# Semestre 2 / Cours 1 - Bonnes pratiques de programmation

ITC MPSI & PCSI – Année 2024-2025

# Sommaire

# Fonctions et effet de bord

# Copie de variables

## Exemple

```
>>> x = 0
>>> y = x
>>> x = 1
>>> L = [0]
>>> Lp = L
>>> L[0] = 1
```

Donner le contenu des variables `y` et `Lp` après exécution du code.

# Copie de variables

Exemple

```
>>> x = 0
>>> y = x
>>> x = 1
>>> L = [0]
>>> Lp = L
>>> L[0] = 1
```

Donner le contenu des variables `y` et `Lp` après exécution du code.

```
>>> y
0
>>> Lp
[1]
```

# Copie de variables

## Explications

- variable $\Leftrightarrow$ étiquette (nom) donné à un emplacement mémoire,

# Copie de variables

## Explications

- variable $\Leftrightarrow$ étiquette (nom) donné à un emplacement mémoire,
- variable non mutable $\Rightarrow$ le nom désigne une valeur (entier, flottant, chaine de caractère,...),

# Copie de variables

## Explications

- variable $\Leftrightarrow$ étiquette (nom) donné à un emplacement mémoire,
- variable non mutable $\Rightarrow$ le nom désigne une valeur (entier, flottant, chaine de caractère,...),
- variable mutable $\Rightarrow$ le nom désigne un pointeur pointant vers une valeur.

# Copie de variables

## Exemple

```
x = 0
y = x
x = 1
L =[0]
Lp = L
L[0] = 1
```

## Représentation

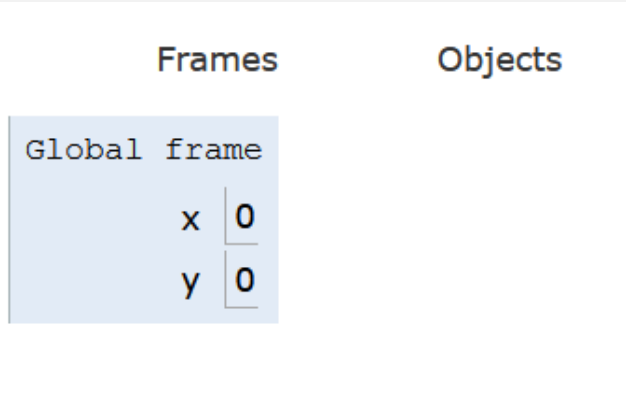



Figure 1 – copie de la variable non mutable x

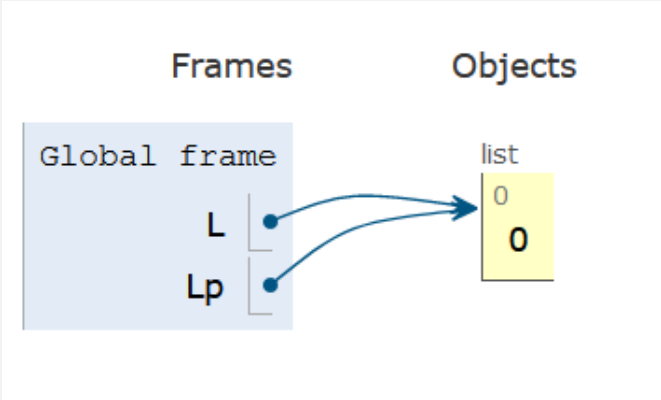



Figure 2 – copie de la variable mutable L

# Fonctions et variables

Exemple

```
def f(x):
    x = 1
x = 0
f(x)

def g(L):
    L[0] = 1
L = [0]
g(L)
```

# Fonctions et variables

Exemple

```
def f(x):
    x = 1
x = 0
f(x)

def g(L):
    L[0] = 1
L = [0]
g(L)
```

Après exécution :

- la valeur associée à la variable `x` est :

Exemple

```
def f(x):
    x = 1
x = 0
f(x)

def g(L):
    L[0] = 1
L = [0]
g(L)
```

Après exécution :

- la valeur associée à la variable x est : 0

Exemple

```
def f(x):
    x = 1
x = 0
f(x)

def g(L):
    L[0] = 1
L = [0]
g(L)
```

Après exécution :

- la valeur associée à la variable x est : 0
- la valeur associée à la variable L est :

Exemple

```
def f(x):
    x = 1
x = 0
f(x)

def g(L):
    L[0] = 1
L = [0]
g(L)
```

Après exécution :

- la valeur associée à la variable `x` est : 0
- la valeur associée à la variable `L` est : `[1]`.

# Fonctions et variables

Exemple

```
def f(x):
    x = 1
x = 0
f(x)

def g(L):
    L[0] = 1
L = [0]
g(L)
```

Après exécution :

- la valeur associée à la variable `x` est : 0
- la valeur associée à la variable `L` est : `[1]`.

$\Rightarrow$ différence de comportement des fonctions suivant le caractère mutable ou non des variables.

Exemple

```
>>> x = 0
>>> f(x)
>>> x
0
>>> L = [0]
>>> g(L)
>>> L
[1]
```

## Représentation

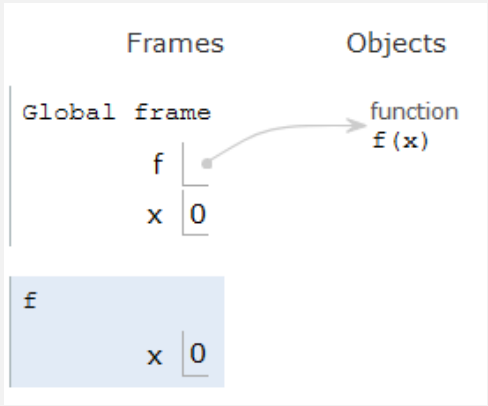



**Figure 3 –** avant modification de **x** par *f*

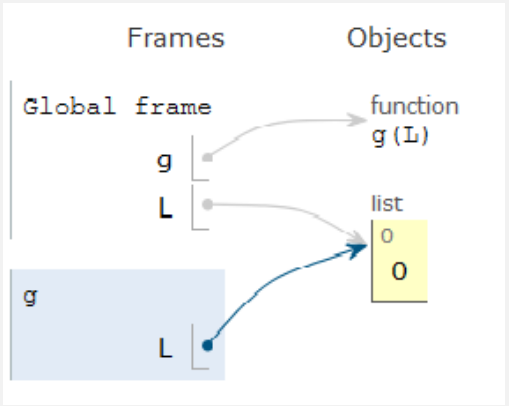



**Figure 4 –** avant modification de **L** par *g*

# Lien avec la copie de variables

## Représentation

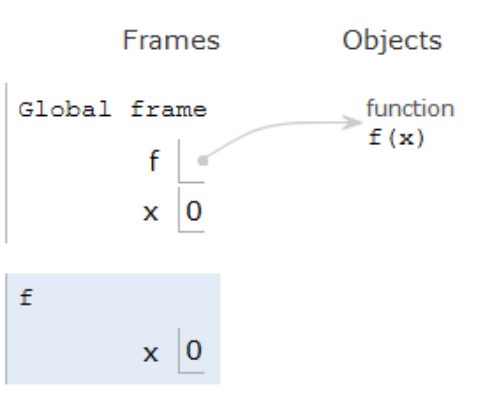



Figure 3 – avant modification de x par *f*

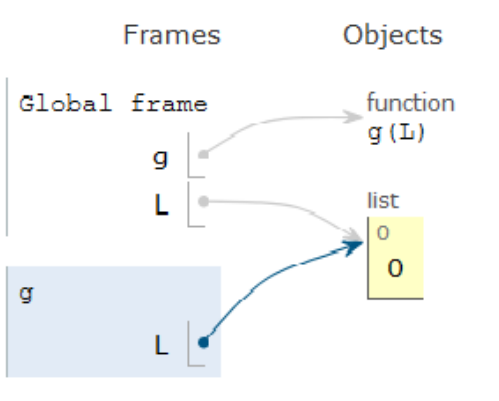



Figure 4 – avant modification de L par *g*

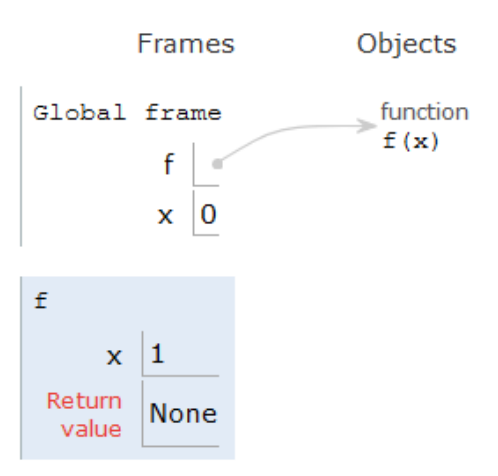



Figure 5 – après modification de x par *f*

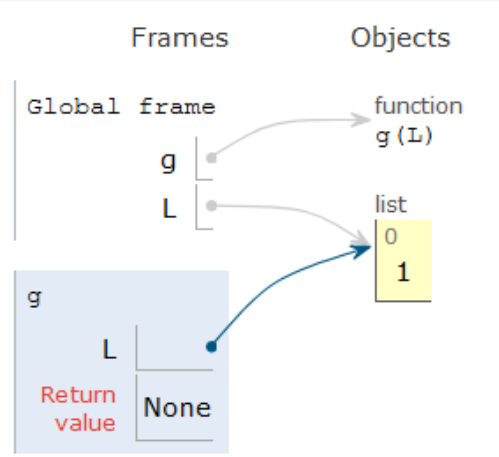



Figure 6 – après modification de L par *g*

**Effet de bord**
En informatique, une fonction est dite à effet de bord si elle modifie un état en dehors de son environnement local, c'est-à-dire a une interaction observable avec le monde extérieur autre que renvoyer une valeur.

