Petit guide Python pour physicien-ne-s*

Léo Esnault

Janvier 2019

Ce guide s'adresse aux scientifiques ayant quelques notions de programmation, et permet de survoler la syntaxe et les spécificités du langage Python ainsi que les erreurs les plus courantes. La version numérique contient de nombreux liens renvoyant vers une documentation plus complète.

1 Python : pourquoi faire ?

La programmation est quasi-indispensable pour faire de la physique aujourd'hui → permet de résoudre des problèmes non solubles analytiquement (pour faire des prédictions théoriques), tracer des courbes et analyser les données d'expérience et de simulation.

Python est un langage de choix pour effectuer ces opérations simplement. Toutefois si le programme nécessite un temps d'exécution important, il peut être intéressant d'utiliser un langage compilé (C, C++, Fortran, ...).

Les spécificités de Python sont :

- Langage interprété : la traduction en langage machine est effectuée à chaque exécution → il est possible d'exécuter des instructions dans une console interactive (tests et débogage plus faciles), mais l'exécution du programme est plus lente que pour un programme compilé
- Polyvalent : nombreux modules pour applications scientifiques, interfaces graphiques, le web, les jeux vidéos, ...
- (Très) Haut niveau : l'interpréteur s'occupe automatiquement de l'architecture machine → plus facile a coder et a lire mais potentiellement moins performant
- Dynamique : pas de déclaration des variables → facilite le codage mais peut induire des erreurs si on ne code pas proprement
- Multi-paradigmes : programmation procédurale, fonctionnelle, orientée objet, ... → on peut utiliser des techniques de programmation plus en plus poussées tout en restant dans le même langage
- Open source : le code source est disponible pour pouvoir être étudié et amélioré par tous → le langage est et restera gratuit pour tous, ce qui facilite la collaboration
- Multiplateformes : disponible sur Linux, Mac, Windows, Android, ...

Python a été conçu pour être le plus lisible possible, car d'après son créateur Guido van Rossum, le code est lu bien plus souvent qu'il n'est écrit. La philosophie du langage est

*Cette œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution - Partage dans les Mêmes Conditions 2.0 France.

résumée dans le `Zen de Python`. Les outils les plus utiles aux physicien-ne-s sont principalement `numpy` pour travailler avec des vecteurs et matrices, ainsi que `matplotlib` pour afficher des graphiques. Ces deux modules font partie de la suite `scipy`, qui contient de nombreux autres outils.

Pour des applications plus spécifiques, il peut être pertinent de faire une recherche dans le Python Package Index (PyPI) avant de coder soi même une solution. Ces modules peuvent être installés à l'aide de l'outil `pip`.

Documentation généraliste

Livre complet sur la programmation en Python :
`http://inforef.be/swi/download/apprendre_python3.pdf`
Autre livre, plus condensé :
`https://perso.limsi.fr/pointal/_media/python:cours:courspython3.pdf`
Cours intensif Python pour scientifiques (Jupyter notebook, en anglais) :
`https://nbviewer.jupyter.org/gist/rpmuller/5920182`
La même chose en français (la partie 1 est disponible en lien dans l'encadré) :
`https://python.developpez.com/tutoriels/cours-intensif-scientifique/module-numpy-scipy/`
Python pour programmeurs :
`https://perso.limsi.fr/pointal/_media/python:pythonpro.pdf`
Tutoriel généraliste très complet :
`https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python`
Un cours sur l'utilisation de la suite `scipy` (en anglais) :
`http://scipy-lectures.org/`
Le site officiel de Python, plein de tutoriels :
`https://docs.python.org/fr/3.7/tutorial/index.html`

2 Syntaxe et utilisation des différents types

La syntaxe du langage est résumée dans les parties suivantes. Ce document est toutefois très incomplet ; pour plus de détails, se reporter au résumé de la syntaxe Python dans les références ou a la documentation officielle Python (dont ce résumé est largement inspiré).

