Algorithmique

— procédures et fonctions —

Jacques TISSEAU

LISYC EA 3883 UBO-ENIB-ENSIETA
Centre Européen de Réalité Virtuelle
Ecole Nationale d'Ingénieurs de Brest

Réutilisabilité des algorithmes

Problème

Comment réutiliser un algorithme existant sans avoir à le réécrire ?

Réutilisabilité des algorithmes

Problème

Comment réutiliser un algorithme existant sans avoir à le réécrire ?

```
>>> n = 3
>>> f = 1
>>> for i in range(1,n+1) :
...     f = f*i
...
>>> f
6
```

Réutilisabilité des algorithmes

Problème

Comment réutiliser un algorithme existant sans avoir à le réécrire ?

```
>>> n = 3
>>> f = 1
>>> for i in range(1,n+1) :
...     f = f*i
...
>>> f
6
```

```
>>> n = 5
>>> f = 1
>>> for i in range(1,n+1) :
...     f = f*i
...
>>> f
120
```

Réutilisabilité des algorithmes

Problème

Comment réutiliser un algorithme existant sans avoir à le réécrire ?

```
>>> n = 3
>>> f = 1
>>> for i in range(1,n+1) :
...     f = f*i
...
>>> f
6
```

```
>>> n = 5
>>> f = 1
>>> for i in range(1,n+1) :
...     f = f*i
...
>>> f
120
```

Elément de réponse

Encapsuler le code dans des fonctions ou des procédures.

Réutilisabilité des algorithmes

Problème

Comment réutiliser un algorithme existant sans avoir à le réécrire ?

```
>>> n = 3
>>> f = 1
>>> for i in range(1,n+1) :
...     f = f*i
...
>>> f
6
```

```
>>> n = 5
>>> f = 1
>>> for i in range(1,n+1) :
...     f = f*i
...
>>> f
120
```

Elément de réponse

Encapsuler le code dans des fonctions ou des procédures.

```
>>> factorielle(3)
6
```

Réutilisabilité des algorithmes

Problème

Comment réutiliser un algorithme existant sans avoir à le réécrire ?

```
>>> n = 3
>>> f = 1
>>> for i in range(1,n+1) :
...     f = f*i
...
>>> f
6
```

```
>>> n = 5
>>> f = 1
>>> for i in range(1,n+1) :
...     f = f*i
...
>>> f
120
```

Elément de réponse

Encapsuler le code dans des fonctions ou des procédures.

```
>>> factorielle(3)
6
```

```
>>> factorielle(5)
120
```

Structuration des algorithmes

Problème

Comment structurer un algorithme pour le rendre plus compréhensible ?

Structuration des algorithmes

Problème

Comment structurer un algorithme pour le rendre plus compréhensible ?

```
ieee_code = []
k_exponent = 8
k_significand = 23
k_ieee = 32
bias = code(127,2,k_exponent)
x_int = int(abs(x))
x_frac = abs(x) - x_int
expo_2 = 0
for i in range(k_ieee) : append(ieee_code,0)

# calcul du signe
sign = int(x < 0)

# calcul de la mantisse
i = 0
significand = []
while (x_int != 0) and (i < k_significand) :
    insert(significand,0,x_int%2)
    x_int = x_int/2
    i = i + 1
if len(significand) > 0 and significand[0] == 1 :
    del significand[0]
    expo_2 = len(significand)
i = len(significand)
while (x_frac != 0) and (i < k_significand) :
    x_frac = x_frac * 2
    x_int = int(x_frac)
    x_frac = x_frac - x_int
    if (x_int == 0) and (i == 0) :
        expo_2 = expo_2 - 1
else :
        append(significand,x_int)
        i = i + 1
```