**Effet de bord**
En informatique, une fonction est dite à effet de bord si elle modifie un état en dehors de son environnement local, c'est-à-dire a une interaction observable avec le monde extérieur autre que renvoyer une valeur.

- la façon dont une fonction modifie le contenu d'une variable d'entrée dépend du caractère mutable ou non de cette variable,

**Effet de bord**
En informatique, une fonction est dite à effet de bord si elle modifie un état en dehors de son environnement local, c'est-à-dire a une interaction observable avec le monde extérieur autre que renvoyer une valeur.

- la façon dont une fonction modifie le contenu d'une variable d'entrée dépend du caractère mutable ou non de cette variable,
- deux comportements distincts :

**Effet de bord**
En informatique, une fonction est dite à effet de bord si elle modifie un état en dehors de son environnement local, c'est-à-dire a une interaction observable avec le monde extérieur autre que renvoyer une valeur.

- la façon dont une fonction modifie le contenu d'une variable d'entrée dépend du caractère mutable ou non de cette variable,
- deux comportements distincts :
  - ◇ modification de la valeur de `x`, non mutable, dans le programme principal ⇒ `x = f(x)`,

**Effet de bord**
En informatique, une fonction est dite à effet de bord si elle modifie un état en dehors de son environnement local, c'est-à-dire a une interaction observable avec le monde extérieur autre que renvoyer une valeur.

- la façon dont une fonction modifie le contenu d'une variable d'entrée dépend du caractère mutable ou non de cette variable,
- deux comportements distincts :
  - ◇ modification de la valeur de `x`, non mutable, dans le programme principal ⇒ `x = f(x)`,
  - ◇ modification de la valeur de `L`, mutable, dans le programme principal ⇒ `g(L)`,

# Instruction VS expression

L'instruction d'affectation =

- réalise une action (associe une valeur à une variable),

# Instruction VS expression

L'instruction d'affectation =

- réalise une action (associe une valeur à une variable),
- ne renvoie rien (comme `import`, `for`, `while`, `if`, `else`),

# Instruction VS expression

L'instruction d'affectation `=`

- réalise une action (associe une valeur à une variable),
- ne renvoie rien (comme `import`, `for`, `while`, `if`, `else`),
- `print(a = 1)`, `if a = 1` renvoient des erreurs.

# Instruction VS expression

## L'instruction d'affectation `=`

- réalise une action (associe une valeur à une variable),
- ne renvoie rien (comme `import`, `for`, `while`, `if`, `else`),
- `print(a = 1)`, `if a = 1` renvoient des erreurs.

## Expression avec `==`

- `==` permet la création d'expressions (`a == 1`),

# Instruction VS expression

## L'instruction d'affectation =

- réalise une action (associe une valeur à une variable),
- ne renvoie rien (comme `import`, `for`, `while`, `if`, `else`),
- `print(a = 1)`, `if a = 1` renvoient des erreurs.

## Expression avec ==

- `==` permet la création d'expressions (`a == 1`),
- valeur renvoyée booléenne `True` ou `False`,

# Instruction VS expression

## L'instruction d'affectation =

- réalise une action (associe une valeur à une variable),
- ne renvoie rien (comme `import`, `for`, `while`, `if`, `else`),
- `print(a = 1)`, `if a = 1` renvoient des erreurs.

## Expression avec ==

- `==` permet la création d'expressions (`a == 1`),
- valeur renvoyée booléenne `True` ou `False`,
- `print(a == 1)` affiche `True` ou `False` (si `a` défini en amont),

# Instruction VS expression

## L'instruction d'affectation =

- réalise une action (associe une valeur à une variable),
- ne renvoie rien (comme `import`, `for`, `while`, `if`, `else`),
- `print(a = 1)`, `if a = 1` renvoient des erreurs.

## Expression avec ==

- `==` permet la création d'expressions (`a == 1`),
- valeur renvoyée booléenne `True` ou `False`,
- `print(a == 1)` affiche `True` ou `False` (si `a` défini en amont),
- `if a == 1` syntaxe acceptée.

# Instruction VS expression

## L'instruction d'affectation =

- réalise une action (associe une valeur à une variable),
- ne renvoie rien (comme `import`, `for`, `while`, `if`, `else`),
- `print(a = 1)`, `if a = 1` renvoient des erreurs.

## Expression avec ==

- `==` permet la création d'expressions (`a == 1`),
- valeur renvoyée booléenne `True` ou `False`,
- `print(a == 1)` affiche `True` ou `False` (si `a` défini en amont),
- `if a == 1` syntaxe acceptée.

## Avantages et inconvénients

- nécessité de différencier lors de la programmation les deux actions (affectation ou comparaison),

# Instruction VS expression

## L'instruction d'affectation =

- réalise une action (associe une valeur à une variable),
- ne renvoie rien (comme `import`, `for`, `while`, `if`, `else`),
- `print(a = 1)`, `if a = 1` renvoient des erreurs.

## Expression avec ==

- `==` permet la création d'expressions (`a == 1`),
- valeur renvoyée booléenne `True` ou `False`,
- `print(a == 1)` affiche `True` ou `False` (si `a` défini en amont),
- `if a == 1` syntaxe acceptée.

## Avantages et inconvénients

- nécessité de différencier lors de la programmation les deux actions (affectation ou comparaison),
- évite la création de bugs issus de confusions entre les deux actions,

# Instruction VS expression

## L'instruction d'affectation =

- réalise une action (associe une valeur à une variable),
- ne renvoie rien (comme `import`, `for`, `while`, `if`, `else`),
- `print(a = 1)`, `if a = 1` renvoient des erreurs.

## Expression avec ==

- `==` permet la création d'expressions (`a == 1`),
- valeur renvoyée booléenne `True` ou `False`,
- `print(a == 1)` affiche `True` ou `False` (si `a` défini en amont),
- `if a == 1` syntaxe acceptée.

## Avantages et inconvénients

- nécessité de différencier lors de la programmation les deux actions (affectation ou comparaison),
- évite la création de bugs issus de confusions entre les deux actions,
- comportement différents dans d'autres langages (exemple en C, `if a = 1` est accepté, mais ne teste pas l'égalité entre `a` et 1).

# Spécifications

# Exprimer un besoin

Pourquoi spécifier une fonction ?

## Pourquoi spécifier une fonction ?

- améliorer la compréhension de la fonction,

# Exprimer un besoin

**Pourquoi spécifier une fonction ?**

- améliorer la compréhension de la fonction,
- clarifier l'interaction entre fonctions,

# Exprimer un besoin

**Pourquoi spécifier une fonction ?**

- améliorer la compréhension de la fonction,
- clarifier l'interaction entre fonctions,
- faciliter la relecture de code,

# Exprimer un besoin

**Pourquoi spécifier une fonction ?**

- améliorer la compréhension de la fonction,
- clarifier l'interaction entre fonctions,
- faciliter la relecture de code,
- faciliter le travail collaboratif.

# Exprimer un besoin

## Pourquoi spécifier une fonction ?

- améliorer la compréhension de la fonction,
- clarifier l'interaction entre fonctions,
- faciliter la relecture de code,
- faciliter le travail collaboratif.

## Qu'est-ce que spécifier une fonction ?

# Exprimer un besoin

## Pourquoi spécifier une fonction ?

- améliorer la compréhension de la fonction,
- clarifier l'interaction entre fonctions,
- faciliter la relecture de code,
- faciliter le travail collaboratif.

## Qu'est-ce que spécifier une fonction ?

- préciser le rôle de la fonction,

# Exprimer un besoin

## Pourquoi spécifier une fonction ?

- améliorer la compréhension de la fonction,
- clarifier l'interaction entre fonctions,
- faciliter la relecture de code,
- faciliter le travail collaboratif.

## Qu'est-ce que spécifier une fonction ?

- préciser le rôle de la fonction,
- préciser la nature des valeurs d'entrée (appelées paramètres ou variables d'entrée),

# Exprimer un besoin

## Pourquoi spécifier une fonction ?

- améliorer la compréhension de la fonction,
- clarifier l'interaction entre fonctions,
- faciliter la relecture de code,
- faciliter le travail collaboratif.

## Qu'est-ce que spécifier une fonction ?

- préciser le rôle de la fonction,
- préciser la nature des valeurs d'entrée (appelées paramètres ou variables d'entrée),
- préciser la nature de la valeur renvoyée.

# Signature d'une fonction

**Principe**
Pour une fonction :

- précise les types des paramètres d'entrée,

# Signature d'une fonction

**Principe**
Pour une fonction :

- précise les types des paramètres d'entrée,
- précise le type de la valeur renvoyée,

# Signature d'une fonction

**Principe**
Pour une fonction :

- précise les types des paramètres d'entrée,
- précise le type de la valeur renvoyée,
- types acceptés : `int`, `float`, `str`, `list`,`dict`, `np.array`, etc.,

# Signature d'une fonction

**Principe**
Pour une fonction :

- précise les types des paramètres d'entrée,
- précise le type de la valeur renvoyée,
- types acceptés : `int`, `float`, `str`, `list`,`dict`, `np.array`, etc.,
- cas particulier d'une fonction ne renvoyant rien : type de sortie **`None`**.

# Signature d'une fonction

**Principe**
Pour une fonction :

- précise les types des paramètres d'entrée,
- précise le type de la valeur renvoyée,
- types acceptés : `int`, `float`, `str`, `list`,`dict`, `np.array`, etc.,
- cas particulier d'une fonction ne renvoyant rien : type de sortie **`None`**.

