Chapitre ITC1

Les fonctions

Section 1

Introduction : de quoi est constitué un programme ?

Instructions et expressions

Les instructions

- ce sont des briques de base du programme
- elles exécutent une action mais ne renvoient aucune valeur
- exemples : `=` (affectation), `import`, `if`, `for`, `while`, `def`,...

Les expressions

- ce sont des ensembles de commandes qui renvoient une valeur
- elles utilisent les variables et des opérateurs : `+`, `-`, ...
- exemples : `1+2`, `a+3`, `a/b+(c-d)*2`,...

Instructions et expressions

Exemple

```
1 a=2
2 b=3
3 c=a+b
4 if c>8:
5     a=a-1
```

Instructions et expressions

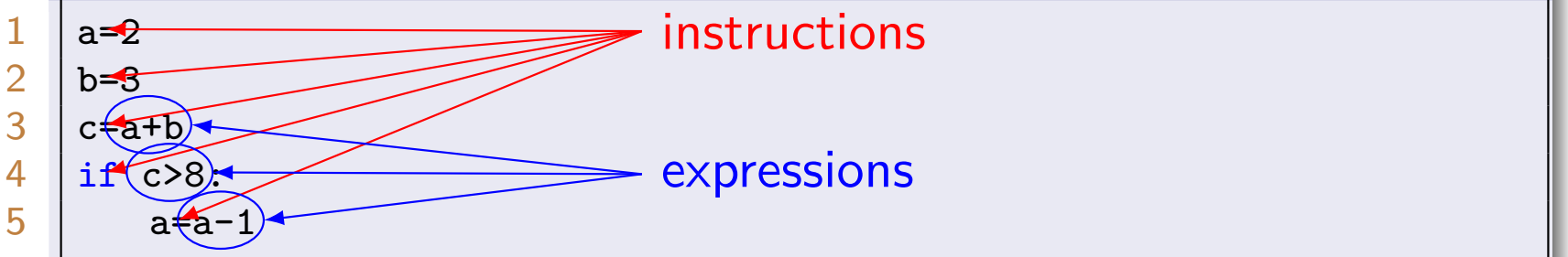
Exemple

```
1 a=2
2 b=3
3 c=a+b
4 if c>8:
5     a=a-1
```

Instructions et expressions

Exemple

```
1  a=2
2  b=3
3  c=a+b
4  if c>8:
5      a=a-1
```



Fonctions et procédures

On veut parfois regrouper des instructions ou des expressions...

... on crée alors des :

- **fonctions** : ensemble d'instructions/expressions qui renvoient une valeur
- **procédures** : ensemble d'instructions/expressions qui exécutent une action sans renvoyer de valeur

En Python, il y en a déjà de toutes prêtes :

- `print` est une procédure qui affiche à l'écran
- `input` est une fonction qui renvoie une valeur

On repère les fonctions et les procédures par les parenthèses `()` qui suivent leur nom.

Fonctions et procédures

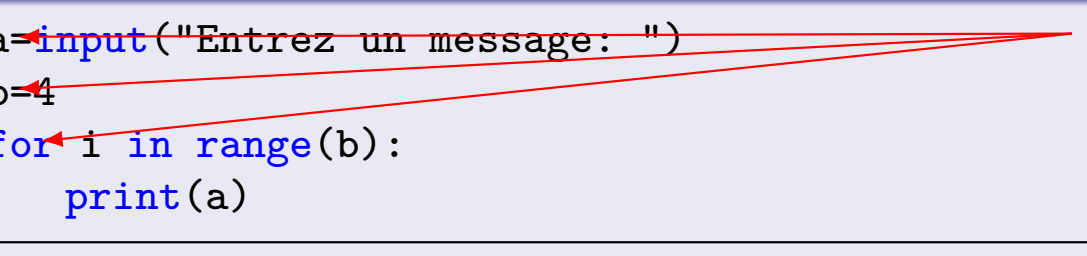
Exemple

```
1 a=input("Entrez un message: ")
2 b=4
3 for i in range(b):
4     print(a)
```

