Conseils pour programmer en Python... et en général

Voici quelques conseils pour programmer en Python, dont la majeure partie s'applique également à tout autre langage de programmation. Après avoir énoncé quelques grands principes, nous présenteront des méthodes pratiques, illustrées par des exemples. Ne vous laissez pas intimider par la longueur de ce document : elle n'est due qu'à l'extrême niveau de détail dans la présentation des exemples.

Principe 1 L'ordinateur est un exécutant idiot (mais puissant). L'activité fondamentale de la programmation consiste à décomposer un algorithme en actions suffisamment élémentaires pour qu'on puisse lui expliquer chacune d'entre elles.

Quand on donne des instructions à un être humain, même si l'explication n'est pas parfaitement complète et exacte, celui-ci utilise son intelligence et son esprit d'initiative pour compenser l'information manquante. À l'inverse, l'ordinateur suit les instructions à la lettre, avec la plus parfaite stupidité.

Exemple :

```
def pate_a_crepes():
    return mixer(500 g farine, 6 oeufs, 2 cs huile, 1 l lait,
        1 pincee sel)
```

Si nous donnons ces instructions à un être humain moyen, il n'y a pas de problème. En revanche, si nous donnons ce programme à un ordinateur, nous risquons de trouver des œufs entiers (coquilles comprises) dans notre pâte à crêpes. Il aurait fallu insérer les instructions suivantes : casser les œufs, jeter les coquilles à la poubelle et mettre uniquement *l'intérieur* des œufs dans la pâte...

Principe 2 Un même programme est toujours beaucoup plus difficile à débugger qu'à écrire. Il vaut mieux redoubler de prudence pendant l'écriture, quitte à y passer un peu plus de temps, plutôt que rester bloqué par la suite sur un bug incompréhensible.

Principe 3 Testez aussi souvent que possible. Autrement dit, n'attendez pas d'avoir écrit beaucoup de lignes avant de tester. C'est une conséquence du principe précédent. Moins vous ajoutez de code entre deux tests successifs, plus il est facile de localiser les bugs éventuels et de les corriger.

Principe 4 N'écrivez pas de façon linéaire (c'est-à-dire comme on lit, de la première à la dernière ligne). Généralement, il est plus facile de commencer par écrire un « squelette », à l'intérieur duquel on insère les détails dans un second temps.

Principe 5 Écrivez le cas général avant de réfléchir aux cas particuliers. Une fois le cas général écrit, vous verrez mieux quelles instances du problème il est capable de traiter et, à l'inverse, lesquelles doivent être considérées comme des cas particuliers.

Principe 6 Utilisez des exemples exemplaires. Pour réfléchir au cas général, prenez

un exemple raisonnablement éloigné des cas particuliers potentiels : vous verrez mieux le mécanisme à implanter. Si possible, considérez un exemple assez petit pour que vous puissiez lui appliquer manuellement l'algorithme (ou une partie de l'algorithme), afin de pouvoir comparer le déroulement de votre programme au déroulement attendu.

À présent, nous allons passer aux méthodes et aux exemples. Ceux que nous utiliseront sont volontairement très simples : ils permettent d'illustrer les méthodes tout en évitant des complications techniques inutiles. Une personne à l'aise en Python serait sans doute capable de les programmer plus rapidement qu'en suivant les conseils ci-dessous. Un programmeur chevronné pourrait même les écrire « à la volée », c'est-à-dire de la première à la dernière ligne sans s'arrêter. Mais les cas où l'on peut procéder ainsi sont très rares, même pour un programmeur expérimenté. Hormis pour des programmes extrêmement simples, les principes et les méthodes présentés dans ce document permettent d'éviter bien des problèmes, que ce soit pour le programmeur débutant ou chevronné.

1 Méthodes générales

Méthode 1 Vérifiez la syntaxe dans la console interactive. Si vous n'avez pas utilisé un élément de syntaxe depuis longtemps (ou jamais), vérifiez rapidement son fonctionnement dans la console interactive.

```
>>> 2^3
1
```

Hum... ce n'est pas ce que je souhaitais.

```
>>> 2**3
8
```

Ah oui, ça me revient : la fonction puissance est une double astérisque en Python.

