INF1 : Algorithmique et Programmation

Cours 6 :
Fonctions et Procédures

Domitile Lourdeaux

Université de technologie de Compiègne

Printemps 2024

Motivations (1)

Pourquoi aurait-on besoin de fonctions

- Par exemple, On dispose de données sur un ensemble d'individus : taille, poids, âge, etc.
- Comment calculer :
  - la taille moyenne,
  - le poids moyen,
  - l'âge moyen ?

Motivations (2)

Première solution

- Calcul de la somme des tailles
- Division du résultat par le nombre d'individus
- Calcul de la somme des poids
- Division du résultat par le nombre d'individus
- Calcul de la somme des âges
- Division du résultat par le nombre d'individus

Motivations (3)

Première solution

```
tailles = [171, 165, 187]
poids = [62, 53, 85]
ages = [24, 21, 35]
```

```
somme = 0
n = len(tailles)
for i in range(n):
    somme = somme + tailles[i]
moyenne_tailles = somme / n
```

```
somme = 0
n = len(poids)
for i in range(n):
    somme = somme + poids[i]
moyenne_poids = somme / n
```

```
somme = 0
n = len(ages)
for i in range(n):
    somme = somme + ages[i]
moyenne_ages = somme / n
```

Portions de code analogues

```
print(f'Taille moyenne : {moyenne_tailles:6.2f} \
        Poids moyen : {moyenne_poids:6.2f} \
        Age moyen : {moyenne_ages:6.2f}')
```

Exécution :

Taille moyenne : 174.33 Poids moyen : 66.67 Age moyen : 26.67

Motivations (4)

Meilleure solution ?

- Ecrire une fonction permettant de calculer la moyenne de n nombres
- Passer les valeurs des tailles, poids et âges en paramètres

Objectifs

Eviter la répétition d'instructions

- correspondant à des traitements analogues (Cf. exemple de la moyenne)
- exécutées plusieurs fois

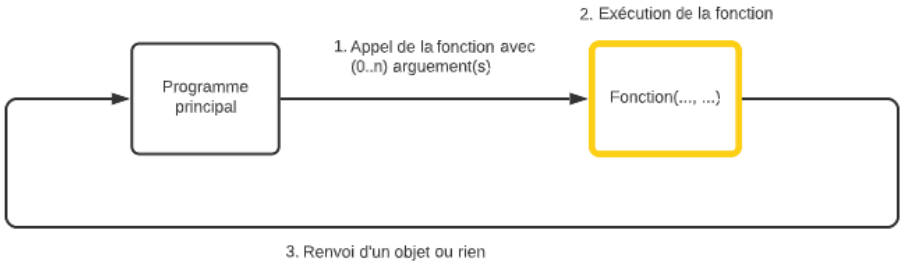
Structurer les programmes

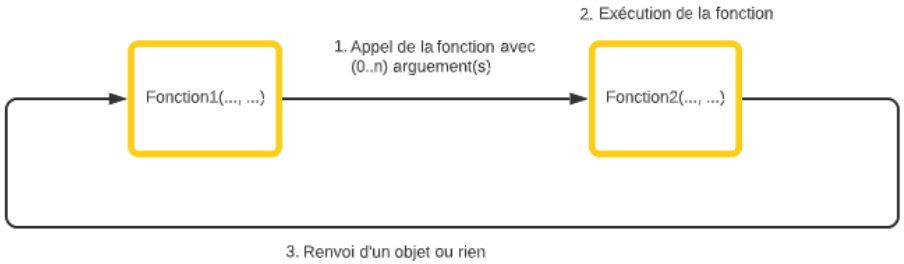
- rendant le code plus lisible en le fractionnant en blocs logiques
- exemples :
  - lecture
  - calcul
  - affichage

Fonctionnement général (1)

Boite noire

- à laquelle on peut passer aucune, une ou plusieurs variable(s) appelés arguments
- qui effectue une action
- qui renvoie un objet Python ou rien du tout.





