Bonnes pratiques d'écriture de projet

Judicaël Courant

Lycée La Martinière-Monplaisir, Lyon

18 novembre 2018

Lignes directrices

1. Introduction
2. Génie logiciel
3. Longueur des lignes
4. Identificateurs
5. Les nombres magiques
6. Conditionnelles
7. Les commentaires
8. Le dire une seule fois
9. Les idiomes de Python
10. Conclusion
11. Copyleft

Quelques prescriptions et interdits

Une prescription classique

Il faut tout commenter.

Un interdit classique

return *au milieu d'une fonction, c'est mal. D'ailleurs dans d'autres langages de programmation ça n'existe pas.*

Variantes

- `break` c'est mal
- `for x in t` c'est mal
- `.append` c'est mal
- ...

À propos des interdits en programmation

Origine

- Des choix véritables
- Des reflets d'une éducation parfois trop stricte (en Pascal ?)

Un monde en évolution (lente)

$\approx$1950 invention de la programmation structurée

1958 **`for`** et **`while`** dans ALGOL.

1968 Article de Dijkstra contre les `goto`.

1985 Pas de **`while`** ni de fonctions dans le BASIC du MO5.

Conséquence

☹ *Pas de règle immuable.*

Lignes directrices

1. Introduction
2. Génie logiciel
3. Longueur des lignes
4. Identificateurs
5. Les nombres magiques
6. Conditionnelles
7. Les commentaires
8. Le dire une seule fois
9. Les idiomes de Python
10. Conclusion
11. Copyleft

Cycle de vie d'un programme

Vie d'un programme

- Gestation : conception et développement initial
- Naissance : mise en service
- Croissance :
  - Correction de bugs
  - Adaptation à la plateforme
  - Évolutions fonctionnelles
- Mort : retrait du logiciel

Lignes de programme

Écrites Une fois ;

Lues De nombreuses fois.

Point de vue économique

Coûts industriels

Matériel Faibles (et en baisse)

Logiciel Élevés (et plutôt en hausse)

Dette technique

Écrire vite et mal un programme est :

Une tentation Croire gagner du temps

Une erreur C'est un *emprunt*, pas un gain

Une dette Payable le reste de sa vie

Optimisation

- Optimisation de
  - la vitesse d'exécution
  - la consommation mémoire
- Antinomique avec la clarté
- Coûte du temps de développement
- Goulots d'étranglement : situation fréquente
- Difficiles à localiser à l'avance

Conséquence

- Travailler d'abord à avoir un code clair qui fonctionne.
- On pourra l'optimiser ensuite.
- L'inverse n'est pas vrai.

Pour les étudiants

Objectif

Écrire des programmes clairs.

Remarque

> *Any fool can write code that a computer can understand.*
> *Good programmers write code that humans can understand.*
> *– Martin Fowler*

Moyens

Interdictions Génèrent incompréhensions et contournements

Autorisations Donner aux étudiants les moyens d'être auteurs

Règles ~~Dogmes divins~~ Guides pour stimuler la réflexion

Quelques facteurs de clarté

- Un bon programme est une communication entre *humains*.
- Questions habituelles en matière de communication :

Concision	À rechercher mais pas à l'excès.
Conventions	Facilitent la compréhension. Souvent implicites.
Idiomes	À connaître et savoir reconnaître.
Cohérence	Plus importante que le choix des conventions.

Remarque

L'art poétique (Boileau, 1674)

Selon que notre idée est plus ou moins obscure,
L'expression la suit, ou moins nette, ou plus pure.
Ce que l'on conçoit bien s'énonce clairement,
Et les mots pour le dire arrivent aisément.

Remarque

L'art poétique (Boileau, 1674)

Selon que notre idée est plus ou moins obscure,
L'expression la suit, ou moins nette, ou plus pure.
Ce que l'on conçoit bien s'énonce clairement,
Et les mots pour le dire arrivent aisément.

Notion de «code qui pue» :

- **Odeur** Symptôme de cadavres dans les placards
- **Objectif** Trouver le cadavre (et non masquer l'odeur)

Réusinage de code

Mal écrire un programme initialement est :

Réusinage de code

Mal écrire un programme initialement est :

- Normal : on ne comprend pas le problème.

Réusinage de code

Mal écrire un programme initialement est :

- Normal : on ne comprend pas le problème.
- Une étape initiale : c'est en écrivant et récrivant qu'on comprend.