Pour toute la suite, l'affectation est réalisée par le signe `=` et on accède aux attributs et méthodes d'un objet par le `.` (en Python, tout est objet y compris les nombres). L'égalité se teste par le signe `==` et la différence par `!=`. Les raccourcis `+=`, `*=`, ... permettent de réaliser l'opération puis l'affectation (par exemple `i = i + 1` ⇔ `i += 1`).

La fonction `help` est très utile pour accéder a la documentation d'un objet ou d'une fonction dans une console interactive.

Types numériques

L'utilisation la plus simple de Python est de s'en servir comme calculatrice (ce qui est grandement facilité par l'utilisation de la console interactive `python` ou `ipython`, servez vous en !). Le langage contient donc les types numériques entiers, réels et complexes, qui sont simplement des nombres. On peut les utiliser comme suit :

Type	Objet Python	Exemples	Conversion
Entier	int	`1`, `-36`	`int("2.0")`
Réel	float	`2.54`, `3e7`	`float(7)`
Complexe	complex	`1+2j`	`complex(7,3)`

puis les combiner à l'aide des opérations `+`, `-`, `*`, `/`, `//` (division entière), `**` (élévation a la puissance) ou les comparer à l'aide de `==`, `!=`, `>`, `<`, `>=`, `<=`. Il est possible de raccourcir le test `i > 10 and i < 25` par `10 < i < 25`.

La fonction `abs` renvoie la valeur absolue et la fonction `round` peut être utile pour faire un arrondi, ainsi que les fonctions numpy `ceil`, `floor`, `trunc` (arrondi supérieur, inférieur, troncature).

En Python2, la division entre deux entiers renvoie un entier (réel en Python3) → il est nécessaire de convertir le numérateur ou dénominateur en réel avant de diviser !

Booléens

Les booléens ont deux valeurs possibles, `True` et `False`. On peut les combiner avec `not` (NON logique), `and` (ET logique) et `or` (OU logique). Ce type est nommé `bool` en Python.

Les valeurs numériques `0`, `0.` et `0j`, les chaînes de caractères vides `""` et les conteneurs vides (ordonnés ou non) `[]`, `()`, {} ainsi que `None` ont une valeur booléenne considérée comme `False`. Toutes les autres valeurs sont considérées comme `True`. On peut utiliser ces caractéristiques pour alléger le code lors de tests `if` ou `while`.

Conteneurs ordonnés

Les conteneurs sont des objets pouvant en contenir d'autres (nombres, chaînes, autres conteneurs, ...). Certains d'entre eux conservent l'ordre dans lequel on les a initialisé, et on les appelle alors conteneurs ordonnés. En Python il y a deux types natifs de conteneurs ordonnés : les listes, et les tuples. En programmation scientifique on utilise aussi très souvent les objets `numpy.array` pour effectuer des calculs.

Les tuples sont des conteneurs ordonnés immuables, i.e. ses éléments (pas forcément du même type) ne peuvent pas être modifiés après l'initialisation. On les initialise grâce aux parenthèses `(elem1, elem2, ...)` ou simplement en les séparant par des virgules `elem1, elem2, ....` On utilise surtout les tuples pour faire des affectations multiples `a, b = 1, 2`, définir des constantes ou initialiser des paramètres par défaut dans une fonction (utiliser des listes peut induire un comportement non désiré). Il est possible de convertir une liste `lst` en tuple en utilisant `tuple(lst)`.

Les listes sont similaires aux tuples mais sont muables (on peut changer la valeur d'un élément ou en ajouter/supprimer). On les initialise grâce aux crochets `["aa",4e8,(1,3,7)]`. Les méthodes associées à ce type les plus utiles sont `append` pour ajouter un élement à la fin de la liste, `insert(i,elem)` pour ajouter l'élément `elem` à la position `i`, ainsi que `sort` pour trier la liste et `copy` pour en créer une copie. La fonction `range(min, max, step)` permet de générer une liste (en Python2) ou un itérable (Python3, utiliser `list(range(min,max,step))` pour créer une liste) de nombre entiers entre `min` inclu et `max` exclu avec un pas de `step` (par exemple `list(range(0,10,2))` renvoie `[0,2,4,6,8]`). Les paramètres `min` et `step` sont optionels (valeurs par défaut `0` et `1`).

