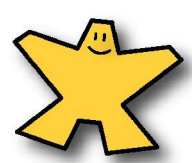


> Python: meilleures pratiques de développement



Stefane Fermigier
<sf@nuxeo.com>

08/07/2005



Qui je suis

- ♦ Développeur Python depuis 1995
- ♦ Contributeur (très occasionnel) à Python et Zope depuis 1998
- ♦ Fondateur et PDG de Nuxeo depuis 2000
 - ♦ Société de 25 personnes basée à Paris
 - ♦ Spécialistes de la gestion de contenu d'entreprises (ECM) et du travail collaboratif
 - ♦ Développeurs de CPS, plateforme libre d'ECM
 - ♦ Clients: grands comptes français et internationaux



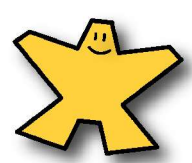
Python – langage de programmation agile

- ♦ (Very) High level language
 - ♦ Expressivité
 - ♦ Lisibilité du code (« *executable pseudocode* »)
 - ♦ Structures de données de haut niveau (listes, dictionnaires)
 - ♦ Optimiser le temps de développement plutôt que le temps d'exécution
 - ♦ Typage dynamique
 - ♦ Duck typing (« *if it quacks like a duck, then it's a duck* »)
 - ♦ Bénéfices: flexibilité et réutilisation
- ♦ Équilibre entre simplicité et puissance



Python, pour quoi faire ?

- ♦ Scripting système ou d'applications complexes
 - ♦ Outils pour générer du des bindings vers C, C++, Fortran, Java, .NET, ObjectiveC...
 - ♦ Exemples: SWIG, SIP, Boost.Python, CXX, Pyrex, JPytype...
- ♦ Utilisations
 - ♦ Scripting de type « shell » (10-50 lignes)
 - ♦ Tests automatiques
 - ♦ Couplage lâche de composants



Python, pour quoi faire ?

- ♦ Prototypage rapide d'applications
 - ♦ Productivité entre 3x et 10x celle de langages non-agiles (selon Bruce Eckel)
 - ♦ $\text{Effort} = (\text{SLOC})^{1.5}$ (loi de Brooks)
 - ♦ $\text{SLOC}(\text{Python}) = \text{SLOC}(\text{C})/4$ ou $\text{SLOC}(\text{Java})/2.5$
 - ♦ *Time to market*, développement évolutif
 - ♦ Certaines parties peuvent être ultérieurement réécrites dans un langage non-agile
- ♦ Applications complètes de grande envergure
 - ♦ Ex: CPS, ERP5, Chandler...



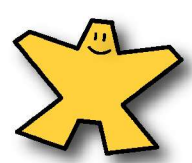
Import this !

The screenshot shows a terminal window titled "fermigier@localhost: /home/fermigier - Shell No. 4 - Kor". The window has a menu bar with "Session", "Edit", "View", "Bookmarks", "Settings", and "Help". The main text area displays the output of the command "import this", which is the Zen of Python by Tim Peters. The text is as follows:

```
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>> █
```

At the bottom of the window, there is a taskbar with icons for "Shell", "Shell No. 2", "Shell No. 3", and "Shell No. 4".



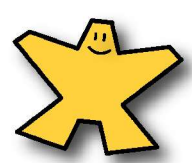
Objectifs

- ♦ Optimiser 3 variables:
 - ♦ Qualité
 - ♦ Durée
 - ♦ Effort (= prix)
- ♦ À propos du prix
 - ♦ Années 50: machines = \$\$\$, informaticiens = gratuits (IBM: « achetez un mainframe, on vous offre 2 informaticiens »)
 - ♦ Années 2000: informaticiens = \$\$\$, machines = quasi-gratuites



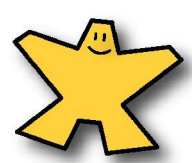
No silver bullet

- ♦ « *Also, it was observed that switching from machine code to a high-level programming language did not guarantee all the benefits that were hoped for. In particular, programmers still produced, as willingly as before, large chunks of ununderstable code, the only difference being that now they did it in a more grandiose scale, and that high-level bugs had replaced low-level ones* »
 - ♦ E.W. Dijkstra