Fonctions et procédures

Exemple

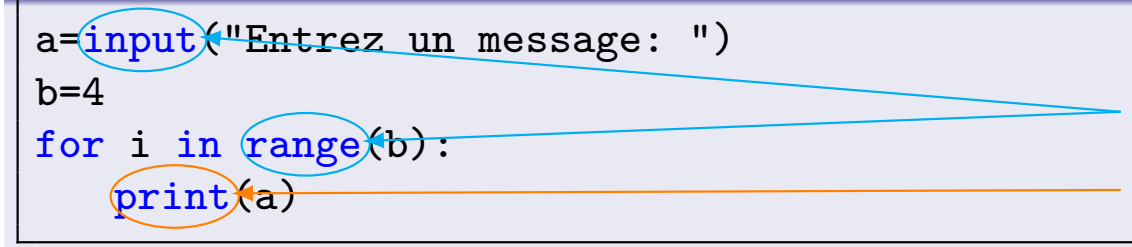
```
1  a=input("Entrez un message: ")
2  b=4
3  for i in range(b):
4      print(a)
```



Fonctions et procédures

Exemple

```
1 a=input("Entrez un message: ")
2 b=4
3 for i in range(b):
4     print(a)
```

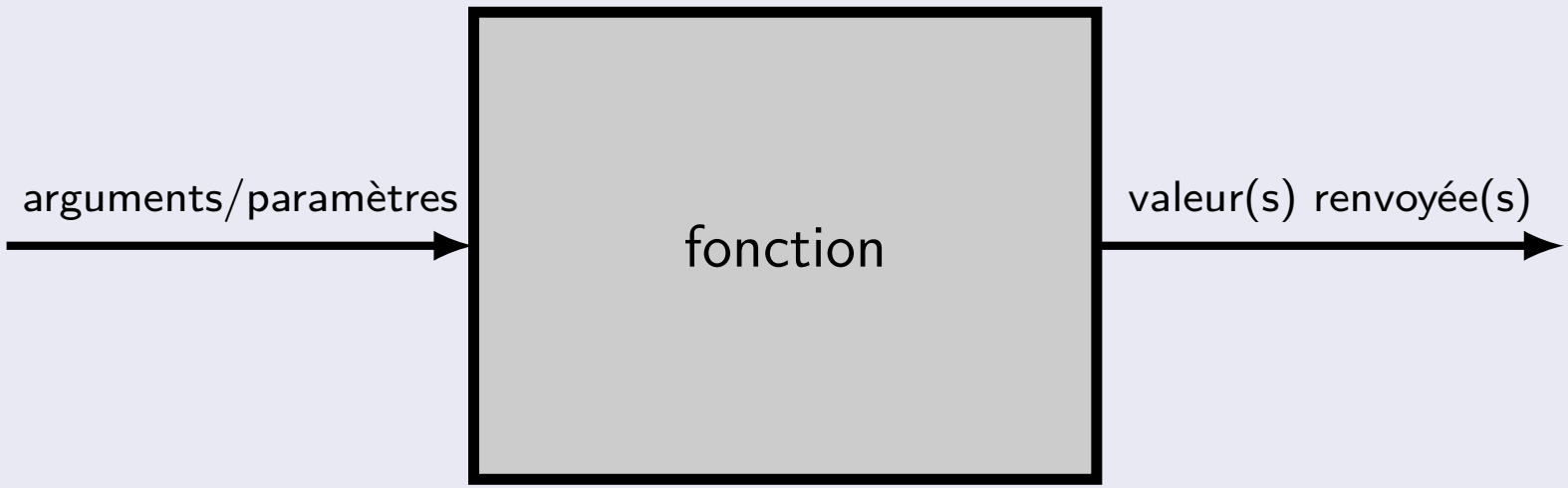


Section 2

Définition d'une fonction

Qu'est-ce qu'une fonction ?

C'est une «boîte noire» qui prend des arguments (ou paramètres) en renvoie une ou des valeurs



Le prototype d'une fonction

Le prototype (ou signature) d'une fonction est sa notice d'utilisation

On y indique :

- son nom et ce qu'elle fait
- les arguments qu'elle prend en entrée, avec leurs types et leurs contraintes
- la/les valeur(s) de retour (sortie) avec son type

Exemple

`moyenne(x:float, y:float)->float` : calcule la moyenne arithmétique de deux réels.

