Python Guide Documentation

Version 0.0.1

Kenneth Reitz

23 February 2017

Table des matières

Salutations, Terriens ! Bienvenue dans le Guide de l'auto-stoppeur Python.

C'est un document vivant, qui respire. Si vous voulez contribuer, forkez-le sur GitHub !

Ce guide artisanal existe pour fournir aux développeurs novices comme experts un manuel des meilleurs pratiques pour l'installation, la configuration et l'usage de Python au quotidien.

Ce guide est **arrêté dans ses opinions** d'une manière qui est presque mais pas toute fait complètement *différente* de la documentation officielle de Python. Vous ne trouverez pas une liste de tous les frameworks web Python disponibles ici. Vous trouverez plutôt une liste concise et agréable d'options fortement recommandées.

Commençons ! Mais en premier, assurons-nous que vous savez où est votre serviette (NDT : cette citation est liée au livre "le guide de l'auto-stoppeur intergalactique")

CHAPITRE 1

Bien démarrer avec Python

Nouveau sur Python ? Commençons par configurer correctement votre environnement Python.

Choisir un interpréteur

L'état de Python (2 vs 3)

Quand vous choisissez un interpréteur Python, une question imminente est toujours présente : "Dois-je choisir Python 2 ou Python 3" ? La réponse n'est pas toujours aussi évidente que l'on pourrait penser.

L'essentiel à retenir de l'état des choses est comme suivant :

1. Python 2.7 a été le standard depuis un temps assez *long*.
2. Python 3 a introduit des changements majeurs dans le langage, avec lesquels de nombreux développeurs ne sont pas contents.
3. Python 2.7 recevra des mises à jour de sécurité nécessaires jusqu'en 2020 [5].
4. Python 3 évolue constamment, comme Python 2 l'a fait dans le passé.

Ainsi, vous pouvez maintenant voir pourquoi ce n'est pas une décision aussi facile.

Recommandations

Je vais être franc :

Utilisez Python 3 si...

— Vous vous en moquez.
— Vous adorez Python 3.
— Vous êtes indifférent vis à vis de 2 vs 3.
— Vous ne savez pas lequel utiliser.
— Vous embrassez le changement.

Utilisez Python 2 si...

— Vous adorez Python 2 et êtes attristé par le fait que le futur soit Python 3.
— Les prérequis de stabilité de votre logiciel seraient améliorés par un langage et un moteur d'exécution qui ne change jamais.
— Un logiciel dont vous dépendez le nécessite.

5. https://www.python.org/dev/peps/pep-0373/#id2

Ainsi... 3 ?

Si vous choisissez un interpréteur Python à utiliser, et que vous n'avez pas d'opinions arrêtées, alors je vous recommande d'utiliser la dernière version de Python 3.x, comme chaque version apporte des modules de la bibliothèque standard nouveaux et améliorés, des corrections de bug et de sécurité. Le progrès est le progrès.

Avec ces éléments, utilisez seulement Python 2 si vous avez une raison importante pour cela, comme une bibliothèque exclusive à Python 2 qui n'a pas d'alternative adéquate disponible en Python 3 , ou vous (comme moi) adorez absolument et êtes inspirés par Python 2.

Consultez Can I Use Python 3? pour voir si un logiciel dont vous dépendez peut bloquer votre adoption de Python 3.

Plus de lecture (en)

Il est possible d'écrire du code qui fonctionne sur Python 2.6, 2.7, et sur Python 3. Cela varie de problèmes triviaux à difficile selon le genre de logiciel que vous écrivez ; si vous êtes un débutant, il y a des choses beaucoup plus importantes à vous soucier.

Implémentations

Quand les gens parlent de *Python*, ils veulent souvent dire pas seulement le langage mais aussi l'implémentation Cython. *Python* est actuellement une spécification pour un langage qui qui peut être implémentée de multiples façons.

CPython

CPython est l'implémentation de référence de Python, écrite en C. Il compile le code Python en un bytecode intermédiaire qui est ensuite interprété par une machine virtuelle. CPython fournit le plus important niveau de compatibilité avec les paquets Python et les modules avec extension en C.

Si vous écrivez du code Python open-source et que vous voulez toucher une audience la plus large possible, le ciblage de CPython est le meilleur à faire. Pour utiliser des paquets qui s'appuient sur des extensions C pour fonctionner, CPython est votre seule option de mise en œuvre.

Toutes les versions du langage Python sont implémentées en C parce que CPython est l'implémentation de référence.

PyPy

PyPy est un interpréteur Python implémenté dans un sous-ensemble restreint statiquement typé du langage Python appelé RPython. L'interpréteur se comporte comme un compilateur "just-in-time" (JIT) et supporte de multiples backends (C, CLI, JVM).

PyPy a pour but une compatibilité maximum avec l'implémentation de référence CPython tout en améliorant les performances.

Si vous cherchez à améliorer les performances de votre code Python, cela vaut la peine d'essayer PyPy. Sur une suite de benchmarks, il est actuellement plus de 5 fois plus rapide que CPython.

PyPy supporte Python 2.7. PyPy3 [1], sorti en bêta, cible Python 3.

Jython

Jython est une implémentation Python qui compile du code Python en bytecode Java qui est ensuite exécuté par la JVM (Java Virtual Machine). De plus, il est capable d'importer et d'utiliser n'importe quelle classe Java comme un module Python.

1. http://pypy.org/compat.html

Si vous avez besoin de vous interfacer avec une base de code Java existante ou avez d'autres raisons d'écrire du code Python pour la JVM, Jython est le meilleur choix.

Jython supporte actuellement jusqu'à la version Python 2.7. [2]

IronPython

IronPython est une implémentation de Python pour le framework .NET. Il peut utiliser à la fois les bibliothèques des frameworks Python et .NET, et peut aussi exposer du code Python vers d'autres langages dans le framework .NET.

Python Tools for Visual Studio intègre IronPython directement dans l'environnement de développement de Visual Studio, ce qui le rend choix idéal pour les développeurs Windows.

IronPython supporte Python 2.7. [3]

PythonNet

Python for .NET est un paquet qui fournit une intégration presque transparente d'une installation native de Python avec le Common Language Runtime (CLR) de .NET . C'est un approche inverse que celle qui est prise par IronPython (voir ci-dessus), qui est plus complémentaire que concurrente de cette dernière.

En association avec Mono, PythonNet permet des installations natives de Python sur des systèmes d'exploitation non-Windows, comme OS X et Linux, pour fonctionner dans le framework .NET. Il peut être exécuté en plus d'IronPython sans conflit.

PythonNet supporte de Python 2.3 jusqu'à Python 2.7. [4]
— Installer Python correctement

1.2 Installer Python correctement

Il y a une bonne chance que vous ayez déjà Python sur votre système d'exploitation.

Si c'est le cas, vous n'avez pas besoin d'installer ou de configurer quoi que ce soit pour utiliser Python. Cela dit, je recommande fortement que vous installiez les outils et les bibliothèques décrites dans les guides ci-dessous avant que vous démarriez à construire des applications Python pour un usage dans le monde réel. En particulier, vous devriez toujours installer Setuptools, Pip, et Virtualenv — elles vous rendent tellement plus faciles d'utiliser d'autres bibliothèques Python tierces.

Guides d'installation

Ces guides passent en revue l'installation correcte de *Python 2.7* pour des objectifs de développement, aussi bien que la configuration de setuptools, pip, et virtualenv.
— *Mac OS X*.
— *Microsoft Windows*.
— *Ubuntu Linux*.

Installation de Python sous Mac OS X

La dernière version de Mac OS X, El Capitan, **vient avec Python 2.7 par défaut**.

Vous n'avez pas besoin d'installer ou de configurer quoi que ce soit pour utiliser Python. Cela dit, je recommande fortement que vous installiez les outils et les bibliothèques décrites dans la prochaine section

2. https://hg.python.org/jython/file/412a8f9445f7/NEWS
3. http://ironpython.codeplex.com/releases/view/81726
4. http://pythonnet.github.io/readme.html

avant que vous démarriez à construire des applications Python pour un usage dans le monde réel. En particulier, vous devriez toujours installer Setuptools, comme il rend plus facile pour vous d'utiliser d'autres bibliothèques Python tierces.

La version de Python qui est livrée avec OS X est bien pour apprendre mais elle n'est pas adaptée pour le développement. La version livrée avec OS X peut être dépassée par rapport à la version officielle courante de Python, qui est considérée comme la version de production stable.

Bien faire les choses

Installons une vraie version de Python.

Avant l'installation de Python, vous devrez avoir installé GCC. GCC peut être obtenu en téléchargeant Xcode, en plus léger, Command Line Tools (vous devez avoir un compte Apple) ou bien le paquet encore plus petit'OSX-GCC-Installer <https://github.com/kennethreitz/osx-gcc-installer#readme>'_.

Note : Si vous avez déjà Xcode installé, n'installez pas OSX-GCC-Installer. En association, le logiciel peut causer des problèmes qui sont difficiles à diagnostiquer.

Note : Si vous effectuez une installation fraîche de Xcode, vous devrez aussi ajouter les outils de la ligne de commande en exécutant `xcode-select --install` dans le terminal.

Bien que OS X vienne avec de nombreux utilitaires UNIX, ceux familiers avec les systèmes d'exploitation Linux remarquerons qu'un composant clé est manquant : un gestionnaire de paquet décent. Homebrew comblera ce manque.

Pour installer Homebrew, ouvrez le `Terminal` ou votre émulateur de terminal OSX favori et exécutez

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/insta
```

Ce script va expliquer quels changements il fera et vous demander avant que l'installation commence. Un fois que vous avez installé Homebrew, insérez le répertoire Homebrew en haut de votre variable d'environnement `PATH`. Vous pouvez faire ceci en ajoutant la ligne qui suit à la fin de votre fichier `~/.profile`

```
export PATH=/usr/local/bin:/usr/local/sbin:$PATH
```

Maintenant, nous pouvons installer Python 2.7 :

```
$ brew install python
```

Cela va prendre une minute ou deux.

Setuptools & Pip

Homebrew installe Setuptools et `pip` pour vous.

Setuptools vous permet de télécharger et d'installer n'importe quel logiciel conforme à Python via un réseau (généralement Internet) avec une seule commande (`easy_install`). Il vous permet aussi d'ajouter cette capacité d'installation par le réseau à votre propre logiciel Python avec très peu de travail.

`pip` est un outil pour installer et gérer facilement des paquets Python, qui est recommandé plutôt que `easy_install`. Il est supérieur à `easy_install` de plusieurs manières, et est activement maintenu.

Environnements virtuels

Un environnement virtuel est un outil pour garder les dépendances requises par différents projets dans des emplacements séparés, en créant des environnements virtuels Python pour eux. Il résout le dilemme "le projet X dépend de la version 1.x mais le projet Y nécessite la 4.x", et garde votre répertoire site-packages global propre et gérable.

Par exemple, vous pouvez travailler sur un projet qui nécessite Django 1.3 tout en maintenant aussi un projet qui nécessite Django 1.0.

Pour commencer à utiliser cela et plus d'informations : documentation sur les *environnements virtuels*.

Cette page est une version remixée d'un autre guide, qui est disponible sous la même licence.

Installation de Python sous Windows

En premier, téléchargez la dernière version de Python 2.7 depuis le site web officiel. Si vous voulez vous assurer que vous installez une version complètement à jour, cliquez sur le lien Downloads > Windows depuis la page d'accueil du site web Python.org .

La version Windows est fournie comme un paquet MSI. Pour l'installer manuellement, faites juste un double-clic sur le fichier. Le format de fichier MSI permet aux administrateurs Windows d'automatiser l'installation avec leurs outils standards.

Par sa conception, Python installe un répertoire avec le numéro de version intégré, c'est à dire que la version 2.7 de Python s'installera dans `C:\Python27\`. Ainsi, vous pouvez avoir de multiples version de Python sur le même système, sans conflits. Bien sûr, un seule interpréteur peut être assigné comme application par défaut pour les fichiers de type Python. Il ne modifie pas non plus automatiquement la variable d'environnement `PATH`, si bien que vous avez toujours le contrôle sur quelle copie de Python est exécutée.

Taper le chemin complet de l'interpréteur Python chaque fois devient rapidement lassant. Ajoutez ainsi les répertoires pour votre version de Python par défaut à la variable d'environnement `PATH`. En assumant que votre installation Python est dans `C:\Python27\`, ajoutez ceci à votre `PATH` :

```
C:\Python27\;C:\Python27\Scripts\
```

Vous pouvez faire cela facilement en exécutant ce qui suit dans `powershell` :

```
[Environment]::SetEnvironmentVariable("Path", "$env:Path;C:\Python27\;C:\Python27\Scripts\", "Us
```

Le second répertoire (`Scripts`) reçoit des fichiers de commande quand certains paquets sont installés. C'est ainsi un ajout très utile. Vous n'aurez pas besoin d'installer ou de configurer quoi que ce soit d'autre pour utiliser Python. Cela dit, je recommande fortement que vous installiez les outils et les bibliothèques décrites dans la prochaine section avant que vous démarriez à construire des applications Python pour un usage dans le monde réel. En particulier, vous devriez toujours installer Setuptools, comme il rend plus facile pour vous d'utiliser d'autres bibliothèques Python tierces.

Setuptools + Pip

Le logiciel tiers de Python le plus crucial de tous est Setuptools, qui étend les capacités de packaging et d'installation fournies par distutils dans la bibliothèque standard. Une fois que vous avez ajouté Setuptools à votre système Python, vous pouvez télécharger et installer n'importe quel produit logiciel conforme à Python avec un seule commande. Setuptools vous permet de télécharger et d'installer n'importe quel logiciel conforme à Python via un réseau (généralement Internet) avec une seule commande (`easy_install`). Il vous permet aussi d'ajouter cette capacité d'installation par le réseau à votre propre logiciel Python avec très peu de travail.

Pour obtenir la dernière version de Setuptools pour Windows, exécutez le script Python disponible ici : ez_setup.py

Vous avez maintenant une nouvelle commande à votre disposition : **easy_install**. Elle est considérée par beaucoup comme étant dépréciée. Ainsi, nous allons installer son remplaçant : **pip**. Pip permet la désinstallation de paquets et est activement maintenu, contrairement à easy_install.

Pour installer pip, exécutez le script Python disponible ici : get-pip.py

Environnements virtuels

Un environnement virtuel est un outil pour garder les dépendances requises par différents projets dans des emplacements séparés, en créant des environnements virtuels Python pour eux. Il résout le dilemme "le projet X dépend de la version 1.x mais le projet Y nécessite la 4.x", et garde votre répertoire site-packages global propre et gérable.

Par exemple, vous pouvez travailler sur un projet qui nécessite Django 1.3 tout en maintenant aussi un projet qui nécessite Django 1.0.

Pour commencer à utiliser cela et plus d'informations : documentation sur les *environnements virtuels*.

Cette page est une version remixée d'un autre guide, qui est disponible sous la même licence.

Installation de Python sous Linux

Les dernières versions de CentOS, Fedora, Redhat Enterprise (RHEL) et Ubuntu **sont fournies avec Python 2.7 par défaut**.

Pour savoir quelle version de Python vous avez installé, ouvrez une ligne de commande et lancez

```
$ python --version
```

Quelques vieilles versions de RHEL et CentOS sont fournies avec Python 2.4 qui est inacceptable pour le développement Python moderne. Heureusement, il y a des paquets supplémentaires pour Enterprise Linux qui incluent des paquets additionnels de haute qualité basés sur leurs versions équivalentes Fedora. Ce dépôt contient un paquet Python 2.6 conçu spécifiquement pour être installé côte à côte avec l'installation système Python 2.4.

Vous n'avez pas besoin d'installer ou de configurer quoi que ce soit pour utiliser Python. Cela dit, je recommande fortement que vous installiez les outils et les bibliothèques décrites dans la prochaine section avant que vous démarriez à construire des applications Python pour un usage dans le monde réel. En particulier, vous devriez toujours installer Setuptools et pip, comme ils rendent plus faciles pour vous d'utiliser d'autres bibliothèques Python tierces.

Setuptools & Pip

Les deux logiciels tiers les plus cruciaux des paquets Python sont setuptools et pip.

Une fois installé, vous pouvez télécharger, installer et désinstaller n'importe quel produit logiciel conforme à Python avec un seule commande. Il vous permet aussi d'ajouter cette capacité d'installation par le réseau à votre propre logiciel Python avec très peu de travail.

Python 2.7.9 et supérieur (sur la série python2), et Python 3.4 et supérieur incluent pip par défaut.

Pour voir si pip est installé, ouvrez une invite de commande et exécutez

```
$ command -v pip
```

Pour installer pip, suivez le guide d'installation pip officiel - cela installera automatiquement la dernière version de setuptools.

Environnements virtuels

Un environnement virtuel est un outil pour garder les dépendances requises par différents projets dans des emplacements séparés, en créant des environnements virtuels Python pour eux. Il résout le dilemme "le projet X dépend de la version 1.x mais le projet Y nécessite la 4.x", et garde votre répertoire site-packages global propre et gérable.

Par exemple, vous pouvez travailler sur un projet qui nécessite Django 1.3 tout en maintenant aussi un projet qui nécessite Django 1.0.

Pour commencer à utiliser cela et plus d'informations : documentation sur les *environnements virtuels*.
Vous pouvez aussi utiliser *virtualenvwrapper* pour rendre plus facile la gestion des environnements virtuels.

CHAPITRE 2

Écrire du “bon” code Python

Cette partie du guide se concentre sur les bonnes pratiques pour écrire du code Python.

Structurer votre projet

Par “structurer” nous entendons les décisions que vous faites concernant comment votre projet atteint au mieux son objectif. Nous avons besoin de considérer comment exploiter au mieux les fonctionnalités de Python pour créer un code propre et efficace. En termes pratiques, “structurer” signifie produire du code propre dont la logique et les dépendances sont claires ainsi que la façon dont les fichiers et dossiers sont organisés dans le système de fichiers.

Quelle fonctions doivent aller dans quels modules ? Comment circule la donnée dans le projet ? Quelles fonctionnalités et fonctions peuvent être groupées ensemble et isolées ? En répondant à des questions comme cela, vous pouvez commencer à planifier, au sens large, ce à quoi votre produit fini ressemblera.

Dans cette section, nous allons jeter un œil de plus près sur les systèmes de module et d’import de Python comme ils sont les éléments centraux pour faire respecter une structure dans votre projet. Nous discuterons ensuite des diverses perspectives sur comment construire du code qui peut être étendu et testé de manière fiable.

Structure du dépôt

C’est important

Juste comme le style de codage, le design d’API, et l’automatisation sont essentiels à un cycle de développement sain, la structure d’un dépôt est une part cruciale de l’architecture de votre projet.

Quand un utilisateur potentiel ou un contributeur arrive sur la page d’un dépôt, ils voient certaines choses :

— Le nom du projet
— La description du projet
— Un tas de fichiers

C’est seulement quand ils font défiler la page que les utilisateurs verront le README de votre projet.

Si votre dépôt est un amas massif de fichiers ou une pagaille imbriquée de répertoires, ils risquent de regarder ailleurs avant même de lire votre belle documentation.

> Habillez vous pour le job que vous voulez, pas pour le job que vous avez.

Bien sûr, les premières impressions ne sont pas tout. Vous et vos collègues allez passer un nombre d’heures incalculable à travailler sur ce dépôt, finalement devenir intimement familier avec tous les coins et recoins. Son organisation est importante.

Dépôt exemple

tl ;dr (acronyme de “Too Long, I Didn’t Read it”) : C’est ce que Kenneth Reitz recommande.

Ce dépôt est disponible sur GitHub.

```
README.rst
LICENSE
setup.py
requirements.txt
sample/__init__.py
sample/core.py
sample/helpers.py
docs/conf.py
docs/index.rst
tests/test_basic.py
tests/test_advanced.py
```

Entrons dans quelques détails.

Le module actuel

Emplacement	`./sample/` ou `./sample.py`
But	Le code qui nous intéresse

Votre paquet de module est le point central du dépôt. Il ne devrait pas être mis à l’écart :

```
./sample/
```

Si votre module consiste en un seul fichier, vous pouvez le placer directement à la racine de votre répertoire :

```
./sample.py
```

Votre bibliothèque n’appartient pas à un sous-répertoire ambigu comme src ou python.

Licence

Emplacement	`./LICENSE`
But	Se couvrir juridiquement.

Ceci est sans doute la partie la plus importante de votre dépôt, en dehors du code source lui-même. Les revendications de copyright et le texte de la licence complet devraient être dans ce fichier.

Si vous n’êtes pas sûr de la licence que vous souhaitez utiliser pour votre projet, consultez choosealicense.com.

Bien sûr, vous êtes aussi libre de publier votre code sans une licence, mais cela risque potentiellement d’empêcher de nombreuses personnes d’utiliser votre code.

Setup.py

Emplacement	`./setup.py`
But	Gestion de la distribution et de la création de paquets

Si votre paquet de module est à la racine de votre dépôt, ceci devrait évidemment être aussi à la racine.

Fichier requirements

Emplacement	`./requirements.txt`
But	Dépendances de développement.

Un fichier requirements de pip devrait être placé à la racine du dépôt. Il devrait spécifier les dépendances requises pour contribuer au projet : les tests, les builds et la génération de la documentation.

Si votre projet n'a pas de dépendances de développement ou vous préférez la configuration de l'environnement de développement via `setup.py`, ce fichier peut s'avérer non nécessaire.

Documentation

Emplacement	`./docs/`
But	Documentation de référence des paquets.

Il y a très peu de raison pour cela qu'il existe ailleurs.

Suite de tests

Emplacement	`./test_sample.py` ou `./tests`
But	Intégration de paquets et tests unitaires

En débutant, une petite suite de tests existera souvent dans un seul fichier :

```
./test_sample.py
```

Une fois que la suite de tests grossit, vous pouvez déplacer vos tests dans un répertoire, comme ceci :

```
tests/test_basic.py
tests/test_advanced.py
```

Évidemment, ces modules de test doivent importer votre module empaqueté pour le tester. Vous pouvez le faire de plusieurs façons :

— Attendre que le paquet soit installé dans site-packages.
— Utiliser un modification de chemin simple (mais *explicite*) pour résoudre le paquet correctement.

Je recommande fortement ce dernier. Demander à un développeur de lancer setup.py develop pour tester une base de code qui change activement lui demande aussi d'avoir une configuration d'environnement isolé pour chaque instance de la base de code.

Pour donner un contexte d'importation aux tests individuels, créez un fichier tests/config.py.

```
import os
import sys
sys.path.insert(0, os.path.abspath('..'))

import sample
```

Ensuite, dans les modules de test individuels, importez le module comme ceci :

```
from .context import sample
```

Cela fonctionnera toujours comme prévu quelle que soit la méthode d'installation.

Certains personnes avanceront que vous devriez distribuer vos tests à l'intérieur du module lui-même – Je ne suis pas d'accord. Cela augmente souvent la complexité pour vos utilisateurs ; de nombreuses suites de test nécessitent souvent des dépendances et des contextes d'exécution supplémentaires.

Makefile

Emplacement	`./Makefile`
But	Tâches de gestion génériques.

Si vous jetez un œil à la plupart de mes projets ou n'importe quel projet de Pocoo, vous remarquerez un Makefile qui traîne autour. Pourquoi ? Ces projets ne sont pas écrits en C... En bref, make est un outil incroyablement utile pour définir des tâches génériques pour votre projet.

Makefile exemple :

```
init:
    pip install -r requirements.txt

test:
    py.test tests
```

D'autres scripts de gestion génériques (comme `manage.py` ou `fabfile.py`) appartiennent aussi à la racine du dépôt.

En ce qui concerne les applications Django

J'ai noté une nouvelle tendance dans les applications Django depuis la sortie de Django 1.4. De nombreux développeurs structurent leurs dépôts de manière médiocre à cause des nouveaux modèles d'applications mis à disposition.

Comment ? Bien, ils vont dans leur nouveau dépôt encore nu et frais et exécutent ce qui suit, comme ils l'ont toujours fait :

```
$ django-admin.py start-project samplesite
```

La structure du dépôt résultant ressemble à ceci :

```
README.rst
samplesite/manage.py
samplesite/samplesite/settings.py
samplesite/samplesite/wsgi.py
samplesite/samplesite/sampleapp/models.py
```

Ne faites pas ça.

Des chemins répétitifs sont sources de confusion à la fois pour vos outils et vos développeurs. Une imbrication inutile n'aide personne (à moins d'être nostalgique des dépôts SVN monolithiques)

Faisons-le proprement :

```
$ django-admin.py start-project samplesite .
```

Notez le "`.`".

La structure résultante :

```
README.rst
manage.py
samplesite/settings.py
samplesite/wsgi.py
samplesite/sampleapp/models.py
```

La structure du code est la clé

Grâce à la façon dont les imports et les modules sont traités en Python, il est relativement facile de structurer un projet Python. Facile, ici, signifie que vous n'avez pas beaucoup de contraintes et que le modèle qui fait l'import du module est facile à comprendre. Par conséquent, vous vous retrouvez avec la tâche architecturale pure de concevoir les différentes parties de votre projet et de leurs interactions.

Une structuration facile de projet signifie que c'est aussi facile de mal le faire. Certains signes d'un projet mal structuré incluent :

— Des dépendances circulaires multiples et désordonnées : si vos classes Table et Chair dans `furn.py` ont besoin d'importer Carpenter depuis `workers.py` pour répondre à une question comme `table.isdoneby()`, et si au contraire la classe Carpenter a besoin d'importer Table et Chair, pour répondre à la question `carpenter.whatdo()`, alors vous avez une dépendance circulaire. Dans ce cas, vous devrez recourir à des hacks fragiles telles que l'utilisation de déclarations d'importation à l'intérieur de méthodes ou de fonctions.

— Couplage caché : chaque changement dans l'implémentation de Table casse 20 tests dans des cas de tests non liés parce qu'il casse le code de Carpenter, qui nécessite une intervention ciblée très prudente pour adapter le changement. Cela signifie que vous avez trop d'hypothèses à propos de Table dans le code de Carpenter ou l'inverse.

— Un usage intensif d'un état ou d'un contexte global : au lieu de passer explicitement `(height, width, type, wood)` de l'un à l'autre, Table et Carpenter s'appuient sur des variables globales qui peuvent être modifiées et sont modifiées à la volée par différents agents. Vous devez examiner tous les accès à ces variables globales pour comprendre pourquoi une table rectangulaire est devenu un carré, et découvrir que le code du modèle distant est aussi en train de modifier ce contexte, mettant le désordre dans les dimensions de la table.

— Code Spaghetti : plusieurs pages de clauses if imbriquée et de boucles for avec beaucoup de code procédural copie-collé et aucune segmentation adéquate sont connus comme du code spaghetti. L'indentation significative de Python (une de ses caractéristiques les plus controversées) rend très difficile de maintenir ce genre de code. Donc, la bonne nouvelle est que vous pourriez ne pas en voir de trop.

— Le code Ravioli est plus probable en Python : il se compose de centaines de petits morceaux semblables de logique, souvent des classes ou des objets, sans structure appropriée. Si vous ne pouvez vous rappeler si vous avez à utiliser FurnitureTable, AssetTable ou Table, ou même TableNew pour votre tâche, vous pourriez être en train de nager dans du code ravioli.

Modules

Les modules Python sont l'une des principales couches d'abstraction disponible et probablement la plus naturelle. Les couches d'abstraction permettent la séparation du code dans des parties contenant des données et des fonctionnalités connexes.

Par exemple, une couche d'un projet peut gérer l'interface avec les actions utilisateurs, tandis qu'un autre gérerait la manipulation de bas-niveau des données. La façon la plus naturelle de séparer ces deux couches est de regrouper toutes les fonctionnalités d'interface dans un seul fichier, et toutes les opérations de bas-niveau dans un autre fichier. Dans ce cas, le fichier d'interface doit importer le fichier de bas-niveau. Cela se fait avec les déclarations `import` et `from ... import`.

Dès que vous utilisez les déclarations *import*, vous utilisez des modules. Ceux-ci peuvent être soit des modules intégrés comme *os* et *sys*, soit des modules tiers que vous avez installé dans votre environnement, ou dans les modules internes de votre projet.

Pour rester aligné avec le guide de style, garder les noms de modules courts, en minuscules, et assurez-vous d'éviter d'utiliser des symboles spéciaux comme le point (.) ou le point d'interrogation (?). Donc, un nom de fichier comme `my.spam.py` est l'un de ceux que vous devriez éviter ! Nommer de cette façon interfère avec la manière dont Python cherche pour les modules.

Dans le cas de my.spam.py', Python attend de trouver un fichier `spam.py` dans un dossier nommé `my`, ce qui n'est pas le cas. Il y a un exemple de la comment la notation par point devrait être utilisée dans la documentation Python.

Si vous souhaitez, vous pouvez nommer votre module `my_spam.py`, mais même notre ami le underscore (tiret bas) ne devrait pas être vu souvent dans les noms de modules.

Mis à part quelques restrictions de nommage, rien de spécial n'est nécessaire pour qu'un fichier Python puisse être un module, mais vous avez besoin de comprendre le mécanisme d'import afin d'utiliser ce concept correctement et éviter certains problèmes.

Concrètement, la déclaration `import modu` va chercher le fichier approprié, qui est `modu.py` dans le même répertoire que l'appelant si il existe. Si il n'est pas trouvé, l'interpréteur Python va rechercher `modu.py` dans le "path" récursivement et lever une exception ImportError si il n'est pas trouvé.

Une fois que `modu.py` est trouvé, l'interpréteur Python va exécuter le module dans une portée isolée. N'importe quelle déclaration à la racine de `modu.py` sera exécutée, incluant les autres imports le cas échéant. Les définitions de fonction et classe sont stockées dans le dictionnaire du module.

Ensuite, les variables, les fonctions et les classes du module seront mises à disposition de l'appelant via l'espace de nom du module, un concept central dans la programmation qui est particulièrement utile et puissant en Python.

Dans de nombreuses langages, une directive `include file` est utilisée par le préprocesseur pour prendre tout le code trouvé dans le fichier et le 'copier' dans le code de l'appelant. C'est différent en Python : le code inclus est isolé dans un espace de nom de module, ce qui signifie que vous n'avez pas généralement à vous soucier que le code inclus puisse avoir des effets indésirables, par exemple remplacer une fonction existante avec le même nom.

Il est possible de simuler le comportement plus standard en utilisant une syntaxe particulière de la déclaration d'import : `from modu import *`. Ceci est généralement considéré comme une mauvaise pratique. **L'utilisation d'`import *` rend le code plus difficile à lire et rend les dépendances moins cloisonnées**.

L'utilisation de `from modu import func` est un moyen de cibler la fonction que vous souhaitez importer et la mettre dans l'espace de nom global. Bien que beaucoup moins néfaste que `import *` parce que cela montre explicitement ce qui est importé dans l'espace de nom global, son seul avantage par rapport à un `import modu`, plus simple est qu'il permettra d'économiser un peu de frappe clavier.

Très mauvais

