développement logiciel

rgpd

intégration continue

logiciel libre

stella magazine

001

Édito

Écrire, c'est une manière pour nous de partager nos passions, nos envies et nos savoirs. Stella bouillonne d'idées à partager avec vous, sur des sujets qui nous tiennent à cœur : le développement logiciel, les données personnelles, l'intégration continue, le logiciel libre…

Au terme d'une année étrange qui aura vu nos corps se confiner, il nous paraît important de faire voyager les mots. C'est l'occasion bien sûr de partager avec le plus grand nombre, mais aussi d'essayer de nouvelles formes, de nouveaux supports. Avec ce magazine, nous avons voulu nous amuser avec la mise en page des mots, pour en quelque sorte compiler ce que nous avions à dire dans un endroit unique.

Nous espérons que vous prendrez plaisir à picorer les mots dans ce recueil. Dénichez les informations qui vous intéressent, osez aborder les sujets qui vous font réfléchir, et laissez-vous porter par les histoires plus légères ! Bien sûr, n'hésitez pas à nous contacter si l'envie vous en vient, nous serons heureux d'avoir avec vous une discussion passionnée sur les sujets qui vous tiennent à cœur.

Guillaume Ayoub
Lucie Anglade

Développement logiciel

L'enfer des paquets Python

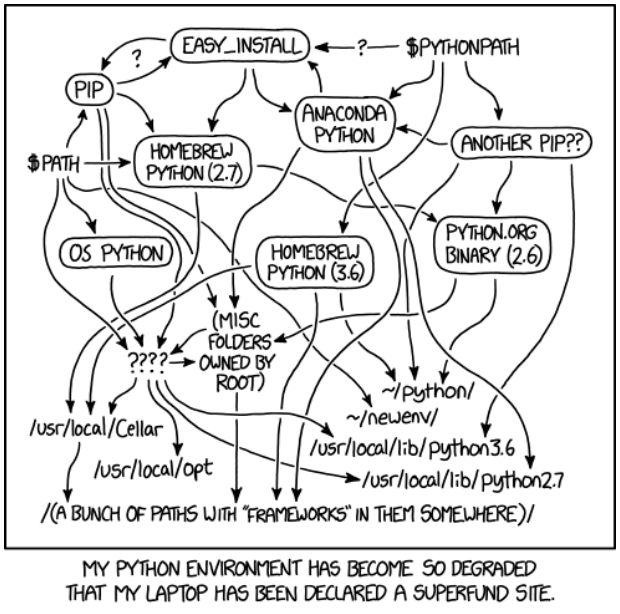
La gestion des paquets Python est parfois un enfer. Vous ne voyez pas de quoi je veux parler ? Je vous montre deux ou trois trucs, et on en reparle après…

C'est quoi, le problème ?

Si vous êtes là, je suppose que vous avez déjà fait un peu de Python. Vous avez déjà installé un projet Python, avec des dépendances. Et pour faire tout cela, il est possible que vous ayez eu à faire certaines choses. Beaucoup de choses. Beaucoup trop de choses. Pas tant que ça ? Je parie que vous avez déjà rencontré des trucs qui s'appellent `pip`, `pipenv`, `poetry`, `setuptools`, `distutils`, `requirements.txt`, `setup.py`, `Pipfile`, `setup.cfg`, `pyproject.toml`, `venv`, `virtualenv`, `wheel`… Je m'arrête là, mais vous en connaissez encore d'autres, sans vous en rendre compte.

C'est trop, et ce n'est pas normal.

Le XKCD obligatoire : suivez les flèches. Reste à savoir lesquelles.



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

Une phrase du Zen de Python dit à juste titre « There should be one– and preferably only one –obvious way to do it ». Vous le voyez, là, le problème ?

L'horreur des débuts

Vous ne vous rappelez peut-être pas, mais à un moment vous ne connaissiez pas Python. Si, si, je vous assure.

Tentez d'expliquer à quelqu'un à l'aise derrière un clavier (mais qui ne connaît pas Python) comment installer un programme Python. À sa place, en bon néophyte du XXI^{e} siècle, je m'attendrais à quelque chose comme :

1. Installe Python en suivant le super tuto super court de la super doc, ou à la limite avec la commande `download-python-and-install-it`.
2. Installe ton programme en cliquant ici, ou à la limite avec la commande `python-install mon-programme`.

Désolé de casser vos rêves : ça ne se passe pas comme ça du tout. Mais alors pas du tout. Le « super tuto super court de la super doc » pour installer Python n'existe pas. Et il existe beaucoup trop de programmes `python-install`, tous incompatibles entre eux, dont aucun n'est fourni avec Python, pour ce que soit sympa d'installer votre programme.

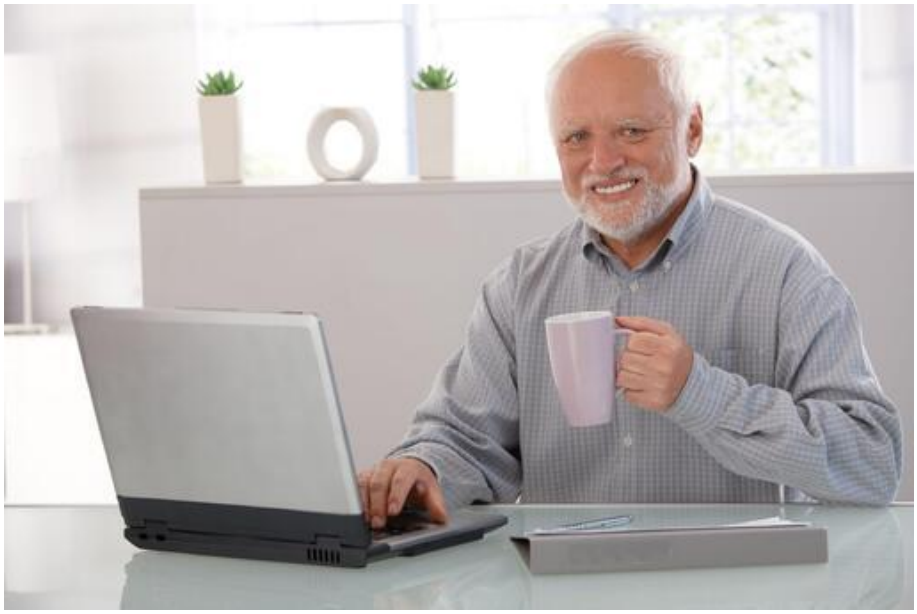
Ouais : 🤦. On est d'accord.

Paye ton tuto

Si on parle de tuto, la meilleure source est sans doute à trouver du côté de The Hitchhiker's Guide to Python. Il y a toujours beaucoup à dire sur la documentation, mais pour apporter une pointe d'optimisme : le guide est traduit en plusieurs langues, et je trouve qu'il est à certains moments vraiment bien fait et plutôt didactique.

Je ne parlerai pas du fait que ce n'est pas la documentation officielle. Ni du fait que tout ce guide est une promotion à peine déguisée à la gloire des projets de son discutable auteur. Non, n'insistez pas, je ne déverserai pas mon fiel.

« Alors tu vois, tu peux pas faire des f-strings avec ton virtualenv parce qu'il est en Python 2 »



Problème : ce tuto n'est ni court, ni simple. Vous êtes sur macOS ? Voici ce qui vous attend dès les premiers paragraphes :

> 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.

Je vous épargne l'enfer 🔥 de l'installation sous Windows, où il existe une liste démoniaque de solutions toutes plus ou moins bancales. Python est dans le Store (c'est ce que dit Microsoft), dans Chocolatey (c'est ce que dit le Hitchhiker's Guide en anglais, mais pas en français), sur `python.org` (c'est ce que dit `python.org`, mais pas sa documentation).

Quelle est la meilleure solution ? Je ne sais pas. En fait il n'y en a pas. Et c'est très mal.

Paye ton installeur

D'accord pour ce premier point un peu désastreux. Mais… On pourrait peut-être se rattraper sur le second point? Installer un paquet après avoir installé Python devrait être un jeu d'enfant. On fait `pip install mon-paquet`, non?

Non.

`sudo pip install mon-paquet` plutôt, non?

Non. Non. Non. Non. Non. Non. Non. Non.

Pour installer un paquet, il faut créer un environnement virtuel, mais en fait pas tout le temps. Pour créer un environnement virtuel, on va utiliser un module intégré dans Python mais en fait pas tout le temps. Et après on utilisera un programme intégré dans Python mais en fait pas tout le temps. Et après le paquet s'installera, mais en fait pas tout le temps.

Si vous ne voulez pas vous prendre la tête, vous pouvez suivre ce que dit le Hitchhiker's Guide et utiliser Pipenv (je vous laisse deviner qui a créé ce joyeux outil). Bon, bien sûr il faudra passer outre le fait que Pipenv a abandonné ses utilisateurs un an et demi avec une version cassée sans mise à jour. Il faudra fermer les yeux sur le fait que le projet est tentaculaire, englobe une quarantaine de projets dupliqués et parfois modifiés, et fait plus de 200 000 lignes de code 🙀…

Hahahahahahaha. Haha. Non.

La solitude de la création de paquets

Tout ça, c'est juste quand on installe un paquet. Si vous voulez créer et distribuer un paquet, armez-vous de patience, attendez-vous à d'autres surprises en coulisses.

Soyons francs : il existe un tuto et un guide pour apprendre à faire des paquets. Ces documents sont plutôt bien rédigés et devraient vous amener rapidement à pouvoir mettre en ligne votre bibliothèque. En théorie, on est pris par la main. En théorie.

À chaque fois que vous croyez avoir fermé la boîte à mauvaises nouvelles, ça s'ouvre sans prévenir.

En pratique, les choses sont beaucoup plus chaotiques. Si l'on prend seulement trois projets comme Flask, Requests et NumPy, on arrive à une liste assez importante de fichiers liés de près ou de loin à la gestion de paquets, sans compter les classiques `README`, `LICENSE` et consorts :

- `setup.py`,
- `setup.cfg`,
- `MANIFEST.in`,
- `Pipfile`,
- `Pipfile.lock`,
- `tox.ini`,
- `pytest.ini`,
- `.coveragerc`,
- `requirements.txt`,
- `pyproject.toml`…

La mauvaise nouvelle, c'est que Python ne fournit pas d'outil pour créer un projet prêt à être distribué. Vous allez soit vous perdre dans des projets tiers qui vont faire ça à leur manière (je te regarde encore de travers, Pipenv), soit vous allez apprendre à la dure à quoi servent ces fichiers bizarres.

(Petite note amusante : `setup.py`, `setup.cfg`, `requirements.txt` et `pyproject.toml` permettent tous les quatre, entre autres, de lister des dépendances d'un projet. Ils utilisent 4 formats de fichiers différents, dont l'un n'est même pas géré par la bibliothèque standard de Python.)

Les différents tutos et aides en ligne ne vous fourniront que des informations parcellaires et souvent obsolètes. Et pour cause : les bonnes pratiques changent constamment. Les outils ont souvent été développés sans spécifications, avec de la documentation au mieux lacunaire (je pense très fort à toi, `setuptools`). Le monde entier propose ses idées, de nouveaux outils fleurissent régulièrement, et tout prend vite l'allure de modes frénétiques qui tombent en désuétude au bout de quelques années (voire quelque mois si vous n'avez pas de chance)…

Je vous entends d'ici défendre votre langage chéri. « Oui, mais c'est compliqué, tout ça tout ça, bla bla bla bla… ». Vous voulez pleurer ? Allez sur la doc de Cargo, le gestionnaire de paquets de Rust. La première page est la page d'installation, elle contient 136 mots (dixit `wc`), commandes incluses. Elle permet d'installer Rust, avec Cargo. 136 mots, c'est un de moins que le Zen de Python. Pour de vrai.

La seconde page donne tout ce qu'il faut pour créer un paquet simple, avec un outil qui crée tous les fichiers. Elle contient le code du paquet, le fichier de métadonnées, les commandes lancées et leur résultat, un arbre des fichiers et dossiers en ASCII-art. Le tout en 183 mots.

Voilà. C'est possible. Vous savez maintenant.

Et après ?

Avec Python, vous allez apprendre des choses au hasard de vos lectures en ligne, sur des blogs douteux comme celui-là. Vous allez tomber sur des solutions StackOverflow totalement dépassées, des astuces non spécifiées exploitant des détails d'implémentation qui changeront la semaine prochaine. On inclut les tests dans le paquet, ou pas ? On les met dans le module, ou pas ? On distribue un paquet source, ou pas ? On indique les versions exactes des dépendances, ou pas ? On met la doc dans un dossier à part, ou pas ? Autant de questions, et bien d'autres, pour lesquelles vous êtes dans la profonde solitude de la création de paquets.

Vous vous dites sans doute que tout est perdu, que les gens qui ont développé tout ça sont incompétents, que Python est un langage finalement pourri et irrécupérable… Pourtant, non. Nous essaierons dans les prochains épisodes de comprendre pourquoi on en est arrivé là et combien les choses ont changé toutes ces années, souvent pour le mieux. Arriverons-nous un jour à construire un joli paquet ? Bien sûr.

La gestion des paquets Python est parfois un enfer. La faute à la stupidité des personnes derrière Python ? Et si c'était un peu plus compliqué que cela ?

Ce n'est pas la bonne techno

Nous avons vu qu'il était parfaitement possible de faire les choses bien concernant la création et la distribution de paquets, qu'il n'y avait qu'à regarder ce que faisaient par exemple les gens de Rust, et que ce n'était pas somme toute si compliqué.

Ce n'est pas totalement faux, mais soyons honnêtes, ce n'est pas totalement vrai non plus. Tout d'abord, Rust bénéficie d'atouts techniques indéniables par rapport à Python, avec par exemple la possibilité de distribuer des bibliothèques et des exécutables compilés. Python étant interprété, se pose la question de la distribution et de l'installation de l'interpréteur, avec tout son lot de

questions annexes (quel interpréteur, quelle version…). Certains outils comme Nuitka visent à apporter des embryons de solutions dans ce sens, mais ils sont condamnés à vivre leur existence en dehors de la bibliothèque standard, et donc probablement d'une utilisation massive.

Au-delà de l'aspect technique, le principal avantage de Rust est son âge. Créé en 2010, il est un jeune enfant comparé à l'antique Python né environ 20 ans plus tôt. Entre 1990 et 2010 sont apparus CVS, Subversion et Git ; Netscape, Internet Explorer, Firefox et Chrome ; HTML, CSS et JavaScript. C'est incroyable, mais c'est vrai, et ça met un peu en perspective la situation relative de Python et Rust.

Ce n'est pas le bon moment

On a du mal à se souvenir ou imaginer comment était l'informatique quand Python a commencé à germer à la fin des années 80, mais on arrive sans grande peine à comprendre pourquoi la création et la distribution de paquets n'étaient pas à la pointe des problèmes à prendre en compte.

Le bel écran de démarrage de Netscape 6 sorti en 2000, la même année que `distutils`.

Python n'a bien sûr pas intégré d'outils pour distribuer le code dès ses débuts. La Python Package Authority (PyPA) maintient un historique très instructif de l'évolution des paquets, où l'on apprend entre autres que :

- distutils a été intégré dans la bibliothèque standard en 2000, pour Python 1.6 ;
- PyPI a été mis en ligne en 2003 ;
- la PyPA a été créée en 2011 pour s'occuper de `pip` (né en 2008) et `virtualenv` (né en 2007) ;
- Python 3.3 a failli intégrer le remplaçant de `distutils` et `setuptools`, mais le projet a été abandonné faute d'investissement ;
- un nombre indécent de PEP ont été proposées et acceptées concernant l'évolution de ces outils.

Le plus frappant est peut-être l'impression d'amateurisme, en particulier à la naissance des premiers outils. Quiconque a déjà utilisé `easy_install` sait que la situation était alors extrêmement douloureuse, que l'installation d'un paquet nécessitait une dose non négligeable de persévérance, de connaissances techniques, et bien sûr de chance. L'accumulation de noms, d'outils, de bibliothèques internes et externes montre que tout le monde est parti bille en tête dans des solutions partielles, hasardeuses, voire carrément bancales.

Comme on peut l'entendre dans cet excellent épisode de Podcast.__init__, l'écriture des outils a longtemps été faite avant de soumettre à discussion et acceptation une spécification formelle. Par manque de temps, et par manque de moyens également : il n'y a pas d'entreprise derrière PyPI ou PyPA, comme il y en a par exemple derrière `npm`. La grande majorité des développements sont faits par des bénévoles, qui ne seraient pas contre un coup de main 🤚 (oui, c'est un appel du pied 🦶).

Certes, cet état des lieux n'explique pas tout. D'autres changements majeurs ont été intégrés dans le langage avec beaucoup plus de tact, en particulier récemment les coroutines. Cette fonctionnalité a nécessité une discussion large et houleuse avant une intégration plutôt appréciée. Avant de graver ces changements de syntaxe dans

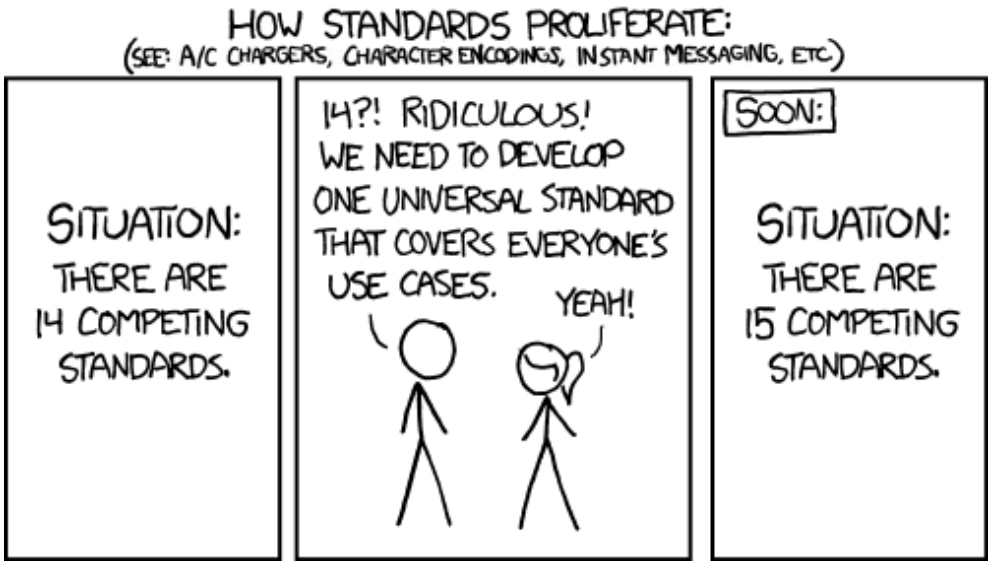
le marbre du langage, des propositions moins intrusives ont été testées et éprouvées, rendant la solution finalement retenue plus largement acceptable.

Il est facile de jeter ses pierres acérées sur les gens qui ont créé ces outils. Il est facile de vilement railler le manque de vision de l'équipe derrière Python, qui a intégré à la va-vite certaines solutions discutables et qui en a laissé d'autres aux portes de la bibliothèque standard. Mais si l'on pense que `distutils` a été intégré à un moment où le navigateur en vogue s'appelait Internet Explorer 5.5…

Et en plus, il n'est pas trop tard pour changer tout cela. Si ?

Ce n'est pas le bon standard

En quelque sorte, malheureusement, si, c'est trop tard.



Le XKCD obligatoire. Qui a la motivation pour créer un standard contre la prolifération des standards ?

Le problème est qu'il existe depuis le début des années 2000 des paquets dans la nature, sur PyPI, dans des dépôts publics et privés. Ces paquets sont utilisés par de nombreuses personnes, parfois sur des systèmes obsolètes (je te regarde, Python 2). Et Python, qui a déjà douloureusement éprouvé les changements brutaux et incompatibles (je te regarde, Python 3) ne voudra certainement pas rendre ininstallable cette masse informe et nébuleuse de paquets.

Sinon, ça va sortir les fourches et les guillotines.

Toutes les propositions, que ce soit dans les fichiers (`requirements.txt`, `setup.py`, `Pipfile`, `setup.cfg`, `pyproject.toml`…) ou dans les outils (`easy_install`, `pip`, `pipenv`, `poetry`, `setuptools`, `distutils`…) ne peuvent que s'ajouter les unes aux autres, sans jamais parfaitement les remplacer, sans jamais jeter aux oubliettes les défauts de leurs illustres prédécesseurs.

Alors évidemment, tout n'est pas perdu. Les changements les plus importants vont aujourd'hui globalement dans le sens d'une spécification, d'une rationalisation et d'une simplification systématiques. De nombreuses PEP sont proposées pour décrire et discuter avant de coder, pour que les détails d'implémentation ne fassent plus la loi au détriment d'idées nées de consensus.

Mais le chemin est long.

Ce n'est pas la bonne solution

Si vous pensez que la gestion de dépendances est un problème résolu de longue date et que les personnes derrière `pip` manquent sérieusement de compétences, rappelez-vous que le nom d'un paquet Python est souvent donné dans le fichier `setup.py`, et qu'il peut donc en théorie dépendre d'informations comme le système d'exploitation, la présence de modules externes ou même de l'heure qu'il est. Connaître les modules dont dépend un paquet nécessite parfois de lancer un interpréteur, et l'arbre de dépendances est donc différent pour chaque personne qui installe un paquet.

En pratique, des solutions ont été trouvées pour pallier cette flexibilité qui vire clairement au cauchemar dans certains cas. Il n'est bien heureusement pas nécessaire de télécharger et d'exécuter toutes les versions possibles de tous les paquets avant de déterminer celles qu'il faut installer. Mais tout cela se fait avec de nouveaux formats et de nouvelles métadonnées qui, par définition, ne font pas partie des paquets précédents.

Concrètement, cela signifie que pour les outils d'installation des paquets, la gestion des solutions obsolètes va encore durer de nombreuses et douloureuses années. Le rythme effréné des

nouvelles versions de `setuptools` et `pip` donnent une idée des améliorations constantes qui se trament sans que nous nous en rendions compte ; mais nous ne sommes pas prêts de sortir de l'historique démoniaque que nous devrons trimballer longtemps encore.

Cependant, pour les personnes qui créent des paquets, la solution s'améliore considérablement. Le sujet est relativement peu traité et il est compliqué de trouver une documentation de référence sur le sujet (même de la part de PyPA, surtout de la part de PyPA). C'est bien dommage, parce qu'il deviendrait presque facile et élégant ces derniers mois de faire des paquets Python.

Sans `setup.py` du tout, par exemple.

Vous aimeriez en savoir plus ? Il nous reste trois articles pour faire un tour de l'outillage existant, mieux définir ce que nous voulons faire, avant de finalement créer notre paquet de toute beauté.

La gestion des paquets Python est parfois un enfer. Mais au fait, c'est quoi, un paquet ? Évidemment, puisque rien n'est simple, puisque rien ne nous sera épargné, la non-réponse à cette question est : ça dépend…

Un paquet ?

Si vous êtes ici, il y a de grandes chances que vous ayez déjà installé un paquet Python dans votre vie. Vous avez peut-être utilisé un environnement virtuel, utilisé `pip` ou installé un module proposé par votre distribution. Quoi qu'il en soit, vous avez sans doute réussi, d'une manière ou d'une autre, à mettre des bibliothèques Python quelque part sur votre système.

Déjà, quand on parle d'un paquet Python, il faudrait être clair. Et ce n'est pas facile, parce qu'il n'existe pas un seul format de paquets, mais toute une ribambelle.

(Avertissement : nous n'avons pas vocation à faire une présentation exhaustive, et nous n'allons donc pas balayer l'intégralité de ce qui a existé et existe encore (un tout petit aperçu pour satisfaire votre

curiosité). Nous emprunterons également certains raccourcis qui prendront quelques libertés avec la vérité, histoire de laisser à l'article la chance d'être à peu près lisible et digeste.)

Ça donne envie, comme ça, un joli paquet Python. Mais dans la vraie vie…

Sources

La façon la plus simple de distribuer le code, c'est d'en faire une archive compressée et de l'envoyer telle quelle. C'est à peu près ce qui se passe lorsque l'on choisit de créer un paquet source pour un module en Python.

Aussi simple qu'elle soit, cette méthode nécessite tout de même un peu de configuration. On doit déterminer quels fichiers sont inclus dans l'archive, y compris les divers fichiers de métadonnées et de configuration d'outils tiers (Tox, Coverage, Pytest…).