Structuration des algorithmes

Problème

Comment structurer un algorithme pour le rendre plus compréhensible ?

```
ieee_code = []
k_exponent = 8
k_significand = 23
k_ieee = 32
bias = code(127,2,k_exponent)
x_int = int(abs(x))
x_frac = abs(x) - x_int
expo_2 = 0
for i in range(k_ieee) : append(ieee_code,0)

# calcul du signe
sign = int(x < 0)

# calcul de la mantisse
i = 0
significand = []
while (x_int != 0) and (i < k_significand) :
    insert(significand,0,x_int%2)
    x_int = x_int/2
    i = i + 1
if len(significand) > 0 and significand[0] == 1 :
    del significand[0]
    expo_2 = len(significand)
i = len(significand)
while (x_frac != 0) and (i < k_significand) :
    x_frac = x_frac * 2
    x_int = int(x_frac)
    x_frac = x_frac - x_int
    if (x_int == 0) and (i == 0) :
        expo_2 = expo_2 - 1
else :
        append(significand,x_int)
        i = i + 1

et quelques 20 lignes plus loin...

    ieee_code[0] = sign
    ieee_code[1 :9] = exponent
    ieee_code[9 :32] = significand
```

Structuration des algorithmes

Elément de réponse

Utiliser des fonctions et des procédures.

Structuration des algorithmes

Elément de réponse

Utiliser des fonctions et des procédures.

```
# calcul du signe
sign = int(x < 0)

# calcul de la mantisse
significand, expo_2 = mantisse(x)

# calcul de l'exposant
exponent = exposant(expo_2,127)

# code IEEE 754
ieee_code[0] = sign
ieee_code[1 :9] = exponent
ieee_code[9 :32] = significand
```

Diviser pour régner

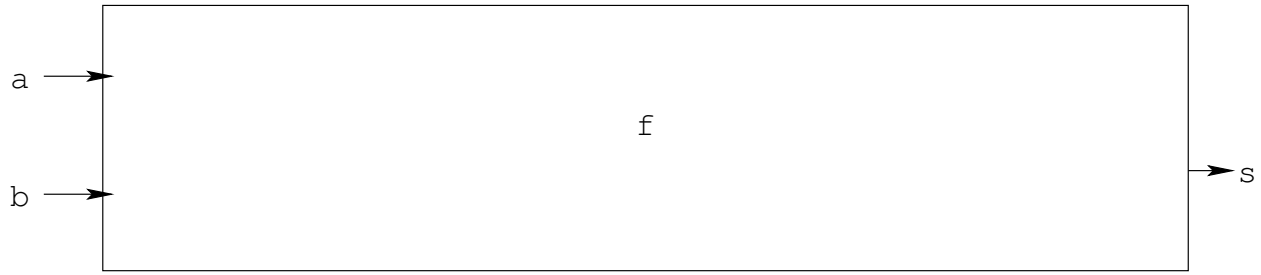
Structuration

Les fonctions et les procédures permettent de décomposer un programme complexe en une série de sous-programmes plus simples, lesquels peuvent à leur tour être décomposés eux-mêmes en fragments plus petits, et ainsi de suite.