```
[...]
from modu import *
[...]
x = sqrt(4)  # Is sqrt part of modu? A builtin? Defined above?
```

Mieux

```
from modu import sqrt
[...]
x = sqrt(4)  # sqrt may be part of modu, if not redefined in between
```

Le mieux

```
import modu
[...]
x = modu.sqrt(4)  # sqrt is visibly part of modu's namespace
```

Comme mentionné dans la section *Style de code*, la lisibilité est l'une des principales caractéristiques de Python. La lisibilité signifie éviter du texte standard inutile et le désordre, donc des efforts sont consacrés pour essayer de parvenir à un certain niveau de concision. Mais le laconisme et l'obscur sont les limites où la brièveté doit cesser. Être en mesure de dire immédiatement d'où une classe ou une fonction provient, comme dans l'idiome `modu.func`, améliore grandement la lisibilité du code et la compréhensibilité dans tous les projets même ceux avec le plus simple fichier unique.

Paquets

Python fournit un système de packaging très simple, qui est simplement une extension du mécanisme de module à un répertoire.

Tout répertoire avec un fichier `__init__.py` est considéré comme un paquet Python. Les différents modules dans le paquet sont importés d'une manière similaire comme des modules simples, mais avec un comportement spécial pour le fichier `__init__.py`, qui est utilisé pour rassembler toutes les définitions à l'échelle des paquets.

Un fichier `modu.py` dans le répertoire `pack/` est importé avec la déclaration `import pack.modu`. Cette déclaration va chercher un fichier `__init__.py` dans `pack`, exécuter toutes ses déclarations de premier niveau. Puis elle va chercher un fichier nommé `pack/modu.py` et exécuter tous ses déclarations de premier niveau. Après ces opérations, n'importe quelle variable, fonction ou classe définie dans `modu.py` est disponible dans l'espace de nom pack.modu.

Un problème couramment vu est d'ajouter trop de code aux fichiers `__init__.py`. Lorsque la complexité du projet grossit, il peut y avoir des sous-paquets et sous-sous-paquets dans une structure de répertoire profonde. Dans ce cas, importer un seul élément à partir d'un sous-sous-paquet nécessitera d'exécuter tous les fichiers `__init__.py` rencontrés en traversant l'arbre.

Laisser un fichier `__init__.py` vide est considéré comme normal et même une bonne pratique, si les modules du paquet et des sous-paquets n'ont pas besoin de partager aucun code.

Enfin, une syntaxe pratique est disponible pour importer des paquets imbriquées profondément : `import very.deep.module as mod`. Cela vous permet d'utiliser *mod* à la place de la répétition verbeuse de `very.deep.module`.

Programmation orientée objet

Python est parfois décrit comme un langage de programmation orienté objet. Cela peut être quelque peu trompeur et doit être clarifié.

En Python, tout est un objet, et peut être manipulé en tant que tel. Voilà ce que l'on entend quand nous disons, par exemple, que les fonctions sont des objets de première classe. Les fonctions, les classes, les chaînes et même les types sont des objets en Python : comme tout objet, ils ont un type, ils peuvent être passés comme arguments de fonction, et ils peuvent avoir des méthodes et propriétés. Sur ce point, Python est un langage orienté objet.

Cependant, contrairement à Java, Python n'impose pas la programmation orientée objet comme paradigme de programmation principal. Il est parfaitement viable pour un projet de Python de ne pas être orienté objet, à savoir de ne pas utiliser ou très peu de définitions de classes, d'héritage de classe, ou d'autres mécanismes qui sont spécifiques à la programmation orientée objet.

En outre, comme on le voit dans la section *modules*, la façon dont Python gère les modules et les espaces de nom donne au développeur un moyen naturel pour assurer l'encapsulation et la séparation des couches d'abstraction, les deux étant les raisons les plus courantes d'utiliser l'orientation objet. Par conséquent, les programmeurs Python ont plus de latitude pour ne pas utiliser l'orientation objet, quand elle n'est pas requise par le modèle métier.

Il y a quelques raisons pour éviter inutilement l'orientation objet. Définir des classes personnalisées est utile lorsque l'on veut coller ensemble un état et certaines fonctionnalités. Le problème, comme l'a souligné les discussions sur la programmation fonctionnelle, vient de la partie "state" de l'équation.

Dans certaines architectures, typiquement des applications Web, plusieurs instances de processus Python sont lancées pour répondre aux demandes externes qui peuvent se produire en même temps. Dans ce cas, tenir quelques états dans des objets instanciés, ce qui signifie garder des informations statiques sur le monde, est sujet à des problèmes de concurrence ou de race-conditions. Parfois, entre l'initialisation de l'état d'un objet (généralement fait avec la méthode `__init__()`) et l'utilisation réelle de l'état de l'objet à travers l'une de ses méthodes, le monde peut avoir changé, et l'état retenu peut ne plus être à jour. Par exemple, une requête peut charger un élément en mémoire et le marquer comme lu par un utilisateur. Si une autre requête nécessite la suppression de cet article dans le même temps, il peut

arriver que la suppression se produise pour de vrai après que le premier processus ait chargé l'élément, et ensuite nous devons marquer comme lu un objet supprimé.

Ceci et d'autres problèmes a conduit à l'idée que l'utilisation des fonctions sans état est un meilleur paradigme de programmation.

Une autre façon de dire la même chose est de suggérer l'utilisation des fonctions et procédures avec le moins de contextes implicites et d'effets de bord possibles. Le contexte d'une fonction implicite est composée de n'importe quelles variables ou objets globaux dans la couche de persistance qui sont accessibles depuis l'intérieur de la fonction. Les effets de bord sont les changements qu'une fonction fait à son contexte implicite. Si une fonction sauve ou supprime la donnée dans une variable globale ou dans la couche de persistance, elle est dite comme ayant un effet de bord.

Isoler soigneusement les fonctions avec un contexte et des effets de bord depuis des fonctions avec une logique (appelé fonctions pures) permet les avantages suivants :

— Les fonctions pures sont déterministes : pour une entrée donnée fixe, la sortie sera toujours la même.
— Les fonctions pures sont beaucoup plus faciles à changer ou remplacer si elles doivent être refactorisées ou optimisées.
— Les fonctions pures sont plus faciles à tester avec des tests unitaires : il y a moins besoin d'une configuration du contexte complexe et d'un nettoyage des données après.
— Les fonctions pures sont plus faciles à manipuler, décorer, et déplacer.

En résumé, les fonctions pures sont des blocs de construction plus efficaces que les classes et les objets pour certaines architectures parce qu'elles n'ont pas de contexte ou d'effets de bord.

Évidemment, l'orientation objet est utile et même nécessaire dans de nombreux cas, par exemple lors du développement d'applications graphiques de bureau ou des jeux, où les choses qui sont manipulés (fenêtres, boutons, avatars, véhicules) ont une vie relativement longue par elle-même dans la mémoire de l'ordinateur.

Décorateurs

Le langage Python fournit une syntaxe simple mais puissante appelée 'décorateurs'. Un décorateur est une fonction ou une classe qui enveloppe (ou décore) une fonction ou une méthode. La fonction ou méthode 'décorée" remplacera la fonction ou méthode originale 'non décorée' . Parce que les fonctions sont des objets de première classe en Python, cela peut être fait 'manuellement', mais utiliser la syntaxe @decorator est plus clair et donc préféré.

```
def foo():
    # do something

def decorator(func):
    # manipulate func
    return func

foo = decorator(foo)  # Manually decorate

@decorator
def bar():
    # Do something
# bar() is decorated
```

Ce mécanisme est utile pour la "Separation of concerns" et évite à une logique externe non liée de "polluer" la logique de base de la fonction ou de la méthode. Un bon exemple de morceau de fonctionnalité qui est mieux géré avec la décoration est la mémorisation ou la mise en cache : vous voulez stocker les résultats d'une fonction coûteuse dans une table et les utiliser directement au lieu de les recalculer quand ils ont déjà été calculés. Ceci ne fait clairement pas partie de la logique de la fonction.

Gestionnaires de contexte

Un gestionnaire de contexte est un objet Python qui fournit une information contextuelle supplémentaire à une action. Cette information supplémentaire prend la forme de l'exécution d'un callable lors de l'initialisation d'un contexte à l'aide de la déclaration `with`, ainsi que l'exécution d'un callable après avoir terminé tout le code à l'intérieur du bloc `with`. L'exemple le plus connu d'utilisation d'un gestionnaire de contexte est montré ici, à l'ouverture d'un fichier :

```
with open('file.txt') as f:
    contents = f.read()
```

Quiconque est familier avec ce pattern sait que l'invocation `open` de cette façon s'assure que la fonction `close``de ``f` sera appelée à un moment donné. Cela réduit la charge cognitive d'un développeur et rend le code plus facile à lire.

Il existe deux moyens faciles d'implémenter cette fonctionnalité vous-même : en utilisant une classe ou à l'aide d'un générateur. Implémentons la fonctionnalité ci-dessus nous-mêmes, en commençant par l'approche par classe :

```
class CustomOpen(object):
    def __init__(self, filename):
      self.file = open(filename)

    def __enter__(self):
        return self.file

    def __exit__(self, ctx_type, ctx_value, ctx_traceback):
        self.file.close()

with CustomOpen('file') as f:
    contents = f.read()
```

C'est juste un objet Python régulier avec deux méthodes supplémentaires qui sont utilisés par la déclaration `with`. CustomOpen est d'abord instancié puis sa méthode `__enter__` est appelée et tout ce que `__enter__` retourne est assigné à `f` dans la partie `as _f` de la déclaration. Lorsque le contenu du bloc `with` a fini de s'exécuter, la méthode `__exit__` est alors appelée.

Et maintenant, l'approche par générateur en utilisant la bibliothèque contextlib de Python :

```
from contextlib import contextmanager

@contextmanager
def custom_open(filename):
    f = open(filename)
    try:
        yield f
    finally:
        f.close()

with custom_open('file') as f:
    contents = f.read()
```

Cela fonctionne exactement de la même manière que l'exemple de classe ci-dessus, mais il est plus laconique. La fonction `custom_open` s'exécute jusqu'à la déclaration `yield`. Il rend alors le contrôle à la déclaration `with`, qui assigne tout ce qui était `yield`'é à *f'dans la portion 'as f*. La clause `finally` s'assure que `close()` est appelé s'il y avait ou non une exception à l'intérieur du `with`.

Étant donné que les deux approches semblent similaires, nous devrions suivre le Zen de Python pour décider quand utiliser laquelle. L'approche classe pourrait être mieux s'il y a une quantité considérable de logique à encapsuler. L'approche fonction pourrait être préférable pour des situations où nous avons affaire à une action simple.

Typage dynamique

Python est typé dynamiquement, ce qui signifie que les variables n'ont pas un type fixe. En fait, en Python, les variables sont très différentes de ce qu'elles sont dans de nombreux autres langages, en particulier les langages typés statiquement. Les variables ne sont pas un segment de la mémoire de l'ordinateur où une certaine valeur est écrite, elles sont des 'tags' ou des 'noms' pointant vers des objets. Il est donc possible pour la variable 'a' d'être définie à la valeur 1, puis à la valeur 'a string', puis à une fonction.

Le typage dynamique de Python est souvent considéré comme une faiblesse, et en effet, elle peut conduire à la complexité et à du code difficile à débugguer. Quelque chose nommée 'a' peut être assigné à de nombreuses choses différentes, et le développeur ou le mainteneur doit suivre ce nom dans le code pour s'assurer qu'il n'a pas été assigné à un objet complètement différent.

Quelques lignes directrices pour éviter ce problème :

— Évitez d'utiliser le même nom de variable pour des choses différentes.

Mauvais

```
a = 1
a = 'a string'
def a():
    pass  # Do something
```

Bon

```
count = 1
msg = 'a string'
def func():
    pass  # Do something
```

Utiliser des fonctions ou des méthodes courtes permet de réduire le risque d'utiliser le même nom pour deux choses indépendantes.

Il est préférable d'utiliser des noms différents, même pour des choses qui sont liées, quand elles ont un type différent :

Mauvais

```
items = 'a b c d'  # This is a string...
items = items.split(' ')  # ...becoming a list
items = set(items)  # ...and then a set
```

Il n'y a pas de gain d'efficacité lors de la réutilisation de noms : les affectations devront créer de nouveaux objets de toute façon. Toutefois, lorsque la complexité augmente et chaque affectation est séparée par d'autres lignes de code, incluant des branches 'si' et des boucles, il devient plus difficile d'établir quel est le type d'une variable donnée.

Certaines pratiques de codage, comme la programmation fonctionnelle, recommandent de ne jamais réaffecter une variable. En Java cela se fait avec le mot-clé *final*. Python n'a pas de mot-clé *final* et cela irait à l'encontre de sa philosophie de toute façon. Cependant, cela peut être une bonne discipline pour éviter d'assigner à une variable plus d'une fois, et cela aide à comprendre le concept de types mutables et immutables.

Types mutables et immutables

Python a deux sortes de types intégrés/définis par l'utilisateur.

Les types mutables sont ceux qui permettent la modification sur place du contenu. Des mutables typiques sont les listes et les dictionnaires : toutes les listes ont des méthodes mutables, comme `list.append()` ou `list.pop()`, et peuvent être modifiées sur place. La même chose vaut pour les dictionnaires.

Les types immuables fournissent aucune méthode pour modifier leur contenu. Par exemple, la variable x définie à l'entier 6 n'a pas de méthode "increment". Si vous voulez calculer x + 1, vous devez créer un autre entier et lui donner un nom.

```
my_list = [1, 2, 3]
my_list[0] = 4
print my_list  # [4, 2, 3] <- The same list as changed

x = 6
x = x + 1  # The new x is another object
```

Une conséquence de cette différence de comportement est que les types mutables ne sont pas "stables", et ne peuvent donc être utilisées comme clés du dictionnaire.

L'utilisation correcte des types mutables pour des choses qui sont mutables par nature et des types immutables pour des choses qui sont fixes par nature aide à clarifier l'intention du code.

Par exemple, l'équivalent immutable d'une liste est le tuple, créé avec `(1, 2)`. Ce tuple est une paire qui ne peut pas être changé sur place, et qui peut être utilisée comme clé pour un dictionnaire.

Une particularité de Python qui peut surprendre les débutants est que les chaînes sont immutables. Cela signifie que lors de la construction d'une chaîne à partir de ses parties, il est beaucoup plus efficace d'accumuler les parties dans une liste, qui est mutable, puis coller ('join') les morceaux ensemble lorsque la chaîne complète est nécessaire. Une chose à remarquer, cependant, est que les compréhensions de liste sont mieux et plus rapides que la construction d'une liste dans une boucle avec des appels à `append()`.

Mauvais

```
# create a concatenated string from 0 to 19 (e.g. "012..1819")
nums = ""
for n in range(20):
  nums += str(n)   # slow and inefficient
print nums
```

Bon

```
# create a concatenated string from 0 to 19 (e.g. "012..1819")
nums = []
for n in range(20):
  nums.append(str(n))
print "".join(nums)  # much more efficient
```

Le mieux

```
# create a concatenated string from 0 to 19 (e.g. "012..1819")
nums = [str(n) for n in range(20)]
print "".join(nums)
```

Une dernière chose à mentionner sur les chaînes est que l'utilisation `join()` n'est est pas toujours ce qu'il y a de mieux. Dans les cas où vous créez une nouvelle chaîne depuis un nombre prédéterminé de chaînes, utiliser l'opérateur d'addition est vraiment plus rapide, mais dans des cas comme ci-dessus ou dans des cas où vous ajoutez à une chaîne existante, utiliser `join()` devrait être votre méthode de préférence.

```
foo = 'foo'
bar = 'bar'

foobar = foo + bar  # This is good
foo += 'ooo'  # This is bad, instead you should do:
foo = ''.join([foo, 'ooo'])
```

Note : Vous pouvez également utiliser l'opérateur de formatage `%` pour concaténer un nombre prédéterminé de chaînes en plus de `str.join()` et +. Cependant, la **PEP 3101**, décourage l'utilisation de l'opérateur `%` en faveur de la méthode `str.format()`.

```
foo = 'foo'
bar = 'bar'

foobar = '%s%s' % (foo, bar) # It is OK
foobar = '{0}{1}'.format(foo, bar) # It is better
foobar = '{foo}{bar}'.format(foo=foo, bar=bar) # It is best
```

Inclure les dépendances dans l'arbre de source de votre dépôt de code

Runners

Lectures complémentaires

— http://docs.python.org/2/library/
— http://www.diveintopython.net/toc/index.html

Style de code

Si vous demandez aux programmeurs Python ce qu'ils aiment le plus à propos de Python, ils citeront souvent sa grande lisibilité. En effet, un haut niveau de lisibilité est au cœur de la conception du langage Python, après le fait reconnu que le code est lu beaucoup plus souvent que ce qui est écrit.

Une raison de la haute lisibilité du code Python est son jeu relativement complet d'instructions de style de code et ses idiomes "Pythoniques".

Quand un développeur Python vétéran (un Pythoniste) appelle des portions de code non "Pythoniques", il veut dire généralement que ces lignes de code ne suivent pas les instructions communes et ne parviennent pas à exprimer leur intention dans ce qui est considéré comme la meilleure façon (entendez : la manière la plus lisible).

Dans certains cas limites, aucune meilleure pratique n'a été convenue sur la façon d'exprimer une intention en code Python, mais ces cas sont rares.

Concepts généraux

Code explicite

Alors que toute sorte de magie noire est possible avec Python, la manière la plus explicite et directe est préférable.

Mauvais

```
def make_complex(*args):
    x, y = args
    return dict(**locals())
```

Bon

```
def make_complex(x, y):
    return {'x': x, 'y': y}
```

Dans le bon code ci-dessus, x et y sont explicitement reçus de l'appelant, et un dictionnaire explicite est retourné. Le développeur utilisant cette fonction sait exactement ce qu'il faut faire en lisant la première et la dernière ligne, ce qui n'est pas le cas avec le mauvais exemple.

Une déclaration par ligne

Bien que certaines déclarations composées comme les compréhensions de liste soient autorisées et appréciées pour leur brièveté et leur expressivité, c'est une mauvaise pratique d'avoir deux déclarations disjointes sur la même ligne de code.

Mauvais

```
print 'one'; print 'two'

if x == 1: print 'one'

if <complex comparison> and <other complex comparison>:
    # do something
```

Bon

```
print 'one'
print 'two'

if x == 1:
    print 'one'

cond1 = <complex comparison>
cond2 = <other complex comparison>
if cond1 and cond2:
    # do something
```

Arguments de fonction

Les arguments peuvent être passés aux fonctions de quatre manières différentes.

1. **Les arguments positionnels** sont obligatoires et n'ont pas de valeurs par défaut. Ils sont la forme la plus simple des arguments et ils peuvent être utilisés pour les quelques arguments de fonction qui sont partie intégrante de la signification de la fonction et leur ordre est naturel. Par exemple, dans `send(message, recipient)` ou `point(x, y)` l'utilisateur de la fonction n'a pas de difficulté à se souvenir que ces deux fonctions nécessitent deux arguments, et dans quel ordre.

Dans ces deux cas, il est possible d'utiliser des noms d'argument lors de l'appel des fonctions et, ce faisant, il est possible de changer l'ordre des arguments, appelant par exemple `send(recipient='World', message='Hello')` et `point(y=2, x=1)`, mais cela réduit la lisibilité et est inutilement verbeux, par rapport aux appels plus simples à `send('Hello', 'World')` et `point(1, 2)`.

2. **Les arguments nommés** ne sont pas obligatoires et ont des valeurs par défaut. Ils sont souvent utilisés pour les paramètres facultatifs envoyés à la fonction. Quand une fonction a plus de deux ou trois paramètres positionnels, sa signature est plus difficile à retenir et l'utilisation d'arguments nommés avec des valeurs par défaut est utile. Par exemple, une fonction `send` plus complète pourrait être définie comme `send(message, to, cc=None, bcc=None)`. Ici `cc` et `bcc` sont facultatifs, et sont évalués à `None` quand ils ne reçoivent pas une autre valeur.

L'appel d'une fonction avec des arguments nommés peut être fait de plusieurs façons en Python. Par exemple, il est possible de suivre l'ordre des arguments dans la définition sans nommer explicitement les arguments, comme dans `send('Hello', 'World', 'Cthulhu', 'God')`, envoyant une copie carbone invisible à God. Il serait également possible de nommer des arguments dans un autre ordre, comme dans `send('Hello again', 'World', bcc='God', cc='Cthulhu')`. Ces deux possibilités sont mieux évitées sans aucune vraie raison de ne pas suivre la syntaxe qui est le plus proche de la définition de la fonction : `send('Hello', 'World', cc='Cthulhu', bcc='God')`.

Comme note à garder de côté, en suivant le principe YAGNI, il est souvent plus difficile d'enlever un argument optionnel (et sa logique dans la fonction) qui a été ajouté "juste au cas où" et n'est apparemment jamais utilisé, que d'ajouter un nouvel argument optionnel et sa logique en cas de besoin.

3. La **liste d'arguments arbitraires** est la troisième façon de passer des arguments à une fonction. Si l'intention de la fonction est mieux exprimée par une signature avec un nombre extensible d'arguments positionnels, elle peut être définie avec les constructions `args`. Dans le corps de la fonction, `args` sera un tuple de tous les arguments positionnels restants. Par exemple, `send(message, *args)` peut être appelé avec chaque destinataire comme argument : `send('Hello', 'God', 'Mom', 'Cthulhu')`, et dans le corps de la fonction `args` sera égal à `('God', 'Mom', 'Cthulhu')`.

Cependant, cette construction présente des inconvénients et doit être utilisée avec prudence. Si une fonction reçoit une liste d'arguments de même nature, il est souvent plus clair de la définir comme une fonction d'un seul argument, cet argument étant une liste ou n'importe quelle séquence. Ici, si `send` a plusieurs destinataires, il est préférable de la définir explicitement : `send(message, recipients)` et de l'appeler avec `send('Hello', ['God', 'Mom', 'Cthulhu'])`. De cette façon, l'utilisateur de la fonction peut manipuler la liste des destinataires sous forme de liste à l'avance, et cela ouvre la possibilité de passer un ordre quelconque, y compris les itérateurs, qui ne peuvent être unpacked comme d'autres séquences.

4. Le **dictionnaire d'arguments mot-clé arbitraire** est le dernier moyen de passer des arguments aux fonctions. Si la fonction nécessite une série indéterminée d'arguments nommés, il est possible d'utiliser la construction `**kwargs`. Dans le corps de la fonction, `kwargs` sera un dictionnaire de tous les arguments nommés passés qui n'ont pas été récupérés par d'autres arguments mot-clés dans la signature de la fonction.

La même prudence que dans le cas de *liste d'arguments arbitraires* est nécessaire, pour des raisons similaires : ces techniques puissantes doivent être utilisés quand il y a une nécessité avérée de les utiliser, et elles ne devraient pas être utilisées si la construction simple et plus claire est suffisante pour exprimer l'intention de la fonction.

Il appartient au programmeur d'écrire la fonction pour déterminer quels arguments sont des arguments positionnels et qui sont des arguments optionnels mots-clés, et de décider d'utiliser ou non les techniques avancées de passage d'arguments arbitraires. Si le conseil ci-dessus est suivi à bon escient, il est possible et agréable d'écrire des fonctions Python qui sont :

— faciles à lire (le nom et les arguments n'ont pas besoin d'explications)
— faciles à changer (l'ajout d'un nouveau mot-clé en argument ne casse pas les autres parties du code)

Éviter la baguette magique

Un outil puissant pour les hackers, Python vient avec un jeu très riche de hooks et d'outils vous permettant de faire presque tout sorte de trucs délicats. Par exemple, il est possible de faire chacun des éléments suivants :

— changer comment les objets sont créés et instanciés
— changer comment l'interpréteur Python importe les modules
— il est même possible (et recommandé si nécessaire) pour intégrer des routines C dans Python.

Cependant, toutes ces options présentent de nombreux inconvénients et il est toujours préférable d'utiliser le moyen le plus simple pour atteindre votre objectif. Le principal inconvénient est que la lisibilité souffre beaucoup lors de l'utilisation de ces constructions. De nombreux outils d'analyse de code, comme pylint ou pyflakes, ne pourront pas analyser ce code "magique".

Nous considérons qu'un développeur Python devrait connaître ces possibilités presque infinies, car cela inspire la confiance qu'aucun problème infranchissable ne sera sur le chemin. Cependant, savoir comment et surtout quand ne **pas** les utiliser est très important.

Comme un maître de kung-fu, un Pythoniste sait comment tuer avec un seul doigt, et ne jamais le faire pour de vrai.

Nous sommes tous des utilisateurs responsables

Comme on le voit ci-dessus, Python permet de nombreuses astuces, et certains d'entre elles sont potentiellement dangereuses. Un bon exemple est que tout code client peut surcharger les propriétés et les méthodes d'un objet : il n'y a pas mot-clé "private" en Python. Cette philosophie, très différente des langages très défensifs comme Java, qui donnent beaucoup de mécanismes pour empêcher toute utilisation abusive, est exprimée par le dicton : "Nous sommes tous les utilisateurs responsables".

Cela ne veut pas dire que, par exemple, que des propriétés ne sont pas considérées comme privées, et qu'aucune encapsulation appropriée n'est possible en Python. Au contraire, au lieu de compter sur les murs de béton érigés par les développeurs entre leur code et les autres, la communauté Python préfère compter sur un ensemble de conventions indiquant que ces éléments ne doivent pas être directement accessibles.

La convention principale pour les propriétés privées et les détails d'implémentation est de préfixer toutes les "caractéristiques internes" avec un tiret bas. Si le code client enfreint cette règle et accède à ces éléments marqués, tous les mauvais comportements ou problèmes rencontrés si le code est modifié est de la responsabilité du code client.

L'utilisation de cette convention généreusement est encouragée : toute méthode ou une propriété qui ne sont pas destinées à être utilisées par le code client doivent être préfixées avec un tiret bas. Cela permettra de garantir une meilleure séparation des tâches et une modification plus facile du code existant ; il sera toujours possible d'exposer une propriété privée, mais rendre une propriété publique privée pourrait être une opération beaucoup plus difficile.

Valeurs retournées

Quand une fonction croît en complexité, il n'est pas rare d'utiliser des instructions de retour multiples à l'intérieur du corps de la fonction. Cependant, afin de maintenir une intention claire et un niveau de lisibilité durable, il est préférable d'éviter de retourner des valeurs significatives à partir de nombreux points de sortie dans le corps.

Il existe deux principaux cas pour retourner des valeurs dans une fonction : le résultat du retour de la fonction quand il a été traité normalement, et les cas d'erreur indiquant un paramètre d'entrée erroné ou toute autre raison pour la fonction de ne pas être capable de compléter son calcul ou sa tâche.

Si vous ne souhaitez pas de lever des exceptions pour le second cas, puis retourner une valeur, comme None ou False, indiquer que la fonction pourrait ne pas fonctionner correctement pourrait être nécessaire. Dans ce cas, il est préférable de la retourner aussitôt que le contexte incorrect a été détecté. Cela aidera à aplatir la structure de la fonction : tout le code après l'instruction retour-parce-que-erreur peut assumer que la condition est remplie pour continuer à calculer le résultat principal de la fonction. Avoir de telles multiples déclarations de retour est souvent nécessaire.

Cependant, lorsqu'une fonction a plusieurs principaux points de sortie pour son fonctionnement normal, il devient difficile de débugguer le résultat retourné, donc il peut être préférable de garder un seul point de sortie. Cela permettra également d'aider à refactoriser quelques chemins dans le code, et les points de sortie multiples sont une indication probable qu'un tel refactoring est nécessaire.