Cette archive est donc relativement simple à créer, mais elle comporte un problème de taille : elle n'offre aucune structuration particulière, aucune organisation pour les métadonnées. Elle donne les fichiers et permet à l'installeur de les installer comme il le ferait avec un dossier contenant les sources sur votre disque dur.

Et alors, me direz-vous ? Eh bien, sans en avoir l'air, cette limitation est absolument terrible, à plusieurs titres.

La première chose, c'est la gestion de dépendances. Les paquets dépendent d'autres paquets, comme vous le savez sans doute. En n'ayant pas d'autres métadonnées, l'installeur est contraint d'aller lire, par exemple, le fichier `setup.py`. Et là, vous avez un aperçu du problème : s'il faut exécuter un fichier Python pour connaître les dépendances d'un paquet (et récursivement de ses dépendances !) vous allez passer à sacré temps à télécharger des paquets, exécuter

du code, résoudre des dépendances, télécharger, exécuter, résoudre, télécharger… Pour la faire court : la résolution de dépendances ne peut pas être résolue à partir d'un simple arbre statique, tous les nœuds sont dynamiques. Et si vous vous posez la question : oui, c'est l'enfer.

Ça ne s'arrête pas là. L'installation, elle aussi, nécessite l'exécution de code, puisqu'elle dépend du fichier `setup.py`. Pour du Python simple dans des paquets simples, ce n'est pas particulièrement gênant, mais c'est une autre histoire dès que l'on a besoin de compiler du C par exemple. L'utilisateur se retrouve obligé d'installer les outils nécessaires à l'installation pour tout ce qui est spécifique au système d'exploitation ou à la version de Python. C'est complexe, c'est long, ça peut vite être très rebutant pour les utilisateurs, et c'est une source infinie de bugs.

Malgré ses limitations, ce format restera sans doute nécessaire pendant longtemps. On peut, pour de nombreuses raisons, vouloir obtenir la source dans le format le plus simple possible. Catastrophique conséquence : les outils d'installation de paquets Python devront supporter ce format, et toute la mécanique complexe qu'il entraîne, pendant de nombreuses années encore.

Egg

Pour pallier ces manques, un format de paquets a été créé en 2004 : les eggs.

Outre la référence habituelle aux Monty Python, l'idée part d'une bonne intention : construire des informations additionnelles qui permettent aux dépendances d'un projet d'être vérifiées et satisfaites à l'exécution. À l'époque, ce format permet avant tout d'utiliser `easy_install` pour installer facilement un module et ses dépendances. Il permet d'inclure des modules prêts à l'emploi, avec des extensions C déjà compilées. Il permet également, par un système d'espaces de noms, de distribuer des plugins. Et surtout, il peut directement se mettre à un endroit accessible à Python, sans requérir d'installation supplémentaire.

Construit au-dessus de `setuptools`, il apporte un grand nombre d'évolutions qui semblent aujourd'hui évidentes : des dépendances avec des versions fixes, une série de métadonnées précalculées, un mécanisme d'accès aux métadonnées et aux fichiers annexes depuis le code du module…

Génial  .

Pourtant, les eggs sont rapidement dépassés par leur architecture aussi novatrice (c'est une blague, c'est totalement pompé sur les jars de Java) que totalement bancale (ça, par contre, ce n'est pas une blague). Utiliser un format d'archive où le code n'a pas à être décompressé rend très complexe la gestion des erreurs, avec des traces menant à des fichiers introuvables sur le disque. L'inclusion du bytecode rend les eggs potentiellement dépendants d'une version de Python, tout comme l'inclusion des fichiers compilés les rend profondément liés à une architecture.

Et surtout, surtout… Le format n'est pas spécifié, donnant à l'implémentation une importance démesurée. Les ajouts successifs à `setuptools`, parfois mal ou pas du tout documentés, en font un grand jeu de roulette russe où la rétrocompatibilité, la reproductibilité et la simplicité sont constamment, allègrement, impitoyablement négligés. Les façons de rendre les eggs utilisables par l'interpréteur se sont multipliées pour gérer des cas toujours plus complexes, jusqu'à arriver à un énorme tas de solutions à base de liens symboliques, de fichiers contenant des chemins absolus, d'archives temporairement décompressées et autres abominations que nous laisserons sagement dans leur boîte de Pandore.

Wheel

Après cette petite erreur de jeunesse, qui aura somme toute fait perdre la raison à toute personne dotée d'un cerveau normalement constitué, une PEP est apparue pour mettre tout le monde d'accord : la PEP 427 introduisant les wheels.

Les wheels sont, à leur création en 2012, une évolution salutaire des eggs, dans le sens où elles en reprennent les bonnes idées sans en garder les abominations. On tient enfin une belle spécification, qui

connaîtra son lot d'améliorations elles aussi spécifiées, et qui permettra à une bonne dose d'outils de fonctionner ensemble sans dépendre de détails d'implémentation.

L'idée géniale (j'exagère à peine) des wheels est d'inclure dans le nom de fichier un certain nombre d'informations simplifiant considérablement la gestion de dépendances. Le code à inclure est différent selon les versions de Python ? Les extensions sont compilées différemment selon le système d'exploitation ? On distribue alors différentes wheels pour une même version d'un module, et l'installeur fera tout seul son marché pour trouver la version qui correspond à l'utilisateur.

Les wheels ont aussi la bonne idée d'embarquer les métadonnées sous une forme spécifiée (par la PEP 345). Ce format permet de décrire de manière complexe des dépendances conditionnelles, des dépendances externes, les versions de Python supportées, et plus globalement tout ce qu'il faut pour gérer correctement la distribution du code, l'installation facile et la possibilité de construire un arbre de dépendances statique.

Aujourd'hui, tout le monde a intérêt à utiliser des wheels pour la distribution et l'installation de paquets. Si tout n'est pas parfait dans la gestion de paquets Python, les wheels sont aujourd'hui l'exemple le plus marquant de ce qui fonctionne très bien. Ils ont permis une transition en douceur jusqu'à arriver à une situation où les eggs ont très largement disparu, et où les paquets sources servent de moins en moins… grâce à tout un écosystème de logiciels créés autour de ce nouveau format et à une rétrocompatibilité astucieuse.

Parce que oui, les wheels sont formidables, mais tout ceci n'aurait pas été possible avec l'arrivée préalable de `pip`, en 2008, destiné à remplacer `easy_install`. Mais n'allons pas trop vite ! Nous aurons le temps d'y revenir un peu plus tard, dans nos prochains articles…

La gestion des paquets Python est parfois un enfer. Pour s'en convaincre, il suffit de se noyer quelques minutes dans la myriade de fichiers utilisables (et utilisés !) pour construire ou installer un paquet.

Mais pourquoi ?

On ne peut pas dire qu'il n'existe pas de guide pour créer des paquets en Python. Le problème, ce n'est pas le manque, c'est la profusion. Des guides, vous en trouverez partout, plus ou moins vieux, plus ou moins pratiques, plus ou moins utiles… Le plus dur n'est pas d'en trouver, c'est de tout lire et de piocher différentes informations dans chacun, jusqu'à vous faire votre propre conviction.

Vous croyiez peut-être trouver dans ces lignes un bon récapitulatif de ce qui existe, mais voici la triste nouvelle : vous n'avez ici qu'une source supplémentaire à laquelle vous référer si deux ou trois choses vous plaisent.

Ceci étant, ce n'est déjà pas si mal…

Quel est le rapport de cette introduction avec les fichiers ? Eh bien, c'est assez simple. On ne peut pas dire qu'il n'existe pas de fichier de configuration pour créer des paquets en Python. Le problème, ce n'est pas le manque, c'est la profusion. Des fichiers, vous en trouverez partout, plus ou moins vieux, plus ou moins pratiques, plus ou moins utiles… Le plus dur n'est pas d'en trouver, c'est de tout lire et de piocher différentes informations dans chacun, jusqu'à vous faire votre propre conviction.

(C'est bon, vous l'avez, le rapport ?)

Je ne vais pas vous refaire le coup de « faut les comprendre, les gens qui font Python, parce que Python c'est vieux, on ne peut pas tout changer d'un coup… ». C'est un peu vrai pour cet enfer de fichiers, mais c'est aussi un peu faux. L'exemple officiel proposé aujourd'hui par PyPA contient 4 fichiers qui servent pour la création de paquets (`setup.py`, `setup.cfg`, `MANIFEST.in` et `pyproject.toml`). Si l'on peut comprendre l'envie de couvrir un maximum de solutions possibles, on peut tout autant condamner l'impression de chaos intersidéral donnée à quelqu'un qui voudrait apprendre.

(Rappel : un projet minimal Rust contient un fichier `Cargo.toml` pour les métadonnées et un fichier `src/main.rs` pour le code du projet. De plus, ces deux petits fichiers sont créés automatiquement pour vous par la commande `cargo new`.)

S'il est vrai qu'il aurait été difficile de penser dès le début à tous les besoins d'un fichier de configuration, il est en revanche beaucoup plus discutable de dire que l'on doit vivre avec ce triste historique pour l'éternité. Contrairement à d'autres sujets, rien ne nous empêcherait de définir un nouveau standard de fichier de configuration. Et rien ne nous empêcherait de faire en sorte que ce nouveau standard permette de générer des paquets identiques à ceux existant. Nous ferions table rase du passé, de ses vieux fichiers et de ses vieux outils, pour n'utiliser qu'un fichier dans tous les cas. Le créateur du paquet s'adapterait à ces nouvelles règles, certes, mais rien ne changerait pour l'utilisateur final, ni pour les outils qu'il utiliserait.

T'en veux des fichiers ? Tu vas en avoir plein les mirettes !

Ce serait beau, n'est-ce pas ? C'est l'heure de la bonne nouvelle : figurez-vous que c'est déjà ce qui s'est passé. Sans blague.

Maintenant que vous avez très envie de connaître la suite (oui, c'est totalement sournois et totalement assumé), nous allons pouvoir nous infliger tout le cheminement de pensée pour arriver à la situation actuelle. Ce qui compte, c'est le chemin, pas la destination, non ?

Une liste pas si longue

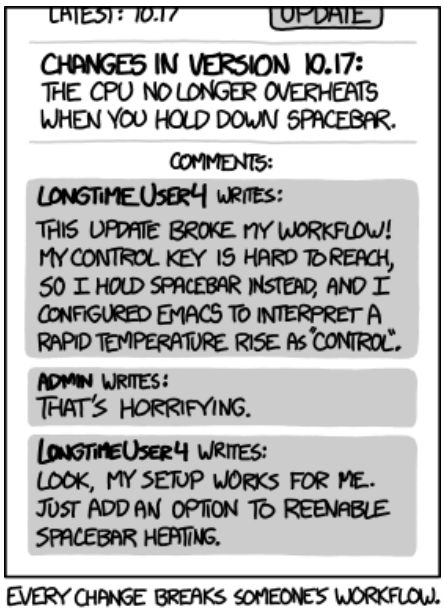
Ce n'est pas la peine de râler : comme à chaque fois, nous n'allons pas voir l'intégralité de ce qui a existé pour créer ou installer des paquets. Ne vous attendez pas à une liste exhaustive, juste à quelques fichiers emblématiques qui permettront de comprendre d'où l'on vient.

`setup.py`

Ce fichier est le premier fichier introduit pour gérer la création de paquets, c'est aussi le plus connu et le plus utilisé aujourd'hui, malgré son grand âge (au moins 20 ans, ça ne nous rajeunit pas).

L'idée derrière `setup.py` est relativement simple : pour mettre en place toute la configuration nécessaire à la création et à l'installation de paquets, on utilise un script Python qui définit un ensemble de métadonnées (le nom du paquet, la liste des fichiers à inclure, etc.) et différentes commandes (créer un paquet source, un paquet binaire, installer, etc). Pour faire cela, Python propose un module appelé `distutils`, qui contient tout ce qu'il faut pour décrire ces métadonnées et ces commandes. Il suffit de l'importer dans `setup.py`, d'appeler les bonnes fonctions, et le tour est joué.

Le XKCD obligatoire. De toute façon, qui utilise encore Emacs sur cette planète ?



Seulement, comme nous l'avons déjà vu plusieurs fois avec les outils gérant les paquets Python, `distutils` est assez limité et ses fonctionnalités ne sont pas strictement définies. Le code devient pernicieusement la référence de ce que l'on peut faire, et la peur (légitime) de tout casser empêche rapidement d'ajouter des fonctionnalités ou de corriger certains dysfonctionnements que d'aucuns auraient confondus avec des fonctionnalités.

Limité par `distutils`, `setup.py` aurait pu être remplacé par une autre solution. Mais on a trouvé mieux: `setuptools`.

`setuptools` est un module qui utilise `distutils` en interne, mais qui fournit des fonctionnalités supplémentaires, telles qu'une gestion plus poussée des fichiers à inclure, la possibilité de créer des exécutables Windows, et surtout… la possibilité de définir des dépendances.

Nous verrons les bibliothèques et les outils plus en détail dans le prochain article, mais il est important de comprendre que `setuptools` va ouvrir sans s'en rendre compte une boîte de Pandore. Puisque le module est externe à Python, il s'encombre beaucoup moins des pincettes de son prédécesseur. Les nouvelles fonctionnalités sont ajoutées au gré des besoins des utilisateurs, dans une joyeuse désorganisation qui a au moins eu le mérite de permettre une chaotique mais large diffusion des paquets Python. La bibliothèque vient avec un exécutable, `easy_install`, qui permet d'installer un paquet et ses dépendances. Elle vient également avec le format de paquets « egg » que nous avons abordé la dernière fois.

À partir du développement parfaitement anarchique de `setuptools`, il a été impossible de spécifier correctement les options et les bonnes pratiques de la création de paquets. `setup.py` a les inconvénients de ses avantages: étant écrit en Python, il permet d'utiliser toute la puissance du langage, pour ce qui devait à la base être quelques lignes de métadonnées et de scripts d'installation. Tout ce qui pourrait être simplement descriptif devient potentiellement dynamique à l'exécution. Des extensions sont proposées, dépendantes ou pas de `setuptools`, offrant une galaxie de possibilités. Les scripts grossissent, sont copiés de projet en projet

sans être compris. Des bouts de code corrigeant des dysfonctionnements pour différentes versions de Python, `distutils` ou `setuptools` sont inclus dans tous les `setup.py` de la Terre.

Et à la fin, on arrive à ça. Bien sûr, ce projet nécessite beaucoup de configuration et il serait difficile de faire tout ce que fait ce script en moins de code. Bien sûr, il est assez aisé de comprendre l'intégralité de ce fichier, par ailleurs assez joliment écrit, si l'on s'en donne la peine.

Non, le problème de `setup.py` n'est même pas sa complexité potentielle, qui peut dans de rares cas être tout de même utile. Le véritable problème, c'est qu'il n'y a eu pendant longtemps aucune alternative simple pour créer des paquets simples en pur Python. L'unique solution a été d'écrire du code, pour ce qui aurait souvent pu être totalement déclaratif. Et qui n'a pas été tenté d'écrire du code, plein de code horrible, même pour faire des choses simples ? Avec cette montagne de code horrible dans d'innombrables projets, `setuptools` a dû inclure des solutions de contournement permettant de contourner les solutions de contournement mises en place dans les scripts pour contourner des problèmes corrigés depuis. `setuptools` a dû copier et inclure différentes fonctions de différentes versions de Python (y compris leurs bugs, bien évidemment) pour être parfaitement rétrocompatible. Pour faire court : `setuptools` est devenu un monstre purulent qui a contaminé les `setup.py` d'une bonne majorité de projets.

`setup.cfg`

Évidemment, l'idée de mettre en place un format déclaratif pour la création de paquets est arrivée assez vite, et une solution a été intégrée à setuptools: `setup.cfg`.

Ce fichier INI n'est rien d'autre qu'une présentation différente de la plupart des options proposées en Python par `setuptools`. On va donc y retrouver les mêmes inconvénients : les mêmes bugs, les mêmes options peu ou pas documentées, les mêmes incohérences.

Surtout, ce fichier ne vient pas remplacer `setup.py`, mais il vient le compléter. On a besoin de garder le script, même presque vide ! Si des données sont en double, celles de `setup.cfg` sont conservées.

Pourquoi a-t-on besoin de garder le fichier `setup.py` ? Tout simplement parce que `setuptools` ne fournit pas de commande externe pour exécuter les commandes intégrées dans le script. Pour générer un paquet source, on utilise `python setup.py sdist` qui exécute directement le script.

Ce qui pourrait n'être qu'un détail se transforme en problème majeur. Qui voudrait utiliser un format statique, alors que l'on peut faire un gros tas de code spaghetti dans un script qu'il faut de toute manière garder ? Comment expliquer à celles et ceux qui découvrent le langage qu'il faut faire un fichier Python et un fichier INI, alors qu'on peut techniquement se passer du fichier INI ? Vous avez compris : on ne peut pas lutter contre l'appel du code.

Ceci explique pourquoi `setup.cfg` est relativement peu utilisé aujourd'hui. Attaché aux deux énormes boulets omniprésents que sont `setuptools` et `setup.py`, il n'apporte au fond qu'une petite dose de simplicité par son côté déclaratif. Tant qu'il transportera avec lui tout l'attirail d'un historique pesant et sclérosant, il restera un choix de seconde zone, une tentative un peu maladroite de résoudre un problème réel.

`requirements.txt`

Voilà un fichier que vous avez sans doute déjà croisé et déjà utilisé. Vanté sans finesse par les tutoriels de seconde zone, loué pour sa simplicité et sa puissance, utilisé par bon nombre de projets renommés, `requirements.txt` est la star de l'installation de dépendances.

Oui mais voilà, disons-le de but en blanc : il n'a rien à voir avec la création de paquets.

`requirements.txt`, c'est une simple liste de paquets à installer, avec la possibilité d'en déterminer les versions, les sources, les branches et les options d'installation.

Il s'utilise souvent avec `pip` et ne sert que pour l'installation. On peut le voir comme une façon pratique de lister des dépendances, dans un format que l'on pourrait passer directement en ligne de commande mais que la flemme et le goût pour les sauts de ligne nous poussent à confiner dans un fichier.

C'est pratique, en particulier pour ce qu'on souhaite partager sous une forme différente de celle d'un paquet. Au hasard : tout sauf les bibliothèques. Un petit script sans prétention ? Un `requirements.txt`. Une application web ? Un `requirements.txt`. Une bibliothèque ? Bon, d'accord, quand même des `requirements.txt` pour la documentation et les tests.

Oui, on peut avoir un `setup.py`, un `setup.cfg` et un `requirements.txt` dans le même projet. Avec tous leurs amis `MANIFEST.in`, `tox.ini`, `pyproject.toml`, `pytest.ini`, et je vous en passe quelques uns. Tout le monde fait sa petite sauce au petit bonheur la chance, en repompant allègrement sur ses petits camarades des trucs qui ont l'air de vaguement fonctionner. On trouvera toujours un cas particulier qui n'est géré qu'avec l'un de ces fichiers, et on sacrifiera la simplicité sur l'autel de la sacro-sainte fonctionnalité.

`MANIFEST.in`

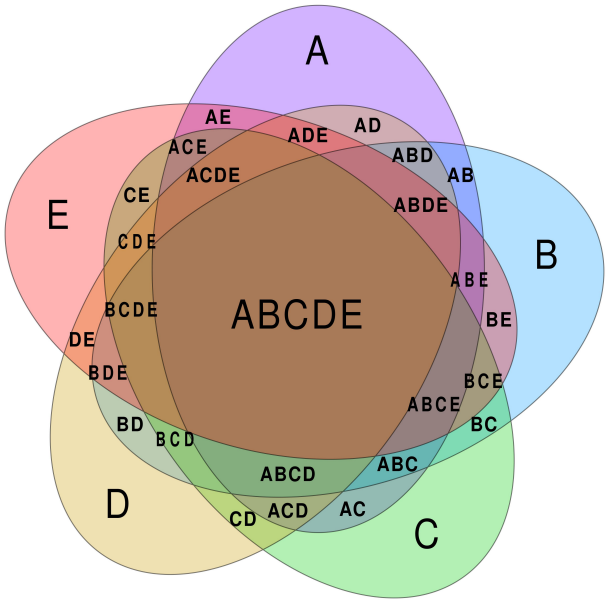
Vous voulez une fonctionnalité bien particulière ? L'inclusion de fichiers annexes dans un paquet source est un bon exemple de casse-tête.

Généralement, quand on distribue un paquet binaire, on le fait pour que les utilisateurs puissent facilement utiliser le code. Les paquets comme les wheels sont des archives prêtes à l'emploi, dont l'installation ne nécessite guère plus qu'une décompression dans le dossier qui va bien. Ces paquets peuvent ne contenir que le strict minimum : le code. Tout ce qui est annexe (la documentation, les tests, les petits-fichiers-super-trognons permettant de décrire les changements…) n'a rien à faire dedans.

C'est différent pour les paquets sources. Ces paquets servent à beaucoup de gens de faire beaucoup de choses : regarder le code, créer des paquets pour les distributions, tester des patchs, installer la

bibliothèque, lancer des tests… Alors on tente d'inclure le maximum de choses dans le paquet, presque tout ce qu'on a dans le dépôt, sauf ce qui concerne l'intégration continue, la configuration du gestionnaire de versions, et autres broutilles qui viennent également polluer notre joli projet.

Pour intégrer des fichiers dans le paquet source, en particulier quand ces fichiers sont à la racine du projet et pas dans le même dossier que le code, on utilise `MANIFEST.in`. Cet énième fichier vient, comme il se doit, avec sa propre syntaxe et ses propres commandes. Et n'ayez crainte : il permet à la fois de faire certaines choses que les autres fichiers permettent et certaines choses que les autres fichiers ne permettent pas.



Voyons voir… Avec quels fichiers je peux déterminer une dépendance optionnelle dynamique qui ne s'installera qu'avec la version 3.7.x de Python sur un Windows 32 bits ?

`pyproject.toml`

On y arrive.

Au premier abord, `pyproject.toml` semble tout droit être un clone de `setup.cfg`, avec un format légèrement différent et un nom discutable. Encore un autre fichier, encore un autre format, mais quelle idée saugrenue ?

En réalité, les choses sont un peu plus complexes. La PEP 518 qui introduit ce fichier s'appelle, si on tente de traduire son titre, « Spécification des dépendances minimales du système de

construction des projets Python ». Ce n'est pas « Encore un nouveau format stupide pour définir les métadonnées de mon paquet » et il y a de bonnes raisons à cela.

Dans la liste des problèmes causés par `setuptools`, en voilà un qui n'a pas encore été abordé : `setup.py` contient les dépendances du paquet, dont les dépendances utilisées pour construire le paquet. Comment faire pour connaître ces dépendances sans exécuter le fichier ? Et comment exécuter le fichier sans connaître les dépendances ? Ce problème de la poule et de l'œuf est problématique pour `setuptools`, mais puisque tout le monde l'utilise pour faire les paquets et qu'il est en dépendance de `pip`, il y a de grandes chances qu'il soit déjà installé avec Python. Par contre, si l'on souhaite utiliser un autre outil, comme une extension à `setuptools`, les choses deviennent tout de suite moins faciles.

L'idée de `pyproject.toml` n'est pas de proposer un nouveau format de métadonnées. L'idée est d'inclure, dans un fichier texte simple, les dépendances nécessaires pour construire un paquet. Réfléchissez bien à cela. Encore un peu.

Voilà. Vous avez compris. On va pouvoir se débarrasser de `setuptools` et `distutils`, au moins pour construire des paquets. Pour de bon.

Bien sûr, dans les cas simples, on peut continuer de les utiliser. `pyproject.toml` permet de stocker toutes les métadonnées que l'on stockait auparavant. Il permet également de stocker les informations plus complexes, telles que les dépendances et les versions de Python supportées, un peu comme dans `setup.cfg`, un peu comme avant.

Mais rien n'empêche d'utiliser un autre outil, qui peut définir lui-même ses options de configuration, indépendantes de celles de `setuptools`. Mieux : le fichier étant spécifié et bien construit, il laisse la place à tous les autres outils annexes (`black`, `pylint`, `coverage`…) d'utiliser eux aussi ce fichier. Et de mettre fin à l'atroce ensemble de confettis de fichiers de configuration.

Reste une chose à régler : définir le point d'entrée de l'outil que l'on va utiliser pour créer le paquet. C'est le rôle de la PEP 517 qui nous permet de nous affranchir totalement de `setuptools`, de `setup.py` et de tous leurs amis.

Mais… Ça marche vraiment ?

Oui. Il ne nous reste qu'à voir quels outils utiliser.