Critères de qualité

- ♦ Critères externes
- ♦ Critères internes



Critères externes

- ♦ Correction (« couverture du périmètre fonctionnel »)
- ♦ Performances
- ♦ Fiabilité
- ♦ Robustesse, intégrité, sécurité
- ♦ Utilisabilité
- ♦ Compatibilité, interopérabilité

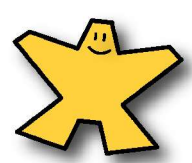


Critères internes

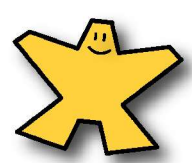
- ♦ Adhérence à un ou des standards de programmation
- ♦ Élégance, simplicité
- ♦ Compréhensibilité, extensibilité, maintenabilité
 - ♦ « *Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.* »
John F. Woods
- ♦ Testabilité, vérifiabilité
- ♦ Réutilisabilité, mutualisation



Quelques principes

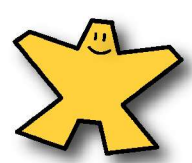


- ♦ « *When writing a program, our audience is not the computer, our audience is human* »
 - ♦ Stefan Holek
- ♦ Un programme passe souvent 80% de sa vie en phase de maintenance



Développement incrémental / agile

- ♦ « *That which remains quiet, is easy to handle.*
- ♦ *That which is not yet developed is easy to manage.*
- ♦ *That which is weak is easy to control.*
- ♦ *That which is still small is easy to direct.*
- ♦ *Deal with little troubles before they become big.*
- ♦ *Attend to little problems before they get out of hand.*
- ♦ *For the largest tree was once a sprout,*
- ♦ *the tallest tower started with the first brick,*
- ♦ *and the longest journey started with the first step. »*
 - ♦ Lao Tseu, Tao Te Ching (500 a.j.c)



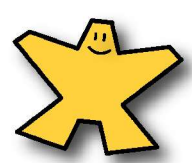
Développement incrémental / agile

- ♦ Les méthodologies agiles visent à livrer rapidement les fonctionnalités les plus importantes pour le client
- ♦ Elles encouragent le dialogue avec le client et l'ouverture au changement
- ♦ 2 piliers au niveau de la programmation:
 - ♦ Développement *test-driven*
 - ♦ *Refactoring*



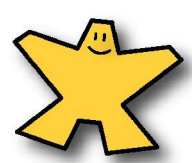
Test-driven

- ♦ Dans sa forme la plus pure
 - ♦ On écrit un test
 - ♦ Il ne passe pas
 - ♦ On écrit la fonctionnalité
 - ♦ Le test passe
- ♦ Formes dégénérées
 - ♦ On écrit les tests à peu près en même temps que le code
 - ♦ On vérifie que la couverture des tests est conforme aux objectifs de qualité avec des outils appropriés



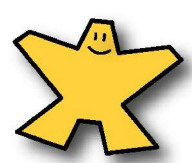
Refactoring

- ♦ Identification des « *bad smells* », par ex:
 - ♦ Mauvais nommage
 - ♦ Duplication de code (DRY, OAOO)
 - ♦ Classe / méthodes trop longues, trop complexes
 - ♦ Manque de clarté
 - ♦ Manque de cohésion
 - ♦ Couplage trop fort
 - ♦ ...



Refactoring

- ♦ Modifications incrémentales en s'appuyant sur les tests unitaires
 - ♦ Déplacement de code (d'une méthode à l'autre, d'une classe à l'autre)
 - ♦ Renommage
 - ♦ Changements d'algorithmes
 - ♦ ...

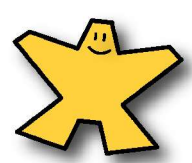


Problèmes avec le refactoring

- ♦ Difficile de changer une API une fois qu'elle est utilisée par des logiciels tiers
- ♦ D'où vieilles API qui doivent être maintenues, *bit rot*, etc.