Pour les tuples et les listes, l'opérateur `+` concatène la liste de gauche avec celle de droite (`[0, 1, 2] + [3, 4]` donne `[0, 1, 2, 3, 4]`) et l'opérateur `*` répète le conteneur à gauche le nombre a droite de fois (`[0] * 3` donne `[0, 0, 0]`), bien que cette approche soit déconseillée pour créer des listes de listes.

Les array sont des conteneurs de **nombres** fournis par la bibliothèque `numpy` et sont les conteneurs les plus souvent utilisés en programmation scientifique (avec les listes). Ce type est muable, et a la différence des listes, les opérateurs effectuent des opérations mathématiques élément par élément (par exemple `a = np.array(range(0,5))` puis `a = a**2` donne `array([0.0, 1.0, 4.0, 9.0, 16.0])`). Toutes les opérations et méthodes numpy sont beaucoup plus rapides que de boucler sur tout les éléments d'une liste (car elles sont en fait codées en C), tirez en parti !

Pour accéder a l'indice `i` d'un conteneur ordonné, on utilise les crochets `cont[i]`. On peux aussi sélectionner la tranche (slice) du conteneur entre les indices `i` inclus et `j` exclus en utilisant `cont[i:j]`. Il est possible d'ommetre un ou les deux indices lors d'une sélection,

les valeurs par défaut étant le premier et le dernier élément. Pour sélectionner une tranche de conteneur entre `i` et `j` avec un pas `k`, on utilise `cont[i:j:k]`. L'indice `-1` corresponds au dernier élement du conteneur, l'indice `-2` a l'avant dernier, ... Pour les conteneurs ordonnés muables, on peu changer la valeur d'un élément ou d'une tranche en utilisant `cont1[i:j] = cont2`.

Une des grandes forces de Python pour la programmation scientifique est la possibilité de pouvoir filtrer facilement un `np.array` avec une condition : pour ce faire, on peut insérer un `np.array` de booléen entre crochets, soit par exemple `a[b == c]` qui renvoie toutes les valeurs de `a` tel que `b == c`.

Les fonctions `sum` et `len` renvoient la somme des éléments, la longueur du conteneur ordonné ; la fonction `reversed` renvoie un conteneur retourné (le début devient la fin) ; la fonction `sorted` renvoie un conteneur trié par valeur croissante ; la fonction `enumerate` renvoie une liste de tuples contenant l'indice et l'élément du conteneur.

D'autres fonctions, moins utilisées, sont aussi disponibles. Se référer à la documentation complémentaire.

Conteneurs non ordonnés

dictionnaires et ensembles.

Chaînes de caractères

Les chaînes de caractères sont utiles pour afficher du texte a l'écran ou exporter des données dans un fichier. Elles peuvent être créées en entourant le texte de `"` ou de `'` (par exemple `"Ceci est une chaîne"`). On peux aussi utiliser trois `"` ou `'` pour écrire sur plusieurs lignes. Pour ajouter une chaîne après une autre (concaténation), on utilise le signe `+`.

Les chaînes de caractères sont itérables (on peut boucler dessus), immuables (on ne peut pas les modifier) et ordonnées (les lettres restent dans le bon ordre). Ce sont en fait des conteneurs ordonnés, et on peut donc utiliser les fonctions définies précédemment (notamment les indices pour sélectionner une tranche de chaîne).

Les caractères `"\n"` sont interprétés comme un retour a la ligne, et les caractères `"\t"` comme une tabulation. Le signe \ est appelé caractère d'échappement et est utile pour empêcher l'interprétation du caractère suivant par le langage (si on veut insérer un `#` dans du texte il faut l'échapper `\#` pour empêcher son interprétation comme commentaire).

