

# Spécification, test

DIU Enseigner l'Informatique au Lycée

E. Jahier, L. Rieg, C. Parent-Vigouroux, B. Wack

UFR IM2AG, Université Grenoble Alpes

juillet 2020

Remerciements : Laure Gonnord et l'équipe du DIU Lyon 1

# Plan

Notion de correction d'un algorithme

- Spécification formelle

- Démarche de tests

En Python

- Mise en pratique

- Bonnes pratiques à encourager

# Plan

Notion de correction d'un algorithme

Spécification formelle

Démarche de tests

En Python

Mise en pratique

Bonnes pratiques à encourager

# Les besoins

- ▶ Quantité astronomique de code en circulation ou en développement  
(Google Chrome ou serveur World of Warcraft : 6 M lignes)  
(Windows 7 ou Microsoft Office 2013 : 40 M lignes)

<http://www.informationisbeautiful.net>

- ▶ Omniprésence dans des systèmes critiques : finance, transports, économie, santé...
- ▶ La moindre erreur peut coûter « cher » : on aimerait qu'un programme s'exécute correctement dans toutes les situations

# Les besoins

- ▶ Quantité astronomique de code en circulation ou en développement  
(Google Chrome ou serveur World of Warcraft : 6 M lignes)  
(Windows 7 ou Microsoft Office 2013 : 40 M lignes)

<http://www.informationisbeautiful.net>

- ▶ Omniprésence dans des systèmes critiques : finance, transports, économie, santé...
- ▶ La moindre erreur peut coûter « cher » : on aimerait qu'un programme s'exécute correctement dans toutes les situations

Mais correct selon quels critères ? Quelles situations à considérer ?

# Les besoins

- ▶ Quantité astronomique de code en circulation ou en développement  
(Google Chrome ou serveur World of Warcraft : 6 M lignes)  
(Windows 7 ou Microsoft Office 2013 : 40 M lignes)

<http://www.informationisbeautiful.net>

- ▶ Omniprésence dans des systèmes critiques : finance, transports, économie, santé...
- ▶ La moindre erreur peut coûter « cher » : on aimerait qu'un programme s'exécute correctement dans toutes les situations

Mais correct selon quels critères ? Quelles situations à considérer ?

- ▶ Spécification
  - ▶ des **données acceptables**
  - ▶ du **résultat attendu** (généralement en fonction des données)
- ▶ exprimée dans un langage **formel**, généralement une propriété logique des données et du résultat.
- ▶ ne décrit **pas comment** fonctionne le programme.

# Les propriétés recherchées

## Terminaison

L'exécution de l'algorithme produit-elle un résultat en temps **fini** **quelles**  
**que soient** les données fournies ?

# Les propriétés recherchées

## Terminaison

L'exécution de l'algorithme produit-elle un résultat en temps **fini** **quelles que soient** les données fournies ?

## Correction partielle

**Lorsque l'algorithme s'arrête**, le résultat calculé est-il **la solution cherchée** quelles que soient les données fournies ?



# Les propriétés recherchées

## Terminaison

L'exécution de l'algorithme produit-elle un résultat en temps **fini** **quelles que soient** les données fournies ?

## Correction partielle

**Lorsque l'algorithme s'arrête**, le résultat calculé est-il **la solution cherchée** quelles que soient les données fournies ?

Terminaison + correction partielle = correction totale

Quelles que soient les données fournies, l'algorithme s'arrête et donne une réponse correcte.

# Les propriétés recherchées

## Terminaison

L'exécution de l'algorithme produit-elle un résultat en temps **fini** **quelles que soient** les données fournies ?

## Correction partielle

**Lorsque l'algorithme s'arrête**, le résultat calculé est-il **la solution cherchée** quelles que soient les données fournies ?

Terminaison + correction partielle = correction totale

Quelles que soient les données fournies, l'algorithme s'arrête et donne une réponse correcte.

Pour certains problèmes il n'existe **que** des algorithmes **partiellement** corrects !

# Une première écriture formelle

Soit un problème instancié par une donnée  $D$  et dont la réponse est fournie par un résultat  $R$ .

# Une première écriture formelle

Soit un problème instancié par une donnée  $D$  et dont la réponse est fournie par un résultat  $R$ .

Une spécification peut être donnée sous la forme de :

- ▶ une propriété  $P(D)$  de la donnée (*précondition*);
- ▶ une propriété  $Q(D, R)$  de la donnée et du résultat (*postcondition*).

# Une première écriture formelle

Soit un problème instancié par une donnée  $D$  et dont la réponse est fournie par un résultat  $R$ .

