Présentation du code, preuve de programme, tests

Ambre Le Berre

2025/2026

MP

Agenda

1. Présentation du code
2. Programmation défensive
3. Preuves de programme

Présentation du code

Nommer ses variables

- Il est important de donner des noms explicites à ses variables : sauf dans les cas vraiment “standard” (`L`, `i`, `j`, ...), on utilise un ou deux mots. C’est d’autant plus vrai pour les fonctions.
-
-
-
-

Nommer ses variables

- Il est important de donner des noms explicites à ses variables : sauf dans les cas vraiment “standard” (`L`, `i`, `j`, ...), on utilise un ou deux mots. C’est d’autant plus vrai pour les fonctions.
- Les noms des variables et fonctions sont en `snake_case` en Python :
- tout en minuscule (sauf les acronymes)
- un underscore _ entre les mots
- soit tout en français sans accent, soit tout en anglais, pas de mélange

Opérateurs de calculs

On peut faire des calculs entre des valeurs ou des variables.

```
a = 3 + je_suis_une_variable
b = 13039 % 12
```

Opérateurs de calculs

On peut faire des calculs entre des valeurs ou des variables.

```
a = 3 + je_suis_une_variable
b = 13039 % 12
```

On met toujours des espaces avant et après un opérateurs binaire. Pour un opérateurs unaire on ne met pas d'espace.

```
calcul = 3 * 5
nombre_negatif = -7
```

Les commentaires

Il est important de commenter votre code dès qu'il n'est pas évident !

- commentaire inutile

```
a = 3 # assigne 3 à la variable a
```

- commentaire utile :

```
cpt = 0 # la variable cpt compte le nombre de bouteilles dans la mer
```

Documenter ses fonctions

On peut (et on doit) ajouter une chaine de documentation à ses fonctions.

```
def salue(personne):
  """
  Prend en argument une personne (chaine de caractère) et la salue.
  Renvoie un booléen, `true` si la salutation a été retournée.
  """
  # code pour saluer une personne
```

Documenter ses fonctions

On peut (et on doit) ajouter une chaine de documentation à ses fonctions.

```
def salue(personne):
  """
  Prend en argument une personne (chaine de caractère) et la salue.
  Renvoie un booléen, `true` si la salutation a été retournée.
  """
  # code pour saluer une personne
```

Ensuite, on peut accéder à cette chaine avec

```
help(salue)
```

Indentation

L'indentation de votre programme doit être parfaitement claire.

À l'écrit, on trace des lignes !

Programmation défensive

Pré-conditions, post-conditions, invariant

Comment faire pour être certain qu'une boucle, ou qu'un bloc d'instruction, est correct ?

Pré-conditions, post-conditions, invariant

Comment faire pour être certain qu'une boucle, ou qu'un bloc d'instruction, est correct ?

On écrit des conditions de la forme suivante :

- une pré-condition est respectée avant l'entrée dans la boucle.
- pendant toute l'execution de la boucle, un invariant reste vrai.
- si c'est deux propriétés sont respectées, on en déduit une post-condition qui sera vrai à la sortie de la bouvle.

Exemple

```
def somme(L):
    """
    Cette fonction prend en argument une liste d'entier, et calcule et
renvoie sa somme.
    """
    S = 0
    # pré-condition
    for i in range(len(L)):
        # invariant
        S += L[i]
    # post-condition
    return S
```

Exemple 2

```
def insere(L_triee, x):
    """ Cette fonction prend en argument une liste triée d'entiers, et
un élément x, et l'insère en place dans la liste. """
    ...

def tri_insertion(L):
    """Cette fonction prend en argument une liste d'entier, et renvoie
une nouvelle liste triée avec l'algorithme du tri par insertion."""
    L_triee = []
    # pré-condition
    for i in range(len(L)):
        # invariant
        insere(L_triee, L[i])
    # post-condition
    return L_triee
```

Assertions

Les assertions permettent de tester les conditions et invariants pendant l'execution du code.

```
# pre-condition : la liste n'est pas vide
assert len(L) > 0
```

Si la condition est fausse, une erreur est levée.

Exemple

```
def somme(L):
    """
    Cette fonction prend en argument une liste d'entier, et calcule et
renvoie sa somme.
    """
    S = 0
    #pre-condition : L est une liste d'entiers
    assert type(L) == list
    for i in range(len(L)):
        # invariant : L[i] est un entier et S contient la somme de
L[0:i+1].
        assert type(L[i]) == int
        S += L[i]
    # post-condition : S contient la somme de L.
    return S
```

Tests

On appelle jeu de test l'ensemble des tests d'un programme. Comment écrire des tests ?

Tests

On appelle jeu de test l'ensemble des tests d'un programme. Comment écrire des tests ?

- Partitionner l'espace d'entrée en catégories, et faire un test par catégorie.
-

Tests

On appelle jeu de test l'ensemble des tests d'un programme. Comment écrire des tests ?

- Partitionner l'espace d'entrée en catégories, et faire un test par catégorie.
- Tester les cas limites.

Preuves de programme

Pourquoi les preuves de programmes ?

Lorsqu'on ne veut pas dépendre d'un test, on peut prouver mathématiquement que notre programme est juste.

Définitions

Terminaison : On dit qu'un programme **termine** si son execution ne boucle **jamais** à l'infini, quelle que soit l'entrée.

Définitions

Terminaison : On dit qu'un programme **termine** si son execution ne boucle **jamais** à l'infini, quelle que soit l'entrée.

```
def exp(x, n):
    resultat = 1
    while n ≠ 0:
        resultat *= x
        n -= 1
    return resultat
```

Prouver la terminaison

```
def exp(x, n):
    """ Prend en entrée un entier x et un entier positif n et renvoie
x^n. """
    resultat = 1
    assert n ⩾ 0
    while n ≠ 0:
        resultat *= x
        n -= 1
    return resultat
```

Prouver la terminaison

```
def exp(x, n):
    """ Prend en entrée un entier x et un entier positif n et renvoie
x^n. """
    resultat = 1
    assert n ⩾ 0
    while n ≠ 0:
        resultat *= x
        n -= 1
    return resultat
```

variant de boucle : quantité positive et strictement décroissante entre chaque tour de boucle.