Notes de cours Python scientifique

Sujets avancés 5 - Apprendre à déboguer du code Python

Par Gaël Varoquaux - Christophe LOUVET (traducteur)

Date de publication : 12 octobre 2016

Ce tutoriel explore les outils pour mieux comprendre la base de votre code : le débogage, trouver et corriger vos bogues.

Ce n'est pas spécifique à la communauté scientifique Python, mais les stratégies employées sont adaptées à leurs besoins.

Conditions préalables :

- Numpy ;
- Ipython ;
- nosetests (**http://readthedocs.org/docs/nose/en/latest/**);
- pyflakes (**http://pypi.python.org/pypi/pyflakes**);
- gdb pour la partie débogage C.

*Retrouvez **Les notes de cours scientifique de Python***

Commentez

1 - Éviter les bogues

1-1 - Les meilleures pratiques de codage pour éviter d'avoir des problèmes

Brian Kernighan :

« Tout le monde sait que le débogage est deux fois plus difficile que l'écriture d'un programme. Donc, si vous êtes aussi intelligent que vous pouvez l'être quand vous écrivez votre code, comment allez-vous déboguer ? »

- Nous écrivons tous du code bogué, acceptez-le et faites avec.
- Écrivez votre code avec le test et le débogage à l'esprit.
- Keep It Simple, Stupid (KISS) (gardez-le simple, stupide).
    - Quelle est la chose la plus simple qui pourrait fonctionner ?
- Ne vous répétez pas.
    - Chaque bout de connaissance doit avoir une seule représentation autorisée et non ambiguë dans un système.
    - Constantes, algorithmes, etc.
- Essayez de limiter l'interdépendance de votre code. (Loose Coupling)
- Donnez à vos variables, fonctions et modules des noms explicites (pas des noms mathématiques).

1-2 - Pyflakes : analyse statique rapide

Il y a plusieurs outils d'analyse statique en Python, pour en citer quelques-uns :

- **pylint** ;
- **pychecker** ;
- **pyflakes** ;
- **pep8** ;
- **flake8**.

Nous nous pencherons ici sur pyflakes qui est l'outil le plus simple :

- rapide, simple ;
- détecte les erreurs de syntaxe, les imports manquants, les fautes de frappe dans les noms.

Une autre bonne recommandation est l'outil flake8, qui est une combinaison de Pyflakes et de pep8. Ainsi, en additions des types d'erreurs interceptés par pyflakes, flacke8 détecte les violations des recommandations du guide de style **PEP8**.

Intégrer pyflakes (ou flake8) dans votre éditeur ou votre IDE est hautement recommandé. **Ils font générer des gains de productivité.**

1-2-1 - Démarrer pyflakes sur le fichier en cours d'édition

Vous pouvez configurer une touche pour lancer pyflakes dans le buffer courant.

- **Dans le menu kate** : réglages -> configurer kate.
    - Dans activer les plugins ‘outils externes'.

    - Dans outils 'externes', ajouter pyflakes :
      kdialog --title "pyflakes %filename" --msgbox "$(pyflakes %filename)"
- **Dans TextMate**
  Menu : TextMate -> Préférences -> Avancées -> variables Shell, ajoutez une variable Shell :
- Donc Ctrl-Shift-V est lié à un rapport pyflakes report.
- **Dans vim** dans votre fichier vimrc (lie F5 à pyflakes) :
- **Dans emacs** dans votre fichier emacs (lie F5 à pyflakes) :

```
(defun pyflakes-thisfile () (interactive)
       (compile (format "pyflakes %s" (buffer-file-name)))
)

(define-minor-mode pyflakes-mode
    "Toggle pyflakes mode.
    With no argument, this command toggles the mode.
    Non-null prefix argument turns on the mode.
    Null prefix argument turns off the mode."
    ;; The initial value.
    nil
    ;; The indicator for the mode line.
    " Pyflakes"
    ;; The minor mode bindings.
    '( ([f5] . pyflakes-thisfile) )
)

(add-hook 'python-mode-hook (lambda () (pyflakes-mode t)))
```

1-2-2 - Une intégration de correcteur orthographique à la volée

- **Dans vim**

    - Utilisez le plugin vim pyflakes :