Fonctionnement général (2)

Une fonction effectue une tâche

- L'algorithme utilisé au sein de la fonction n'intéresse pas directement l'utilisateur
- Par exemple, il est inutile de savoir comment la fonction len(l) calcule la longueur de la liste l
- On a juste besoin de savoir qu'il faut lui passer en argument une liste et qu'elle renvoie la longueur de cette liste
- Ce qui se passe à l'intérieur de la fonction ne regarde que le programmeur

source `https://python.sdv.univ-paris-diderot.fr/09_fonctions/`

Fonctionnement général (3)

Chaque fonction effectue en général une tâche unique et précise

- Si cela se complique, il est plus judicieux d'écrire plusieurs fonctions (qui peuvent éventuellement s'appeler les unes les autres)
- Cette modularité améliore la qualité générale et la lisibilité du code

source `https://python.sdv.univ-paris-diderot.fr/09_fonctions/`

Fonctions et procédures : différence

Fonction

Une fonction permet de définir un traitement autonome

- nommé par un identificateur
- appelable par cet identificateur

Une fonction retourne généralement une valeur

Procédure

Lorsqu'une fonction ne retourne pas de valeur, on parle parfois de procédure

En Python

Seule la notion de fonction existe

Fonctions et procédures : exemple

Algorithm 1 Moyenne

```
function moyenne (tab : tableau de réels) : réel
    somme ← 0
    pour chaque element e ∈ tab faire
    |   somme ← somme + e
    fin
    moyenne ← somme/longueur(tab)
    retourner moyenne

début
    lecture ou initialisation des tableaux : tab_tailles, tab_poids
    moyenne_taille ← moyenne(tab_tailles)
    moyenne_poids ← moyenne(tab_poids)
fin
```

Fonctions et méthodes

Méthode

Une méthode est une fonction qui agit sur un objet auquel elle est attachée par un point)

Exemple

- **ma_liste.append("toto")**
- la méthode append() ici :
  - passe "toto" en argument
  - ajoute la chaine "toto" à l'objet **ma_liste**
  - ne renvoie rien

Fonctions, procédures et méthodes : exemples en Python

```python
# Calcul du carre
def carre(x):
    return x**2

# somme des carres des 10 premiers entiers
s = 0
for i in range (1, 11) :
    s += carre(i)
print(s)

#affichage du carre d'un nombre
def  afficheCarre(x) :
    print(f"Carre de {x} : {x**2}")
    # print(f"Carre de {x} : {carre(x)}")

# Affichage des carres des 10 premiers entiers
for i in range (1, 11) :
    afficheCarre(i)

# Cree la liste des carres des 10 premiers entiers
s = []
for i in range (1, 11) :
    s.append(carre(i))
print(s)
```

```
385
Carre de 1 : 1
Carre de 2 : 4
Carre de 3 : 9
Carre de 4 : 16
Carre de 5 : 25
Carre de 6 : 36
Carre de 7 : 49
Carre de 8 : 64
Carre de 9 : 81
Carre de 10 : 100
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Fonctions déjà rencontrées

Fonctions et procédures

- Exemples : print(), input(), len(), range(), chr(), ord(), ...

Méthodes

- Exemples : ch.upper(), tab.append()...

Définition en Python (1)

Le mot-clé def permet de définir une fonction

Il est suivi d'un nom de fonction, d'une liste de paramètres formels entre parenthèses et du caractère :

Remarque

la syntaxe de def utilise : comme les boucles for et while ainsi que les tests if, un bloc d'instructions est donc attendu comme dans les boucles et les tests

Indentation

- les instructions qui définissent la fonction doivent être indentées

Syntaxe

```
def nomFonction(liste de parametres):
    <bloc d instructions>
```

Définition en Python (2)

Valeur retournée (ou renvoyée)

- pour retourner explicitement une valeur, on utilise l'instruction return
- si aucune valeur n'est retournée explicitement, la fonction retourne None
- la valeur retournée peut être récupérée dans une variable