**Notation générale**

```
def func(param1:type1, param2:type2, ...)->typeSortie:
```

# Exemples de signatures

- fonction d'addition : `add(x:float,y:float)->float`

# Exemples de signatures

- fonction d'addition : `add(x:float,y:float)->float`
- fonction de tri d'une liste renvoyant une nouvelle liste triée : `sort(L:list)->list`

# Exemples de signatures

- fonction d'addition : `add(x:float,y:float)->float`
- fonction de tri d'une liste renvoyant une nouvelle liste triée : `sort(L:list)->list`
- fonction de tri d'une liste effectuant le tri de la liste passée en argument par effet de bord (et donc ne renvoyant rien) : `sort(L:list)->None`

Possibilité de spécifier le type des éléments qui constituent une structure composée (comme listes et tableaux)

# Exemples de signatures

- fonction d'addition : `add(x:float,y:float)->float`
- fonction de tri d'une liste renvoyant une nouvelle liste triée : `sort(L:list)->list`
- fonction de tri d'une liste effectuant le tri de la liste passée en argument par effet de bord (et donc ne renvoyant rien) : `sort(L:list)->None`

Possibilité de spécifier le type des éléments qui constituent une structure composée (comme listes et tableaux)

- Pour une fonction travaillant sur une liste d'entiers `f(L:[int])->int` (au lieu de `f(L:list)->int`),

# Exemples de signatures

- fonction d'addition : `add(x:float,y:float)->float`
- fonction de tri d'une liste renvoyant une nouvelle liste triée : `sort(L:list)->list`
- fonction de tri d'une liste effectuant le tri de la liste passée en argument par effet de bord (et donc ne renvoyant rien) : `sort(L:list)->None`

Possibilité de spécifier le type des éléments qui constituent une structure composée (comme listes et tableaux)

- Pour une fonction travaillant sur une liste d'entiers `f(L:[int])->int` (au lieu de `f(L:list)->int`),
- pour une fonction renvoyant une liste de listes composées chacune d'un entier et d'une chaine de caractère `f(d:dict)->[[int,str]]`.

# Docstring d'une fonction

**Principe**
Texte placé juste après le nom de la fonction, délimité par des triples quotes (`""" """`) et contenant :

# Docstring d'une fonction

**Principe**
Texte placé juste après le nom de la fonction, délimité par des triples quotes (`""" """`) et contenant :

- une brève description du rôle de la fonction,

# Docstring d'une fonction

**Principe**
Texte placé juste après le nom de la fonction, délimité par des triples quotes (`" " "`) et contenant :

- une brève description du rôle de la fonction,
- une section Parameters précisant le nom,type et la description de chacune des variables d'entrée,

# Docstring d'une fonction

**Principe**
Texte placé juste après le nom de la fonction, délimité par des triples quotes (`""" """`) et contenant :

- une brève description du rôle de la fonction,
- une section Parameters précisant le nom,type et la description de chacune des variables d'entrée,
- une section Returns précisant type et description de la valeur de sortie,

# Docstring d'une fonction

**Principe**
Texte placé juste après le nom de la fonction, délimité par des triples quotes (`""" """`) et contenant :

- une brève description du rôle de la fonction,
- une section Parameters précisant le nom,type et la description de chacune des variables d'entrée,
- une section Returns précisant type et description de la valeur de sortie,
- une section Examples (optionnelle mais recommandée) illustrant le fonctionnement de la fonction.

# Docstring d'une fonction

## Exemple

Pour la fonction `add(x:float,y:float)->float` :

```
def add(x:float,y:float)->float:
    """
    Calculate the sum of 'x' and 'y'
    Parameters
    ----------
    x : float
        first value to add
    y : float
        second value to add
    Returns
    -------
    float
        the sum of 'x' and 'y'

    Examples
    --------
    >>> add(1,2)
    3

    """
    return x+y
```

# Usages des signatures et doctrings

- utilisation simultanée de la signature et doctring non nécessaire,

# Usages des signatures et doctrings

- utilisation simultanée de la signature et doctring non nécessaire,
- signature : à privilégier lors de la description d'une fonction, sans écriture du code de celle-ci (parler de `add(x:float,y:float)->float` plutôt que de `add(x,y)` ),

# Usages des signatures et doctrings

- utilisation simultanée de la signature et doctring non nécessaire,
- signature : à privilégier lors de la description d'une fonction, sans écriture du code de celle-ci (parler de `add(x:float,y:float)->float` plutôt que de `add(x,y)` ),
- docstring : obligatoire lors de l'écriture du code d'une fonction,

# Usages des signatures et doctrings

- utilisation simultanée de la signature et doctring non nécessaire,
- signature : à privilégier lors de la description d'une fonction, sans écriture du code de celle-ci (parler de `add(x:float,y:float)->float` plutôt que de `add(x,y)` ),
- docstring : obligatoire lors de l'écriture du code d'une fonction,
- la signature ne doit pas être utilisée lors de l'appel de la fonction,

# Usages des signatures et doctrings

- utilisation simultanée de la signature et doctring non nécessaire,
- signature : à privilégier lors de la description d'une fonction, sans écriture du code de celle-ci (parler de `add(x:float,y:float)->float` plutôt que de `add(x,y)` ),
- docstring : obligatoire lors de l'écriture du code d'une fonction,
- la signature ne doit pas être utilisée lors de l'appel de la fonction,
  - ◇ bonne utilisation : `>>>add(2,3)` renvoie 5,

# Usages des signatures et doctrings

- utilisation simultanée de la signature et doctring non nécessaire,
- signature : à privilégier lors de la description d'une fonction, sans écriture du code de celle-ci (parler de `add(x:float,y:float)->float` plutôt que de `add(x,y)` ),
- docstring : obligatoire lors de l'écriture du code d'une fonction,
- la signature ne doit pas être utilisée lors de l'appel de la fonction,
  - ◇ bonne utilisation : `>>>add(2,3)` renvoie 5,
  - ◇ mauvaise utilisation : `>>>add(2:float,3:float)->float` génère une erreur.

# Exercice

Fonction inconnue

```
def f(a,b):
    if b == 0 :
        return a
    return f(b, a%b)
```

# Exercice

## Fonction inconnue

```
def f(a,b):
    if b == 0 :
        return a
    return f(b, a%b)
```

## Questions

- Que renvoient `f(15,10)` et `f(35,21)` ? Conjecturer le rôle de `f(a,b)` pour `a` et `b` quelconques.
- Préciser la signature et la docstring pour `f`, et renommer la fonction.

# Fonction inconnue

- Pour $a = 15$ et $b = 10$, on obtient successivement pour le couple $(a, b)$ : $(10, 5)$, $(5, 0)$. La valeur renvoyée est donc 5.

# Fonction inconnue

- Pour $a = 15$ et $b = 10$, on obtient successivement pour le couple $(a, b)$ : $(10, 5)$, $(5, 0)$. La valeur renvoyée est donc 5.
- Pour $a = 35$ et $b = 21$, on obtient successivement pour le couple $(a, b)$ : $(21, 14)$, $(14, 7)$, $(7, 0)$. La valeur renvoyée est donc 7.

# Fonction inconnue

- Pour $a = 15$ et $b = 10$, on obtient successivement pour le couple $(a, b)$ : $(10, 5)$, $(5, 0)$. La valeur renvoyée est donc 5.
- Pour $a = 35$ et $b = 21$, on obtient successivement pour le couple $(a, b)$ : $(21, 14)$, $(14, 7)$, $(7, 0)$. La valeur renvoyée est donc 7.
- Dans ces deux exemples, la fonction renvoie le pgcd (plus grand commun diviseur) de $a, b$.

# Fonction renommée et avec spécifications

```
def pgcd(a:int,b:int)->int:
    """ Returns the pgcd of 'a' and 'b'
    Parameters
    ----------
    a : int  first value to add
    b : int  second value to add
    Returns
    -------
    int      pgcd of 'a' and 'b'
    Examples
    --------
    >>> pgcd(15,10)
    5
    >>> pgcd(35,21)
    7
    """
    if b == 0 :
        return a
    return pgcd(b, a%b)
```

# Recherche dichotomique

```
def dicho(L, v):
    i_deb = 0
    i_fin = len(L)
    trouve = False
    while not trouve and i_deb < i_fin:
        i_m = (i_deb + i_fin) // 2
        if L[i_m] == v:
            trouve = True
        elif L[i_m] < v:
            i_deb = i_m + 1
        else:
            i_fin = i_m
    return trouve
```

# Recherche dichotomique

```
def dicho(L, v):
    i_deb = 0
    i_fin = len(L)
    trouve = False
    while not trouve and i_deb < i_fin:
        i_m = (i_deb + i_fin) // 2
        if L[i_m] == v:
            trouve = True
        elif L[i_m] < v:
            i_deb = i_m + 1
        else:
            i_fin = i_m
    return trouve
```

Préciser la signature et la spécification de cette fonction.