Cette méthode découle des principes 2 et 3 : si vous intégrez la syntaxe `2^3` dans votre programme en croyant qu'il s'agit d'une puissance, sans l'avoir testée, ce bug risque d'être difficile à identifier. Dans la console interactive, c'est immédiat.

Méthode 2 Vérifiez la syntaxe sur Internet. C'est surtout utile dans les deux cas suivants.

1. Il vous manque le nom d'une commande Python.
2. Vous vous rappelez le nom de la commande désirée mais vous avez un doute sur sa syntaxe. Et malheureusement, un test rapide dans la console interactive n'a pas permis de vous éclairer.

Exemple : vous ne vous souvenez plus comment on calcule le logarithme en Python (ou vous n'avez jamais eu l'occasion d'utiliser cette fonction). Une recherche rapide sur Internet vous permettra de savoir qu'il suffit d'importer le module `math` et d'utiliser la fonction `math.log`.

Méthode 3 Faites des tests de principe dans la console interactive.

Exemple : vous souhaitez séparer une liste `list` en deux parties de même taille, à un élément près (par exemple, pour programmer un tri fusion). Cependant, comme Python numérote les éléments de `0` à `len(list) - 1`, vous avez un doute : cela va-t-il fonctionner si on fixe la séparation à `len(list) // 2` ? Un petit test permettra de s'en assurer. Pour en être sûr, il faut manifestement effectuer le test pour une liste de taille paire et pour une autre de taille impaire.

```
>>> list = [1, 2, 3, 4]
>>> list[:len(list) // 2]
[1, 2]
>>> list[len(list) // 2:]
[3, 4]
```

Pour une liste de taille paire, c'est bon !

```
>>> list = [1, 2, 3, 4, 5]
>>> list[:len(list) // 2]
[1, 2]
>>> list[len(list) // 2:]
[3, 4, 5]
```

Pour une liste de taille paire, ça fonctionne aussi ! Je peux utiliser ces expressions en toute confiance dans mon programme.

Méthode 4 Faites confiance à l'éditeur. Quand vous tapez le nom d'une commande, l'éditeur (IEP, PyCharm) vous rappelle quels arguments elle requiert.

En outre, un éditeur perfectionné (comme PyCharm) peut détecter certaines erreurs de syntaxe et souligner le code correspondant en rouge. Si vous passez votre souris à cet endroit, une info-bulle vous indiquera le type d'erreur détecté.

Méthode 5 Fermez vos parenthèses dès que vous les ouvrez. Idem pour les crochets, accolades, etc.

Exemple : assigner la valeur $\frac{a(b-c)}{d+e^{f(g+h)}}$ à la variable x. Voici un déroulement chronologique possible de votre travail.

```
x = () / ()
x = (a * ()) / ()
x = (a * (b - c)) / ()
x = (a * (b - c)) / (d + math.exp())
x = (a * (b - c)) / (d + math.exp(f * ()))
x = (a * (b - c)) / (d + math.exp(f * (g + h)))  # Final version
```

Même si cela peut paraître peu naturel de ne pas écrire tout simplement de gauche à droite, vous verrez à la longue que c'est beaucoup plus rapide et sûr de cette manière.

Méthode 6 Écrivez en anglais. Nommez vos variables en anglais et tapez vos commentaires en anglais. Dans la vie professionnelle, vous pourrez ainsi plus facilement partager

votre code (avec des collaborateurs internationaux et/ou sur Internet). Accessoirement, cela vous évitera la tentation d'utiliser des caractères accentués, qui sont susceptibles de poser des problèmes d'encodage.

2 Méthodes pour la récursivité

Méthode 7 Commencez par écrire le cas général. Le raisonnement est le suivant. Étant donné une instance du problème, je suppose que mon programme sait faire le travail pour toutes les instances strictement « plus petites ». Je m'autorise à exploiter ce fait pour traiter l'instance qui m'intéresse.

Exemple : calculer a^n, où a est un nombre quelconque et n un entier naturel. Si l'on imagine que le programme sait calculer a^{n-1}, alors il suffit de multiplier ce résultat par a. Voici donc à quoi ressemble la première étape de notre travail.

```
def power(a, n):
    ...
    return power(a, n - 1) * a
```

Méthode 8 Ensuite seulement, insérez les cas particuliers. Il s'agit d'identifier les cas où le raisonnement qu'on a fait pour le cas général n'est pas valable.