La gestion des paquets Python est parfois un enfer. Pour créer, envoyer, installer des paquets, nous avons à disposition beaucoup d'outils, qui font parfois la même chose, mais pas de la même manière.

Des outils pour tout faire

Des bibliothèques, des scripts, des exécutables… Même en tant que simple utilisateur, il faut connaitre et tester une bonne dose d'outils avant d'utiliser un programme Python. Vous allez devoir créer un environnement virtuel pour installer des paquets. Avec le temps, vous aurez vos petites habitudes, qui changeront au gré des évolutions et des bonnes pratiques.

Nous n'allons pas vraiment parler de cela. Mais un peu quand même.

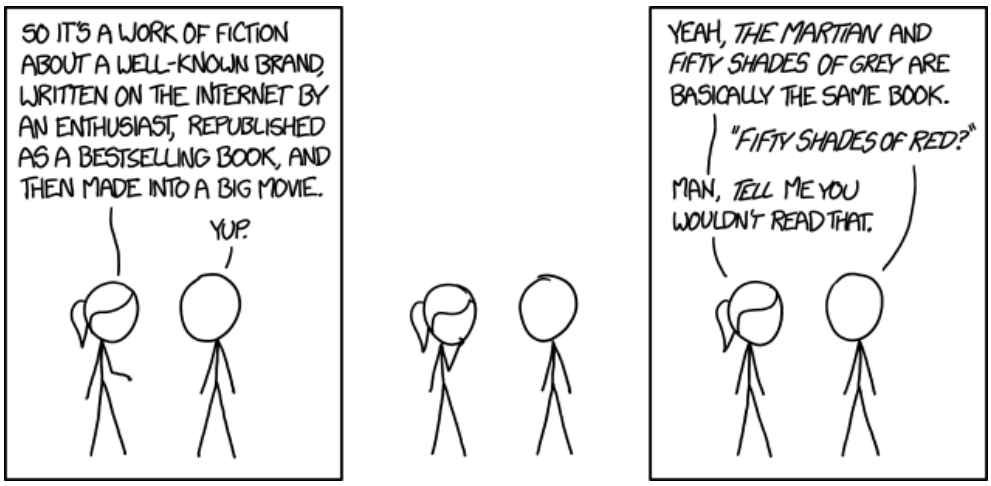
Si l'on se concentre sur la gestion de paquets, nous pouvons déterminer trois opérations spécifiques : l'installation, la création, l'envoi sur un dépôt de paquets. Chacune de ces opérations peut être découpée en de plus petites étapes, et ce serait intéressant de comprendre dans le détail comment tout cela fonctionne.

Nous n'allons pas parler de cela en détail non plus.

Ce que nous allons tenter de faire : dresser un tableau partial, superficiel, approximatif et non-exhaustif de ce que nous pouvons utiliser pour faire ces trois opérations de base. Ça n'envoie pas du rêve mais c'est déjà, mine de rien, une tâche complexe.

Nous aurons forcément à mettre un pied dans le détail de fonctionnement des opérations, nous aurons forcément à mettre un pied dans les environnements virtuels. Mais nous tâcherons de ne pas nous perdre dans des détails, sinon nous en aurions pour – au bas mot – des heures. Et vous avez mieux à faire.

Le XKCD obligatoire. Prendre quelques points de comparaison n'a jamais été suffisant. N'essayez jamais ça chez vous.



Commençons donc par deux tableaux. Le premier liste les bibliothèques, utilisables par d'autres outils.

Bibliothèque	Installation	Création	Envoi
distutils	Oui	Oui	Non
setuptools	Oui, basé sur distutils	Oui, basé sur distutils	Oui, jusqu'à la version 42

Le second tableau liste les outils, qui proposent des exécutables.

Outil	Installation	Création	Envoi
easy_install	Oui, distribué avec setuptools	Non	Non
pip	Oui, inclut une copie partielle de setuptools	Oui, des wheels avec setuptools et wheel	Non
wheel	Non	Oui, des wheels	Non
twine	Non	Non	Oui
pipenv	Oui, inclut une copie modifiée de pip	Non	Non
pipx	Oui, basé sur pip	Non	Non
poetry	Oui, basé sur pip	Oui	Oui
flit	Oui, basé sur pip	Oui	Oui

Ces deux tableaux ne listent que les fonctionnalités qui nous intéressent, mais certains outils font bien d'autres choses. Ce n'est donc pas la peine de les comparer bêtement selon le nombre de cases qu'ils cochent, d'autant plus que d'autres paramètres doivent rentrer en compte dans cette comparaison, y compris des petits détails tels que la qualité et la maintenabilité du code.

Et d'ailleurs, nous n'allons pas à proprement parler comparer ces outils ; nous allons plutôt présenter les trois fonctionnalités et décrire les bibliothèques et outils qui les accomplissent. Ce n'est pas la peine de bouder, je vous assure que ce sera plus simple ainsi.

Installation de paquets

Au début, lorsque la notion de paquets a été introduite dans Python en 2000, c'est `distutils` qui a été chargé de créer et installer les paquets. Nous l'avons déjà dit mais nous allons le répéter : il l'a fait sans gestion de dépendances et sans dépôt de paquets.

`distutils` est une bibliothèque qui permet avant tout d'écrire des fichiers `setup.py`, qui ont longtemps été la base des paquets Python. Grâce à `distutils` qu'ils importent, ces fichiers sont exécutables et offrent deux commandes : `install` pour installer, `sdist` pour créer le paquet. On peut dès lors partager des archives qui contiennent l'ensemble du code, et les installer après les avoir décompressées. En d'autres termes : ces archives sont des paquets.

L'idée d'avoir un endroit où stocker et partager ces paquets arrive vite. PyPI est mis en ligne trois ans après l'arrivée de `distutils` ; il permet de partager et de retrouver, à un endroit public central, un grand nombre de paquets Python.

La bibliothèque `setuptools` arrive en 2004 pour apporter de nouvelles fonctionnalités, en particulier la gestion de dépendances. Au niveau de l'installation de paquets, c'est la révolution : `easy_install` est inclus dans la bibliothèque et permet d'installer les paquets de PyPI directement avec leur nom.

Cependant, `setuptools` et `easy_install` vont eux aussi commencer à montrer leurs limites. Créés sans véritable spécification, basés sur un format de paquets imparfait (les eggs), ils vont attiser les volontés de remplacement.

Ce sera le cas pour `easy_install` qui sera remplacé 4 ans plus tard par `pip`. `pip` a pour but premier d'installer des paquets Python en gérant correctement leurs métadonnées, ce qui permet entre autres de lister et désinstaller les paquets installés.

`pip` a beaucoup évolué et est aujourd'hui l'application de référence concernant l'installation de paquets. L'outil a su s'adapter aux différentes évolutions et est aujourd'hui capable de se débrouiller avec les paquets sources et les wheels. Son architecture réfléchie et sa large utilisation lui ont permis d'intégrer des évolutions étape par étape, et de rester vivant depuis plus de 10 ans.

L'outil est utilisé ou inclus dans tous les outils récents d'installation de paquets. Certains, qui sont largement utilisés comme `Pipenv`, `Poetry` et `pipx`, lancent `pip` pour l'installation de paquets.

Alors, pourquoi utiliser d'autres outils que `pip` ? `Pipenv`, `Poetry`, `pipx` et d'autres permettent, chacun à leur manière, de cloisonner les installations. Par défaut, `pip` installe les paquets dans un dossier central, ce qui est assez gênant lorsque l'on veut avoir différents projets qui utilisent différentes versions de la même bibliothèque.

`Pipenv` et `Poetry` offrent à peu près la même mécanique pour l'installation : ils créent un environnement virtuel par projet, dans lequel ils installent les dépendances. Pour cet usage, ils se comportent en réalité comme une simple capsule autour de `pip` et `venv`, avec des commandes permettant de gérer les cas simples.

`pipx` a un but différent : il propose la même interface que `pip` pour installer des exécutables. Il s'arrange automatiquement pour créer un environnement virtuel par exécutable, et pour rendre cet exécutable accessible à l'utilisateur. Il est donc plus adapté aux utilisateurs finaux qu'au développeurs, cibles privilégiées de `Pipenv` et `Poetry`.

Une dernière chose étonnante au sujet de `pip` : il crée désormais des paquets, pour pouvoir mieux les installer. Mais oui, mais oui.

Création de paquets

Oui, vous avez bien lu : `pip` crée désormais des paquets. Avec la volonté sous-jacente de se séparer de `setuptools` et des autres formats de paquets, `pip` se transforme doucement en un simple installeur de wheels. Lorsqu'il a devant lui un paquet source, il tente désormais de transformer ce paquet source en wheel avant de l'installer, au lieu de passer par l'installeur de `setuptools`.

Rappel : ce n'est pas parce que l'emballage est joli que le contenu va nécessairement plaire.

Ce système comporte de nombreux avantages. Tout d'abord, cela signifie qu'à terme `pip` pourrait n'être qu'un installeur de wheels (qui est le format le plus simple à installer) couplé à un transformateur de paquets source en wheels. Un tel fonctionnement simplifierait grandement le code source de `pip`, qui aujourd'hui fait beaucoup d'autres choses, dont installer des paquets depuis les sources en utilisant `setuptools`.

Il faut également souligner le fait que, pour créer des paquets, il n'est désormais plus nécessaire d'utiliser `setuptools`. D'autres outils existent pour générer un paquet source ou un paquet wheel. Cela signifie que, de la création à l'installation, on commence à entrevoir le bout du tunnel : il est possible d'utiliser des outils relativement simples, basés en grandes partie sur des spécifications, qui n'ont pas à gérer l'ancienneté que trimballe `setuptools` avec lui.

`Poetry` et `Flit` sont sur ce point-là assez proches. Les deux sont capables de créer des paquets sans `setuptools` et donc sans fichier `setup.py`. Suivant les PEP 517 et 518, utilisant le fichier `pyproject.toml` comme unique source d'information, ils proposent une solution alternative pour créer des paquets source et des wheels classiques, installables par `pip`.

`Flit` ne fait à peu près que cela. Il contient également de quoi installer un paquet avec `pip` et par liens symboliques, ce qui est utile pour le développement. `Poetry` est beaucoup plus complet : il propose comme `Pipenv` la création d'environnements virtuels utiles pour le développement.

Bien sûr, le passage d'un fichier `setup.py` en Python à un fichier `pyproject.toml` limite les possibilités. Malgré les efforts de ces outils pour proposer une grande flexibilité, il n'est pas possible de faire tout ce que l'on pouvait faire avec `setuptools`, laissant à l'honorable bibliothèque le soin de gérer les cas complexes dont la plupart ne sont que le résultats d'esprits malades relevant plus de la psychiatrie que de l'informatique.

Envoi de paquets

De la même manière, `Poetry` et `Flit` proposent tous deux l'envoi de paquets sur PyPI ou sur des serveurs compatibles. Cette fonctionnalité ne nécessite guère que de suivre les APIs HTTP proposées par PyPI, et peut sembler assez simple au premier abord.

Cela n'a cependant pas toujours été le cas. `setuptools` a longtemps proposé une commande pour envoyer les paquets, non sans problèmes. Afin d'assurer une compatibilité avec toutes les versions de Python, il a fallu jongler avec les versions de protocoles TLS supportées, les failles de sécurité, les mots de passe… Et bien sûr, ce qui devait rester assez simple s'est vite transformé en triste cauchemar de soupe de code indigeste.

Pour régler ce problème, le projet `Twine` a été développé. `Twine` a comme unique but d'envoyer les paquets sur PyPI, et de le faire bien. Il propose en bonus la possibilité de stocker le mot de passe dans le gestionnaire de mots de passe du système, plutôt qu'en clair dans un

fichier texte comme le proposait `setuptools`. Autre détail : `Twine` envoie tels quels les fichiers générés par l'outil de créations de paquets. Cela peut paraître comme la moindre des choses, mais il faut savoir que `setuptools` recrée le paquet avant de l'envoyer, compliquant grandement les tests.

On résume (enfin, on essaie)

Il y eut une longue période de dépendance à `setuptools`, ses incompréhensibles fichiers de configuration, son implémentation qui fait foi, son architecture vieillissante et ses commandes discutables. Mais cette époque est bientôt révolue, et d'autres solutions existent d'ores et déjà pour créer et envoyer ses paquets.

Nous sommes dans une période d'incertitudes. Il est difficile, voire impossible, de savoir quels outils seront utilisés demain. Il est difficile de construire des paquets avec une architecture qui résistera à l'épreuve du temps. Mais une chose est sûre : nous avons plus de libertés et de possibilités que jamais.

Après tout, avoir de nombreux outils pour créer les paquets n'est pas un mal en soi, tant qu'ils créent tous des paquets interopérables, installables par les mêmes outils. On n'a pas les mêmes besoins lorsqu'on crée un tout petit paquet en Python pur dans son coin, ou un paquet contenant du C destiné à toutes les plateformes.

À ce sujet `Flit` propose une vision intéressante :

> « Rendre faciles les choses faciles, et possibles les choses difficiles », telle est une vieille devise de la communauté Perl. Flit se concentre uniquement sur la partie « choses faciles », et laisse les choses difficiles aux autres outils.

(On en arrive à s'inspirer de la communauté Perl, tout est possible…)

Tout dépend de ce que l'on veut faire.

La gestion des paquets Python est parfois un enfer. C'est d'autant plus un enfer que quand on parle de création ou d'installation d'un paquet, on devrait déjà commencer par définir précisément ce qu'on entend par là.

Papa, Maman : comment on fait les paquets ?

C'est bien beau de parler de paquets en long et en large, mais on en a parlé souvent de travers. À vrai dire, on a beaucoup parlé de ce qu'il y avait dedans, de ce qu'on pouvait en faire, mais pas dans le détail de comment les différentes étapes de sa vie, dont sa naissance, se déroulaient.

L'idée n'est pas de faire un exposé technique du fonctionnement de `setuptools` (vous pouvez me remercier). L'idée est plutôt de prendre conscience que la notion de paquet regroupe des réalités très diverses, et qu'on ne devrait pas utiliser les mêmes outils et les mêmes protocoles selon la réalité à laquelle on est confrontée.

Le XKCD obligatoire. Ces gens qui font des paquets Python sont tous incompétents, donnez-moi 5 minutes et je vous résous le problème une bonne fois pour toutes.

Vous avez toujours pensé que créer un paquet, c'était juste mettre les fichiers d'un dossier dans une archive ? Désolé de casser l'ambiance, mais ce n'est pas ça, non. Ce n'est pas ça du tout. Sinon, je ne serais pas là pour en parler, et vous ne seriez pas là pour lire cet article.

Le théâtre des opérations

On y est. Vous avez fini votre code, vous voulez le diffuser, et pour cela vous voulez créer un paquet. Vous avez lu d'innombrables tutoriels qui vous ont vanté d'innombrables techniques infaillibles, d'innombrables outils formidables, d'innombrables fichiers magiques.

Je vais vous donner mon avis (vous êtes là pour ça, non ?). La bonne question à vous poser, c'est : « quelles opérations devront être effectuées pour créer et installer mon paquet ? ». À partir de là, vous pourrez piocher dans votre bibliothèque le tutoriel le plus adapté à votre situation.

Vous lancer tête baissée dans la création de votre paquet, c'est prendre un double risque : vous prendre la tête à en faire trop, ou vous prendre la tête à ne pas pouvoir en faire assez. Croyez-moi sur parole : vous ne voulez ni l'un, ni l'autre. La vie est trop courte.

Alors, qu'est-ce qui se passe dans nos paquets ?

Le code du module

Évidemment, si vous voulez distribuer un paquet Python, il y a d'immenses chances que vous ayez du code Python à inclure dans votre paquet. Et quand on parle de code, le plus souvent on parle d'un module.

Un module Python, dans son expression la plus simple, c'est un simple fichier, ou un dossier qui contient plusieurs fichiers, et potentiellement des sous-dossiers qui composent autant de sous-modules.

Si l'on met de côté l'installation de scripts exécutables (nous en parlerons plus tard), cette étape est sans surprise particulièrement bien gérée par les différents outils disponibles. Que vous souhaitiez spécifier un dossier et inclure automatiquement le code à l'intérieur, que vous souhaitiez spécifier à la main la liste des fichiers et dossiers à inclure, que vous ayez même plusieurs modules à inclure dans votre paquet, cette étape ne devrait pas poser de problème particulier.

Si vous cherchez vraiment la complication (ça vous a traversé l'esprit, on ne va pas se mentir), vous aurez tout de même la tentation de n'inclure certains fichiers que pour certaines plateformes : un fichier pour Windows, un fichier pour les systèmes l'exploitation normaux. Vous voudrez appliquer quelques modifications au code pour une version de Python, ou selon la présence de certaines dépendances. N'ayez crainte, votre vice maladif ne sera pas mis sous le tapis : nous verrons cela plus tard, lors des opérations à la création et l'installation.

Les métadonnées

Avec votre paquet, peut-être même sans vous en rendre compte, vous voulez trimballer une ribambelle de métadonnées. Trois fois rien, vraiment. Un nom de paquet, un numéro de version, une adresse mail de contact. Et les version de Python supportées, les dépendances, obligatoires et optionnelles. Et une description, des mots-clés, quelques classificateurs, quelques liens normés à mettre sur PyPI vers la documentation et le dépôt source. Et des options spécifiques pour lancer les tests ou pour construire l'aide. Et…

OK. Pas trois fois rien.

Pour notre santé mentale, la plupart de ces métadonnées sont normées et facilement intégrables. Tant que l'on veut intégrer ce qu'il est prévu d'intégrer. Soyez raisonnable. Pas de vagues. Tranquille.

Vous la sentez venir, l'embrouille ?

Vous pourriez avoir la tentation d'intégrer des métadonnées dans des fichiers. Pas des fichiers de configuration, pas des fichiers à intégrer dans le module, des fichiers à mettre à côté. Un `README`, un `CHANGELOG`, des trucs comme ça. Un code de conduite, une liste des contributrices et contributeurs, une feuille de route…

Et tout cela, uniquement pour le paquet source. Pour la wheel, on ne veut pas ces fichiers annexes. Évidemment. Cela va de soi.

Rassurez-vous, vous pouvez. Un petit détail, cependant : par défaut, le fichier de configuration qui permet de faire cela est basé sur un pseudo-code comprenant 8 commandes différentes et une syntaxe spécifique d'expressions régulières. Ce fichier ne sert qu'à définir la

liste des fichiers à inclure dans le paquet source, il a un nom EN MAJUSCULES, un point, et une extension de deux lettres en minuscules (on ne citera pas son nom). Il a des règles implicites qui dépendent de la version de `setuptools`, il s'adapte aux dépôts CVS et Subversion (!), il crée automatiquement et obligatoirement des fichiers que l'on ne peut pas changer.

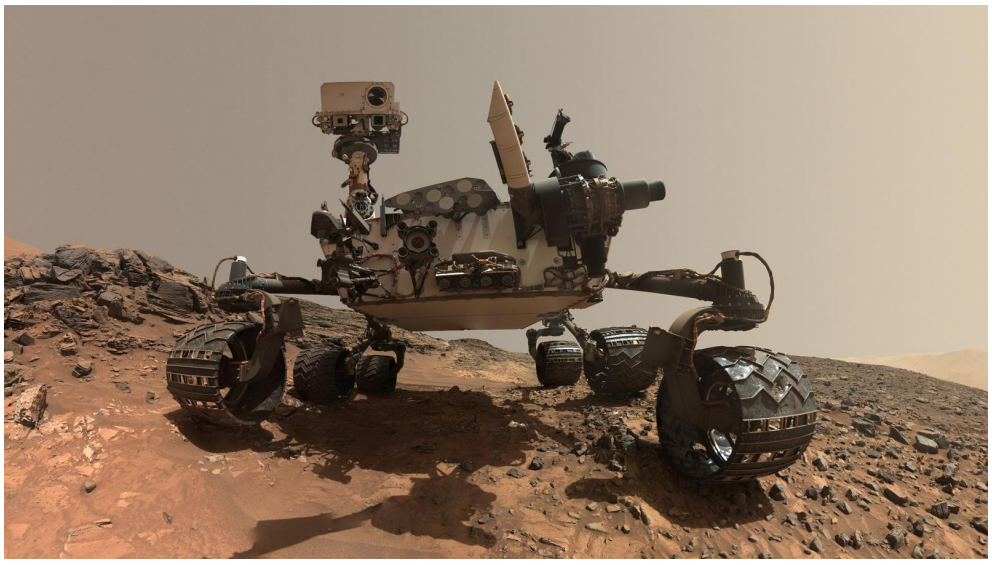
À part ce microscopique détail qui ressemble à s'y méprendre une verrue purulente datant du jurassique, rien à signaler.

Les fichiers annexes du module

Vous faites un correcteur orthographique et vous voulez intégrer des dictionnaires. Vous faites un jeu et vous voulez intégrer des images. Vous faites un simulateur de combats de poulets géants et vous voulez générer des cartes à partir des données géospatiales de Mars. Certes, pourquoi pas.

Le point commun ? Vous voulez intégrer dans votre module des fichiers qui ne sont pas du code Python.

Ces fichiers ne sont pas des métadonnées. Ce sont bien des données à installer avec le code, directement utilisées par le code, et sans lesquelles votre module ne pourrait pas fonctionner. D'ailleurs, sauf si vous souffrez d'une grave carence d'empathie envers le reste de l'espèce humaine, vous mettrez ces données à l'intérieur du dossier de votre module, si possible dans un sous-dossier dédié.



Ce n'est pas parce que Curiosity n'a pas trouvé de poulets géants qu'ils n'existent pas.

J'ai une bonne nouvelle. Tout ce qui reste assez classique est aisément intégrable. Tout ce qui est à gérer sur mesure peut être géré avec le doigté (et la patience) nécessaire.

Oh, avant de passer à la suite, j'ai une petite chose à vous dire… Pour accéder à ces fichiers depuis votre code, vous aurez sans aucun doute la naïveté d'aller les chercher avec leur nom de fichier, à partir du chemin relatif de votre code. C'est sans compter avec les eggs qui peuvent s'utiliser sans se décompresser, sans compter sur les exécutables Windows que l'on peut construire et qui contiennent toutes les données en un fichier, et sans compter sur les personnes qui viendront vous voir pour vous expliquer que sur Mars, les poulets géants utilisent un format d'archives quantiques qui s'utilisent à la fois compressés et décompressés.

J'exagère à peine. Dans le doute, utilisez `importlib.resources`. Comment ça, c'est uniquement à partir de Python 3.7 ? Eh bien… Faites preuve d'un peu de résilience, que diable ! Après quelques semaines à lire tous les forums de la Terre vous devriez aisément trouver une solution qui fonctionne dans tous les cas.

Les exécutables

Cette partie est simple : n'intégrez pas d'exécutable.

Bon, d'accord. Vous râlez parce que vous ne comprenez pas comment, sans exécutable, on pourra lancer votre superbe logiciel qui incruste un Nyan Cat sur votre fond d'écran. Et vous avez bien raison.

Ce chat qui vole sera du plus bel effet sur votre fond d'écran. Réfléchissez-y sérieusement.

Mais j'ai raison aussi. Pour avoir un exécutable installé par votre paquet, vous n'avez pas besoin de l'écrire. Python vous propose un astucieux système de points d'entrée pour vous économiser une partie du travail.

Ces points d'entrée sont des fonctions qui seront automatiquement transformées en exécutables à l'installation de votre paquet. Cette solution offre pas mal d'avantages, comme celui de pouvoir utiliser votre application `backnyan` en lançant `backnyan` dans votre terminal (ou en cliquant sur l'icône de l'exécutable installé), mais également comme module avec `python -m backnyan`. Et voilà, sans vous en rendre compte, vous avez gagné la possibilité d'utiliser un autre module avec votre application. Au hasard, avec `python -m pdb -m backnyan` vous pouvez désormais vous adonner aux joies de la correction interactive d'erreurs.

Je vous souhaite bien du plaisir avec pdb. C'est cadeau. Paquet cadeau.

Les opérations à la création

Jusqu'à ce moment précis, je vois dans vos yeux les innocentes lueurs de l'espoir, de celles qui animent les êtres dotés d'une raison de vivre encore immaculée, enivrés par l'alléchante fragrance de l'atteignable réussite.

Je vous propose de vous arrêter là.

Tant pis pour vous, si vous continuez, n'allez pas vous plaindre.

Nous avons vu à maintes reprises que `setup.py` est un fichier Python classique qui permet d'exécuter toutes sortes de fantaisies. Et par « fantaisies », je ne veux pas parler de licornes, je suis plutôt Cerbère ou Minotaure. Des trucs qui mordent et qui font mal.