Réusinage de code

Mal écrire un programme initialement est :

- Normal : on ne comprend pas le problème.
- Une étape initiale : c'est en écrivant et récrivant qu'on comprend.
- Sans gravité : le premier jet n'est qu'un *brouillon*.

Réusinage de code

Mal écrire un programme initialement est :

- Normal : on ne comprend pas le problème.
- Une étape initiale : c'est en écrivant et récrivant qu'on comprend.
- Sans gravité : le premier jet n'est qu'un *brouillon*.

Refactoring (réusinage de code)

- Opération consistant à retravailler le code d'un programme.
- Il y a des livres entiers sur le sujet.
- Responsabilité de l'enseignant/chef de projet :

Réusinage de code

Mal écrire un programme initialement est :

- Normal : on ne comprend pas le problème.
- Une étape initiale : c'est en écrivant et récrivant qu'on comprend.
- Sans gravité : le premier jet n'est qu'un *brouillon*.

Refactoring (réusinage de code)

- Opération consistant à retravailler le code d'un programme.
- Il y a des livres entiers sur le sujet.
- Responsabilité de l'enseignant/chef de projet :

```
while not est_clair(programme):
    faire_recrire(programme)
```

Avertissement

- Un catalogue de règles va suivre.
- Ce ne sont que des guides, pas des dogmes.
- Exemples : extraits (simplifiés) de projets.

Lignes directrices

1. Introduction
2. Génie logiciel
3. **Longueur des lignes**
4. Identificateurs
5. Les nombres magiques
6. Conditionnelles
7. Les commentaires
8. Le dire une seule fois
9. Les idiomes de Python
10. Conclusion
11. Copyleft

Longueur des lignes

Lignes trop longues :

- coupées à l'impression
- ou reviennent à la ligne (illisibles)

Exemple :

```
couleur_direction7 = [plateau[i,j-k] for k in range(1,2*n)
index_direction7 = [[i,j-k] for k in range(1,2*n) if (j-k)
couleur_direction8 = [plateau[i-k,j-k] for k in range(1,2*
index_direction8 = [[i-k,j-k] for k in range(1,2*n) if (i-
return [couleur_direction1,couleur_direction2,couleur_dire
```

Exemple (retours à la ligne forcés)

```
couleur_direction7 = [plateau[i,j-k] for k in
↪  range(1,2*n) if (j-k) >= 0]
index_direction7 = [[i,j-k] for k in range(1,2*n) if
↪  (j-k) >= 0]
couleur_direction8 = [plateau[i-k,j-k] for k in
↪  range(1,2*n) if (i-k) >= 0 and (j-k) >= 0]
index_direction8 = [[i-k,j-k] for k in range(1,2*n) if
↪  (i-k) >= 0 and (j-k) >= 0]
return [couleur_direction1,couleur_direction2,couleur_d⌋
↪  irection3,couleur_direction4,couleur_direction5,cou⌋
↪  leur_direction6,couleur_direction7,couleur_directio⌋
↪  n8],[index_direction1,index_direction2,index_direct⌋
↪  ion3,index_direction4,index_direction5,index_direct⌋
↪  ion6,index_direction7,index_direction8]
```

Lignes trop longues

Quand une ligne est trop longue :

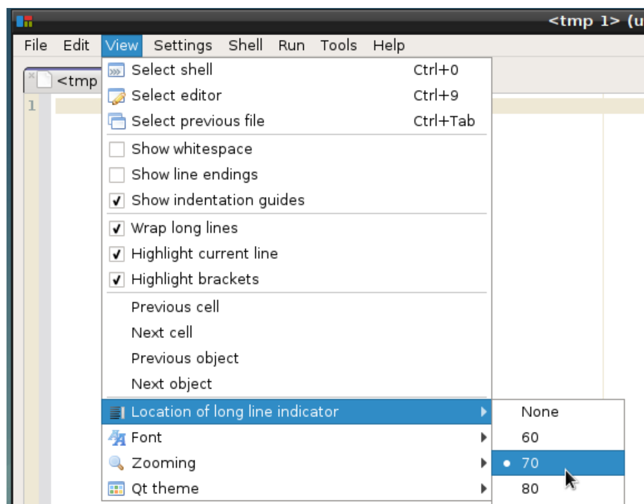
- il ne suffit pas de la découper
- il faut la récrire (mieux)

Règle (PEP 8)

Commentaires Formater à 72 caractères.