```
def complex_function(a, b, c):
    if not a:
        return None  # Raising an exception might be better
    if not b:
        return None  # Raising an exception might be better
    # Some complex code trying to compute x from a, b and c
    # Resist temptation to return x if succeeded
    if not x:
        # Some Plan-B computation of x
    return x  # One single exit point for the returned value x will help
              # when maintaining the code.
```

Idiomes

Un idiome de langage, dit simplement, est un *moyen* d'écrire du code. La notion d'idiomes de programmation est discutée amplement sur c2 et sur Stack Overflow.

Du code Python idiomatique est souvent désigné comme étant *Pythonique*.

Bien qu'il y ait habituellement une — et de préférence une seule — manière évidente de le faire ; *la* façon d'écrire du code Python idiomatique peut être non évidente pour les débutants Python. Donc, les bons idiomes doivent être consciemment acquis.

Certains idiomes Python communs suivent :

Unpacking

Si vous connaissez la longueur d'une liste ou d'un tuple, vous pouvez attribuer des noms à ses éléments avec l'unpacking. Par exemple, étant donné que `enumerate()` fournira un tuple de deux éléments pour chaque élément dans la liste :

```
for index, item in enumerate(some_list):
    # do something with index and item
```

Vous pouvez l'utiliser pour intervertir des variables ainsi :

```
a, b = b, a
```

L'unpacking imbriqué marche aussi :

```
a, (b, c) = 1, (2, 3)
```

En Python 3, une nouvelle méthode d'unpacking étendue a été introduite par la **PEP 3132** :

```
a, *rest = [1, 2, 3]
# a = 1, rest = [2, 3]
a, *middle, c = [1, 2, 3, 4]
# a = 1, middle = [2, 3], c = 4
```

Créer une variable ignorée

Si vous avez besoin d'assigner quelque chose (par exemple, dans *Unpacking*) mais que vous n'avez pas besoin de cette variable, utilisez __ :

```
filename = 'foobar.txt'
basename, __, ext = filename.rpartition('.')
```

Note : Beaucoup de guides de style Python recommandent l'utilisation d'un seul tiret bas "_" pour les variables jetables plutôt que les tirets bas doubles "__" recommandés ici. Le problème est que "_" est couramment utilisé comme un alias pour la fonction `gettext()`, et est également utilisé dans l'invite interactive pour garder la valeur de la dernière opération. L'utilisation d'un tiret bas double est à la place est tout aussi clair et presque aussi pratique, et élimine le risque d'interférer accidentellement avec l'un de ces autres cas d'utilisation.

Créer un liste de longueur N de la même chose

Utilisez l'opérateur de liste * de Python :

```
four_nones = [None] * 4
```

Créez un liste de longueur N de listes

parce que les listes sont mutables, l'opérateur * (comme ci-dessus) créera une liste de N références vers la *même* liste, ce qui est probablement pas ce que vous voulez. A la place, utilisez une compréhension de liste :

```
four_lists = [[] for __ in xrange(4)]
```

Note : utilisez range() à la place de xrange() en Python 3

Créer une chaîne depuis une liste

Un idiome commun pour la création de chaînes est d'utiliser `str.join()` sur une chaîne vide.

```
letters = ['s', 'p', 'a', 'm']
word = ''.join(letters)
```

Cela va définir la valeur de la variable *word* à 'spam'. Cet idiome peut être appliqué à des listes et des tuples.

Rechercher un élément dans une collection

Parfois, nous devons chercher à travers une collection de choses. Regardons deux options : lists et sets.

Prenez le code suivant pour exemple :

```
s = set(['s', 'p', 'a', 'm'])
l = ['s', 'p', 'a', 'm']

def lookup_set(s):
    return 's' in s

def lookup_list(l):
    return 's' in l
```

Même si les deux fonctions semblent identiques, parce que *lookup_set* utilise le fait que les sets en Python sont des tables de hashage, les performances de recherche entre les deux sont très différentes. Pour déterminer si un élément se trouve dans une liste, Python devra parcourir chaque élément jusqu'à ce qu'il trouve un élément correspondant. Cela prend du temps, surtout pour de longues listes. Dans un set, d'autre part, le hachage de l'élément dira à Python où dans le set chercher un élément correspondant. En conséquence, la recherche peut être faite rapidement, même si le set est grand. La recherche dans les dictionnaires fonctionne de la même façon. Pour plus d'informations, voir cette page StackOverflow. Pour plus d'informations sur la quantité de temps nécessaire pour les différentes opérations courantes pour chacune de ces structures de données, voir cette page.

En raison de ces différences de performance, c'est souvent une bonne idée d'utiliser des sets ou des dictionnaires au lieu de listes dans les cas où :

— La collection contiendra un grand nombre d'éléments
— Vous rechercherez de manière répétitive les éléments dans la collection
— Vous n'avez pas pas d'éléments dupliqués.

Pour les petites collections, ou les collections que vous n'avez pas fréquemment à rechercher, le temps additionnel et la mémoire requise pour configurer la table de hashage seront souvent plus longs que le temps gagné grâce à l'amélioration de la vitesse de recherche.

Le Zen de Python

Aussi connu comme **PEP 20**, les principes directeurs pour la conception de Python.

```
>>> 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!
```

Pour quelques exemples de bons styles Python, voir ces diapositives d'un groupe d'utilisateurs Python.

PEP 8

PEP 8 est le guide de fait du style de code pour Python. Une version de haute qualité, facile à lire de la PEP 8 est également disponible sur pep8.org.

C'est une lecture fortement recommandée. La communauté Python entière fait de son mieux pour adhérer aux instructions énoncées dans le présent document. Certains projets peuvent osciller autour d'elles de temps à autre, tandis que d'autres peuvent modifier ses recommandations.

Cela étant dit, conformer votre code Python à PEP 8 est généralement une bonne idée et contribue à rendre le code plus consistant lorsque vous travaillez sur des projets avec d'autres développeurs. Il existe un programme en ligne de commande, pep8, qui peut vérifier la conformité de votre code. Installez-le en exécutant la commande suivante dans votre terminal :

```
$ pip install pep8
```

Ensuite, exécutez-le sur un fichier ou une série de fichiers pour avoir un rapport de toutes violations.

```
$ pep8 optparse.py
optparse.py:69:11: E401 multiple imports on one line
optparse.py:77:1: E302 expected 2 blank lines, found 1
optparse.py:88:5: E301 expected 1 blank line, found 0
optparse.py:222:34: W602 deprecated form of raising exception
```

```
optparse.py:347:31: E211 whitespace before '('
optparse.py:357:17: E201 whitespace after '{'
optparse.py:472:29: E221 multiple spaces before operator
optparse.py:544:21: W601 .has_key() is deprecated, use 'in'
```

Le programme autopep8 peut être utilisé pour reformater automatiquement le code dans le style PEP 8. Installez le programme avec :

```
$ pip install autopep8
```

Utilisez-le pour formater un fichier sur place avec :

```
$ autopep8 --in-place optparse.py
```

Exclure l'option `--in-place` va mener le programme à renvoyer en sortie le code modifié directement dans la console pour examen. L'option `--aggressive` effectuera des changements plus importants et peut être appliquée à plusieurs reprises pour plus d'effet.

Conventions

Voici quelques conventions que vous devriez suivre pour rendre votre code plus facile à lire.

Vérifier sir la variable est égale à une constante

Vous n'avez pas besoin de comparer explicitement une valeur à True ou None, ou 0 - vous pouvez simplement l'ajouter à l'instruction if. Voir le Test des valeurs à True pour une liste de ce qui est considéré comme False.

Mauvais :

```
if attr == True:
    print 'True!'

if attr == None:
    print 'attr is None!'
```

Bon :

```
# Just check the value
if attr:
    print 'attr is truthy!'

# or check for the opposite
if not attr:
    print 'attr is falsey!'

# or, since None is considered false, explicitly check for it
if attr is None:
    print 'attr is None!'
```

Accéder à un élément de dictionnaire

N'utilisez pas la méthode `dict.has_key()`. A la place,utilisez la syntaxe `x in d`, ou passez un argument par défaut à `dict.get()`.

Mauvais :

```
d = {'hello': 'world'}
if d.has_key('hello'):
    print d['hello']    # prints 'world'
else:
    print 'default_value'
```

Bon :

```
d = {'hello': 'world'}

print d.get('hello', 'default_value') # prints 'world'
print d.get('thingy', 'default_value') # prints 'default_value'

# Or:
if 'hello' in d:
    print d['hello']
```

Façons courtes de manipuler des listes

Les listes en compréhensions fournissent un manière puissante et concise de travailler avec les listes. En outre, les fonctions `map()` et `filter()` peuvent effectuer des opérations sur des listes en utilisant une syntaxe différente et plus concise.

Mauvais :

```
# Filter elements greater than 4
a = [3, 4, 5]
b = []
for i in a:
    if i > 4:
        b.append(i)
```

Bon :

```
a = [3, 4, 5]
b = [i for i in a if i > 4]
# Or:
b = filter(lambda x: x > 4, a)
```

Mauvais :

```
# Add three to all list members.
a = [3, 4, 5]
for i in range(len(a)):
    a[i] += 3
```

Bon :

```
a = [3, 4, 5]
a = [i + 3 for i in a]
# Or:
a = map(lambda i: i + 3, a)
```

Utilisez `enumerate()` tient un compte de votre position dans la liste.

```
a = [3, 4, 5]
for i, item in enumerate(a):
    print i, item
# prints
# 0 3
# 1 4
# 2 5
```

La fonction `enumerate()` a une meilleure lisibilité que la gestion d'un compteur manuellement. De plus, elle est plus optimisés pour les itérateurs.

Lire depuis un fichier

Utilisez la syntaxe `with open` pour lire depuis des fichiers. Cela fermera automatiquement les fichiers pour vous.

Mauvais :

```
f = open('file.txt')
a = f.read()
print a
f.close()
```

Bon :

```
with open('file.txt') as f:
    for line in f:
        print line
```

La déclaration `with` est meilleure parce qu'elle assure que vous fermez toujours le fichier, même si une exception est levée à l'intérieur du block `with`.

Continuations de ligne

Quand une ligne logique de code est plus longue que la limite acceptée, vous devez la diviser sur plusieurs lignes physiques. L'interpréteur Python rejoindra lignes consécutives si le dernier caractère de la ligne est une barre oblique inverse. Ceci est utile dans certains cas, mais doit généralement être évitée en raison de sa fragilité : un espace blanc ajouté à la fin de la ligne, après la barre oblique inverse, va casser le code et peut avoir des résultats inattendus.

Une meilleure solution est d'utiliser des parenthèses autour de vos éléments. Avec une parenthèse non fermée laissée à la fin d'une ligne, l'interpréteur Python joindra la ligne suivante jusqu'à ce que les parenthèses soient fermées. Le même comportement est valable pour des accolades et des crochets.

Mauvais :

```
my_very_big_string = """For a long time I used to go to bed early. Sometimes, \
    when I had put out my candle, my eyes would close so quickly that I had not even \
    time to say "I'm going to sleep."""

from some.deep.module.inside.a.module import a_nice_function, another_nice_function, \
    yet_another_nice_function
```

Bon :

```
my_very_big_string = (
    "For a long time I used to go to bed early. Sometimes, "
    "when I had put out my candle, my eyes would close so quickly "
    "that I had not even time to say "I'm going to sleep.""
```

```
)

from some.deep.module.inside.a.module import (
    a_nice_function, another_nice_function, yet_another_nice_function)
```

Cependant, le plus souvent, avoir à couper une longue ligne logique est un signe que vous essayez de faire trop de choses en même temps, ce qui peut gêner la lisibilité.

Lire du bon code

L'un des principaux principes derrière la conception de Python est la création d'un code lisible. La motivation derrière cette conception est simple : La première chose que les programmeurs Python font est de lire le code.

L'un des secrets pour devenir un grand programmeur Python est de lire, de comprendre et d'appréhender un excellent code.

Une excellent code suit généralement les lignes directrices énoncées dans *Style de code*, et fait de son mieux pour exprimer une intention claire et concise au lecteur.

Ce qui est inclus ci-dessous est une liste de projets Python recommandés pour lecture. Chacun de ces projets est un parangon de codage Python.

- Howdoi Howdoi est un outil de recherche de code, écrit en Python.
- Flask Flask est un microframework pour Python basé sur Werkzeug et Jinja2. Il est destiné pour bien démarrer très rapidement et a été développé avec les meilleures intentions à l'esprit.
- Diamond Diamond est un daemon python qui recueille les métriques et les publie vers Graphite ou d'autres backends. Il est capable de collecter les métriques du cpu, de la mémoire, du réseau, des i/o, de la charge et des disques. En outre, il dispose d'une API pour implémenter des collecteurs personnalisés pour la collecte des métriques à partir de presque toutes les sources.
- Werkzeug Werkzeug a commencé comme simple collection de divers utilitaires pour les applications WSGI et est devenu l'un des modules utilitaire WSGI les plus avancées. Il comprend un débogueur puissant, la gestion des objets request et response complet, des utilitaires HTTP pour gérer les tags d'entité, les en-têtes de contrôle de cache, les dates HTTP, la gestion des cookies, le téléchargement de fichiers, un système puissant de routage URL et un tas de modules addons contribués par la communauté.
- Requests Requests est une bibliothèque HTTP sous licence Apache 2, écrite en Python, pour les êtres humains.
- Tablib Tablib est une bibliothèque pour la gestion de jeux de données tabulaires, agnostique en terme de format, écrite en Python.

À faire

Inclure des exemples de code de code exemplaire de chacun des projets énumérés. Expliquer pourquoi c'est un excellent code. Utilisez des exemples complexes.

À faire

Expliquer les techniques pour identifier rapidement les structures de données, les algorithmes et déterminer ce que le code fait.

Documentation

La lisibilité est une priorité pour les développeurs Python, à la fois dans la documentation du projet et du code. Suivre quelques bonnes pratiques simples peut vous sauver, vous et d'autres, beaucoup de temps.

Documentation de projet

Un fichier `README` à la racine du répertoire devrait donner des informations générales à la fois pour les utilisateurs et les mainteneurs d'un projet. Il devrait être du texte brut ou écrit dans avec un markup très facile à lire, par exemple *reStructuredText* ou Markdown. Il devrait contenir quelques lignes expliquant le but du projet ou de la bibliothèque (sans supposer que l'utilisateur ne sait rien sur le projet), l'URL de la source principale du logiciel, ainsi que des informations de crédit basiques. Ce fichier est le principal point pour les lecteurs du code.

Un fichier `INSTALL` est moins nécessaire avec Python. Les instructions d'installation sont souvent réduites à une commande, comme `pip install module` ou `python setup.py install` et ajoutée au fichier `README`.

Un fichier `LICENSE` devrait *toujours* être présent et préciser la licence sous laquelle le logiciel est mis à la disposition du public.

Un fichier `TODO` file ou une section `TODO` dans le `README` devrait lister le développement prévu pour le code.

Un fichier `CHANGELOG` ou une section dans dans le `README` doit compiler un bref aperçu des changements dans la base de code pour les versions les plus récentes.

Publication de projet

Selon le projet, la documentation peut inclure tout ou partie des composants suivants :

— Une *introduction* devrait montrer une très bref aperçu de ce qui peut être fait avec le produit, en utilisant un ou deux cas d'utilisation extrêmement simplifiés. C'est le pitch de 30 seconds de votre projet.
— Un *tutoriel* devrait montrer certains cas d'utilisation de base plus en détail. Le lecteur suivra une procédure pas à pas pour configurer un prototype fonctionnel.
— Une *référence d'API* est généralement générée à partir du code (voir *docstrings*). Elle liste toutes les interfaces, les paramètres et les valeurs de retour publiques.
— *La documentation développeur* est destinée aux contributeurs potentiels. Cela peut inclure les conventions de code et la stratégie de conception générale du projet.

Sphinx

Sphinx est de loin l'outil le plus populaire de la documentation Python. **Utilisez-le** Il convertit le langage de markup *reStructuredText* dans une gamme de formats de sortie incluant HTML, LaTeX (pour les versions imprimables PDF), les pages de manuel et le texte brut.

Il y a aussi un hébergement **génial** et **gratuit** pour votre documentation Sphinx : Read The Docs. Utilisez-le. Vous pouvez le configurer avec des hooks de commit liés à votre dépôt source. Ainsi, la re-génération de votre documentation se fera automatiquement.

Note : Sphinx est célèbre pour sa production d'API, mais il fonctionne aussi bien pour la documentation générale de projet. Ce guide est construit avec Sphinx et est hébergé sur Read The Docs

---

reStructuredText

La plupart de la documentation Python est écrite avec reStructuredText. C'est comme le Markdown avec toutes les extensions optionnelles intégrées.

Le reStructuredText Primer et le reStructuredText Quick Reference devraient vous aider à vous familiariser avec sa syntaxe.

Conseils de documentation de code

Les commentaires clarifient le code et ils sont ajoutés dans le but de rendre le code plus facile à comprendre. En Python, les commentaires commencent par un dièse (signe dièse)(#). En Python, les *docstrings* décrivent les modules, les classes et les fonctions :

```
def square_and_rooter(x):
    """Returns the square root of self times self."""
    ...
```

En général, suivez la section des commentaires de la **PEP 8#comments** (le "guide de style Python"). Plus d'informations sur les docstrings peuvent être trouvées sur la **PEP 0257#specification** (Le guide convention Docstring).

Commenter des section de code

N'utilisez pas des chaînes avec triple-guillemets pour commenter le code. Ce n'est pas une bonne pratique, parce que les outils orientés lignes en ligne de commande tels que grep ne seront pas conscients que le code commenté est inactif. Il est préférable d'ajouter des dièses au niveau d'indentation correct pour chaque ligne commentée. Votre éditeur a probablement la capacité à faire cela facilement, et cela vaut la peine d'apprendre le bouton pour commenter/décommenter.

Docstrings et la magie

Certains outils utilisent docstrings pour intégrer un comportement étant plus que de la documentation, tels que la logique de test unitaire. Ceux-ci peuvent être bien, mais vous ne vous tromperez jamais avec "voici ce que cela fait."

Docstrings versus commentaires de bloc

Ceux-ci ne sont pas interchangeables. Pour une fonction ou une classe, le premier bloc de commentaire est une note d'un programmeur. Le docstring décrit le *fonctionnement* de la fonction ou de la classe :

```
# This function slows down program execution for some reason.
def square_and_rooter(x):
    """Returns the square root of self times self."""
    ...
```

Autres outils

Vous pouvez voir ceux-là dans la nature. Utilisez *Sphinx*.

Pycco Pycco est un "générateur de documentation basé sur un style de Programmation lettrée" et est un port de Docco en node.js. Il transforme le code en un rendu HTML avec code et documentation côte à côte.

Ronn Ronn génère les manuels Unix. Il convertit des fichiers texte lisibles par l'homme en roff pour affichage dans le terminal, et également en format HTML pour le web.

Epydoc Epydoc est interrompue. Utilisez *Sphinx* à la place.

MkDocs MkDocs est un générateur de site statique simple et rapide qui est orientée vers la construction de la documentation de projet avec Markdown.

Tester votre code

Tester votre code est très important.

S'habituer à écrire le code de test et le code en cours d'exécution en parallèle est maintenant considéré comme une bonne habitude. Utilisé à bon escient, cette méthode vous aide à définir plus précisément l'intention de votre code et à avoir une architecture plus découplée.

Quelques règles générales de test :

- Un test unitaire doit se concentrer sur un tout petit morceau de fonctionnalité et prouver qu'il est correct.
- Chaque test unitaire doit être complètement indépendant. Chacun d'eux doit être capable de s'exécuter seul, et aussi à l'intérieur de la suite de tests, indépendamment de l'ordre dans lesquels ils sont appelés. L'implication de cette règle est que chaque test doit être chargé avec un nouveau jeu de données et peut avoir à faire un peu de nettoyage après. Cela est généralement géré par les méthodes `setUp()` et `tearDown()`.
- Essayez très fort de faire des tests qui s'exécutent vite. Si un seul test a besoin de plus de quelques millisecondes pour s'exécuter, le développement sera ralenti ou les tests ne seront pas exécutés aussi souvent que ce serait souhaitable. Dans certains cas, les tests ne peuvent pas être rapides parce qu'ils ont besoin d'une structure de données complexes sur laquelle travailler, et cette structure de données doit être chargée chaque fois que le test s'exécute. Gardez ces tests plus lourds dans une suite de tests séparés qui est gérée par une tâche planifiée, et exécutez tous les autres tests aussi souvent que nécessaire.
- Apprenez vos outils et apprenez à gérer un seul test ou une série de tests. Puis, lors du développement d'une fonction à l'intérieur d'un module, exécutez cette fonction de tests très souvent, idéalement automatiquement lorsque vous enregistrez le code.
- Exécutez toujours la suite de tests complète avant une session de codage, et exécutez-la à nouveau après. Cela vous donnera plus de confiance en vérifiant que que vous n'avez rien cassé dans le reste du code.
- C'est une bonne idée d'implémenter un hook qui exécute tous les tests avant de pousser le code vers un dépôt partagé.
- Si vous êtes au milieu d'une session de développement et avez à interrompre votre travail, c'est une bonne idée d'écrire un test unitaire cassé sur ce que vous voulez développer prochainement. En reprenant votre travail, vous aurez un pointeur à l'endroit où vous étiez et pourrez revenir plus rapidement sur la bonne voie.
- La première étape lorsque vous débugguez votre code est d'écrire un nouveau test localisant exactement le bug. Bien qu'il ne soit pas toujours possible de faire, ces tests pour attraper les bugs sont parmi les morceaux les plus précieux de code dans votre projet.
- Utilisez des noms longs et descriptifs pour les fonctions de test. Le guide de style ici est légèrement différent de celui du code s'exécutant, où les noms courts sont souvent préférés. La raison est de tester les fonctions qui ne sont jamais appelées explicitement. `square()` ou même `sqr()` est ok dans un code en cours d'exécution, mais dans le code de test, vous auriez des noms tels que `test_square_of_number_2()`, `test_square_negative_number()`. Ces noms de fonction sont affichés quand un test échoue, et devraient être aussi descriptifs que possible.
- Quand quelque chose va mal ou doit être changé, et si votre code a une bonne série de tests, vous ou d'autres mainteneurs allez vous reposer en grande partie sur la suite de tests pour corriger le problème ou modifier un comportement donné. Par conséquent, le code de test sera lu autant ou même plus que le code en cours d'exécution. Un test unitaire dont le la finalité est incertaine est pas très utile dans ce cas.
- Une autre utilisation du code de test est comme une introduction aux nouveaux développeurs. Quand quelqu'un aura à travailler sur la base de code, exécuter et lire le code de test lié est souvent le mieux qu'ils peuvent faire. Ils vont ou devraient découvrir les points chauds, où la plupart des difficultés sont rencontrées, et les cas limites. Si ils doivent ajouter des fonctionnalités, la première étape devrait être d'ajouter un test et, par ce moyen, de s'assurer que la nouvelle fonctionnalité est pas déjà un chemin de travail qui n'a pas été branché sur l'interface.

Les basiques

Unittest

`unittest` est le module de test "tout en un" dans la bibliothèque standard Python. Son API sera familière à quiconque a déjà utilisé une des séries d'outils JUnit/nUnit/CppUnit.

Créer des cas de test est réalisé en faisant des sous-classes de `unittest.TestCase`.

```
import unittest

def fun(x):
    return x + 1

class MyTest(unittest.TestCase):
    def test(self):
        self.assertEqual(fun(3), 4)
```

A partir de Python 2.7, unittest comprend également ses propres mécanismes de découverte de tests.

> unittest dans la documentation de la bibliothèque standard

Doctest

Le module `doctest` recherche des morceaux de texte qui ressemblent à des sessions de Python interactifs en docstrings, puis exécute ces sessions pour vérifier qu'ils fonctionnent exactement comme indiqué.

Les doctests ont un cas d'utilisation différent des tests unitaires appropriés : ils sont généralement moins détaillés et ne capturent pas des cas particuliers ou des bugs de régression obscurs. Ils sont utiles en tant que documentation expressive des principaux cas d'utilisation d'un module et de ses composants. Cependant, les doctests doivent exécuter automatiquement à chaque fois que la suite de tests complète s'exécute.

Un simple doctest dans une fonction :

```
def square(x):
    """Return the square of x.

    >>> square(2)
    4
    >>> square(-2)
    4
    """

    return x * x

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

Lors de l'exécution de ce module à partir de la ligne de commande comme dans `python module.py`, les doctests vont s'exécuter et se plaindre si rien ne se comporte comme décrit dans les docstrings.

Outils

py.test

py.test est une alternative sans boilerplate au module unittest standard Python.

```
$ pip install pytest
```

En dépit d'être un outil de test plein de fonctionnalités et extensible, il bénéficie d'une syntaxe simple. Créer une suite de tests est aussi facile qu'écrire un module avec un couple de fonctions :

```
# content of test_sample.py
def func(x):
    return x + 1

def test_answer():
    assert func(3) == 5
```

et ensuite en exécutant la commande *py.test*

```
$ py.test
========================== test session starts ===========================
platform darwin -- Python 2.7.1 -- pytest-2.2.1
collecting ... collected 1 items

test_sample.py F

================================= FAILURES =================================
_____________________________________ test_answer _____________________________________

    def test_answer():
>       assert func(3) == 5
E       assert 4 == 5
E        +  where 4 = func(3)

test_sample.py:5: AssertionError
========================= 1 failed in 0.02 seconds =========================
```

est beaucoup moins de travail que ce qui serait nécessaire pour la fonctionnalité équivalente avec le module unittest !

py.test

Nose

nose étend unittest pour rendre les tests plus faciles.

```
$ pip install nose
```

nose fournit la découverte automatique de tests pour vous épargner les tracas de créer manuellement des suites de tests. Il fournit également de nombreux plugins pour des fonctionnalités telles que la sortie de test compatible xUnit, les rapports sur la couverture et la sélection de tests.

nose

tox

tox est un outil pour automatiser la gestion de l'environnement de test et les tests ciblant des configurations d'interpréteurs multiples

```
$ pip install tox
```

tox vous permet de configurer des matrices de test multi-paramètres complexes via un simple fichier de configuration de type INI.

tox

Unittest2

unittest2 est un portage du module unittest Python 2.7 qui a une API améliorée et de meilleures assertions par rapport à celui disponible dans les versions précédentes de Python.

Si vous utilisez Python 2.6 ou inférieur, vous pouvez l'installer avec pip

```
$ pip install unittest2
```

Vous pouvez vouloir faire l'import du module sous le nom unittest pour rendre le portage du code pour les nouvelles versions du module plus facile dans le futur

```
import unittest2 as unittest

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

De cette façon, si jamais vous basculez à une version plus récente de Python et n'avez plus besoin du module unittest2, vous pouvez simplement changer l'import dans votre module de test sans avoir besoin de changer aucun autre code.

unittest2

mock

`unittest.mock` est une bibliothèque pour les tests en Python. A partir de Python 3.3, elle est disponible dans la bibliothèque standard.

Pour les versions anciennes de Python :

```
$ pip install mock
```

Il vous permet de remplacer les parties de votre système en test avec des objets mock et de faire des assertions sur la façon dont ils ont été utilisés.

Par exemple, vous pouvez monkey-patcher une méthode :

```
from mock import MagicMock
thing = ProductionClass()
thing.method = MagicMock(return_value=3)
thing.method(3, 4, 5, key='value')

thing.method.assert_called_with(3, 4, 5, key='value')
```

Pour mocker des classes ou des objets dans un module en test, utilisez le décorateur `patch`. Dans l'exemple ci-dessous, un système de recherche externe est remplacé par un mock qui retourne toujours le même résultat (mais seulement pour la durée du test).

```
def mock_search(self):
    class MockSearchQuerySet(SearchQuerySet):
        def __iter__(self):
            return iter(["foo", "bar", "baz"])
    return MockSearchQuerySet()

# SearchForm here refers to the imported class reference in myapp,
# not where the SearchForm class itself is imported from
@mock.patch('myapp.SearchForm.search', mock_search)
def test_new_watchlist_activities(self):
    # get_search_results runs a search and iterates over the result
    self.assertEqual(len(myapp.get_search_results(q="fish")), 3)
```

Mock a beaucoup d'autres façons pour que vous le configuriez et contrôliez son comportement.

mock

Logging

Le module `logging` fait parti de la bibliothèque standard de Python depuis la version 2.3. Il est succinctement décrit dans la **PEP 282**. La documentation est notoirement difficile à lire, à l'exception du basic logging tutorial.

Le logging sert deux buts :

- **Le logging de diagnostic** enregistre les événements liés au fonctionnement de l'application. Si un utilisateur appelle pour signaler une erreur, par exemple, les logs peuvent être recherchés pour avoir un contexte.
- **logging d'audit** enregistre les événements pour l'analyse des affaires. Les transactions d'un utilisateur peuvent être extraites et combinées avec d'autres détails de l'utilisateur pour les rapports ou pour optimiser un objectif d'affaires.

... ou Print ?

La seule fois que `print` est une meilleure option que le logging est lorsque l'objectif est d'afficher une déclaration d'aide pour une application en ligne de commande. D'autres raisons pour lesquelles le logging est mieux que `print` :

- Le log record, qui a été créé avec chaque événement de logging, contient des informations de diagnostic facilement disponibles tels que le nom du fichier, le chemin complet, la fonction et le numéro de ligne de l'événement de logging.
- Les événements loggués dans les modules inclus sont automatiquement accessibles via la racine du logger au flux de logs de votre application, à moins que vous ne les filtriez.
- Le logging peut être réduite au silence de manière sélective en utilisant la méthode `logging.Logger.setLevel()` ou désactivée en définissant l'attribut `logging.Logger.disabled` à `True`.

Logging dans une bibliothèque

Les notes pour configuring logging for a library sont dans le logging tutorial. Parce que *l'utilisateur*, pas la bibliothèque, devrait dicter ce qui se passe quand un événement de logging a lieu, une admonition à accepter est de se répéter :

Note : Il est fortement recommandé de ne pas ajouter de handlers autres que NullHandler aux loggers de votre bibliothèque.

Les meilleures pratiques lors de l'instanciation des loggers dans une bibliothèque est de seulement les créer en utilisant la variable globale `__name__` : le module `logging` crée une hiérarchie de loggers en utilisant la notation point, donc utiliser `__name__` assure l'absence de collisions de noms.

Voici un exemple de bonne pratique depuis la requests source – placez ceci dans votre `__init__.py`

```
# Set default logging handler to avoid "No handler found" warnings.
import logging
try:  # Python 2.7+
    from logging import NullHandler
except ImportError:
    class NullHandler(logging.Handler):
        def emit(self, record):
            pass
```

```
logging.getLogger(__name__).addHandler(NullHandler())
```

Logging dans une application

Le site twelve factor app, une référence faisant autorité pour les bonnes pratiques dans le développement d'application, contient une section sur les meilleures pratiques de logging. Elle prône avec insistance pour le traitement des événements de journal comme un flux d'événements, et pour envoyer ce flux d'événements à la sortie standard pour être traité par l'environnement d'application.

Il y a au moins 3 manières de configurer un logger :

- **En utilisant un fichier formaté sous forme INI**
  - **Avantages** : possible de mettre à jour la configuration lors de l'exécution en utilisant la fonction `logging.config.listen()` pour écouter sur un socket.
  - **Inconvénients** : moins de contrôle (*par exemple* pour des loggers ou filtres comme sous-classes) que possible lors de la configuration d'un logger dans le code.
- **En utilisant un dictionnaire ou un fichier formaté sous forme JSON :**
  - **Avantages** : en plus de la mise à jour tout en exécutant, il est possible de charger à partir d'un fichier en utilisant le module `json`, dans la bibliothèque standard depuis Python 2.6.
  - **Inconvénients** : moins de contrôle que quand configuration d'un logger dans le code.
- **En utilisant du code :**
  - **Avantages** : contrôle complet sur la configuration.
  - **Inconvénients** : modifications nécessitent un changement du code source.

Exemple de configuration via un fichier INI

Disons que le fichier est nommé `logging_config.ini`. Plus de détails pour le format de fichier se trouvent dans la section logging configuration du logging tutorial.