Dans les opérations que peut faire `setup.py`, et si vous le voulez bien, nous déterminerons deux groupes distincts : celles faites avant de créer le paquet, et celles faites après. Nous commencerons ici par le premier groupe.

À la création d'un paquet, on pourrait avoir l'envie déviante de tripatouiller les fichiers. On voudrait à la volée créer des fichiers à inclure dans le paquet, ou en récupérer en ligne. On voudrait faire quelques ajustements pour créer des paquets optimisés pour une version de Python particulière, ou pour un système d'exploitation particulier.

Oh, mais attendez, ça donne des idées. On pourrait compiler du code C pour intégrer des versions différentes dans des wheels spécifiques. On pourrait directement créer des exécutables ou des archives spécifiques. On pourrait obfusquer du code propriétaire.

Au lieu d'écrire un module, on pourrait écrire un méta-module qui génère le code du module à la volée.

Vous avez saisi l'idée.

Si ces exemples vous semblent étranges, voire farfelus, prenez quelques minutes pour y réfléchir sérieusement. Ce ne sont que des exemples tirés de faits réels, mis en place sans mauvaise foi. True story. Et Python permet de faire cela sans trop de peine, puisque `setup.py` est un simple fichier Python.

Cela montre également que sans fichier `setup.py`, il serait très difficile d'effectuer tout cela. On ne peut pas faire de simple fichier de configuration qui prend en compte tous ces cas.

On commence à comprendre ici pourquoi il est illusoire en Python d'avoir un outil unique pour créer des paquets. Entre la simplicité et la complexité, entre un format statique et du code dynamique, la bonne solution dépend du contexte. C'est pour cela que nous aurons encore pendant longtemps de nombreux tutoriels, chacun axé sur une solution particulière, sans échappatoire unique possible.

Les opérations à l'installation

Nous voilà arrivés à l'étape ultime.

Même si l'on peut faire beaucoup de choses complexes au moment de la création d'un paquet, parfois pour des raisons honnêtes, cela reste cantonné à la responsabilité de celle ou celui qui crée le paquet.

À la limite, toutes ces opérations pourraient être faites en dehors de l'outil de création de paquet, par un script externe exécuté avant d'utiliser la pile logicielle classique pour générer l'archive.

Dit comme ça, c'est presque facile.

La vraie complexité, c'est d'exécuter du code à l'installation. Pour cela, on est (presque) obligé de dépendre de ce que `pip` fournit, et donc de retomber dans les aléas de `setuptools`.

Pire. Le code étant exécuté sur la machine où le paquet est installé, il doit s'adapter à ses spécificités : son système d'exploitation, son système de fichiers, sa version de Python, ses outils installés… Il est donc souvent nécessaire de faire preuve d'inventivité et de dextérité pour mettre en place du code qui s'adapte à la cible.

Mais, pour quoi faire ?

Dans le cas de paquets sources, on peut vouloir faire à l'installation à peu près tout ce que l'on voulait faire à la création d'une wheel. Mais on peut cette fois-ci le faire en utilisant tout ce qui est à disposition sur la machine hôte : compilation optimisée selon l'architecture, interfaçage avec des bibliothèques spécifiques installées, adaptation à la version de Python, du système d'exploitation, de certaines dépendances…

`setuptools`, pour ne citer que lui, offre une bonne dose d'outils pour simplifier ces tâches, et en particulier la compilation. Écrire du code C au milieu de sa bibliothèque Python, pour en optimiser certaines parties, est une pratique assez répandue. Et dans ce cas, soit on a les moyens pour générer des wheels pour toutes les plateformes (utopique), soit on laisse `pip` se débrouiller à l'installation.

Malheureusement, dépendre de l'hôte signifie dépendre des outils qui y sont installés. Il faut donc espérer que votre cible possède un compilateur en état de marche, adapté à ce que `setuptools` sait faire. Dans le cas contraire, le travail nécessaire pour installer votre trésor de code risque fort de décourager même les plus téméraires.

On va où, maintenant ?

Maintenant que nous avons entraperçu l'étendue de ce que l'on peut faire avec un paquet Python, nous voilà bien avancés. On va où, maintenant ?

Mine de rien, nous avons bien avancé. Si vous avez bien suivi (et je n'en doute pas), vous devriez pouvoir vous en sortir. Après avoir déterminé plus précisément ce que vous souhaitez et devez faire avec votre code, vous devriez être à même d'exploiter le tableau de la boîte à outils pour choisir avec précision les armes les plus adaptées.

Évidemment, il va vous falloir lire une bonne dose de documentation, d'articles et autres forums pour vous faire une idée plus précise.

Nous y sommes. Nous avons défait des nœuds, creusé jusqu'aux racines, décortiqué des formats, éparpillé des fichiers, ouvert une boîte à outils et défini des besoins. Il nous faudrait peut-être penser à faire un paquet maintenant !

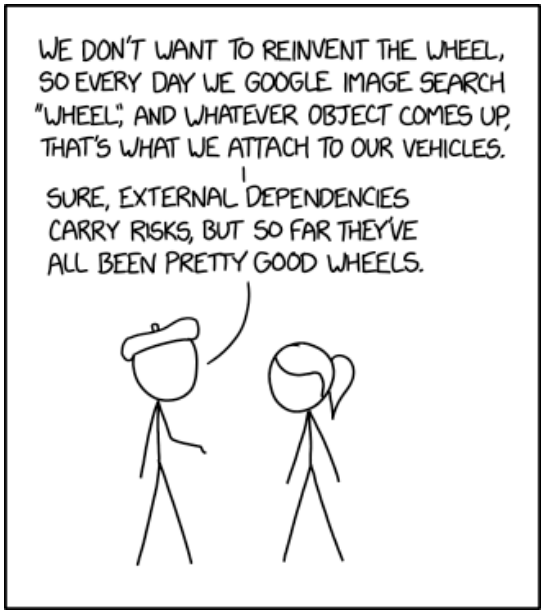
C'est où qu'on met quoi ?

Nous avons vu tout ce que l'on peut mettre dans nos paquets. Ça fait une bonne dose de choses à inclure, mais on n'a pas encore vraiment déterminé ni où ni comment.

Comme cet article n'est qu'un exemple parmi tant d'autres, nous n'aurons aucun scrupule à proposer des avis purement subjectifs basés sur une sensibilité toute particulière. Ce n'est pas la vérité absolue, c'est juste un avis. Mais comme c'est le nôtre il est forcément bien, sinon vous seriez en train de lire autre chose.

Le paquet wheel

Le paquet wheel, c'est le niveau 1 du paquet. C'est non seulement le plus simple à créer, mais également celui qui sera le plus utilisé.



Le XKCD obligatoire. Et si la construction de votre wheel échoue, n'abandonnez pas, la routourne va tourner.

Le but de ce paquet est de fournir le code fonctionnel déjà tout prêt pour la cible. Le `pip` qui l'installera n'aura rien à faire d'autre que de décompresser l'archive, récupérer quelques informations (comme les dépendances) et mettre le dossier décompressé sur le disque. Pas de compilation, pas d'exécution de code Python arbitraire, pas de cas particulier pour telle ou telle plateforme.

Cela signifie accessoirement que, si votre code n'est pas le même selon la version de Python ou le système d'exploitation, vous devrez créer autant de paquets qu'il y a de combinaisons différentes d'ordinateurs sur lesquels installer votre superbe création. Faut bien bosser un peu, quand même.

Qu'importe : nous avons des outils à disposition pour faire cela. L'important est de se mettre déjà d'accord sur ce que l'on veut mettre dedans. À ce sujet, mon avis est simple : le code, les métadonnées, et à la limite un ou deux fichiers comme le `README` ou la licence.

Le reste, ça dégage.

Pas d'états d'âme. Qui a déjà installé une wheel pour aller regarder le `CHANGELOG` à l'intérieur ? Qui a déjà installé une wheel pour exécuter les tests unitaires ? Qui a déjà installé une wheel pour reconstruire la documentation du module ? Si vous l'avez déjà fait, il est grand temps

pour vous de découvrir un outil formidable qui s'appelle Internet. C'est plein de pages super jolies qui présentent la documentation avec des couleurs, des images, et même des liens.

Plus sérieusement : les wheels n'ont pas d'autre vocation que de s'installer. Peu importe comment elles fonctionnent, ce qu'elles contiennent, l'important est qu'elles s'installent correctement avec `pip`. Tout le reste est secondaire. Vraiment. Adieu les tests, adieu la doc, adieu les fichiers de configuration.

Le paquet source

Les sources, c'est la base, c'est la solution de rechange, c'est le dernier recours, c'est l'exhaustivité ultime, et c'est pour ça que les wheels n'enverront jamais dans l'oubli ce vénérable format qui a accompagné les utilisateurs de Python depuis les débuts de la diffusion de paquets.

Avant de déterminer ce que l'on mettra dedans, réglons tout de suite une question déterminante : à quoi va servir concrètement notre paquet source ? Je crois que la question, elle est vite répondue :

1. à lire les sources,
2. à créer des paquets pour d'autres gestionnaires de paquets (par exemple pour les distributions Linux),
3. à installer des paquets dans les cas désespérés.

Dans ces cas-là, il est souvent utile d'avoir accès à un peu plus d'informations que celles contenues dans le paquet wheel. On aime avoir quelques fichiers à côté qui nous expliquent les changements à chaque version, quelques tests pour voir comment ça marche, un peu de documentation à lire avec un éditeur de texte… On peut avoir également envie de voir comment le paquet est configuré, de jouer avec le `setup.py`, voire de s'aventurer à tripatouiller le code.

Le paquet source contiendra donc, à peu de choses près, le même contenu que celui du dépôt versionné. On doit récupérer dans les grandes lignes ce que l'on aurait récupéré avec un `git clone` (ou l'équivalent avec votre logiciel de gestion de versions favori, on n'est

pas sectaires). On enlèvera évidemment le versionnement en lui-même et quelques détails comme la configuration de l'intégration continue.

À l'attaque

Avec toutes ces précisions, il nous est désormais possible de nous aventurer dans la création de paquets. Nous n'irons pas dans le détail de chaque ligne de configuration ou de code à écrire (il faut bien vous laisser un peu de liberté !), mais nous tâcherons au moins de mettre en place les bases nécessaires à la création de vos paquets.

À paquet simple, solution simple

Nous allons nous débarrasser comme ça, en quelques mots, d'un énorme non-dit qui nous encombre. Vous voulez faire un paquet ? Disons que votre code ne contient que du Python, et que vous voulez suivre les règles que nous avons fixées auparavant. Voilà, ça va déjà mieux.

Envie de planter un clou ? Utilisez donc un marteau ! Cet outil est parfait pour planter des clous, et… C'est tout, en fait. Ça plante des clous, ça ne fait que ça, mais ça le fait bien.

Nous sommes d'accord ? Sinon, vous pouvez toujours vous consoler avec un autre article, quelque part sur Internet, qui parle de `setuptools`. Je vous laisse chercher.

L'outil

Sans plus de suspense, l'outil que nous allons utiliser s'appelle Flit.

Flit, c'est le marteau de la création de paquets. C'est limité, ça ne fait qu'une chose, mais c'est clair, limpide, efficace. Nous voulons créer et diffuser des paquets avec des règles simples à suivre bêtement, c'est tout.

Flit, c'est aussi l'un des outils à la base des PEP 517 et PEP 518. Oui, c'est entre autres grâce à son créateur Thomas Kluyver que nous avons désormais la possibilité de nous affranchir de `setuptools` et de `setup.py`. Respect.

Flit, c'est avant tout la simplicité. Si vous ne voulez pas vous poser de questions sur la création de paquets, si vous ne voulez pas écrire de code autre que votre module, c'est le choix de la raison.

L'architecture

Oubliez les tonnes de fichiers et les configurations à n'en plus finir. Si l'on vise la simplicité, il va nous falloir éradiquer l'obésité des dossiers racines. Nous allons drastiquement alléger la page d'accueil de votre dépôt.

Sans nous replonger dans l'ensemble des projets que nous avons déjà cités, prenons seulement un exemple de ce que nous ne voulons pas : Requests.

(Requests n'est pas le mal absolu, attention, ne me faites pas dire ce que je n'ai pas écrit. Simplement, c'est un bon exemple de ce que nous ne voulons pas.)

Le dossier racine contient 22 fichiers et dossiers. On retrouve à l'intérieur les suspects habituels de la création de paquets : `setup.py`, `setup.cfg`, `MANIFEST.in`, `Pipfile`… On retrouve également les jolis fichiers de configuration des outils tiers : Tox, Coverage, Pytest. Quelques métadonnées, quelques dossiers, voilà de quoi impressionner n'importe quelle personne qui voudrait prendre ce projet très connu en exemple pour comprendre comment fonctionnent les modules Python.

À l'opposé de ce projet, je vous propose 3 dossiers et 3 fichiers de base à inclure dans votre dossier racine :

1. le dossier contenant le code de votre projet,
2. un dossier `doc`,
3. un dossier `tests`,
4. un fichier `LICENSE`,
5. un fichier `README`,
6. un fichier `pyproject.toml`.

Bien sûr, ce n'est qu'une base que vous pourrez adapter à vos besoins. Mais se restreindre à garder un dossier racine léger et propre est également un bon prétexte pour réfléchir à l'hygiène et la structuration de son projet. Voyons de ce pas ce que nous pouvons faire entrer dans ces petites cases…

Les dossiers

Dans le dossier contenant le code de votre projet, vous allez avant tout mettre… le code de votre projet. Cela n'a l'air de rien, mais si l'on se tient à notre idée d'avoir une wheel minimaliste, on comprend rapidement que ce dossier sera celui qui sera prêt à être décompressé. Dans le simple but d'installer le module, le reste n'est que décoration.

Une conséquence de ce découpage est que ce dossier doit inclure les fichiers annexes nécessaires au fonctionnement du module. Les images de votre jeu ? À inclure dans ce dossier. Les listes de mots de passes connus pour votre outil de piratage de la NSA ? À inclure aussi.

Cela explique également pourquoi on n'inclut dans ce dossier ni tests, ni documentation. Tout le monde sait que les tests et la documentation ne servent à rien quand le code est limpide et dénué d'erreurs. Cependant, dans le doute, en attendant que tous les humains soient omniscients, en attendant de pouvoir parfaitement nous passer de ces broutilles, nous pourrions conserver toutes ces reliques, mais seulement dans le paquet source.

Les tests, s'ils suivent les conventions de nommage de votre outil favori, seront automatiquement découverts. À cet égard, `tests` semble être un nom particulièrement adapté, à la fois pour les

humains et pour les outils (Flit ou Pytest, par exemple). Après, libre à vous d'organiser vos tests comme bon vous semble, mais vous aiderez tout le monde en les mettant tous dans le même dossier, à la racine de votre projet.

La documentation a, elle aussi, de bonnes raisons d'être dans le paquet source. Vous donnez aux curieuses et aux curieux la possibilité d'aller fouiller dans les limpides explications sur votre projet, tout à côté de votre code, sans accès Internet requis. Vous donnez aux distributions Linux la possibilité d'inclure une appréciable présentation de votre outil, mais également d'éventuelles pages de manuel. En fait, vous donnez à n'importe qui la possibilité de faire n'importe quoi avec du contenu qui aide les gens, et dans ce cas-là vous n'êtes jamais à l'abri de bonnes surprises.

Cette documentation est également l'endroit idéal où stocker certaines informations que l'on pourrait avoir envie de mettre à la racine, comme un `CHANGELOG` ou des exemples de configuration. Elles seront ainsi disponibles dans un format agréable à lire, en plus d'être toujours accessibles dans des fichiers texte. Les plateformes d'hébergement de code proposent aussi des pages dédiées à certaines de ces informations, rendant inutiles bon nombre de fichiers superflus à la racine. Et rien ne vous empêche, si vous en avez réellement envie, de mettre des liens dans votre `README` pour aiguiller les gens qui ne cherchent qu'à la racine.

Les fichiers

Le fichier `README` est la base de votre projet. Dans un format en pur texte ou avec un balisage léger, c'est la porte d'entrée par laquelle une bonne partie des gens intéressés par la technique vont venir. Il a également de bonnes chances d'être mis en avant dans votre forge logicielle et sur PyPI.

Voilà donc une excellente raison de bien travailler votre `README`. Au-delà de la description du projet, vous devez pointer toutes les informations nécessaires que l'on aime généralement trouver rapidement quand on découvre un projet : la licence, les versions de Python supportées, les moyens de contacter celles et ceux qui s'occupent de la maintenance…

D'ailleurs, placer la licence à la racine du projet, dans un fichier à part, est un choix extrêmement classique mais discutable. Après tout, cette information juridique n'aurait-elle pas sa place dans la documentation ? Une ligne dans le `README` ne suffirait-elle pas à indiquer la licence qui s'applique au projet ?

Si, sans doute. Mais beaucoup d'outils s'attendent à trouver à la racine ce fichier, voire même le lisent pour en déduire la licence du projet. S'il est facile de faire changer les habitudes des gens, qui se satisferaient sans doute d'une indication dans la documentation, il est plus complexe de faire évoluer les habitudes des machines. Alors… Disons que ce choix est un petit arrangement entre l'idéal et la réalité. Œuvrons pour que dans quelques années nous puissions nous en passer plus simplement.

Enfin, il reste bien sûr le plat de résistance : `pyproject.toml`. Ce fichier vous permet tout d'abord d'indiquer tout ce qu'il faut pour la création du paquet. Les choix par défaut de Flit étant particulièrement bons (en toute objectivité), vous ne devriez pas avoir besoin de changer grand chose aux valeurs proposées. Mais sachez que si l'envie s'en fait sentir, vous trouverez une bonne liste d'options qui devraient satisfaire vos idées les plus folles.

Avec Flit, `pyproject.toml` va remplacer ce que vous pouvez faire avec (au moins) `setup.py`, `setup.cfg`, `requirements.txt` et `MANIFEST.in`. Évidemment, les possibilités sont limitées, ne serait-ce que parce que vous ne pouvez pas écrire de code Python pour commettre des atrocités exécutées lors de la création ou l'installation d'un paquet. Mais ce n'est pas une limitation, c'est une fonctionnalité : fini de jouer avec cette idée initialement intéressante mais devenue plus qu'immonde, il serait peut-être plus utile pour la postérité d'écrire votre module.

C'est également ce fichier qui va vous permettre de configurer une très grande partie des outils annexes que vous utilisez : Tox, Black, Pytest, Coverage.py, isort, Pylint… Oui, cela signifie que vous pouvez dire adieu aux montagnes de fichiers de configuration utilisant chacun leur convention de nommage et leur format ! La liste des

outils supportés s'agrandit régulièrement, n'hésitez pas à jeter un coup d'œil de temps en temps pour voir si votre projet favori a osé sauter le pas.

De la création au déploiement

Ne vous attendez pas à un tutoriel dans lequel je vous tiendrais la main, j'écrirais vos fichiers de configuration, et je vous donnerais toutes les astuces pour utiliser Flit. L'article ne s'appelle pas « Les 7 trucs que vous ne connaissez pas sur Flit – le cinquième va vous surprendre » (pas sûr qu'on explose les statistiques de visites avec un titre comme ça, cela dit).

Pourquoi ? Simplement parce que la documentation de Flit est formidable. Vous y trouverez ce qu'il faut pour installer et utiliser Flit les yeux fermés (ou presque). C'est limpide, c'est rapide, et surtout ça marche.

`init`, `install`, `build`, `publish`. C'est tout ce dont vous aurez besoin pour modeler vos petits paquets avec amour. Plus besoin de subir ma prose désobligeante, je vous laisse entre les mots délicats d'un outil qui l'est tout autant.

Prenez plaisir à voler de vos propres ailes, laissez-vous porter au gré du vent.

Oui, des autruches. Quand on ne trouve pas d'image de papillon ou de libellule, on fait ce qu'on peut avec ce qu'on a…

Pour le reste…

Ne nous mentons pas : Flit ne résout pas tout. Nous avons déjà vu ses limitations et ses petites faiblesses, mais il n'est heureusement pas seul.

Vous voulez un autre outil qui vaut également largement le détour ? Poetry peut faire ce que fait Flit, mais il fait beaucoup plus encore : de la gestion d'environnements virtuels, de la résolution de dépendances, de l'installation de paquets, de la numérotation de versions… Si vous aimez les outils tout-en-un qui évitent l'écueil de devenir des mastodontes tentaculaires et mégalomanes impossibles à maintenir (je ne donnerai pas de nom), vous trouverez en Poetry un élégant remplaçant à Pipenv (oups, j'ai craqué, désolé).

Mais… Que ce soit sur Flit ou Poetry, une grande ombre plane : l'abandon. Flit et Poetry ont beau être largement utilisés, ils n'en restent pas moins des outils tiers qui ne sont pas supportés comme l'est `setuptools`. À l'instar de nombre de projets libres, ils ont d'ailleurs déjà eu des coups de pompe, et ils en connaîtront encore d'autres.

Heureusement, les PEP sont maintenant largement adoptées, laissant la porte ouverte à d'autres projets futurs. Sortis du carcan de `setuptools`, nous pouvons à loisir utiliser d'autres outils basés sur `pyproject.toml`. Les clés et valeurs des options changeront, mais au moins n'aurons nous plus besoin de dépendre d'une implémentation unique sclérosée par le poids de l'historique et le besoin de rétrocompatibilité.

Les wheels resteront les wheels, les sources resteront les sources, et tout sera pour le mieux dans le meilleur des mondes.

Les outils passent mais les formats restent.

Un site léger en 2020

Aujourd'hui nous allons parler de notre site des outils que nous avons utilisés pour le créer.

De quoi avions-nous besoin ?

Nous voulions avoir un site simple, rapide et léger. Hors de question donc d'utiliser des outils tels que Bootstrap et ses 156 ko de CSS (minifié), ou des librairies JavaScript. D'ailleurs pourquoi mettre du JavaScript alors que nous n'avions besoin que de deux pages ?

C'est ainsi que nous décidâmes de développer notre site uniquement avec du vieux HTML et CSS. Quoi de plus rapide que des pages statiques servies directement avec le strict minimum ?

Nous étions alors plutôt satisfaits de nos deux pages. Certes le header et le footer se trouvaient dupliqués. Mais bon, s'il fallait les changer, ce n'était que dans deux fichiers HTML. Ça va.

Puis nous voulûmes rajouter un blog…

De nouveaux besoins

Avec un blog, ça n'allait plus être juste deux pages et donc deux fichiers. Non. Potentiellement ça allait être une infinité de pages et donc de fichiers. Ce n'était pas raisonnable de dupliquer du contenu autant de fois.

Il nous fallait alors quelque chose qui nous permette d'utiliser des templates. Et pourquoi pas en utilisant un framework web ?

Hugo, Jekyll, Pelican, ColdCMS… Le choix est large. Tous ces frameworks proposent de nombreuses fonctionnalités et sont plutôt faciles à utiliser. C'est bien, c'est souvent pratique. Mais ce n'est pas ce qui nous convenait.

Nous voulions un outil puissant, modulable et qui nous permettrait de faire ce que nous voudrions plus tard sans être contraints par un modèle de base. Nous pensâmes alors à Flask.

Cette idée semblait plutôt intéressante et simple. Mais avec Flask, contrairement aux autres outils, ce ne serait plus des pages statiques servies directement. Adieu la rapidité.

Nous nous tournâmes alors vers Frozen-Flask.