Code Une ligne de plus de 79 caractères pue.

Réglage avec pyzo



Lignes directrices

1. Introduction
2. Génie logiciel
3. Longueur des lignes
4. Identificateurs
5. Les nombres magiques
6. Conditionnelles
7. Les commentaires
8. Le dire une seule fois
9. Les idiomes de Python
10. Conclusion
11. Copyleft

Identificateurs

Une prescription classique

Il faut choisir des noms de variables/fonctions aussi explicites que possibles.

Est-ce vraiment une bonne idée ?

Exemple

Que fait ce code ?

```
def echanger_deux_bonbons(G,coord1,coord2):
    W=np.array(G)
    # On intervertit les positions de coord1 et coord2
    W[coord1[0],coord1[1]],W[coord2[0],coord2[1]]=W[coo
    ↪  rd2[0],coord2[1]],W[coord1[0],coord1[1]]
    if coord1[0]==coord2[0] and
    ↪  abs(coord1[1]-coord2[1])==1:
        #On vérifie si les deux bonbons échangés sont
        ↪  sur la
        # même ligne et côte à côte
        ...
```

Exemple (suite)

Et celui-ci ?

```
def echanger_deux_bonbons(G, c1, c2):
    W = np.array(G)
    i1, j1 = c1
    i2, j2 = c2
    W[i1, j1], W[i2,j2] = W[i2, j2], W[i1, j1]
    if i1 == i2 and abs(j1 - j2) == 1:
        ...
```

Exemple (suite)

Et celui-ci ?

```
def echanger_deux_bonbons(G, c1, c2):
    W = np.array(G)
    i1, j1 = c1
    i2, j2 = c2
    W[i1, j1], W[i2,j2] = W[i2, j2], W[i1, j1]
    if i1 == i2 and abs(j1 - j2) == 1:
        ...
```

Remarque

On peut même écrire

```
W[c1], W[c2] = W[c2], W[c1]
```

si `c1` et `c2` sont des couples (et non des listes)

Exemple 2

Que fait ce code ?

```
def matrice_2():

    plateau = matrice_1()

    PIECEABOUGER = [[7, 5], [5, 7], [5, 11], [7, 13], [11,

    for i in(PIECEABOUGER):

        abscisse = i[0]

        ordonnee = i[1]

        plateau[abscisse - 1, ordonnee], plateau[abscisse,

        plateau[abscisse, ordonnee + 1], plateau[abscisse

    return plateau
```

Exemple 2 (suite)

Et celui-ci ?

```
PIECEABOUGER = [ [7,  5], [ 5,  7], [ 5, 11], [ 7, 13],
                 [11, 13], [13, 11], [13,  7], [11,  5]]

def matrice_2():
    p = matrice_1()
    for c in PIECEABOUGER:
        i, j = c
        p[i-1, j], p[i, j-1] = p[i, j-1], p[i-1, j]
        p[i, j+1], p[i+1, j] = p[i+1, j], p[i, j+1]
    return p
```

Exemple 2 (suite)

Et celui-ci ?

```
PIECEABOUGER = [ [7,  5], [ 5,  7], [ 5, 11], [ 7, 13],
                [11, 13], [13, 11], [13,  7], [11,  5]]

def matrice_2():
    p = matrice_1()
    for c in PIECEABOUGER:
        i, j = c
        p[i-1, j], p[i, j-1] = p[i, j-1], p[i-1, j]
        p[i, j+1], p[i+1, j] = p[i+1, j], p[i, j+1]
    return p
```

Remarque

- Plus besoin d'aérer avec des lignes vides.
- On pourrait introduire une fonction `echange(m, c1, c2)` pour échanger deux cases d'une matrice.

Identificateurs (suite)

Règle

Objets globaux Nom signifiants (plus facile à retenir).

Objets locaux Nom courts (pour les lire facilement).

Identificateurs (fin)

`NomLong` ou `nom_long` ou `nomlong` ?

- En Python, c'est de préférence `nomlong` pour les variables (globales) et les fonctions, et `nom_long` quand c'est plus lisible.
- Préférer `NOM_LONG` pour les constantes.
- Voir PEP 20 pour tous les détails.

Identificateurs (fin)

`NomLong` ou `nom_long` ou `nomlong` ?