        1. Téléchargez le fichier zip depuis **http://www.vim.org/scripts/script.php?script_id=2441** ;
        2. Faites l'extraction des fichiers dans ~/.vim/ftplugin/python ;
        3. Assurez-vous que votre fichier vimrc ait filetype plugin indent on

```
                def compute_log_likelihood(obs):
                    return self._log_emissionprob[:, obs].T
```

    - Sinon : utilisez le plugin **syntastic**. Celui-ci peut être configuré pour utiliser flake8 aussi et gérer également un contrôle à la volée pour plusieurs autres langages.

```
            if __name__=='__main__'
                data = load_data('exercices/data.txt')
                print ('min: %f' %min(data)) # 10.20
                print ('max: %f' %max(data)) # 61.30
```

- **Dans emacs** utilisez le mode flymake avec pyflakes, documenté ici : **http://www.plope.com/Members/chrism/flymake-mode** : ajoutez le code suivant dans votre fichier .emacs :

```
(when (load "flymake" t)
        (defun flymake-pyflakes-init ()
        (let* ((temp-file (flymake-init-create-temp-buffer-copy
                           'flymake-create-temp-inplace))
            (local-file (file-relative-name
                         temp-file
                         (file-name-directory buffer-file-name))))
            (list "pyflakes" (list local-file))))

        (add-to-list 'flymake-allowed-file-name-masks
                '("\\.py\\'" flymake-pyflakes-init)))

(add-hook 'find-file-hook 'flymake-find-file-hook)
```

2 - Le travail de débogage

Si vous avez un bogue non banal, c'est là que les stratégies de débogage vont rentrer en ligne de compte. Il n'y a pas de solution miracle, les stratégies aideront.

Pour déboguer un problème donné, la situation favorable est quand le problème est isolé dans un petit nombre de lignes de code, en dehors de framework ou de code d'application, avec un cycle court de modification, lancement, échec.

1 Faites échouer le code de façon fiable : trouvez un cas de test qui fait échouer le code à chaque fois.
2 Diviser et conquérir : une fois que vous avez un cas de test échouant, isolez le code coupable.

- Quel module.
- Quelle fonction.
- Quelle ligne de code.

=> Isolez une petite erreur reproductible : un cas de test.
3 Changez une seule chose à chaque fois et réexécutez le cas de test d'échec.
4 Utilisez le débogueur pour comprendre ce qui ne va pas.
5 Prenez des notes et soyez patient, ça peut prendre un moment.

Une fois que vous avez procédé à cette étape, isolez un petit bout de code reproduisant le bogue et corrigez celui-ci en utilisant ce bout de code, ajoutez ce code dans votre suite de test.

3 - Utiliser le débogueur de Python

Le débogueur python, pdb : **https://docs.python.org/library/pdb.html**, vous permet d'inspecter votre code de façon interactive.

Plus précisément, il vous permet :

- de voir votre code source ;
- de monter et descendre la pile d'appel ;
- d'inspecter les valeurs des variables ;
- de modifier les valeurs des variables ;
- de poser des breakpoints.

print

Oui, les déclarations print fonctionnent comme un outil de débogage. Cependant, pour inspecter l'exécution, il est souvent plus efficace d'utiliser le débogueur.

3-1 - Invoquer le débogueur

Les façons de lancer le débogueur :

1 Postmortem, lancez celui-ci après une erreur de module ;
2 Lancez le module avec le débogueur ;

3 Appelez le débogueur à l'intérieur du module.

3-1-1 - Postmortem

Situation : vous travaillez dans IPython et vous obtenez un traceback.