Frozen-Flask

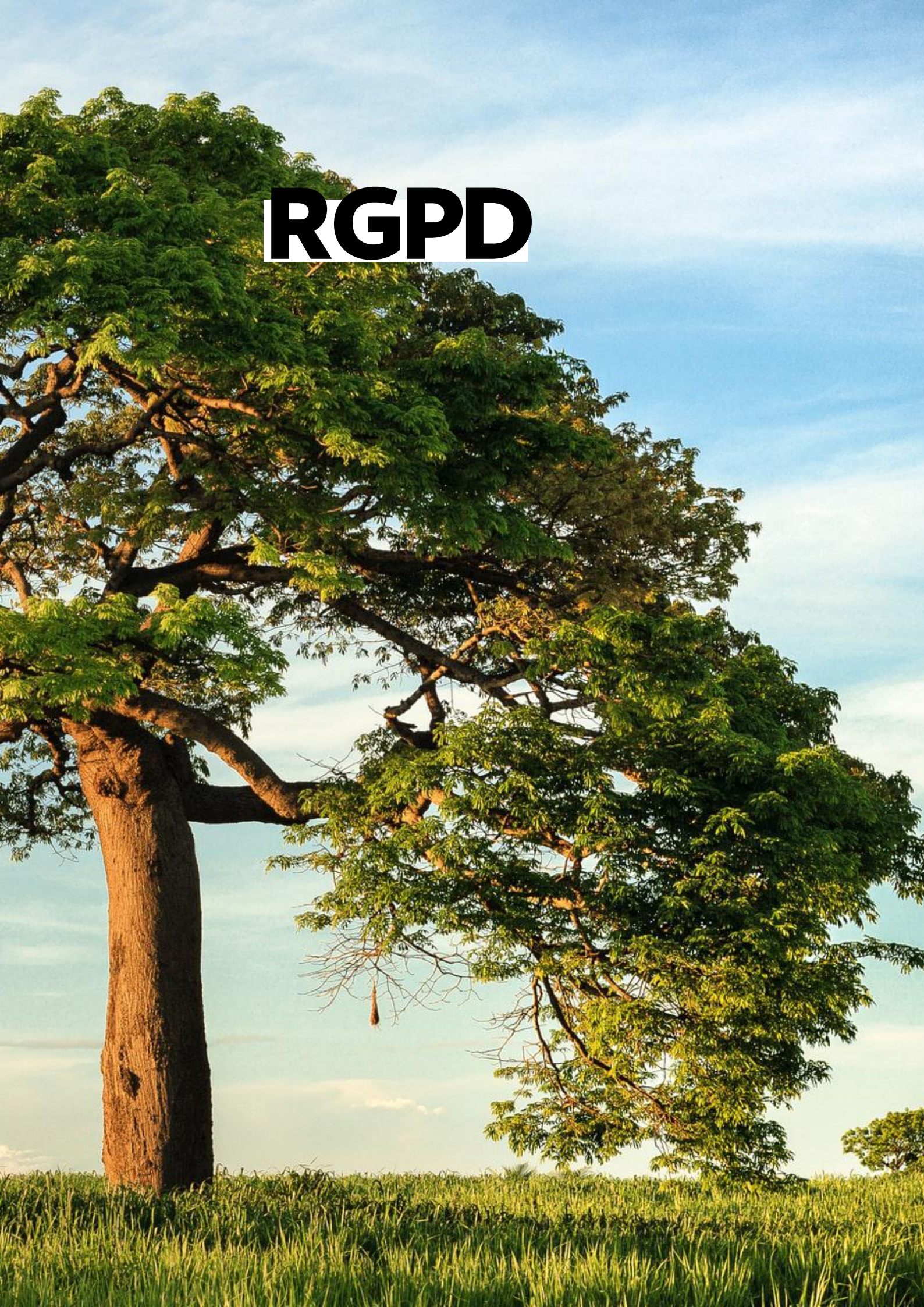
Cet outil permet de générer les fichiers statiques correspondants à une application Flask. Nous étions sauvés ! Nous n'avions plus qu'à transformer notre site en application Flask et à nous les fichiers statiques !

Grâce à la magie des templates, plus de duplication de header et de footer. Ils sont bien rangés dans un seul et unique template ; ce qui est bien plus facile à maintenir ;)

Et il est lourd comment ?

Toujours sans aucun JavaScript et uniquement en statique, aller sur la page d'accueil du site ne coûtera que 312 ko de bande passante. Alors que d'après l'écoIndex, la médiane est à 1.41Mo…

Si vous souhaitez en savoir plus sur les entrailles de notre site, le code est en libre accès sur GitLab.


RGPD

L'arbre généalogique du RGPD

Le RGPD n'est pas la première loi destinée à encadrer l'utilisation des données personnelles. Dans cet article, nous allons voir les principales lois qui y ont participé et continuent à y participer.

Les années 70

Au début des années 70, l'idée d'avoir un comité de surveillance et un tribunal de l'informatique commence à germer. Et c'est vers la fin de cette décénnie, en 1978, que la loi informatique et libertés est signée.

Cette loi permet alors de règlementer les traitements d'informations nominatives, terme qui sera modifié quelques années plus tard.

La loi informatique et libertés place alors la législation sur l'informatique dans le cadre des droits de l'homme dans son premier article.

> L'informatique doit être au service de chaque citoyen. Son développement doit s'opérer dans le cadre de la coopération internationale. Elle ne doit porter atteinte ni à l'identité humaine, ni aux droits de l'homme, ni à la vie privée, ni aux libertés individuelles ou publiques.

La Commission Nationale de l'Informatique et des Libertés (CNIL) est créée. Sa mission est définie : elle est chargée de veiller au respect de cette loi.

L'exercice du droit d'accès est également défini. Toute personne peut demander à accéder aux informations nominatives qu'un organisme dispose sur elle. Elle peut également demander leur modification ou leur suppression.

Un autre droit important est aussi spécifié : le droit d'opposition. Une personne peut s'opposer à ce que ses informations soient traitées.

La loi limite aussi les traitements informatisés sur certains types de données particuliers. En effet, si les données indiquent les origines raciales, les opinions politiques, philosophiques ou religieuses, ou encore les appartenances syndicales, leur traitement est soumis au consentement des personnes concernées.

La loi informatique et libertés définit donc dès les années 70 des concepts toujours d'actualité. Elle sera mise à jour au fil du temps.

Les années 2000

En 1995, l'Union Européenne publie une directive concernant la protection des personnes physiques par rapport aux traitements de données à caractère personnel et la circulation de ces données. Elle essaye alors d'harmoniser les règles sur les traitements de données personnelles entre les différents pays de l'UE.

Cette directive indique qu'un traitement doit respecter trois principes : la proportionalité, la transparence et la finalité légitime. La directive reprend aussi les droits d'accès, de modification et de suppression définit dans la loi informatique et libertés ; ainsi que le droit d'opposition.

Cette directive de 1995 crée le G29, qui sera remplacé plus tard, qui a pour but, entre autres, de conseiller la Commission Européenne sur les sujets liés à la protection des données personnelles.

Cette directive ne sera appliquée en France qu'en 2004. En effet, une directive doit être transposée dans le droit de chaque pays.

Cette transposition entraine des modifications dans la loi informatique et libertés, notamment en définissant de façon plus précise le concept de données personnelles qui remplace alors celui d'information nominative.

La loi informatique et libertés est modifiée plusieurs fois pendant les années 2000 par différents décrets.

Parmi ces modifications, on peut noter une harmonisation des règles entre le secteur public et privé. En effet, le public n'a plus besoin de demander d'autorisation à la CNIL lors de la constitution de fichiers contenant des données personnelles, mais doit les déclarer auprès de la CNIL.

Les modifications des années 2000 créent également le Correspondant Informatique et Libertés qui est alors en charge d'appliquer la loi dans son organisme.

Harmoniser encore et toujours

En 2016, le fameux Règlement Général sur la Protection des Données est voté par le Parlement Européen, entre en vigueur en 2018 et abroge la directive de 1995.

Contrairement aux directives, un règlement n'a pas besoin d'être transposé dans la législation de chaque pays mais est directement applicable.

Le G29 est remplacé par le Comité Européen de la Protection des Données qui a pour mission de garantir l'application du règlement de manière cohérente.

L'entrée en vigueur du RGPD entraine de nouvelles modifications dans la loi informatique et libertés pour la rendre plus cohérente avec la législation européenne.

Depuis sa création en 1978, la loi informatique et libertés a donc connu de nombreuses modifications afin de s'adapter aux évolutions techniques et législatives. Mais il est intéressant de noter que les principes de base sont encore et toujours d'actualité.

Quels sont les avantages du RGPD pour votre entreprise ?

Applicable depuis maintenant deux ans, le RGPD a beaucoup fait parler de lui et a pu susciter quelques appréhensions sur sa mise en place au sein de certaines entreprises.

Aujourd'hui, nous allons revenir sur ce Règlement et voir les avantages qu'il apporte à votre entreprise.

Le RGPD, qu'est ce que c'est ?

Le Règlement Général sur la Protection des Données (RGPD) permet d'encadrer les traitements des données personnelles dans l'Union Européenne et de créer un cadre juridique uniforme.

En France, il existe déjà une règlementation pour ces traitements via la loi Informatique et Libertés de 1978. Le RGPD s'inscrit dans sa continuité et permet de renforcer le contrôle des gens sur leurs données.

Tous les organismes sont concernés par le RGPD du moment qu'ils se trouvent dans l'Union Européenne, ou que leur activité est en direction des citoyens de l'Union Européenne, et qu'ils traitent des données personnelles.

Quels avantages en tant qu'entreprise ?

Le RGPD peut sembler être une nouvelle lourdeur juridique pour une entreprise, mais en réalité, il lui apporte de nombreux atouts.

Une confiance accrue

Le Règlement permet aux personnes d'avoir une plus grande maîtrise de l'usage qui est fait de leurs données. En respectant ces droits, l'entreprise montre un visage sérieux et responsable ; ce qui rassure les clients, futurs clients et les utilisateurs de vos services. L'image de marque de l'entreprise se trouve alors bonifiée.

Une meilleure efficacité commerciale

Lors de la recherche de nouveaux clients, il est facile de se retrouver avec beaucoup de données personnelles pas forcément à jour. Avec l'obligation de maintenir ces données à jour, l'entreprise dispose de données plus fiables et peut donc mieux cibler sa clientèle.

Des process plus clairs

Trier et mettre à jour les données et leurs traitements est l'occasion de se rendre compte que certaines données ne sont en réalité pas utilisées ou que des traitements n'ont plus lieu d'être. La suppression de ces tâches obsolètes rend alors les activités de l'entreprise plus simples et plus claires.

Une sécurité des données augmentée

Prendre réellement en compte la sécurité des données que l'entreprise traite, que ce soit les données de ses membres ou de ses clients, c'est se donner les moyens d'avancer posément. Il est primordial de sécuriser les données, c'est moins de stress, moins de fuites, et toujours une meilleure image pour l'entreprise.

Un avantage concurrentiel

Les personnes sont de plus en plus sensibilisées aux enjeux liés au traitement de leurs données. En montrant sa prise en compte du RGPD et du respect des droits des gens, l'entreprise rassure ses clients, ses prospects, ses fournisseurs. Le RGPD devient alors un avantage concurrentiel.

De nouveaux services

La prise en compte du Règlement peut donner lieu à de nouveaux besoins. C'est l'occasion de créer de nouveaux outils y répondant. Et surtout de le prendre en compte tout au long de leur développement afin d'améliorer l'expérience utilisateur.

Appliqué correctement et mis en avant, le RGPD permet donc à une entreprise de se démarquer de ses concurrents, de gagner de nouveaux clients, de proposer de nouveaux services et de mieux s'y retrouver dans ses process.

Le privacy by design, qu'est ce que c'est ?

Mis en avant par le RGPD, le concept du Privacy by Design connait une hausse d'intérêt. Mais saviez-vous que ce concept n'est pas tout jeune ?

Un peu d'histoire

En effet, les premières traces du Privacy by Design remontent à l'an 1995 où il fût décrit par Ann Cavoukian ; et formalisé la même année dans un rapport écrit par la Commissaire à l'information et à la vie privée de l'Ontario, la CNIL néerlandaise et l'organisation néerlandaise pour la recherche scientifique appliquée.

Ce concept a pour but de préserver la liberté de choix des utilisateurs et leur contrôle sur les données les concernant en mettant au cœur du développement de services les problématiques liées à la vie privée.

Les sept principes du Privacy by Design

Le Privacy by Design se définit autour de sept principes.

Proactif plutôt que réactif, préventif plutôt que correctif :
: Les incidents liés à la vie privée doivent être anticipés au lieu d'attendre qu'ils arrivent pour réagir. Ainsi, il y a moins de risques qu'un incident ait lieu.

Privacy by default :
: La confidentialité est le paramètre par défaut. L'utilisateur ne doit pas avoir besoin d'aller modifier des paramètres pour que sa vie privée soit respectée. Cette notion rejoint celle de minimisation de la collecte de données également abordée par le RGPD.

La confidentialité dès la conception :
: La confidentialité est prise en compte dès la conception d'un service. Ce ne doit pas être quelque chose qui est ajouté par la suite, mais au contraire un élément essentiel.

Ne pas nuire aux fonctionnalités :
: Le respect de la vie privée ne doit pas être un frein aux diverses fonctionnalités. L'équilibre entre les intérêts du fournisseur de service et ses utilisateurs est également recherché.

La sécurité de bout en bout :
: Des mesures de sécurité sont prises afin de garantir du mieux possible la sécurité des données de leur collecte à leur destruction.

Visibilité et transparence :
: Toutes les parties prenantes autour d'un service ont une visibilité sur ce qui est fait, que ce soit pour les fournisseurs ou les utilisateurs. Cette transparence permet d'avoir une meilleure confiance entre les différents acteurs.

Le respect de la vie privée des utilisateurs :
: Surtout, les intérêts des utilisateurs doivent être au centre des préoccupations, que ce soit en mettant en place des mesures de sécurité fortes, en mettant à disposition une information claire, etc.

Appliquer les principes du Privacy by Design

Malgré son adoption dans de nombreux pays, le Privacy by Design rencontre des critiques, essentiellement car ses principes sont jugés trop flous et complexes à mettre en place dans certains systèmes. Beaucoup de paramètres sont à prendre en compte et difficiles à implémenter.

Mais des recommandations existent pour aider à l'application du Privacy by Design, comme celles de l'ENISA. Parmi celles-ci on retrouve :

Minimiser :
: Uniquement les données strictement nécessaires sont collectées. Il ne peut rien arriver aux données que l'on ne possède pas.

Dissimuler :
: Les liens entre les données sont dissimulés, ainsi que les données elles-mêmes. On peut chiffrer les données, utiliser différents réseaux lors des transferts, anonymiser, pseudonymiser…

Séparer :
: Les données sont stockées à différents endroits de façon à ce que toutes les données sur un même individu ne se retrouvent pas toutes ensemble.

Aggréger :
: Suivant les besoins, les données peuvent être aggrégées. De cette façon, les risques d'identification des personnes sont par exemple réduits.

Informer :
: Pour répondre au principe de transparence, les différents acteurs sont informés des traitements, des technologies utilisées, des fuites de données si cela se produit…

Contrôler :
: Les utilisateurs doivent être capable d'exercer leurs droits sur les données les concernant, et de le faire d'une manière sécurisée mais aussi intuitive.

Renforcer :

Une politique de sécurité doit être en place et revue régulièrement.

Démontrer :

Être capable de montrer que n'importe quoi n'est pas fait avec les données, qu'il y a une politique de sécurité et qu'elle est respectée.

Le Privacy by Design est une composante importante dans le respect de la vie privée des individus. Il apporte donc des éléments de réflexion plus que de réponse sur comment mettre l'utilisateur et ses droits au centre de la conception d'un service.

Transférer des données aux États-Unis

En 2015, la Cour de Justice de l'Union Européenne invalide le Safe Harbor qui permettait le transfert de données personnelles de l'Espace Économique Européen vers les États-Unis. Le Privacy Shield fut mis en place quelques temps plus tard, mais…

Au début, le Safe Harbor

Le Safe Harbor permettait de transférer des données personnelles vers les États-Unis depuis l'Espace Économique Européen du moment que les entreprises destinataires certifiaient respecter la législation de l'EEE.

Cette certification était obtenue par un contrôle du respect des principes du Safe Harbor, contrôle pouvant être effectué par l'entreprise elle-même.

Mais, en 2015, surgit alors Maximilliam Schrems qui obtint alors son invalidation, les États-Unis n'offrant pas un niveau de protection suffisant.

Dès l'année suivante, la Commission Européenne et les États-Unis travaillent main dans la main pour créer le Privacy Shield.

La courte vie du Privacy Shield

Début 2016, le Privacy Shield naquit. Tout comme son prédécesseur, il repose sur un mécanisme d'auto-certification des entreprises américaines et un engagement du gouvernement américain à retreindre l'accès aux données aux autorités.

Dès lors qu'une entreprise était certifiée, les organismes européens pouvaient lui transférer des données personnelles.

Mais, surgit de nouveau Maximilliam Schrems qui obtint alors l'invalidation du Privacy Shield, les États-Unis n'offrant toujours pas un niveau de protection suffisant. En effet, les différents programmes de surveillance américains permettent aux autorités de se servir dans toutes les données des entreprises américaines, et ce sans distinction et sans préavis.

Le glas du Privacy Shield sonna alors le 16 juillet 2020.

Les Clauses Contractuelles Types

Les Clauses Contractuelles Types n'ont, quant à elles, pas été annulées. Elles restent donc des outils valables pour encadrer les transferts de données personnelles en dehors de l'Union Européenne, États-Unis compris.

Cependant, la Cour de Justice de l'Union Européenne indique que l'exportateur et l'importateur de données doivent évaluer si les lois du pays importateur respectent le niveau de protection attendu par l'Union Européenne.

Le Privacy Shield ayant été annulé à cause d'une protection des données des résidents européens trop faible, comment les clauses contractuelles entre entités européennes et américaines peuvent être valides ?

Utiliser des services américains

On pourrait alors se dire que, du moment que l'entreprise dont on utilise le service s'engage à stocker les données uniquement sur des serveurs situés en Europe, c'est bon. Les données restent bien en Europe et ne sont pas transférées de l'autre côté de l'Atlantique.

Présentée comme ça, cette solution a l'air séduisante, une sorte de compromis entre tout le monde. Mais c'est sans compter sur le Cloud Act.

En effet, le Cloud Act est une loi américaine, mise en place en 2018, qui permet aux autorités d'avoir accès aux données des entreprises américaines, peu importe où se situent les serveurs. Et ce, sans obligation d'informer les personnes.

Même en essayant de montrer de la bonne volonté sur le respect des données personnelles, les entreprises américaines sont toujours soumises à la loi de leur pays.

Utiliser des services américains semble donc être juridiquement instable et peu recommandable.

La meilleure solution est probablement de se tourner vers les entreprises européennes.

Le droit à la portabilité des données

Ou le droit de récupérer les données que l'on a transmises.

Qu'est ce que c'est ?

Le droit à la portabilité est un droit qui a été amené par le RGPD. Il permet aux personnes de récupérer les données qu'elles ont transmises à un organisme pour les réutiliser.

Une personne peut donc reprendre les données qui la concernent :

- pour son usage personnel, par exemple en récupérant son historique d'écoute de musique depuis une application pour faire des statistiques
- pour les transférer à un autre organisme, par exemple si une personne utilise une application pour faire de la course à pied et voir sa progression et qu'elle décide de changer d'application, elle peut vouloir transférer les données de la précédente application vers la nouvelle.

Quelles données sont concernées ?

Lorsqu'une personne fait une demande de portabilité, toutes les données la concernant ne lui sont pas obligatoirement transmises.

En effet, les données sur lesquelles s'appliquent le droit à la portabilité sont :

- les données sciemment fournies par la personne et traitées sur la base légale du consentement ou de l'exécution d'un contrat
- les données générées par l'activité de la personne.

Pour reprendre l'exemple de l'application d'écoute de musique, lorsqu'une personne invoque son droit à la portabilité, elle reçoit les données qu'elle a données lors de la création de son compte. Comme par exemple, une adresse mail, une adresse de facturation, un identifiant… Ce sont des données appartenant à la première catégorie. Elle reçoit également son historique d'écoute, ses playlists, la liste d'artistes suivis… Ce sont des données de la deuxième catégorie.

Les données anonymisées ne sont pas concernées par la portabilité, car une donnée anonyme n'est pas considérée comme une donnée personnelle.

Les données générées par un organisme ne sont elles aussi pas concernées par la portabilité. Si l'entreprise proposant l'application d'écoute de musiques crée un profil de la personne à partir de ses écoutes, pour par exemple lui recommander de nouvelles musiques, elle n'a pas à le lui transmettre.

Respecter les données de tierces personnes

Il peut arriver que les données sujettes à la portabilité contiennent des données personnelles qui ne concernent pas la personne qui exerce son droit de portabilité.

Par exemple, si une personne utilise un service de carnet d'adresses et fait une demande de portabilité, son carnet d'adresses lui sera transmis avec les informations qu'elle aura créées. Et ces informations contiennent alors des données personnelles de tierces personnes.

Lorsque la personne qui a exercé son droit de portabilité transmet les données à un nouvel organisme, ce dernier ne doit pas utiliser les données pour son propre compte.

Si une fois son carnet d'adresses récupéré, la personne souhaite l'importer dans une nouvelle application de carnet d'adresses, l'entreprise gérant cette application ne peut qu'importer les adresses pour les mettre à disposition de son nouvel utilisateur. Elle ne peut pas s'en servir pour, par exemple, faire des opérations marketing sur les personnes contenues dans le carnet importé, ou enrichir le profil des personnes de ce carnet qui utiliseraient cette même application.

Le droit à la portabilité des données permet donc aux personnes d'avoir une meilleure maîtrise de leurs données personnelles, mais ne s'applique pas tout le temps sur toutes les données. Il donne aussi la possibilité aux entreprises de se démarquer du lot en permettant par exemple d'importer facilement les données de diverses applications.

Des interfaces au service des gens

De la poignée de porte au bandeau pour les cookies, comment les interfaces nous servent elles ?

Les interfaces sont des dispositifs permettant des échanges et des interactions entre différentes choses. Tous les objets en possèdent, ce qui nous permet de les utiliser. Une poignée de porte est une interface, permettant à un humain d'ouvrir et de fermer une porte. Un verre est une interface, permettant de boire plus facilement. Un volant est une interface, permettant de diriger une voiture. Bref, il y a des interfaces partout, et ces dernières nous aident dans nos tâches quotidiennes.

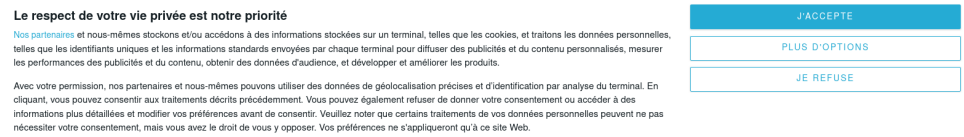
L'apparition des ordinateurs a entraîné la création de nombreuses interfaces afin de les contrôler, de traiter des informations, d'en partager et de rendre l'informatique accessible au plus grand nombre d'entre nous.

Au milieu de toutes ces interfaces, on trouve les interfaces graphiques, qui permettent de présenter et d'échanger des informations, ou encore d'effectuer des actions.

Comme toute interface, une interface graphique est là pour nous aider. Et comme dans tout genre d'interfaces, des « règles » se mettent en place pour en définir les codes. De la même manière que l'on sait de quel côté ouvrir un robinet pour avoir de l'eau chaude ou de l'eau froide sans réfléchir, on sait que cliquer sur une flèche qui va vers la gauche nous fera revenir à la page précédente. Une coche indique que l'action entreprise s'est bien passée, une croix informe du contraire.

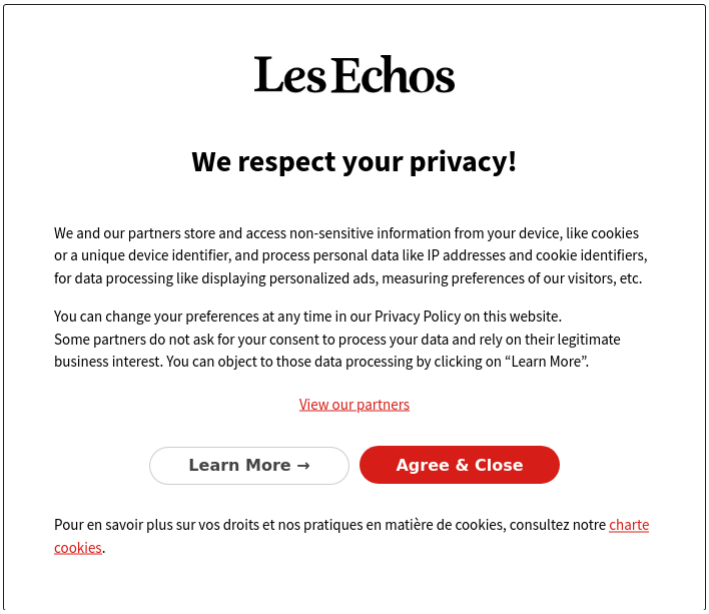
Alors que l'objectif d'une interface est de nous aider, et d'être le plus facilement compréhensible et intuitive, certaines d'entre elles ne le sont pas du tout. En particulier lorsque l'objectif est d'obtenir un accord pour de la collecte de données ou d'obtenir plus de données que nécessaire.