```
#Ecrire une fonction retournant le cube d'un nombre x
def cube(x) :
    return x**3

# Quelques exemples d'utilisation
y = cube(3)

print(cube(3*t + 1))

volume = 4/3 * Pi * cube(r)
```

Exemple : Moyenne

```python
# Fonction Moyenne
def moyenne (tab) :
    somme = 0
    for e in tab :
        somme += e
    moy = somme / len(tab)
    return moy

tailles = [171, 165, 187]
poids = [62, 53, 85]
ages = [24, 21, 35]

moyenne_tailles = moyenne(tailles)
moyenne_poids = moyenne(poids)
moyenne_ages = moyenne(ages)
print(f'Taille moyenne : {moyenne_tailles:6.2f} \
        Poids moyen : {moyenne_poids:6.2f} \
        Age moyen : {moyenne_ages:6.2f} ')
```

print et return

Attention à ne pas confondre print et return

- Exemple avec print

```
>>> def cube(x):
...     print(x**3)    # dans ce cas la valeur retournée est None
...
>>> x = 2
>>> y = 3 * cube(x) + 1
8
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'
```

- Exemple avec return

```
>>> def cube(x):
...     return x**3
...
>>> x = 2
>>> y = 3 * cube(x) + 1
>>> y
25
```

Particularités : Renvoi d'objets

Renvoi d'objets

- En Python, les fonctions sont capables de renvoyer plusieurs objets à la fois
- On peut faire une affectation multiple

```
>>> def carre_cube(x):
...     return x**2, x**3
...
>>> carre_cube(2)
(4, 8)
>>> a, b = carre_cube(2)
>>> a
4
>>> b
8
>>> z = carre_cube(2)
>>> z
(4, 8)
```

Source : `https://python.sdv.univ-paris-diderot.fr/09_fonctions/`

Paramètres (1)

Une fonction peut ne contenir aucun paramètre

```
>>> def alphabet() :
...     debut = ord('A')
...     for i in range (26) :
...         print(chr(debut+i), end='')
...
>>> alphabet()
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Paramètres (2)

Une fonction peut contenir un ou plusieurs paramètres

- Une liste de paramètres est constituée d'un ou plusieurs noms entre parenthèses
- Chaque nom correspond à un paramètre. Les paramètres sont séparés par des virgules
- Les paramètres d'une fonction sont parfois appelés paramètres formels
- Un paramètre se comporte comme une variable pour la fonction
- En python, le type de chaque paramètre est déterminé dynamiquement lors de l'appel de la fonction

Paramètres (3)

Une fonction qui retourne une valeur s'utilise au niveau d'une expression

- Exemples :
  - $m = moyenne(t)$
  - $z = 3 * cube(x) + 1$

Une fonction qui ne retourne pas de valeur (procédure) s'utilise au niveau d'une instruction

- Exemples :
  - $alphabet()$
  - $print(x)$

Paramètres et arguments

Paramètres

Les paramètres interviennent dans la définition d'une fonction

Arguments

Les arguments sont utilisés lors de l'appel de la fonction

```
>>> def affiche(message): # paramètre : message
...     print(message)
... def cube(x):          # paramètre x
...     return x**3
```

```
>>> affiche('Bonjour à tous') # argument :'Bonjour à tous'
... y = 3
... print(cube(y))   # argument : y
...
...
Bonjour à tous
27
```

Remarques

- on aurait pu écrire x à la place de y
- Dans ce cas, l'argument x n'a rien à voir avec le paramètre x. Seule la valeur de l'argument x est transmise au paramètre x

Arguments positionnels

Argument positionnel

- Lorsqu'on définit une fonction :

  def fct(x, y) :

  les arguments x et y sont appelés **arguments positionnels**
- Il est obligatoire de les **préciser** lors de l'appel de la fonction
- Il est nécessaire de respecter le **même ordre** lors de l'appel que dans la définition de la fonction

Ici :

- 2 correspondra à x
- 3 correspondra à y

```
>>> def plus (x, y) :
...     return x + y
...
>>> plus(2, 3)
5
```