Best practices Python



Idiomes de programmation Python

- ♦ Cf. par exemple le *Python cookbook*
- ♦ Ex:
 - ♦ Good: `for x in l: f(x)`
 - ♦ Bad: `for i in range(0, len(l)): f(l[i])`
- ♦ Ex:
 - ♦ Good: `s = l.join("")`
 - ♦ Bad: `for x in l: s += x`
- ♦ Ex:
 - ♦ `if __name__ == '__main__': ...`

- ♦ Indentation = 4 caractères, pas de tabs
- ♦ 1 instruction par ligne
- ♦ Pas plus de 80 caractères par ligne
- ♦ Couper les expressions trop longues en sous-expressions, éviter les continuations de ligne
- ♦ Variables = small_caps
- ♦ Classe = MixedCase
- ♦ Méthodes = mixedCase
- ♦ (Pseudo)constantes = ALL_CAPS



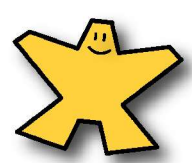
Conventions de codage + métaphores

- ♦ Extension de PEP8 car celui-ci ne couvre pas tout
- ♦ Besoin de s'adapter en fonction du contexte de chaque projet
- ♦ Vocabulaire, vision, métaphores communs au projet



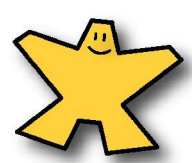
Nommage

- ♦ « *The Named is the Mother of all Things* »
 - ♦ Lao Tseu, Tao Te Ching
- ♦ Impact des *design patterns* = possibilité de **nommer** des choses qui restaient jusqu'à présent non-dites



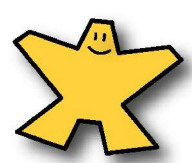
Nommage

- ♦ Utiliser l'anglais
- ♦ Ne pas mélanger les mots issus du domaine métier et les mots techniques (list, dict, etc.)
- ♦ Choisir des noms « pas trop courts, pas trop longs »
 - ♦ Eviter les abbréviations (cnt, lst...)



Nommage des classes

- ♦ Le nom d'une classe doit clairement désigner sa raison d'être
- ♦ Si vous n'arrivez pas à trouver des bons noms pour vos classes, vous devez probablement revoir vos abstractions



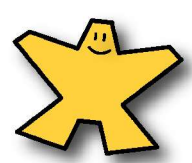
Nommage des méthodes

- ♦ Première partie du nom est un verbe qui désigne une action
 - ♦ addToCart(), trimWhitespace()...



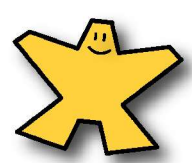
Examples

- ♦ Good: `shopping_cart.add(products, quantity=1)`
- ♦ Bad: `cart.addToCart(prod_list, 1)`
- ♦ Good: `to_pay = shopping_cart.total(currency)`
- ♦ Bad: `sum = cart_session.computeTotal(crcy)`



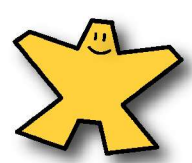
Commentaires

- ♦ En anglais
- ♦ Le commentaire explique le *pourquoi* du code, l'intention du programmeur
 - ♦ Ex: pourquoi tel algorithme plutôt que tel autre, dans quel
- ♦ Le *comment* doit être clair à la lecture du code, sinon = bad smell -> refactoring



Docstrings

- ♦ Les docstrings sont une forme de commentaire qui décrit le contrat que remplit une classe ou une méthode (ou éventuellement un objet)
- ♦ Peuvent raconter une histoire
- ♦ Très utile pour documenter une API
 - ♦ Ne doit pas être un substitut à un mauvais nommage des méthodes
- ♦ Elles peuvent facilement être manipulées par introspection
 - ♦ `myFunction.__doc__`
 - ♦ `help(myFunction)`
 - ♦ Pydoc, Happydoc



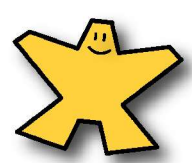
Docstrings

- ♦ Possibilité d'utiliser des conventions pour spécifier précisément le type des arguments
- ♦ Ont été détournées par la passé (avant l'introduction des décorateurs) pour ajouter des fonctions au langages
 - ♦ Ex: assertions de sécurité dans le anciennes versions de Python, programmation par contrats
 - ♦ Mais maintenant on a les décorateurs. c'est plus propre



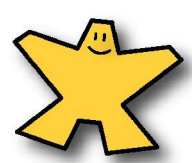
Connaitre les librairies Python

- ♦ Librairie standard
- ♦ Librairies non-standards
- ♦ Choisir le ou les bons frameworks pour le travail à faire
 - ♦ Ex: Zope, Twisted, PEAK...