Par exemple, certains bandeaux à cookies permettent de tous les refuser ; même si il y a une certaine quantité de texte, il est rapide et facile de refuser.



Il est possible de tout refuser (« Accepter » est quand même mis plus en avant par sa couleur)

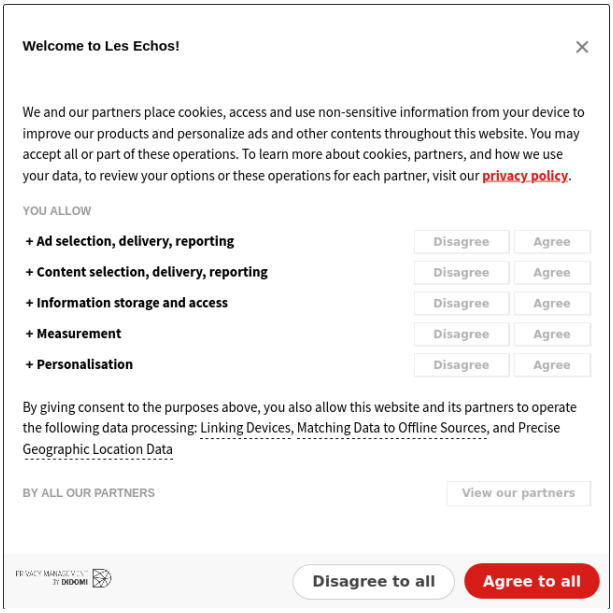
Alors que sur d'autres sites, ce n'est pas la même affaire…



Mais où est le bouton « Tout refuser » ?

Pour arriver à tout refuser, il faut cliquer sur « Learn more », qui nous propose alors ceci :

Toutes les utilisations prévues (on saluera la présence de « Disagree to all »)



Il s'agit de « dark pattern ». C'est à dire que l'interface est conçue dans le but d'être complexe, fatigante ou bien trompeuse.

Ce genre de pratique s'appuie sur différents biais ; ce qui pousse l'utilisateur à accepter plus que nécessaire. Que ce soit par distraction (un bouton mis en avant plus qu'un autre), par un changement de code classique (une couleur rouge utilisée pour une action d'acceptation), par obfuscation d'un processus (devoir parcourir plusieurs pages pour faire l'action voulue), et bien d'autres encore.

Mais alors que les choix des utilisateurs peuvent être biaisés par des interfaces, peut-on considérer un consentement comme étant libre et éclairé ? Pas vraiment.

Bien que l'on puisse se dire qu'un consentement est forcément biaisé par nature, ne pourrait-on pas créer de nouvelles solutions qui permettent aux utilisateurs de mieux comprendre ce qu'entraîne un clic sur un bouton « Accepter » afin de rendre le consentement plus libre et éclairé ?

Les Do et Don’t des bandeaux à cookies

L’usage de certains cookies requiert de recueillir le consentement des utilisateurs. Comment avoir un bandeau à cookies rgpd-friendly ?

Quand faut-il recueillir le consentement ?

Les cookies sont des petits fichiers qui sont déposés sur l’ordinateur (ou tablette, téléphone…) lorsqu’un utilisateur navigue sur un site. Les cookies permettent de stocker des informations en vue d’une réutilisation future.

Les cookies peuvent autant servir à se souvenir des préférences de l’utilisateur, par exemple pour la personnalisation de l’interface du service, qu’à analyser et tracer la navigation de l’utilisateur.

Les premiers ne nécessitent pas de recueillir le consentement des utilisateurs, même s’il est toujours bon d’informer de leur utilisation, mais les deuxièmes oui.

En effet, dès lors qu’un cookie est utilisé pour afficher de la publicité personnalisée, suivre l’utilisateur sur d’autres services, générer des statistiques non-anonymes etc, il est obligatoire d’avoir le consentement de l’utilisateur pour déposer et utiliser ces cookies.

Comment recueillir le consentement ?

Pour recueillir un consentement valide, cinq choses sont à faire :

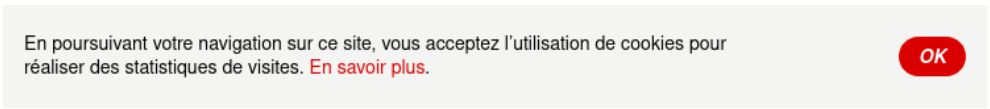
1. informer l'utilisateur. À quoi sert ce cookie ? Qui est responsable du traitement de données ?
2. permettre à l'utilisateur de donner son consentement par un acte clair
3. laisser l'utilisateur choisir par finalité. Si un cookie sert pour deux traitements différents, l'utilisateur doit pouvoir exprimer son accord pour un seul des traitements, les deux ou aucun
4. faire en sorte que les différents choix puissent s'exercer avec la même simplicité
5. permetter à l'utilisateur de revenir sur son choix.

Pourquoi ces bandeaux ne sont pas bons ?

Certains bandeaux à cookies se retrouvent souvent sur différents sites, mais ils présentent certaines erreurs sur le recueil du consentement. Mais qu'est ce qui ne va pas ?

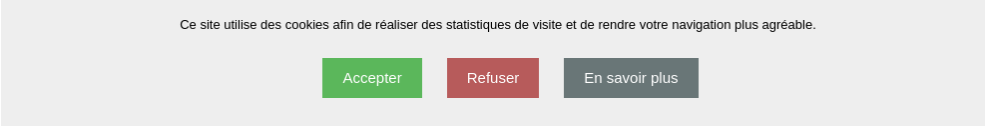
On utilise des cookies, ok ?

On utilise des cookies, ok ?



Ce type de bandeau où l'utilisateur est juste informé de l'utilisation de cookies n'est possible que si uniquement des cookies exempts de consentement sont utilisés. Ce qui est très rarement le cas.

All in One



All in one

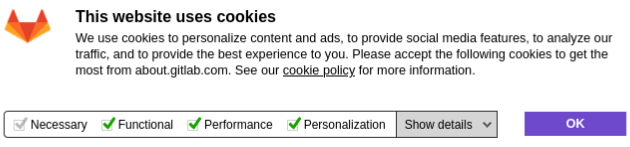
Ce bandeau a le mérite de proposer le refus et l'acceptation au même niveau et sans discrimination pour le refus. Mais il ne permet pas de choisir pour chaque finalité différente.

L'enfer des clics

L'enfer des clics.

Vous pouvez cliquer sur « Accepter & Fermer ». Après vous pouvez toujours tout refuser pour tous les partenaires. Ça ne vous demandera que 17 clics. Simple, non ?

Presque



Presque.

L'information est relativement claire, on peut avoir le détail de l'utilité de chaque cookie facilement. Mais… tout est précoché.

Quels outils pour gérer les bandeaux ?

Recueillir un consentement valide demande donc un minimum de rigueur.

Vous pouvez créer vous-même votre propre bandeau à cookies en suivant les recommandations de la CNIL.

Mais il y a aussi des outils pour vous aider, comme par exemple Tarte au Citron ou Cookiebot.

Anonymisation vs. pseudonymisation

L'anonymisation et la pseudonymisation sont des concepts qui reviennent fréquemment sur les sujets liés aux données personnelles collectées et utilisées par diverses organisations et dans divers buts.

Ces concepts trouvent leur utilité dans différents domaines : par exemple en informatique, où ils permettent de répondre à des problématiques de sécurisation de données ; ou alors encore pour de la publication de données, comme pour l'ouverture des données publiques pour laquelle les éléments à caractère personnel sont à éclipser.

La différence entre l'anonymisation et la pseudonymisation est que pour la première le processus est irréversible, tandis que pour la deuxième on peut revenir en arrière.

Anonymisation

Pour qu'un ensemble de données soit considéré comme anonymisé, il faut qu'il valide trois points :

- l'individualisation : on ne peut pas isoler une personne dans cet ensemble ;
- la corrélation : on ne peut pas retrouver les informations personnelles en croisant l'ensemble anonymisé avec un autre ensemble ;
- l'inférence : on ne peut pas déduire des nouvelles informations sur une personne.

Il est impossible de retrouver des informations personnelles à partir d'un ensemble de données anonymisé. Cela peut être intéressant pour, par exemple, garder des données après leur date limite de conservation.

Pseudonymisation

De l'autre côté, avec la pseudonymisation, il est possible de retrouver des informations en les croisant avec d'autres informations supplémentaires.

En effet, la pseudonymisation ne fait que remplacer les données qui permettent d'identifier directement une personne. Par exemple, pour un ensemble de données contenant des noms et des adresses, les noms peuvent être remplacés par des chaînes aléatoires tandis que les adresses sont laissées telles quelles. À partir de l'ensemble pseudonymisé, on ne peut pas retrouver les noms des personnes, mais avec la table de correspondance entre les noms et les chaînes aléatoires, c'est possible.

La pseudonymisation est recommandée par le RGPD pour limiter les risques liés au traitement de données personnelles.

Un exemple

Sur une liste de noms associés à un montant de commandes, une anonymisation et une pseudonymisation pourraient donner quelque chose comme :

Originel	Anonymisé	Pseudonymisé
François Dupont - 65€	xxxx - 65€	TDSF - 65€
Francine Michu - 43€	xxxx - 43€	UMEF - 43€
Joseph Clinton - 28€	xxxx - 28€	NCHJ - 28€
François Dupont - 12€	xxxx - 12€	TDSF - 12€
Julie Fernand - 74€	xxxx - 74€	DFEJ - 74€

Avec les données anonymisées, on peut seulement savoir qu'il y a 5 commandes dans le jeu de données originel et on peut calculer un montant moyen de commandes global.

Avec la pseudonymisation, on sait en plus que deux des entrées concernent une même personne, ce qui permet d'avoir d'autres statistiques comme le montant moyen de commande par personne.

La pseudonymisation permet de conserver plus d'informations sur les données sans en connaître le contenu exact. Cependant, il suffit d'avoir accès à la table de correspondance entre les données originelles et les données pseudonymisées pour retrouver les données initiales ; ce que ne permet pas l'anonymisation.

L'anonymisation et la pseudonymisation sont donc deux processus distincts, qui répondent tous deux à la problématique d'utilisation de données personnelles tout en respectant les droits des personnes. Les cas d'usages ne sont pas forcément identiques. Par exemple, l'anonymisation permet de conserver des données plus longtemps que la durée définie alors que la pseudonymisation ne le permet pas.

Derrière la pseudonymisation

Et si nous regardions plus en détail ce qui se cache derrière la pseudonymisation ?

Dans quels scénarios ?

La pseudonymisation est amenée à être un élément important dans le cadre du RGPD en tant que mesure de sécurité. En effet, elle permet de protéger, dans une certaine mesure, l'identité des personnes. Elle permet également de minimiser le nombre de données personnelles traitées quand il n'est pas nécessaire de connaître la vraie identité des utilisateurs pour un traitement particulier.

La pseudonymisation peut être utilisée dans différents scénarios dépendants des buts des organismes prenant part aux traitements comme :

- pour un usage interne à une entité : améliorer la sécurité des données lors du partage des informations entre différents services et en cas d'incident relatif à la sécurité de manière générale ;
- pour le partage avec une entité tierce : les données sont collectées par un organisme qui fait appel à un autre organisme, par exemple pour des analyses statistiques sur l'usage d'un service ;
- dans l'exploitation de données depuis une entité tierce : les données sont collectées et pseudonymisées par un autre organisme qui va ensuite les partager avec l'organisme principal, par exemple pour de la mesure d'audience.

Quelles techniques ?

Pour pseudonymiser une donnée, il suffit de lui appliquer une fonction qui la transforme. Pour deux données distinctes, la fonction de pseudonymisation renvoie deux pseudonymes distincts. Pour cela, différentes méthodes existent (liste non exhaustive) :

Le compteur
: La donnée à pseudonymiser est simplement associée à sa position dans la liste des données à traiter. La première donnée est associée à l'identifiant 1, la deuxième à l'identifiant 2, la troisième à l'identifiant 3, etc.

Le générateur de nombre aléatoire
: Comme pour le compteur, une donnée est associée à un nombre. Mais ce coup-ci, le nombre est généré aléatoirement ; ce qui présente l'intérêt de ne pas dévoiler l'ordre des données.

La fonction de hachage cryptographique
: À partir de données de différentes tailles, cette fonction renvoie de nouvelles données de même taille. L'avantage de cette méthode est qu'elle permet de s'assurer, par définition, qu'une donnée a un pseudonyme unique. Quelques exemples de fonction de hachage : MD5, SHA-1, SHA-2…

Le code d'authentification de message (MAC)
: Cette méthode est similaire à la précédente. La différence est que pour transformer une donnée, une clé secrète est utilisée en plus de la fonction de hachage ; ce qui rend la méthode plus robuste, tant que la clé secrète est toujours secrète.

Un petit exemple

Voici un exemple avec des adresses mail des différentes méthodes de pseudonymisation. Pour le hachage et le code d'authentification de message, l'algorithme MD5 a été utilisé.

Adresses mail	Compteur	Nombre aléatoire	Hachage	MAC
francois@dupont.fr	1	12	466123c…	bd9de45…
francine@michu.fr	2	51	86a313d…	da93151…
joseph@clinton.fr	3	42	0d2d86c…	cca373a…
julie@fernand.fr	4	66	c9bb585…	c5e6177…

L'adresse a été traitée en un seul bloc, mais on peut appliquer chaque méthode sur une partie de l'adresse pour conserver l'information sur la nature de la donnée. Ce qui donnerait quelque chose comme `12@666` pour la méthode du nombre aléatoire.

Les limites de la pseudonymisation

Des personnes ou des entités peuvent avoir un intérêt à casser la pseudonymisation d'un ensemble de données ; que ce soit pour trouver la méthode de pseudonymisation utilisée, ré-identifier les données ou isoler une information précise de l'ensemble.

Pour cela, différentes techniques peuvent être mises en œuvre, suivant l'objectif visé, les connaissances sur l'ensemble de données, la taille de l'ensemble, etc. On peut par exemple utiliser l'attaque par force brute, la recherche par dictionnaire ou encore par conjectures.

Même si la pseudonymisation est cassable, sa plus grande limite est qu'il est facile de lier des données entre elles, même lorsqu'elles sont pseudonymisées et d'en tirer des informations.

Un exemple

Une sociéte met en ligne un service sur lequel les personnes peuvent s'inscrire. Cette société fait appel à un prestataire pour gérer la sécurité de son service et de ses accès.

Pour permetter au prestataire d'accomplir sa mission, la société lui transmet les fichiers contenant les adresses IP pseudonymisées et les heures d'accès de ces IPs sur le service.

Même si le prestataire n'arrive pas à casser la pseudonymisation des adresses IP, il peut avoir d'assez bonnes estimations sur l'utilisation du service avec les métriques suivantes par exemple :

- le nombre d'utilisateurs uniques ;
- le nombre de nouveaux inscrits ;
- le nombre d'utilisateurs qui reviennent sur le service.

Ces métriques reposant sur les adresses IP, il peut y avoir des imprecisions. Mais les IPs changeant peu, les estimations sont valables.

Le prestataire n'a pas d'intérêt à connaître la véritable identité des utilisateurs, mais les informations qu'il est capable d'extraire des données, même pseudonymisées, lui donnent un avantage au niveau des accords et négociations avec la société cliente.

La pseudonymisation reste donc une mesure de sécurité utile lorsque l'on traite des données personnelles. Mais elle possède des limites car elle laisse transparaître beaucoup d'informations sur les données, même si la méthode utilisée pour les pseudonymiser n'est pas partagée. La pseudonymisation toute seule ne constitue pas une réelle sécurité mais fait partie d'un ensemble de mesures ayant pour but un certain niveau de sécurité.

Pour aller plus loin sur le sujet, un guide sur les techniques de pseudonymisation réalisé par l'ENISA est disponible.

Sous le masque de l'anonymisation

Qu'est ce que c'est ?

L'anonymisation d'un ensemble de données est un procédé qui a pour but de rendre impossible l'identification des personnes auxquelles les données sont rattachées. C'est une opération qui est donc irréversible. Une fois l'anonymisation effective, les données ne sont plus soumises au RGPD car elles ne sont plus considérées comme des données personnelles.

Un ensemble de données est considéré anonymisé si il est insensible à trois points :

- l'individualisation : obtenir des informations sur une personne précise ;
- la corrélation : retrouver des données identifiantes en croisant l'ensemble anonymisé avec un autre ensemble ;
- l'inférence : déduire des informations sur une personne.

Comment faire pour anonymiser ?

Pour anonymiser un ensemble de données, il est préférable de supprimer les données qui permettent d'identifier directement les personnes et les données qu'il n'est pas nécessaire de conserver. Il faut également choisir quelles techniques vont être mises en place et le degré de précision que l'on souhaite.

Il existe deux grandes familles dans les techniques d'anonymisation : la randomisation et la généralisation.

La randomisation modifie les valeurs des données pour en réduire leur précision, mais en conservant la répartition globale. Cela permet de protéger les données du risque d'inférence.

La généralisation, quant à elle, généralise les valeurs des données pour que des données soient identiques pour plusieurs personnes. Grâce à cela, les risques de corrélation et d'individualisation sont limités.

La randomisation

Ajout de bruit

L'ajout de bruit consiste à modifier la valeur de certains attributs de l'ensemble de données. On peut également rajouter de fausses données. Dans les deux cas, on essaye de conserver au mieux la distribution générale et on supprime au préalable les données quasi-identifiantes.

Par exemple, l'ensemble de données suivant :

Nom	Âge	Myope (1 = oui, 0 = non)
Clara	28	1
Jean	33	0
Julie	57	0
Lionel	42	1

peut ressembler à ça une fois du bruit ajouté :

Nom	Âge	Myope (1 = oui, 0 = non)
*	23	1
*	38	0
*	52	0
*	47	1
*	40	0
*	47	1
*	33	0
*	40	1

Les prénoms ont été supprimés, car ils sont quasi-identifiants. Quatres lignes ont été rajoutées et les âges des « vraies » lignes ont été modifiés avec un écart de plus ou moins 5. Les données sont moins précises que celles originales, mais la moyenne d'âge du groupe entier a été conservée ainsi que le ratio de personnes myopes.

Cependant, le bruit aurait pu être rajouté de façon à conserver d'autres propriétés de l'ensemble de données ; par exemple en tenant compte du nombre de personnes myopes par tranche d'âge. Tout dépend de quelles informations on souhaite conserver.

Permutation

La permutation, comme son nom l'indique, consiste à permuter les valeurs des attributs entre eux. Les valeurs des attributs quasi-identifiants sont là aussi supprimées.

Sur l'exemple précédent, une permutation pourrait donner ceci :

Nom	Âge	Myope (1 = oui, 0 = non)
*	33	1
*	28	0
*	42	0
*	57	1

Si la permutation permet de garder des informations sur l'ensemble des données, elle ne permet cependant pas de conserver des informations sur les sous-ensembles de données.

Confidentialité différentielle

La confidentialité différentielle est un indicateur qui est utilisé lorsque l'on met à disposition d'un tiers des sous-ensembles de l'ensemble données ; par exemple pour des statistiques.

Lorsque le tiers demande un sous-ensemble, il faut que l'algorithme utilisé pour construire ce sous-ensemble ne permette pas de déduire des informations sur des individus en particulier.

Sur notre exemple, si un tiers a à disposition un algorithme lui retournant la somme partielle des X premières ligne de l'ensemble et que ce tiers sait que Julie est la troisième personne de la liste, il peut facilement savoir si Julie est myope ou non.

En effet, il lui suffit de demander la somme pour les trois premières lignes et celle pour les deux premières. Ainsi, il se retrouve avec les valeurs `1` et `1`. Une rapide soustraction lui permet d'obtenir `0` et donc de savoir que Julie n'est pas myope.

De ce fait, cet algorithme n'est pas confidentiellement différentiel.

Pour pallier les risques de ré-identification, les algorithmes doivent être vérifiés et il est bon de rajouter du bruit sur les sous-ensembles partagés.

Attention, par ailleurs, tant que l'ensemble originel existe, les sous-ensembles générés sont considérés comme des données personnelles et sont donc soumis aux règlementations.

La généralisation

L'agrégation et le k-anonymat

Pour généraliser des données, on les agrège entre elles suivant le niveau de précision dont on a besoin.

Le k-anonymat permet, en quelque sorte, de mesurer la qualité de l'anonymisation induite par l'agrégation.

Par exemple, pour cet ensemble de données :

Nom	Localisation	Âge	Maladie
Julie	47677	29	Diabète
Joël	47602	22	Diabète
Stéphanie	47678	27	Diabète
Amanda	47905	43	Grippe
Gabriel	47909	52	Diabète
Léo	47906	47	Cancer
Sacha	47605	30	Diabète
Aurélie	47673	36	Cancer
Paco	47607	32	Cancer

une généralisation peut aboutir à :

Nom	Localisation	Âge	Maladie
*	476**	2*	Diabète
*	476**	2*	Diabète
*	476**	2*	Diabète
*	4790*	< 40	Grippe
*	4790*	< 40	Diabète
*	4790*	< 40	Cancer
*	47***	3*	Diabète
*	47***	3*	Cancer
*	47***	3*	Cancer

Cette généralisation est 3-anonyme car chaque combinaison `localisation + âge` apparait trois fois.

Cependant, cette généralisation n'est pas suffisante. En effet, si l'on sait que Julie est dans cet ensemble de données et que l'on sait qu'elle se trouve dans la ville 47677 et qu'elle a une vingtaine d'années, on peut en déduire qu'elle est diabétique.

l-diversité

Pour améliorer la généralisation, un autre indicateur a vu le jour : la l-diversité.

Cette mesure prend en compte le fait que pour chaque classe d'équivalence, les valeurs des données sensibles ne soient pas uniques, contrairement à précédemment où il était possible de connaitre la maladie de Julie car c'était la seule présente dans le sous-ensemble correspondant à ses caractéristiques.

Par exemple, avec cet ensemble de données :

Nom	Localisation	Âge	Maladie	Salaire
Julie	47677	29	Ulcère gastrique	3000
Joël	47602	22	Gastrite	4000
Stéphanie	47678	27	Ulcère gastrique	5000
Amanda	47905	43	Gastrite	6000
Gabriel	47909	52	Grippe	11000
Léo	47906	47	Bronchite	8000
Sacha	47605	30	Brochite	7000
Aurélie	47673	36	Pneumonie	9000
Paco	47607	32	Cancer de l'estomac	10000

une généralisation peut donner :

Nom	Localisation	Âge	Maladie	Salaire
*	476**	2*	Ulcère gastrique	3000
*	476**	2*	Gastrite	4000
*	476**	2*	Cancer de l'estomac	5000
*	4790*	> 40	Gastrite	6000
*	4790*	> 40	Grippe	11000
*	4790*	> 40	Bronchite	8000
*	476**	3*	Bronchite	7000
*	476**	3*	Pneumonie	9000
*	476**	3*	Cancer de l'estomac	10000

Cette généralisation est 3-diverse car pour chaque combinaison `localisation + âge`, il y a au moins trois valeurs de maladie (et de salaire) différentes. Cependant, sans connaître d'information particulière sur Julie mais uniquement en sachant que ses données sont dans le groupe correspondant à l'une des trois premières lignes, une personne peut de nouveau savoir que Julie a une maladie au niveau de l'estomac, et aussi avoir un ordre d'idée sur son salaire.

t-proximité

Pour de nouveau améliorer la généralisation, un autre indicateur a lui aussi vu le jour : la t-proximité.

Cet indicateur mesure le taux de diversité des valeurs des données sensibles pour chaque classe d'équivalence, afin de réduire les risques d'inférence.

Une version avec un anonymat encore plus solide que précédemment de l'exemple peut être :

Nom	Localisation	Âge	Maladie	Salaire
*	4767*	< 40	Ulcère gastrique	3000
*	4767*	< 40	Cancer de l'estomac	5000
*	4767*	< 40	Pneumonie	9000
*	4790*	> 40	Gastrite	6000
*	4790*	> 40	Grippe	11000
*	4790*	> 40	Bronchite	8000
*	4760*	< 40	Gastrite	4000
*	4760*	< 40	Bronchite	7000
*	4760*	< 40	Cancer de l'estomac	10000

La généralisation plus large au niveau de l'âge permet d'éviter l'exploitation de la corrélation, par exemple, entre les personnes de vingt ans et les maladies de l'estomac.

Même si quelqu'un sait que les données de Julie sont dans le premier groupe (correspondant aux premières lignes), il ne pourra pas savoir quel type de maladie a Julie, ni dans quelle fourchette précise se situe son salaire.

