPUnit), mais plus récemment avec Python. Voici une liste d'avantage qu'on peut trouver à ce genre d'outil :
– Entièrement scriptable
– Fonctionne avec des regex
– Indépendant de tout langages ou frameworks (PHPunit, unittest, nosetests...)
– Possibilité d'utiliser la notification du système d'exploitation
– Réutilisable d'un projet à l'autre
– Met à disposition toutes les richesses de Ruby

4.5.2 Installation

L'installation se fait simplement avec rubygem, ou directement avec le dépôt git. Comme l'utilisation de rubygem est plus propre, voici comment, en root :

```
gem install watchr
```

4.5.3 Mode de fonctionnement

L'idée, et de faire correspondre à des fichiers "surveillés" une suite de commande, que l'on aura scripté dans un fichier de configuration au préalable. Ces règles vont être écrites dans un fichier appelé watchrc.rb, et la syntaxe est celle du Ruby comme on peut le deviner. Le nom du fichier est arbitraire et passé en argument.

4.5.4 Python et la syntaxe

Un premier exemple permet de vérifier la syntaxe du fichier à chaque sauvegarde dans une console séparé. Certes des plugins de Vim on une approche similaire pour la syntaxe de Python, mais comme on le vera, watchr offre d'autres possibilités.

watchrc.rb :

```
watch '^src/(.*)\.py$' do |match|
    pychecker match[0]
end

def pychecker file
  system("pychecker --stdlib #{file}") if File.exists?(file)
end
```

On lance l'application :

```
watchr watchrc.rb
```

Puis on commence à éditer un fichier Python dans le répertoire 'src'. Alors qu'une fois l'application lancé, rien ne s'affichait, et semblait attendre, dès lors qu'on sauvegarde le fichier, la vérification syntaxique se fait automatiquement, avec la sortie en console.

4.5.5 Python et les tests

On va automatiser les tests à la sauvegarde de fichier, en surveillant d'une part le répertoire *src* et *tests*. Dans un premier temps, on va au plus simple, c'est à dire que si l'un est déclenché, on lance la totalité des tests. Cependant, dans un cas où le projet est plus gros, on ne souhaite pas forcément lancer tout les tests, mais seulement ceux correspondant au fichier édité. Mais chaque exemple en son temps.

watchrc.rb :

```
watch '^src/(.*)\.py$' do |match|
    unittest
end

watch '^tests/(.*)\.py$' do |match|
    unittest
end

def unittest
  system("nosetests")
end
```

Pour s'assurer que ça fonctionne, on démarre watchr comme vu précédemment, et on édite un des fichiers surveillé, si tout ce passe bien, le tests devrait s'enclencher automatiquement après la sauvegarde.

```
watchr watchrc.rb
```

4.5.6 Python et un test

Comme dit plus haut, si la suite de tests prend trop de temps à s'exécuter, il est préférable de ne sélectionner qu'un seul à effectuer. Pour faire cela, ça dépends des conventions utilisés par chacun, pour ma part, je fais toujours correspondre :
– src/mon_fichier.py
– tests/test_mon_fichier.py
L'astuce sera donc de passer en argument le fichier de tests a exécuter, si le fichier édité est le fichier source, une substitution de la chaine sera requise, si le fichier édité est le fichier de test, on peut l'exécuter tel quel.

watchrc.rb :

```
watch '^src/(.*)\.py$' do |match|
    unittest match[0].sub('src/', 'tests/test_')
end

watch '^tests/(.*)\.py$' do |match|
    unittest match[0]
end

def unittest file
    system("nosetests #{file}") if File.exists? (file)
end
```

Le résultat attendu est obtenu, en console.

4.5.7 Notification système

Maintenant que le script fonctionne, rajoutons une notification, afin de ne pas avoir besoin de garder la console sur le bureau courent, tout en aillant une indication. La notification est simple, indiquant si le test passe ou non, en rajoutant une petite icon (‘ok vert’, ‘pas bon rouge’) afin de garder un reperd visuel, pour un gain de confort et de temps.

watchrc.rb :

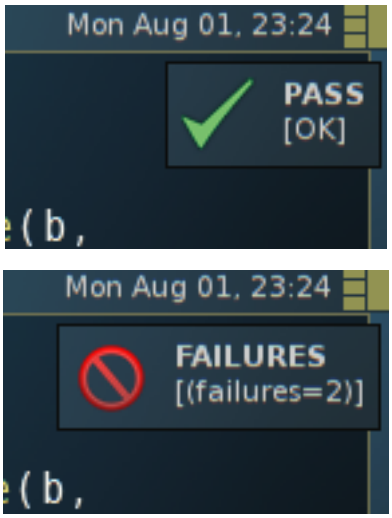
```
watch '^src/(.*)\.py$' do |match|
    unittest match[0].sub('src/', 'tests/test_')
end

watch '^tests/(.*)\.py$' do |match|
    unittest match[0]
end

def unittest file
    cmd = "nosetests-2.7 #{file} 2>&1"
    out = `#{cmd}`
    puts(out)
    notify out
end

def notify (message)
    result = message.split(' ').last(1)
    title = result.find { |e| /failures/ =~ e } ? "FAILURES" : "PASS"
    if title == "PASS"
        icon = "~/ok.png"
    else
        icon = "~/fail.png"
    end
    system ("notify-send -i #{icon} #{title} #{result}")
end
```

Et le résultat en image :



4.5.8 Pour les autres langages

Il est évident que ce code peut être adapté à tout framework de test unitaire ou de test de syntaxe, correspondant au divers langages, comme par exemple PHP avec phpunit. Il suffit d'adapter la commande et les structures de répertoires selon le besoin.

À titre informatif, je mets un exemple d'une personne aillant rédigé pour du php un billet [2]_(en), et une configuration [11] assez complète pour Rails.

4.5.9 Conclusion

Watchr est un utilitaire bien pratique et surtout malléable, toujours bon à connaître.

11. https ://raw.github.com/gist/276317/45b7ca8a20f0585acc46bc75fade09a260155a61/tests.watchr