Nombre d'arguments

Nombre d'arguments

- Erreur si le nombre d'arguments ne correspond pas au nombre de paramètres requis

```
>>> def plus (x, y) :
...     return x + y
...
>>> plus(2, 3)
5
>>> plus(2)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: plus() missing 1 required positional argument: 'y'
```

Arguments facultatifs (1)

Valeur par défaut

- Il est possible (et souvent souhaitable) de définir un **argument par défaut** pour un, plusieurs ou tous les paramètres
- On obtient une fonction qui peut être appelée avec une **partie des arguments attendus**

```
>>> def politesse(nom, vedette = 'Madame') :
...     print("Veuillez agréer,", vedette, nom, ", mes sincères salutations" )
...
>>> politesse("Toto")
Veuillez agréer, Madame Toto , mes sincères salutations
>>> politesse("Toto", "Monsieur")
Veuillez agréer, Monsieur Toto , mes sincères salutations
```

Arguments facultatifs (2)

- Connaissez-vous des fonctions, que vous avez déjà manipulées, qui ont des paramètres par défaut ?

Arguments facultatifs (2)

- Connaissez-vous des fonctions, que vous avez déjà manipulées, qui ont des paramètres par défaut ?

```
>>> range(10)
range(0, 10)
>>> range(1, 10)
range(1, 10)
>>> range(1, 10, 2)
range(1, 10, 2)
```

Particularités : Arguments sans ordre (1)

Arguments avec étiquettes

- Dans la plupart des langages de programmation, les arguments doivent être passés **dans le même ordre** que la défition des paramètres
- Python autorise une souplesse, si tous les paramètres ont été définis avec une valeur par défaut, on peut faire appel à la fonction en fournissant les arguments correspondants **dans n'importe quel ordre, à la condition de désigner nommément les paramètres correspondants**

Source : G. Swinnen. « Apprendre à programmer avec Python 3 »

Particularités : Arguments sans ordre (2)

```
>>> def fct(x=0, y=0, z=0):
...     return x, y, z
...
>>> fct()
(0, 0, 0)
>>> fct(10)
(10, 0, 0)
>>> fct(y=20, z=30, x=10)
(10, 20, 30)
>>> fct(y=20, z=30)
(0, 20, 30)
>>> fct(1, z=30)
(1, 0, 30)
```

```
>>> def fct(a, b, x=0, y=0, z=0):
...     return a, b, x, y, z
...
>>> fct(1, 1)
(1, 1, 0, 0, 0)
>>> fct(1, 1, y=20)
(1, 1, 0, 20, 0)
>>> fct(z=30)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: fct() missing 2 required positional arguments: 'a' and 'b'
```

Source : https://python.sdv.univ-paris-diderot.fr/09_fonctions/

Particularités : Typage dynamique

Typage dynamique

- Le type des arguments n'est pas spécifié dans la définiton de la fonction
- Python reconnaît le type des variables au moment de l'exécution
- Il est préfarable que chaque argument ait un type précis et non l'un ou l'autre

```
>>> def fois(x, y):
...     return x*y
...
>>> fois(2, 3)
6
>>> fois('to', 2)
'toto'
>>> fois([1,3], 2)
[1, 3, 1, 3]
```