```
[loggers]
keys=root

[handlers]
keys=stream_handler

[formatters]
keys=formatter

[logger_root]
level=DEBUG
handlers=stream_handler

[handler_stream_handler]
class=StreamHandler
level=DEBUG
formatter=formatter
args=(sys.stderr,)

[formatter_formatter]
format=%(asctime)s %(name)-12s %(levelname)-8s %(message)s
```

Ensuite, utilisez `logging.config.fileConfig()` dans le code :

```
import logging
from logging.config import fileConfig

fileConfig('logging_config.ini')
logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Exemple de configuration via un dictionnaire

A partir de Python 2.7, vous pouvez utiliser un dictionnaire avec les détails de configuration. La **PEP 391** contient une liste des éléments obligatoires et optionnels dans le dictionnaire de configuration.

```
import logging
from logging.config import dictConfig

logging_config = dict(
    version = 1,
    formatters = {
        'f': {'format':
              '%(asctime)s %(name)-12s %(levelname)-8s %(message)s'}
        },
    handlers = {
        'h': {'class': 'logging.StreamHandler',
              'formatter': 'f',
              'level': logging.DEBUG}
        },
    root = {
        'handlers': ['h'],
        'level': logging.DEBUG,
        },
)

dictConfig(logging_config)

logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Exemple de configuration directement dans le code

```
import logging

logger = logging.getLogger()
handler = logging.StreamHandler()
formatter = logging.Formatter(
        '%(asctime)s %(name)-12s %(levelname)-8s %(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Pièges communs

Pour la plus grande partie, Python vise à être un langage propre et cohérent qui permet d'éviter des surprises. Cependant, il y a quelques cas qui peuvent être sources de confusion pour les nouveaux arrivants.

Certains de ces cas sont intentionnels, mais peuvent être potentiellement surprenants. Certains pourraient sans doute être considérés comme des verrues du langage. En général, ce qui suit est une collection de comportements potentiellement délicats qui pourraient sembler étranges à première vue, mais qui sont généralement raisonnables une fois que vous êtes au courant de la cause sous-jacente de cette surprise.

Arguments par défaut mutables

Apparemment la surprise *la plus* commune que les nouveaux programmeurs Python rencontre est le traitement Python des arguments par défaut mutables dans les définitions de fonction.

Ce que vous écrivez

```
def append_to(element, to=[]):
    to.append(element)
    return to
```

Qu'est-ce que vous auriez pu attendre qu'il se passe

```
my_list = append_to(12)
print my_list

my_other_list = append_to(42)
print my_other_list
```

Une nouvelle liste est créée chaque fois que la fonction est appelée si un second argument n'est pas fourni, de sorte que la sortie est :

```
[12]
[42]
```

Ce qui se passe

```
[12]
[12, 42]
```

Une nouvelle liste est créée *une seule fois* quand la fonction est définie, et la même liste est utilisée dans chaque appel successif.

Les arguments par défaut de Python sont évalués *une seule fois* lorsque la fonction est définie, pas chaque fois que la fonction est appelée (comme c'est le cas, disons en Ruby). Cela signifie que si vous utilisez un argument par défaut mutable et le mutez, vous *aurez* muté l'argument ici et pour tous les futurs appels à la fonction aussi.

Ce que vous devriez faire à la place

Créez un nouvel objet à chaque fois que la fonction est appelée, en utilisant un argument par défaut pour signaler que aucun argument n'a été fourni (`None` est souvent un bon choix).

```
def append_to(element, to=None):
    if to is None:
        to = []
    to.append(element)
    return to
```

Quand le piège n'est pas un piège

Parfois, vous pouvez spécifiquement "exploiter" (lisez : utilisé comme prévu) ce comportement pour maintenir l'état entre les appels d'une fonction. C'est souvent fait lors de l'écriture d'une fonction de mise en cache.

Closures des bindings tardives

Une autre source de confusion est la manière dont Python bind ses variables dans les closures (ou dans la portée globale entourante)

Ce que vous écrivez

```
def create_multipliers():
    return [lambda x : i * x for i in range(5)]
```

Qu'est-ce que vous auriez pu attendre qu'il se passe

```
for multiplier in create_multipliers():
    print multiplier(2)
```

Une liste contenant cinq fonctions qui ont chacun leur propre variable `i` fermée sur elle-même qui multiplie leur argument, produisant :

```
0
2
4
6
8
```

Ce qui se passe

```
8
8
8
8
8
```

Cinq fonctions sont créées ; au lieu que toutes ne multiplient juste `x` par 4.

Les closures de Python sont des *late binding*. Cela signifie que les valeurs des variables utilisées dans les closures sont regardées au moment où la fonction interne est appelée.

Ici, chaque fois que *n'importe lesquelles* des fonctions retournées sont appelées, la valeur de `i` est recherché dans la portée environnante au moment de l'appel. D'ici là, la boucle est terminée et `i` est laissé à sa valeur finale de 4.

Ce qui est particulièrement déplaisant sur ce piège est la désinformation apparemment répandue que cela a quelque chose à voir avec les lambdas en Python. Les fonctions créées avec une expression `lambda` ne sont en aucune façon particulière, et en fait le même comportement est exposé en utilisant simplement un ordinaire `def` :

```
def create_multipliers():
    multipliers = []

    for i in range(5):
        def multiplier(x):
            return i * x
        multipliers.append(multiplier)

    return multipliers
```

Ce que vous devriez faire à la place

La solution la plus générale est sans doute une forme de hack. En raison du comportement de Python déjà mentionné concernant l'évaluation des arguments par défaut aux fonctions (voir *Arguments par défaut mutables*), vous pouvez créer une closure qui se bind immédiatement à ses arguments en utilisant un argument par défaut comme ceci :

```
def create_multipliers():
    return [lambda x, i=i : i * x for i in range(5)]
```

Alternativement, vous pouvez utiliser la fonction functools.partial :

```
from functools import partial
from operator import mul

def create_multipliers():
    return [partial(mul, i) for i in range(5)]
```

Quand le piège n'est pas un piège

Parfois, vous voulez que vos closures se comportent de cette façon. Les late binding sont biens dans beaucoup de situations. Boucler pour créer des fonctions uniques est malheureusement un cas où ils peuvent causer le hoquet.

Fichiers Bytecode (.pyc) partout !

Par défaut, lors de l'exécution du code Python à partir de fichiers, l'interpréteur Python va automatiquement écrire une version bytecode de ce fichier sur le disque, par exemple `module.pyc`.

Ces fichiers `.pyc` ne doivent pas être versionnés dans vos dépôts de code source.

Théoriquement, ce comportement est activé par défaut, pour des raisons de performance. Sans ces fichiers bytecode présents, Python regénérerait le bytecode chaque fois que le fichier est chargé.

Désactiver les fichiers Bytecode (.pyc)

Heureusement, le processus de génération du bytecode est extrêmement rapide, et n'est pas quelque chose dont vous avez besoin de vous soucier quand vous développez votre code.

Ces fichiers sont ennuyeux, débarrassons-nous d'eux !

```
$ export PYTHONDONTWRITEBYTECODE=1
```

Avec la variable d'environnement `$PYTHONDONTWRITEBYTECODE` définie, Python n'écrira pas plus longtemps ces fichiers sur le disque, et votre environnement de développement restera agréable et propre.

Je recommande la définition de cette variable d'environnement dans votre `~/.profile`.

Enlever les fichiers Bytecode (.pyc)

Voici une astuce sympathique pour enlever tous ces fichiers, s'ils existent déjà :

```
$ find . -name "*.pyc" -delete
```

Exécutez ceci depuis le répertoire racine de votre projet, et tous les fichiers `.pyc` vont soudainement disparaître. Beaucoup mieux.

Choix d'une licence

Votre publication de source *nécessite* une licence. Aux Etats-Unis, si aucune licence est spécifiée, les utilisateurs ont pas le droit légal de télécharger, modifier ou distribuer. En outre, les gens ne peuvent pas contribuer à votre code, sauf si vous leur dites quelles sont les règles pour jouer. Le choix d'une licence est compliqué, alors voici quelques conseils :

Open source. Il y a beaucoup de licences open source disponibles pour faire son choix.

En général, ces licences ont tendance à tomber dans l'une des deux catégories :

1. les licences qui se concentrent davantage sur la liberté de l'utilisateur à faire ce qu'il veut avec le logiciel (ce sont les licences open source plus permissives telles que le MIT, BSD et Apache).
2. les licences qui se concentrent davantage à faire en sorte que le code lui-même - y compris toutes les modifications apportées et distribuées avec lui - restent toujours libres (ce sont les licences de logiciels libres moins permissives telles que la GPL et LGPL).

Ces dernières sont moins permissives dans le sens où elles ne permettent pas à quelqu'un d'ajouter du code au logiciel et de le distribuer sans inclure également le code source suite aux changements.

Pour vous aider à en choisir une pour votre projet, il y a un site pour l'aide au choix de licence, **utilisez-le**.

Plus permissives
- PSFL (Python Software Foundation License) – pour la contribution à Python lui-même
- MIT / BSD / ISC
  - MIT (X11)
  - New BSD
  - ISC
- Apache

Moins permissives :
- LGPL
- GPL
  - GPLv2
  - GPLv3

Un bon aperçu des licences avec des explications sur ce que l'on peut, ne peut pas, et ce qu'on doit faire en utilisant un logiciel en particulier peut être trouvé sur tl;drLegal.

CHAPITRE 3

Guide de scénario pour des applications Python

Cette partie du guide se concentre sur les conseils pour les outils et modules basés sur les différents scénarios.

Applications réseau

HTTP

Le “Hypertext Transfer Protocol” (HTTP) est un protocole d’application pour les systèmes d’information distribués, collaboratifs et hypermédia. HTTP est le fondement de la communication de données pour le World Wide Web.

Requests

Le module urllib2 de la bibliothèque standard Python fournit la plupart des fonctionnalités HTTP dont vous avez besoin, mais l’API est complètement cassée. Il a été créé pour une autre période - et un web différent. Il nécessite une énorme quantité de travail (même avec les surcharges de méthode) pour effectuer la plus simple des tâches.

Requests prend la main sur tout le travail effectué sur HTTP par Python - rendant votre intégration avec des services Web parfaite. Il n’y a pas besoin d’ajouter manuellement les chaînes de requête à vos URL, ou à encoder vos données POST comme des formulaires. Keep-alive et le pooling de connexion HTTP sont 100% automatiques, motorisés par urllib3, qui est encapsulée dans Requests.

— Documentation
— PyPi
— GitHub

Systèmes distribués

ZeroMQ

ØMQ (écrit aussi ZeroMQ, 0MQ ou ZMQ) est une bibliothèque de messaging asynchrone haute performance destinée à une utilisation dans des applications distribuées scalables ou concurrentes. Il fournit une queue de messages, mais contrairement à un middleware orienté message, un système ØMQ peut fonctionner sans un broker de message dédié. La bibliothèque est conçue pour avoir une API familière dans le style des sockets.

RabbitMQ

RabbitMQ est un logiciel open source de type broker de messages qui implémente Advanced Message Queuing Protocol (AMQP). Le serveur RabbitMQ est écrit dans le langage de programmation Erlang et est construit sur le framework

Open Telecom Platform pour le clustering et le failover. Des bibliothèques clientes pour s'interfacer avec le broker sont disponibles pour tous les principaux langages de programmation.

— Page d'accueil
— Organisation GitHub

Applications Web

En tant que puissant langage de scripting adapté à la fois au prototypage rapide et à de plus gros projets, Python est largement utilisé dans le développement d'applications web.

Contexte

WSGI

Le Web Server Gateway Interface (ou "WSGI" pour faire court) est une interface standard entre les serveurs Web et les frameworks web Python. En normalisant le comportement et la communication entre les serveurs Web et les frameworks web Python, WSGI permet d'écrire du code web Python portable qui peut être déployé dans tout *serveur Web conforme à WSGI*. WSGI est documentée dans la **PEP 3333**.

Frameworks

D'une manière générale, un framework web se compose d'un ensemble de bibliothèques et un gestionnaire principal au sein duquel vous pouvez construire un code personnalisé pour implémenter une application Web (par exemple, un site web interactif). La plupart des frameworks web incluent des modèles et des utilitaires pour accomplir au moins ce qui suit :

Routing d'URL Fait correspondre une requête HTTP entrante vers un bout de code Python particulier pour être invoqué

Objets Requête et Réponse Encapsule l'information reçue ou expédiée au navigateur d'un utilisateur

Moteur de template Permet la séparation du code Python implémentant la logique d'une application de la sortie HTML (ou autre) qu'elle produit

Serveur web de développement Exécute un serveur HTTP sur les machines de développement pour permettre un développement rapide ; souvent recharge automatiquement le code côté serveur quand les fichiers sont mis à jour

Django

Django est un framework d'application web "tout en un", et est un excellent choix pour la création de sites Web orientés contenu. En fournissant utilitaires et patterns par défaut, Django a pour but de permettre de construire rapidement des applications web complexes Web, reposant sur des base de données, tout en encourageant les bonnes pratiques dans le code écrit qui l'utilise.

Django a une communauté importante et active, et de nombreux modules pré-construits réutilisables qui peuvent être incorporés dans un nouveau projet tel quel, ou personnalisé pour répondre à vos besoins.

Il y a des conférences annuelles de Django aux États-Unis et en Europe.

La majorité des nouvelles applications web Python d'aujourd'hui sont construites avec Django.

Flask

Flask est un "microframework" pour Python, et est un excellent choix pour la construction de petites applications, d'APIs et des services Web.

Construire une application avec Flask est un peu comme l'écriture des modules standards de Python, sauf certaines fonctions ont des routes qui leur sont rattachées. C'est vraiment très beau.

Plutôt que de viser à fournir tout ce dont vous pourriez avoir besoin, Flask implémente les composants noyaux les plus couramment utilisés d'un framework web, comme le routage d'URL, les objets request et response et les templates.

Si vous utilisez Flask, c'est à vous de choisir d'autres composants pour votre application, le cas échéant. Par exemple, l'accès aux base de données ou la génération et la validation de formulaires ne sont pas des fonctions natives dans Flask.

C'est super, car de nombreuses applications Web ne nécessitent pas ces fonctionnalités. Pour celles qui en ont besoin, il y a de nombreuses extensions disponibles qui peuvent répondre à vos besoins. Ou, vous pouvez facilement utiliser toute bibliothèque dont vous voulez vous-même !

Flask est le choix par défaut pour toutes les applications web Python qui ne sont pas adaptées à Django.

Tornado

Tornado est un framework web asynchrone pour Python qui a sa propre boucle d'événements. Cela lui permet de supporter nativement les WebSockets, par exemple. Les applications Tornado bien écrites sont connues pour avoir d'excellentes performances.

Je ne recommande pas d'utiliser Tornado sauf si vous pensez que vous en avez besoin.

Pyramid

Pyramid est un framework très souple se concentrant fortement sur la modularité. Il est livré avec un petit nombre de bibliothèques ("toutes en unes") intégrées, et encourage les utilisateurs à étendre ses fonctionnalités de base.

Pyramid n'a pas une grosse base d'utilisateurs, contrairement à Django et Flask. C'est un framework intelligent, mais pas un choix très populaire pour les nouvelles applications web Python de nos jours.

Serveurs web

Nginx

Nginx (prononcé "engine-x ") est un serveur web et un reverse-proxy pour HTTP, SMTP et d'autres protocoles. Il est connu pour sa haute performance, la simplicité relative, et sa compatibilité avec de nombreux serveurs d'applications (comme les serveurs WSGI). Il inclut aussi des fonctionnalités pratiques comme le load-balancing, l'authentification basique, le streaming, et d'autres encore. Conçu pour servir des sites à forte charge, Nginx est progressivement en train de devenir populaire.

Serveurs WSGI

Les serveurs WSGI autonomes utilisent généralement moins de ressources que les serveurs web traditionnels et offrent les meilleures performances [3].

3. Benchmark de serveurs WSGI Python

Gunicorn

Gunicorn (Green Unicorn) est un serveur de WSGI en pur-python utilisé pour servir des applications Python. Contrairement à d'autres serveurs web Python, il a une interface utilisateur réfléchie, et est extrêmement facile à utiliser et à configurer.

Gunicorn a des configurations par défaut saines et raisonnables. Cependant, certains autres serveurs, comme uWSGI, sont largement plus personnalisables, et par conséquent, sont beaucoup plus difficiles à utiliser efficacement.

Gunicorn est le choix recommandé pour de nouvelles applications Python web aujourd'hui.

Waitress

Waitress est un serveur de WSGI pur-python qui affirme avoir des "performances très acceptables". Sa documentation est pas très détaillée, mais elle offre quelques fonctionnalités intéressantes que Gunicorn n'a pas (par exemple la mise en mémoire tampon de requête HTTP).

Waitress gagne en popularité au sein de la communauté de développement web Python.

uWSGI

uWSGI est une pile complète pour construire des services d'hébergement. En plus de la gestion des processus, la surveillance des processus, et d'autres fonctionnalités, uWSGI agit comme un serveur d'applications pour différents langages de programmation et protocoles - y compris Python et WSGI. uWSGI peut soit être exécuté comme un routeur Web autonome, ou être exécuté derrière un serveur web complet (comme Nginx ou Apache). Dans ce dernier cas, un serveur Web peut configurer uWSGI et le fonctionnement d'une application par dessus le protocole uwsgi. Le support d'un serveur Web uWSGI permet la configuration dynamique de Python, en passant des variables d'environnement et plus de réglages. Pour plus de détails, reportez-vous aux variables magiques uWSGI.

Je ne recommande pas d'utiliser uWSGI sauf si vous savez pourquoi vous en avez besoin.

Meilleures pratiques serveur

La majorité des applications auto-hébergées Python aujourd'hui est hébergée avec un serveur WSGI tels que *Gunicorn*, soit directement, soit derrière un serveur web léger tel que *nginx*.

Les serveurs WSGI servent les applications Python pendant que le serveur Web gère les tâches mieux adaptées pour lui comme la mise à disposition des fichiers statiques, les requêtes de routage, la protection DDoS, et l'authentification basique.

Hébergement

Un Platform-as-a-Service (PaaS) est un type d'infrastructure de "cloud computing" qui fait abstraction et gère l'infrastructure, le routage et le scaling des applications Web. Lorsque vous utilisez un PaaS, les développeurs d'applications peuvent se concentrer sur l'écriture du code de l'application plutôt que de se préoccuper des détails du déploiement.

Heroku

Heroku offre un soutien de première classe pour les applications Python 2,7–3,5

Heroku supporte tous les types d'applications Python Web, serveurs et frameworks. Les applications peuvent être développées sur Heroku gratuitement. Une fois que votre application est prête pour la production, vous pouvez passer votre application en formule Hobby ou Professional.

Heroku maintient des articles détaillés sur l'utilisation de Python avec Heroku, ainsi que des instructions pas à pas sur comment configurer votre première application.

Heroku est le PaaS recommandé pour le déploiement d'applications web Python aujourd'hui.

Gondor

Gondor est un PaaS spécialisé pour le déploiement d'applications Django et Pinax. Gondor recommande Django version 1.6 et prend en charge toutes les applications WSGI sur Python version 2.7. Gondor peut configurer automatiquement votre site Django si vous utilisez `local_settings.py` pour les informations de configuration spécifiques au site.

Gondor a un guide sur le déploiement de projets Django.

Gondor est géré par une petite entreprise et se concentre pour aider les entreprises à trouver le succès avec Python et Django.

Templating

La plupart des applications WSGI répondent à des requêtes HTTP pour servir du contenu en HTML ou d'autres langages de balisage. Au lieu de générer directement le contenu textuel depuis Python, le concept de séparation des problèmes nous conseille d'utiliser des templates. Un moteur de template gère un ensemble de fichiers de templates, avec un système de hiérarchie et d'inclusion pour éviter des répétitions inutiles, et est en charge du rendu (génération) du contenu proprement dit, remplissant le contenu statique des templates avec le contenu dynamique généré par l'application .

Comme les fichiers de template sont parfois écrits par des designers ou développeurs front-end, il peut être difficile de gérer une complexité croissante.

Quelques bonnes pratiques générales s'appliquent à la partie de l'application qui passe le contenu dynamique au moteur de template, et aux templates eux-mêmes

- Les fichiers de template ne doivent passer que le contenu dynamique nécessaire pour faire le rendu du template. Évitez la tentation de passer du contenu supplémentaire "juste au cas où" : il est plus facile d'ajouter une variable manquante en cas de besoin que de supprimer une variable inutilisée plus tard.
- Beaucoup de moteurs de template des déclarations complexes ou des affectations dans le template lui-même, et beaucoup autorisent qu'un peu de code Python soit évalué dans les templates. Cette facilité peut entraîner une augmentation incontrôlée de la complexité, et souvent rendre plus difficile de trouver les bugs.
- Il est souvent nécessaire de mélanger les templates JavaScript avec des templates HTML. Une approche saine à cette conception est d'isoler les parties où le template HTML passe un contenu variable au code JavaScript.

Jinja2

Jinja2 est un moteur de template très respecté.

Il utilise un langage de template basé sur du texte et peut donc être utilisé pour générer n'importe quel type de markup, et pas seulement du HTML. Il permet la personnalisation des filtres, de tags, de tests et des globals. Il dispose de nombreuses améliorations par rapport au système de template de Django.

Voici quelques balises html importantes dans Jinja2 :

```
{# This is a comment #}

{# The next tag is a variable output: #}
{{title}}

{# Tag for a block, can be replaced through inheritance with other html code #}
```

```
{% block head %}
<h1>This is the head!</h1>
{% endblock %}

{# Output of an array as an iteration #}
{% for item in list %}
<li>{{ item }}</li>
{% endfor %}
```

L'ensemble des listings suivants est un exemple de site web en combinaison avec le serveur web Tornado. Tornado est pas très compliqué à utiliser.

```
# import Jinja2
from jinja2 import Environment, FileSystemLoader

# import Tornado
import tornado.ioloop
import tornado.web

# Load template file templates/site.html
TEMPLATE_FILE = "site.html"
templateLoader = FileSystemLoader( searchpath="templates/" )
templateEnv = Environment( loader=templateLoader )
template = templateEnv.get_template(TEMPLATE_FILE)

# List for famous movie rendering
movie_list = [[1,"The Hitchhiker's Guide to the Galaxy"],[2,"Back to future"],[3,"Matrix"]]

# template.render() returns a string which contains the rendered html
html_output = template.render(list=movie_list,
                        title="Here is my favorite movie list")

# Handler for main page
class MainHandler(tornado.web.RequestHandler):
    def get(self):
        # Returns rendered template string to the browser request
        self.write(html_output)

# Assign handler to the server root  (127.0.0.1:PORT/)
application = tornado.web.Application([
    (r"/", MainHandler),
])
PORT=8884
if __name__ == "__main__":
    # Setup the server
    application.listen(PORT)
    tornado.ioloop.IOLoop.instance().start()
```

Le fichier `base.html` peut être utilisé comme base pour toutes les pages du site qui sont par exemple implémentées dans le bloc de contenu.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{{title}} - My Webpage</title>
</head>
```

```
<body>
<div id="content">
    {# In the next line the content from the site.html template will be added #}
    {% block content %}{% endblock %}
</div>
<div id="footer">
    {% block footer %}
    &copy; Copyright 2013 by <a href="http://domain.invalid/">you</a>.
    {% endblock %}
</div>
</body>
```

Le prochain listing est notre page de site (`site.html`) chargé dans l'application Python qui étend `base.html`. Le bloc de contenu est automatiquement défini dans le bloc correspondant dans la page `base.html`.

```
<!{% extends "base.html" %}
{% block content %}
    <p class="important">
    <div id="content">
        <h2>{{title}}</h2>
        <p>{{ list_title }}</p>
        <ul>
             {% for item in list %}
             <li>{{ item[0]}} :  {{ item[1]}}</li>
             {% endfor %}
        </ul>
    </div>
    </p>
{% endblock %}
```

Jinja2 est la bibliothèque de templating recommandée pour les nouvelles applications Python web.

Chameleon

Les Page Templates de Chameleon sont l'implémentation dans un moteur de template HTML/XML des syntaxes Template Attribute Language (TAL), TAL Expression Syntax (TALES), et Macro Expansion TAL (Metal).

Chameleon est disponible pour Python 2.5 et plus (y compris 3.x et pypy), et est couramment utilisé par le Framework Pyramid.

"Page Templates" ajoute au sein de votre structure de document des éléments d'attributs spéciaux et du texte de markup. En utilisant un ensemble de constructions de langage simples, vous contrôlez le flux du document, la répétition d'élément , le remplacement de texte et de la traduction. A cause de la syntaxe basée sur des attributs, les templates de page non-rendus sont valides HTML et peuvent être regardés dans un navigateur et même édités dans des éditeurs WYSIWYG. Cela peut permettre la collaboration par aller-retour avec des designers et le prototypage avec des fichiers statiques dans un navigateur plus facile.

Le langage TAL de base est assez simple à comprendre à partir d'un exemple :

```
<html>
  <body>
  <h1>Hello, <span tal:replace="context.name">World</span>!</h1>
    <table>
      <tr tal:repeat="row 'apple', 'banana', 'pineapple'">
        <td tal:repeat="col 'juice', 'muffin', 'pie'">
           <span tal:replace="row.capitalize()" /> <span tal:replace="col" />
        </td>
      </tr>
```

```
    </table>
  </body>
</html>
```

Le pattern *<span tal :replace="expression" />* pour l'insertion de texte est suffisamment courant pour que vous ne demandiez pas une validité stricte dans vos templates non-rendus. Vous pouvez le remplacer par une syntaxe plus concise et lisible qui utilise le pattern *${expression}*, comme suit :

```
<html>
  <body>
    <h1>Hello, ${world}!</h1>
    <table>
      <tr tal:repeat="row 'apple', 'banana', 'pineapple'">
        <td tal:repeat="col 'juice', 'muffin', 'pie'">
           ${row.capitalize()} ${col}
        </td>
      </tr>
    </table>
  </body>
</html>
```

Mais gardez à l'esprit que la syntaxe complète *<span tal :replace="expression">texte par défaut* permet aussi d'avoir un contenu par défaut dans le modèle non-rendu.

Venant du monde Pyramid, Chameleon n'est pas largement utilisé.

Mako

Mako est un langage de template qui compile en Python pour une performance maximale. Sa syntaxe et son api sont empruntées des meilleures parties d'autres langages de templating comme les modèles Django et Jinja2. Il est le langage de template par défaut inclus avec les frameworks web Pylons et Pyramid.

Un template exemple en Mako ressemble à ceci :

```
<%inherit file="base.html"/>
<%
    rows = [[v for v in range(0,10)] for row in range(0,10)]
%>
<table>
    % for row in rows:
        ${makerow(row)}
    % endfor
</table>

<%def name="makerow(row)">
    <tr>
    % for name in row:
        <td>${name}</td>\
    % endfor
    </tr>
</%def>
```

Pour faire le rendu d'un template très basique, vous pouvez faire ce qui suit :

```
from mako.template import Template
print(Template("hello ${data}!").render(data="world"))
```

Mako est bien respecté au sein de la communauté web Python.

Références

Scraping HTML

Web Scraping

Les sites web sont écrits en utilisant HTML, ce qui signifie que chaque page Web est un document structuré. Parfois, il serait bien d'obtenir des données à partir d'elles et de préserver la structure pendant que nous y sommes. Les sites web ne fournissent pas toujours leurs données dans des formats confortables tels que le `csv` ou `json`.

C'est là où le web scraping entre en jeu. Le web scraping est la pratique d'utiliser un programme d'ordinateur pour passer au crible d'une page Web et de recueillir les données dont vous avez besoin dans un format plus utile pour vous tout en préservant la structure de la donnée.

lxml et Requests

lxml est une bibliothèque assez étendue, écrite pour analyser des documents XML et HTML très rapidement, manipulant même des balises invalides au cours du processus. Nous allons aussi utiliser le module Requests à la place du module urllib2 natif à cause d'améliorations en terme de vitesse et de lisibilité. Vous pouvez facilement installer les deux en utilisant `pip install lxml` et `pip install requests`.

Commençons avec les imports :

```
from lxml import html
import requests
```

Ensuite, nous utiliserons `requests.get` pour récupérer la page web avec notre donnée, la parser en utilisant le module `html` et sauver les résultats dans ‘‘ tree‘‘ :