Exemple : calculer a^n (suite et fin).

- Si $n = 0$, alors notre programme va essayer de calculer `power(a, - 1) * a`. Or, il n'est pas prévu que notre programme sache calculer `power(a, - 1)`. Donc $n = 0$ doit être ajouté comme un cas particulier.
- Si $n = 1$, alors notre programme va essayer de calculer `power(a, 0) * a`. En supposant que notre programme fonctionne pour $n = 0$ (hypothèse de récurrence), le résultat sera $1 \times a = a$, ce qui est correct. Donc $n = 1$ rentre dans le cas général.

Après ces réflexions, voici notre programme final.

```
def power(a, n):
    if n == 0:
        return 1
    return power(a, n - 1) * a
```

Vous noterez qu'il n'est ici pas nécessaire d'utiliser `else` : en effet, `return 1` interrompt immédiatement le déroulement de la fonction. Si l'exécution parvient à la dernière ligne, c'est donc nécessairement qu'on est dans le cas général $n \neq 0$.

3 Méthodes pour les boucles

Méthode 9 Comment rédiger une boucle `for`.

Exemple : calculer la factorielle d'un entier naturel.

1. **Écrivez le squelette**, en incluant **l'action** à accomplir dans le pas de boucle.

```
def factorial(n):
    result = ...
    for i in range(...):
        result = result * i
    return result
```

2. **Déterminez les bornes de l'itération** : ceci est plus facile une fois que l'action est écrite. Ici, le premier i intéressant est 2 car il est inutile d'effectuer la multiplication par 1. La dernière multiplication à effectuer est par n. En Python, il ne faut pas indiquer la dernière valeur à inclure mais la première à exclure, donc il s'agira de $n + 1$.

```
def factorial(n):
    result = ...
    for i in range(2, n + 1):
        result = result * i
    return result
```

3. **Initialisez les variables annexes si nécessaire.** Pour cela, imaginons un exemple représentatif du cas général, disons $n = 3$. On sait que la boucle va multiplier `result` par 2 puis 3. Pour obtenir le résultat correct, `result` doit donc être initialisé à 1.

```
def factorial(n):
    result = 1
    for i in range(2, n + 1):
        result = result * i
    return result
```

4. **Vérifiez les cas particuliers** et traitez-les à part si nécessaire. Dans notre exemple, il est clair que si des gags se produisent, ce sera pour $n = 0$ ou $n = 1$ (on peut aussi vérifier $n = 2$ par acquis de conscience). Suivons mentalement le déroulement du programme dans ces deux cas.
   - Pour $n = 0$, `result` est initialisé à 1 puis on exécute `for i in range(2, 1)`, c'est-à-dire pour tous les entiers i tels que $i \geq 2$ et $i < 1$. Aucun entier i ne vérifie ces conditions, donc le pas de boucle n'est jamais exécuté et le programme renvoie 1.
   - Pour $n = 1$, de la même façon, le pas de boucle n'est jamais exécuté et le programme renvoie également 1.

   Dans les deux cas, il s'agit du résultat désiré, donc aucun cas particulier ne nécessite un traitement spécial dans cet exemple.

Méthode 10 Comment rédiger une boucle `while`.

Exemple : déterminer l'écriture d'un entier naturel en base 8. Tout d'abord, faisons quelques tests dans la console interactive, en choisissant un exemple exemplaire, c'est-à-dire raisonnablement éloigné d'un cas particulier. Nous allons construire cet exemple de façon à connaître la solution à l'avance : ainsi, nous pourrons vérifier que tout fonctionne.

```
>>> n = 3 * 8**2 + 5 * 8 + 2
>>> n
234
```

À présent, testons une ébauche de notre algorithme.

```
>>> (remains, digit) = divmod(n, 8)
>>> remains, digit
(29, 2)
>>> (remains, digit) = divmod(remains, 8)
>>> remains, digit
(3, 5)
>>> (remains, digit) = divmod(remains, 8)
>>> remains, digit
(0, 3)
>>> s = str(3) + str(5) + str(2)
>>> s
'352'
```

Maintenant que nous avons vérifié le principe, écrivons notre programme.