Erreurs à éviter

- ♦ `from my_module import *`
- ♦ `try/except` non qualifié
- ♦ Fautes d'orthographe dans les API



Marqueurs (TODO, FIXME...)

- ♦ Permettent de repérer les passages à corriger ou à améliorer
- ♦ Ne pas abuser des FIXME, « don't live with broken windows »



Attributs vs. accesseurs

- ♦ Python ne décourage pas l'accès direct aux attributs
 - ♦ Ex: `obj.attr += 1`
- ♦ Ce n'est pas une « violation de l'encapsulation » comme le prétendent les intégristes de Java
- ♦ Possibilité de passer par des propriétés ultérieurement (refactoring)
 - ♦

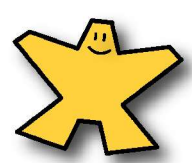
```
class C(object):  
    def getx(self): return self.__x  
    def setx(self, value): self.__x = value  
    def delx(self): del self.__x  
    x = property(getx, setx, delx, "I'm the 'x' property.")
```

- ♦ Tests unitaires
 - ♦ Niveau packages ou modules isolés les uns des autres
- ♦ Tests d'intégration
 - ♦ Tests de l'application dans son ensemble
 - ♦ Attaque directe de l'API
- ♦ Tests fonctionnels
 - ♦ Attaquent l'application de l'extérieur = *black box testing* (via HTTP pour les applications web ou outils de tests spécifiques pour les GUI)



Tests unitaires avec PyUnit

- ♦ Portage sous Python de SUnit (SmallTalk) et JUnit (Java) de Kent Beck & al
- ♦ Module standard (unittest) de la librairie Python
- ♦ Syntaxe un peu lourde, « unpythonic » selon certains, projets de remplacement (ex: py.test)



PyUnit - Principe

- ♦ Cas de tests regroupés dans des classes héritant de TestCase
- ♦ Méthodes testXXX() appelées les unes après les autres (pas d'ordre particulier)
- ♦ Méthodes setUp() et tearDown() appelées respectivement juste avant et juste après chaque test
- ♦ Cas de tests regroupés dans des TestSuites
- ♦ Tests dans sous-répertoire tests
- ♦ Selon XP/TDD, $SLOC(tests) = 3x SLOC(\text{code à tester})$



PyUnit – Exemple



The screenshot shows a terminal window titled "fermigier@localhost: /home/fermigier/slides/rmill-2005 -". The window contains a Python script for a unit test using unittest. The script defines a class A with a hello() method that returns "Hello World!". It then defines a class ATestCase that inherits from unittest.TestCase. ATestCase has a setUp() method that creates an instance of A, and a testHelloWorld() method that calls hello() and asserts the result is "Hello World!". Finally, it calls unittest.main(). The terminal shows the output "10,0-1" and "All".

```
fermigier@localhost: /home/fermigier/slides/rmill-2005 -
Session Edit View Bookmarks Settings Help

import unittest

class A:
    def hello(self):
        return "Hello World!"

class ATestCase(unittest.TestCase):
    def setUp(self):
        self.obj = A()

    def testHelloWorld(self):
        self.assertEqual(self.obj.hello(), "Hello World!")

unittest.main()

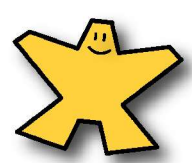
~
~

10,0-1 All
```



Tests fonctionnels

- ♦ Peuvent aussi utiliser PyUnit
 - ♦ Nécessitent une couche d'accès à l'application à tester (HTTP, CORBA...)
 - ♦ La lisibilité peut poser un problème
- ♦ Outils spécifiques clef en main
 - ♦ Functional doctests de Zope3
- ♦ Outils externes
 - ♦ Rational, PushToTest, Parasoft... (\$\$\$)
 - ♦ Maxq, TestMaker...