La déclaration d'une fonction

La définition d'une fonction de fait avec le mot-clé `def`

```
1 # prototype
2 def fonction(arguments):
3     """ commentaire éventuel """
4     instructions
5     return valeur
```

Exemple

```
1 # moyenne(x:float, y:float)->float: calcule la moyenne arithmétique de c
2 def moyenne(x,y):
3     m=(x+y)/2
4     return m
```

On peut indiquer aussi le type des arguments attendus dans la définition, mais cela n'a aucun caractère obligatoire :

```
1 def moyenne(x: float,y: float) -> float:
2     """ calcule la moyenne arithmétique de deux réels """
3     ...
```

L'appel d'une fonction

Une fonction s'exécute quand on l'appelle

L'appel de la fonction se fait en écrivant son nom suivi de ses arguments entre parenthèse : `moyenne(2,4)`

On peut mettre des variables comme argument ; le nom des variables n'a rien à voir avec celui des paramètres

Par exemple :

```
1 def moyenne(x:float,y:float)->float:
2     m=(x+y)/2
3     return m
4 x=3
5 print(moyenne(1,x))
```

fonctionne : la fonction recevra un argument `x` qui vaudra 1 et un argument `y` qui vaudra 3.

Fonctionnement de l'appel d'une fonction

Les variables créées dans une fonction sont stockées dans un espace à part (cf. section 4)

Exécutons ligne par ligne le code suivant :

```
1 def moyenne(x:float,y:float)->float:
2     m=(x+y)/2
3     return m
4 n=3
5 print(moyenne(1,n))
```

mémoire globale

Au départ, il n'y a rien en mémoire. Le programme commence à la ligne 1

Fonctionnement de l'appel d'une fonction

Les variables créées dans une fonction sont stockées dans un espace à part (cf. section 4)

Exécutons ligne par ligne le code suivant :

```
1 def moyenne(x:float,y:float)->float:
2     m=(x+y)/2
3     return m
4 n=3
5 print(moyenne(1,n))
```

mémoire globale	
moyenne	func

Le programme lit la ligne 1 : il y voit une définition de fonction, qu'il stocke en mémoire pour plus tard.

Fonctionnement de l'appel d'une fonction

Les variables créées dans une fonction sont stockées dans un espace à part (cf. section 4)

Exécutons ligne par ligne le code suivant :

```
1 def moyenne(x:float,y:float)->float:
2     m=(x+y)/2
3     return m
4 n=3
5 print(moyenne(1,n))
```

mémoire globale	
moyenne	func
n	3

Il saute à la ligne 4 ; il y lit une affectation et crée une case mémoire de nom n et de valeur 3.

Fonctionnement de l'appel d'une fonction

Les variables créées dans une fonction sont stockées dans un espace à part (cf. section 4)

Exécutons ligne par ligne le code suivant :

```
1 def moyenne(x:float,y:float)->float:
2     m=(x+y)/2
3     return m
4 n=3
5 print(moyenne(1,n))
```

mémoire globale	
moyenne	func
n	3

mémoire de la fonction	
x	1
y	3

Ligne 5 : appels imbriqués. On commence par l'appel le plus intérieur, comme en maths. Le programme se prépare donc à exécuter la fonction, et crée un espace mémoire réservé à la fonction dans lequel il stocke les valeurs données aux arguments, à savoir 1 et 3.

Fonctionnement de l'appel d'une fonction

Les variables créées dans une fonction sont stockées dans un espace à part (cf. section 4)

Exécutons ligne par ligne le code suivant :

```
1 def moyenne(x:float,y:float)->float:
2     m=(x+y)/2
3     return m
4 n=3
5 print(moyenne(1,n))
```

mémoire globale	
moyenne	func
n	3

mémoire de la fonction	
x	1
y	3
m	2.0

Ligne 2, le programme calcule (x+y)/2 et affecte le résultat dans une case mémoire m.

Fonctionnement de l'appel d'une fonction

Les variables créées dans une fonction sont stockées dans un espace à part (cf. section 4)

Exécutons ligne par ligne le code suivant :

```
1 def moyenne(x:float,y:float)->float:
2     m=(x+y)/2
3     return m
4 n=3
5 print(moyenne(1,n))
```

mémoire globale	
moyenne	func
n	3

mémoire de la fonction	
x	1
y	3
m	2.0

Ligne 3, le programme renvoie la valeur contenue dans m ; la mémoire locale de la fonction est effacée.