Diviser pour régner

Structuration

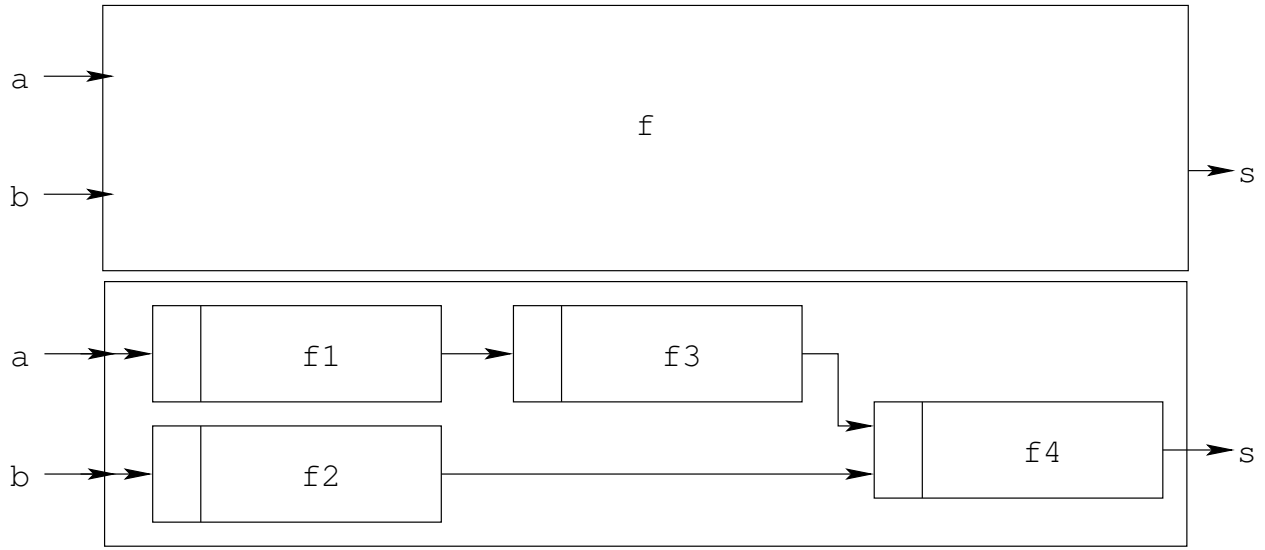
Les fonctions et les procédures permettent de décomposer un programme complexe en une série de sous-programmes plus simples, lesquels peuvent à leur tour être décomposés eux-mêmes en fragments plus petits, et ainsi de suite.



Diviser pour régner

Structuration

Les fonctions et les procédures permettent de décomposer un programme complexe en une série de sous-programmes plus simples, lesquels peuvent à leur tour être décomposés eux-mêmes en fragments plus petits, et ainsi de suite.



Fonctions

Fonctions

Une fonction est une suite ordonnée d'instructions qui *retourne* une valeur (bloc d'instructions nommé et paramétré).

Fonctions

Fonctions

Une fonction est une suite ordonnée d'instructions qui *retourne* une valeur (bloc d'instructions nommé et paramétré).

Fonction ≡ expression

Une fonction joue le rôle d'une expression.

Fonctions

Fonctions

Une fonction est une suite ordonnée d'instructions qui *retourne* une valeur (bloc d'instructions nommé et paramétré).

Fonction ≡ expression

Une fonction joue le rôle d'une expression.
Elle enrichit le jeu des expressions possibles.

Fonctions

Fonctions

Une fonction est une suite ordonnée d'instructions qui *retourne* une valeur (bloc d'instructions nommé et paramétré).

Fonction ≡ expression

Une fonction joue le rôle d'une expression.
Elle enrichit le jeu des expressions possibles.

Exemple

`y = sin(x)` renvoie la valeur du sinus de x

nom : `sin`

paramètres : `x :float` → `sin(x) :float`

Procédures

Procédures

Une procédure est une suite ordonnée d'instructions qui *ne retourne pas* de valeur (bloc d'instructions nommé et paramétré).

Procédures

Procédures

Une procédure est une suite ordonnée d'instructions qui *ne retourne pas* de valeur (bloc d'instructions nommé et paramétré).

Procédure ≡ instruction

Une procédure joue le rôle d'une instruction.

Procédures

Procédures

Une procédure est une suite ordonnée d'instructions qui *ne retourne pas* de valeur (bloc d'instructions nommé et paramétré).

Procédure ≡ instruction

Une procédure joue le rôle d'une instruction.
Elle enrichit le jeu des instructions existantes.

Procédures

Procédures

Une procédure est une suite ordonnée d'instructions qui *ne retourne pas* de valeur (bloc d'instructions nommé et paramétré).

Procédure ≡ instruction

Une procédure joue le rôle d'une instruction.
Elle enrichit le jeu des instructions existantes.