Une spécification peut être donnée sous la forme de :

- ▶ une propriété  $P(D)$  de la donnée (*précondition*);
- ▶ une propriété  $Q(D, R)$  de la donnée et du résultat (*postcondition*).

Un programme **satisfait** cette spécification si :

Pour toute donnée  $D$  qui vérifie la propriété  $P$ ,  
l'exécution du programme donne un résultat  $R$  qui vérifie  $Q$ .

Le programme est alors dit *correct par rapport* à cette spécification.

# Division euclidienne

## Division par soustractions

$\text{DIV}(a, b)$

**Données** : Deux entiers  $a$  et  $b$

**Résultat** : Le quotient  $q$  et le reste  $r$  de la division euclidienne de  $a$  par  $b$

$r := a$

$q := 0$

**while**  $r \geq b$

┌  $r := r - b$

└  $q := q + 1$

**return**  $q, r$

# Division euclidienne

## Division par soustractions

$\text{DIV}(a, b)$

**Données** : Deux entiers  $a$  et  $b$

**Résultat** : Le quotient  $q$  et le reste  $r$  de la division euclidienne de  $a$  par  $b$

$r := a$

$q := 0$

**while**  $r \geq b$

┌  $r := r - b$

└  $q := q + 1$

**return**  $q, r$

## Données acceptables

# Division euclidienne

## Division par soustractions

DIV( $a, b$ )

**Données** : Deux entiers  $a$  et  $b$

**Résultat** : Le quotient  $q$  et le reste  $r$  de la division euclidienne de  $a$  par  $b$

$r := a$

$q := 0$

**while**  $r \geq b$

┌  $r := r - b$

└  $q := q + 1$

**return**  $q, r$

## Données acceptables

- ▶  $b \neq 0$  sinon le problème n'a pas de sens (et la boucle non plus)



# Division euclidienne

## Division par soustractions

DIV( $a, b$ )

**Données** : Deux entiers  $a$  et  $b$

**Résultat** : Le quotient  $q$  et le reste  $r$  de la division euclidienne de  $a$  par  $b$

$r := a$

$q := 0$

**while**  $r \geq b$

┌  $r := r - b$

└  $q := q + 1$

**return**  $q, r$

## Données acceptables

- ▶  $b \neq 0$  sinon le problème n'a pas de sens (et la boucle non plus)
- ▶  $b > 0$  (?)

# Division euclidienne

## Division par soustractions

DIV( $a, b$ )

**Données** : Deux entiers  $a$  et  $b$

**Résultat** : Le quotient  $q$  et le reste  $r$  de la division euclidienne de  $a$  par  $b$

$r := a$

$q := 0$

**while**  $r \geq b$

┌  $r := r - b$

└  $q := q + 1$

**return**  $q, r$

## Données acceptables

- ▶  $b \neq 0$  sinon le problème n'a pas de sens (et la boucle non plus)
- ▶  $b > 0$  (?)
- ▶  $a > 0$  (?)

# Division euclidienne

## Division par soustractions

$\text{DIV}(a, b)$

**Données** : Deux entiers  $a$  et  $b$

**Résultat** : Le quotient  $q$  et le reste  $r$  de la division euclidienne de  $a$  par  $b$

$r := a$

$q := 0$

**while**  $r \geq b$

┌  $r := r - b$

└  $q := q + 1$

**return**  $q, r$

Résultat attendu

# Division euclidienne

## Division par soustractions

DIV( $a, b$ )

**Données** : Deux entiers  $a$  et  $b$

**Résultat** : Le quotient  $q$  et le reste  $r$  de la division euclidienne de  $a$  par  $b$

$r := a$

$q := 0$

**while**  $r \geq b$

┌  $r := r - b$

└  $q := q + 1$

**return**  $q, r$

## Résultat attendu

►  $a = q \times b + r$

# Division euclidienne

## Division par soustractions

DIV( $a, b$ )

**Données** : Deux entiers  $a$  et  $b$

**Résultat** : Le quotient  $q$  et le reste  $r$  de la division euclidienne de  $a$  par  $b$

$r := a$

$q := 0$

**while**  $r \geq b$

┌  $r := r - b$

└  $q := q + 1$

**return**  $q, r$

## Résultat attendu

- ▶  $a = q \times b + r$
- ▶  $0 \leq r < b$

# Division euclidienne

## Division par soustractions

DIV( $a, b$ )

**Données** : Deux entiers  $a$  et  $b$

**Résultat** : Le quotient  $q$  et le reste  $r$  de la division euclidienne de  $a$  par  $b$

$r := a$

$q := 0$

**while**  $r \geq b$

┌  $r := r - b$

└  $q := q + 1$

**return**  $q, r$

## Résultat attendu

- ▶  $a = q \times b + r$
- ▶  $0 \leq r < b$
- ▶  $a$  et  $b$  inchangés

# Test ou preuve ?

*"Testing shows the presence, not the absence of bugs"*

E. W. Dijkstra

# Test ou preuve ?

*“Testing shows the presence, not the absence of bugs”*

E. W. Dijkstra

La preuve d'algorithme :