Fonctionnement de l'appel d'une fonction

Les variables créées dans une fonction sont stockées dans un espace à part (cf. section 4)

Exécutons ligne par ligne le code suivant :

```
1 def moyenne(x:float,y:float)->float:
2     m=(x+y)/2
3     return m
4 n=3
5 print(moyenne(1,n))
```

2.0

mémoire globale	
moyenne	func
n	3

Le programme reprend à la ligne 5, l'appel à la fonction est remplacé par le résultat renvoyé, et le programme affiche 2.0.

Récupération de(s) valeurs(s) renvoyée(s)

La valeur renvoyée doit être affectée à une variable si on veut la stocker :

```
1 m=moyenne(1,2)
2 print(m)
```

Si la fonction renvoie une liste de valeurs, on peut les stocker dans une liste de variables :

```
1 # encadre(x:float)->[float]: encadre le réel x entre deux entiers consé
2 def encadre(x:float)->[float]:
3     import math
4     n=math.floor(x)
5     return [n,n+1]
6 i,j=encadre(1.5)
7 print(i) # affiche 1
```

Les procédures

Une procédure est une fonction qui ne renvoie rien

mais qui accomplit une action, par exemple un affichage.

```
1 # affiche_entiers(n:int): affiche les entiers de 1 à n
2 def affiche_entiers(n:int):
3     for i in range(1,n+1):
4         print(i)
```

Section 3

Vérification des arguments

Les mauvais arguments, source d'erreur

En Python, le type des arguments n'est pas précisé

C'est un des problèmes de Python : on peut passer n'importe quoi comme argument, et tenter, par exemple, de faire la moyenne entre un tableau et une chaîne de caractères.

Pour éviter cela, on place au début des fonctions des **assertions** qui testent les arguments et, s'ils ne correspondent pas à ce qu'on attend, provoquent un message d'erreur.

L'instruction assert

C'est une instruction, pas une fonction : JAMAIS de parenthèses

```
assert condition, message d'erreur
```

Exemples d'assertion sur un paramètre `x` :

```
assert x>0 , "l'argument doit être strictement positif"
assert x!=0 , "l'argument doit être non nul"
assert isinstance(x,int) , "l'argument doit être entier"
assert isinstance(x,(int,float)) , "l'argument doit être réel"
```

etc...

Section 4

Variables locales et globales

Variables globales et locales

Une variable définie en-dehors d'une fonction est **globale**

Elle est alors utilisable dans **tout le programme**

Une variable définie dans d'une fonction est **locale**

Elle n'est alors utilisable que dans **la fonction** : une fois la fonction terminée, elle est effacée de la mémoire.

Exemple d'erreur classique

```
1 def plus_un(x:float)->float:
2     y=x+1
3     return y
4 plus_un(3)
5 print(y) # erreur, y n'existe plus
```

Variables globales et locales

Une variable définie en-dehors d'une fonction est **globale**

Elle est alors utilisable dans **tout le programme**

Une variable définie dans d'une fonction est **locale**

Elle n'est alors utilisable que dans **la fonction** : une fois la fonction terminée, elle est effacée de la mémoire.

Pour récupérer une valeur d'une fonction, il faut la renvoyer et la stocker dans une autre variable

```
1 def plus_un(x:float)->float:
2     y=x+1
3     return y
4 z=plus_un(3)
5 print(z) # ok
```

Variables locales

Les variables locales sont de deux origines :

- les arguments sont définis comme des variables locales
- toutes les variables définies dans la fonction

Dans cette fonction, on a `x`, `y` et `m` :

```
1 def moyenne(x:float,y:float)->float:
2     m=(x+y)/2
3     return m
```

Variables globales dans une fonction

Dans une fonction, toute variable globale est utilisable :

```
1 g=9.8 # accélération de la pesanteur
2 def poids(m:float)->float:
3     return m*g
4 print(poids(1.4)) # affiche 13.72
```

Mais si une variable locale a le même nom qu'une variable globale, la variable globale est masquée :

```
1 g=9.8 # accélération de la pesanteur
2 def poids(m:float)->float:
3     g=9.83 # on est au pôle nord
4     return m*g
5 print(poids(1.4)) # affiche 13.762
6 print(g) # affiche 9.8
```