```
page = requests.get('http://econpy.pythonanywhere.com/ex/001.html')
tree = html.fromstring(page.content)
```

(Nous avons besoin d'utiliser `page.content` plutôt que `page.text` parce que `html.fromstring` attend implicitement des `bytes` en entrée.)

`tree` contient maintenant le fichier HTML entier dans une belle structure arborescente que nous pouvons parcourir de deux façons différentes : XPath et CSSSelect. Dans cet exemple, nous allons nous concentrer sur le premier.

XPath est un moyen de trouver de l'information dans des documents structurés tels que des documents HTML ou XML. Une bonne introduction à XPath est sur W3Schools .

Il existe également divers outils pour obtenir le XPath d'éléments comme Firebug pour Firefox ou l'Inspecteur Chrome. Si vous utilisez Chrome, vous pouvez faire un clic droit sur un élément, choisissez ‘Inspecter’, mettez en surbrillance le code, faites un clic droit à nouveau et choisissez ‘Copy’ puis ‘Copy XPath’

Après une analyse rapide, nous voyons que dans notre page les données sont contenues dans deux éléments - l'un est une div avec le titre ‘buyer-name’ et l'autre est un span avec la classe ‘item-price’ :

```
<div title="buyer-name">Carson Busses</div>
<span class="item-price">$29.95</span>
```

Sachant cela, nous pouvons créer la requête XPath correcte et utiliser la fonction `xpath` lxml comme ceci :

```
#This will create a list of buyers:
buyers = tree.xpath('//div[@title="buyer-name"]/text()')
#This will create a list of prices
prices = tree.xpath('//span[@class="item-price"]/text()')
```

Voyons voir ce que nous avons exactement :

```
print 'Buyers: ', buyers
print 'Prices: ', prices
```

```
Buyers:  ['Carson Busses', 'Earl E. Byrd', 'Patty Cakes',
'Derri Anne Connecticut', 'Moe Dess', 'Leda Doggslife', 'Dan Druff',
'Al Fresco', 'Ido Hoe', 'Howie Kisses', 'Len Lease', 'Phil Meup',
'Ira Pent', 'Ben D. Rules', 'Ave Sectomy', 'Gary Shattire',
'Bobbi Soks', 'Sheila Takya', 'Rose Tattoo', 'Moe Tell']

Prices:  ['$29.95', '$8.37', '$15.26', '$19.25', '$19.25',
'$13.99', '$31.57', '$8.49', '$14.47', '$15.86', '$11.11',
'$15.98', '$16.27', '$7.50', '$50.85', '$14.26', '$5.68',
'$15.00', '$114.07', '$10.09']
```

Félicitations ! Nous avons scrapé avec succès toutes les données que nous voulions à partir d'une page Web en utilisant lxml et Requests. Nous les avons stocké en mémoire, dans deux listes. Maintenant, nous pouvons faire toutes sortes de trucs cools avec elles : nous pouvons l'analyser en utilisant Python ou nous pouvons l'enregistrer dans un fichier et les partager avec le monde.

Quelques idées plus cools sur lesquelles réfléchir sont de modifier ce script pour parcourir le reste des pages de ce jeu de données exemple, ou la réécriture de cette application pour utiliser les threads pour améliorer la vitesse.

Applications en ligne de commande

Les applications en ligne de commande, également appelées Applications console, sont des programmes informatiques conçus pour être utilisés à partir d'une interface texte, telle qu'un shell. Les applications en ligne de commande acceptent généralement différentes entrées comme arguments, souvent appelés paramètres ou sous-commandes, aussi bien que options, souvent désignées comme 'flags' ou 'switches'.

Quelques applications en ligne de commande populaires incluent :

- Grep - un utilitaire de recherche de données en texte-plein
- curl - un outil pour le transfert de données avec une syntaxe URL
- httpie - un client HTTP en ligne de commande, un remplaçant de cURL convivial
- git - Un système de gestion de version distribué
- mercurial - Un système de gestion de version distribué principalement écrit en Python

Clint

clint est un module Python qui est contient plein d'outils très utiles pour le développement d'applications en ligne de commande. Il prend en charge des fonctionnalités telles que ; la couleur et l'indentation de la console/invite, l'impression de colonnes simple et puissante, les barres de progression basées sur des itérateurs et la manipulation d'argument implicite.

Click

click est un package Python à venir pour créer des interfaces de ligne de commande d'une manière composable avec le moins de code possible. Cette "Command-Line Interface Creation Kit" est hautement configurable, mais est livrée avec une bonne configuration par défaut directement.

docopt

docopt est un paquet très léger et très Pythonique qui permet de créer des interfaces de ligne de commande simple et intuitive, en parsant les instructions d'usage dans le style de POSIX.

Plac

Plac est un simple wrapper sur la bibliothèque standard Python argparse, qui cache la majeure partie de sa complexité en utilisant une interface déclarative : le parser d'argument est inféré plutôt qu'écrit de manière impérative. Ce module cible en particulier les utilisateurs non-avertis, les programmeurs, les administrateurs système, les scientifiques et en général les gens qui écrivent des scripts jetables pour eux-mêmes, qui choisissent de créer une interface de ligne de commande, car c'est rapide et simple.

Cliff

Cliff est un framework pour construire des programmes en ligne de commande. Il utilise les points d'entrée de setuptools pour fournir des sous-commandes, des formatteurs de sortie, et d'autres extensions. Le framework est destiné à être utilisé pour créer des commandes à plusieurs niveaux tels que subversion et git, où le programme principal gère le parsing d'argument de base, puis invoque une sous-commande pour faire le travail.

Applications avec interface graphique utilisateur (GUI)

Liste alphabétique des applications avec interface graphique utilisateur (GUI)

Camelot

Camelot fournit des composants pour construire des applications par dessus Python, SQLAlchemy et Qt. Il est inspiré par l'interface d'administration de Django.

La ressource principale pour des informations est le site web : http://www.python-camelot.com and the mailing list https://groups.google.com/forum/#!forum/project-camelot

Cocoa

Note : Le framework Cocoa est uniquement disponible sur OS X. Ne l'utilisez pas si vous écrivez une application multi-plateformes.

GTk

PyGTK fournit des bindings Python pour la boîte à outils GTK+. Comme la bibliothèque GTK+ elle-même, il est actuellement sous licence GNU LGPL. Il est intéressant de noter que PyGTK ne supporte actuellement que l'API Gtk-2.X (PAS Gtk-3.0). Il est actuellement recommandé de ne pas utiliser PyGTK pour les nouveaux projets et que les applications existantes soient portées de PyGTK à PyGObject.

PyGObject connu aussi comme (PyGi)

PyGObject fournit des bindings Python, qui donnent un accès complet à la plate-forme logicielle GNOME. Il est entièrement compatible avec GTK+ 3. Voici un tutoriel pour commencer intitulé Python GTK+ 3 Tutorial.

Référence de l'API

Kivy

Kivy est une bibliothèque Python pour le développement d'applications riches en média supportant le multi-touch. L'objectif est de permettre la conception d'interaction rapide et facile et le prototypage rapide, tout en rendant votre code réutilisable et déployable.

Kivy est écrit en Python, basé sur OpenGL et supporte les différents dispositifs d'entrée tels que : souris, la souris double, TUIO, WiiMote, WM_TOUCH, HIDtouch, les produits d'Apple et ainsi de suite.

Kivy est activement développé par une communauté et est libre à utiliser. Il fonctionne sur toutes les principales plateformes (Linux, OSX, Windows, Android).

La ressource principale pour des informations est le site web : http://kivy.org

PyObjC

Note : Uniquement disponible sur OS X. Ne l'utilisez pas si vous écrivez une application multi-plateformes.

PySide

PySide est un binding Python de la boîte à outils graphique multi-plateformes Qt.

```
pip install pyside
```

https://wiki.qt.io/Category:LanguageBindings::PySide::Downloads

PyQt

Note : Si votre logiciel ne se conforme pas complètement à la GPL, vous aurez besoin d'une licence commerciale !

PyQt fournit des bindings Python pour le framework Qt (voir ci-dessous).

http://www.riverbankcomputing.co.uk/software/pyqt/download

PyjamasDesktop (pyjs Desktop)

PyjamasDesktop est un port de Pyjamas. PyjamasDesktop est un jeu de widgets pour applications prévu pour le bureautique et un framework multi-plateformes. (Après la mise à disposition de la v0.6, PyjamasDesktop est un morceau de Pyjamas (Pyjs)). En bref, il permet au même code d'une application web Python d'être exécuté comme une application de bureau autonome.

Wiki Python pour PyjamasDesktop.

Le site principal ; pyjs Desktop.

Qt

Qt est un framework multi-plateformes qui est largement utilisé pour développer de logiciels avec une interface graphique, mais peut également être utilisé pour des applications sans interface graphique.

Tk

Tkinter est une mince couche orientée par dessus Tcl/Tk. **Il a l'avantage d'être inclus dans la bibliothèque standard Python, ce qui le rend la boîte à outils la plus pratique et compatible pour programmer.**

Tk et Tkinter sont tous les deux disponibles sur la plupart des plateformes Unix, ainsi que sur les systèmes Windows et Macintosh. Depuis la version 8.0, Tk offre une apparence et une impression native sur toutes les plateformes.

Il y a un bon tutoriel Tk multi-langages avec des exemples Python sur TkDocs. Plus d'informations sont disponibles sur Wiki Python.

wxPython

wxPython est une boîte à outils d'interface graphique pour le langage de programmation Python. Il permet aux programmeurs Python de créer des programmes avec une interface utilisateur graphique robuste, très fonctionnel, simple et facile. Il est implémenté comme un module d'extension Python (code natif) qui enveloppe la populaire bibliothèque graphique multi-plateformes wxWidgets, qui est écrite en C ++.

Installez wxPython (Stable) *allez sur http ://www.wxpython.org/download.php#stable et téléchargez le paquet approprié pour votre système d'exploitation.*

Bases de données

DB-API

La Database API (DB-API) de Python définit une interface standard pour les modules d'accès à une base de données. Elle est documentée dans la **PEP 249**. Presque tous les modules de base de données Python comme *sqlite3*, *psycopg* et *mysql-python* se conforment à cette interface.

Tutoriels qui expliquent comment travailler avec les modules qui sont conformes à cette interface peuvent être trouvés ici et ici.

SQLAlchemy

SQLAlchemy est une boîte à outils pour base de données largement utilisée. Contrairement à de nombreuses bibliothèques de base de données, il ne fournit pas seulement une couche ORM, mais aussi une API généralisée pour l'écriture du code agnostique aux bases de données, sans SQL.

```
$ pip install sqlalchemy
```

Records

Records est une bibliothèque de SQL minimaliste, conçue pour envoyer des requêtes SQL brutes à diverses bases de données. Les données peuvent être utilisées par programmation, ou exportées vers un certain nombre de formats de données utiles.

```
$ pip install records
```

Inclut aussi un outil de ligne de commande for exporter les données SQL.

Django ORM

Django ORM est l'interface utilisée par Django pour fournir l'accès aux bases de données.

C'est basé sur l'idée des modèles, une abstraction qui rend plus facile de manipuler des données dans Python.

Les basiques :

— Chaque modèle est une classe Python qui sous-classe django.db.models.Model.
— Chaque attribut du modèle représente un champ de base de données.
— Django vous donne une API d'accès aux bases de données générés automatiquement ; voir Faire des requêtes.

peewee

peewee est un autre ORM dont le but est d'être léger avec le support de Python 2.6+ et 3.2+, qui supporte SQLite, MySQL et Postgres par défaut. La couche du modèle est similaire à celle de Django ORM et il a des méthodes de type SQL pour requêter les données. Alors que SQLite, MySQL et Postgres sont pris en charge directement, il y a une collection d'add-ons disponible.

PonyORM

PonyORM est un ORM qui prend une approche différente pour interroger la base de données. Au lieu d'écrire un langage similaire à SQL ou des expressions booléennes, la syntaxe pour les générateurs Python est utilisée. Il y a aussi un éditeur de schéma graphique qui peut générer des entités PonyORM pour vous. Il supporte Python 2.6+ et Python 3.3+ et peut se connecter à SQLite, MySQL, PostgreSQL et Oracle

SQLObject

SQLObject est encore un autre ORM. Il prend en charge une grande variété de bases de données : des systèmes de base de données communs comme MySQL, Postgres et SQLite et des systèmes plus exotiques comme SAP DB, SyBase et MSSQL. Il supporte seulement Python 2 à partir de Python 2.6 et supérieur.

Réseau

Twisted

Twisted est un moteur réseau piloté par événement. Il peut être utilisé pour construire des applications autour de nombreux protocoles réseau différents, incluant les serveurs et les clients http, les applications utilisant les protocoles SMTP, POP3, IMAP ou SSH, la messagerie instantanée et plus encore.

PyZMQ

PyZMQ est le binding Python pour ZeroMQ, qui est une bibliothèque de messagerie asynchrone haute performance. Un grand avantage de ZeroMQ est qu'il peut être utilisé pour la gestion de queue de messages sans broker de messages. Les patterns de base pour cela sont :

— request-reply : connecte un jeu de clients à un jeu de services ; C'est un pattern d'appel de procédure à distance et de distribution de tâches
— publish-subscribe : connecte un jeu de publicateurs à un jeu d'abonnés. C'est un pattern de distribution de données.
— push-pull (ou pipeline) : connecte les nœuds suivant un pattern fan-in / fan-out qui peut avoir plusieurs étapes, et des boucles. C'est un pattern de répartition et de collecte de tâches en parallèle.

Pour un démarrage rapie, lire le guide ZeroMQ.

gevent

gevent est une bibliothèque de réseau de Python basée sur les coroutines qui utilise greenlets pour fournir une API synchrone de haut niveau sur le dessus de la boucle d'événement de libev.

Administration système

Fabric

Fabric est une bibliothèque pour simplifier les tâches d'administration du système. Alors que Chef et Puppet ont tendance à se concentrer sur la gestion des serveurs et des bibliothèques du système, Fabric est plus concentré sur des tâches au niveau des applications telles que le déploiement.

Installer Fabric :

```
$ pip install fabric
```

Le code suivant va créer deux tâches que nous pouvons utiliser : `memory_usage` et `deploy`. La première va retourner l'utilisation de la mémoire sur chaque machine. La dernière va se connecter en ssh sur chaque serveur, faire un cd dans notre répertoire de projet, activer l'environnement virtuel, faire un pull de la base de code la plus récente et redémarrer le serveur d'applications.

```
from fabric.api import cd, env, prefix, run, task

env.hosts = ['my_server1', 'my_server2']

@task
def memory_usage():
    run('free -m')

@task
def deploy():
    with cd('/var/www/project-env/project'):
        with prefix('. ../bin/activate'):
            run('git pull')
            run('touch app.wsgi')
```

Avec le code précédent enregistré dans un fichier nommé `fabfile.py`, nous pouvons vérifier l'utilisation de la mémoire avec :

```
$ fab memory_usage
[my_server1] Executing task 'memory'
[my_server1] run: free -m
[my_server1] out:              total     used     free   shared  buffers   cached
[my_server1] out: Mem:          6964     1897     5067        0      166      222
[my_server1] out: -/+ buffers/cache:     1509     5455
```

```
[my_server1] out: Swap:            0        0        0

[my_server2] Executing task 'memory'
[my_server2] run: free -m
[my_server2] out:              total     used     free   shared  buffers   cached
[my_server2] out: Mem:          1666      902      764        0      180      572
[my_server2] out: -/+ buffers/cache:      148     1517
[my_server2] out: Swap:          895        1      894
```

et nous pouvons déployer avec :

```
$ fab deploy
```

Des fonctionnalités additionnelles incluent l'exécution en parallèle, l'interaction avec les programmes à distance, et le regroupement d'hôtes.

Documentation Fabric

Salt

Salt est un outil de gestion d'infrastructure open source. Il supporte l'exécution de commandes à distance à partir d'un point central (hôte maître) à plusieurs hôtes (minions). Il supporte également des états système qui peuvent être utilisés pour configurer plusieurs serveurs à l'aide de fichiers modèle simples.

Salt supporte les versions de Python 2.6 et 2.7 et peut être installé via pip :

```
$ pip install salt
```

Après avoir configuré un serveur maître et un n'importe quel nombre d'hôtes de minion, nous pouvons exécuter des commandes shell arbitraires ou utiliser des modules pré-intégrés de commandes complexes sur nos minions.

La commande suivante liste tous les hôtes de minion disponibles, en utilisant le module ping.

```
$ salt '*' test.ping
```

Le filtrage de l'hôte est accompli en faisant correspondre l'identifiant du minion, ou en utilisant le système de grains. Le système grains utilise des informations d'hôte statique comme la version du système d'exploitation ou de l'architecture du processeur pour fournir une taxonomie d'hôte pour les modules Salt.

La commande suivante liste tous les minions disponibles exécutant CentOS, en utilisant le système grains.

```
$ salt -G 'os:CentOS' test.ping
```

Salt propose également un système d'état. Les états peuvent être utilisés pour configurer les hôtes de minion.

Par exemple, quand un hôte de minion est commandé de lire le fichier d'état suivant, il va installer et démarrer le serveur Apache :

```
apache:
  pkg:
    - installed
  service:
    - running
    - enable: True
    - require:
      - pkg: apache
```

Les fichiers d'état peuvent être écrit en utilisant le YAML, le système de template Jinja2 ou du Python pur.

Documentation Salt

Psutil

Psutil est une interface pour différentes informations système (par exemple, le CPU, la mémoire, les disques, le réseau, les utilisateurs et les processus).

Voici un exemple pour être au courant de la surcharge d'un serveur. Si l'un des tests (net, CPU) échoue, il va envoyer un e-mail.

```
# Functions to get system values:
from psutil import cpu_percent, net_io_counters
# Functions to take a break:
from time import sleep
# Package for email services:
import smtplib
import string
MAX_NET_USAGE = 400000
MAX_ATTACKS = 4
attack = 0
counter = 0
while attack <= MAX_ATTACKS:
    sleep(4)
    counter = counter + 1
    # Check the cpu usage
    if cpu_percent(interval = 1) > 70:
        attack = attack + 1
    # Check the net usage
    neti1 = net_io_counters()[1]
    neto1 = net_io_counters()[0]
    sleep(1)
    neti2 = net_io_counters()[1]
    neto2 = net_io_counters()[0]
    # Calculate the bytes per second
    net = ((neti2+neto2) - (neti1+neto1))/2
    if net > MAX_NET_USAGE:
        attack = attack + 1
    if counter > 25:
        attack = 0
        counter = 0
# Write a very important email if attack is higher than 4
TO = "you@your_email.com"
FROM = "webmaster@your_domain.com"
SUBJECT = "Your domain is out of system resources!"
text = "Go and fix your server!"
BODY = string.join(("From: %s" %FROM,"To: %s" %TO,"Subject: %s" %SUBJECT, "",text), "\r\n")
server = smtplib.SMTP('127.0.0.1')
server.sendmail(FROM, [TO], BODY)
server.quit()
```

Une application complète dans le terminal similaire à une version de 'top' très étendue, basée sur psutil et avec une capacité de monitoring client-serveur est glance.

Ansible

Ansible est un outil d'automatisation système open source. Le plus grand avantage sur Puppet ou Chef est qu'il ne nécessite pas un agent sur la machine cliente. Les playbooks sont la configuration, le déploiement, et le langage d'orchestration d'Ansible. Ils sont rédigés en YAML avec Jinja2 pour le templating.

Ansible supporte les versions 2.6 et 2.7 de Python et peut être installé via pip :

```
$ pip install ansible
```

Ansible nécessite un fichier inventory qui décrit les hôtes auxquels il a accès. Voici ci-dessous un exemple d'un hôte et d'un playbook qui va faire un ping sur tous les hôtes dans le fichier inventory.

Voici un exemple de fichier inventory : `hosts.yml`

```
[server_name]
127.0.0.1
```

Voici un exemple de playbook : `ping.yml`

```
---
- hosts: all

  tasks:
    - name: ping
      action: ping
```

Pour exécuter le playbook :

```
$ ansible-playbook ping.yml -i hosts.yml --ask-pass
```

Le playbook Ansible fera un ping sur tous les serveurs dans le fichier `hosts.yml`. Vous pouvez également sélectionner des groupes de serveurs utilisant Ansible. Pour plus d'informations sur Ansible, lisez la *documentation Ansible <http ://docs.ansible.com/> _*.

An Ansible tutorial est aussi une bonne introduction, bien détaillée, pour commencer avec Ansible.

Chef

Chef est un framework d'automatisation système et infrastructure cloud qui rend facile de déployer des serveurs et des applications à tout emplacement physique, virtuelle ou dans le cloud. Dans le cas où c'est votre choix pour la gestion de la configuration, vous utiliserez principalement Ruby pour écrire votre code d'infrastructure.

Les clients Chef s'exécutent sur chaque serveur qui fait partie de votre infrastructure et ceux-ci vérifient régulièrement avec votre serveur Chef pour assurer que votre système est toujours aligné et représente l'état désiré. Étant donné que chaque serveur possède son propre client Chef distinct, chaque serveur se configure lui-même et cette approche distribuée fait de Chef une plate-forme d'automatisation évolutive.

Chef fonctionne en utilisant des recettes personnalisées (éléments de configuration), implémentés dans des cookbooks. Les cookbooks, qui sont essentiellement des paquets pour des choix d'infrastructure, sont généralement stockés dans votre serveur Chef. Lisez la série de tutoriels de Digital Ocean sur Chef pour apprendre comment créer un Serveur Chef simple.

Pour créer un cookbook simple, la commande knife est utilisée :

```
knife cookbook create cookbook_name
```

Getting started with Chef est un bon point de départ pour les débutants en Chef. De nombreux cookbooks maintenus par la communauté qui peuvent servir comme une bonne référence ou personnalisés pour servir vos besoins de configuration de infrastructure peuvent être trouvés sur le Supermarket Chef.

— Documentation Chef

Puppet

Puppet est un logiciel de gestion de configuration et d'automatisation de Puppet Labs qui permet aux administrateurs système de définir l'état de leur infrastructure informatique, fournissant ainsi une manière élégante de gérer leur flotte de machines physiques et virtuelles.

Puppet est disponible à la fois en variante Open Source et Entreprise. Les modules sont de petites unités de code partageables écrites pour automatiser ou définir l'état d'un système. La forge Puppet est un dépôt pour les modules écrits par la communauté pour Puppet Open Source et Entreprise.

Les agents de Puppet sont installés sur des nœuds dont l'état doit être contrôlé ou changé. Un serveur désigné connu comme le Puppet Master est responsable de l'orchestration de l'agent des nœuds.

L'agent des nœuds envoie des facts de base à propos du système comme le système d'exploitation, le kernel, l'adresse ip, le nom d'hôte, etc. au Puppet Master. Le Puppet Master ensuite compile un catalogue avec les informations fournies par les agents sur comment chaque nœud doit être configuré et l'envoie à l'agent. L'agent applique le changement comme prescrit dans le catalogue et renvoie un rapport au Puppet Master.

Facter est un outil intéressant qui est livré avec Puppet qui récupère des facts de base sur le système. Ces facts peuvent être référencés comme une variable tout en écrivant vos modules Puppet.

```
$ facter kernel
Linux
```

```
$ facter operatingsystem
Ubuntu
```

L'écriture des modules dans Puppet est assez simple. Les manifestes Puppet forment ensemble les modules Puppet. Les manifestes Puppet se terminent avec une extension `.pp`. Voici un exemple de 'Hello World' en Puppet.

```
notify { 'This message is getting logged into the agent node':

    #As nothing is specified in the body the resource title
    #the notification message by default.
}
```

Voici un autre exemple avec une logique basée sur le système. Notez comment le fact du système d'exploitation est utilisé comme une variable préfixé avec le signe `$`. De même, cela vaut pour d'autres facts tels que le nom d'hôte qui peut être référencé par `$hostname`

```
notify{ 'Mac Warning':
    message => $operatingsystem ? {
        'Darwin' => 'This seems to be a Mac.',
        default  => 'I am a PC.',
    },
}
```

Il existe plusieurs types de ressources pour Puppet, mais le paradigme package-fichier-service est tout ce qu'il faut pour entreprendre la majorité de la gestion de la configuration. Le code Puppet suivant s'assure que le paquet OpenSSH-Server est installé dans un système et le service sshd est averti de redémarrer chaque fois que le fichier de configuration de sshd est modifié.

```
package { 'openssh-server':
    ensure => installed,
}

file { '/etc/ssh/sshd_config':
    source   => 'puppet:///modules/sshd/sshd_config',
    owner    => 'root',
```

```
    group    => 'root',
    mode     => '640',
    notify   =>  Service['sshd'], # sshd will restart
                                  # whenever you edit this
                                  # file
    require  => Package['openssh-server'],

}

service { 'sshd':
    ensure    => running,
    enable    => true,
    hasstatus => true,
    hasrestart=> true,
}
```

Pour plus d'informations, se référer à la Documentation Puppet Labs

Blueprint

À faire

Ecrire à propos de Blueprint

Buildout

Buildout est un outil de build logiciel open source. Buildout est créé en utilisant le langage de programmation Python. Il implémente un principe de séparation de configuration à partir des scripts qui font la configuration. Buildout est principalement utilisé pour télécharger et configurer des dépendances dans le format eggs Python du logiciel en cours de développement ou déployé. Les recettes pour des tâches de build dans tous les environnements peuvent être créés, et beaucoup sont déjà disponibles.

Buidout est écrit en Python.

Intégration continue

Pourquoi ?

Martin Fowler, qui a été le premier à écrire à propos de l'intégration continue (Continuous Integration ou CI) avec Kent Beck, décrit la CI comme ce qui suit :

> L'intégration continue est une pratique de développement logiciel, où les membres d'une équipe intègrent leur travail souvent, habituellement chaque personne intègre au moins quotidiennement - conduisant à de multiples intégrations par jour. Chaque intégration est vérifiée par un build automatique (y compris le test) pour détecter les erreurs d'intégration aussi rapidement que possible. De nombreuses équipes trouvent que cette approche conduit à une réduction significative des problèmes d'intégration et permet à une équipe de développer des logiciels cohérents plus rapidement.

Jenkins

Jenkins CI est un moteur d'intégration continue extensible. Utilisez-le.

Buildbot

Buildbot est un système Python pour automatiser le cycle de compilation/test pour valider les changements de code.

Tox

tox est un outil d'automatisation fournissant le packaging, les tests et le déploiement de logiciels Python directement depuis la console ou le serveur d'intégration continue. C'est un outil en ligne de commande de gestion d'environnement virtuel virtualenv générique et de test qui fournit les fonctionnalités suivantes :

- Vérifiant que les paquets s'installent correctement avec les différentes versions de Python et ses différents interpréteurs
- Exécutant des tests dans chacun des environnements, configurant l'outil de test de votre choix
- Agissant comme un front-end pour les serveurs d'intégration continue, réduisant les surcouches et fusionnant la CI et les tests basés sur le shell.

Travis-CI

Travis-CI est un serveur de CI distribué qui exécute des tests pour les projets open source gratuitement. Il fournit plusieurs workers pour exécuter des tests Python sur et s'intègre de façon transparente avec GitHub. Vous pouvez même faire des commentaires sur vos Pull Requests si ce changeset particulier casse le build ou non. Donc, si vous hébergez votre code sur GitHub, travis-ci est un bonne et facile manière de commencer avec l'intégration continue.

Pour commencer, ajoutez un fichier `.travis.yml` à votre dépôt avec cet exemple de contenu :

```
language: python
python:
  - "2.6"
  - "2.7"
  - "3.2"
  - "3.3"
# command to install dependencies
script: python tests/test_all_of_the_units.py
branches:
  only:
    - master
```

Cela vous permettre d'avoir votre projet testé sur toutes les versions de Python listées en exécutant le script donné, et ne fera que builder la branche master. Il y a beaucoup plus d'options que vous pouvez activer, comme les notifications, avant et après les étapes et bien plus encore. La documentation de travis-ci explique toutes ces options, et est très complète.

Afin d'activer les tests pour votre projet, allez sur le site travis-ci et connectez-vous avec votre compte GitHub. Ensuite, activez votre projet dans vos paramètres de profil et vous êtes bon. A partir de maintenant, les tests de votre projet seront exécutés sur tous les push vers GitHub.

Vitesse

CPython, l'implémentation la plus couramment utilisée de Python, est lente pour les tâches liées au CPU (processeur). PyPy est rapide.

En utilisant une version légèrement modifiée du code de test lié au CPU de David Beazley's (boucle ajoutée pour de multiples tests), vous pouvez voir la différence entre le traitement CPython et PyPy.

```
# PyPy
$ ./pypy -V
Python 2.7.1 (7773f8fc4223, Nov 18 2011, 18:47:10)
[PyPy 1.7.0 with GCC 4.4.3]
$ ./pypy measure2.py
0.0683999061584
0.0483210086823
0.0388588905334
0.0440690517426
0.0695300102234
```

```
# CPython
$ ./python -V
Python 2.7.1
$ ./python measure2.py
1.06774401665
1.45412397385
1.51485204697
1.54693889618
1.60109114647
```

Contexte

Le GIL

The GIL (Global Interpreter Lock) est comment Python permet à plusieurs threads de fonctionner en même temps. La gestion de la mémoire de Python est pas entièrement thread-safe, de sorte que le GIL est requis pour empêcher plusieurs threads d'exécuter le même code Python à la fois.

David Beazley a un bon guide sur la manière dont le GIL opère. I lcouvre aussi le new GIL dans Python 3.2. Ses résultats montrent que maximiser les performances dans une application Python nécessite une bonne compréhension de la GIL, comment il affecte votre application spécifique, combien de cœurs que vous avez, et où sont vos goulots d'étranglement dans l'application.

Extensions C

Le GIL

Une Special care doit être prise lors de l'écriture d'extensions C pour vous assurer que vous enregistrez vos threads avec l'interpréteur.

Extensions C

Cython

Cython implémente un sur-ensemble du langage Python avec lequel vous êtes en mesure d'écrire des modules C et C ++ pour Python. Cython vous permet également d'appeler des fonctions depuis des bibliothèques compilées C. Utiliser Cython vous permet de tirer avantage du typage fort des variables Python et des opérations.

Voici un exemple de typage fort avec Cython :

```
def primes(int kmax):
"""Calculation of prime numbers with additional
Cython keywords"""

    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

L'implémentation d'un algorithme pour trouver des nombres premiers a quelques mots-clés supplémentaires par rapport à la suivante, qui est implémentée en pur Python :

```
def primes(kmax):
"""Calculation of prime numbers in standard Python syntax"""

    p = range(1000)
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

Notez que dans la version Cython, vous déclarez les entiers et les tableaux d'entiers qui seront compilés en types C, tout en créant aussi une liste Python :

```
def primes(int kmax):
    """Calculation of prime numbers with additional
    Cython keywords"""

    cdef int n, k, i
    cdef int p[1000]
    result = []
```

```
def primes(kmax):
    """Calculation of prime numbers in standard Python syntax"""
```

```
    p = range(1000)
    result = []
```

Quelle est la différence ? Dans la version Cython ci-dessus, vous pouvez voir la déclaration des types de variables et le tableau entier d'une manière similaire à celle du standard C. Par exemple *cdef int n,k,i* dans la ligne 3. Cette déclaration de type supplémentaire (c'est à dire entier) permet au compilateur Cython de générer du code C plus efficace à partir de la deuxième version. Alors que le code standard de Python est sauvé dans des fichiers `*.py`, le code Cython est sauvé dans des fichiers `*.pyx`.

Quelle est la différence de vitesse ? Essayons !