- ▶ fournit une **garantie** incontestable sur le fond de l'algorithme
- ▶ mais n'élimine pas (complètement) les erreurs de programmation
- ▶ et devient vite difficile à grande échelle



# Test ou preuve ?

*“Testing shows the presence, not the absence of bugs”*

E. W. Dijkstra

La preuve d'algorithme :

- ▶ fournit une **garantie** incontestable sur le fond de l'algorithme
- ▶ mais n'élimine pas (complètement) les erreurs de programmation
- ▶ et devient vite difficile à grande échelle

Le test :

- ▶ valide une **implantation** plutôt qu'un algorithme
- ▶ permet des vérifications rapides
- ▶ peut être utilisé en cours de développement
- ▶ fait apparaître les bugs et permet de les corriger

# Plan

## Notion de correction d'un algorithme

Spécification formelle

Démarche de tests

## En Python

Mise en pratique

Bonnes pratiques à encourager

# Validation expérimentale d'un programme

## Valider un programme

= vérifier que ce qu'il fait **correspond à sa spécification.**

## Pour quoi faire ?

- ▶ « **Trouver des erreurs** » plutôt que « vérifier que ça marche »
- ▶ Attention, en général un ou plusieurs test(s) positif(s) ne constituent pas une **preuve** de validité
- ▶ On ne cherche pas spécialement à vérifier que le programme « ne plante pas » (robustesse)

Dans notre cas, on se contentera de tests (mais d'autres méthodes existent : model-checking, analyse statique...)

# Bonnes pratiques de test

1. identifier les données du programme, et, parmi elles, le domaine de valeurs sur lequel on souhaite tester le programme

# Bonnes pratiques de test

1. identifier les données du programme, et, parmi elles, le domaine de valeurs sur lequel on souhaite tester le programme
2. écrire un **jeu d'essai** (**plusieurs** tests) tel que :
  - ▶ chaque test soit **pertinent**
  - ▶ on **couvre** bien le domaine recherché
  - ▶ si on connaît le code du programme (boîte blanche), tous les chemins d'exécution soient testés
  - ▶ éviter de tester des valeurs hors du domaine, notamment ne remplissant pas les préconditions (robustesse encore)

# Bonnes pratiques de test

1. identifier les données du programme, et, parmi elles, le domaine de valeurs sur lequel on souhaite tester le programme
2. écrire un **jeu d'essai** (**plusieurs** tests) tel que :
  - ▶ chaque test soit **pertinent**
  - ▶ on **couvre** bien le domaine recherché
  - ▶ si on connaît le code du programme (boîte blanche), tous les chemins d'exécution soient testés
  - ▶ éviter de tester des valeurs hors du domaine, notamment ne remplissant pas les préconditions (robustesse encore)
3. éventuellement, **générer le jeu d'essai** automatiquement

# Bonnes pratiques de test

1. identifier les données du programme, et, parmi elles, le domaine de valeurs sur lequel on souhaite tester le programme
2. écrire un **jeu d'essai** (**plusieurs** tests) tel que :
  - ▶ chaque test soit **pertinent**
  - ▶ on **couvre** bien le domaine recherché
  - ▶ si on connaît le code du programme (boîte blanche), tous les chemins d'exécution soient testés
  - ▶ éviter de tester des valeurs hors du domaine, notamment ne remplissant pas les préconditions (robustesse encore)
3. éventuellement, **générer le jeu d'essai** automatiquement
4. pour chaque test du jeu d'essai :
  - ▶ exécuter et vérifier à la main la validité du résultat
  - ▶ ou mieux, écrire un **oracle** : un programme vérifiant que le résultat est conforme

# Plan

Notion de correction d'un algorithme

Spécification formelle

Démarche de tests

En Python

Mise en pratique

Bonnes pratiques à encourager



# Préconditions : assert est votre ami - exemple 1

```
numbers = [1.5, 2.3, 0.7, -0.001, 4.4]
total = 0.0
for n in numbers:
    assert n > 0.0, 'Data should only contain positive values'
    total += n
print('total is:', total)
```

```
Traceback (most recent call last):
  File "demo_assert.py", line 4, in <module>
    assert n > 0.0, 'Data should only contain positive values'
AssertionError: Data should only contain positive values
```

Permet aussi de prévenir : division par 0, etc...