Variables globales dans une fonction

Pour être modifiée dans une fonction, une variable globale doit être déclarée avec le mot-clé `global` :

```
1 g=9.8 # accélération de la pesanteur
2 def poids(m:float)->float:
3     global g
4     g=9.83 # on est au pôle nord
5     return m*g
6 print(poids(1.4)) # affiche 13.762
7 print(g) # affiche 9.83
```

Variables globales=danger

Il faut éviter d'utiliser une variable globale dans une fonction

- sauf pour les valeurs constantes (par exemple $G = 6.67e-11$ dans un pb de physique, ou les autres données du problème)
- si on a besoin vraiment de l'utiliser, c'est mieux de la passer comme argument, même si c'est plus lourd

Section 5

Effets de bord

Variables mutables et non mutables

Variable mutable/non mutable

L'instruction `x=...` crée une case en mémoire appelée x et contenant une valeur.
Que se passe-t-il si on modifie la valeur de x avec une nouvelle instruction `x=...` ? En Python,

- si la variable est de type *mutable*, la case mémoire est modifiée : c'est le cas des listes et des dictionnaires
- sinon, la variable est *non-mutable* : une nouvelle case mémoire est créée avec la nouvelle valeur ; c'est le cas des entiers, des réels, des chaînes de caractères, des booléens, des tuples (listes non-mutables)

Variables mutables et non mutables

Copie de variables

Quand on copie une variable dans une autre avec une instruction du type `x=y`, alors, en Python,

- si la variable y est de type non-mutable, alors une nouvelle case mémoire x est créée avec la valeur de y
- si la variable y est de type mutable, alors y et x correspondent à la même case mémoire

Exemple 1

```
1 x=3
2 y=x
3 y=y+1
4 print(x,y)
```

Ce programme affiche 3 et 4

Variables mutables et non mutables

Copie de variables

Quand on copie une variable dans une autre avec une instruction du type `x=y`, alors, en Python,

- si la variable y est de type non-mutable, alors une nouvelle case mémoire x est créée avec la valeur de y
- si la variable y est de type mutable, alors y et x correspondent à la même case mémoire

Exemple 2

```
1 x="Bonjour"
2 y=x
3 y=y+" tout le monde"
4 print(x,y)
```

Ce programme affiche Bonjour et Bonjour tout le monde

Variables mutables et non mutables

Copie de variables

Quand on copie une variable dans une autre avec une instruction du type `x=y`, alors, en Python,

- si la variable y est de type non-mutable, alors une nouvelle case mémoire x est créée avec la valeur de y
- si la variable y est de type mutable, alors y et x correspondent à la même case mémoire

Exemple 3

```
1 x=[1,2,3]
2 y=x
3 y[0]=4
4 print(x,y)
```

Ce programme affiche [4,2,3] et [4,2,3]

Variables mutables et non mutables

Copie de variables

Quand on copie une variable dans une autre avec une instruction du type `x=y`, alors, en Python,

- si la variable y est de type non-mutable, alors une nouvelle case mémoire x est créée avec la valeur de y
- si la variable y est de type mutable, alors y et x correspondent à la même case mémoire

Exemple 4

```
1 x=[1,2,3]
2 y=x.copy()
3 y[0]=4
4 print(x,y)
```

Ce programme affiche [1,2,3] et [4,2,3]

Variables mutables et non mutables

Copie de variables

Quand on copie une variable dans une autre avec une instruction du type `x=y`, alors, en Python,

- si la variable y est de type non-mutable, alors une nouvelle case mémoire x est créée avec la valeur de y
- si la variable y est de type mutable, alors y et x correspondent à la même case mémoire

Exemple 5

```
1 x=[1,2,3]
2 y=x+[6,7]
3 y[0]=4
4 print(x,y)
```

Ce programme affiche [1,2,3] et [4,2,3,6,7]

Passage d'arguments par valeur

En Python, les variables mutables sont passées par copie

Cela signifie que la fonction reçoit comme argument une copie de la variable initiale : elle ignore la case mémoire d'où provient cette valeur, donc elle ne peut pas la modifier :

```
1 def incremente(y:float)->float:
2     """ renvoie l'argument augmenté de 1 """
3     y=y+1
4     return y
5 x=3
6 print(incremente(x)) # affiche 4
7 print(x) # affiche 3
```

La fonction reçoit un 3 mais ignore qu'il provient de x. Elle le stocke dans une variable locale y, lui ajoute 1 et la renvoie ; la variable y est alors détruite.