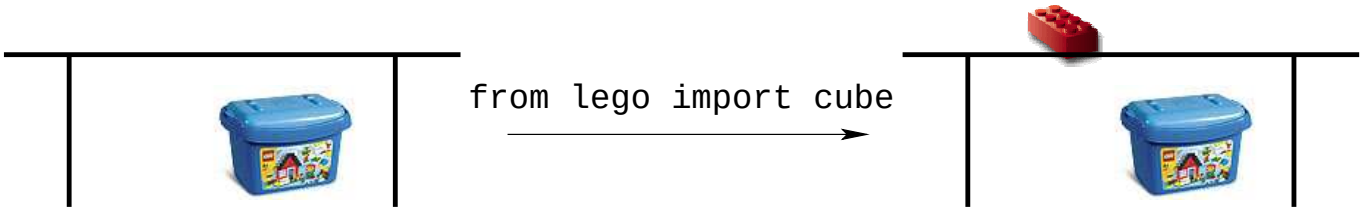
Exemple

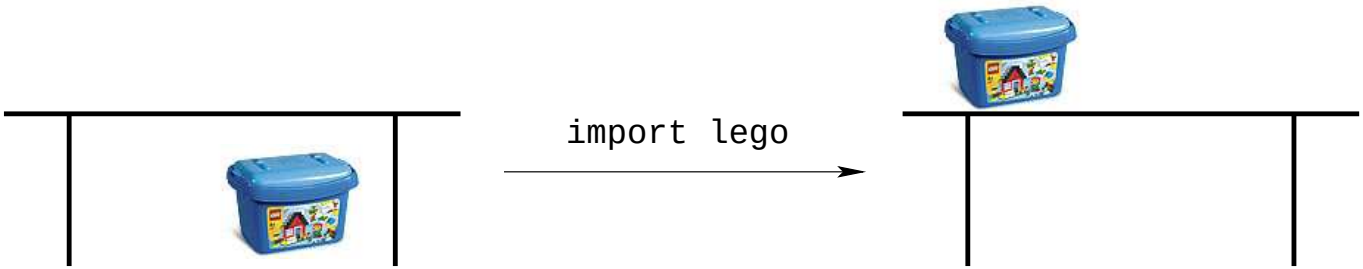
`print(x, y, z)` affiche les valeurs de x, y et z

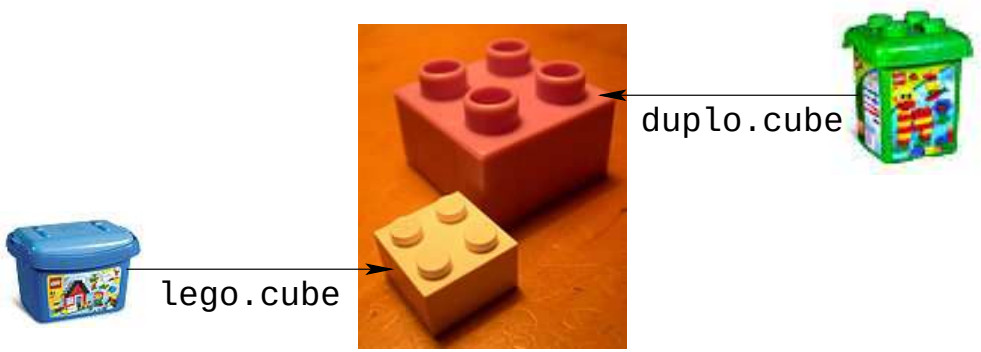
nom : `print`

paramètres : `x`, `y`, `z` $\rightarrow \square$

Modules







Définition d'une fonction

Les 6 étapes de définition

Nom : un identificateur suffisamment explicite.

Définition d'une fonction

Les 6 étapes de définition

Nom : un identificateur suffisamment explicite.

Paramètres : la liste des paramètres d'entrée-sortie de l'algorithme.

Définition d'une fonction

Les 6 étapes de définition

Nom : un identificateur suffisamment explicite.

Paramètres : la liste des paramètres d'entrée-sortie de l'algorithme.

Préconditions : une liste d'expressions booléennes qui précisent les conditions d'application de l'algorithme.