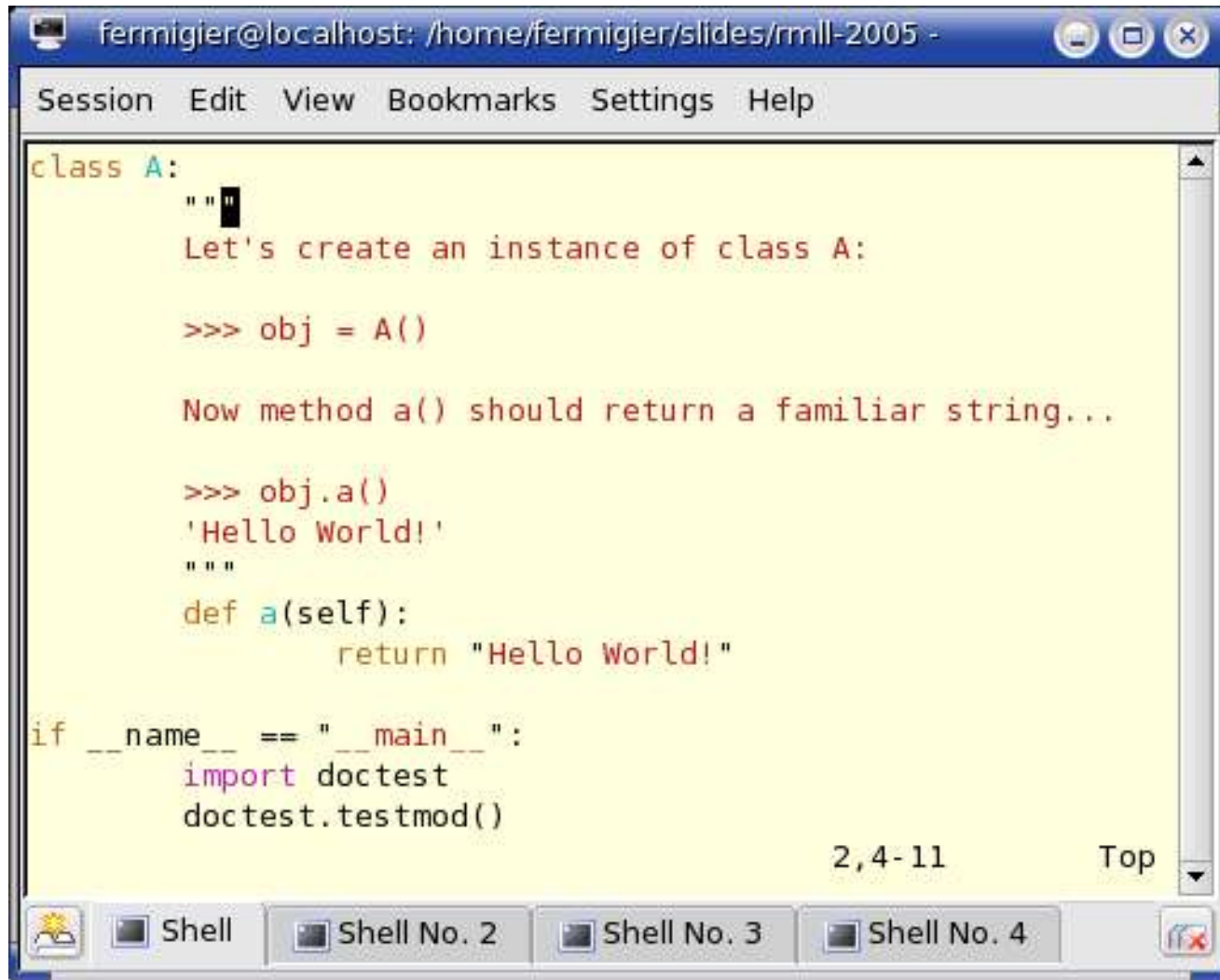


Doctests

- ♦ Tests unitaires (ou pas) insérés dans les docstrings
- ♦ Racontent une histoire
- ♦ Avantage: mélanger doc et tests, pousser à l'extrême une des idées de XP, on est « sûr » que les docs sont synchro avec le code



