3 Écrire mieux

Un code est écrit une fois, modifié dix fois, et relu cent fois. Il faut donc simplifier son écriture, anticiper les modifications futures, et ne jamais négilger la lisibilité.

Proverbe de programmeur·se (issu de `https://ocaml.org/docs/guidelines`)

Toujours penser à la bonne poire qui devra déboguer et maintenir votre code : c'est le vous du futur. Prenez soin du vous du futur.

Proverbe personnel

Un·e programmeur·se passe (beaucoup) plus de temps à relire et éditer du code qu'à en écrire ; aussi créer du code de qualité n'est jamais du temps perdu[8]. Je présente ici quelques recommendations fondamentales. Vous pouvez exceptionnellement en dévier, uniquement exceptionnellement et pour de bonnes raisons.

3.0 Nommer

- **Utiliser des identifiants explicites**. Par exemple, si une variable `p` contient le nombre d'éléments pairs d'un tableau, ne l'appelez pas `p` ; appelez-le `nb_pairs`.
  - **Pour les variables locales, vous pouvez utiliser des abriéviations pour raccourcir le nom ; c'est même recommandé** : une variable locale est souvent utilisée, les raccourcir permet d'accélérer la relecture.[9]. Et si jamais la·e relecteur·ice a un doute sur le sens de l'abréviation, la déclaration est sous ses yeux (une fonction est courte) pour lever le doute !
  - **Cependant, pour les identifiants de fonctions et de variables globales, il est déconseillé d'utiliser des abréviations** : en effet, elles sont déclarées loin avant dans le fichier (voire dansns un autre fichier), et c'est fatigant de retourner chercher leurs définitions. Préférez un nom le plus explicite possible quitte à ce qu'il soit long.
- **Évitez les variables gobales mutables**. Devoir suivre les changements de valeur d'u
- **N'utilisez pas des noms trop similaires dans une même portée**. Lorsque deux noms de variables ne diffèrent que par un ou deux caractères, on fait des erreurs d'écriture/relecture.ne variable globale, c'est *fatigant*.
- **Si dans différentes fonctions on retrouve des variables locales qui jouent le même rôle, donnez-leur le même nom**.[10]
- **Faites attention avec `i`, `j` ou `k`**. Ce sont de bons noms pour des variables qui sont juste des indices ; mais si c'est un indice qui a aussi un sens supplémentaire alors ce sont des noms catastrophiques. On en revient à la première règle : utiliser des noms explicites !
- **Choisissez une convention de casse et tenez-vous y**. Il existe deux grandes conventions pour écrire des noms de variables : séparer les sous-mots par des underscore (`par_exemple_cela` , on parle de **snake_case**), ou par des majuscules (`parExempleCela` , on parle de **camelCase**).

8. Même en TP à l'oral de concours : quand on découvre un bug dans une fonction écrite il y a 40min, il vaut mieux que le code soit bien écrit pour retrouver et corriger rapidement le bug.
9. Je précise : « court » ne signifie pas « une seule lettre ». Cela signifie « moins de dix lettres ».
10. Comme en maths où f désigne toujours les fonctions et x les arguments.

3.1 Structurer

- **Parenthésez pour lever les ambiguïtés**, ou pour accélérer la relecture du code. Ne parenthésez pas lorsque cela n'aide pas.
- **Identez votre code**. Python impose l'indentation (et c'est très bien) ; en C et OCaml il faut y faire attention. Voici des règles générales :
  - **À chaque entrée dans un bloc syntaxique, ajoutez un niveau d'indentation. À chaque sortie, enlevez-en un.** [11]
  - **Une indentation mesure toujours le même nombre d'espaces (2 ou 4)**.
  - **Utilisez des espaces ou des tabulations pour indenter, mais pas un mélange des deux**. Dans le cas des tabulations, les éditeurs de texte de développement proposent une option pour régler leur largeur ou pour les transformer automatiquement en espace.
  - **Une accolade fermante a le même niveau d'indentation que la ligne de l'accolande ouvrante associée**.
- **Utilisez des lignes vides pour séparer les blocs logiques de votre code**. Séparez toujours les fonctions du même nombre de lignes vides (typiquement, 2 ou 3). Dans vos fonctions, faites une ligne vide uniquement pour aider à la relecture en divisant la fonction en « paragraphes » (en groupes de lignes qui ont chacun un rôle précis). Inutile toutefois de découper une fonction si elle est déjà simple : laissez les choses simples être simples.
- **Gardez vos fonctions courtes : évitez de dépasser 20 lignes**.
- **Gardez vos lignes courtes : évitez de dépasser 80 caractères**.
- **Une ligne doit correspondre à une seule action**. N'écrivez pas $h(g(f(x)))$, c'est illisible. [12]
- **Si créer une variable rend le code plus lisible, créez une variable**. Une variable ne coûte (vraiment) pas cher [13] ; alors qu'ajouter un nom à une quantité intermédiaire peut aider à relire le code. En particulier, évitez les « nombres magiques » : si un nombre traine dans votre code sans que l'on comprenne son sens, c'est un mauvais code.
- **Déclarez les variables locales au plus proche de leur utilisation**. Lire une déclaration d'une variable locale qui ne servira que dans 20 lignes, c'est *fatigant*.
- **Si créer une fonction rend le code plus lisible, créez une fonction**. Cela permet de donner un nom explicite au rôle d'un ou plusieurs paragraphes du code.
- **Si plusieurs morceaux de code réalisent la même tâche, créez et utilisez une fonction réalisant cette tâche**. Cela évite la redondance de code qui est source d'erreur : si plusieurs morceaux de code font la même tâche, il est vite arrivé de mettre à jour un des morceaux de code mais pas l'autre et de créer ainsi un bug.
- **Découpez votre code en plusieurs fichiers**. Chaque fichier doit correspondre à une famille de tâche (définir une structure de données, définir des tests, etc). Cela simplifiera *vraiment* la relecture et le débogage.