# Recherche dichotomique

```
def dicho(L:list, v:int)->bool:
    """
    Determines if value 'v' belongs to 'L' by dichotomic method.
    Parameters
    ----------
    L : list
        list of values sorted with L[0]<= L[1] <= ... <= L[n-1]
    v : int
        value to be tested
    Returns
    -------
    bool
        True if value 'v' is in 'L', False in the other case
    Examples
    --------
    >>> dicho([1,3,5],5)
    True
    >>> dicho([1,3,5],4)
    False
    >>> dicho([],1)
    False
    """
```

# Fonction de tri (tri à bulles)

```
def tri_bulle(L):
    n = len(L)
    t = n-1
    fini = False
    while t > 0 and not fini:
        fini = True
        for j in range(t):
            if L[j] > L[j+1]:
                L[j],L[j+1] = L[j+1],L[j]
                fini = False
        t -= 1
```

# Fonction de tri (tri à bulles)

```
def tri_bulle(L):
    n = len(L)
    t = n-1
    fini = False
    while t > 0 and not fini:
        fini = True
        for j in range(t):
            if L[j] > L[j+1]:
                L[j],L[j+1] = L[j+1],L[j]
                fini = False
        t -= 1
```

Préciser la signature et la spécification de cette fonction.

# Fonction de tri (tri à bulles)

```
def tri_bulle(L:list)->None:
    """
    Sort the list in ascending order and return None.
    After execution, the list is sorted in ascending order.
    Parameters
    ----------
    L : list
        list to be sorted
    Returns
    -------
    None
    Examples
    --------
    >>> L = [2,3,1]
    >>> tri_bulle(L)
    >>> L
    [1,2,3]
    """
```

# INSTRUCTION `assert`

## Présentation

- spécification $\Rightarrow$ conditions sur les paramètres d'entrée,

# INSTRUCTION `assert`

## Présentation

- spécification $\Rightarrow$ conditions sur les paramètres d'entrée,
- volonté de tester le respect de ces conditions,

# INSTRUCTION `assert`

## Présentation

- spécification ⇒ conditions sur les paramètres d'entrée,
- volonté de tester le respect de ces conditions,
- instruction :

```
assert condition [,message]
```

*(la partie entre crochets est optionnelle)*

# INSTRUCTION `assert`

## Présentation

- spécification $\Rightarrow$ conditions sur les paramètres d'entrée,
- volonté de tester le respect de ces conditions,
- instruction :

```
assert condition [,message]
```

  *(la partie entre crochets est optionnelle)*

- si la condition est vérifiée, on passe à la suite,

# INSTRUCTION `assert`

## Présentation

- spécification $\Rightarrow$ conditions sur les paramètres d'entrée,
- volonté de tester le respect de ces conditions,
- instruction :

```
assert condition [,message]
```

*(la partie entre crochets est optionnelle)*

- si la condition est vérifiée, on passe à la suite,
- sinon, on arrête le programme et on affiche le message (optionnel).

# INSTRUCTION `assert`

Exemples

- `assert x >= 0`
- `assert x >= 0, "la variable 'x' est négative"`
- `assert len(L) != 0, "la liste est vide"`
- `assert type(eps) == float`
- `assert type(x) == float or type(x) == int`

# Instruction `assert`-Exercice

Soit la fonction

```
mot_en_place(mot:str,texte:str,i:int)->bool
```

qui renvoie **`True`** si $mot[j] = texte[i + j], \forall j \in [\![0, m - 1]\!]$, avec `m=len(mot)`, et **`False`** sinon.

# Instruction `assert`-Exercice

Soit la fonction

```
mot_en_place(mot:str,texte:str,i:int)->bool
```

qui renvoie `True` si $mot[j] = texte[i + j], \forall j \in [\![0, m - 1]\!]$, avec `m=len(mot)`, et `False` sinon.

- Déterminer les contraintes que doivent vérifier les variables d'entrée de cette fonction.

# Instruction `assert`-Exercice

Soit la fonction

```
mot_en_place(mot:str,texte:str,i:int)->bool
```

qui renvoie `True` si $mot[j] = texte[i + j], \forall j \in [\![0, m - 1]\!]$, avec `m=len(mot)`, et `False` sinon.

- Déterminer les contraintes que doivent vérifier les variables d'entrée de cette fonction.
- Écrire les lignes de code correspondantes en utilisant `assert`.

# Instruction `assert` : Exercice

- ◇ `mot` et `texte` doivent être des chaines de caractère,

# Instruction `assert` : Exercice

- ◇ `mot` et `texte` doivent être des chaines de caractère,
  ◇ `i` doit être un entier positif ou nul,

# INSTRUCTION `assert` : EXERCICE

- ◇ `mot` et `texte` doivent être des chaines de caractère,
  ◇ `i` doit être un entier positif ou nul,
  ◇ on doit avoir $i + \text{len(mot)} \leq \text{len(texte)}$.

# Instruction `assert` : Exercice

- ◇ `mot` et `texte` doivent être des chaines de caractère,
  ◇ `i` doit être un entier positif ou nul,
  ◇ on doit avoir $i + \text{len(mot)} \leq \text{len(texte)}$.
- ```
  n, m = len(texte), len(mot)
  assert type(mot) == str
  assert type(texte) == str
  assert type(i) == int
  assert i >= 0
  assert i+m <= n
  ```
  ```

# Annotations d'un bloc d'instructions

# Commentaires

- Un code non commenté est un code inexploitable.

# Commentaires

- Un code non commenté est un code inexploitable.
- Plusieurs possibilités pour commenter :
  - ◇ commentaire en ligne : sur la même ligne qu'une ligne de code (code, caractère #, espace, commentaire).

# Commentaires

- Un code non commenté est un code inexploitable.
- Plusieurs possibilités pour commenter :
  - ◇ commentaire en ligne : sur la même ligne qu'une ligne de code (code, caractère #, espace, commentaire).
  - ◇ Bloc de commentaire : avant une partie de code (caractère #, espace, commentaire)

# Commentaires

## Exemple

```
# On initialise deux variables x et y à 1. Puis on \
↪ calcule la somme de x et y. (bloc de commentaire)
x = 1  # Initialisation de x (commentaire en ligne)
y = 1
z = x+y
```

# Commentaires

## Exemple

```
# On initialise deux variables x et y à 1. Puis on \
↪ calcule la somme de x et y. (bloc de commentaire)
x = 1  # Initialisation de x (commentaire en ligne)
y = 1
z = x+y
```

# Utilisation des Commentaires

Problématique usuelle concernant les codes :

- Est-ce que le code se termine ?

# Utilisation des Commentaires

Problématique usuelle concernant les codes :

- Est-ce que le code se termine ?
- Si oui, le résultat obtenu est-il celui attendu ?

# Utilisation des Commentaires

Problématique usuelle concernant les codes :

- Est-ce que le code se termine ?
- Si oui, le résultat obtenu est-il celui attendu ?

Outils utilisables pouvant figurer dans le code sous forme de commentaire :

- Précondition

# Utilisation des Commentaires

Problématique usuelle concernant les codes :

- Est-ce que le code se termine ?
- Si oui, le résultat obtenu est-il celui attendu ?

Outils utilisables pouvant figurer dans le code sous forme de commentaire :

- Précondition
- Postcondition

# Utilisation des Commentaires

Problématique usuelle concernant les codes :

- Est-ce que le code se termine ?
- Si oui, le résultat obtenu est-il celui attendu ?

Outils utilisables pouvant figurer dans le code sous forme de commentaire :

- Précondition
- Postcondition
- Invariant de boucles

# Précondition

Précondition :

propriété ($P$) qui doit être vérifiée avant l'exécution du code.

# Précondition

Précondition :

propriété ($P$) qui doit être vérifiée avant l'exécution du code.

- Exemple de code avec avec $a$ et $b$ entiers naturels

```
x, y = a, b
while x != y:
    if x > y:
        x = x - y
    else:
        y = y - x
```

# Précondition

**Précondition :**

**propriété ($P$) qui doit être vérifiée avant l'exécution du code.**

- Exemple de code avec avec $a$ et $b$ entiers naturels

```
x, y = a, b
while x != y:
    if x > y:
        x = x - y
    else:
        y = y - x
```

- $S$ est donc défini pour $(a, b) \in \mathbb{N}^2$. Examinons l'évolution des valeurs du couple de variables $(x, y)$ dans deux cas.

# Précondition

**Précondition :**

**propriété ($P$) qui doit être vérifiée avant l'exécution du code.**

- Exemple de code avec avec $a$ et $b$ entiers naturels

```
x, y = a, b
while x != y:
    if x > y:
        x = x - y
    else:
        y = y - x
```

- $S$ est donc défini pour $(a, b) \in \mathbb{N}^2$. Examinons l'évolution des valeurs du couple de variables $(x, y)$ dans deux cas.
  - ◇ Pour $a = 14$ et $b = 9$, on a :

**Précondition :**

**propriété ($P$) qui doit être vérifiée avant l'exécution du code.**

- Exemple de code avec avec $a$ et $b$ entiers naturels

```
x, y = a, b
while x != y:
    if x > y:
        x = x - y
    else:
        y = y - x
```

- $S$ est donc défini pour $(a, b) \in \mathbb{N}^2$. Examinons l'évolution des valeurs du couple de variables $(x, y)$ dans deux cas.
  - ◇ Pour $a = 14$ et $b = 9$, on a :

    $$(14, 9) \to (5, 9) \to (5, 4) \to (1, 4) \to (1, 3) \to (1, 2) \to (1, 1)$$

    la boucle s'arrête et on obtient $x = y = 1$.
  - ◇ Pour $a = 14$ et $b = 0$, on a :

**Précondition :**

**propriété ($P$) qui doit être vérifiée avant l'exécution du code.**

- Exemple de code avec avec $a$ et $b$ entiers naturels