```
import time
#activate pyx compiler
import pyximport
pyximport.install()
#primes implemented with Cython
import primesCy
#primes implemented with Python
import primes

print "Cython:"
t1= time.time()
print primesCy.primes(500)
t2= time.time()
print "Cython time: %s" %(t2-t1)
print ""
print "Python"
t1= time.time()
print primes.primes(500)
t2= time.time()
print "Python time: %s" %(t2-t1)
```

Ces deux lignes nécessitent une remarque :

```
import pyximport
pyximport.install()
```

Le module *pyximport* vous permet d'importer les fichiers `*.pyx` (par exemple, `primesCy.pyx`) avec la version compilée par Cython de la fonction *primes*. La commande *pyximport.install()'permet à l'interpréteur Python de démarrer directement le compilateur Cython pour générer le code C, qui est automatiquement compilé en une bibliothèque C :file :'*.so*. Cython est alors capable d'importer cette bibliothèque pour vous dans votre code Python, facilement et efficacement. Avec la fonction *time.time()*, vous êtes en mesure de comparer le temps entre ces 2 différents appels pour trouver 500 nombres premiers. Sur un ordinateur portable standard (dual core AMD E-450 1,6 GHz), les valeurs mesurées sont :

```
Cython time: 0.0054 seconds

Python time: 0.0566 seconds
```

Et voici la sortie sur une machine ARM beaglebone intégreé :

```
Cython time: 0.0196 seconds

Python time: 0.3302 seconds
```

Pyrex

Shedskin ?

Numba

À faire

Écrire à propos de Numba et du compilateur autojit pour NumPy

Concurrence

Concurrent.futures

Le module concurrent.futures est un module dans la bibliothèque standard qui fournit une "interface de haut-niveau pour exécuter des callables de manière asynchrone". Il abstrait une grande partie des détails les plus compliqués sur l'utilisation de plusieurs threads ou processus pour la concurrence, et permet à l'utilisateur de se concentrer sur l'accomplissement de la tâche à accomplir.

Le module concurrent.futures expose deux classes principales, *ThreadPoolExecutor* et *ProcessPoolExecutor*. Le ThreadPoolExecutor va créer un pool de worker threads auquel un utilisateur peut soumettre des jobs à faire. Ces jobs seront ensuite exécutés dans un autre thread quand le prochain worker thread va devenir disponible.

Le ProcessPoolExecutor fonctionne de la même manière, sauf au lieu d'utiliser plusieurs threads pour ses workers, elle utilisera de multiples processus. Cela permet de mettre de côté le GIL, cependant à cause de la façon dont les choses sont passées à des processus workers, seuls les objets picklables peuvent être exécutés et retournés.

En raison de la manière dont le GIL fonctionne, une bonne règle de base est d'utiliser une ThreadPoolExecutor lorsque la tâche en cours d'exécution implique beaucoup de blocage (à savoir faire des requêtes sur le réseau) et d'utiliser un exécuteur ProcessPoolExecutor lorsque la tâche est informatiquement coûteuse.

Il existe deux principales manières d'exécuter des choses en parallèle en utilisant les deux exécuteurs. Une façon est avec la méthode *map(func, iterables)*. Cela fonctionne presque exactement comme la fonction intégrée *map()*, sauf qu'il exécutera tout en parallèle. :

```
from concurrent.futures import ThreadPoolExecutor
import requests

def get_webpage(url):
    page = requests.get(url)
    return page

pool = ThreadPoolExecutor(max_workers=5)

my_urls = ['http://google.com/']*10  # Create a list of urls

for page in pool.map(get_webpage, my_urls):
    # Do something with the result
    print(page.text)
```

Pour encore plus de contrôle, la méthode *submit(func, *args, **kwargs)* programmera qu'un callable soit exécuté (comme *func(*args, **kwargs)*) et retourne un objet Future qui représente l'exécution du callable.

L'objet Future fournit diverses méthodes qui peuvent être utilisées pour vérifier l'état d'avancement du callable programmé. Cela inclut :

cancel() Tentative d'annulation de l'appel.

cancelled() Retourne True si l'appel a été annulé avec succès.

running() Retourne True si l'appel a été exécuté à ce moment et ne peut pas annulé.

done() Retourne True si l'appel a été annulé avec succès ou a fini de s'exécuter.

result() Retourne la valeur retournée par l'appel. Notez que cet appel sera bloquant jusqu'à le callable programmé soit retourné par défaut.

exception() Retourne l'exception levée par l'appel. Si aucune exception n'a été levée alors cela retourne *None*. Notez que cela va bloquer tout comme *result()*.

add_done_callback(fn) Attache une fonction de callback qui sera exécutée (comme *fn(future)*) quand le callable prévu sera retourné.

```
from concurrent.futures import ProcessPoolExecutor, as_completed

def is_prime(n):
    if n % 2 == 0:
        return n, False

    sqrt_n = int(n**0.5)
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return n, False
    return n, True

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

futures = []
with ProcessPoolExecutor(max_workers=4) as pool:
    # Schedule the ProcessPoolExecutor to check if a number is prime
    # and add the returned Future to our list of futures
    for p in PRIMES:
        fut = pool.submit(is_prime, p)
        futures.append(fut)

# As the jobs are completed, print out the results
for number, result in as_completed(futures):
    if result:
        print("{} is prime".format(number))
    else:
        print("{} is not prime".format(number))
```

Le module concurrent.futures contient deux helpers pour travailler avec Futures. La fonction *as_completed(futures)* retourne un itérateur sur la liste des futures, en faisant un yield des futures jusqu'à ce qu'elles soient complètes.

La fonction *wait(futures)* va tout simplement bloquer jusqu'à ce que toutes les futures dans la liste des futures soient terminées.

Pour plus d'informations, sur l'utilisation du module concurrent.futures, consulter la documentation officielle.

Threading

La bibliothèque standard est livré avec un module threading qui permet à un utilisateur de travailler avec plusieurs threads manuellement.

Exécuter une fonction dans un autre thread est aussi simple que de passer un callable et ses arguments constructeur du *Thread' et d'appeler ensuite 'start()* :

```
from threading import Thread
import requests

def get_webpage(url):
    page = requests.get(url)
    return page

some_thread = Thread(get_webpage, 'http://google.com/')
some_thread.start()
```

Pour attendre jusqu'à ce que le thread soit terminé, appelez *join()* :

```
some_thread.join()
```

Après l'appel du *join()*, c'est toujours une bonne idée de vérifier si le thread est toujours en vie (parce que l'appel join a expiré) :

```
if some_thread.is_alive():
    print("join() must have timed out.")
else:
    print("Our thread has terminated.")
```

Parce que plusieurs threads ont accès à la même section de la mémoire, parfois il peut y avoir des situations où deux ou plusieurs threads tentent d'écrire sur la même ressource en même temps ou lorsque la sortie est dépendante de la séquence ou du timing de certains événements. Ceci est appelé une data race ou "race condition". Lorsque cela se produit, la sortie sera dénaturée ou vous pouvez rencontrer des problèmes qui sont difficiles à déboguer. Un bon exemple est ce stackoverflow post.

La façon dont cela peut être évité est d'utiliser un **'Lock'_** que chaque thread aura besoin d'acquérir avant d'écrire dans une ressource partagée. Les locks peuvent être acquis et libérés soit par le protocole de contextmanager (déclaration *with*), ou en utilisant *acquire()* et *release()* directement. Voici un exemple (plutôt artificiel) :

```
from threading import Lock, Thread

file_lock = Lock()

def log(msg):
    with file_lock:
        open('website_changes.log', 'w') as f:
            f.write(changes)

def monitor_website(some_website):
    """
    Monitor a website and then if there are any changes,
    log them to disk.
    """
    while True:
        changes = check_for_changes(some_website)
        if changes:
            log(changes)
```

```
websites = ['http://google.com/', ... ]
for website in websites:
    t = Thread(monitor_website, website)
    t.start()
```

Ici, nous avons un tas de threads vérifiant des changements sur une liste de sites et chaque fois qu'il y a des changements, ils tentent d'écrire ces modifications dans un fichier en appelant *log(changes)*. Lorsque *log()* est appelé, il attendra d'acquérir le lock avec *avec file_lock :*. Cela garantit qu'à tout moment, seulement un seul thread est en train d'écrire dans le fichier.

Processus de spawning

Multiprocessing

Applications scientifiques

Contexte

Python est fréquemment utilisé pour des applications scientifiques haute-performances. Il est largement utilisé dans les projets universitaires et scientifiques car il est facile à écrire et donne de bons résultats.

En raison de sa nature haute performance, le calcul scientifique en Python utilise souvent des bibliothèques externes, généralement écrites dans des langages plus rapides (comme C ou FORTRAN pour les opérations matricielles). Les principales bibliothèques utilisées sont NumPy, SciPy et Matplotlib. Entrer dans le détail de ces bibliothèques va au-delà du périmètre couvert par ce guide Python. Cependant, une introduction complète à l'écosystème Python scientifique peut être trouvée dans les notes de lecture pour le Python scientifique (en)

Outils

IPython

IPython est une version améliorée de l'interpréteur Python, qui fournit des fonctionnalités d'un grand intérêt pour les scientifiques. Le *mode inline* permet l'affichage de graphiques et de diagrammes dans le terminal (pour la version basée sur Qt). De plus, le mode *notebook* supporte la programmation lettrée et la science reproductible en générant un notebook Python basé sur le Web. Ce notebook vous permet de stocker des bouts de code Python à côté des résultats et des commentaires supplémentaires (HTML, LaTeX, Markdown). Le notebook peut alors être partagé et exporté dans divers formats de fichiers.

Bibliothèques

NumPy

NumPy est une bibliothèque de bas niveau écrite en C (et FORTRAN) pour les fonctions mathématiques de haut niveau. NumPy dépasse habilement le problème d'exécuter des algorithmes plus lents sur Python en utilisant des tableaux multidimensionnels et des fonctions qui opèrent sur des tableaux. Tout algorithme peut alors être exprimé comme une fonction sur des tableaux, permettant aux algorithmes de exécuter rapidement.

NumPy fait partie du projet SciPy, et est mis à disposition comme bibliothèque séparée afin que les gens qui ont seulement besoin des exigences de base puissent l'utiliser sans installer le reste de SciPy.

NumPy est compatible avec les versions de Python 2.4 à 2.7.2 et 3.1+.

Numba

Numba est un compilateur Python conscient de NumPy (compilateur spécialisé just-in-time (JIT)) qui compile le code Python (et NumPy) annoté pour LLVM (Low Level Virtual Machine) via des décorateurs spéciaux. En bref, Numba utilise un système qui compile le code Python avec LLVM en code qui peut être exécuté nativement à l'exécution.

SciPy

SciPy est une bibliothèque qui utilise NumPy pour plus de fonctions mathématiques. SciPy utilise des tableaux numpy comme structure de base pour les données, et est livré avec des modules pour diverses tâches couramment utilisées dans la programmation scientifique, incluant l'algèbre linéaire, le calcul intégral (calcul différentiel), la résolution d'équation différentielle ordinaire et le traitement du signal.

Matplotlib

Matplotlib est une bibliothèque de création de diagrammes flexible pour créer des diagrammes 2D et 3D interactifs qui peuvent également être enregistrés comme des figures d'une qualité suffisante pour illustrer des manuscrits. L'API reflète de plusieurs façons celle de MATLAB, facilitant la transition des utilisateurs MATLAB à Python. De nombreux exemples, avec le code source pour les recréer, sont disponibles dans la galerie de matplotlib.

Pandas

Pandas est une bibliothèque de manipulation de données basée sur Numpy qui fournit de nombreuses fonctions utiles pour accéder, indexer, fusionner et le regrouper des données facilement. La structure de données principale (DataFrame) est proche de ce qui peut être trouvé dans le paquet statistique R ; autrement dit, des tableaux de données hétérogènes avec l'indexation par nom, les opérations sur les séries temporelles et l'auto-alignement des données.

Rpy2

Rpy2 est le binding Python pour le logiciel statistique R permettant l'exécution de fonctions de R depuis Python et transmettant les données dans les deux sens entre les deux environnements. Rpy2 est l'implémentation orientée objet des bindings Rpy.

PsychoPy

PsychoPy est une bibliothèque pour la psychologie cognitive et lex expérimentations en neuroscience. La bibliothèque gère la présentation des stimuli, le scripting de conception expérimentale et la collecte de données.

Ressources

L'installation de paquets Python scientifiques peut être compliquée, comme beaucoup de ces paquets sont mis en œuvre comme des extensions Python en C qui doivent être compilées. Cette section liste les différentes distributions Python dites scientifiques qui fournissent des collections précompilées et faciles à installer de paquets Python scientifiques.

Binaires Windows non-officiels pour les paquets d'extension Python

Beaucoup de gens qui font le calcul scientifique sont sous Windows, et encore beaucoup de paquets de calcul scientifique sont notoirement difficiles à builder et à installer sur cette plateforme. Christoph Gohlke cependant, a compilé une liste de binaires Windows pour de nombreux paquets Python utiles. La liste des paquets a grossi en devenant une ressource Python principalement scientifique à une liste plus générale. Si vous êtes sur Windows, vous pouvez y jeter un œil.

Anaconda

Continuum Analytics met à disposition la distribution Python Anaconda Python qui inclue tous les paquets Python scientifiques les plus courants, ainsi que de nombreux paquets liés à l'analyse de données et au big data. Anaconda lui-même est gratuit, et Continuum vend un certain nombre d'add-ons propriétaires. Des licences gratuites pour ces add-ons sont disponibles pour les universitaires et les chercheurs.

Canopy

Canopy est une autre distribution de Python scientifique, produit par Enthought. Une variante limitée 'Canopy Express' est disponible gratuitement, mais Enthought facture pour la distribution complète. Des licences gratuites sont disponibles pour les universitaires.

Manipulation d'images

La plupart des techniques de traitement et de manipulation d'images peuvent être effectuées efficacement en utilisant deux bibliothèques : Python Imaging Library (PIL) et Open Source Computer Vision (OpenCV).

Une brève description des deux est donnée ci-dessous.

Bibliothèque Python Imaging Library (PIL)

La Python Imaging Library, ou PIL, est l'une des bibliothèques de base pour la manipulation d'images en Python. Malheureusement, son développement a stagné, avec sa dernière version en 2009.

Heureusement pour vous, il y a un fork développé activement de PIL appelé Pillow - il est plus facile à installer, fonctionne sur tous les systèmes d'exploitation, et supporte Python 3.

Installation

Avant d'installer Pillow, vous devrez installer les prérequis pour Pillow. Vous trouverez les instructions de votre plateforme dans les instructions d'installation de Pillow.

Après cela, c'est direct :

```
$ pip install Pillow
```

Exemple

```
from PIL import Image, ImageFilter
#Read image
im = Image.open( 'image.jpg' )
#Display image
im.show()

#Applying a filter to the image
im_sharp = im.filter( ImageFilter.SHARPEN )
#Saving the filtered image to a new file
im_sharp.save( 'image_sharpened.jpg', 'JPEG' )

#Splitting the image into its respective bands, i.e. Red, Green,
#and Blue for RGB
r,g,b = im_sharp.split()

#Viewing EXIF data embedded in image
exif_data = im._getexif()
exif_data
```

Il y a plus d'exemples pour la bibliothèque Pillow dans le tutoriel Pillow.

OpenSource Computer Vision (OpenCV)

OpenSource Computer Vision, plus connu comme OpenCV, est un logiciel plus avancé de manipulation et de traitement d'images que PIL. Il a été implémenté en plusieurs langages et est largement utilisé.

Installation

En Python, le traitement d'image en utilisant OpenCV est implémenté en utilisant les modules `cv2` et `NumPy`. Les instructions d'installation pour OpenCV devrait vous guider dans la configuration du projet pour vous-même.

NumPy peut être téléchargé à partir du Python Package Index (PyPI) :

```
$ pip install numpy
```

Exemple

```
from cv2 import *
import numpy as np
#Read Image
img = cv2.imread('testimg.jpg')
#Display Image
cv2.imshow('image',img)
cv2.waitKey(0)
cv2.destroyAllWindows()

#Applying Grayscale filter to image
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

#Saving filtered image to new file
cv2.imwrite('graytest.jpg',gray)
```

Il y a plus d'exemples d'OpenCV implémentés en Python dans cette collection de tutoriels.

Sérialisation de données

Qu'est ce que la sérialisation de données ?

La sérialisation de données est le concept de conversion de données structurées dans un format qui lui permet d'être partagé ou stocké de manière à ce que sa structure d'origine puisse être récupérée. Dans certains cas, l'intention secondaire de sérialisation de données est de minimiser la taille des données sérialisé, ce qui minimise alors les exigences d'espace disque ou de bande passante.

Pickle

Le module de sérialisation de données natif pour Python s'appelle Pickle.

Voici un exemple :

```
import pickle

#Here's an example dict
grades = { 'Alice': 89, 'Bob': 72, 'Charles': 87 }

#Use dumps to convert the object to a serialized string
serial_grades = pickle.dumps( grades )

#Use loads to de-serialize an object
received_grades = pickle.loads( serial_grades )
```

Protobuf

Si vous êtes à la recherche d'un module de sérialisation qui supporte plusieurs langages, la bibliothèque de Google Protobuf est une option.

Parsage XML

untangle

untangle est une bibliothèque simple qui prend un document XML et retourne un objet Python qui reflète les nœuds et les attributs dans sa structure.

Par exemple, un fichier XML comme ceci :

```
<?xml version="1.0"?>
<root>
    <child name="child1">
</root>
```

peut être chargé comme ceci :

```
import untangle
obj = untangle.parse('path/to/file.xml')
```

et ensuite vous pouvez obtenir le nom des éléments fils comme ceci :

```
obj.root.child['name']
```

untangle supporte aussi le chargement de XML depuis une chaîne de caractères ou une URL.

xmltodict

xmltodict est une autre bibliothèque simple qui vise à faire que le XML donne l'impression de travailler avec JSON

Un fichier XML comme ceci :

```
<mydocument has="an attribute">
  <and>
    <many>elements</many>
    <many>more elements</many>
  </and>
  <plus a="complex">
    element as well
  </plus>
</mydocument>
```

peut être chargé dans un dictionnaire Python comme ceci :

```
import xmltodict

with open('path/to/file.xml') as fd:
    doc = xmltodict.parse(fd.read())
```

et ensuite, vous pouvez accéder aux éléments, attributs et valeurs comme ceci :

```
doc['mydocument']['@has'] # == u'an attribute'
doc['mydocument']['and']['many'] # == [u'elements', u'more elements']
doc['mydocument']['plus']['@a'] # == u'complex'
doc['mydocument']['plus']['#text'] # == u'element as well'
```

xmltodict vous permet aussi des aller-retours au XML avec la fonction unparse, dispose d'un mode streaming adapté à la manipulation des fichiers qui ne rentrent pas en mémoire et supporte les espaces de noms.

JSON

La bibliothèque json peut parser un JSON depuis des chaînes de caractères ou des fichiers. La bibliothèque parse le JSON en un dictionnaire ou une liste Python. Elle peut également convertir les dictionnaires Python ou des listes en chaînes JSON.

Parsage de JSON

Prenez la chaîne de caractères suivante contenant des données JSON :

```
json_string = '{"first_name": "Guido", "last_name":"Rossum"}'
```

Elle peut être parsée comme ceci :

```
import json
parsed_json = json.loads(json_string)
```

et peut maintenant être utilisée comme un dictionnaire normal :

```
print(parsed_json['first_name'])
"Guido"
```

Vous pouvez aussi convertir ce qui suit en JSON :

```
d = {
    'first_name': 'Guido',
    'second_name': 'Rossum',
    'titles': ['BDFL', 'Developer'],
}

print(json.dumps(d))
'{"first_name": "Guido", "last_name": "Rossum", "titles": ["BDFL", "Developer"]}'
```

simplejson

La bibliothèque JSON a été ajoutée à Python dans la version 2.6. Si vous utilisez une version antérieure de Python, la bibliothèque simplejson est disponible via PyPI.

simplejson imite la bibliothèque standard json. Il est disponible pour que les développeurs qui utilisent d'anciennes versions de Python puissent utiliser les dernières fonctionnalités disponibles dans la bibliothèque json.

Vous pouvez commencer à utiliser simplejson lorsque la bibliothèque json n'est pas disponible par l'importation de simplejson sous un autre nom :

```
import simplejson as json
```

Après l'importation de simplejson comme json, les exemples ci-dessus fonctionneront tous comme si vous utilisiez la bibliothèque json standard.

Cryptography

Cryptography

Cryptography est une bibliothèque développée activement qui fournit des recettes et des primitives cryptographiques. Elle supporte Python 2,6-2,7, Python 3.3+ et PyPy.

Cryptography est divisé en deux couches : recettes et matières dangereuses ("hazardous material" ou hazmat). La couche de recettes fournit une API simple pour un chiffrement symétrique correct et la couche hazmat fournit des primitives cryptographiques de bas niveau.

Installation

```
$ pip install cryptography
```

Exemple

Code exemple utilisant la recette de chiffrement symétrique de haut niveau :

```
from cryptography.fernet import Fernet
key = Fernet.generate_key()
cipher_suite = Fernet(key)
cipher_text = cipher_suite.encrypt(b"A really secret message. Not for prying eyes.")
plain_text = cipher_suite.decrypt(cipher_text)
```

PyCrypto

PyCrypto est une autre bibliothèque, qui fournit des fonctions de hash sécurisées et différents algorithmes de chiffrement. Elle supporte Python version 2.1 à 3.3.

Installation

```
$ pip install pycrypto
```

Exemple

```
from Crypto.Cipher import AES
# Encryption
encryption_suite = AES.new('This is a key123', AES.MODE_CBC, 'This is an IV456')
cipher_text = encryption_suite.encrypt("A really secret message. Not for prying eyes.")

# Decryption
decryption_suite = AES.new('This is a key123', AES.MODE_CBC, 'This is an IV456')
plain_text = decryption_suite.decrypt(cipher_text)
```

Interfaçage avec les bibliothèques C/C++

C Foreign Function Interface

CFFI fournit un mécanisme simple à utiliser pour l'interfaçage avec C à la fois depuis CPython et PyPy. Il prend en charge deux modes : un mode de compatibilité ABI inline (exemple ci-dessous), ce qui vous permet de charger et exécuter dynamiquement des fonctions depuis des modules exécutables (exposant essentiellement les mêmes fonctionnalités que LoadLibrary ou dlopen), et un mode API, qui vous permet de construire des modules d'extension C.

Interaction avec ABI (Application Binary Interface)

```
1 from cffi import FFI
2 ffi = FFI()
3 ffi.cdef("size_t strlen(const char*);")
4 clib = ffi.dlopen(None)
5 length = clib.strlen("String to be evaluated.")
6 # prints: 23
7 print("{}".format(length))
```

ctypes

ctypes est la bibliothèque de fait pour l'interfaçage avec C/C ++ depuis CPython, et il fournit non seulement un accès complet à l'interface C native de la plupart des principaux systèmes d'exploitation (par exemple, kernel32 sur Windows, ou libc sur *nix), mais fournit également le support pour le chargement et l'interfaçage avec des bibliothèques dynamiques, telles que les DLL ou les objets partagés (Shared Objects ou SO) lors de l'exécution. Il embarque avec lui toute une série de types pour interagir avec les APIs système, et vous permet de définir assez facilement vos propres types complexes, tels que les structs et les unions, et vous permet de modifier des choses telles que le padding et l'alignement, si nécessaire. Il peut être un peu retors à utiliser, mais conjointement avec le module struct, vous recevez essentiellement un contrôle complet sur la façon dont vos types de données sont traduits en quelque chose d'utilisable par une méthode C(++) pure.

Équivalents de struct

MyStruct.h

```
struct my_struct {
    int a;
    int b;
};
```

MyStruct.py

```
import ctypes
class my_struct(ctypes.Structure):
    _fields_ = [("a", c_int),
                ("b", c_int)]
```

SWIG

SWIG, bien que pas strictement orienté Python (il prend en charge un grand nombre de langages de scripting), est un outil pour générer des bindings pour les langages interprétés à partir des fichiers d'en-tête C/C++. Il est extrêmement simple de l'utiliser : le consommateur doit simplement définir un fichier d'interface (détaillée dans le tutoriel et la documentation), inclure les en-têtes C/C ++ nécessaires, et exécuter l'outil de build sur eux. Alors qu'il a quelques limites, (il semble actuellement avoir des problèmes avec un petit sous-ensemble de nouvelles fonctionnalités C++, et avoir du code avec des templates partout pour travailler peut être un peu verbeux), il est très puissant et expose beaucoup de fonctionnalités pour Python avec peu d'effort. En outre, vous pouvez facilement étendre les bindings SWIG créés (dans le fichier d'interface) pour surcharger les opérateurs et les méthodes intégrées, recaster les exceptions C++ pour être capturables par Python, etc.

Exemple : Overloading __repr__

MyClass.h

```
#include <string>
class MyClass {
private:
    std::string name;
public:
    std::string getName();
};
```

myclass.i

```
%include "string.i"

%module myclass
%{
#include <string>
#include "MyClass.h"
%}

%extend MyClass {
    std::string __repr__()
    {
        return $self->getName();
    }
}

%include "MyClass.h"
```

Boost.Python

Boost.Python nécessite un peu plus de travail manuel pour exposer la fonctionnalité d'un objet C++, mais il est capable de fournir tous les fonctionnalités que SWIG fournit, et quelques unes pour inclure des wrappers pour accéder à des objets Python en C ++, en extrayant des objets wrappés SWIG-, et même permettant d'incorporer des morceaux de Python dans votre code C++.

CHAPITRE 4

Délivrer du “bon” code Python

Cette partie du guide se concentre sur le déploiement de votre code Python.

Packager votre code

Packagez votre code pour le partager avec d’autres développeurs. Par exemple, pour partager une bibliothèque pour que d’autres développeurs l’utilise dans leur application, ou pour les outils de développement comme ‘py.test’.

Un avantage de cette méthode de distribution est son écosystème bien établi d’outils tels que PyPI et pip, qui rendent facile pour d’autres développeurs de télécharger et d’installer votre paquet soit pour des expérimentations occasionnelles, ou comme un morceau de grands systèmes professionnels.

C’est une convention bien établie pour le code Python d’être partagé de cette façon. Si votre code n’est pas packagé sur PyPI, alors il sera plus difficile pour les autres développeurs de le trouver, et de l’utiliser dans le cadre de leur processus existant. Ils vont considérer de tels projets avec la suspicion importante qu’il soit mal géré ou abandonné.

L’inconvénient de la distribution de code comme ceci est qu’il repose sur le fait que le destinataire comprenne comment installer la version requise de Python, et soit en mesure et désireux d’utiliser des outils tels que pip pour installer les autres dépendances de votre code. C’est bien quand la distribution est pour les autres développeurs, mais rend cette méthode inadaptée pour la distribution d’applications à des utilisateurs finaux.

Le Python Packaging Guide fournit un guide complet sur la création et la maintenance des paquets Python.

Alternatives au packaging

Pour distribuer des applications à des utilisateurs finaux, vous devriez *geler votre application*.

Sous Linux, vous pouvez également envisager de *créer un paquet de distribution Linux* (c’est à dire des fichiers .deb pour Debian ou Ubuntu.)

Pour les développeurs Python

Si vous écrivez un module Python open source, PyPI , plus connu comme le *Cheeseshop* (magasin de fromages), est l’emplacement où l’héberger.

Pip vs. easy_install

Utilisez pip. Plus de détails ici

PyPi personnel

Si vous voulez installer des paquets à partir d'une autre source que PyPI, (par exemple, si vos paquets sont *propriétaires*), vous pouvez le faire hébergeant un simple serveur http, s'exécutant à partir du répertoire qui contient les paquets qui doivent être installés.

Montrer un exemple est toujours bénéfique

Par exemple, si vous souhaitez installer un paquet appelé `MyPackage.tar.gz`, et en supposant que c'est votre structure de répertoire :

- **archive**
  - **MyPackage**
    - MyPackage.tar.gz

Allez à votre invite de commande et tapez :

```
$ cd archive
$ python -m SimpleHTTPServer 9000
```

Cela exécute un simple serveur http, fonctionnant sur le port 9000 et listera tous les paquets (comme **MyPackage**). Maintenant, vous pouvez installer **MyPackage** en utilisant n'importe quel installateur de paquets Python. En utilisant Pip, vous devriez le faire avec :