- En Python, c'est de préférence `nomlong` pour les variables (globales) et les fonctions, et `nom_long` quand c'est plus lisible.
- Préférer `NOM_LONG` pour les constantes.
- Voir PEP 20 pour tous les détails.

Remarque

- Cohérence : prime sur le choix de la convention.

Identificateurs (fin)

`NomLong` ou `nom_long` ou `nomlong` ?

- En Python, c'est de préférence `nomlong` pour les variables (globales) et les fonctions, et `nom_long` quand c'est plus lisible.
- Préférer `NOM_LONG` pour les constantes.
- Voir PEP 20 pour tous les détails.

Remarque

- Cohérence : prime sur le choix de la convention.

Envie de vous arracher les cheveux ?

- Prenez deux fonctions presque identiques
- Appelez l'une `ma_fonction(x, y)`
- L'autre `mafonction(x, y)`
- Revenez trois jours plus tard

Lignes directrices

1. Introduction
2. Génie logiciel
3. Longueur des lignes
4. Identificateurs
5. **Les nombres magiques**
6. Conditionnelles
7. Les commentaires
8. Le dire une seule fois
9. Les idiomes de Python
10. Conclusion
11. Copyleft

Les nombres magiques

Exemple de code peu clair

```
T=np.zeros((81,81))
for N in range(81):
    [i,j]=inversef(N)
    ...
    elif i==17:
        if j==1:
            T[N,f([i,j+2])]=1
            T[N,f([i-2,j])]=1
```

- Que représentent 2, 17 et 81 ?
- Le nombre 1 a-t-il la même signification dans `j==1` et `T[...] = 1` ?

Les nombres magiques

Remplacer les nombres magiques par des constantes en début de programme.

```
NB_CASES = 81
NB_COL = 17
VOISINES = 1
...
T=np.zeros((NB_CASES, NB_CASES))
for n in range(NB_CASES):
    [i,j]=inversef(n)
    ...
    elif i == NB_COL:
        if j == 1:
            T[N,f([i,j+2])] = VOISINES
            T[N,f([i-2,j])] = VOISINES
```

Remarque

Un code où des valeurs autres que -1, 0, 1 et 2 apparaissent pue.

Lignes directrices

1. Introduction
2. Génie logiciel
3. Longueur des lignes
4. Identificateurs
5. Les nombres magiques
6. Conditionnelles
7. Les commentaires
8. Le dire une seule fois
9. Les idiomes de Python
10. Conclusion
11. Copyleft

Remarque

- Les conditionnelles sont liées aux booléens.
- Il *faut* comprendre les booléens pour programmer.
- En particulier :
  - Les booléens sont des objets comme les autres.
  - `return x > 0` devrait être aussi naturel que `return x+2`.
  - Tant qu'un étudiant écrit

    ```
    if x > 0:
        return True
    else:
        return False
    ```

    c'est qu'il lui reste encore à comprendre. . .

- `x == True` doit être remplacé par `x`
- `x == False` doit être remplacé par **`not`** `x`

Exemple

```
if not est_valide(x):
    ...
```

se lit : «si x n'est pas valide ... »

Booléens : exemple

```
if verifplacegrandplateau (PLATEAUJOUEUR, PION, A, i, j) ==
    if verifpetitplateau (PLATEAUJOUEUR, PION, A, i, j) ==
        if verifplateauadv (PLATEAUJOUEUR, PLATEAUADV1,
                            PLATEAUADV2, PLATEAUADV3, PION,
                            i, j) == True :
            PLACEVERIFIEE.append([A, [i, j]])
```

s'améliore en :

```
if place_grand_plateau_correct(pj, p, a, i, j) \
   and petit_plateau_correct (pj, p, a, i, j) \
   and plateau_adv_correct(pj,pa1,pa2,pa3,p,a,i,j):
            pv.append([a, [i, j]])
```

(il reste trop de variables : mauvais choix de modélisation)

Conditionnelles imbriquées

- Éviter d'imbriquer des conditionnelles
- En particulier un `if` juste après un `if`, `elif` ou `else` doit être changé.