```
x, y = a, b
while x != y:
    if x > y:
        x = x - y
    else:
        y = y - x
```

- $S$ est donc défini pour $(a, b) \in \mathbb{N}^2$. Examinons l'évolution des valeurs du couple de variables $(x, y)$ dans deux cas.
  - ◇ Pour $a = 14$ et $b = 9$, on a :

    $$(14, 9) \to (5, 9) \to (5, 4) \to (1, 4) \to (1, 3) \to (1, 2) \to (1, 1)$$

    la boucle s'arrête et on obtient $x = y = 1$.
  - ◇ Pour $a = 14$ et $b = 0$, on a : $(14, 0) \to (14, 0) \to (14, 0) \cdots$ la boucle est infinie, $S$ ne se termine donc jamais dans ce cas.

# Précondition

On peut donc partitionner l'ensemble de départ de $S$ ($\mathbb{N}^2$ ici) en deux :

# Précondition

On peut donc partitionner l'ensemble de départ de $S$ ($\mathbb{N}^2$ ici) en deux :

- L'ensemble $\{(a, b) \in \mathbb{N}^2 \ / \ S \text{ se termine et renvoie le résultat attendu}\}$,

# Précondition

On peut donc partitionner l'ensemble de départ de $S$ ($\mathbb{N}^2$ ici) en deux :

- L'ensemble $\{(a, b) \in \mathbb{N}^2 \ / \ S \text{ se termine et renvoie le résultat attendu}\}$,
- le complémentaire de cet ensemble qui conduit à un résultat erroné en sortie de boucle.

# Précondition

On peut donc partitionner l'ensemble de départ de $S$ ($\mathbb{N}^2$ ici) en deux :

- L'ensemble $\{(a, b) \in \mathbb{N}^2 \ / \ S \text{ se termine et renvoie le résultat attendu}\}$,
- le complémentaire de cet ensemble qui conduit à un résultat erroné en sortie de boucle.

Précondition pour le code $S$ :

$$(P) : \text{« } (a, b) \in \mathbb{N}^2,\ a > 0,\ b > 0 \text{ ».}$$

# Précondition

On peut donc partitionner l'ensemble de départ de $S$ ($\mathbb{N}^2$ ici) en deux :

- L'ensemble $\{(a,b) \in \mathbb{N}^2 \ / \ S \text{ se termine et renvoie le résultat attendu}\}$,
- le complémentaire de cet ensemble qui conduit à un résultat erroné en sortie de boucle.

Précondition pour le code $S$ :

$$(P) : \text{« } (a,b) \in \mathbb{N}^2,\ a > 0,\ b > 0 \text{ ».}$$

Cette précondition peut être annotée en commentaire :

```
# Précondition (P): a et b sont des entiers naturels strictement \
↪ positifs
x, y = a, b
while x != y:
    if x > y:
        x = x-y
    else:
        y = y-x
```

# Postcondition : caractérisation du résultat

Postcondition :

propriété ($Q$) qui doit être vérifiée après l'exécution du code.

# Postcondition : caractérisation du résultat

Postcondition :

**propriété ($Q$) qui doit être vérifiée après l'exécution du code**.

- La postcondition peut figurer dans le code sous forme d'un commentaire,

# Postcondition : caractérisation du résultat

Postcondition :

**propriété ($Q$) qui doit être vérifiée après l'exécution du code**.

- La postcondition peut figurer dans le code sous forme d'un commentaire,
- lorsque le code $S$ est le corps d'une fonction, alors la précondition ($P$) et la postcondition ($Q$) peuvent être mentionnées dans le docstring de la fonction.

# Postcondition : caractérisation du résultat

Postcondition :

**propriété ($Q$) qui doit être vérifiée après l'exécution du code.**

- La postcondition peut figurer dans le code sous forme d'un commentaire,
- lorsque le code $S$ est le corps d'une fonction, alors la précondition ($P$) et la postcondition ($Q$) peuvent être mentionnées dans le docstring de la fonction.

Dans l'exemple précédent, la postcondition ($Q$) pourrait être : « $x = y$ et $x = \text{pgcd}(a, b)$ ».

# Postcondition : caractérisation du résultat

Postcondition :

**propriété ($Q$) qui doit être vérifiée après l'exécution du code.**

- La postcondition peut figurer dans le code sous forme d'un commentaire,
- lorsque le code $S$ est le corps d'une fonction, alors la précondition ($P$) et la postcondition ($Q$) peuvent être mentionnées dans le docstring de la fonction.

Dans l'exemple précédent, la postcondition ($Q$) pourrait être : « $x = y$ et $x = \text{pgcd}(a, b)$ ».

```
# Précondition (P): a et b sont des naturels strictement positifs
x, y = a, b
while x != y:
    if x > y:
        x = x - y
    else:
        y = y - x
# Postcondition (Q): x = y et x = pgcd(a,b)
```

# Notion d'invariant

Invariant :

**proposition qui reste vraie tout au long de l'exécution du code** (ou d'une portion de code)

# Notion d'invariant

Invariant :

**proposition qui reste vraie tout au long de l'exécution du code** (ou d'une portion de code)

- Lors de la preuve d'algorithme (objet du chapitre suivant), il faut prouver l'invariant,

# Notion d'invariant

**Invariant** :

**proposition qui reste vraie tout au long de l'exécution du code** (ou d'une portion de code)

- Lors de la preuve d'algorithme (objet du chapitre suivant), il faut prouver l'invariant,
- on utilise l'invariant pour établir la preuve du programme,

# Notion d'invariant

**Invariant** :

**proposition qui reste vraie tout au long de l'exécution du code** (ou d'une portion de code)

- Lors de la preuve d'algorithme (objet du chapitre suivant), il faut prouver l'invariant,
- on utilise l'invariant pour établir la preuve du programme,
- on peut alors noter cet invariant sous forme de commentaire dans le code,

# Notion d'invariant

**Invariant** :

**proposition qui reste vraie tout au long de l'exécution du code** (ou d'une portion de code)

- Lors de la preuve d'algorithme (objet du chapitre suivant), il faut prouver l'invariant,
- on utilise l'invariant pour établir la preuve du programme,
- on peut alors noter cet invariant sous forme de commentaire dans le code,
- dans l'exemple précédent l'invariant est :
  « $x$ et $y$ sont des naturels non nuls, et $\text{pgcd}(x, y) = \text{pgcd}(a, b)$ ».

# Notion d'invariant : exemple

```
# Précondition (P): a et b sont des naturels strictement positifs
x, y = a, b
while x != y:
# Invariant: x et y sont des naturels strictement positifs
# et pgcd(x,y) = pgcd(a,b).
    if x > y:
        x = x - y  # pgcd(x, y) = pgcd(x - y, y)
    else:
        y = y - x  # pgcd(x, y) = pgcd(x, y - x)
# Postcondition (Q): x = y et x = pgcd(a,b)
```

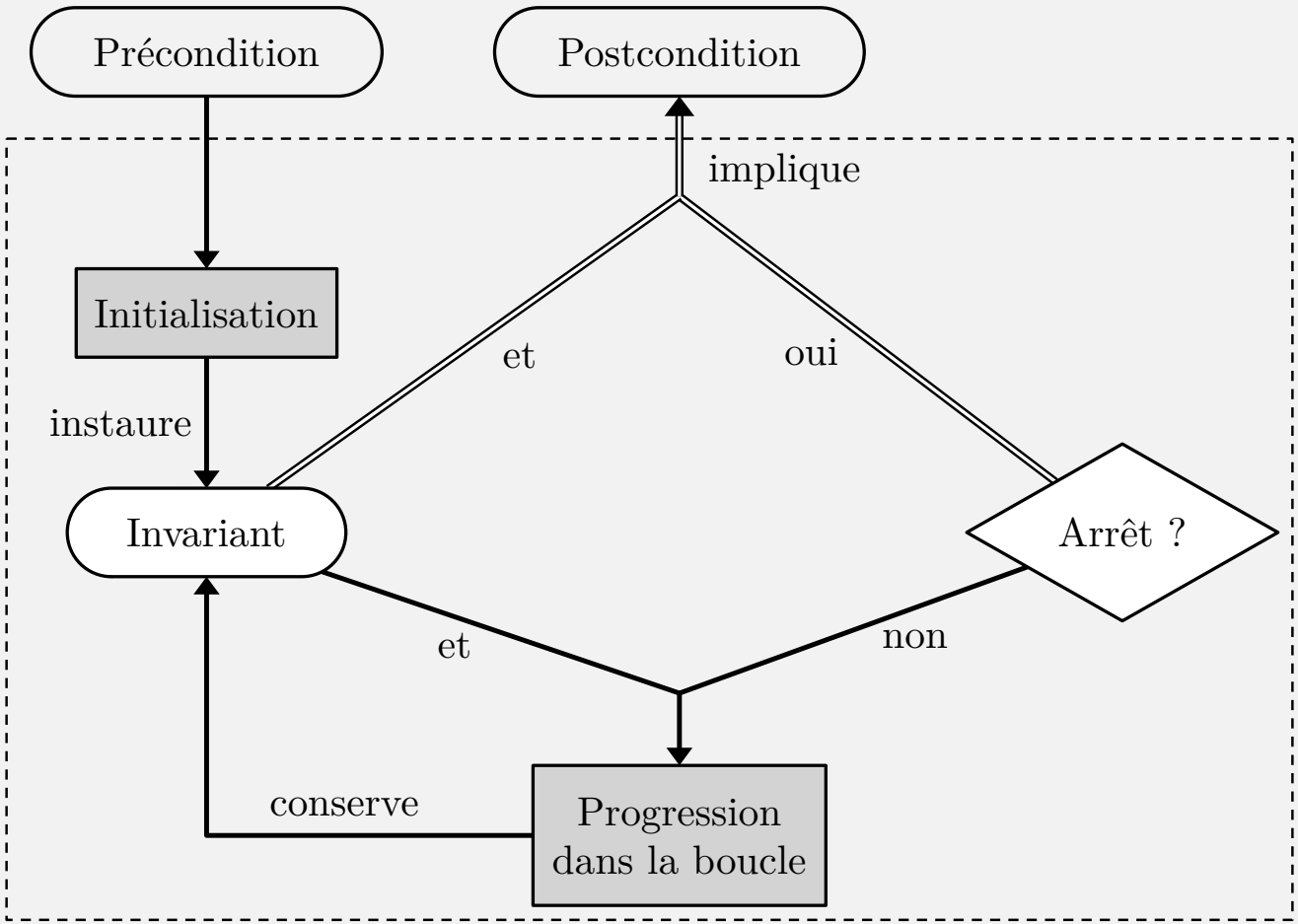
# Notion d'invariant : exemple