Ici, nous déboguons le fichier **index_error.py**. Lors de son lancement, une IndexError est levée. Tapez %debug et placez-vous dans le débogueur.

```
In [1]: %run index_error.py
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/home/varoquau/dev/scipy-lecture-notes/advanced/debugging/index_error.py in <module>()
      6
      7 if __name__ == '__main__':
----> 8     index_error()
      9

/home/varoquau/dev/scipy-lecture-notes/advanced/debugging/index_error.py in index_error()
      3 def index_error():
      4     lst = list('foobar')
----> 5     print lst[len(lst)]
      6
      7 if __name__ == '__main__':

IndexError: list index out of range

In [2]: %debug
> /home/varoquau/dev/scipy-lecture-notes/advanced/debugging/index_error.py(5)index_error()
      4     lst = list('foobar')
----> 5     print lst[len(lst)]
      6

ipdb> list
      1 """Small snippet to raise an IndexError."""
      2
      3 def index_error():
      4     lst = list('foobar')
----> 5     print lst[len(lst)]
      6
      7 if __name__ == '__main__':
      8     index_error()
      9

ipdb> len(lst)
6
ipdb> print lst[len(lst)-1]
r
ipdb> quit

In [3]:
```

Débogage post-mortem sans Ipython :

Dans certaines situations, vous ne pouvez pas utiliser IPython, par exemple pour déboguer un script qui peut être appelé depuis la ligne de commande. Dans ce cas, vous pouvez appeler le script avec python -m pdb script.py :

```
$ python -m pdb index_error.py
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/index_error.py(1)<module>()
-> """Small snippet to raise an IndexError."""
(Pdb) continue
Traceback (most recent call last):
File "/usr/lib/python2.6/pdb.py", line 1296, in main
    pdb._runscript(mainpyfile)
File "/usr/lib/python2.6/pdb.py", line 1215, in _runscript
    self.run(statement)
```

```
File "/usr/lib/python2.6/bdb.py", line 372, in run
    exec cmd in globals, locals
File "<string>", line 1, in <module>
File "index_error.py", line 8, in <module>
    index_error()
File "index_error.py", line 5, in index_error
    print lst[len(lst)]
IndexError: list index out of range
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/index_error.py(5)index_error()
-> print lst[len(lst)]
(Pdb)
```

3-1-2 - Exécution pas-à-pas

Situation : vous pensez qu'un bogue existe dans un module, mais vous n'êtes pas sûr de l'endroit.

Par exemple, vous essayez de déboguer **wiener_filtering.py**. En effet le code fonctionne, mais le filtrage ne fonctionne pas bien.

- Démarrez le script dans IPython avec le débogueur en utilisant `%run -d wiener_filtering.p` :

```
In [1]: %run -d wiener_filtering.py
*** Blank or comment
*** Blank or comment
*** Blank or comment
Breakpoint 1
 at /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.py:4
NOTE: Enter 'c' at the ipdb>  prompt to start your script.
> <string>(1)<module>()
```

- Placez un point d'arrêt à la ligne 34 en utilisant `b 34` :

```
ipdb> n
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.py(4)<module>()
      3
1---> 4 import numpy as np
      5 import scipy as sp

ipdb> b 34
Breakpoint 2
 at /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.py:34
```

- Continuez l'exécution jusqu'au prochain point d'arrêt avec `c(ont(inue))` :

```
ipdb> c
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.py(34)iterated_wiener()
     33     """
2--> 34     noisy_img = noisy_img
     35     denoised_img = local_mean(noisy_img, size=size)
```

- Continuez dans le code avec `n(ext)` et `s(tep)` : `next` saute à la prochaine déclaration dans le contexte d'exécution courant, alors que `step` va traverser les contextes d'exécution, c'est-à-dire permettre l'exploration à l'intérieur des appels de fonction :

```
ipdb> s
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.py(35)iterated_wiener()
2    34     noisy_img = noisy_img
---> 35     denoised_img = local_mean(noisy_img, size=size)
     36     l_var = local_var(noisy_img, size=size)

ipdb> n
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.py(36)iterated_wiener()
     35     denoised_img = local_mean(noisy_img, size=size)
---> 36     l_var = local_var(noisy_img, size=size)
     37     for i in range(3):
```

- Exécutez pas à pas quelques lignes et explorez les variables locales :

```
ipdb> n
> /home/varoquau/dev/scipy-lecture-notes/advanced/optimizing/wiener_filtering.py(37)iterated_wiener()
     36     l_var = local_var(noisy_img, size=size)
---> 37     for i in range(3):
     38         res = noisy_img - denoised_img
ipdb> print l_var
[[5868 5379 5316 ..., 5071 4799 5149]
 [5013  363  437 ...,  346  262 4355]
 [5379  410  344 ...,  392  604 3377]
 ...,
 [ 435  362  308 ...,  275  198 1632]
 [ 548  392  290 ...,  248  263 1653]
 [ 466  789  736 ..., 1835 1725 1940]]
ipdb> print l_var.min()
0
```

Voici notre bogue : rien que des entiers et 0 variation. Nous faisons une levée d'exception integer arithmetic sur les erreurs numériques.