1. **Écrivez le squelette**, en incluant **l'action** à accomplir dans le pas de boucle. Par rapport à nos tests, ne nous précipitons pas sur ce que nous avons fait dans le premier cas, c'est-à-dire `(remains, digit) = divmod(n, 8)` : cela ne fonctionne qu'au début. Dans le cas général, nous avons utilisé l'instruction `(remains, digit) = divmod(remains, 8)`.

   ```
   def octal(n):
       s = ""
       remains = ...
       while ...:
           (remains, digit) = divmod(remains, 8)
           s = str(digit) + s
       return s
   ```

2. **Écrivez la condition du `while`** : comme pour une boucle `for`, ceci est plus facile une fois que l'action est écrite.

```
def octal(n):
    s = ""
    remains = ...
    while remains > 0:
        (remains, digit) = divmod(remains, 8)
        s = str(digit) + s
    return s
```

3. **Initialisez les variables.** Dans notre exemple, il suffit de regarder nos tests en console interactive pour voir que `remains` doit être initialisé à n.

```
def octal(n):
    s = ""
    remains = n
    while remains > 0:
        (remains, digit) = divmod(remains, 8)
        s = str(digit) + s
    return s
```

4. **Vérifiez les cas particuliers** et traitez-les à part si nécessaire. Dans notre exemple, il est clair que si des gags se produisent, ce sera pour $n = 0$ (on peut aussi vérifier $n = 1$ par acquis de conscience). Suivons mentalement le déroulement du programme dans ce cas. La variable s est initialisée à la chaîne de caractères vide, `remains` est initialisé à 0. La condition du `while`, c'est-à-dire `remains > 0`, est tout de suite fausse, donc le pas de boucle n'est pas exécuté et le programme renvoie s, c'est-à-dire la chaîne vide. Ce n'est pas ce que l'on veut : nous souhaiterions la chaîne `"0"`.

```
def octal(n):
    if n == 0:
        return "0"
    s = ""
    remains = n
    while remains > 0:
        (remains, digit) = divmod(remains, 8)
        s = str(digit) + s
    return s
```

Méthode 11 Comment rédiger une itération simple avec interruption, en utilisant `while`.

Il s'agit d'un type de boucle très courant, qu'on peut définir de la façon suivante.

- On itère sur une plage connue (entiers de 0 à $n-1$, éléments d'une liste, etc.). Si on arrive à la fin de cette plage, il faudra s'arrêter de toute façon.
- Mais il existe aussi une condition particulière qui peut faire sortir de la boucle avant d'avoir parcouru toute la plage (par exemple, on cherchait un élément et on l'a trouvé).

L'approche classique consiste à écrire ce type de boucle avec `while`, comme nous allons le faire tout de suite. En Python, il est également possible d'utiliser `for` et `break`, ce que nous détaillerons dans la méthode suivante.

Exemple : étant donné une liste dont les éléments 0 à $i-1$ sont triés, ranger l'élément d'indice i en l'insérant au bon endroit (opération de base d'un tri par insertion). Commençons par quelques tests en ligne de commande en choisissant un exemple simple mais raisonnablement éloigné d'un cas particulier.

```
>>> list = [1, 2, 4, 5, 6, 3, 42]
>>> i = 5
>>> list[i]
3
```

Notre exemple étant défini, commençons les tests.

```
>>> list[i], list[i - 1] = list[i - 1], list[i]
>>> list
[1, 2, 4, 5, 3, 6, 42]
>>> list[i - 1], list[i - 2] = list[i - 2], list[i - 1]
>>> list
[1, 2, 4, 3, 5, 6, 42]
>>> list[i - 2], list[i - 3] = list[i - 3], list[i - 2]
>>> list
[1, 2, 3, 4, 5, 6, 42]
```

À présent, nous pouvons rédiger la boucle.