```
$ pip install --extra-index-url=http://127.0.0.1:9000/ MyPackage
```

Avoir un dossier avec le même nom que le nom du package est **cruciale** ici. J'ai été trompé par cela, une seule fois. Mais si vous avez l'impression que créer un dossier appelé `MyPackage` et garder un fichier `MyPackage.tar.gz` dedans est *redondant*, vous pouvez toujours installer MyPackage en utilisant :

```
$ pip install  http://127.0.0.1:9000/MyPackage.tar.gz
```

pypiserver

Pypiserver est un serveur minimal compatible avec PyPi. Il peut être utilisé pour servir un jeu de paquets pour easy_install ou pip. Il inclue des fonctionnalités utiles comme une commande administrative (`-U`) qui mettra à jour tous ses paquets à leurs dernières versions trouvées sur PyPI.

PyPi hébergé sur S3

Une option simple pour un serveur PyPI personnel est d'utiliser Amazon S3. Un prérequis pour cela est que vous ayez un compte Amazon AWS avec un bucket S3.

1. **Installer tous vos requirements depuis PyPi ou une autre source**
2. **Installer pip2pi**

— `pip install git+https://github.com/wolever/pip2pi.git`

3. **Suivre le README pip2pi pour les commandes de pip2tgz et dir2pi**

— `pip2tgz packages/ MyPackage` (ou `pip2tgz packages/ -r requirements.txt`)
— `dir2pi packages/`

4. **Uploader les nouveaux fichiers**

— Utilisez un client comme Cyberduck pour synchroniser l'ensemble du dossier `packages` avec votre bucket s3

— Assurez-vous que vous uploadez `packages/simple/index.html` ainsi que tous les nouveaux fichiers et répertoires.

5. **Corriger les nouvelles permissions du fichier**

— Par défaut, lorsque vous uploadez de nouveaux fichiers dans le bucket S3, ils auront les mauvaises permissions définies.
— Utilisez la console web Amazon pour définir l'autorisation READ des fichiers à EVERYONE.
— Si vous avez un code HTTP 403 lorsque vous essayez d'installer un paquet, assurez-vous que vous avez configuré correctement les permissions.

6. **Terminé**

— Vous pouvez maintenant installer votre paquet avec `pip install --index-url=http://your-s3-bucket/packages/simple/ MyPackage`

Pour les distribution Linux

Créer un paquet pour distribution Linux est sans doute la "bonne manière" de distribuer votre code sous Linux.

Parce qu'un paquet de distribution n'inclue pas l'interpréteur Python, il rend le téléchargement et l'installation environ 2MB plus petit que *le gel de votre application*.

En outre, si une distribution met à disposition une nouvelle mise à jour de sécurité pour Python, votre application commencera automatiquement à utiliser cette nouvelle version de Python.

La commande bdist_rpm rend triviale la production d'un fichier RPM pour l'utilisation par des distributions comme Red Hat ou SuSE.

Cependant, créer et maintenir les différentes configurations requises pour chaque format de distribution (par exemple .deb pour Debian/Ubuntu, .rpm pour Red Hat/Fedora, etc) est une bonne quantité de travail. Si votre code est une application que vous envisagez de distribuer sur d'autres plates-formes, alors vous aurez également à créer et maintenir la configuration séparée nécessaire pour geler votre application pour Windows et OSX. Ce serait beaucoup moins de travail de simplement créer et maintenir une seule configuration pour un des *outils de freezing* multi-plateformes, qui va produire des exécutables autonomes pour toutes les distributions de Linux, ainsi que Windows et OSX.

Créer un paquet de distribution est également problématique si votre code est pour une version de Python qui n'est pas supportée par une distribution. Avoir à dire à des utilisateurs finaux de *certaines versions* d'Ubuntu qu'ils ont besoin d'ajouter le PPA 'dead-snakes' en utilisant la commande *sudo apt-repository* avant de pouvoir installer votre fichier .deb rend l'expérience hostile à l'utilisateur. Non seulement cela, mais vous auriez à maintenir un équivalent personnalisé de ces instructions pour chaque distribution, et pire encore, à demander à vos utilisateurs de lire, de comprendre et d'agir sur eux.

Cela dit, voici comment faire :
— Fedora
— Debian et Ubuntu
— Arch

Outils utiles

— fpm
— alien

Geler votre code

"Geler" ("Freezing") votre code c'est créer un exécutable avec un seul fichier à distribuer aux utilisateurs finaux, qui contient tout le code de l'application ainsi que l'interpréteur Python.

Les appplications comme 'Dropbox', 'Eve Online', 'Civilisation IV', et des clients BitTorrent font cela.

L'avantage de la distribution de cette façon est que votre application va "juste marcher", même si l'utilisateur n'a pas déjà la version requise de Python (ou aucune) installée. Sous Windows, et même sur de nombreuses distributions Linux et OS X, la bonne version de Python ne sera pas déjà installé.

En outre, le logiciel utilisateur final doit toujours être dans un format exécutable. Les fichiers se terminant par `.py` sont pour les ingénieurs logiciels et les administrateurs système.

Un inconvénient du freezing est qu'il augmentera la taille de votre distribution d'environ 2 à 12MB. En outre, vous serez responsable de délivrer des versions mises à jour de votre application lorsque des failles de sécurité Python sont patchées.

Alternatives au Freezing

Packager votre code est pour la distribution des bibliothèques ou d'outils pour d'autres développeurs.

Sous Linux, une alternative au freezing est de *créer un paquet de distribution Linux* (c'est à dire des fichiers .deb pour Debian ou Ubuntu, ou des fichiers .rpm pour Red Hat et SuSE.)

À faire

Compléter le stub "Geler votre code"

Comparaison des outils de Freezing

Solutions et plateformes/fonctionnalités supportées :

Solution	Windows	Linux	OS X	Python 3	Licence	Mode fichier unique	Import de fichier Zip	Eggs	support pkg_resources
bbFreeze	oui	oui	oui	non	MIT	non	oui	oui	oui
py2exe	oui	non	non	oui	MIT	oui	oui	non	non
pyInstaller	oui	oui	oui	oui	GPL	oui	non	oui	non
cx_Freeze	oui	oui	oui	oui	PSF	non	oui	oui	non
py2app	non	non	oui	oui	MIT	non	oui	oui	oui

Note : Geler du code Python sous Linux dans un exécutable Windows a un jour été supporté dans PyInstaller et ensuite été supprimé..

Note : Toutes les solutions nécessitent que les dll MS Visual C++ soient installées sur la machine cible, excepté py2app. Seul Pyinstaller fait des .exe auto-exécutables qui embarque les dll quand on passe `--onefile` à `Configure.py`.

Windows

bbFreeze

Le prérequis est d'installer *Python, Setuptools et la dépendance pywin32 sous Windows*.

À faire

Écrire les étapes pour les exécutables les basiques

py2exe

Le prérequis est d'installer *Python sous Windows*.

1. Télécharger et installer http://sourceforge.net/projects/py2exe/files/py2exe/
2. Écrit `setup.py` (Liste des options de configuration) :

```
from distutils.core import setup
import py2exe

setup(
    windows=[{'script': 'foobar.py'}],
)
```

3. (Optionnellement) inclure l'icône
4. (Optionnellement) Mode fichier unique
5. Générez un `.exe` dans le répertoire `dist` :

```
$ python setup.py py2exe
```

6. Fournissez les DLL du runtime Microsoft Visual C++. Deux options : installer globalement les dll sur ma machine cible ou distribuer les dll à côté du .exe.

PyInstaller

Le prérequis est d'avoir installé *Python, Setuptools et la dépendance pywin32 sous Windows*.

— Tutoriel le plus basique
— Manuel

OS X

py2app

PyInstaller

PyInstaller peut être utilisé pour construire des exécutables Unix et applications fenêtrées sur Mac OS X 10.6 (Snow Leopard) ou plus récent.

Pour installer PyInstaller, utilisez pip :

```
$ pip install pyinstaller
```

Pour créer un exécutable Unix standard, depuis disons `script.py`, utilisez :

```
$ pyinstaller script.py
```

cela créé,

— un fichier `script.spec`, analogue à un fichier `make`
— un dossier `build`, qui contient quelques fichiers de log
— un dossier `dist`, qui contient l'exécutable principal `script`, et quelques bibliothèques Python dépendantes,

toutes dans le même dossier que `script.py`. PyInstaller met toutes les bibliothèques Python utilisées dans `script.py` dans le dossier `dist`. Donc lors de la distribution de l'exécutable, distribuez l'ensemble du dossier `dist`.

Le fichier `script.spec` peut être édité pour customiser le build, avec des options comme

— Embarquer les fichiers de données avec l'exécutable
— y compris les bibliothèques de run-time (fichiers `.dll` ou `.so`) que PyInstaller ne peut pas déduire automatiquement
— ajout des options du run-time Python à l'exécutable,

Maintenant `script.spec` peut être exécuter avec `pyinstaller` (plutôt que d'utiliser encore `script.py`) :

```
$ pyinstaller script.spec
```

Pour créer une application OS X autonome fenêtrée, utilisez l'option `--windowed`

```
$ pyinstaller --windowed script.spec
```

Cela créé `script.app` dans le dossier `dist`. Assurez-vous d'utiliser des paquets graphiques dans votre code Python, comme PyQt ou PySide, pour contrôler les parties graphiques de votre application.

Il y a plusieurs option `script.spec` liées à l'encapsulation d'applications Mac OS X ici. Par exemple, pour spécifier une icône pour l'application, utilisez l'option `icon=\path\to\icon.icns`.

Linux

bbFreeze

PyInstaller

CHAPITRE 5

Environnements de développement Python

Cette partie du guide se concentre sur l'environnement de développement Python et les outils pour les meilleures pratiques qui sont disponibles pour écrire du code Python.

Votre environnement de développement

Éditeurs de texte

Tout ce qui peut éditer du texte brut peut fonctionner pour l'écriture de code Python. Cependant, utiliser un éditeur plus puissant peut vous rendre la vie un peu plus facile.

Vim

Vim est un éditeur de texte qui utilise des raccourcis clavier pour éditer au lieu de menus ou d'icônes. Il y a des couples plugins et paramètres pour l'éditeur Vim pour faciliter le développement Python. Si vous développez seulement en Python, un bon point de départ est de définir les paramètres par défaut pour l'indentation et les valeurs pour les retours à la ligne conformes à **PEP 8**. Dans votre répertoire "home", ouvrez un fichier appelé `.vimrc` et ajoutez les lignes suivantes :

```
set textwidth=79  " lines longer than 79 columns will be broken
set shiftwidth=4  " operation >> indents 4 columns; << unindents 4 columns
set tabstop=4     " a hard TAB displays as 4 columns
set expandtab     " insert spaces when hitting TABs
set softtabstop=4 " insert/delete 4 spaces when hitting a TAB/BACKSPACE
set shiftround    " round indent to multiple of 'shiftwidth'
set autoindent    " align the new line indent with the previous line
```

Avec ces configurations, des retours à la ligne sont insérés après 79 caractères et l'indentation est définie à 4 espaces par tabulation. Si vous utilisez Vim pour d'autres langages, il y a un plugin pratique appelé indent, qui peut gérer la configuration de l'indentation des fichiers source Python.

Il y a aussi un plugin de syntaxe pratique appelée syntax avec quelques améliorations par rapport au fichier de syntaxe inclus dans Vim 6.1.

Ces plugins vous fournissent un environnement de base pour le développement en Python. Pour tirer le meilleur parti de Vim, vous devriez vérifier en continu votre code pour les erreurs de syntaxe et la conformité à PEP8. Heureusement, PEP8 and Pyflakes vont le faire pour vous. Si votre Vim est compilé avec `+python`, vous pouvez également utiliser certains plugins très pratiques pour faire ces vérifications depuis l'éditeur.

Pour vérification PEP8 et pyflakes, vous pouvez installer vim-flake8. Maintenant, vous pouvez mapper la fonction `Flake8` à tout raccourci clavier ou de action que vous voulez dans Vim. Le plugin affichera les erreurs au bas de l'écran, et fournira un moyen facile de sauter à la ligne correspondante. Il est très pratique d'appeler cette fonction chaque fois que vous enregistrez un fichier. Pour ce faire, ajoutez la ligne suivante à votre `.vimrc` :

```
autocmd BufWritePost *.py call Flake8()
```

Si vous utilisez déjà syntastic, vous pouvez le configurer pour exécuter Pyflakes à l'écriture et montrer les erreurs et avertissements dans la fenêtre de correction rapide. Un exemple de configuration pour faire cela et qui montre également le statut et les messages d'alerte dans la barre d'état serait

```
set statusline+=%#warningmsg#
set statusline+=%{SyntasticStatuslineFlag()}
set statusline+=%*
let g:syntastic_auto_loc_list=1
let g:syntastic_loc_list_height=5
```

Python-mode

Python-mode est une solution complexe pour travailler avec du code Python dans Vim. Il a :

- Vérification de code Python asynchrone (`pylint`, `pyflakes`, `pep8`, `mccabe`) avec n'importe quelle combinaison
- Refactoring de code et autocomplétion avec Rope
- Dépliage de code Python rapide
- Support de virtualenv
- Recherche dans la documentation Python et exécution de code Python
- Correction automatiques des erreurs PEP8

Et plus.

SuperTab

SuperTab est un petit plugin Vim qui rend la complétion de code plus pratique en utilisant la touche `<Tab>` (tabulation) ou n'importe quelles autres touches personnalisées.

Emacs

Emacs est un autre éditeur de texte puissant. Il est entièrement programmable (Lisp), mais il peut nécessiter un peu de travail pour être correctement configurer. Un bon début si vous êtes déjà un utilisateur Emacs est Python Programming in Emacs sur EmacsWiki.

1. Emacs lui-même contient un mode pour Python.

TextMate

> TextMate apporte l'approche d'Apple aux systèmes d'exploitation dans le monde des éditeurs de texte. En établissant un pont entre les fondements UNIX et les interfaces graphiques,TextMate choisit le meilleur des deux mondes au bénéfice aussi bien des scripteurs experts que des utilisateurs novices.

Sublime Text

> Sublime Text est un éditeur de texte raffiné pour le code, le balisage et la prose. Vous allez adorer l'interface utilisateur agréable, les fonctionnalités extraordinaires et les performances incroyables.

Sublime Text offre un excellent support pour l'édition du code Python et utilise Python pour son API de plugin. Il a également une grande diversité de plugins, certains permettent une vérification PEP8 dans l'éditeur et le "linting" de code.

Atom

> Atom est un éditeur de texte hackable pour le 21e siècle, construit sur atom-shell et sur la base de tout ce que nous aimons dans nos éditeurs favoris.

Atom est basé sur le web (HTML, CSS, JS), en se concentrant sur une conception modulaire et le développement de plugin facile. Il est livré avec un gestionnaire de paquets natif et une pléthore de paquets. Il est recommandé pour le développement Python d'utiliser Linter combiné avec linter-flake8.

IDEs

PyCharm / IntelliJ IDEA

PyCharm est développé par JetBrains, aussi connu pour IntelliJ IDEA. Les deux partagent la même base de code et la plupart des fonctionnalités de PyCharm peuvent être accessibles dans IntelliJ avec le Plug-in Python gratuit. Il y a deux versions de PyCharm : une édition Professionnel (avec 30 jours d'essai gratuits) et une édition Communautaire (sous licence Apache 2.0) avec moins de fonctionnalités.

Enthought Canopy

Enthought Canopy est un IDE Python qui est axé vers les scientifiques et les ingénieurs car il fournit des bibliothèques pré-installées pour l'analyse de données.

Eclipse

Le plugin Eclipse le plus populaire pour le développement Python est PyDev de Aptana.

Komodo IDE

Komodo IDE est développé par ActiveState et est un IDE commercial pour Windows, Mac et Linux. KomodoEdit est l'alternative open source.

Spyder

Spyder est un IDE spécifiquement orienté vers l'utilisation de bibliothèques Python scientifiques (notamment Scipy). Il inclut l'intégration avec pyflakes, pylint et rope.

Spyder est open-source (libre), offre la complétion de code, la coloration syntaxique, un navigateur de classes et de fonctions et l'inspection d'objets.

WingIDE

WingIDE est un IDE spécifique Python. Il fonctionne sur Linux, Windows et Mac (comme application X11, ce qui frustre certains utilisateurs de Mac).

WingIDE offre la complétion de code, la coloration syntaxique, la navigation de sources, le débogueur graphique et le support pour les systèmes de gestion de version.

NINJA-IDE

NINJA-IDE (depuis l'acronyme récursif : "Ninja-IDE Is Not Just Another IDE") est un IDE multi-plateformes, spécialement conçu pour créer des applications Python, et qui fonctionne sous les systèmes d'exploitation de bureau Linux/X11, Mac OS X et Windows. Les installateurs pour ces plates-formes peuvent être téléchargés à partir du site web.

NINJA-IDE est un logiciel open-source (licence GPLv3) et est développé en Python et Qt. Les fichiers source peuvent être téléchargés depuis GitHub.

Eric (l'IDE Python Eric)

Eric est un IDE Python très complet offrant l'autocomplétion du code source, la coloration syntaxique, le support des systèmes de gestion de version, le support Python 3, un navigateur Web intégré, un shell python, un débogueur intégré et un système de plug-in flexible. Écrit en Python, il est basé sur la boîte à outils graphique Qt, intégrant le contrôle de l'éditeur Scintilla. Eric est un projet de logiciel open-source (licence GPLv3) avec plus de dix ans de développement actif.

Outils pour l'interpréteur

Environnements virtuels

Les environnements virtuels fournissent un moyen puissant pour isoler les dépendances de paquets d'un projet. Cela signifie que vous pouvez utiliser des paquets particuliers à un projet Python sans les installer sur l'ensemble du système et ainsi en évitant ainsi la les conflits de version potentiels.

Pour commencer à l'utiliser et plus d'informations : documents sur les environnements virtuels.

pyenv

pyenv est un outil pour autoriser l'installation de plusieurs versions de l'interpréteur Python en même temps. Cela résout le problème d'avoir différents projets nécessitant différentes versions de Python. Par exemple, il devient très facile d'installer Python 2.7 pour la compatibilité dans un projet, tout en utilisant Python 3.4 comme l'interpréteur par défaut. pyenv n'est pas limité aux versions CPython - installera également les interpréteurs PyPy, anaconda, miniconda, Stackless, Jython et IronPython.

pyenv fonctionne en remplissant un répertoire `shims` avec des fausses versions de l'interpréteur Python (plus d'autres outils comme `pip` et `2to3`). Quand le système recherche un programme nommé `python`, il regarde en premier dans le répertoire `shims`, et utilise la fausse version, qui à son tour transmet la commande à pyenv. pyenv ensuite détermine quelle version de Python doit être exécutée en fonction des variables d'environnement, de fichiers `.python-version` et de l'environnement global par défaut.

pyenv n'est pas un outil pour la gestion des environnements virtuels, mais il y a le plugin pyenv-virtualenv qui automatise la création de différents environnements, et permet également de utiliser les outils de pyenv existants pour passer à des différents environnements à partir de variables d'environnement ou de fichiers `.python-version`.

Autres outils

IDLE

IDLE est un environnement de développement intégré (IDE) qui fait partie de la bibliothèque Python standard. Il est entièrement écrit en Python et utilise la boîte à outils graphique Tkinter. Bien IDLE ne convienne pas pour le développement à part entière en utilisant Python, il est très utile pour essayer de petits bouts de code Python et expérimenter avec les différentes fonctionnalités en Python.

Il fournit les fonctionnalités suivantes :

- Fenêtre du shell Python (interpréteur)
- Éditeur de texte multi-fenêtres qui colorie le code Python
- Facilité de débugage minimale

IPython

IPython fournit une boîte à outils riche pour vous aider à tirer le meilleur parti de l'interactivité de Python. Ses principales composants sont :

- Shells Python puissants (basés sur le terminal et Qt)
- Un "notebook" basé sur le Web avec les mêmes caractéristiques de base, mais le support de médias riches, du texte, du code, des expressions mathématiques et des graphiques intégrés "inline".
- Support pour la visualisation interactive de données et l'utilisation de boîtes à outils avec interface graphique.
- Interpréteurs flexibles, intégrables à charger dans vos propres projets.
- Outils pour le traitement parallèle interactif et de haut niveau.

```
$ pip install ipython
```

Pour télécharger et installer IPython avec toutes ses dépendances optionnelles pour le notebook, qtconsole, les tests et les autres fonctionnalités

```
$ pip install ipython[all]
```

BPython

bpython est une interface alternative à l'interpréteur Python pour les systèmes d'exploitation de type Unix. Il a les caractéristiques suivantes :

- Coloration syntaxique "in-line".
- Autocomplétion similaire à readline avec suggestions affichées au fur à mesure que vous tapez.
- Liste des paramètres attendus pour n'importe quelle fonction Python.
- Fonction "Rewind" pour réafficher la dernière ligne de code depuis la mémoire et la ré-évaluer.
- Envoyez le code entré vers un pastebin.
- Sauvegarde du texte entré dans un fichier.
- Auto-indentation
- Support Python 3.

```
$ pip install bpython
```

ptpython

ptpython est une ligne de commande interactive (REPL) construite par dessus la bibliothèque prompt_toolkit. Elle est considérée comme une alternative à BPython. Les fonctionnalités incluent :

- Coloration syntaxique
- Autocomplétion

— Édition multi-lignes
— Emacs et mode VIM
— Encapsulation d'une ligne de commande interactive (REPL) à l'intérieur de votre code
— Validation de syntaxe
— Pages de tabulation
— Support pour l'intégration avec le shell IPython, en installant IPython `pip install ipython` et en exécutant `ptipython`.

```
$ pip install ptpython
```

Environnements virtuels

Un environnement virtuel est un outil pour garder les dépendances requises par différents projets dans des emplacements séparés, en créant des environnements virtuels Python pour eux. Il résout le dilemme "le projet X dépend de la version 1.x mais le projet Y nécessite la 4.x", et garde votre répertoire site-packages global propre et gérable.

Par exemple, vous pouvez travailler sur un projet qui nécessite Django 1.3 tout en maintenant aussi un projet qui nécessite Django 1.0.

virtualenv

virtualenv est un outil pour créer des environnements virtuels Python isolés. virtualenv crée un dossier qui contient tous les exécutables nécessaires pour utiliser les paquets qu'un projet Python pourrait nécessiter.

Installez virtualenv via pip :

```
$ pip install virtualenv
```

Usage basique

1. Créer un environnement virtuel pour un projet :

```
$ cd my_project_folder
$ virtualenv venv
```

`virtualenv venv` créera un dossier dans le répertoire courant qui contiendra les fichiers exécutables Python, et une copie de la bibliothèque `pip` que vous pouvez utiliser pour installer d'autres paquets. Le nom de l'environnement virtuel (dans ce cas, c'était `venv`) peut être n'importe quoi. Omettre le nom placera les fichiers dans le répertoire courant à la place.

Cela crée une copie de Python selon le répertoire où vous avez exécuté la commande, en le plaçant dans un dossier nommé `venv`.

Vous pouvez utiliser un interpréteur Python de votre choix.

```
$ virtualenv -p /usr/bin/python2.7 venv
```

Cela utilisera l'interpréteur Python dans `/usr/bin/python2.7`

2. Pour commencer à utiliser l'environnement virtuel, il doit être activé :

```
$ source venv/bin/activate
```

Le nom de l'environnement virtuel actuel apparaît maintenant sur la gauche de l'invite (c'est à dire `(venv)Votre-Ordinateur:votre projet VotreNomUtilisateur$`) pour vous indiquer qu'il est actif. A partir de maintenant, tous les paquets que vous installez en utilisant pip seront placés dans le dossier `venv`, isolés de l'installation globale de Python.

Installez les paquets comme d'habitude, par exemple :

```
$ pip install requests
```

3. Si vous avez terminé de travailler dans l'environnement virtuel pour le moment, vous pouvez le désactiver :

```
$ deactivate
```

Cela vous fait revenir à l'interpréteur par défaut Python du système, avec toutes ses bibliothèques installées.

Pour supprimer une environnement virtuel, supprimez juste son dossier. (Dans ce cas, ce serait `rm -rf venv`.)

Après un certain temps, cependant, vous pourriez vous retrouver avec un grand nombre d'environnements virtuels un peu partout dans votre système, et il est possible que vous oubliez leurs noms ou où ils ont été placés.

Autres notes

Exécuter `virtualenv` avec l'option `--no-site-packages` n'inclura pas les paquets qui sont installés globalement. Cela peut être utile pour garder la liste des paquets propre dans le cas où il est nécessaire d'y accéder plus tard. [Ceci est le comportement par défaut de `virtualenv` 1.7 et supérieur.]

Afin de garder votre environnement cohérent, c'est une bonne idée de "geler" l'état actuel des paquets de l'environnement. Pour ce faire, exécutez

```
$ pip freeze > requirements.txt
```

Cela va créer un fichier `requirements.txt`, qui contient une liste simple de tous les paquets qui sont dans l'environnement actuel, et leurs versions respectives.Vous pouvez voir la liste des paquets installés sans le format requirements en utilisant "pip list". Plus tard, il sera plus facile pour un développeur différent (ou vous, si vous avez besoin de recréer l'environnement) d'installer les paquets en utilisant les mêmes versions :

```
$ pip install -r requirements.txt
```

Cela peut aider à assurer la cohérence entre installations, entre déploiements et entre développeurs.

Enfin, rappelez-vous d'exclure le dossier de l'environnement virtuel du gestionnaire de version en l'ajoutant à la liste des fichiers ignorés.

virtualenvwrapper

virtualenvwrapper fournit un jeu de commandes qui permet le travail avec des environnements virtuels beaucoup plus agréable. Il place également tous vos environnements virtuels dans un seul endroit.

Pour installer (assurez-vous que **virtualenv** est déjà installé) :

```
$ pip install virtualenvwrapper
$ export WORKON_HOME=~/Envs
$ source /usr/local/bin/virtualenvwrapper.sh
```

(Instructions d'installation virtualenvwrapper complètes.)

Pour Windows, vous pouvez utiliser le virtualenvwrapper-win.

Pour installer (assurez-vous que **virtualenv** est déjà installé) :

```
$ pip install virtualenvwrapper-win
```

Sous Windows, le chemin par défaut pour WORKON_HOME est %USERPROFILE%Envs

Usage basique

1. Créez un environnement virtuel :

```
$ mkvirtualenv venv
```

Cela créé un dossier `venv` dans `~/Envs`.

2. Travailler sur un environnement virtuel :

```
$ workon venv
```

Alternativement, vous pouvez faire un projet, ce qui crée un environnement virtuel, et aussi un répertoire de projet à l'intérieur de `$PROJECT_HOME`, dans lequel vous êtes automatiquement placé via `cd` quand vous lancez `workon myproject`.

```
$ mkproject myproject
```

virtualenvwrapper fournit une auto-complétion à la tabulation sur les noms d'environnement. Il aide vraiment quand vous avez beaucoup d'environnements et avez du mal vous souvenir de leurs noms.

`workon` désactive aussi n'importe quel des environnements où vous êtes actuellement, de sorte que vous pouvez rapidement basculer entre les environnements.

3. La désactivation est toujours la même :

```
$ deactivate
```

4. Pour supprimer :

```
$ rmvirtualenv venv
```

Autres commandes utiles

`lsvirtualenv` Lister tous les environnements

`cdvirtualenv` Naviguez dans le répertoire de l'environnement virtuel actuellement activé, de sorte que vous pouvez parcourir son `site-packages`, par exemple.

`cdsitepackages` Come ci-dessus, mais directement dans le répertoire `site-packages`.

`lssitepackages` Montre le contenu du répertoire `site-packages`.

Liste complète des commandes de virtualenvwrapper <http://virtualenvwrapper.readthedocs.org/en/latest/command_ref.html>‘_.

virtualenv-burrito

Avec virtualenv-burrito, vous pouvez avoir un environnement de travail virtualenv + virtualenvwrapper en une seule commande.

autoenv

Quand vous faites un `cd` dans un répertoire contenant un `.env`, autoenv active l'environment automatiquement, par magie .

Installez-le sous Mac OS X en utilisant `brew` :

```
$ brew install autoenv
```

Et sous Linux :

```
$ git clone git://github.com/kennethreitz/autoenv.git ~/.autoenv
$ echo 'source ~/.autoenv/activate.sh' >> ~/.bashrc
```

Configuration supplémentaire de Pip et Virtualenv

Obliger l'utilisation d'un environnement virtuel actif pour `pip`

Maintenant, il devrait être clair que l'utilisation d'environnements virtuels est une excellente façon de garder votre environnement de développement propre et de garder les 'requirements' des différents projets distincts.

Lorsque vous commencez à travailler sur de nombreux projets différents, il peut être difficile de se rappeler d'activer l'environnement virtuel associé quand vous revenez à un projet spécifique. De ce fait, il est très facile d'installer des paquets globalement tout en pensant que vous installez vraiment le paquet pour l'environnement virtuel du projet. Au fil du temps, cela peut se traduire par une liste globale de paquets désordonnée.

Pour vous assurer que vous installez des paquets dans votre environnement virtuel actif lorsque vous utilisez `pip install`, envisagez d'ajouter les deux lignes suivantes à votre fichier `~/.bashrc` :

```
export PIP_REQUIRE_VIRTUALENV=true
```

Après avoir enregistré ce changement et rechargé le fichier `~/.bashrc` avec `source ~/.bashrc`, pip ne vous laissera plus installer des paquets si vous n'êtes pas dans un environnement virtuel. Si vous essayez d'utiliser `pip install` en dehors d'un environnement virtuel, pip va gentiment vous rappeler qu'un environnement virtuel actif est nécessaire pour installer des paquets.