Doctests – Example



The screenshot shows a terminal window with a blue title bar. The title bar text is "fermigier@localhost: /home/fermigier/slides/rmll-2005 -". Below the title bar is a menu bar with "Session", "Edit", "View", "Bookmarks", "Settings", and "Help". The main area of the terminal has a yellow background and contains the following Python code:

```
class A:
    """
    Let's create an instance of class A:

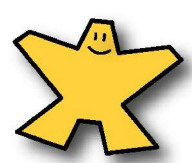
    >>> obj = A()

    Now method a() should return a familiar string...

    >>> obj.a()
    'Hello World!'
    """
    def a(self):
        return "Hello World!"

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

At the bottom right of the code area, there is a line number "2,4-11" and a "Top" link. The bottom of the terminal window has a taskbar with icons for "Shell", "Shell No. 2", "Shell No. 3", and "Shell No. 4".



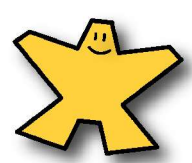
Liens tests et SCM

- ♦ Lancent les tests unitaires / fonctionnels de façon régulière
- ♦ Exemples
 - ♦ Mozilla tinderbox
 - ♦ Buildbot
 - ♦ Apache Gump

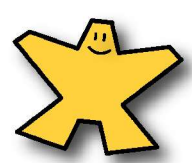


Contrôle automatique de qualité

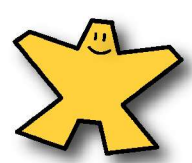
- ♦ Ex:
 - ♦ Pychecker
 - ♦ PyLint
- ♦ Savent vérifier certaines erreurs de base
 - ♦ Fautes de frappes
 - ♦ `import *`
 - ♦ Eventuellement non-respect de conventions
 - ♦ Complexité cyclomatique
- ♦ Pas mal de faux positifs en général
- ♦ On devrait pouvoir faire mieux (inférence de type par exemple)
 - ♦ Projet PyPy



Livraison / déploiement



- ♦ Outils de *build* et de *packaging* multiplateformes standard de Python
- ♦ Usage
 - ♦ `python setup.py build`
 - ♦ `python setup.py install`
 - ♦ `python setup.p bdist_rpm`
- ♦ Métadonnées et instructions de *build* exprimées sous forme procédurale et non déclarative



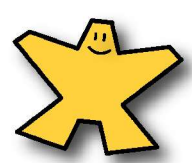
Python eggs

- ♦ Invention récente de Philip Eby
- ♦ Similaire aux .jar de Java
- ♦ Fichiers ZIP qui contiennent le code et les métadonnées
- ♦ Système de distribution global (à la CPAN, Ruby gems...) en cours d'élaboration



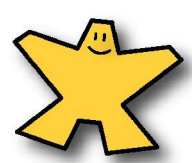
Optimisation

- ♦ Si nécessaire...
- ♦ Identifier les parties critiques à l'aide du profiler Python
- ♦ Les réécrire en C ou autre (éventuellement via Pyrex)
- ♦ Bonne pratique: maintenir en parallèle la version Python pure (qui sert de référence = executable pseudocode) et la version C
 - ♦ Ex: ElementTree et cElementTree



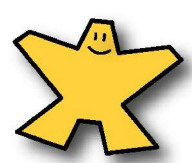
Méthodologie

- ♦ Outils de suivi
 - ♦ Bug / feature requests tracker
- ♦ Mesurer ce qui doit l'être
 - ♦ Couverture des tests unitaires et fonctionnels (coverage.py)
- ♦ *Pair programming* / inspections / revues de code
 - ♦ *Readability counts* (once again)



IDEs

- ♦ Eclipse + PyDev
- ♦ Eric
- ♦ Boa
- ♦ Wing (propriétaire et très bien)

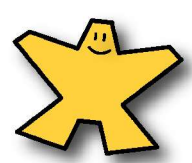


Zen of Python by Tim Peters



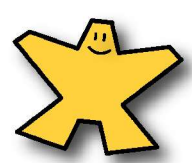
Zen of Python

- ♦ 1. Beautiful is better than ugly
 - ♦ On code pour les humains, pas pour les CPU
 - ♦ « *How do we convince people that in programming, simplicity and clarity – in short, what mathematicians call 'elegance' – are not a dispensable luxury, but a crucial mater that decides between success and failure* » E.W. Dijkstra