## Préconditions : assert est votre ami - exemple 2

Ma fonction est prévue pour des listes de 2 entiers :

```
def print_coordinates(pt):  
    """ prints out the coordinates in a  
    fashionable maneer """  
    assert type(pt) == list, 'input should be a list'  
    assert len(pt) == 2, 'print_coordinates: data should be of len 2'  
    assert type(pt[0]) == int and type(pt[1]) == int, 'data should be of ty  
    print("( x=", pt[0],", y=", pt[1], ")")
```

- ▶ Surtout si vous fournissez une librairie à vos élèves.

# Test

## Test unitaire

En programmation, le test unitaire est une procédure permettant de vérifier le bon fonctionnement d'une partie précise d'un logiciel ou d'un programme (appelée « unité » ou « module »).

## Zoom sur doctest

doctest est un module python permettant de les réaliser simplement :

- ▶ facile à mettre en oeuvre
- ▶ intégré comme des exemples à la documentation
- ▶ possibilité de lancer automatiquement les tests
- ▶ messages d'erreurs lisibles

Inconvénients :

- ▶ finit par encombrer la documentation
- ▶ très sensible à l'écriture du résultat attendu

# Un exemple simple

```
def renverser(L):
    """Renvoie une nouvelle liste contenant les memes elements que L
    en ordre inverse.

    >>> renverser([])
    []
    >>> renverser([1,2,3,4])
    [4, 3, 2, 1]
    >>> renverser(renverser([x*x for x in range (100)])) == [x*x for x in r
    True
```

Les autres iterables ne sont pas acceptes :

```
>>> renverser("abc")
Traceback (most recent call last):
...
AssertionError: L'argument doit etre une liste
"""
```

```
assert type(L) == list, "L'argument doit etre une liste"
M = []
for i in range(len(L)):
    M.append(L[len(L)-1-i])

assert len(L)==len(M), "Oh oh, on a perdu un element en route..."
return M
```

# Utilisation automatique

Ajouter à la fin du fichier :

```
if __name__ == "__main__":  
    import doctest  
    doctest.testmod()
```

Puis exécuter le fichier comme un script (ou dans un terminal) :

- ▶ Si rien ne se passe c'est que tout va bien !
- ▶ Sinon :

```
File "__main__", line 5, in __main__.renverser  
Failed example:  
    renverser([1,2,3,4])  
Expected:  
    [4, 3, 2, 1]  
Got:  
    [2, 1, 4, 3]
```

# Utilisation automatique

Ajouter à la fin du fichier :

```
if __name__ == "__main__":  
    import doctest  
    doctest.testmod()
```

Puis exécuter le fichier comme un script (ou dans un terminal) :

- ▶ Si rien ne se passe c'est que tout va bien !
- ▶ Sinon :

```
File "__main__", line 5, in __main__.renverser  
Failed example:  
    renverser([1,2,3,4])  
Expected:  
    [4, 3, 2, 1]  
Got:  
    [2, 1, 4, 3]
```

- ▶ voir aussi unittest ou pytest

# Plan

Notion de correction d'un algorithme

Spécification formelle

Démarche de tests

En Python

Mise en pratique

Bonnes pratiques à encourager

# Le niveau 0 : bien nommer ses identifiants

- ▶ Les fonctions avec des *underscore* : `ma_jolie_fonction` et préférer les verbes
- ▶ Les classes (plus tard) avec une majuscule
- ▶ Les variables avec des noms signifiants
- ▶ Les constantes tout en majuscules
- ▶ Choisir français ou anglais et ne pas mélanger
- ▶ Jamais « 1 » pour les listes, au pire utiliser L (éviter aussi 0 et I)



## Niveau 1 : documenter

```
def print_coordinates(pt):  
    """Prints out the coordinates in a  
    fashionable maneer."""
```

Commenter aussi le code :

- ▶ sans le paraphraser
- ▶ ni le contredire
- ▶ sur une ligne complète

Lire aussi <https://pep8.org>

## Niveau 2 : programmer défensivement

Traduction libre de <https://swcarpentry.github.io/python-novice-inflammation/08-defensive/index.html>

- ▶ Programmer **défensivement**, i.e., présupposer que des erreurs vont se produire, et écrire du code pour les détecter.
- ▶ Placer des **assertions** dans vos programmes pour vérifier que l'état mémoire reste correct, et pour aider les lecteurs à comprendre ce qui est censé se passer.
- ▶ Utiliser des **préconditions** pour vérifier que les données d'une fonction ne créeront pas de problème.
- ▶ Utiliser des **postconditions** pour vérifier que les résultats d'une fonction sont sans risque.
- ▶ Écrire des **tests** *avant* d'écrire du code pour préciser ce que le code est censé faire.