```
$ pip install requests
Could not find an activated virtualenv (required).
```

Vous pouvez également faire cette configuration en éditant votre fichier `pip.conf` ou `pip.ini`. `pip.conf` est utilisé par les systèmes d'exploitation Unix et Mac OS X et il peut être trouvé dans :

```
$HOME/.pip/pip.conf
```

De même, le fichier `pip.ini` est utilisé par les systèmes d'exploitations Windows et peut trouvé dans :

```
%HOME%\pip\pip.ini
```

Si vous n'avez pas de fichiers a `pip.conf` ou `pip.ini` dans ces emplacements, vous pouvez créer un nouveau fichier avec le nom correct pour votre système d'exploitation.

Si vous avez déjà un fichier de configuration, ajoutez juste la ligne suivante à sous la configuration `[global]` pour obliger l'usage d'un environnement virtuel actif :

```
require-virtualenv = true
```

Si vous n'avez pas un fichier de configuration, vous devrez en créer un nouveau et ajouter les lignes suivantes à votre nouveau fichier :

```
[global]
require-virtualenv = true
```

Vous aurez bien sûr besoin d'installer certains paquets globalement (généralement ceux que vous utilisez dans différents projets systématiquement) et cela peut être accompli en ajoutant ce qui suit à votre fichier `~/.bashrc` :

```
gpip() {
    PIP_REQUIRE_VIRTUALENV="" pip "$@"
}
```

Après avoir sauvé les changements et rechargé votre fichier `~/.bashrc`, vous pouvez maintenant installer des paquets globalement en exécutant `gpip install`. Vous pouvez changer le nom de la fonction à votre guise, gardez juste à l'esprit que vous devrez utiliser ce nom lorsque vous essayerez d'installer des paquets globalement avec pip.

Mettre en cache les paquets pour les utilisations futures

Chaque développeur a ses bibliothèques préférées et quand vous travaillez sur un grand nombre de projets différents, vous êtes amenés à avoir un certain chevauchement entre les bibliothèques que vous utilisez. Par exemple, vous utilisez peut-être la bibliothèque `requests` dans un grand nombre de projets différents.

Il est certainement inutile de retélécharger les mêmes paquets/bibliothèques chaque fois que vous commencez à travailler sur un nouveau projet (et dans un nouvel environnement virtuel par conséquent). Heureusement, vous pouvez configurer pip de manière à ce qu'il essaie de réutiliser les paquets déjà installés.

Sur les systèmes UNIX, vous pouvez ajouter la ligne suivante à votre fichier `.bashrc` ou `.bash_profile`.

```
export PIP_DOWNLOAD_CACHE=$HOME/.pip/cache
```

Vous pouvez définir les chemins n'importe où (à partir du moment où vous avez les droits en écriture). Après avoir ajouté cette ligne, faites un `source` de votre fichier `.bashrc` (ou `.bash_profile`) et vous serez fin prêt.

Une autre manière de faire la même configuration est via les fichiers `pip.conf` ou `pip.ini`, selon votre système d'exploitation. Si vous êtes sous Windows, vous pouvez ajouter la ligne suivante à votre fichier `pip.ini` sous la configuration `[global]` :

```
download-cache = %HOME%\pip\cache
```

De même, sur les systèmes UNIX, vous devriez simplement ajouter la ligne suivante à votre fichier `pip.conf` sous la configuration `[global]` :

```
download-cache = $HOME/.pip/cache
```

Même si vous pouvez utiliser le chemin que vous voulez pour stocker votre cache, il est recommandé que vous créiez un nouveau dossier *dans* le dossier où votre fichier `pip.conf` ou `pip.ini` est situé. Si vous ne vous faites pas confiance complètement sur cette utilisation obscure des chemins, il suffit d'utiliser les valeurs fournies ici et tout ira bien.

CHAPITRE 6

Notes supplémentaires

Cette partie du guide, qui est principalement de la prose, commence avec quelques informations de contexte à propos de Python, puis met l'accent sur sur les prochaines étapes.

Introduction

Depuis le site web Python officiel :

Python est langage de programmation généraliste de haut niveau similaire à Tcl, Perl, Ruby, Scheme ou Java. Quelques-unes des principales fonctionnalités clés incluent :

— **une syntaxe très claire, lisible**
La philosophie de Python s'axe sur la lisibilité, que ce soit à partir de blocs de code délimités par des espaces significatifs à des mots clés intuitifs à la place de l'usage d'une ponctuation insondable.

— **de vastes bibliothèques standard et des modules tiers pour pratiquement toute tâche**
Python est parfois décrit avec les mots "batteries included" (ou "batteries incluses") en raison de sa vaste bibliothèque standard, qui comprend des modules pour les expressions régulières, les entrées/sorties sur les fichiers, la manipulation de fractions, la sérialisation d'objets, et bien plus encore.
En outre, l'index des paquets Python (PYthon Package Index ou PYPI) est mis à disposition pour les utilisateurs afin de soumettre leurs paquets pour une utilisation large, similaire à CPAN de Perl. Il y a une communauté florissante de frameworks et d'outils Python très puissants comme le framework web Django et l'ensemble de routines mathématiques de NumPy.

— **intégration avec d'autres systèmes**
Python peut s'intégrer avec les bibliothèques Java, ce qui lui permet d'être utilisé avec le riche environnement Java auxquels les programmeurs Entreprise sont habitués. Il peut également être étendu par des modules C ou C ++ lorsque la vitesse est primordiale.

— **Ubiquité sur les ordinateurs**
Python est disponible sur Windows, *nix, et Mac. Il fonctionne partout où la machine virtuelle Java fonctionne, et l'implémentation de référence CPython peut aider à porter Python n'importe où il y a un compilateur fonctionnel C.

— **une communauté accueillante**
Python a une communauté grande et dynamique *communauté* qui maintient des wikis, des conférences, d'innombrables dépôts, des listes de diffusion, des canaux IRC, et bien plus encore. Vérifiez, la communauté Python aide même à écrire ce guide !

A propos de ce guide

But

Le guide de l'auto-stoppeur pour Python existe pour fournir aux développeurs novices comme experts un manuel des meilleurs pratiques pour l'installation, la configuration et l'usage de Python au quotidien.

Par la communauté

Ce guide est architecturé et maintenu par Kenneth Reitz de manière ouverte. C'est une effort piloté par la communauté qui sert un but : servir la communauté.

Pour la communauté

Toutes les contributions au Guide sont les bienvenues, de Pythoneux de tous les niveaux. Si vous pensez qu'il y a une lacune dans ce que le guide couvre, forkez le guide sur GitHub et soumettez une "pull request".

Les contributions sont bienvenues de la part de tout le monde, que ce soit une vieille main ou un jeune Pythoneux, et les auteurs du guide seront heureux de vous aider si vous avez des questions sur la pertinence, l'exhaustivité ou l'exactitude d'une contribution.

Pour commencer à travailler sur le guide de l'auto-stoppeur, voir la page Contribuer.

La communauté

BDFL

Guido van Rossum, le créateur de Python, est souvent désigné comme le dictateur bienveillant pour la vie (Benevolent Dictator For Life ou BDFL).

Python Software Foundation

La mission de la Python Software Foundation est de promouvoir, de protéger et de faire progresser le langage de programmation Python et de soutenir et faciliter la croissance d'une communauté variée et internationale de programmeurs Python.

En apprendre plus sur la PSF.

PEPs

Les PEPs sont les *Python Enhancement Proposals* (Propositions d'amélioration de Python). Elles décrivent des modifications sur Python lui-même, ou sur les standards autour du langage.

Il y a trois types différents de PEPs (comme défini par la **PEP 1**) :

Standards Décrit une nouvelle fonctionnalité ou une implémentation.

Informatif Décrit un problème de conception, des orientations générales ou un information pour la communauté.

Procédés Décrit un processu lié à Python.

PEPs notables

Il y a quelques PEPs qui peuvent être considérées comme une lecture obligatoire :

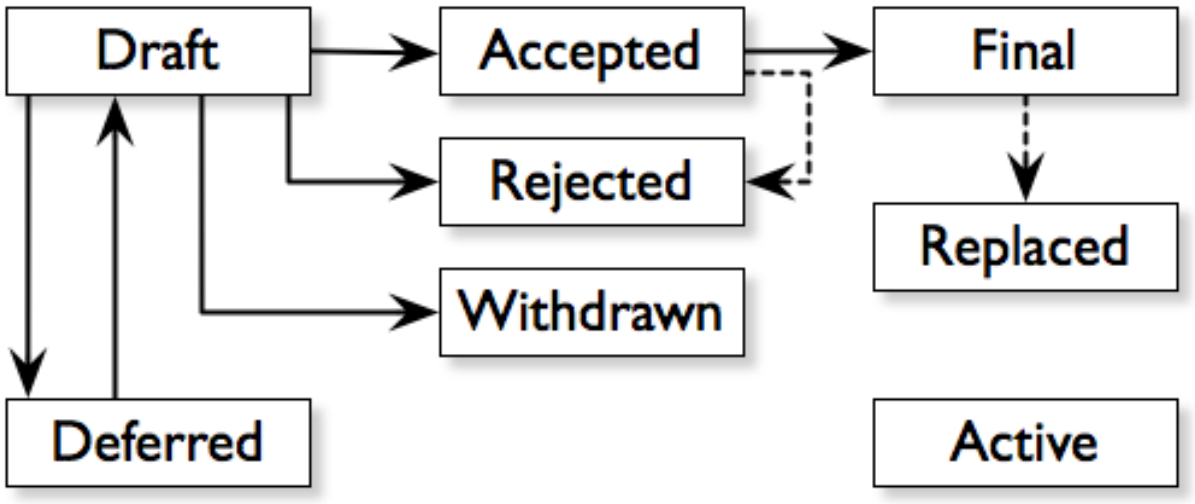
- **PEP 8 : Le guide de style Python.** Lisez-le, en entier. Suivez-le.
- **PEP 20 : Le "Zen of Python".** Une liste de 19 déclarations qui expliquent brièvement la philosophie derrière Python.
- **PEP 257 : Les conventions docstring.** Donne les lignes directrices pour la sémantique et les conventions associées avec les docstrings Python.

Vous pouvez en lire plus sur L'index des PEPs.

Soumission d'une PEP

Les PEPs sont évaluées par des pairs et acceptées/rejetées après beaucoup de discussions. Tout le monde peut écrire et soumettre une PEP pour évaluation.

Voici un aperçu du workflow d'acceptation PEP :



Conférences Python

Les événements majeurs pour la communauté Python sont des conférences de développeurs. Les deux conférences les plus notables sont la PyCon, qui se tient aux Etats-Unis, et son homologue européenne, EuroPython.

Une liste complète de conférences est maintenue sur pycon.org.

Groupes d'utilisateurs Python

Les groupes d'utilisateurs sont où un groupe de développeurs Python qui se réunissent pour présenter ou parler de sujets d'intérêts Python. Une liste des groupes d'utilisateurs locaux est maintenue à la Python Software Foundation Wiki.

Apprendre Python

Débutant

Le tutoriel Python

C'est le tutoriel officiel. Il couvre tous les basiques et offre une visite guidée du langage et de la bibliothèque standard. Il est recommandé pour ceux qui ont besoin d'un guide pour démarrer rapidement sur le langage.

Le tutoriel Python (fr)

Python for Beginners

thepythonguru.com est un tutoriel, en anglais, se concentrant sur les programmeurs débutants. Il couvre de nombreux concepts Python en profondeur. Il vous enseigne aussi des concepts avancés de Python comme les expressions lambda ou les expressions régulières. Enfin, il termine le tutoriel sur"Comment accéder à une base de données MySQL en utilisant Python"

Python for beginners

Tutoriel interactif Learn Python

Learnpython.org est une manière facile et non-intimidante d'être introduit à Python. Le site web prend la même approche utilisée par le site web populaire'Try Ruby <http://tryruby.org/>'_. Il a un interpréteur Python interactif encapsulé dans le site qui vous permet de parcourir les leçons sans avoir à installer Python localement.

Learn Python

Si vous voulez une livre plus traditionnel, *Python For You and Me* est une excellente ressource pour apprendre tous les aspects du langage.

Python for You and Me

Online Python Tutor

Online Python Tutor vous donne une représentation visuelle pas à pas de la façon dont votre programme fonctionne. Python Tutor aide les gens à dépasser une barrière fondamentale pour apprendre à programmer en comprenant ce qu'il se passe quand l'ordinateur exécute chaque ligne de source code du programme.

Online Python Tutor

Invent Your Own Computer Games with Python

Ce livre pour débutant est pour ceux qui n'ont pas d'expérience en programmation du tout. Chaque chapitre a un code source pour une petit jeu, utilisant ces programmes exemple pour démontrer des concepts de programmation pour donner au lecteur une idée de ce à quoi les programmes "ressemblent".

Invent Your Own Computer Games with Python

Hacking Secret Ciphers with Python

Ce livre enseigne la programmation Python et de la cryptographie de base pour les débutants absolus. Les chapitres fournissent le code source pour divers chiffrements, ainsi que des programmes qui peuvent les briser.

Hacking Secret Ciphers with Python

Learn Python the Hard Way

Ceci est un excellent guide pour le programmeur débutant sur Python. Il couvre “hello world” de la console jusqu’au web.

Learn Python the Hard Way

Crash into Python

Aussi connu sous le nom *Python for Programmers with 3 Hours*, ce guide donne aux développeurs expérimentés venant d’autres langages un cours accéléré sur Python.

Crash into Python

Dive Into Python 3

Dive Into Python 3 est un bon livre pour ceux qui sont prêts à passer à Python 3. C’est une bonne lecture si vous passer de Python 2 à 3 ou si vous avez déjà un peu d’expérience en programmation dans un autre langage.

Dive Into Python 3

Think Python : How to Think Like a Computer Scientist

Think Python tente de donner une introduction aux concepts de base en informatique via l’utilisation du langage Python. L’objectif était de créer un livre avec de nombreux exercices, un jargon minimal et une section dans chaque chapitre consacré au débugage.

Tout en explorant les différentes fonctionnalités disponibles dans le langage Python, l’auteur tisse dans divers design patterns et les meilleures pratiques.

Le livre inclue également plusieurs études de cas qui font explorer par le lecteur les thèmes abordés dans le livre plus en détail en appliquant ces sujets à des exemples du monde réel. Les études de cas comprennent des devoirs sur les interfaces graphiques et l’analyse de Markov.

Think Python

Python Koans

Python Koans est un port de Ruby Koans par Edgecase. Il utilise un approche dirigé par le test, voir la SECTION CONCEPTION PILOTÉE PAR LES TESTS pour fournir un tutoriel interactif pour enseigner les concepts de base Python. En corrigeant les déclarations d’assertion qui ne échouent dans un script de test, cela fournit des étapes séquentielles pour apprendre Python.

Pour ceux qui sont habitués aux langages et à résoudre des puzzles eux-mêmes, cela peut être un option attrayante, amusante. Pour ceux qui découvrent Python et la programmation, avoir une ressource ou référence supplémentaire peut s’avérer utile.

Python Koans

Plus d’informations sur le développement piloté par les tests peuvent être trouvées via ces ressources :

Test Driven Development

A Byte of Python

Un livre d’introduction gratuit qui enseigne Python au niveau débutant. Il assume aucun expérience préalable en programmation.

A Byte of Python for Python 2.x A Byte of Python for Python 3.x

Apprendre à programmer en Python avec Codecademy (fr)

Un cours Codecademy pour le vrai débutant Python. Ce cours gratuit et interactif offre et enseigne les bases (et au-delà) de la programmation Python tout en testant les connaissances de l'utilisateur au fur et à mesure de la progression. Ce cours comprend également un interpréteur intégré pour avoir un retour immédiat sur votre apprentissage.

Apprendre à programmer en Python avec Codecademy (fr)

Intermédiaire

Effective Python

Ce livre contient 59 façons spécifiques pour améliorer l'écriture de code "pythonique". Avec 227 pages, c'est un très bref aperçu de quelques-unes des adaptations les plus communes que les programmeurs doivent faire pour devenir des programmeurs Python de niveau intermédiaire.

Effective Python

Avancé

Pro Python

Ce livre est pour les programmeurs Python, intermédiaires à avancés, qui cherchent à comprendre comment et pourquoi Python fonctionne comme il le fait et comment ils peuvent faire passer leur code au niveau supérieur.

Pro Python

Expert Python Programming

Expert Python Programming traite des meilleures pratiques en matière de programmation Python et se concentre sur le public plus avancé.

Il commence avec des sujets tels que les décorateurs (avec les études de cas pour la mise en cache, les proxy et les gestionnaires de contexte), l'ordre de résolution des méthodes (Method Resolution Order ou MRO), en utilisant super() et la méta-programmation, et les meilleures pratiques **PEP 8**.

Il a une étude de cas multi-chapitres détaillée sur l'écriture et la mise à disposition d'un paquet, éventuellement d'une application, incluant un chapitre sur l'utilisation de zc.buildout. Les chapitres ultérieurs détaillent les meilleures pratiques telles que l'écriture de la documentation, le développement piloté par les tests, la gestion de version, l'optimisation et le profilage.

Expert Python Programming

A Guide to Python's Magic Methods

This is a collection of blog posts by Rafe Kettler which explain 'magic methods' in Python. Magic methods are surrounded by double underscores (i.e. __init__) and can make classes and objects behave in different and magical ways.Ceci est une collection de billets de blog par Rafe Kettler qui expliquent les 'méthodes magiques' en Python. les méthodes magiques sont entourées par des underscores doubles (comme __init__) et peuvent faire que les classes et les objets se comportent de façons différentes et magiques.

A Guide to Python's Magic Methods

Pour les ingénieurs et les scientifiques

A Primer on Scientific Programming with Python

A Primer on Scientific Programming with Python, écrit par Hans Petter Langtangen, couvre principalement l'usage de Python dans le domaine scientifique. Dans le livre, les exemples sont choisis dans les scieneces mathématiques et naturelles.

A Primer on Scientific Programming with Python

Numerical Methods in Engineering with Python

Numerical Methods in Engineering with Python, écrit par Jaan Kiusalaas, met l'accent sur les méthodes numériques et la façon de les implémenter en Python.

Numerical Methods in Engineering with Python

Sujets divers

Problem Solving with Algorithms and Data Structures

Problem Solving with Algorithms and Data Structures couvre une étendue de structures de données et algorithmes. Tous les concepts sont illustrés avec du code Python accompagné d'exemples interactifs qui peuvent être exécutés directement dans le navigateur.

Problem Solving with Algorithms and Data Structures

Programming Collective Intelligence

Programming Collective Intelligence introduit un large éventail de méthodes de base sur le machine learning et le data mining. L'exposition est pas très formelle mathématiquement, mais met l'accent sur l'explication d'intuitions sous-jacentes et montre comment implémenter les algorithmes en Python.

Programming Collective Intelligence

Transforming Code into Beautiful, Idiomatic Python

Transforming Code into Beautiful, Idiomatic Python est une vidéo par Raymond Hettinger. Apprenez à mieux tirer parti des meilleures fonctionnalités de Python et à améliorer le code existant via une série de transformations de code, "Quand vous voyez ceci, faites cela à la place."

Transforming Code into Beautiful, Idiomatic Python

Fullstack Python

Fullstack Python offre une ressource complète couvrant toute la chaîne de production pour le développement web en utilisant Python.

De la mise en place du serveur web, à la conception du front-end, au choix d'une base de données, à l'optimisation/dimensionnement, etc.

Comme le nom le suggère, il couvre tout ce dont vont avez besoin pour construire et exécuter une application web en partant de zéro.

Fullstack Python

Références

Python in a Nutshell

Python in a Nutshell, écrit par Alex Martelli, couvre la plupart des usages Python multi-plateforme, de sa syntaxe pour les bibliothèques intégrées aux sujets avancés comme l'écriture d'extensions C.

Python in a Nutshell

La référence du langage Python

C'est le manuel de référence de Python officiel. Il couvre la syntaxe et la sémantique de base du langage.

La référence du langage Python (en)

Python Essential Reference

Python Essential Reference, écrit par David Beazley, est le guide de référence définitif pour Python. Il explique de manière concise tant le cœur du langage et les parties les plus essentielles de la bibliothèque standard. Il couvre les versions de Python 3 et 2.6.

Python Essential Reference

Python Pocket Reference

Python Pocket Reference, écrit par Mark Lutz, est une référence pour le cœur du langage facile à utiliser, avec les descriptions des modules et des boîtes à outils utilisés généralement. Il couvre les versions de Python 3 et 2.6.

Python Pocket Reference

Python Cookbook

Python Cookbook, écrit par David Beazley et Brian K. Jones, est associé à des exemples pratiques. Ce livre couvre le cœur du langage Python ainsi que des tâches communes à une grande variété de domaines d'application.

Python Cookbook

Writing Idiomatic Python (Écrire du Python idiomatique)

"Writing Idiomatic Python", écrit par Jeff Knupp, contient les idiomes Python les plus courants et les plus importants dans un format qui maximise l'identification et la compréhension. Chaque idiome est présenté comme une recommandation sur la façon d'écrire un morceau de code couramment utilisé, suivi d'une explication du pourquoi cet idiome est important. Il contient aussi deux exemples de code pour chaque idiome : la manière "Nocive" de l'écrire et la manière "Idiomatique".

Pour Python 2.7.3+

Pour Python 3.3+

Documentation

Documentation officielle

La documentation de la bibliothèque et du langage Python officiel peut être trouvée ici :

— Python 2.x
— Python 3.x

Read the Docs

Read the Docs est un projet communautaire populaire qui héberge la documentation de logiciels open source. Il contient la documentation de nombreux modules Python, à la fois ceux populaires comme ceux exotiques.

Read the Docs

pydoc

pydoc est un utilitaire qui est installé lorsque vous installez Python. Il vous permet de récupérer et rechercher rapidement de la documentation depuis votre terminal. Par exemple, si vous avez besoin d'un rapide rappel sur le module `time`, récupérer la documentation serait aussi simple que

```
$ pydoc time
```

La ligne de commande ci-dessus est essentiellement l'équivalent d'ouvrir la ligne de commande Python interactive et de l'exécuter

```
>>> help(time)
```

Actualités

Planet Python

C'est une agrégation d'actualités Python provenant d'un nombre grandissant de développeurs.

Planet Python

/r/python

/r/python est la communauté Reddit sur Python où les utilisateurs contribuent et votent sur les actualités liées à Python.

/r/python

Pycoder's Weekly

Pycoder's Weekly est une lettre d'actualité hebdomadaire gratuite sur Python pour des développeurs Python par des développeurs Python (projets, articles, actualités et emplois).

Pycoder's Weekly

Python Weekly

Python Weekly est une lettre d'actualité hebdomadaire gratuite sélectionnant des actualités, articles, nouvelles sorties logicielles, emplois, etc. liées à Python.

Python Weekly

Python News

Python News est la section actualités dans le site web officiel Python (www.python.org). Il met en avant brièvement les actualités de la communauté Python.

Python News

Import Python Weekly

Weekly Python Newsletter contient des articles, projets, vidéos, tweets sur Python, directement envoyés dans votre boite mail. Gardez vos compétences de programmation Python à jour.

Import Python Weekly Newsletter

Awesome Python Newsletter

Un aperçu hebdomadaire des actualités, articles et paquets Python les plus populaires.

Awesome Python Newsletter

Note : Les notes définies dans les échelles musicales diatoniques et chromatiques ont été intentionnellement exclues de la liste des notes additionnelles. D'où la présence de cette note.

Notes de contributions et informations légales (pour ceux intéressés)

Contribuer

Le guide de Python est en développement actif et les contributeurs sont bienvenus.

Si vous avez une demande de fonctionnalité, une suggestion, ou un rapport de bug, merci d'ouvrir un nouveau ticket sur GitHub. Pour soumettre des patches, merci de faire une "pull request" sur GitHub. Une fois vos changements intégrés, vous serez automatiquement ajouté à la liste des contributeurs.

Guide de style

Pour toutes les contributions, merci de suivre le *Le guide de style du guide*.

Liste de choses à faire ("Todo")

Si vous voulez contribuer, il y a plein de choses à faire. Voici un courte liste de todo.

— Établir les recommandations entre "utilisez ceci" contre "les alternatives sont...."

À faire
Ecrire à propos de Blueprint

(L'entrée originale se trouve dans /home/docs/checkouts/readthedocs.org/user_builds/python-guide-fr/checkouts/latest/docs/scenarios/admin.rst, à la ligne 369.)

À faire
Écrire à propos de Numba et du compilateur autojit pour NumPy

---

(L'entrée originale se trouve dans /home/docs/checkouts/readthedocs.org/user_builds/python-guide-fr/checkouts/latest/docs/scenarios/speed.rst, à la ligne 227.)

À faire
Compléter le stub "Geler votre code"

(L'entrée originale se trouve dans /home/docs/checkouts/readthedocs.org/user_builds/python-guide-fr/checkouts/latest/docs/shipping/freezing.rst, à la ligne 37.)

À faire
Écrire les étapes pour les exécutables les basiques

(L'entrée originale se trouve dans /home/docs/checkouts/readthedocs.org/user_builds/python-guide-fr/checkouts/latest/docs/shipping/freezing.rst, à la ligne 73.)

À faire
Inclure des exemples de code de code exemplaire de chacun des projets énumérés. Expliquer pourquoi c'est un excellent code. Utilisez des exemples complexes.

(L'entrée originale se trouve dans /home/docs/checkouts/readthedocs.org/user_builds/python-guide-fr/checkouts/latest/docs/writing/reading.rst, à la ligne 57.)

À faire
Expliquer les techniques pour identifier rapidement les structures de données, les algorithmes et déterminer ce que le code fait.

(L'entrée originale se trouve dans /home/docs/checkouts/readthedocs.org/user_builds/python-guide-fr/checkouts/latest/docs/writing/reading.rst, à la ligne 59.)

Licence

Le guide de style du guide

Comme avec toute documentation, avoir un format consistant aide à rendre le document plus compréhensible. Afin de rendre ce guide plus facile à assimiler, toutes les contributions doivent se tenir aux règles de ce guide de style, le cas échéant.

Le guide est écrit avec du *reStructuredText*.

Note : Des parties du guide peuvent ne pas respecter encore le guide de style. N'hésitez pas à mettre à jour ces parties pour les mettre en conformité avec le guide de style du guide.

Note : Sur n'importe quelle page du HTML rendu, vous pouvez cliquer "Afficher le code source de la page" pour voir comment les auteurs ont stylé la page.

Pertinence

Efforcez-vous de garder toutes les contributions relatives aux *but du guide*.

- Évitez d'inclure trop d'information sur des sujets qui ne sont pas directement liés au développement Python.
- Préférez faire un lien vers d'autres sources si l'information est déjà sortie. Assurez-vous de décrire quoi et pourquoi vous faites un lien.
- Citez les références quand c'est nécessaire.
- Si un sujet n'est pas directement pertinent pour Python, mais est utile en conjonction avec Python (c'est à dire Git, GitHub, les bases de données), référencez en faisant un lien vers des ressources utiles, et décrivez pourquoi c'est utile pour Python.
- En cas de doute, demandez.

Rubriques

Utilisez les styles suivants pour les rubriques.

Titre du chapitre :

```
#########
Chapter 1
#########
```

Titre de page :

```
====================
Time is an Illusion
====================
```

Rubriques de la section :

```
Lunchtime Doubly So
-------------------
```

Rubriques de la sous-section :

```
Very Deep
~~~~~~~~~
```

Prose

Faites des retours à la ligne à 78 caractères. Quand c'est nécessaire, les lignes peuvent dépasser 78 caractères, particulièrement si faire des retours à la ligne rend le texte source plus difficile à lire.

Utilisez la virgule de série (aussi connue comme la virgule d'Oxford) est 100% non-optionnel. Toute tentative pour soumettre un contenu avec une virgule manquante donnera lieu à un bannissement permanent de ce projet, pour cause d'absence complète et totale de goût.

Bannissement ? C'est une blague ? Nous espérons ne jamais avoir à le découvrir.

Exemples de code

Faites des retours à la ligne à 70 caractères pour tous les exemples de code, pour éviter des barres de défilement horizontales.

Exemples de ligne de commande :

```
.. code-block:: console

    $ run command --help
    $ ls ..
```

Assurez-vous d'inclure le préfixe `$` avant chaque ligne.

Exemples d'interpréteur Python :

```
Label the example::

.. code-block:: python

    >>> import this
```

Exemples d'interpréteur Python :

```
Descriptive title::

.. code-block:: python

    def get_answer():
        return 42
```

Liens externes

— Préférez des étiquettes pour les sujets biens connus (ex : des noms complets) quand vous faites des liens :

```
    Sphinx_ is used to document Python.

    .. _Sphinx: http://sphinx.pocoo.org
```

— Préférez utiliser des étiquettes descriptives pour les liens "inline" plutôt que de laisser des liens en brut :

```
    Read the `Sphinx Tutorial <http://sphinx.pocoo.org/tutorial.html>`_
```

— Evitez d'utiliser des étiquettes comme "cliquez ici", "ceci", etc. Préférez des étiquettes descriptives (notamment pour le référencement) à la place.

Liens vers des sections dans le guide

Pour faire des références croisées à d'autres parties de cette documentation, utilisez mot-clé et étiquettes :ref:.

Pour faire référence aux étiquettes plus claires et uniques, ajoutez toujours un suffixe‘‘-ref‘‘ :

```
.. _some-section-ref:

Some Section
------------
```

Notes et alertes

Utilisez les directives d'admonitions appropriées quand vous ajoutez des notes.

Notes :

```
.. note::
    The Hitchhiker's Guide to the Galaxy has a few things to say
    on the subject of towels. A towel, it says, is about the most
    massively useful thing an interstellar hitch hiker can have.
```

Alertes :

```
.. warning:: DON'T PANIC
```

TODOs

Merci de marquer tous les endroits incomplets du guide avec une directive todo. Pour éviter d'encombrer la *Liste de choses à faire ("Todo")*, utilisez un seul `todo` pour les bouts de code ou les sections incomplètes importantes.

```
.. todo::
    Learn the Ultimate Answer to the Ultimate Question
    of Life, The Universe, and Everything
```

P

V