Les chaînes de caractère ont de nombreuses méthodes très utiles, telles que `format` (pour insérer des valeurs dans du texte), `split` et `splitlines` (pour créer une liste d'éléments à partir d'une chaîne, une liste de lignes), ou encore `replace` (qui remplace une sous-chaîne par une autre), ... On peut y accéder en utilisant `chaine.methode`. L'opérateur `in` peut être utilisé pour tester si une sous-chaîne est contenue dans une autre (`"exemple" in "Ceci est un exemple."` renvoie `True`).

Pour insérer une valeur numérique `val` avec une unité `unit` dans du texte, on peux utiliser la fonction de conversion `str(val)` ou utiliser les formatages suivants :

- `"Valeur : %fmt %s"%(val,unit)` (plutôt Python2)
- `"Valeur : {v:fmt} {u:s}".format(v=val,u=unit)` (plutôt Python3)

Le tableau suivant résume les principaux formats (en remplacant `fmt`) :

`fmt`	Type de conversion
s	str
i	int
.2f	float (2 décimales)
.2e	float en notation scientifique (2 décimales)
.2g	Équivalent à e si l'exposant est > 4, f sinon

On peut aussi fixer le nombre de caractères à utiliser pour combler avec des espaces en indiquant un nombre avant le `.` (`%10.4f` pour un réel de 4 décimales sur 10 caractères), réserver un caractère pour le signe avec un espace (`% i` pour un entier signé), ... Voire documentation complémentaire.

Pour afficher du texte a l'écran, on utilise la fonction `print` (instruction en Python2).

Pour écrire dans un fichier on doit ouvrir le fichier à l'aide de `open("nom_fichier.txt","w")` puis utiliser la méthode `write` de l'objet créé, et enfin ne pas oublier de fermer le fichier avec la méthode `close`. Pour lire un fichier on utilise la fonction `open("nom_fichier.txt","r")` puis on boucle souvent sur la liste renvoyée par sa méthode `readlines()` (typiquement `f=open("test.txt","r")` puis `for line in f.readlines():` et `elem1, elem2 = line.split()`). Les fonctions `numpy savetxt` et `loadtxt` permettent de simplifier l'écriture/la lecture de fichiers.

Il est aussi possible d'exécuter le code contenu dans une chaîne à l'aide de la fonction `exec` (instruction en Python2), ou d'évaluer le résultat d'un calcul (contenant ou non des variables définies dans le script) à l'aide de la fonction `eval` (par exemple `x = 1` puis `eval("x+1")`).

En Python2 il peut y avoir quelques problèmes pour utiliser les accents (du à une mauvaise gestion de l'unicode). Si tel est le cas, il faut insérer la ligne `# coding:utf8` au début du script et insérer la lettre `u` avant chaque chaîne de caractère pour la convertir en unicode utf8 (`u"Chaîne de caractères avec accents"`).

Autres types

bytes, None

Instructions

Certains mot clés sont réservés par le langage (et ne peuvent pas être redéfinis) afin de pouvoir effectuer différentes instructions, dont voici quelques exemples :

Instruction	Utilisation
`import mod`	Importe le module `mod`
`from mod import func`	Importe la fonction `func` du module `mod`
`import mod as m`	Importe le module `mod` et définit l'alias `m`
`del var`	Suppression de la variable `var`
`if txt == "test":`	Exécute le bloc d'instructions suivant si la chaîne `txt` vaut `"test"`
`else:`	Sinon, exécute le bloc qui suit cette instruction
`elif txt == "tmp":`	Combinaison de `else` et `if`
`while i < 10:`	Exécute un bloc d'instructions (boucle) tant que `i<10`
`for i in range(5):`	Fait boucler la variable `i` de `0` à `4` en exécutant le bloc suivant
`pass`	Ne fait rien. Utile pour coder un prototype a terminer plus tard
`break`	Termine la boucle en cours
`continue`	Passe a l'itération suivante dans une boucle
`def func(a,b=2):`	Définit la fonction `func` prenant deux paramètres dont un par défaut
`return a**2`	Retourne la valeur `a**2`. A utiliser uniquement dans une fonction
`class ExempleDeClasse:`	Définit la classe `ExempleDeClasse`
`raise NameError('msg')`	Lève une exception (erreur) de type `NameError` avec le message `'msg'`
`try:`	Exécute le bloc suivant si aucune exception n'est levée
`except TypeError:`	Exécute le bloc suivant si une exception `TypeError` est levée
`finally:`	Exécute le bloc suivant après avoir géré les exceptions
`with`	...
`lambda x:  x**2`	Définit une fonction anonyme qui retourne `x**2`