3.2 Documenter

- **Spécifiez vos fonctions**. Cette spécification se fait typiquement dans l'interface.
- **Utilisez les commentaires pour expliquer les morceaux de code compliqué**. Un commentaire d'une ligne pour un paragraphe de code suffit en général : il faut simplement dire *à quoi servent* ces lignes de code. Ne commentez pas ce qui est déjà simple à relire, et ne paraphrasez pas le code.
- **Annotez les boucles compliquées d'un commentaire en donnant un invariant utile**. On le fait généralement en langage naturel et non en langage mathématique.
- **Si une fonction correspond à un algorithme compliqué, vous pouvez à la débuter par un gros commentaire explicatif**.

11. Notez que c'est ainsi que Python définit ses blocs syntaxiques.
12. En programmation orientée objet, c'est d'ailleurs une des sources les plus courantes de code inutilement compliqué à relire.
13. Et le compilateur peut les enlever lors de ses optimisations du code.

3.3 Anticiper

- **Sachez ce que vous allez coder avant de le coder : de quoi aurez-vous besoin, où et comment allez-vous stocker les données, comment les manipuler, etc ; en bref comment votre code sera organisé**. Si vous n'avez aucune idée du code à écrire, sortez du brouillon et réflechissez au brouillon.
- **Un bon code ne génère aucun Warning à la compilation**. En C, vous devez utiliser `-Wall -Wextra` pour avoir tous les warnings possibles.
- **Faites de la programmation défensive**. Utilisez des `assert` pour garantir que les pré-conditions sont vérifiées. Il faut que le non-respect des pré-conditions soit élémentaire à déboguer !
- **Garantissez l'exhaustivité des disjonctions de cas**. Toute disjonction de cas (resp. filtrage par motif) qui n'est pas trivialement exhaustive doit se terminer par un `else` (resp. par un `| _ ->`), quitte à ce que cette branche ne contienne qu'une levée d'exception. [14]
- **Faites des messages d'erreurs explicites**. Les `assert` c'est très bien, mais ils donnent un message d'erreur peu lisible [15]. Utiliser `if` qui affiche un message d'erreur avant d'interrompre le programme ou de lever une exception, c'est encore mieux.
- **Testez vos fonctions** !! Plus d'information à ce sujet dans la prochaine section.

3.4 Simplifier

- **K.I.S.S. K**eep **I**t **S**tupid **S**imple : *gardez votre code stupidement simple*. Ne vous surestimez pas, et utilisez tous les conseils précédents pour rendre le code stupidement simple. Évitez les optimisations qui rapportent peu mais nuisent à la lisibilité. Évitez les usines à gaz, concevez plutôt des fonctions et librairies élémentaires qui font une seule chose mais la font très bien.
- **Premature optimisation is the root of all evil** : *l'optimisation prématurée est la racine de tous les maux* (citation de D. Knuth). Faites d'abord un programme qui marche, et ensuite seulement un programme optimisé (et uniquement si nécessaire). N'ajoutez qu'une optimisation à la fois, de la plus simple à la plus compliquée - et testez à chaque fois ! Pour réaliser ces mises à jour successives sans pour autant perdre la version de base qui fonctionne, n'hésitez pas à faire correspondre cela à plusieurs implémentations d'une même interface.

3.5 Progresser

Vos autres conseils à vous-même ici :

14. Pensez par exemple à `List.hd` : si on lui donne une liste vide (ce que l'on est pas sensés faire), elle lève une exception.
15. Mieux vaut des `assert` que rien !!