1. **Écrivez le squelette**.
   (a) Écrivez **l'action** à accomplir. Par rapport à nos tests, ne nous précipitons pas sur ce que nous avons fait dans le premier cas, c'est-à-dire `list[i], list[i - 1] = list[i - 1], list[i]` : cela ne fonctionne qu'au début. Dans le cas général, nous avons utilisé une instruction du type `list[j], list[j - 1] = list[j - 1], list[j]`, pour une certaine valeur de j. À ce stade, il est conseillé d'écrire un commentaire qui explique en langage humain ce que représente *concrètement* la variable utilisée.

```
def sort_element_i(list, i):
    ...
    while ...:
        # j is the current position of element to sort
        list[j], list[j - 1] = list[j - 1], list[j]
        ...
    return list
```

(b) Écrivez aussitôt **l'incrémentation** (ou la décrémentation) pour ne pas l'oublier.

```
def sort_element_i(list, i):
    ...
    while ...:
        # j is the current position of element to sort
        list[j], list[j - 1] = list[j - 1], list[j]
        j -= 1
    return list
```

2. **Écrivez la condition du `while`.** Quels sont les cas où l'on veut effectuer le pas de boucle ? Premièrement, il faut qu'on ait $j > 0$: en effet, si on avait $j = 0$, on devrait échanger `list[0]` avec `list[-1]`, ce qui n'est pas pertinent. Deuxièmement, il faut que l'élément soit mal rangé, c'est-à-dire qu'on ait `list[j - 1] > list[j]`.

```
def sort_element_i(list, i):
    j = ...
    while j > 0 and list[j - 1] > list[j]:
        # j is the current position of element to sort
        list[j], list[j - 1] = list[j - 1], list[j]
        j -= 1
    return list
```

3. **Initialisez les variables**. Ici, comme nous avons commenté clairement ce que représente j, cela nous facilite le travail. Puisqu'il s'agit de la position actuelle de l'élément à trier, sa valeur initiale est i.

```
def sort_element_i(list, i):
    j = i
    while j > 0 and list[j - 1] > list[j]:
        # j is the current position of element to sort
        list[j], list[j - 1] = list[j - 1], list[j]
        j -= 1
    return list
```

4. **Vérifiez les cas particuliers** et traitez-les à part si nécessaire. Dans notre exemple, on peut vérifier par acquis de conscience le cas $i = 0$. Il est facile de voir que dans ce cas, on renvoie la liste sans modification, ce qui est bien le comportement désiré. Il n'y a donc aucun cas particulier à traiter ici.

Méthode 12 Comment rédiger une itération simple avec interruption, en utilisant `for` et `break`.

En Python, l'instruction `break` interrompt immédiatement l'exécution de la boucle en cours. S'il y a plusieurs boucles imbriquées, elle n'interrompt que la boucle la plus « à l'intérieur », c'est-à-dire celle avec le plus grand niveau d'indentation.

Pour une itération simple avec interruption, ceci donne une façon de rédiger qui est généralement plus simple qu'en utilisant `while`. Afin de pouvoir comparer, nous allons considérer le même exemple que dans la méthode précédente.

1. **Écrivez le squelette**, en incluant **l'action** à accomplir dans le pas de boucle.

```
def sort_element_i(list, i):
    for j in range(...):
        # j is the current position of element to sort
        if list[j] < list[j - 1]:
            list[j], list[j - 1] = list[j - 1], list[j]
        else:
            break
    return list
```

2. **Déterminez les bornes de l'itération.** De la même façon que dans la méthode précédente, il faut aller de $j = i$ à $j = 1$ en décrémentant j à chaque pas de boucle. Le premier j à exclure est donc 0.

```
def sort_element_i(list, i):
    for j in range(i, 0, -1):
        # j is the current position of element to sort
        if list[j] < list[j - 1]:
            list[j], list[j - 1] = list[j - 1], list[j]
        else:
            break
    return list
```

3. **Initialisez les variables annexes si nécessaire.** Dans cet exemple, il n'y en a pas.
4. **Vérifiez les cas particuliers** et traitez-les à part si nécessaire. Ici, on peut vérifier le cas $i = 0$ par acquis de conscience, mais il n'y a pas de problème.

Méthode 13 Quel type de boucle choisir ? Tout d'abord, notons qu'il est *toujours* possible de traduire une boucle `for` par une boucle `while`, et qu'il est *souvent* possible de traduire une boucle `while` par une boucle `for`. En théorie, il est donc fréquent que les deux solutions soient possibles. Mais en pratique, un choix éclairé facilitera l'écriture du programme et limitera les risques d'erreur.