```
# Précondition (P): a et b sont des naturels strictement positifs
x, y = a, b
while x != y:
# Invariant: x et y sont des naturels strictement positifs
# et pgcd(x,y) = pgcd(a,b).
    if x > y:
        x = x - y  # pgcd(x, y) = pgcd(x - y, y)
    else:
        y = y - x  # pgcd(x, y) = pgcd(x, y - x)
# Postcondition (Q): x = y et x = pgcd(a,b)
```

Lorsque la condition n'est plus remplie, c'est à dire lorsque $x$ et $y$ sont égaux, on sort de la boucle, on a alors $x = y$ et d'après l'invariant on a aussi $\text{pgcd}(x, y) = \text{pgcd}(a, b)$, or $\text{pgcd}(x, y) = \text{pgcd}(x, x) = x$ ($x$ est un naturel non nul), ce qui prouve la postcondition $(Q)$.

# Notion d'invariant

En résumé, lorsque nous avons affaire à une boucle, nous pouvons représenter la situation ainsi :




Les cases grisées correspondent à une action.

# Exercice

Écrire la fonction `somme(n: int) -> int` respectant la spécification suivante :

- Précondition : $n \in \mathbb{N}$.
- Postcondition : la valeur renvoyée est $\sum_{k=0}^{n} k$.

On précisera dans le code un invariant de la boucle.

# Solution

```
def somme(n: int) -> int:
    """
    Parameters
    ----------
    n: int assumed positive (précondition)
    Returns
    -------
    int
        0+1+...+n (postcondition)
    Examples
    --------
    >>> somme(10)
    55
    """
    resultat = 0  # variable qui contiendra la somme cherchée
    for k in range(1,n+1):  # pour k allant de 1 à n
    # Invariant : resultat est la somme des entiers de 0 à k
        resultat = resultat + k
    return resultat
```

# Jeux de tests

# Utilité des tests

Une fois qu'un programme est prouvé (on a démontré qu'il se comporte comme attendu dans toutes les situations possibles) il peut sembler inutile de le tester, mais :

# Utilité des tests

Une fois qu'un programme est prouvé (on a démontré qu'il se comporte comme attendu dans toutes les situations possibles) il peut sembler inutile de le tester, mais :

- Des erreurs peuvent avoir été commises dans l'étude théorique

# Utilité des tests

Une fois qu'un programme est prouvé (on a démontré qu'il se comporte comme attendu dans toutes les situations possibles) il peut sembler inutile de le tester, mais :

- Des erreurs peuvent avoir été commises dans l'étude théorique
- la preuve peut être très longue et/ou très délicate à établir.

# Utilité des tests

Une fois qu'un programme est prouvé (on a démontré qu'il se comporte comme attendu dans toutes les situations possibles) il peut sembler inutile de le tester, mais :

- Des erreurs peuvent avoir été commises dans l'étude théorique
- la preuve peut être très longue et/ou très délicate à établir.

Dans tous les cas, il est ainsi nécessaire de multiplier les tests, pour chaque fonction écrite, et pour l'ensemble du programme.

# Utilité des tests

- Un test compare le résultat d'une fonction (ou d'un bout de code) exécutée avec des valeurs particulières pour les paramètres d'entrée et la valeur attendue en retour.

# Utilité des tests

- Un test compare le résultat d'une fonction (ou d'un bout de code) exécutée avec des valeurs particulières pour les paramètres d'entrée et la valeur attendue en retour.
- Le choix des tests se porte en général sur des cas limites des valeurs d'entrée (qui dépendent du code testé).

# Utilité des tests

- Un test compare le résultat d'une fonction (ou d'un bout de code) exécutée avec des valeurs particulières pour les paramètres d'entrée et la valeur attendue en retour.
- Le choix des tests se porte en général sur des cas limites des valeurs d'entrée (qui dépendent du code testé).
- Un test ne prouve pas le code :
  « le test de programmes peut être une façon très efficace de montrer la présence de bugs, mais il est désespérément inadéquat pour prouver leur absence » (Edsger Dijkstra).

# Utilité des tests

- Un test compare le résultat d'une fonction (ou d'un bout de code) exécutée avec des valeurs particulières pour les paramètres d'entrée et la valeur attendue en retour.
- Le choix des tests se porte en général sur des cas limites des valeurs d'entrée (qui dépendent du code testé).
- Un test ne prouve pas le code :
  « le test de programmes peut être une façon très efficace de montrer la présence de bugs, mais il est désespérément inadéquat pour prouver leur absence » (Edsger Dijkstra).
- Quand prévoir un jeu de tests ?

# Utilité des tests

- Un test compare le résultat d'une fonction (ou d'un bout de code) exécutée avec des valeurs particulières pour les paramètres d'entrée et la valeur attendue en retour.
- Le choix des tests se porte en général sur des cas limites des valeurs d'entrée (qui dépendent du code testé).
- Un test ne prouve pas le code :
  « le test de programmes peut être une façon très efficace de montrer la présence de bugs, mais il est désespérément inadéquat pour prouver leur absence » (Edsger Dijkstra).
- Quand prévoir un jeu de tests ? Dès la spécification.

# Exemple

- Soit la fonction `division(a, b)` dont la spécification est la suivante :

# Exemple

- Soit la fonction `division(a, b)` dont la spécification est la suivante :
  - Paramètres d'entrée : $a$ et $b$ sont des entiers positifs avec $b$ non nul.

# Exemple

- Soit la fonction `division(a, b)` dont la spécification est la suivante :
  - ◇ Paramètres d'entrée : $a$ et $b$ sont des entiers positifs avec $b$ non nul.
  - ◇ Résultat renvoyé : le résultat de la fonction est le tuple $(q, r)$ où $q$ et $r$ sont respectivement le quotient et le reste de la division euclidienne de $a$ par $b$ (c'est à dire vérifiant $a = bq + r$ avec $0 \leqslant r < b$).

# Exemple

- Soit la fonction `division(a, b)` dont la spécification est la suivante :
  - ◇ Paramètres d'entrée : $a$ et $b$ sont des entiers positifs avec $b$ non nul.
  - ◇ Résultat renvoyé : le résultat de la fonction est le tuple $(q, r)$ où $q$ et $r$ sont respectivement le quotient et le reste de la division euclidienne de $a$ par $b$ (c'est à dire vérifiant $a = bq + r$ avec $0 \leqslant r < b$).
- Choix d'action si la contrainte sur les paramètres d'entrée (précondition) n'est pas remplie :

# Exemple

- Soit la fonction `division(a, b)` dont la spécification est la suivante :
  - ◇ Paramètres d'entrée : $a$ et $b$ sont des entiers positifs avec $b$ non nul.
  - ◇ Résultat renvoyé : le résultat de la fonction est le tuple $(q, r)$ où $q$ et $r$ sont respectivement le quotient et le reste de la division euclidienne de $a$ par $b$ (c'est à dire vérifiant $a = bq + r$ avec $0 \leqslant r < b$).
- Choix d'action si la contrainte sur les paramètres d'entrée (précondition) n'est pas remplie :
  - ◇ générer une erreur avec l'instruction `assert`,

# Exemple

- Soit la fonction `division(a, b)` dont la spécification est la suivante :
  - ◇ Paramètres d'entrée : $a$ et $b$ sont des entiers positifs avec $b$ non nul.
  - ◇ Résultat renvoyé : le résultat de la fonction est le tuple $(q, r)$ où $q$ et $r$ sont respectivement le quotient et le reste de la division euclidienne de $a$ par $b$ (c'est à dire vérifiant $a = bq + r$ avec $0 \leqslant r < b$).
- Choix d'action si la contrainte sur les paramètres d'entrée (précondition) n'est pas remplie :
  - ◇ générer une erreur avec l'instruction `assert`,
  - ◇ renvoyer un résultat particulier.

# Exemple

- Soit la fonction `division(a, b)` dont la spécification est la suivante :
  - ◇ Paramètres d'entrée : $a$ et $b$ sont des entiers positifs avec $b$ non nul.
  - ◇ Résultat renvoyé : le résultat de la fonction est le tuple $(q, r)$ où $q$ et $r$ sont respectivement le quotient et le reste de la division euclidienne de $a$ par $b$ (c'est à dire vérifiant $a = bq + r$ avec $0 \leqslant r < b$).
- Choix d'action si la contrainte sur les paramètres d'entrée (précondition) n'est pas remplie :
  - ◇ générer une erreur avec l'instruction `assert`,
  - ◇ renvoyer un résultat particulier.
- Comportement respectifs :
  - ◇ 1er cas : la fin du programme avec éventuellement un message d'erreur, ce qui n'est pas toujours souhaitable

# Exemple

- Soit la fonction `division(a, b)` dont la spécification est la suivante :
  - ◇ Paramètres d'entrée : $a$ et $b$ sont des entiers positifs avec $b$ non nul.
  - ◇ Résultat renvoyé : le résultat de la fonction est le tuple $(q, r)$ où $q$ et $r$ sont respectivement le quotient et le reste de la division euclidienne de $a$ par $b$ (c'est à dire vérifiant $a = bq + r$ avec $0 \leqslant r < b$).
- Choix d'action si la contrainte sur les paramètres d'entrée (précondition) n'est pas remplie :
  - ◇ générer une erreur avec l'instruction `assert`,
  - ◇ renvoyer un résultat particulier.
- Comportement respectifs :
  - ◇ 1er cas : la fin du programme avec éventuellement un message d'erreur, ce qui n'est pas toujours souhaitable,
  - ◇ 2ième cas : on peut choisir de renvoyer `None` si la contrainte n'est pas remplie. Cette convention devra être mentionnée dans la docstring.

# Exemple

- Exemples de tests :
  - ◇ $a$ et $b$ strictement positifs : `division(19, 7)` doit renvoyer `(2, 5)`,

# Exemple

- Exemples de tests :
  - ◇ $a$ et $b$ strictement positifs : `division(19, 7)` doit renvoyer `(2, 5)`,
  - ◇ $a$ et $b$ strictement positifs : `division(7, 19)` doit renvoyer `(0, 7)`,

# Exemple

- Exemples de tests :
  - $a$ et $b$ strictement positifs : `division(19, 7)` doit renvoyer `(2, 5)`,
  - $a$ et $b$ strictement positifs : `division(7, 19)` doit renvoyer `(0, 7)`,
  - $a$ nul et $b$ strictement positif : `division(0, 19)` doit renvoyer `(0, 0)`,
  - $a$ positif et $b$ nul : `division(19, 0)` doit renvoyer **`None`**,
  - $a$ positif et $b$ négatif : `division(19, -7)` doit renvoyer **`None`**,
  - $a$ négatif et $b$ positif : `division(-19, 7)` doit renvoyer **`None`**,
  - $a$ négatif et $b$ nul : `division(-19, 0)` doit renvoyer **`None`**,
  - $a$ négatif et $b$ négatif : `division(-19, -7)` doit renvoyer **`None`**.

# Exemple

- Exemples de tests :
  - $a$ et $b$ strictement positifs : `division(19, 7)` doit renvoyer `(2, 5)`,
  - $a$ et $b$ strictement positifs : `division(7, 19)` doit renvoyer `(0, 7)`,
  - $a$ nul et $b$ strictement positif : `division(0, 19)` doit renvoyer `(0, 0)`,
  - $a$ positif et $b$ nul : `division(19, 0)` doit renvoyer **`None`**,
  - $a$ positif et $b$ négatif : `division(19, -7)` doit renvoyer **`None`**,
  - $a$ négatif et $b$ positif : `division(-19, 7)` doit renvoyer **`None`**,
  - $a$ négatif et $b$ nul : `division(-19, 0)` doit renvoyer **`None`**,
  - $a$ négatif et $b$ négatif : `division(-19, -7)` doit renvoyer **`None`**.
- On pourrait bien sûr imaginer d'autres valeurs numériques, ou même choisir des valeurs aléatoirement dans chaque cas. Nous allons faire figurer ces tests dans la docstring sous forme d'exemples.

# Exemple