Conditionnelles imbriquées : exemple

```
if i==1:
    if j==1: action(1)
    elif j==17: action(2)
    else: action(3)
elif i==17:
    if j==1: action(4)
    elif j==17: action(5)
    else: action(6)
else:
    if j==1: action(7)
    elif j==17: action(8)
    else: action(9)
```

(actions des conditionnelles remplacées par `action(1), ...`)

Conditionnelles imbriquées : exemple (2)

Se récrit en:

```
if   (i, j) == ( 1,  1): action(1)
elif (i, j) == ( 1, 17): action(2)
elif i == 1            : action(3)
elif (i, j) == (17,  1): action(4)
elif (i, j) == (17, 17): action(5)
elif i == 17           : action(6)
elif j ==  1           : action(7)
elif j == 17           : action(8)
else                   : action(9)
```

Ce qui met en évidence qu'il y aurait un ordre plus naturel

Conditionnelles imbriquées : exemple (3)

Après réordonnancement :

```
if   (i, j) == ( 1,  1): action(1)
elif (i, j) == ( 1, 17): action(2)
elif (i, j) == (17,  1): action(4)
elif (i, j) == (17, 17): action(5)
elif i == 1            : action(3)
elif i == 17           : action(6)
elif j ==  1           : action(7)
elif j == 17           : action(8)
else                   : action(9)
```

Lignes directrices

1. Introduction
2. Génie logiciel
3. Longueur des lignes
4. Identificateurs
5. Les nombres magiques
6. Conditionnelles
7. **Les commentaires**
8. Le dire une seule fois
9. Les idiomes de Python
10. Conclusion
11. Copyleft

Commentaires : objectif et risque

Objectif

Améliorer la clarté du programme

Risque

Obscurcir le programme :

- en ajoutant du verbiage (paraphrase)
- en dispensant le programmeur d'écrire proprement
- en induisant le lecteur en erreur

Devinette

Point commun entre un programme et une blague ?

Commentaires : objectif et risque

Objectif

Améliorer la clarté du programme

Risque

Obscurcir le programme :

- en ajoutant du verbiage (paraphrase)
- en dispensant le programmeur d'écrire proprement
- en induisant le lecteur en erreur

Devinette

Point commun entre un programme et une blague ?
S'il y a besoin de les expliquer, ce n'est pas terrible.

Éviter la paraphrase

Ce commentaire n'apporte rien au code, sauf si on ne connaît pas l'instruction `+=` :

```
x += 1 # on augmente x
```

Éviter la paraphrase

Ce commentaire n'apporte rien au code, sauf si on ne connaît pas l'instruction += :

```
x += 1 # on augmente x
```

Remarque

Un commentaire n'est pas là pour expliquer une notion de Python.

Commenter n'excuse pas un mauvais programme

Ce programme doit être récrit :

```
if i==1: #S'il est sur la première ligne
    if j==1: #S'il est sur la première colonne
        T[N,f([i,j+2])]=1 #Lien avec son voisin de droite
        T[N,f([i+2,j])]=1 #Lien avec son voisin de dessous
    elif j==17: #S'il est sur la dernière colonne
        T[N,f([i,j-2])]=1 #Lien avec son voisin de gauche
        T[N,f([i+2,j])]=1 #Lien avec son voisin de dessous
    else:
        T[N,f([i,j+2])]=1 #Lien avec son voisin de droite
        T[N,f([i,j-2])]=1 #Lien avec son voisin de gauche
        T[N,f([i+2,j])]=1 #Lien avec son voisin de dessous
# ... plus 25 lignes similaires
```

Ne jamais contredire le programme

C'est pire que pas de commentaire du tout !

```
while len(F_Celibataires) > 0:
    # on parcourt la liste des hommes célibataires
    for f in F_Celibataires:
        # on nomme la femme préférée de l'homme i
        h_pref = FC[f][0]
        ...
```

Quoi commenter

- Expliquer les structures de données utilisées
- Spécifier les fonctions
- Quelques rares commentaires dans le corps des fonctions

Structures de données

> *Si vous me montrez vos fonctions et me cachez vos structures de données, je resterai perplexe. Si vous me montrez vos structures de données, je n'aurai pas besoin de vos fonctions : elles seront évidentes.*
> *– F. Brooks, The Mythical Man-Month (traduction moderne)*

Points fondamentaux :

- Choix de modélisation d'un problème
- Choix des structures de données

En conséquence, expliquer :

- Ce qu'on modélise
- Comment on le modélise

Structure de données : exemple

```
# On modélise le problème des N dames
# https://fr.wikipedia.org/wiki/Probl%C3%A8me_des_huit_dames
N = 8
```

Variante 1 :

```
# Les coordonnées sur l'échiquier seront représentées par
# des couples (i, j) (numéro de ligne, numéro de colonne),
# la numérotation commençant à zéro.
# On représente la solution en cours de construction par un
# tableau t de longueur au plus N donnant les coordonnées
# de chaque dame déjà placée.
```

Variante 2 :

```
# On numérote les dames de 0 à N-1. La dame numéro i est
# placée ligne i. La solution en cours de construction est
# représentée par un tableau t de longueur N, où t[i] donne
# la colonne de la dame numéro i, et None si la dame n'est
# pas encore placée.
```

Spécification des fonctions

En tête de fonction donner la *spécification* de la fonction.