Définition d'une fonction

Les 6 étapes de définition

Nom : un identificateur suffisamment explicite.

Paramètres : la liste des paramètres d'entrée-sortie de l'algorithme.

Préconditions : une liste d'expressions booléennes qui précisent les conditions d'application de l'algorithme.

Description : une phrase qui dit ce que fait l'algorithme.

Définition d'une fonction

Les 6 étapes de définition

Nom : un identificateur suffisamment explicite.

Paramètres : la liste des paramètres d'entrée-sortie de l'algorithme.

Préconditions : une liste d'expressions booléennes qui précisent les conditions d'application de l'algorithme.

Description : une phrase qui dit ce que fait l'algorithme.

Appel : des exemples d'utilisation de l'algorithme avec les résultats attendus.

Définition d'une fonction

Les 6 étapes de définition

Nom : un identificateur suffisamment explicite.

Paramètres : la liste des paramètres d'entrée-sortie de l'algorithme.

Préconditions : une liste d'expressions booléennes qui précisent les conditions d'application de l'algorithme.

Description : une phrase qui dit ce que fait l'algorithme.

Appel : des exemples d'utilisation de l'algorithme avec les résultats attendus.

Code : la séquence d'instructions nécessaires à la résolution du problème.

Nom et paramètres d'entrée-sortie

1. *nom*

```
def factorielle() :
    return
```

1. *nom*

```
>>> factorielle()
>>>
```

Nom et paramètres d'entrée-sortie

1. *nom*

```
def factorielle() :
    return
```

2. *paramètres d'entrée-sortie*

```
def factorielle(n) :
    f = 1
    return f
```

1. *nom*

```
>>> factorielle()
>>>
```

2. *paramètres d'entrée-sortie*

```
>>> factorielle(5)
1
>>> factorielle(-5)
1
>>> factorielle('toto')
1
```

Jeu de tests

5 *jeu de tests*

```
def factorielle(n) :
    """
    f = n!
    >>> for i in range(8) :
    ...     print factorielle(i),
    1 1 2 6 24 120 720 5040
    """
    f = 1
    return f
```

5 *jeu de tests*

```
>>> for i in range(8) :
...     print factorielle(i),
...
1 1 1 1 1 1 1 1
```

Jeu de tests

5 *jeu de tests*

```
def factorielle(n) :
    """
    f = n!
    >>> for i in range(8) :
    ...     print factorielle(i),
    1 1 2 6 24 120 720 5040
    """
    f = 1
    return f
```

5 *jeu de tests*

```
>>> for i in range(8) :
...     print factorielle(i),
...
1 1 1 1 1 1 1 1
```

Remarque sur la validité de l'algorithme

A chaque étape de la spécification, le code de la fonction doit toujours être exécutable même s'il ne donne pas encore le bon résultat.

Jeu de tests

5 *jeu de tests*

```
def factorielle(n) :
    """
    f = n!
    >>> for i in range(8) :
    ...     print factorielle(i),
    1 1 2 6 24 120 720 5040
    """
    f = 1
    return f
```

5 *jeu de tests*

```
>>> for i in range(8) :
...     print factorielle(i),
...
1 1 1 1 1 1 1 1
```

Remarque sur la validité de l'algorithme

A chaque étape de la spécification, le code de la fonction doit toujours être exécutable même s'il ne donne pas encore le bon résultat.
Le jeu de tests ne pourra être vérifié qu'une fois l'implémentation correctement définie.