Quand nous lançons le fichier **wiener_filtering.py**, les avertissements suivants sont levés :

```
In [2]: %run wiener_filtering.py
wiener_filtering.py:40: RuntimeWarning: divide by zero encountered in divide
    noise_level = (1 - noise/l_var )
```

Nous pouvons transformer ces warnings en exception, ce qui nous permet de faire du débogage post-mortem, et trouver notre problème plus rapidement :

```
In [3]: np.seterr(all='raise')
Out[3]: {'divide': 'print', 'invalid': 'print', 'over': 'print', 'under': 'ignore'}
In [4]: %run wiener_filtering.py
---------------------------------------------------------------------------
FloatingPointError                        Traceback (most recent call last)
/home/esc/anaconda/lib/python2.7/site-packages/IPython/utils/py3compat.pyc in execfile(fname, *where)
    176             else:
    177                 filename = fname
--> 178             __builtin__.execfile(filename, *where)

/home/esc/physique-cuso-python-2013/scipy-lecture-notes/advanced/debugging/wiener_filtering.py in <module>()
     55 pl.matshow(noisy_face[cut], cmap=pl.cm.gray)
     56
---> 57 denoised_face = iterated_wiener(noisy_face)
     58 pl.matshow(denoised_face[cut], cmap=pl.cm.gray)
     59

/home/esc/physique-cuso-python-2013/scipy-lecture-notes/advanced/debugging/wiener_filtering.py in iterated_wiener
 size)
     38         res = noisy_img - denoised_img
     39         noise = (res**2).sum()/res.size
---> 40         noise_level = (1 - noise/l_var )
     41         noise_level[noise_level<0] = 0
     42         denoised_img += noise_level*res

FloatingPointError: divide by zero encountered in divide
```

3-1-3 - Autres façons de démarrer le débogueur

- **Déclenchement d'une exception comme un point d'arrêt du pauvre :**
  Si vous trouvez fastidieux de noter le numéro de ligne pour poser un point d'arrêt, vous pouvez simplement lever une exception au point que vous souhaitez inspecter et utiliser la fonction `%debug` de Ipython. Notez que dans ce cas, vous ne pouvez pas faire du pas-à-pas et continuez l'exécution.
- **Test de débogage d'échec en utilisant nosetests**

Vous pouvez lancer `nosetests --pdb` pour faire un saut dans des exceptions en mode post-mortem, et `nosetests --pdb-failure` pour inspecter les échecs de test en utilisant le débogueur.
En outre, vous pouvez utiliser l'interface d'IPython pour le débogueur dans nose en installant le plugin nose **ipdbplugin**. Vous pouvez passer les options `--ipdb` et `--ipdb-failure` à nosetests.

- **Appeler explicitement le débogueur**
  Insérez la ligne suivante à l'endroit où vous voulez stopper le débogueur :

```
import pdb; pdb.set_trace()
```


Lors du lancement de nosetests, la sortie est capturée, et ainsi, il semble que le débogueur ne fonctionne pas. Lancez simplement les nosetests avec l'option -s.

Débogueurs graphiques et alternatifs

- Pour faire du pas-à-pas à travers le code et inspecter les variables, vous trouverez peut-être plus pratique d'utiliser un débogueur graphique tel que **winpdb**.
- Sinon, **pudb** est un bon débogueur semi-graphique avec une interface utilisateur texte dans la console.
- Le projet **pydbgr**, également, vaut probablement le coup qu'on y jette un œil.

3-1-4 - Commandes de débogage et interactions

l(list)	Liste le code à la position courante
u(p)	Monte à la pile d'appel
d(own)	Descend à la pile d'appel
n(ext)	Exécute la prochaine ligne (ne va pas à l'intérieur d'une nouvelle fonction)
s(tep)	Exécute la prochaine déclaration (va à l'intérieur d'une nouvelle fonction)
bt	Affiche la pile d'appel
a	Affiche les variables locales
!command	Exécute la commande **Python** donnée (par opposition à une commande pdb)

Les commandes de débogage ne sont pas du code Python.