Spécification :

- **Contrat** Entre fournisseur et clients
- **Fournisseur** Celui qui écrit la fonction
- **Clients** Ceux qui l'utilisent

Fournisseur :

- doit respecter sa part du contrat
- dégagé de toute responsabilité si le client ne respecte pas sa part
- *libre* de changer son code à tout moment

Les clients doivent compter sur :

- le contrat, *seulement* le contrat
- *pas* sur la cuisine interne du fournisseur

Spécification des fonctions : exemple

```python
def plateau(n, p):
    """Retourne un plateau de jeu de taille n*n,
    avec un nombre d'ennemis p.
    Préconditions : n >= 0 et 0 <= p <= n**2 - 1"""
```

- Les utilisateurs ne doivent rien attendre de plus que ce contrat.
- Si `plateau` est appelé avec $p < 0$, tout peut arriver (boucle infinie, retour de données incohérentes, plantage, etc.)

Remarque

Il faut avoir expliqué auparavant les notions :

- plateau de jeu
- ennemis

Qu'est-ce qu'une bonne spécification ?

- Elle permet d'utiliser la fonction sans aller lire le code.
- Elle donne aux clients toutes les propriétés dont ils ont besoin.

Spécification : les mots qui sentent mauvais

Ces mots sont signes d'une spécification mal écrite :

Cette fonction permet de
: Introduction d'une explication trop vague ; on veut savoir ce qu'elle *fait* pas ce qu'elle *permet*.

Cette fonction est utilisée par
: Ça ne dit pas ce qu'elle fait (et les clients peuvent changer)

Cette fonction utilise la fonction
: Les clients s'en moquent (cuisine interne)

Dans le corps d'une fonction

Deux types de commentaires classiques :

- Références
- Invariants

Références bienvenues

Un commentaire est utile pour donner une référence expliquant un choix :

- algorithme publié
- règle du jeu
- cadre légal

```
# art 194 du CGI au 16/10/2018
if nb_enfants < 3:
    parts += nb_enfants * 0.5
else:
    parts += 2 * 0.5 + (nb_enfants - 2) * 1
```

Invariant

- Notion générale utilisée pour montrer la correction de programmes
- Hors programme BCPST mais au programme du tronc commun MPSI/PCSI/PTSI
- Utile pour les étudiants de BCPST (même sans faire de preuve de programme)

Qu'est-ce que c'est

Une propriété (mathématique) :

- exprimée à un endroit du programme (sous forme de commentaire)
- vraie à chaque fois que l'exécution passe à cet endroit

Exemple

```python
def fibo(n):
    """Renvoie F(n) où F est la suite de Fibonacci.
    Précondition : n >= 0."""
    a = 0
    b = 1
    for i in range(n):
        # a == F(i), b == F(i+1)
        a, b = b, a+b
    return a
```

Exemple élémentaire

```
if x >= 0 and y >= 0:
    ...
elif x >= 0 and y < 0:
    ...
elif y >= 0:
    # x < 0
    ...
else:
    # x < 0 et y < 0
    ...
```

Lignes directrices

1. Introduction
2. Génie logiciel
3. Longueur des lignes
4. Identificateurs
5. Les nombres magiques
6. Conditionnelles
7. Les commentaires
8. Le dire une seule fois
9. Les idiomes de Python
10. Conclusion
11. Copyleft

Quand on répète plusieurs fois la même chose. . .

```
def directions(i, j, plateau):
    couleur_direction1 = [plateau[(i-k),j]for k in range(1,2*n) i
    couleur_direction2 = [plateau[i-k,j+k] for k in range(1,2*n)
    couleur_direction3 = [plateau[i,j+k] for k in range(1,2*n) if
    couleur_direction4 = [plateau[i+k,j+k] for k in range(1,2*n)
    couleur_direction5 = [plateau[i+k,j] for k in range(1,2*n) if
    couleur_direction6 = [plateau[i+k,j-k] for k in range(1,2*n)
    couleur_direction7 = [plateau[i,j-k] for k in range(1,2*n) if
    couleur_direction8 = [plateau[i-k,j-k] for k in range(1,2*n)
    return [couleur_direction1,couleur_direction2,couleur_directi
```