Description et préconditions

3. *description*

```
def factorielle(n) :
    """ f = n! """
    f = 1
    return f
```

3. *description*

```
>>> factorielle(5)
1
>>> factorielle(-5)
1
>>> factorielle('toto')
1
```

Description et préconditions

3. *description*

```
def factorielle(n) :
    """ f = n! """
    f = 1
    return f
```

3. *description*

```
>>> factorielle(5)
1
>>> factorielle(-5)
1
>>> factorielle('toto')
1
```

4. *préconditions*

```
def factorielle(n)
    """ f = n! """
    assert type(n) is int
    assert n >= 0
    f = 1
    return f
```

4. *préconditions*

```
>>> factorielle(5)
1
>>> factorielle(-5)
assert n >= 0
>>> factorielle('toto')
assert type(n) is int
```

factorielle(n) : tout en un

```
1  def factorielle(n):
2      """
3      f = n!
4      >>> for i in range(10):
5      ...     print factorielle(i),
6      1 1 2 6 24 120 720 5040 40320 362880
7      >>> factorielle(15)
8      1307674368000L
9      """
10     assert type(n) is int
11     assert n >= 0
12
13     f = 1
14     for i in range(1,n+1): f = f * i
15
16     return f
```

sommeArithmetique(n) : tout en un

```
1  def sommeArithmetique(n):
2      """
3      somme s des n premiers entiers
4
5      >>> for n in range(7):
6      ...     print sommeArithmetique(n) == n*(n+1)/2,
7      True True True True True True True
8      """
9      assert type(n) is int
10     assert n >= 0
11
12     s = n*(n+1)/2
13
14     return s
```

Spécification et implémentation

Spécification d'un algorithme — Quoi ?

La spécification décrit la fonction et l'utilisation d'un algorithme (ce que fait l'algorithme).

Spécification et implémentation

Spécification d'un algorithme — Quoi ?

La spécification décrit la fonction et l'utilisation d'un algorithme (ce que fait l'algorithme).
L'algorithme est vu comme une boîte noire dont on ne connaît pas le fonctionnement interne.

Spécification et implémentation

Spécification d'un algorithme — Quoi ?

La spécification décrit la fonction et l'utilisation d'un algorithme (ce que fait l'algorithme).
L'algorithme est vu comme une boîte noire dont on ne connaît pas le fonctionnement interne.

Implémentation d'un algorithme — Comment ?

L'implémentation décrit le fonctionnement interne de l'algorithme (comment fait l'algorithme).

Spécification et implémentation

Spécification d'un algorithme — Quoi ?

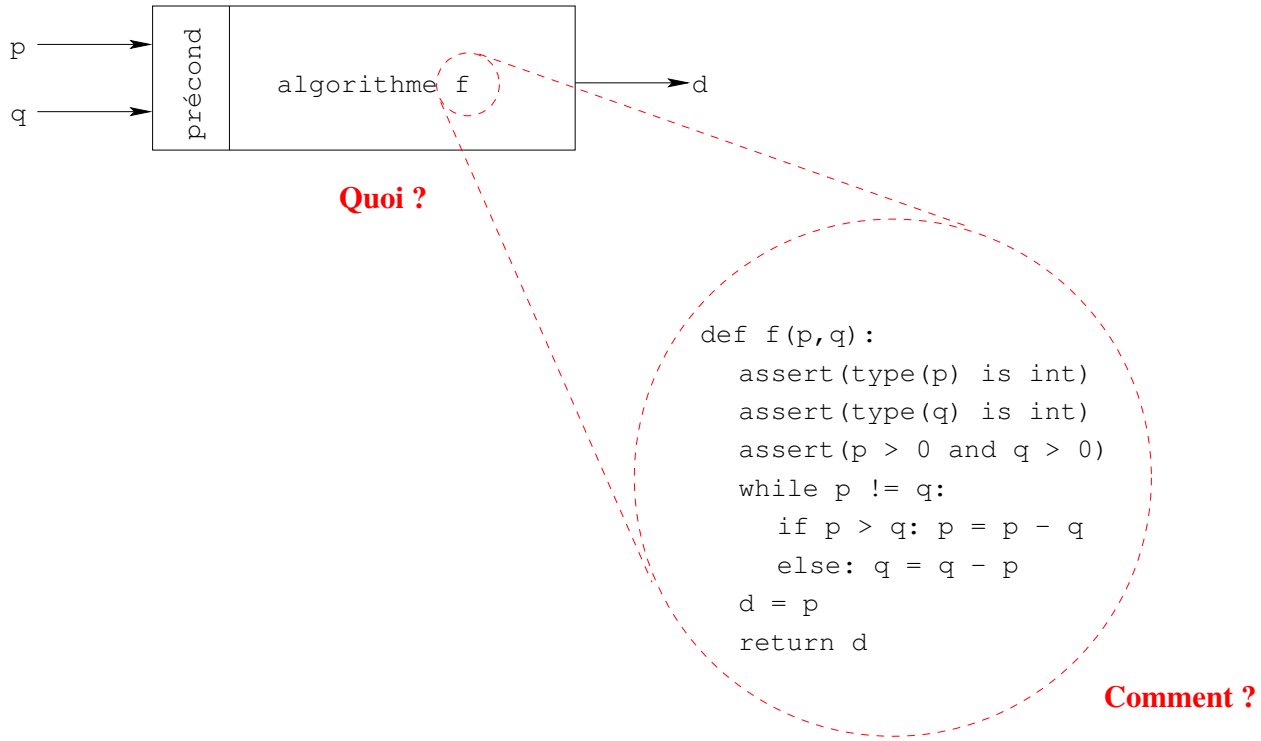
La spécification décrit la fonction et l'utilisation d'un algorithme (ce que fait l'algorithme).
L'algorithme est vu comme une boîte noire dont on ne connaît pas le fonctionnement interne.