 - ♦ Beautiful = « pythonic », ugly = « unpythonic »
 - ♦ Il n'y a pas de substitut à l'expérience et au bon goût
 - ♦ Facteurs psychologiques : « fun », « flow »



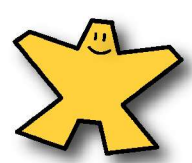
Zen of Python

- ♦ 2. Explicit is better than implicit
 - ♦ Limiter l'effort de mémoire du développeur
 - ♦ Attention: il est possible de changer plein de choses profondes et magiques dans le langage
 - ♦ Méthodes "magiques" (`__getattr__`, `__setattr__`, ...)
 - ♦ Métaclasses
 - ♦ Outils très puissants, mais ne pas en abuser !



Zen of Python

- ♦ 3. Complex is better than complicated
 - ♦ Certains problèmes sont intrinsèquement complexes
 - ♦ Exemple: si on doit s'interfacer ou implémenter des standards existants



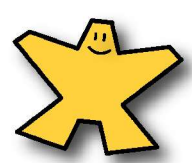
Zen of Python

- ♦ 4. Flat is better than nested
- ♦ 5. Sparse is better than dense
 - ♦ Lisibilité encore
 - ♦ Dictionnaires plutôt que tableaux



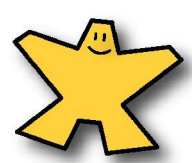
Zen of Python

- ♦ 6. Readability counts
 - ♦ « Readable is better than unreadable »
 - ♦ See #1



Zen of Python

- ♦ 7. Special cases aren't special enough to break the rules
 - ♦ La cohérence comme critère de simplicité
- ♦ 8. Although practicality beats purity
 - ♦ La simplicité au service du pragmatisme



Zen of Python

- ♦ 9. Errors should never pass silently
 - ♦ Traîter correctement les exceptions
- ♦ 10. Unless explicitly silenced
 - ♦ Pas d'excepts non qualifiés
 - ♦ Grosse erreur récurrente dans les projets Zope



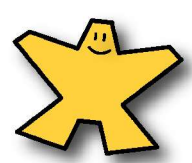
Zen of Python

- ♦ 11. In the face of ambiguity, refuse the temptation to guess
 - ♦ Simplifier la compréhension
 - ♦ Ex: arguments nommés
 - ♦ `frob(1, 0, 3)` vs `frob(verbose=1, gui=0, debug=3)`
- ♦ 12. There should be one-- and preferably only one --obvious way to do it
 - ♦ Opposé de la philosophie Perl
- ♦ 13. Although that way may not be obvious at first unless you're Dutch
 - ♦ Guido = BDFL



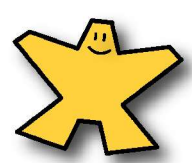
Zen of Python

- ♦ 14. Now is better than never
 - ♦ Prototypage rapide, développement évolutif
- ♦ 15. Although never is often better than *right* now
 - ♦ S'applique au langage (ou aux frameworks) plus qu'aux applications
 - ♦ Résister à l'introduction de nouvelles fonctionnalités tant qu'il n'y a pas de choix absolument clair



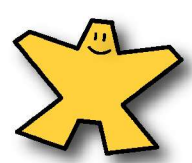
Zen of Python

- ♦ 16. If the implementation is hard to explain, it's a bad idea
- ♦ 17. If the implementation is easy to explain, it may be a good idea
 - ♦ La simplicité de l'implémentation n'est pas le seul critère



Zen of Python

- ♦ 18. Namespaces are one honking great idea -- let's do more of those!
 - ♦ Les dictionnaires comme couteau suisse du développeur
 - ♦ Contredit un peu PZ #4



- ♦ **Pythonic is better than unpythonic**



Références / Thanks to

- ♦ Kent Beck, *Extreme Programming Explained* and *Test Driven Development*
- ♦ Martin Fowler, *Refactoring*
- ♦ Stefan « rock star » Holek, *Choosing Good Names*
- ♦ Alex Martelli *et al*, *The Python Cookbook*
- ♦ David Ascher, *Dynamic Languages — ready for the next challenges, by design*
- ♦ Frederic Brooks, *The Mythical Man-Month*
- ♦ Guido van Rossum, Tom Gilb, Steve Pemberton, Tim Peters, Bruce Eckel...