```
def division(a: int, b: int) -> (int, int):
    """
    Renvoie le quotient et le reste de la division de a par b
    Paramètres:
    ----------
        a: int, entier naturel
        b: int, entier strictement positif
    Retour:
    ------
        tuple (q, r) tel que a=bq+r avec 0<=r<b
        ou None si a<0 ou b<=0
    Exemples:
    --------
    >>> division(19,7)
    (2,5)
    >>> division(7,19)
    (0,7)
    >>> division(0,19)
    (0,0)
    >>> division(19,0)
    None
    >>> division(19,-7)
    None
```

# Exemple

```
    """
    >>> division(-19,7)
    None
    >>> division(-19,0)
    None
    >>> division(-19,-7)
    None
    """
```

# Exemple

```
    """
    >>> division(-19,7)
    None
    >>> division(-19,0)
    None
    >>> division(-19,-7)
    None
    """
```

Nous pouvons maintenant passer à l'écriture du code.

# Exemple

```
    """
    >>> division(-19,7)
    None
    >>> division(-19,0)
    None
    >>> division(-19,-7)
    None
    """
```

Nous pouvons maintenant passer à l'écriture du code.

- La postcondition nous fournit pratiquement l'invariant de la boucle que nous allons devoir écrire : $a = bq + r$.

# Exemple

```
    """
    >>> division(-19,7)
    None
    >>> division(-19,0)
    None
    >>> division(-19,-7)
    None
    """
```

Nous pouvons maintenant passer à l'écriture du code.

- La postcondition nous fournit pratiquement l'invariant de la boucle que nous allons devoir écrire : $a = bq + r$.
- $q$ et $r$ seront deux variables locales, il faut les initialiser de sorte que l'invariant soit vérifié.

# Exemple

```
    """
    >>> division(-19,7)
    None
    >>> division(-19,0)
    None
    >>> division(-19,-7)
    None
    """
```

Nous pouvons maintenant passer à l'écriture du code.

- La postcondition nous fournit pratiquement l'invariant de la boucle que nous allons devoir écrire : $a = bq + r$.
- $q$ et $r$ seront deux variables locales, il faut les initialiser de sorte que l'invariant soit vérifié. Il suffit de prendre $q = 0$ et $r = a$, la précondition nous dit alors que $0 \leqslant r$.

# Exemple

```
    """
    >>> division(-19,7)
    None
    >>> division(-19,0)
    None
    >>> division(-19,-7)
    None
    """
```

Nous pouvons maintenant passer à l'écriture du code.

- La postcondition nous fournit pratiquement l'invariant de la boucle que nous allons devoir écrire : $a = bq + r$.
- $q$ et $r$ seront deux variables locales, il faut les initialiser de sorte que l'invariant soit vérifié. Il suffit de prendre $q = 0$ et $r = a$, la précondition nous dit alors que $0 \leqslant r$.
- Si $r < b$ la division est terminée,

# Exemple

```
    """
    >>> division(-19,7)
    None
    >>> division(-19,0)
    None
    >>> division(-19,-7)
    None
    """
```

Nous pouvons maintenant passer à l'écriture du code.

- La postcondition nous fournit pratiquement l'invariant de la boucle que nous allons devoir écrire : $a = bq + r$.
- $q$ et $r$ seront deux variables locales, il faut les initialiser de sorte que l'invariant soit vérifié. Il suffit de prendre $q = 0$ et $r = a$, la précondition nous dit alors que $0 \leqslant r$.
- Si $r < b$ la division est terminée, mais si $r \geqslant b$, alors on enlève $b$ à $r$ et on ajoute 1 à $q$ car $bq + r = b(q + 1) + (r - b)$,

# Exemple

```
    """
    >>> division(-19,7)
    None
    >>> division(-19,0)
    None
    >>> division(-19,-7)
    None
    """
```

Nous pouvons maintenant passer à l'écriture du code.

- La postcondition nous fournit pratiquement l'invariant de la boucle que nous allons devoir écrire : $a = bq + r$.
- $q$ et $r$ seront deux variables locales, il faut les initialiser de sorte que l'invariant soit vérifié. Il suffit de prendre $q = 0$ et $r = a$, la précondition nous dit alors que $0 \leqslant r$.
- Si $r < b$ la division est terminée, mais si $r \geqslant b$, alors on enlève $b$ à $r$ et on ajoute 1 à $q$ car $bq + r = b(q + 1) + (r - b)$, l'invariant est bien conservé,

# Exemple

```
    """
    >>> division(-19,7)
    None
    >>> division(-19,0)
    None
    >>> division(-19,-7)
    None
    """
```

Nous pouvons maintenant passer à l'écriture du code.

- La postcondition nous fournit pratiquement l'invariant de la boucle que nous allons devoir écrire : $a = bq + r$.
- $q$ et $r$ seront deux variables locales, il faut les initialiser de sorte que l'invariant soit vérifié. Il suffit de prendre $q = 0$ et $r = a$, la précondition nous dit alors que $0 \leqslant r$.
- Si $r < b$ la division est terminée, mais si $r \geqslant b$, alors on enlève $b$ à $r$ et on ajoute 1 à $q$ car $bq + r = b(q + 1) + (r - b)$, l'invariant est bien conservé, et on recommence le test sur $r$ (boucle).

# Exemple

```
def division(a: int, b: int) -> (int, int):
    """
    Renvoie le quotient et le reste de la division de a par b
    Paramètres:
    ----------
        a: int, entier naturel
        b: int, entier strictement positif (précondition)
    Retour:
    ------
        tuple (q, r) tel que a=bq+r avec 0<=r<b
        ou None si a<0 ou b<=0   (postcondition)
    """
    if (a<0) or (b<=0):  # précondition non remplie
        return None  # la fonction se termine en renvoyant None
    q, r = 0, a
    while r >= b:
    # Invariant: a=bq+r et 0<=r
        q += 1
        r -= b  # bq+r = b(q+1)+(r-b)
    return (q, r)
```

# Effectuer les tests

On peut proposer trois façons de procéder pour exécuter les tests :

# Effectuer les tests

On peut proposer trois façons de procéder pour exécuter les tests :

1. **La méthode naïve** : on ajoute à la suite de notre fonction une succession de `print` (un par test), du style : `print(division(19,7) == (2,5))`, ce qui provoquera à l'exécution l'affichage de `True` ou bien `False` suivant que le test est positif ou négatif.

# Effectuer les tests

On peut proposer trois façons de procéder pour exécuter les tests :

2. **Un peu plus élaboré** : on écrit une fonction dédiée aux tests qui va utiliser l'instruction `assert` pour chacun des tests

# Effectuer les tests

On peut proposer trois façons de procéder pour exécuter les tests :

2. **Un peu plus élaboré** : on écrit une fonction dédiée aux tests qui va utiliser l'instruction `assert` pour chacun des tests :