Cet ensemble anonymisé a comme indicateurs 0.167-proximité pour les salaires et 0.278-proximité pour les maladies. Alors que sur l'ensemble anonymisé de la l-diversité, les t-proximités étaient de 0.375 pour les salaires et de 0.5 pour les maladies. Une meilleure agrégation a donc permis une meilleure anonymisation. Si vous aimez les grosses formules et que vous voulez savoir comment calculer ces valeurs, vous trouverez votre bonheur dans ce papier qui définit la t-proximité (c'est en anglais).

L'anonymisation offre donc la possibilité de conserver des données plus longtemps que prévu initialement de part la perte de leur nature personnelles. C'est donc un processus qui peut facilement séduire.

Cependant, mettre en place une réelle anonymisation est complexe. En effet, il faut choisir les bonnes techniques et les appliquer du mieux possible. Mais il faut aussi toujours se tenir informer des failles découvertes et des nouveautés sur le sujet.

Et même lorsque l'anonymisation semble être solide, il est possible de ré-identifier les données. Comme cela a été le cas pour Netflix lorsque des chercheurs ont pu ré-identifier des données pourtant anonymisées en les croisant avec IMDb.

Le meilleur moyen de réduire les risques reste peut-être de ne pas chercher à tout stocker tout le temps… ;)

Intégration continue

Déployer un site statique avec GitLabCI

Un site statique permet de présenter un même contenu fixe à tous les visiteurs. Il est généralement utilisé pour présenter un produit ou une marque et est plutôt simple à mettre en ligne. En effet, il suffit de mettre les fichiers sur un serveur et de les rendre accessibles via un serveur web, comme Nginx par exemple.

Une fois créé et mis en ligne, il arrive un moment où l'on va vouloir modifier ce site. On va alors effectuer les changements sur les fichiers concernés, puis il faut penser à aller les mettre à jour sur le serveur.

Et cette mise à jour peut se révéler fastidieuse. Ce n'est pas particulièrement intéressant et on risque d'oublier des choses. C'est pourquoi, utiliser un outil, comme GitLabCI, pour faire cette étape à notre place se révèle utile.

Ingrédients

- un projet sur GitLab qui contient les fichiers sources
- un accès au serveur cible

Préparation

Étape 1

Sur le serveur, générer une clé ssh en utilisant la commande `ssh-keygen`.

Étape 2

Sur GitLab, dans les paramètres pour l'intégration continue, rajouter une variable qui contient le contenu de la clé privée générée à l'étape précédente.

Étape 3

Dans le projet, créez un fichier `.gitlab-ci.yml`. Ce fichier va permettre d'indiquer les actions à effectuer pour déployer le site.

```
deploy:
  image: ubuntu
  before_script:
    ## Installation du ssh-agent, nécessaire pour Docker
    - 'which ssh-agent || ( apt update -y &&
      apt install openssh-client -y )'
    ## Lancement du ssh-agent
    - eval $(ssh-agent -s)
    ## Ajout de la clé ssh stockée à l'étape 2
    - echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -
    ## Création du dossier ssh et mise des permissions
    - mkdir -p ~/.ssh
    - chmod 700 ~/.ssh
    ## Scan des clés du serveur cible
    - ssh-keyscan -p <port> <server_url> ~/.ssh/known_hosts
    - chmod 644 ~/.ssh/known_hosts

  script:
    ## Connection au serveur et mise à jour du dossier du projet
    - ssh <user>@<server_url> -p <port> \
      "cd <path_to_project_directory> && git pull && exit"
```

Étape 4

Mettre `gitlab.com` dans la liste des hôtes connus sur le serveur cible avec la commande `ssh-keyscan -t rsa gitlab.com >>> .ssh/known_hosts`.

Mettre la clé publique créée à l'étape 2 dans la liste des clés autorisées dans `.ssh/authorized_keys`.

Et voilà, le site sera désormais mis à jour à chaque nouvelle modification !

Passer de Travis CI à GitHub Actions

En dehors de Stella, on s'occupe de projets libres (comme WeasyPrint par exemple). Et bien sûr, ils ont des tests ! Après avoir utilisé Travis CI, nous avons décidé de passer sur GitHub Actions. En voici un petit retour d'expérience :).

Pourquoi Travis ?

De nombreux services proposent de l'intégration continue, peu importe la plateforme où sont les dépôts des projets (dans notre cas, c'est GitHub). Parmi ces services, on retrouve Travis CI, GitLab CI ou encore GitHub Actions.

Au début, Travis a été choisi tout simplement parce que c'était le seul service largement utilisé qui existait et qui était gratuit, au moins pour les projets libres.

Pourquoi changer ?

Travis offre des fonctionnalités plutôt pratiques, comme par exemple de pouvoir lancer les tests sur MacOS et Windows. C'est plutôt bien et ce n'est pas forcément le cas de tous les services de CI.

La configuration se fait assez simplement avec un fichier `.travis.yml` à mettre à la racine du projet.

Par contre, Python ne fait pas partie des langages supportés pour Windows et MacOS, ce qui est un peu dommage quand on a des projets en Python. Malgré cela, on peut quand même s'en sortir.

Mais surtout… c'est lent.

Changer, oui. Mais pour quoi ?

Alors oui c'est lent, mais il nous fallait un service qui dispose de fonctionnalités au moins équivalentes. Hors de question de perdre les tests sous Windows qui avaient été enfin mis !

Dans les choix possibles, nous avions GitLab CI et GitHub Actions qui proposent un service gratuit sans limite de minutes d'utilisation trop restrictives pour notre utilisation.

Malgré notre grand amour pour GitLab, il n'est pas possible de lancer des tests sous MacOS et la fonctionnalité est en bêta pour Windows…

Alors oui, il est possible d'installer soi-même des runners sur ses machines pour avoir tout ça. Mais quand on n'a pas de Windows ou de Mac sous la main… c'est de suite plus compliqué. (Note : il y a des runners partagés pour Windows.)

Et avec GitHub Actions, on peut lancer des tests sur Windows et MacOS. C'est fantastique, voici l'élu !

GitHub Actions en action

De la même manière que Travis, la configuration se fait par un fichier YAML à mettre dans un dossier à la racine du dépôt. On peut y mettre plusieurs fichiers si l'on veut faire différentes choses et bien les séparer (mais nous, on veut juste lancer les tests, pour l'instant).

Dans un fichier de configuration GitHub Actions, chaque ligne qui est exécutée est une Action. Et GitHub Actions dispose d'Actions déjà toutes faites qui nécessitent juste d'être appelées (et tout le monde peut proposer les siennes aussi :)). Ainsi une instance « vide » est facilement customisable pour préparer tout ce dont on a besoin pour lancer nos tests.

Dans ce large univers d'Actions, il y en a une qui nous intéresse tout particulièrement : setup-python. C'est elle qui va installer le minimum requis pour faire des choses avec du Python. Du genre : Python et pip par exemple, très pratique. Pour Linux, MacOS et Windows. Avec la version de Python souhaitée. Vraiment très pratique.

Après, il n'y a rien de bien différent. On peut faire des matrices pour gérer les combinaisons de versions et d'OS.

Un petit comparatif peut-être ?

Pour un projet plutôt basique en Python, nous avions ce fichier pour Travis :

```
language: python
env: PYTHON=python3

jobs:
  include:
    - python: 3.6
    - python: 3.7
    - python: 3.8
    - python: pypy3
    - os: osx
      language: generic
    - os: windows
      language: shell
      env: PYTHON=/c/Python38/python

install:
  - if [[ "$TRAVIS_OS_NAME" == "windows" ]]; then \
    choco install -y python; fi
  - $PYTHON -m pip install --upgrade pip setuptools
  - $PYTHON -m pip install .[test]

script:
  - $PYTHON -m pytest --isort --flake8 \
    --ignore=tests/css-parsing-tests
```

Ce qui lance des tests sur Linux pour quatre différentes versions de Python, ainsi que sur MacOS et Windows pour une seule version. Soit au total six environnements de test. Travis met entre `2 min 30` et `15 min` pour tout faire.

Pour GitHub Actions, nous avons à présent ce fichier :

```yaml
name: tinycss2's tests
on: [push]

jobs:
  tests:
    name: ${{ matrix.os }} - ${{ matrix.python-version }}
    runs-on: ${{ matrix.os }}
    strategy:
      # En cas d'échec sur un environnement,
      # les autres ne sont pas interrompus
      fail-fast: false
      matrix:
        os: [ubuntu-latest, macos-latest, windows-latest]
        python-version: [3.6, 3.7, 3.8, pypy3]
        # On peut exclure des combinaisons de la matrice
        exclude:
          # pytest-isort not working
          - os: windows-latest
            python-version: pypy3
    steps:
      # Actions qui clone le dépôt
      - uses: actions/checkout@v2
        # Ce projet utilise des sous-modules
        with:
          submodules: true
      # L'environnement python est préparé pour
      # les versions définies dans la matrice
      - uses: actions/setup-python@v2
        with:
          python-version: ${{ matrix.python-version }}
      - name: Upgrade pip and setuptools
        run: python -m pip install --upgrade pip setuptools
      - name: Install tests's requirements
        run: python -m pip install .[test]
      - name: Launch tests
        run: python -m pytest --isort --flake8 \
          --ignore=tests/css-parsing-tests
```

Ce sont onze environnements de test qui sont lancés, pour un temps d'exécution entre 1 minute 20 et 4 minutes.

GitHub Actions est bien plus rapide que Travis et nous permet de couvrir plus d'environnements différents !

Pour aller plus loin

Nous avons pris en exemple un projet qui ne nécessite que très peu de mise en place.

Pour des projets où l'on veut exécuter des choses uniquement sur certains environnements, GitHub Actions offre une configuration plus élégante.

Par exemple, pour WeasyPrint avec Travis nous avions :

```
- if [["$TRAVIS_OS_NAME" == "windows" ]]; then \
  export "PATH=$PATH;C:\msys64\mingw64\bin"; fi
- if [[ "$TRAVIS_OS_NAME" == "windows" ]]; then \
  choco install -y python dejavufonts; fi
- if [[ "$TRAVIS_OS_NAME" == "windows" ]]; then \
  wget "http://repo.msys2.org/distrib/x86_64/\
  msys2-base-x86_64-20180531.tar.xz"; fi
- if [[ "$TRAVIS_OS_NAME" == "windows" ]]; then \
  7z e msys2-base-x86_64-20180531.tar.xz; fi
- if [[ "$TRAVIS_OS_NAME" == "windows" ]]; then \
  7z x -y msys2-base-x86_64-20180531.tar -oc:\\; fi
- if [[ "$TRAVIS_OS_NAME" == "windows" ]]; then \
  powershell "c:\msys64\usr\bin\bash -lc 'pacman -s \
  mingw-w64-x86_64-gtk3 --noconfirm'"; fi

- if [[ "$TRAVIS_OS_NAME" == "osx" ]]; then \
  brew tap homebrew/cask-fonts; fi
- if [[ "$TRAVIS_OS_NAME" == "osx" ]]; then \
  brew cask install font-dejavu; fi
- if [[ "$TRAVIS_OS_NAME" == "osx" ]]; then \
  brew install cairo pango gdk-pixbuf libffi; fi
```

Et maintenant avec GitHub Actions :

```
- name: Install DejaVu (Ubuntu)
  if: matrix.os == 'ubuntu-latest'
  run: sudo apt update -y && sudo apt install ttf-dejavu -y
- name: Install DejaVu, cairo, pango, gdk-pixbuf and libffi
  if: matrix.os == 'macos-latest'
  run: |
    brew tap homebrew/cask-fonts
    brew cask install font-dejavu
    brew install cairo pango gdk-pixbuf libffi
- name: Install msys2 and DejaVu (Windows)
  if: matrix.os == 'windows-latest'
  run: |
    choco install -y --no-progress msys2
    powershell "C:\tools\msys64\usr\bin\bash -lc \
      'pacman -S mingw-w64-x86_64-ttf-dejavu \
      mingw-w64-x86_64-gtk3 --noconfirm'"
    xcopy "C:\tools\msys64\mingw64\share\fonts\TTF"\
      "C:\Users\runneradmin\.fonts" /e /i
    $env:PATH += ";C:\tools\msys64\mingw64\bin"
    echo "::set-env name=PATH::$env:PATH"
```

Nous avons beaucoup parlé des environnements Python sur GitHub Actions, mais ce n'est pas le seul langage supporté. Il y a de nombreuses Actions pour plein de langages et pour faire plein de choses ;).

Logiciel libre

Au bout d'un projet libre

Lancer un projet libre ? Vous y avez forcément déjà pensé, et vous avez peut-être même déjà créé un dépôt avec un peu de code dedans. Évidemment, après quelques semaines d'une passion fiévreuse, il est un peu tombé dans l'oubli… Comment font tous ces gens qui y arrivent ?

Un projet libre, pour quoi faire ?

Créer un projet libre, avec des utilisateurs et une petite communauté, c'est difficile. On a beau mettre toute son énergie et toute son envie, on arrive rarement à ses fins.

Déjà… Arriver à dépasser l'esquisse, le fichier de quelques dizaines ou centaines de lignes, c'est une étape qui peut sembler inaccessible. Et puis, pour être parfaitement franc : démarrer, déjà, c'est dur.

Toutes les raisons sont bonnes pour ne pas aller au bout : ce logiciel est nul, il ne sert à rien, je ne sais pas coder, les autres codent mieux que moi, je n'ai jamais fait de libre, je n'ai pas de temps, j'ai déjà tout ce qu'il me faut… On les connait, ces excuses, et pour cause : on les a toutes déjà utilisées personnellement.

L'envie

Là, c'est différent. Vous avez en vous l'envie d'aller au bout. Quelque part en vous, vous le sentez : c'est le moment.

L'envie de faire un projet libre, c'est un peu la condition sine qua non pour faire ce projet libre. Ça peut paraître évident, mais ça ne l'est pas forcément quand on y regarde de plus près. Il arrive souvent d'écrire du code parce qu'on y est contraint, que ce soit un petit script pour automatiser quelque chose sur son ordinateur ou pour corriger un

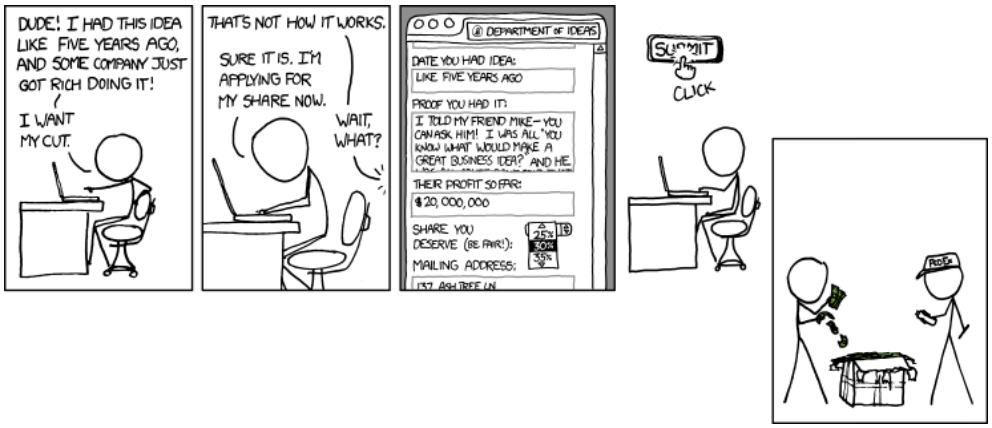
bug dans un programme qu'on utilise. Sous contrainte, ce n'est sans doute pas la peine de se lancer dans quelque chose de durable. Ça ne fonctionnera pas.

L'envie, c'est ce qu'il vous restera quand il ne vous restera plus rien. Et au début, on n'a souvent que cela. L'envie. Alors partez de cela, c'est déjà très bien.

L'envie, ce n'est ni la compétence, ni la légitimité, ni le besoin. L'envie, c'est quelque chose qui doit dépasser les peurs et les craintes. Elle n'a pas besoin de justification. Laissez donc ces doutes loin de votre tête, personne n'est là pour vous regarder pour l'instant. Cette arme est suffisante, vous n'avez pas besoin d'autre chose, et ceux qui vous diront le contraire seront des oiseaux de mauvais augure. Vous y compris.

Oui, c'est un travail qui peut être considérable pour certaines et pour certains. Mais ne vous inquiétez pas : nous sommes entre nous. Vous pouvez avoir confiance en vous.

L'idée



Le XKCD obligatoire. Vous avez eu toutes ces idées avant tout le monde. Un système d'exploitation libre ? Votre idée. Une suite bureautique libre ? Votre idée aussi. C'est insupportable, tous ces gens qui volent ce qui traîne dans votre tête.

L'idée, c'est la grande star du storytelling. « Je me suis réveillé un matin de juillet, et je me suis dit, en regardant la tranquille mer monégasque s'étendre à l'horizon, que créer mon entreprise de location de yachts en or massif était l'idée disruptive qui allait fonder ma fortune ». Gros plan sur la chevelure d'un trentenaire en chemise un peu ouverte. « Ce n'est pas donné à tout le monde d'avoir une idée brillante comme ça. C'est pour ça que je suis millionaire. Une idée

comme ça, c'est du travail, vous savez. Je me suis fait tout seul, sans personne, à part ma mère banquière d'affaires qui m'a filé un petit coup de main au début ».

Vous voulez un avis personnel tranché ? L'idée ne sert à rien. Vous n'avez pas d'idée de projet ? Tant mieux. On va faire sans.

Des projets qui manquent, à vous, à d'autres, il y en a des pelles, des montagnes, des torrents. Tous ces projets propriétaires qui n'existent pas en libre. Tous ces projets libres dont l'interface vous gonfle. Tous ces projets en ligne de commande qui seraient mieux avec une fenêtre et quelques boutons. Tous ces projets en ligne qui seraient mieux avec un client lourd. Tous ces clients lourds qui seraient mieux en ligne.

Et plus généralement : tous ces logiciels sur lesquels vous aimeriez changer deux ou trois trucs.

Bien sûr, vous pouvez également contribuer à des projets libres existants. Mais si vous sentez que vous feriez tout un peu autrement, si vous avez l'impression que l'organisation globale de cette interface mériterait d'être revue de fond en comble, en bref : si l'envie est trop forte, alors laissez-là vous envahir. Créez un projet.

Créer un projet, c'est souvent prendre les bonnes idées des autres et y ajouter un petit grain de sel personnel. Pas besoin de tout réinventer, parce qu'une bonne partie existe déjà. Vous pouvez même reprendre du code si la licence vous le permet. C'est fait pour. C'est pour ça que le libre est libre : il vous permet d'apprendre en copiant. Il n'y a pas de honte à cela.

Allez, on commence.

Le grand secret

Comment savoir si notre projet est un bon projet ? Pas facile… Pourtant, il existe une technique secrète qui peut vous donner quelques indications sur le degré de coolitude de votre création.

C'est un secret qui se transmet de génération en génération. Il reste à l'abri de l'air et de la lumière dans cette grande cave derrière de vieux grimoires que mes ancêtres, et leurs ancêtres avant eux, ont fait semblant de lire pour se donner un style dans les soirées mousse entre amis. Celles qu'on ne peut plus faire à cause de vous-savez-quoi. Mouais. Je vais peut-être lire tout ça, finalement.

Cette technique secrète, c'est de répondre à la question « Quand est-ce que vous utiliserez régulièrement votre projet ? » Si la réponse est « Dans pas très longtemps », alors vous partez sur une base carrément solide. Et sinon, il va falloir réfléchir un peu plus.

Pourquoi cette technique fonctionne ? Parce que si vous utilisez votre logiciel rapidement, vous vous assurez d'un seul coup :

1. que c'est utile ;
2. que vous aurez envie de l'améliorer, assez pour qu'il soit sympa à utiliser ;
3. que vous corrigerez les bugs les plus importants ;
4. que vous aurez un intérêt à le maintenir.

Toutes ces raisons-là sont des indicateurs assez pertinents du développement et de la longévité d'un projet.

Voilà, c'est tout.

Si votre réponse est un peu différente ? Alors là, ça dépend. Si vous pensez que vous l'utiliserez dans longtemps, parce que cela nécessite beaucoup, beaucoup de travail avant d'être utilisable, alors il vous faudra de la patience. Si vous avez peur d'utiliser un autre logiciel par facilité, même après avoir écrit un logiciel utilisable (mais moins bien), il vous faudra un peu de discipline. Si vous pensez que vous ne l'utiliserez qu'une fois, alors il n'a peut-être pas intérêt à devenir un projet libre utilisable par tous et maintenu.

Enfin, si vous ne l'utiliserez jamais, alors c'est autre chose qu'un projet libre. Ce peut être une bonne chose pour apprendre, pour se former, pour découvrir. Ce peut être un essai, une tentative qui finira vite dans la corbeille. Mais ça risque de ne jamais être un vrai projet libre.

L'aventure

Lancez-vous dans cette aventure dont vous êtes l'héroïne ou le héros. Pour toute aventure il y a les premiers pas, les premières lignes. Tous les logiciels ont commencé comme cela.

Ce n'est pas la peine de réfléchir à une analyse sociologique ou scientifique de votre projet. La tentation est grande, même après avoir passé avec succès l'épreuve ardue du grand secret, de trouver de bonnes raisons de se laisser gagner par la procrastination. Et pour cela, vous rencontrerez la tentation de questions plus métaphysiques les unes que les autres. Qui suis-je pour me permettre de créer ? Que vaux-je par rapport à toutes ces créations géniales ? À quoi sert-il de refaire un projet déjà fait mille fois ? Ma contribution sera-t-elle réellement déterminante dans l'histoire de l'humanité ? De l'univers ? D'ailleurs, l'univers existe-t-il réellement ?

Laissez les autres répondre à ces questions pour vous. Vous avez plus important à faire : créer un projet.

Ça y est. Vous avez envie de créer un dépôt et de vous jeter sur le code. Halte là ! Ça vaut le coup de se pencher sur quelques questions techniques auparavant.

Après tout, libre à vous de faire ce que vous voulez, les principes sont faits pour être bouleversés. Mais pour contourner les règles, autant commencer par les connaître !

Les mains pas dans le cambouis

En attendant le chat

Partons du principe (on ne prend pas trop de risques) que vous aimez programmer. Pour créer un programme, vous allez vous lancer sur votre éditeur favori (peu importe lequel, ce n'est pas le sujet), évidemment, bien évidemment.

Ce n'est pas forcément une bonne idée. Je vous assure.

J'adore passer du temps à écrire du code, des fois sans trop savoir ce que j'écris. J'aime ça. Bricoler la logique, bidouiller les variables, bâtir les algorithmes, empiler les fonctions. Mais ce n'est pas parce que j'aime ça que c'est forcément quelque chose de formidable. Après tout, j'en connais pas mal qui aiment indécemment enchaîner les tartines de chocolat, de pâté de campagne ou de fromage de chèvre. On aime, mais est-ce une raison suffisante ?

Si le but est de se lancer dans un marathon, de créer un logiciel vraiment utilisé par de vraies gens, alors mieux vaut ne pas gâcher tout le plaisir dès le début. En réfléchissant un peu, nous nous gardons une bonne dose de plaisir pour la suite. Croyez-moi, nous en aurons besoin.

Le projet libre est comme un petit chaton trop mignon. Au début, on veut lui faire des câlins tout le temps, on en prend soin, on lui donne toute notre attention. Le temps passant, les choses sont moins claires : bien sûr qu'on l'aime, même quand il nous réveille la nuit, même quand il fait pipi partout, même quand il nous griffe quand il n'aime pas ce qu'on lui donne gracieusement à manger.

Et puis un jour, on le voit pour ce qu'il est : un être égoïste (d'autres disent pudiquement « indépendant »), une machine à embaumer sa caisse, un monstre à dévorer des repas, un gouffre financier en

vétérinaire. On l'aime, hein, faut pas croire. Mais bon. Ça devient dur de s'en occuper, même un minimum. On changera la litière demain. Ou la semaine prochaine.

Vous saviez pourtant comment cela allait se terminer…

Alors mieux vaut prévoir un peu au début, histoire d'être content des objectifs atteints et de trouver une source de motivation avec des objectifs à long terme.

Objectif Lune

Fixer des objectifs, c'est une manière simple d'activer le circuit de récompense qui vous apportera joie, bonheur et prospérité. Alors mieux vaut en fixer une bonne tripotée dès le début, avec des premiers échelons facilement accessibles et très valorisants. Lorsqu'on les a, encore faut-il les écrire, histoire de s'en souvenir après et d'avoir une bonne idée du chemin déjà parcouru.