Fonctions supplémentaires

Python est livré "piles incluses", i.e. avec un grand nombre de modules par défaut (mathématiques avec le module `math`, aléatoire avec `random`, parallélisation avec `multiprocessing`, ...). Il est très souvent pertinent de chercher dans la documentation de ces modules plutôt que de re coder des fonctions déjà présentes dans le langage (et souvent mieux implémentées). Les modules de la suite `scipy` ajoutent de nombreuses autres fonctions (intégration, dérivation, résolution d'équations différentielles ordinaires, ...) qui ont été optimisées pour diminuer le temps de calcul : profitez en !

Voici quelques fonctions parmi les plus utiles de la suite scipy (`numpy` est noté `np` et `matplotlib.pyplot` est noté `plt`):

Fonction	Utilisation
`np.array(lst)`	Converti la liste `lst` en objet `np.array`
`np.linspace(min, max, n)`	Créé un vecteur de `min` a `max` avec `n` bins
`np.arange(min, max, dx)`	Créé un vecteur de `min` a `max` avec un espacement de `dx`
`np.logspace(min, max, n)`	Créé un vecteur de `min` a `max` avec `n` bins espacés en log
`np.transpose(mat)`	Effectue la transposition de la matrice `mat`
`np.dot(vec1,vec2)`	Effectue le produit scalaire entre deux vecteurs
`np.sin(x)`, `np.exp(x)`	Calcule le sinus, l'exponentielle de `x`
`np.trapz(y,x)`	Intègre `y` par rapport à `x` via la méthode des trapèzes
`np.gradient(y,x)`	Dérive `y` par rapport a `x`
`np.mean(x)`	Calcule la valeur moyenne de `x`
`np.histogram(data)`	Créé un histogramme 1 dimension du vecteur `data`
`np.histogramdd(data)`	Créé un histogramme n dimensions des données `data`
`np.meshgrid(x,y,indexing='ij')`	Créé une matrice 2D à partir de 2 vecteurs 1D
`np.rand.rand(20,20)`	Créé une matrice 20x20 avec valeurs aléatoires entre 0 et 1
`plt.figure(0)`	Créé ou sélectionne la figure `0`
`plt.clf()`	Nettoie la figure courante
`plt.show()`	Affiche une figure
`plt.ion()`, `plt.ioff()`	Active, désactive le mode interactif
`plt.plot(x,y,label="data")`	Trace `y` en fonction de `x` avec la légende `"data"`
`plt.step(x,y,label="data")`	Idem précédent, mais avec un style histogramme
`plt.pcolormesh(gx,gy,data)`	Trace une carte 2D de `data` en fonction de `gx` et `gy`
`plt.colorbar()`	Affiche l'échelle de couleur (pour carte 2D)
`plt.legend()`	Affiche la légende
`plt.xscale('log')`	Passe l'échelle des abscisses en log (`"linear"` pour linéaire)
`plt.xlabel('$x$ [$m$]')`	Change le label de l'axe des abscisses (format LaTeX)

Modules `xml` pour les fichiers xml, module `json` pour les fichiers JSON.