```
def test_division():
    assert division(19,7) == (2,5), "erreur lorsque a=19 et b=7"
    assert division(7,19) == (0,7), "erreur lorsque a=7 et b=19"
    # ... etc
    assert division(-19,-7) == None, "erreur lorsque a=-19 et b=-7"
    print("Tous les tests ont été réussis.")
```

# Effectuer les tests

On peut proposer trois façons de procéder pour exécuter les tests :

2. **Un peu plus élaboré** : on écrit une fonction dédiée aux tests qui va utiliser l'instruction `assert` pour chacun des tests :

```
def test_division():
    assert division(19,7) == (2,5), "erreur lorsque a=19 et b=7"
    assert division(7,19) == (0,7), "erreur lorsque a=7 et b=19"
    # ... etc
    assert division(-19,-7) == None, "erreur lorsque a=-19 et b=-7"
    print("Tous les tests ont été réussis.")
```

   Après exécution, s'il n'y a pas d'erreur d'assertion, on affiche que tous les tests ont été passés avec succès.

# Effectuer les tests

On peut proposer trois façons de procéder pour exécuter les tests :

2. **Un peu plus élaboré** : on écrit une fonction dédiée aux tests qui va utiliser l'instruction `assert` pour chacun des tests :

```
def test_division():
    assert division(19,7) == (2,5), "erreur lorsque a=19 et b=7"
    assert division(7,19) == (0,7), "erreur lorsque a=7 et b=19"
    # ... etc
    assert division(-19,-7) == None, "erreur lorsque a=-19 et b=-7"
    print("Tous les tests ont été réussis.")
```

Après exécution, s'il n'y a pas d'erreur d'assertion, on affiche que tous les tests ont été passés avec succès.

Attention cependant, si un des tests provoque une boucle infinie le programme ne se terminera pas, et on ne saura pas quel est le test défectueux.

# Effectuer les tests

On peut proposer trois façons de procéder pour exécuter les tests :

3. **Tests automatiques** : tests explicités dans la docstring (sous une certaine forme) et utilisation de la fonction `testmode()` du module `doctest` :
   - Analyse de la docstring et exécution des lignes commençant par `>>>`

# Effectuer les tests

On peut proposer trois façons de procéder pour exécuter les tests :

3. **Tests automatiques** : tests explicités dans la docstring (sous une certaine forme) et utilisation de la fonction `testmode()` du module `doctest` :
   - Analyse de la docstring et exécution des lignes commençant par `>>>`
   - comparaison du résultat avec le contenu de la ligne suivante. En cas de différence, une erreur est signalée.

# Effectuer les tests

On peut proposer trois façons de procéder pour exécuter les tests :

3. **Tests automatiques** : tests explicités dans la docstring (sous une certaine forme) et utilisation de la fonction `testmode()` du module `doctest` :
   - Analyse de la docstring et exécution des lignes commençant par `>>>`
   - comparaison du résultat avec le contenu de la ligne suivante. En cas de différence, une erreur est signalée.
   - Attention : la docstring est une chaîne de caractères, il faut donc faire très attention à la façon dont on écrit les résultats attendus car ce sont des chaînes de caractères qui vont être comparées.

# Effectuer les tests

- Par exemple si on écrit dans la docstring de notre fonction.

```
def division(a: int, b: int) -> (int, int):
    """

    ...
    >>> division(19,7)
    (2,5)
    ...
    """
    # ...
```

# Effectuer les tests

- Par exemple si on écrit dans la docstring de notre fonction.

```
def division(a: int, b: int) -> (int, int):
    """

    ...
    >>> division(19,7)
    (2,5)
    ...
    """
    # ...
```

alors à l'exécution de l'instruction `doctest.testmod()` nous verrons l'erreur suivante :

```
**********************
File "val.py", line 250, in __main__.division
Failed example:
    division(19,7)
Expected:
    (2,5)
Got:
    (2, 5)
```

notez l'espace manquante après la virgule dans la docstring...

# Effectuer les tests

- De même, si on écrit :

```
def division(a: int, b: int) -> (int, int):
    """
    ...
    >>> division(19,-7)
    None
    ...
    """
```

alors à l'exécution de l'instruction `doctest.testmod()` nous verrons l'erreur suivante :

```
**********************
File "val.py", line 264, in __main__.division
Failed example:
    division(-19,-7)
Expected:
    None
Got nothing
```

car `None` et `"None"` ce n'est pas la même chose !

# Effectuer les tests

- De même, si on écrit :

```
def division(a: int, b: int) -> (int, int):
    """
    ...
    >>> division(19,-7)
    None
    ...
    """
```

alors à l'exécution de l'instruction `doctest.testmod()` nous verrons l'erreur suivante :

```
**********************
File "val.py", line 264, in __main__.division
Failed example:
    division(-19,-7)
Expected:
    None
Got nothing
```

car `None` et `"None"` ce n'est pas la même chose !

# Effectuer les tests

Il est préférable d'opter pour l'écriture suivante :

```
def division(a: int, b: int) -> (int, int):
    """
    ...
    >>> division(19,-7) == None
    True
    ...
    """
    # ...
```

# Effectuer les tests

Il est préférable d'opter pour l'écriture suivante :

```
def division(a: int, b: int) -> (int, int):
    """
    ...
    >>> division(19,-7) == None
    True
    ...
    """
    # ...
```

alors à l'exécution de l'instruction `doctest.testmod()` il n'y aura plus d'erreur (à condition d'écrire `True` correctement, et sans espace avant ni après !).

# Effectuer les tests

Pour conclure, nous pouvons proposer ce code pour tester notre fonction :

```
def division(a: int, b: int) -> (int, int):
    """
    Renvoie le quotient et le reste de la division de a par b
    Paramètres:
    ----------
        a: int, entier natuel
        b: int, entier strictement positif
    Retour:
    ------
        tuple (q, r) tel que a=bq+r avec 0<=r<b
        ou None si a<0 ou b<=0
    Exemples:
    --------
    >>> division(19,7) == (2,5)
    True
    >>> division(7,19) == (0,7)
    True
    >>> division(0,19) == (0,0)
    True
    >>> division(19,0) == None
    True
```

# Effectuer les tests

Suite :

```
    """
    >>> division(19,-7) == None
    True
    >>> division(-19,7) == None
    True
    >>> division(-19,0) == None
    True
    >>> division(-19,-7) == None
    True
    """
    if (a<0) or (b<=0):  # pré-condition non remplie
        return None
    q, r = 0, a
    while r >= b:
    # Invariant: a=bq+r et 0<=r
        q += 1
        r -= b  # bq+r = b(q+1)+(r-b)
    return (q, r)
```

# Tests de performance

- Pour comparer des algorithmes, on peut être amené à effectuer des tests de performance en temps d'exécution.

# Tests de performance

- Pour comparer des algorithmes, on peut être amené à effectuer des tests de performance en temps d'exécution.
- Avec le module `time`, il est possible de faire ces mesures. On relève un instant initial, on exécute un certain nombre de fois la fonction, on relève l'instant final et il n'y a plus qu'à faire la différence.

# Tests de performance

- Pour comparer des algorithmes, on peut être amené à effectuer des tests de performance en temps d'exécution.
- Avec le module **`time`**, il est possible de faire ces mesures. On relève un instant initial, on exécute un certain nombre de fois la fonction, on relève l'instant final et il n'y a plus qu'à faire la différence.
- Exemple :

```
from time import time  # fonction time du module time
t1 = time()  # instant initial
for _ in range(1000):  # pour 1000 exécutions par exemple
    r = fonction_a_tester()
t2 = time()  # instant final
print("durée: ", (t2-t1)/1000)  # en secondes
```

# Tests de performance

- Pour comparer des algorithmes, on peut être amené à effectuer des tests de performance en temps d'exécution.
- Avec le module **`time`**, il est possible de faire ces mesures. On relève un instant initial, on exécute un certain nombre de fois la fonction, on relève l'instant final et il n'y a plus qu'à faire la différence.
- Exemple :

```
from time import time  # fonction time du module time
t1 = time()  # instant initial
for _ in range(1000):  # pour 1000 exécutions par exemple
    r = fonction_a_tester()
t2 = time()  # instant final
print("durée: ", (t2-t1)/1000)  # en secondes
```

- Suivant le système d'exploitation, une seule exécution n'est pas forcément suffisante pour avoir une mesure fiable.

# Tests de performance

- Pour comparer des algorithmes, on peut être amené à effectuer des tests de performance en temps d'exécution.
- Avec le module **`time`**, il est possible de faire ces mesures. On relève un instant initial, on exécute un certain nombre de fois la fonction, on relève l'instant final et il n'y a plus qu'à faire la différence.
- Exemple :

```
from time import time  # fonction time du module time
t1 = time()  # instant initial
for _ in range(1000):  # pour 1000 exécutions par exemple
    r = fonction_a_tester()
t2 = time()  # instant final
print("durée: ", (t2-t1)/1000)  # en secondes
```

- Suivant le système d'exploitation, une seule exécution n'est pas forcément suffisante pour avoir une mesure fiable.
- Si on travaille dans un notebook, alors on peut plus simplement utiliser l'instruction `timeit fonction_a_tester()` qui va mesurer automatiquement le temps d'exécution de la fonction.