Pour obtenir rapidement quelque chose de valorisant (et manger quelques tartines pour fêter ça), il est plutôt intéressant de se lancer dans des premières étapes à la fois faciles, rapides et utilisables. Plus vite on aura un joujou agréable à utiliser, plus on sera motivé pour l'améliorer. Et quand on fait joujou, vous le savez bien, l'envie de voir les bugs corrigés et les améliorations s'empiler ne connaît aucune limite. L'utilisation motive le développement, même quand c'est la même personne qui incarne les deux rôles.

Le long terme est la seconde face d'une même médaille. Le court terme vous aide à atteindre le long terme, si tant est que vous savez à peu près où va ce long terme. On dit qu'il faut viser la Lune, et que si

l'on échoue on finira dans les étoiles. C'est très poétique (et un brin gnangnan), c'est aussi un petit peu vrai : viser loin vous donne du courage. La vision à long terme transforme les empileurs de pierres en constructeurs de cathédrales, juste par un astucieux (et parfaitement fallacieux) changement de point de vue.

Allez-y, mentez-vous. Allègrement. Vous voulez écrire un concurrent à Photoshop ? Aucun problème. Assurez-vous avant d'obtenir un petit logiciel pour faire du pixel art au bout de quelques centaines de lignes de code, histoire de vous transformer en utilisatrice ou utilisateur le plus tôt possible.

Les objectifs à court terme doivent prendre le pas sur les objectifs à long terme. Ce n'est pas la peine de lire des milliards de papiers de recherche sur l'optimisation des matrices de convolutions à n dimensions si vous n'avez pas déjà de quoi dessiner une image de quelques pixels à l'écran. Le long terme reste dans un coin de la tête, il peut influencer certains choix, mais il ne doit pas brider des objectifs à court terme facilement atteignables.

Il sera bien temps d'améliorer et de nettoyer du code qui fonctionne quand il fonctionnera et que vous l'utiliserez. Vous pouvez écrire des montagnes de joli code, mais si c'est une très belle coquille vide, vous ne trouverez jamais la motivation de remplir tout cet espace laissé vaquant.

On fait quoi ?

Parce que c'est notre projet

Nous avons vu que l'idée n'était pas un problème. Vous avez forcément un truc qui vous dérange, forcément un manque, une chose qui vous frustre, un vide à combler.

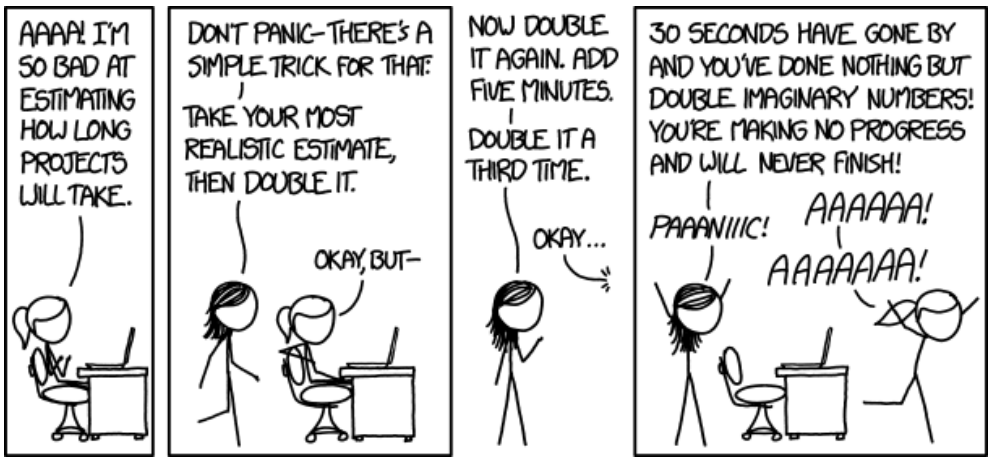
Prenez-le, quel qu'il soit. C'est votre projet. Mieux : c'est votre objectif à long terme. Ce logiciel parfait dont vous rêvez depuis si longtemps.

Pour moi, c'est un lecteur de flux RSS en ligne. Il y en a déjà plein, mais ils sont payants, ou avec un compte, ou surchargés et lents comme des grosses vaches, ou limités en nombre de flux. J'ai envie d'un truc clair et pas prise de tête.

Ce projet, vous n'allez sans doute pas le construire en peu de temps. Vous avez un marathon devant vous. Comme tous les marathons, on commence par les dix premiers mètres. C'est ridicule, quand on y pense, dix mètres, à l'échelle d'un marathon. C'est tellement ridicule qu'on pourrait être tenté d'abandonner tout de suite.

Que nenni. Votre premier objectif, c'est dix mètres. Pas un de plus. Après dix mètres : tartine. Et vous l'aurez bien méritée, parce que vous serez désormais plus loin que toutes celles et ceux qui auront déjà rendu leur tablier.

Au football, on dit qu'il faut jouer les matchs un par un. Quand on chasse, on dit qu'il ne faut pas courir plusieurs lièvres à la fois. C'est une source de moqueries évidentes, parce que ce serait quand même très étrange de jouer plusieurs matchs ou de courir après plusieurs animaux en même temps (#premierdegré, j'assume). Moquez-vous, moquez-vous. Mais quand votre tête vous dit que dessiner des pixels un par un, ce n'est pas la même chose que de faire des flous gaussiens dynamiques, vous rigolerez moins. Votre premier match, votre premier lièvre, c'est les pixels, et rien d'autre.



Le XKCD obligatoire. Dire que l'on joue les matches les uns après les autres, c'est déjà anticiper les matchs après le match suivant. La seule manière de vraiment jouer les matchs les uns après les autres, quand on y pense, c'est de ne pas y penser.

Écrivez vos objectifs. Ouvrez des tickets sur une forge logicielle, mettez des Post-it sur les murs de chez vous, faites une belle liste avec des cases à cocher, peu importe. Écrivez. Puis fermez les tickets, barrez les Post-it, cochez les cases au fur et à mesure. C'est votre

barre de progression, celle que vous regardez en mangeant vos tartines. Ce n'est pas la peine d'y penser quand vous développez, vous aurez à loisir de l'admirer pendant vos repos bien mérités.

Il est venu le temps des cathédrales

Dernier passage obligé avant d'ouvrir votre éditeur (enfin !) : l'architecture.

Vous allez construire une cathédrale, c'est sûr. Mais dans un premier temps vous allez plutôt empiler trois cubes en mousse, comme un enfant. Comme un enfant, ce sera une tâche compliquée. Vous avez l'impression que c'est très facile d'empiler trois cubes, mais prenez le temps. Réfléchissez.

Ces trois cubes en mousse ne seront sans doute pas dans la cathédrale finale. Pourtant, ils sont une étape importante. Définissez avec précision comment ils sont mis les uns sur les autres. Un cube sur deux autres, pour plus de stabilité ? Les trois cubes empilés, pour prendre de la hauteur ?

À ce moment-là du projet, allez là où vous prendrez le plus de plaisir à utiliser votre création. Ce n'est pas encore la peine de mettre en place des outils complexes ou des bonnes pratiques, ni de réfléchir pendant des semaines au matériau à utiliser. N'importe quelle mousse facile d'accès fera l'affaire pour empiler trois cubes. Les problèmes viendront, mais plus tard. Vous les réglerez en temps voulu.

Faites des dessins, des maquettes, des gribouillages. Plus vous savez ou vous allez, plus il sera facile de faire du code. Il est très dur, pendant qu'on code, de résoudre en parallèle des problèmes aussi complexes que de lister les fonctionnalités que l'on veut ou les interfaces que l'on désire. Faites ce travail avant, quand vous avez encore un peu de lucidité, avant de vous lancer dans le tunnel du développement. Ne visez pas loin, mais visez juste.

Lorsque vous aurez obtenu quelque chose de très excitant (même si ce n'est pas forcément beau, vous n'êtes pas nécessairement graphiste) et d'assez précis, c'est le moment tant attendu.

Du code, du code, DU COOOOOOOOOOOOOOODE.

Avant de se lancer

L'idée

Si vous avez bien suivi, vous devez avoir quelques idées en tête pour créer des applications. Choisissez-en une, définissez bien où vous voulez aller à long terme, et bichonnez le plan de vos premières briques.

Chacun ses passions. J'en ai plein, vraiment plein, mais pour aujourd'hui j'ai choisi d'en retenir deux : les flux RSS et les clients lourds. Ne jugez pas mes passions, je ne jugerai pas les vôtres, OK ?

Alors sans surprise, nous allons ici nous lancer dans la réalisation d'un lecteur de flux RSS. Étonnant, non ?

L'objectif

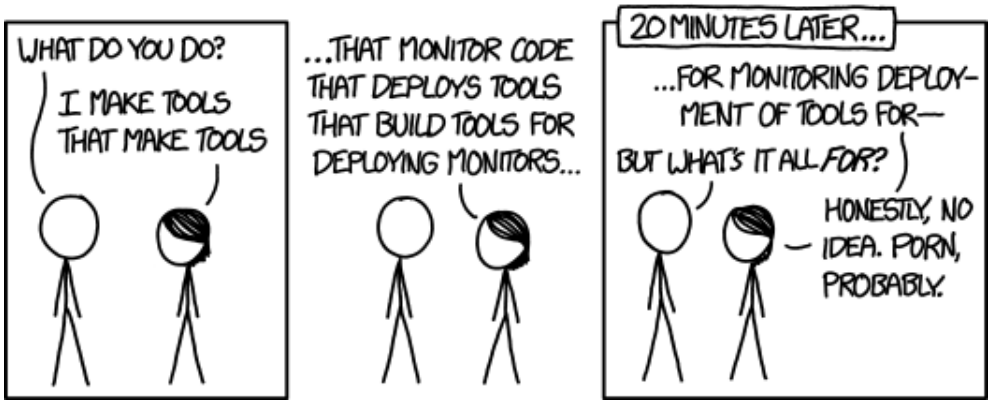
Nous ne ferons pas le lecteur RSS le plus puissant de tous les temps, en tout cas pas tout de suite. Nous allons plutôt nous atteler à faire un petit lecteur mignon, craquant, choupi, mais surtout… fonctionnel.

En effet, l'objectif principal de cette première mouture est d'avoir un programme à utiliser dans la vraie vie. Dans notre cas, notre lecteur affichera les articles par flux, et ouvrira les pages lorsque l'on cliquera dessus. Il ira chercher ses flux dans un fichier de configuration. Simple mais efficace.

D'un point de vue de l'interface, nous aurons un panneau à gauche avec la liste des flux, et à droite la liste des liens du flux sélectionné.

Les outils

Pour faire un petit lecteur de flux en Python, nous allons chercher des outils adaptés. C'est important, pour se simplifier la vie, de prendre le temps de trouver les modules qui nous permettront d'écrire moins de code et d'avoir de meilleures fonctionnalités.



Le XKCD obligatoire. Avant d'arriver à une orgie ingérable d'outils enchevêtrés, on va plutôt choisir quelques petits modules innocents.

Voici les principaux modules que nous utiliserons :

- GTK+3 pour l'interface,
- feedparser pour lire les flux,
- configparser pour le fichier de configuration.

Et c'est parti pour le show

Nous voilà embarqués dans un petit projet. Quelques minutes, quelques heures, quelques jours, cela dépendra de votre idée et de vos compétences. Mais plus vite vous aurez du code utilisé, plus vite vous trouverez la motivation de l'améliorer à long terme.

Vous pouvez lire le code de ce petit projet. Nous allons ici voir, commit par commit, comment arriver au résultat, pour mettre en avant le cheminement.

Commit n°1 : fenêtre

Première étape, et non des moindres : essayons déjà d'afficher une fenêtre. Cette étape se fait avec le code de base de GTK pour créer une application.

La fonction `do_activate` est appelée lorsque `run` est lancé. Elle crée une fenêtre, assure que l'application se ferme lorsque la fenêtre se ferme, et affiche la fenêtre.

Commit n°2 : liste de flux

Deuxième étape : ajoutons les flux à gauche !

Notre fenêtre ne sera plus une fenêtre standard, nous allons créer une classe `Window` qui en hérite. À la création, nous ajouterons un conteneur horizontal de boîtes, avec les flux (`Gtk.StackSidebar`) qui activeront les liens (`Gtk.Stack`).

Commit n°3 : liste d'articles

On ajoute la liste des liens. Chaque liste de liens (correspondant à un flux) est un `TreeView`, et ces listes seront ajoutées dans la pile.

Commit n°4 : ouverture des liens

Pour ouvrir les liens, on utilise le module `webbrowser` qui se charge d'ouvrir le navigateur avec l'URL données. Le navigateur est ouvert lorsque la ligne correspondant à l'article est activée.

Commit n°5 : lecteur de flux

On pourrait croire que la lecture des flux est une opération délicate. Il n'en est rien. Avec `feedparser`, il nous suffit de trois petites lignes pour changer notre jeu de test en de vraies données.

Commit n°6 : configuration

À la place des données mises dans le code, il serait bon d'utiliser un fichier de configuration. Ce fichier de configuration sera un brave fichier INI, ouvert par `ConfigParser`. On remplacera ensuite les flux factices par ceux du fichier de configuration.

Le format du fichier est simple. Les sections correspondent aux flux, elles contiennent une seule clé : `url`, qui contient sans grande surprise l'URL du flux.

Commit n°7 : mise à jour automatique

Mettons à jour les flux ! Pour cela, nous utiliserons un `timeout` gentiment proposé par `GLib`, et qui lancera la fonction `update` toutes les 300 secondes. Cette fonction `update` ajoute les flux, comme c'était fait avant une seule fois au début. Comme cette fonction est lancée plusieurs fois, elle vide d'abord les flux avant de les rajouter.

Commit n°8 : sélection du fil actif après mise-à-jour

Vider les flux, c'est bien, mais si le flux était sélectionné il ne le sera plus. Pas très pratique ! On a différentes solutions pour corriger cela. La plus simple est de garder en mémoire le flux sélectionné, de tout mettre à jour et de sélectionner le flux précédemment sélectionné.

Engrenage

Ce n'est pas le meilleur lecteur de flux RSS, d'accord. Mais avouez que la prouesse est belle : nous avons avec moins de 70 lignes un programme fonctionnel que nous pouvons utiliser au jour le jour. À ce moment-là, vous êtes déjà à la meilleure place pour vous encourager. Forcez-vous quelques temps à utiliser votre création, et si tout se passe bien vous aurez des démangeaisons à chaque bug trouvé, à chaque fonctionnalité que vous aimeriez avoir.

Vous voyez l'histoire du battement d'aile de papillon qui provoque une tempête à l'autre bout de la Terre ? Bah là, on en est juste au battement d'aile. C'est déjà un début.

Avec le temps, vous aurez un outil qui correspond à vos besoins. Cet outil devrait commencer à plus vous ressembler, et à moins ressembler aux autres outils du genre.

C'est une bonne base.

Avoir du code qui fonctionne, ce n'est pas la fin. Bien au contraire… Vous voilà plutôt au début de l'aventure.

D'un outil pour soi à un outil pour d'autres

Le premier commentaire

Au début, nous n'allons pas nous mentir : il n'y aura que vous. Personne d'autre pour utiliser, pour tester, pour donner son avis sur votre création. Ce n'est pas la peine de faire la tête, profitez plutôt de votre tranquillité. Parce que si tout se passe bien, elle ne va pas durer.

Vous allez peut-être parler de votre logiciel à quelques connaissances, des personnes susceptibles d'utiliser votre merveille, ou au moins d'y jeter un coup d'œil. Il risque d'y avoir encore quelques semaines, quelques mois… Mais si vous continuez à utiliser votre logiciel, et à le développer un minimum, arrivera ce qui devait arriver : un commentaire.

Là, tout dépend des outils que vous avez utilisés. Généralement, vous mettrez votre code sur une forge logicielle, avec un minimum de métadonnées : un README, une description, quelques étiquettes… La prise de contact avec le monde extérieur risque d'arriver sous la forme d'un ticket.

Un seul conseil : ne paniquez pas.

La persévérance

Facile : c'est tout droit.

Cette quête est longue et fastidieuse. Vous aurez peut-être quelques personnes qui viendront vous voir au bout de quelques années. Peut-être n'est-ce qu'un début. On ne sait pas à l'avance quels projets fonctionneront… L'important est de durer assez pour que les gens affluent, ou que le logiciel devienne obsolète.

Prenez soin des gens qui viennent. Vous pouvez les bichonner, répondre à leurs attentes (parfois floues, souvent discutables), et les encourager à donner leur avis. Après quelques messages, on peut parfois avoir un effet boule de neige, avec de plus en plus de personnes qui réagissent, qui en amènent d'autres, et encore d'autres, et encore d'autres…

Avec un minimum de réactivité, tout le monde devrait être content, ou au moins éclairé sur vos choix. Cette réactivité est dure à tenir, surtout après plusieurs mois de calme plat, mais c'est votre meilleure alliée pour toucher de nouvelles personnes : quand quelqu'un qui utilise votre logiciel est content, il a beaucoup plus de chances d'amener vers vous d'autres personnes.

Vous courez un marathon. Allez-y doucement, mais avec constance.

Un minimum de communication

Vous allez faire des choses sur votre logiciel. C'est évidemment très dur de communiquer ce qu'il y a dans votre tête, parce que vous passez déjà tellement de temps à développer… Et pourtant, il faut au moins s'assurer d'accueillir convenablement les gens de l'extérieur, leur donner envie de venir vers vous.

Si vous arrivez sur un projet avec juste du code, sans savoir ce que ça fait, sans savoir comment ça s'utilise, sans savoir comment prendre contact avec l'équipe de développement, vous avez peu de chances de vous lancer, beaucoup plus de partir en courant. C'est pareil pour tout le monde, c'est pareil pour votre logiciel.

Le minimum est d'accueillir les arrivants avec quelques mots rassurants : une petite description du but et des fonctionnalités, quelques lignes pour savoir comment utiliser le logiciel, un moyen de vous joindre. Si vous avez le courage, vous pouvez même publier des versions, avec quelques notes présentant les nouveautés.

Et bien sûr, si vous vous sentez l'âme communicante, n'hésitez pas à créer un petit site, de la documentation poussée, du contenu sur les réseaux sociaux… Mais comme pour le reste, assurez-vous de tenir la distance : vous courez un marathon. Rien de plus triste qu'un compte Twitter avec seulement un message d'ouverture de compte, ou un blog avec un seul article qui date d'il y a 5 ans. Mieux vaut faire peu régulièrement que beaucoup d'un coup au début sans plus rien après !

La communauté

La gentillesse, toujours

La chose la plus importante que j'ai réalisée au long de mes gestions de projets est l'incroyable puissance de gentillesse.

Avec le temps, je me suis rendu compte que j'avais beaucoup plus envie de rapporter des bugs et donner mon avis aux équipes de développement les plus gentilles. Et de la même manière, les gens qui contribuaient le plus à mes projets étaient les gens avec qui j'avais de bons rapports.

Ça paraît simple, mais c'est en réalité compliqué.

Être gentil, tout le temps, de manière inconditionnelle, ce n'est pas facile. C'est d'autant plus vrai que la méthode d'échange principale est souvent l'écrit, avec son potentiel manque de chaleur, ses lacunes en communication non-verbale, et les incompréhensions qui vont avec.

Être gentil, c'est être poli, accueillir les gens, passer du temps à essayer de comprendre leurs besoins. C'est expliquer beaucoup de choses dans le détail, à des gens qui n'ont parfois pas du tout le même bagage technique que vous. C'est long, et il arrive parfois que ce soit très frustrant. Et comme tout le reste, c'est difficile à tenir sur la distance.

Mais après, les gens sont contents. Et vous aussi.

Être gentil, ça ne veut pas dire tout accepter. Vous pouvez refuser des fonctionnalités qui ne vous plaisent pas, vous pouvez refuser de passer du temps à aider certaines personnes parce que c'est trop long ou compliqué. Au début, j'ai été très étonné de voir que les gens étaient vraiment reconnaissants lorsque je prenais le temps d'expliquer pourquoi une fonctionnalité de m'intéressait pas, même lorsque du temps avait été passé pour proposer du code.

Être gentil, ça ne veut pas non plus dire tout laisser passer. Avoir un code de conduite aide beaucoup à régler les problèmes amenés par des personnes énervées, vulgaires, insultantes, ou simplement maladroites. On peut dire gentiment qu'un commentaire ne suit pas le code de conduite. On peut expliquer gentiment que le comportement n'est pas acceptable selon les règles mises en place. Et souvent, répondre avec sourire et fermeté à de l'agressivité désamorce rapidement la tension.

La ligne directrice

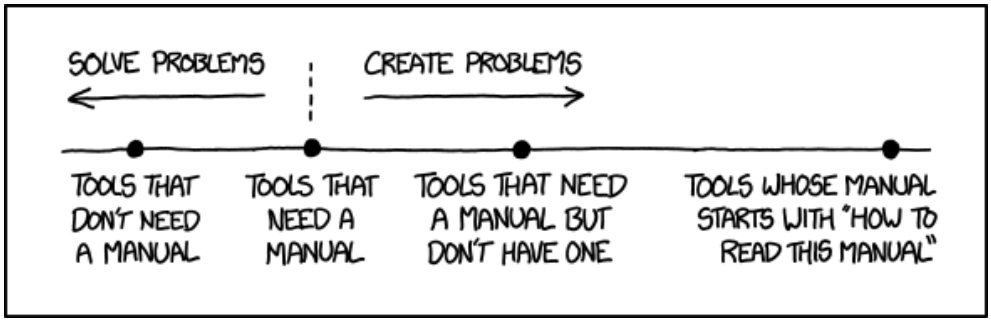
Personne ne peut être dans votre tête.

Il est facile, lorsque l'on vient vous voir avec une question déjà mille fois posée, de dire que la réponse se trouve dans un ticket ou dans la documentation, d'une phrase lapidaire et cruelle. C'est tentant.

Mais vous pouvez également réfléchir : si mille personnes vous posent mille fois la même question, c'est peut-être que le problème vient de vous.

Il est très difficile de faire une bonne documentation, et encore plus difficile de faire un logiciel qui n'a pas besoin de documentation. Mais c'est votre devoir, si vous voulez avoir des utilisatrices et des utilisateurs heureux, d'éviter qu'ils ne se posent des questions, ou du moins qu'ils trouvent rapidement leurs réponses.

La première des documentations est d'expliquer ce que fait votre projet, mais aussi ce qu'il ne fait pas.



Le XKCD obligatoire. C'est quand même pas compliqué, la réponse à cette question est dans la note de bas de page de la préface de la documentation. C'est écrit noir sur blanc avec des lettres, quoi. Des lettres normales, hein, que tout le monde peut lire. Les gens font jamais, jamais d'efforts…

Avoir une ligne directrice forte, clivante, c'est une bonne manière d'attirer à vous des personnes réellement intéressées par votre logiciel et par vos idées. Et lorsque cette ligne sera claire et bien comprise, vous aurez tout le plaisir de combler de bonheur celles et ceux qui utiliseront votre projet.

Courir votre marathon nécessitera de faire des choix. Si vous portez trop de choses, vous finirez par vous épuiser, et par frustrer tout le monde autour. Au contraire, si vous commencez avec des objectifs simples et faciles à communiquer, vous irez plus loin, vous attirerez d'autres personnes capables de porter de nouvelles choses avec vous.

Accepter la fin

Comme tout le reste, votre projet mourra un jour. Autant accepter tout de suite cette évidence.

Ce n'est pas grave.

Lorsque vous en aurez marre, quelle que soit la raison, vous pouvez laisser votre projet. Si d'autres personnes le souhaitent, elles pourront s'en occuper à votre place, et l'emmèneront là où elles voudront.

Votre projet, ce n'est pas vous. C'est quelque chose qui peut grandir avec tout ce que vous lui donnerez, et pour lequel vous pouvez avoir un réel attachement sentimental. Mais à un moment, il faudra accepter de le laisser s'envoler avec d'autres, ou de reposer en paix.

L'une des beautés de l'informatique, et du développement de logiciel libre en particulier, est de donner aux gens la possibilité de créer beaucoup de choses avec peu de moyens, et de les partager avec qui voudra. Ce n'est pas parce qu'un projet s'arrête que tout s'arrête, c'est au contraire la possibilité de faire autre chose.

Et si, à la fin, vous vient l'idée que vous avez perdu du temps, rendez-vous à l'évidence : ces innombrables heures consumées en cendres se sont transformées en expérience, en découvertes, en souvenirs…

www.stella.coop

@stellalyon69

stella-lyon