Passage d'arguments par référence

En Python, les objets mutables comme les listes sont passés par référence

Cela signifie que la fonction reçoit l'adresse en mémoire des données ; si elle modifie ces données, cela modifie la variable d'origine :

```
1 def fonction_inutile(liste:[float])->[float]:
2     """ ajoute 1 au premier terme de la liste et renvoie ce terme """
3     liste[0]=liste[0]+1
4     return liste[0]
5 L=[1,3,4,2,5]
6 print(fonction_inutile(L)) # affiche 2
7 print(L) # affiche [2,3,4,2,5]
```

La liste de départ a été modifiée : on dit qu'il y a eu un **effet de bord**.

L'effet de bord peut être voulu ou pas

Il faut le préciser dans la documentation

```
1  def valeur_absolue(L:[float])->[float]:
2      """ renvoie la liste des valeurs absolues des termes de la liste ini
3      L2=[]
4      for i in range(len(L)):
5          if L[i]<0:
6              L2.append(-L[i])
7          else:
8              L2.append(L[i])
9      return L2
10 liste=[1,2,4,3,-2,-7]
11 print(valeur_absolue(liste)) # pas d'effet de bord
12 print(liste) # affiche [1,2,4,3,-2,-7]
```

L'effet de bord peut être voulu ou pas

Il faut le préciser dans la documentation

```
1 def valeur_absolue(L:[float])->[float]:
2     """ remplace chaque terme de la liste par sa valeur absolue """
3     for i in range(len(L)):
4         if L[i]<0:
5             L[i]=-L[i]
6     return L
7 liste=[1,2,4,3,-2,-7]
8 print(valeur_absolue(liste)) # effet de bord
9 print(liste) # affiche [1,2,4,3,2,7]
```

L'effet de bord peut être voulu ou pas

Il faut le préciser dans la documentation

```
1  def valeur_absolue(L:[float])->[float]:
2      """ renvoie la liste des valeurs absolues des termes de la liste ini
3      L2=L.copy() # crée une copie de la liste de départ
4      for i in range(len(L2)):
5          if L2[i]<0:
6              L2[i]=-L2[i]
7      return L2
8  liste=[1,2,4,3,-2,-7]
9  print(valeur_absolue(liste)) # pas d'effet de bord
10 print(liste) # affiche [1,2,4,3,-2,-7]
```

Section 6

Tests

Les tests de fonctions

Un jeu de tests sert à vérifier qu'une fonction fait bien ce qu'on lui demande

- on vérifie que la fonction renvoie la bonne valeurs pour certains arguments choisis
- les arguments choisis doivent tester le plus possible les cas particuliers
- si un seul test échoue, la fonction est mauvaise
- l'idéal est d'écrire le test AVANT d'écrire la fonction

Exemple :

```
1 # carre(x:float)->float: renvoie le carré de x
2 def teste_carre():
3     if carre(2.0)!=4.0:
4         return False
5     return True
6 def carre(x:float)->float:
7     # à écrire ...
```

Jeu de tests complet

S'il y a des conditions dans la fonction, il faut les tester

```
1  # valeur_absolue(x:float)->float: renvoie la valeur absolue de x
2  def teste_valeur_absolue():
3      if valeur_absolue(2.0)!=2.0:
4          return False
5      if valeur_absolue(-2.0)!=2.0:
6          return False
7      return True
8  def valeur_absolue(x:float)->float:
9      if x<0:
10         return -x
11     else:
12         return x
```

Jeu de tests complets

Parfois, il faut imaginer où on pourrait faire des erreurs

Avec l'expérience...Par exemple, pour une fonction cherchant le minimum d'une liste, les erreurs classiques sont :

- de ne pas parcourir la liste jusqu'au bout
- d'initialiser le minimum à 0

```
1  # minimum(L:[float])->float: renvoie le minimum de la liste
2  def teste_minimum():
3      if minimum([1,-3,2])!=-3: # liste quelconque
4          return False
5      if minimum([1,2,4,3])!=1: # valeurs toutes positives
6          return False
7      if minimum([1,4,5,0])!=0: # minimum à la fin
8          return False
9      return True
```