Implémentation d'un algorithme — Comment ?

L'implémentation décrit le fonctionnement interne de l'algorithme (comment fait l'algorithme).
L'implémentation précise l'enchaînement des instructions nécessaires à la résolution du problème considéré.

Spécification et implémentation

La spécification décrit la fonction et l'utilisation d'un algorithme



L'implémentation décrit le fonctionnement interne de l'algorithme

Une spécification, des implémentations

```
#-------------------------------------
def sommeArithmetique(n):
#-------------------------------------
    """
    somme s des n premiers entiers

    >>> for n in range(7):
    ...     print sommeArithmetique(n)\
                  == n*(n+1)/2
    True True True True True True True
    """
    assert type(n) is int
    assert n >= 0


    s = n*(n+1)/2

    return s
#-------------------------------------
```

```
#-------------------------------------
def sommeArithmetique(n):
#-------------------------------------
    """
    somme s des n premiers entiers

    >>> for n in range(7):
    ...     print sommeArithmetique(n)\
                  == n*(n+1)/2
    True True True True True True True
    """
    assert type(n) is int
    assert n >= 0

    s = 0
    for i  in range(n+1): s = s + i

    return s
#-------------------------------------
```

Concepteur versus Utilisateur

Concepteur

Le concepteur d'un algorithme définit l'interface et l'implémentation de l'algorithme.

Concepteur versus Utilisateur

Concepteur

Le concepteur d'un algorithme définit l'interface et l'implémentation de l'algorithme.

Utilisateur

L'utilisateur d'un algorithme n'a pas à connaître son implémentation ; seule l'interface de l'algorithme le concerne.

Concepteur versus Utilisateur

Concepteur

Le concepteur d'un algorithme définit l'interface et l'implémentation de l'algorithme.

Utilisateur

L'utilisateur d'un algorithme n'a pas à connaître son implémentation ; seule l'interface de l'algorithme le concerne. Selon la spécification de l'algorithme, l'utilisateur appelle (utilise) l'algorithme sous forme d'une procédure ou d'une fonction.

Fonction = spécification + implémentation

Propriétés d'un algorithme

validité : être conforme aux jeux de tests

Fonction = spécification + implémentation

Propriétés d'un algorithme

validité : être conforme aux jeux de tests

robustesse : vérifier les préconditions

Fonction = spécification + implémentation

Propriétés d'un algorithme

validité : être conforme aux jeux de tests

robustesse : vérifier les préconditions

réutilisabilité : être correctement paramétré