Dans la plupart des langages de programmation, il y a essentiellement deux cas, qu'on peut distinguer par cette question : connaît-on à l'avance le nombre de pas de boucles à effectuer ?

- Oui : en général, il est plus pratique d'utiliser un `for`.
- Non : en général, il est nécessaire d'utiliser un `while`.

En Python, le `for` est très puissant grâce à diverses améliorations (interruption par `break` ou `return`, itération sur les éléments d'une liste, sur ceux d'un dictionnaire, etc.). En conséquence, le `while` est plus rarement utilisé que dans d'autres langages car un `for` est généralement plus facile à écrire et conduit à un programme plus court et souvent plus lisible.

En particulier, nous avons vu le cas extrêmement fréquent d'une itération simple avec interruption : même s'il est possible d'écrire une telle boucle avec `while`, il est généralement plus facile de le faire avec `for` et `break`.

Méthode 14 Documentez les variables de boucles. Comme nous l'avons déjà dit, donnez-leur un sens le plus *concret* possible, cela vous aidera à écrire votre programme.

Éventuellement, indiquez quelle propriété vous souhaitez voir vérifiée en début et/ou en fin de boucle. D'un point de vue théorique, cette approche prépare déjà la preuve de l'algorithme (par la mise en évidence d'un invariant de boucle). En pratique, cela vous aidera à savoir exactement quelles actions il faut effectuer dans chaque pas de boucle.

Exemple : tri par insertion. Écrivons d'abord le squelette du programme.

```
def insertion_sort(list):
    for i in range(...):
        # Elements 0 to i - 1 are sorted
        # i : initial position of element to sort
        ...
        # Elements 0 to i are sorted
    return list
```

Ensuite, écrivons l'action à accomplir pour chaque pas de boucle. Il suffit de copier-coller le code de la fonction `sort_element_i` vue plus haut (en adaptant l'indentation).

```
def insertion_sort(list):
    for i in range(...):
        # Elements 0 to i - 1 are sorted
        # i : initial position of element to sort
        for j in range(i, 0, -1):
            # j is the current position of element to sort
            if list[j] < list[j - 1]:
                list[j], list[j - 1] = list[j - 1], list[j]
            else:
                break
        # Elements 0 to i are sorted
    return list
```

Déterminons les bornes de l'itération. On peut commencer à $i = 1$: en effet, la liste constituée des éléments d'indice 0 à $i - 1 = 0$ ne contient qu'un élément donc elle est déjà triée.

```
def insertion_sort(list):
    for i in range(1, len(list)):
        # Elements 0 to i - 1 are sorted
        # i : initial position of element to sort
        for j in range(i, 0, -1):
            if list[j] < list[j - 1]:
                list[j], list[j - 1] = list[j - 1], list[j]
            else:
                break
        # Elements 0 to i are sorted
    return list
```

4 Méthodes pour débugger

Méthode 15 Quand le programme ne compile pas du tout...

Il s'agit d'un problème de syntaxe. Lisez le message d'erreur : à quelle ligne intervient-elle ? De quel catégorie d'erreur s'agit-il ? Voici les catégories les plus courantes.

- `TypeError` : par exemple, vous essayez d'additionner un entier avec une chaîne de caractères. Faites un test en console interactive, ou ajoutez un `print` des variables impliquées juste avant la ligne fautive pour comprendre ce qu'il se passe.
- `IndexError` : vous utilisez un indice qui est hors de la liste considérée. Là encore, un `print` des variables impliquées juste avant l'erreur vous permettra d'en comprendre la raison.

Parfois, Python produit une erreur apparemment incompréhensible, sur une ligne qui semble *a priori* correcte : attention, il peut s'agir d'une parenthèse non fermée à la ligne *précédente*.

Méthode 16 Quand le programme compile mais semble ne pas terminer...

Deux possibilités :

- Vous lui avez donné un exemple trop grand à traiter et votre programme est lent. Essayez avec un exemple plus petit.
- Le programme est parti en boucle infinie. Ajoutez des `print` afin de comprendre pourquoi.

Méthode 17 Quand le programme compile mais ne donne pas le bon résultat...

Deux méthodes complémentaires :

- Exécutez votre programme mentalement sur un exemple simple.
- Ajoutez des `print` pour suivre son déroulement plus facilement.