Documentation complémentaire

Résumé de la syntaxe Python (28 pages) :
`http://www.xavierdupre.fr/site2013/documents/python/resume_utile.pdf`
Aide mémoire Python3 (2 pages à garder sous le coude !) :
`https://perso.limsi.fr/pointal/_media/python:cours:mementopython3.pdf`
Formatage de chaînes de caractères :
`https://pyformat.info/`
Liste des types natifs de Python3 :
`https://docs.python.org/fr/3.5/library/stdtypes.html`
Liste des modules standart de Python3 :
`https://docs.python.org/3/py-modindex.html`

Liste des fonctions natives de Python3 :
`https://docs.python.org/fr/3.5/library/functions.html`
Liste des fonctions numpy :
`https://docs.scipy.org/doc/numpy/reference/routines.html`
Tutoriels matplotlib :
`https://matplotlib.org/tutorials/index.html`
Modules de la suite scipy :
`https://scipy.org/docs.html`
Base de donnée des modules Python (Python Package Index) :
`https://pypi.org/`

3 Coder proprement

Dès qu'on a une pratique régulière de la programmation et que les bases sont acquises, il est très utile d'adopter quelques bonnes pratiques concernant la structure et le format de son code. Il peut sembler au premier abord que ces recommandations sont superflues, mais les nombreuses erreurs générées par un code de mauvaise qualité peuvent souvent être évitées en employant quelques pratiques simples ; cela permet donc au final de gagner du temps et de laisser un code facilement compréhensible et réutilisable si vous devez transmettre ou collaborer sur votre programme. La recommandation la plus importante est sans doute de d'abord privilégier la clarté du code, en choisissant des noms de variables suffisamment clairs, en segmentant son code en fonctions et en laissant de nombreux commentaires.

Il est important de noter que certains articles scientifiques publiés ont du être dé-publiés a cause de conclusions erronées liées à des erreurs de codes. Pour plus d'informations, voire l'article "" dans la documentation complémentaire. Cet article donne les recommandations suivantes :

3.1 Codez pour les humains, pas pour les ordinateurs

La structure de votre programme doit rester simple, afin de pouvoir être comprise rapidement et d'isoler facilement les parties du code qui peuvent potentiellement poser problème. Cela se fait en codant des fonctions qui ont un rôle très spécifique et bien définit (on appelle cela modulariser son code). Chaque suite d'instructions effectuée plus de deux fois doit être transformée en fonction.

Les noms de variables et fonctions doivent être suffisamment explicites. Évitez les noms `a, b, xx, fonction, ...` a moins que leur sens ne soit explicitement indiqué dans les commentaires. L'utilisation de fonctions permet d'utiliser plusieurs fois les mêmes noms de variables (dans sa portée locale).

Utilisez un style de code cohérent (convention de nommage). Pour Python, il est conseillé d'utiliser des minuscules avec tirets bas pour les variables et les fonctions (`exemple_de_variable` et `exemple_de_fonction`), les majuscules avec tirets bas pour les constantes (variables ne devant pas être modifiées, `EXEMPLE_DE_CONSTANTE`), les majuscules aux initiales pour les classes (`ExempleDeClasse`), les minuscules pour les modules (`exempledemodule`).

3.2 Laissez l'ordinateur faire le travail

Make the computer repeat tasks. Automatisez votre travail. L'import de fichier, leur traitement puis l'export de figures peuvent être automatisés via un script. Programmer des fonctions les plus générales possibles, regroupées en modules permet de grandement simplifier l'écriture et la lecture de ces scripts. Les tests du code peuvent aussi être automatisés via un script ou via des librairies spécialisées.

Programmez des fonctions pouvant être ré-utilisées des scripts Save recent commands in a file for re-use.

Créez des modules contenant toutes les fonctions nécessaires a votre programme, et programmez des scripts Use a build tool to automate workflows.

3.3 Améliorez votre code étape par étape

Programmez uniquement ce dont vous avez besoin, et améliorez vos fonctions ou la structure de votre code étape par étape. Cela permet de ...

Utilisez un système de gestion de version (voire partie suivante). Work in small steps with frequent feedback and course correction. Use a version control system. Put everything that has been created manually in version control.

3.4 Ne vous répétez pas

Chaque donnée (constante physique, paramètre expérimental, paramètre numérique, ...) doit être définie une seule fois. Cela évite d'avoir à modifier le code a plusieurs endroits lorsqu'une modification est nécessaire, et tout définir au même endroit améliore la lisibilité de la structure du code.