Il faut factoriser

```
TAILLE_PLATEAU = 8
DIRECTIONS = [(di, dj) for di in (-1,0,1)
                       for dj in (-1,0,1) if (di, dj) != (0,0)]

def est_valide(i, j):
    """Dit si la case de coordonnées (i, j) existe"""
    return 0 <= i < TAILLE_PLATEAU and 0 <= j < TAILLE_PLATEAU

def directions(i, j, p):
    dirs = []
    for di, dj in DIRECTIONS:
        c = [(i + k*di, j + k*dj) for k in range(1, 2*n)]
        dirs.append([p[i, j] for i,j in c if est_valide(i, j)])
    return dirs
```

Évolution du programme

Finalement, on veut retourner non seulement les `p[i,j]` mais aussi les `[i, j]` :

Évolution du programme

Finalement, on veut retourner non seulement les `p[i,j]` mais aussi les `[i, j]` :

```
def directions(i, j, plateau):

    couleur_direction1 = [plateau[(i-k),j]for k in range(1,2*n) i

    index_direction1 = [[i-k,j]for k in range(1,2*n) if (i-k) >=



    couleur_direction2 = [plateau[i-k,j+k] for k in range(1,2*n)

    index_direction2 = [[i-k,j+k] for k in range(1,2*n) if (i-k)
...
    index_direction8 = [[i-k,j-k] for k in range(1,2*n) if (i-k)

    return [couleur_direction1,couleur_direction2,couleur_directi
```

Gain important en cas d'évolution du programme

```
TAILLE_PLATEAU = 8
DIRECTIONS = [(di, dj) for di in (-1,0,1)
                       for dj in (-1,0,1) if (di, dj) != (0,0)]

def est_valide(i, j):
    """Dit si la case de coordonnées (i, j) existe"""
    return 0 <= i < TAILLE_PLATEAU and 0 <= j < TAILLE_PLATEAU

def directions(i, j, p):
    dirs = []; ind = []
    for di, dj in DIRECTIONS:
        c = [(i + k*di, j + k*dj) for k in range(1, 2*n)]
        ind.append([[i, j] for i,j in  c if est_valide(i, j)])
        dirs.append([p[i, j] for i,j in c if est_valide(i, j)])
    return (dirs, ind)
```

Exercice

Récrire le code de la fonction `matriceinitiale`

```
def matriceinitiale():
    '''Création de la matrice d'incidence initiale'''
    T=np.zeros((81,81))
    for N in range(81):
        [i,j]=inversef(N)
        if i==1: #S'il est sur la première ligne
            if j==1: #S'il est sur la première colonne
                T[N,f([i,j+2])]=1 #Lien avec son voisin de
                T[N,f([i+2,j])]=1 #Lien avec son voisin de
            elif j==17: #S'il est sur la dernière colonne
...
```

Lignes directrices

1. Introduction
2. Génie logiciel
3. Longueur des lignes
4. Identificateurs
5. Les nombres magiques
6. Conditionnelles
7. Les commentaires
8. Le dire une seule fois
9. Les idiomes de Python
10. Conclusion
11. Copyleft

Numéroter à partir de 0.

- On compte à partir de 0.
- Les intervalles sont semi-ouverts
  - `range(a,b)` est l'intervalle $[\![a, b[\![$.
  - Si `f(t, i, j)` travaille sur une portion du tableau `t`, c'est (sans doute) celle d'indices dans $[\![i, j[\![$.