Vous ne pouvez pas appeler les variables comme vous le voulez. Par exemple, si vous ne pouvez pas surcharger les variables dans la structure en cours avec le même nom, ***utilisez des noms différents de vos variables locales lors de la saisie de code dans votre débogueur.***

3-1-5 - Obtenir de l'aide en cours de débogage

Tapez `h` ou `help` pour accéder à l'aide interactif :

```
ipdb> help

Documented commands (type help <topic>):
========================================
EOF     bt          cont      enable  jump  pdef   r        tbreak   w
a       c           continue  exit    l     pdoc   restart  u        whatis
```

```
alias   cl          d          h       list   pinfo   return    unalias   where
args    clear       debug      help    n      pp      run       unt
b       commands    disable    ignore  next   q       s         until
break   condition   down       j       p      quit    step      up

Miscellaneous help topics:
==========================
exec  pdb

Undocumented commands:
======================
retval  rv
```

4 - Débogage Segmentation fault en utilisant GDB

Si vous avez une erreur segmentation fault, vous ne pouvez pas la déboguer avec pdb, puisqu'elle crash l'interpréteur Python avant qu'il ne puisse passer dans le débogueur. De même, si vous avez un bug dans du code C embarqué dans Python , pdb est inutilisable. Pour cela, nous nous tournons vers le débogueur GNU **gdb**, disponible sur Linux.

Avant que nous commencions avec gdb, ajoutons-lui quelques outils spécifiques à Python. Pour cela, nous ajoutons quelques macros à notre fichier ~/.gbdinit. Le choix optimal de macro dépend de votre version de Python et de gdb. J'ai ajouté une version simplifiée dans **gdbinit**, mais n'hésitez pas à lire **DebuggingWithGdb**.

Pour déboguer le script Python **segfault.py** avec gdb, nous pouvons lancer le script dans gdb comme suit :

```
$ gdb python
...
(gdb) run segfault.py
Starting program: /usr/bin/python segfault.py
[Thread debugging using libthread_db enabled]

Program received signal SIGSEGV, Segmentation fault.
_strided_byte_copy (dst=0x8537478 "\360\343G", outstrides=4, src=
    0x86c0690 <Address 0x86c0690 out of bounds>, instrides=32, N=3,
    elsize=4)
        at numpy/core/src/multiarray/ctors.c:365
365             _FAST_MOVE(Int32);
(gdb)
```

Nous obtenons un segfault, et gdb le capture pour du débogage post-mortem dans la pile niveau C (pas la pile d'appel Python). Nous pouvons déboguer la pile C en utilisant les commandes gdb :

```
(gdb) up
#1  0x004af4f5 in _copy_from_same_shape (dest=<value optimized out>,
    src=<value optimized out>, myfunc=0x496780 <_strided_byte_copy>,
    swap=0)
at numpy/core/src/multiarray/ctors.c:748
748         myfunc(dit->dataptr, dest->strides[maxaxis],
```

Comme vous pouvez le voir maintenant, vous êtes dans le code C de numpy. Nous voudrions savoir ce qui déclenche ce segfault dans le code Python, nous montons donc dans la pile jusqu'à atteindre la boucle d'exécution Python :

```
(gdb) up
#8  0x080ddd23 in call_function (f=
    Frame 0x85371ec, for file /home/varoquau/usr/lib/python2.6/site-packages/numpy/core/
arrayprint.py, line 156, in _leading_trailing (a=<numpy.ndarray at remote 0x85371b0>, _nc=<module
 at remote 0xb7f93a64>), throwflag=0)
    at ../Python/ceval.c:3750
3750    ../Python/ceval.c: No such file or directory.
        in ../Python/ceval.c

(gdb) up
#9  PyEval_EvalFrameEx (f=
```

```
    Frame 0x85371ec, for file /home/varoquau/usr/lib/python2.6/site-packages/numpy/core/
arrayprint.py, line 156, in _leading_trailing (a=<numpy.ndarray at remote 0x85371b0>, _nc=<module
 at remote 0xb7f93a64>), throwflag=0)
    at ../Python/ceval.c:2412
2412    in ../Python/ceval.c
(gdb)
```