Modularisez votre code au lieu de le copier-coller. Cela peut être fait en programmant des fonctions suffisamment générales pour être réutilisées dans plusieurs situations. On peut ajouter des paramètres optionnels aux fonctions pour gérer les cas particuliers.

Ré-utilisez le code au lieu de le ré-écrire. De nombreux modules de programmation scientifique existent en Python, profitez en !

Cela facilite aussi la maintenance et améliore la lisibilité du code. Chaque partie du code doit aussi avoir un but précis et définit.

Every piece of data must have a single authoritative representation in the system. Modularize code rather than copying and pasting. Re-use code instead of rewriting it.

3.5 Prévoyez les erreurs

Testez les valeurs de retour de vos fonctions avec des cas simples. Cela permet de repérer rapidement un effet de bord

Utilisez une librairie de test. Plusieurs existent en Python, donc doctest, pytest ou ...

Add assertions to programs to check their operation. Use an off-the-shelf unit testing library.

Turn bugs into test cases.

Use a symbolic debugger. Utilisez un débogueur. Dans une console ipython, la commande `%%debug` permet d'entrer dans la fonctions ayant produit une erreur, afin d'afficher les valeurs des variables intermédiaires au calcul.

3.6 Optimisez le code seulement quand il fonctionne

Utilisez un profileur pour identifier les goulots d'étranglements. En Python, le programme ... permet de savoir combien de temps dans quelle fonction. Si de bonnes performances sont nécessaires, il est possible d'executer du code Fortran dans un script Python à l'aide de f2py, et du code C,C++ à l'aide de ... Use a profiler to identify bottlenecks.

Write code in the highest-level language possible. Utilisez un langage de plus haut niveau possible. Python est un bon choix pour des applications légères. C++ gère la programmation orientée objet et l'allocation dynamique, et Fortran .

3.7 Documentez le "pourquoi", pas le "comment"

Document interfaces and reasons, not implementations. Documentez le but d'une instruction ou d'une fonction, et pas son implémentation. Cela permet de clairement indiquer ses intentions, et aide parfois a trouver une meilleure facon de résoudre le même problème. Refactor code in preference to explaining how it works. Si une partie du code a besoin d'explications poussées, il est souvent possible et préférable de ré-écrire cette partie du code de façon plus logique afin qu'elle n'ait plus besoin d'explications complexes. Quand cela n'est pas possible, il faut tenter de modulariser le code afin d'en isoler les parties complexes, et noter les références permettant de comprendre les opérations effectuées. Embed the documentation for a piece of software in that software. La documentation doit faire partie intégrante du programme. Les outils de documentation automatique permettent de générer un documentation à partir des commentaires d'un code.

3.8 Collaborez

Use pre-merge code reviews.

Use pair programming when bringing someone new up to speed and when tackling particularly tricky problems. Faites relire votre programme et expliquez votre logique. Ceci est particulièrement utile quand vous rencontrez une difficulté précise. Use an issue tracking tool. Utilisez un outil de gestion de bogues (nécessaire pour les gros programmes). Les systèmes de gestion de version en intègrent souvent par défaut.

Structure du code

Fonctions atomisées → structure claire, plus facile a déboguer.

Commentaires

Utilisation de help()

Modules

Si pratique régulière de la programmation, intéressant de se former à l'orienté objet.

Convention de nommage

Quand on programme pour soi, ça peut paraître inutile au premier abord d'adopter des règles claires concernant le nom des variables, fonctions et modules. Toutefois, à mesure que le programme se complexifie, il devient vite très facile d'oublier le but d'une instruction ou fonction (d'où l'importance des commentaires). Qui plus est, comme en Python l'allocation est dynamique, il arrive régulièrement d'écraser accidentellement la valeur d'une variable en la définissant une seconde fois dans le code (d'où l'intérêt d'avoir des noms de variables suffisamment explicites).