Why numbering should start at zero — **EWD831.html**

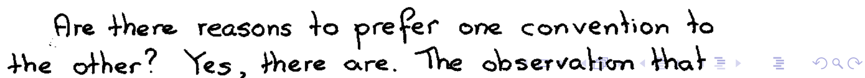
To denote the subsequence of natural numbers 2, 3, ..., 12 without the pernicious three dots, four conventions are open to us:

a) $2 \leq i < 13$

b) $1 < i \leq 12$

c) $2 \leq i \leq 12$

d) $1 < i < 13$


Are there reasons to prefer one convention to the other? Yes, there are. The observation that

Utiliser **for** quand on connaît le nombre d'itérations à l'avance.
Comparer les programmes suivants

```
i = 0
while i < n:
    print('bonjour')
    i += 1
```

```
i = 0
while i <= n:
    i += 1
    print('bonjour')
```

```
i = 0
while i <= n:
    print('bonjour')
    i += 1
```

```
for i in range(n):
    print('bonjour')
```

Itération (sur les éléments d'une structure de donnée)

Quand a besoin d'itérer sur les éléments d'une structure de données :

- utiliser `for`
- itérer sur les éléments et non sur un index

Exemple : afficher les éléments d'un tableau.

```
for x in t:
    print(x)
```

```
for i in range(len(t)):
    print(t[i])
```

Itération (suite)

- **`for`** reste intéressant même si on arrête l'itération prématurément.
- Exemple : écrire une fonction `contient_zero(t)` renvoyant `True` si le tableau `t` contient au moins une fois la valeur `0`, et `False` sinon.

Itération (suite)

```python
def contient_zero(t):
    i = 0
    b = False # zéro trouvé ?
    while not b and i<len(t):
        if t[i] == 0:
            b = True
        i += 1
    return b
```

```python
def contient_zero(t):
    i = 0
    while i<len(t) and t[i]!=0:
        i += 1
    return i < len(t)
```

```python
def contient_zero(t):
    for x in t:
        if x == 0:
            return True
    return False
```

```python
def contient_zero(t):
    for i in range(len(t)):
        if t[i] == 0:
            return True
    return False
```

Construction de tableau

Exemple : construire un tableau des n premières valeurs de la suite définie par $u_0 = a$ et $\forall n \in \mathbb{N} \quad u_{n+1} = f(u_n)$.

- Version non idiomatique :

```
t = [None] * n
t = [a]
for i in range(n-1):
    t[i+1] = f(t[i])
```

- Version idiomatique :

```
t = [a]
for _ in range(n-1):
    t.append(f(t[-1]))
```

Construction de tableau (suite)

Exemple : construction du triangle de Pascal.

```
t = [[1]] # tableau réduit à la ligne «0 parmi 0».
for i in range(1,n):
    li = i + 1 # longueur de la nouvelle ligne
    r = [1] # sa première case
    for j in range(1, li - 1):
        r.append(t[-1][j-1] + t[-1][j])
    r.append(1) # i parmi i
    t.append(r)
```

Notation en compréhension

En mathématique : $\{ f(x) \mid x \in u, P(x) \}$
En Python :

```
t = [f(x) for x in u if P(x)]
```

Plutôt que

```
t = []
for x in u:
    if P(x):
        t.append(f(x))
```

Exemple : construction de la matrice des $\left(\frac{1}{i+j+1}\right)_{(i,j)\in[\![0,n[\![^2}$.

```
h = [[1/(i+j+1) for j in range(n)] for i in range(n)]
```

Lignes directrices

1 Introduction
2 Génie logiciel
3 Longueur des lignes
4 Identificateurs
5 Les nombres magiques
6 Conditionnelles
7 Les commentaires
8 Le dire une seule fois
9 Les idiomes de Python
10 Conclusion
11 Copyleft

Conclusion

Boileau, 1674

Hâtez-vous lentement, et, sans perdre courage,
Vingt fois sur le métier remettez votre ouvrage
Polissez-le sans cesse et le repolissez ;
Ajoutez quelquefois, et souvent effacez.

Pour prolonger

- Kernighan et Pike, *The practice of programming*, 1999 (*La programmation en pratique*).
- PEP 8 – Style Guide for Python Code.
- PEP 20 – The Zen of Python (`import this`).
- Frederick Brooks, *The mythical man-month*, 1975 et 1995 (*Le mythe du mois-homme*).
- Dijkstra, *Why numbering should start at zero*, 1982.

Lignes directrices

1. Introduction
2. Génie logiciel
3. Longueur des lignes
4. Identificateurs
5. Les nombres magiques
6. Conditionnelles
7. Les commentaires
8. Le dire une seule fois
9. Les idiomes de Python
10. Conclusion
11. Copyleft

Copyleft

Cette oeuvre est libre, vous pouvez la copier, la diffuser et la modifier selon les termes de la Licence Art Libre http://artlibre.org, version 1.3 ou ultérieure.