Une fois que nous sommes dans la boucle d'exécution de Python, nous pouvons utiliser notre fonction Python spéciale d'assistance. Par exemple, nous pouvons trouver le code correspondant suivant :

```
(gdb) pyframe
/home/varoquau/usr/lib/python2.6/site-packages/numpy/core/arrayprint.py (158): _leading_trailing
(gdb)
```

Ceci est du code numpy, nous avons besoin de remonter jusqu'à trouver du code que nous avons écrit :

```
(gdb) up
...
(gdb) up
#34 0x080dc97a in PyEval_EvalFrameEx (f=
    Frame 0x82f064c, for file segfault.py, line 11, in
 print_big_array (small_array=<numpy.ndarray at remote 0x853ecf0>, big_array=<numpy.ndarray at
 remote 0x853ed20>), throwflag=0) at ../Python/ceval.c:1630
1630    ../Python/ceval.c: No such file or directory.
        in ../Python/ceval.c
(gdb) pyframe
segfault.py (12): print_big_array
```

Le code correspondant est :

```
def make_big_array(small_array):
    big_array = stride_tricks.as_strided(small_array,
                                         shape=(2e6, 2e6), strides=(32, 32))
    return big_array

def print_big_array(small_array):
    big_array = make_big_array(small_array)
```

Ainsi, le segfault arrive lors de l'affichage de big_array[-10:]. La raison est simplement que big_array a été alloué avec son extrémité hors de la mémoire du programme.

Pour une liste des commandes spécifiques Python définies dans gdbinit, lisez la source de ce fichier.

Exercice de conclusion

Le script suivant est bien documenté et nous espérons qu'il est lisible. Il vise à répondre au problème d'intérêt réel pour le calcul numérique, mais ne fonctionne pas. Pouvez-vous le déboguer ?

Code source Python : to_debug.py

```
"""
A script to compare different root-finding algorithms.

This version of the script is buggy and does not execute. It is your task
to find an fix these bugs.

The output of the script sould look like:

    Benching 1D root-finder optimizers from scipy.optimize:
                brenth:   604678 total function calls
```

```
                brentq:   594454 total function calls
                ridder:   778394 total function calls
                bisect:  2148380 total function calls
"""
from itertools import product

import numpy as np
from scipy import optimize

FUNCTIONS = (np.tan,  # Dilating map
             np.tanh, # Contracting map
             lambda x: x**3 + 1e-4*x, # Almost null gradient at the root
             lambda x: x+np.sin(2*x), # Non monotonous function
             lambda x: 1.1*x+np.sin(4*x), # Fonction with several local maxima
            )

OPTIMIZERS = (optimize.brenth, optimize.brentq, optimize.ridder,
              optimize.bisect)


def apply_optimizer(optimizer, func, a, b):
    """ Return the number of function calls given an root-finding optimizer,
        a function and upper and lower bounds.
    """
    return optimizer(func, a, b, full_output=True)[1].function_calls,


def bench_optimizer(optimizer, param_grid):
    """ Find roots for all the functions, and upper and lower bounds
        given and return the total number of function calls.
    """
    return sum(apply_optimizer(optimizer, func, a, b)
               for func, a, b in param_grid)


def compare_optimizers(optimizers):
    """ Compare all the optimizers given on a grid of a few different
        functions all admitting a signle root in zero and a upper and
        lower bounds.
    """
    random_a = -1.3 + np.random.random(size=100)
    random_b =   .3 + np.random.random(size=100)
    param_grid = product(FUNCTIONS, random_a, random_b)
    print "Benching 1D root-finder optimizers from scipy.optimize:"
    for optimizer in OPTIMIZERS:
        print '% 20s: % 8i total function calls' % (
                    optimizer.__name__,
                    bench_optimizer(optimizer, param_grid)
                )


if __name__ == '__main__':
    compare_optimizers(OPTIMIZERS)
```

5 - Remerciements

Nous tenons à remercier **Chrtophe** pour la traduction, **Wiztricks** pour la relecture technique et **Milkoseck** pour la correction orthographique.