La communauté Python adopte des règles claires concernant le nommage, afin d'éviter les problèmes précédents, et pour être plus facilement lu par soi même ou d'autres :

- Les variables : `exemple_de_variable`
- Les constantes (variables ne devant pas être modifiées) : `EXEMPLE_DE_CONSTANTE`
- Les fonctions : `exemple_de_fonction` ou `exempleDeFonction`
- Les classes : `ExempleDeClasse`
- Les attributs ou méthodes privées (en programmation orientée objet) : `_attribut_ou_methode`
- Les modules : `exempledemodule` ou `exemple_de_module`

Pour indiquer que l'on affecte une valeur a une variable inutile, on utilise souvent l'underscore _ (par exemple : `utile, _ = ('Données utiles', "On s'en fout")`). On doit aussi éviter les variables globales (sauf cas très particuliers), ainsi que les accents/caractères spéciaux comme noms de variables. Il est aussi recommandé d'utiliser des alias pour les modules (par exemple utiliser `import numpy as np` plutôt que `from numpy import *`) afin d'éviter d'écraser des fonctions précédemment définies. Comme les instructions sont en anglais, le plus clair est de coder et commenter en anglais si possible.

Documentation complémentaire

Rapide résumé de pourquoi et comment bien coder :
`https://perso.liris.cnrs.fr/pierre-antoine.champin/enseignement/algo/cours/algo/bonnes_pratiques.html`
Beaucoup de choses très utiles pour écrire du bon code :
`https://python-guide-pt-br.readthedocs.io/fr/latest/index.html`
Un court article scientifique sur l'importance de bien coder (la liste précédente en est largement inspirée) :
`https://doi.org/10.1371/journal.pbio.1001745`
La référence concernant les bonnes pratiques à adopter en Python (PEP8) :
`https://www.python.org/dev/peps/pep-0008/`

4 Outils utiles

Environnement de développement intégré

En anglais : IDE

Générer une documentation

Documentation très utile, mais souvent très laborieux. En utilisant la syntaxe qui va bien pour les commentaires de fonctions, il est très facile de génerer une documentation en pdf ou html à l'aide de Sphinx ou Doxygen. `https://fr.wikipedia.org/wiki/G%C3%A9n%C3%A9rateur_de_documentation`

Gérer l'évolution de son code et collaborer en ligne

Outil permettant de suivre l'historique des modifications de un ou plusieurs fichiers, et de le synchroniser avec d'autres appareils (disque dur externe, serveur auto-hébergé, hébergement en ligne, ...).

Plusieurs outils : Git, Mercurial, Subversion, ...

Git est le plus utilisé, hébergement en ligne sur Github (dépots publics gratuits), Gitlab (dépots publics et privés gratuits), BitBucket, Framagit, ...

GitKraken,

Documentation complémentaire

Liste d'IDE Python :
`https://wiki.python.org/moin/IntegratedDevelopmentEnvironments`
IDE Pyzo :
`https://pyzo.org/`
IDE Spyder :
`https://www.spyder-ide.org/`

Liste d'éditeurs Python :
https://wiki.python.org/moin/PythonEditors
Éditeur Atom :
https://atom.io/
Éditeur Geany :
https://www.geany.org/
Jupyter Notebook :

Liste de générateurs de documentation :
https://en.wikipedia.org/wiki/Comparison_of_documentation_generators
Python Sphinx :
http://www.sphinx-doc.org/en/master/
Doxygen :
http://www.doxygen.nl/
Site officiel de git :
https://git-scm.com/
Introduction a git en français (très bien fait) :
https://perso.liris.cnrs.fr/pierre-antoine.champin/enseignement/intro-git/
Aide mémoire git (2 pages) :
https://services.github.com/on-demand/downloads/fr/github-git-cheat-sheet.pdf
Comparaison des offres d'hébergement de code en ligne (Github, Bitbucket, Gitlab, ...) :
https://en.wikipedia.org/wiki/Comparison_of_source-code-hosting_facilities