Définitions

Variables globales

Une variable du corps principal du programme est dite **globale**

Variables locales à la fonction

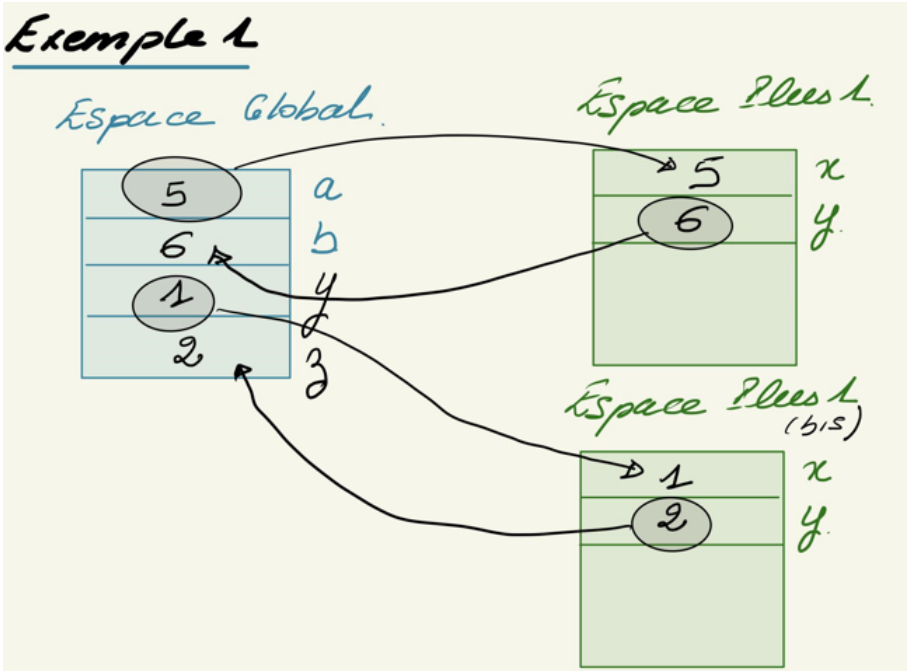
- Une variable « déclarée » à l'intérieure d'une fonction est dite **locale**
- Elle n'existera et ne sera **visible** que lors de l'exécution de la fonction
- Chaque fois que la fonction est appelée, un **espace mémoire** est réservé (différent)
- Ces variables ne sont pas **accessibles** en déhors de la fonction

Exemple (1)

```
>>> def plus1(x) :
...     y = x + 1
...     return y
...
>>> a = 5
>>> b = plus1(a)
>>> b
6
>>> print(y)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'y' is not defined
>>> y = 1
>>> z = plus1(y)
>>> z
2
>>> y # ???
```

Exemple (1)

```
>>> def plus1(x) :
...     y = x + 1
...     return y
...
>>> a = 5
>>> b = plus1(a)
>>> b
6
>>> print(y)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'y' is not defined
>>> y = 1
>>> z = plus1(y)
>>> z
2
>>> y # ???
```

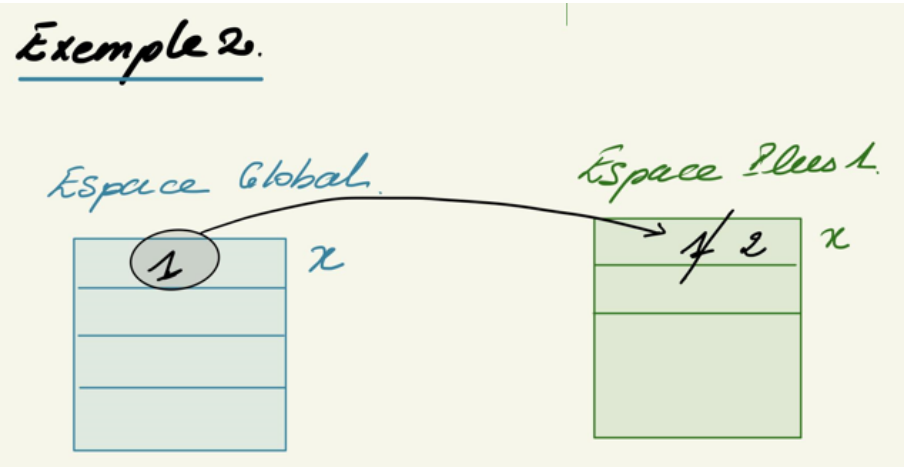


Exemple (2)

```
>>> def plus1(x) :
...         x += 1
...         return x
...
>>> x = 1
>>> plus1(x)
2
>>> x # ??
```

Exemple (2)

```
>>> def plus1(x) :
...         x += 1
...         return x
...
>>> x = 1
>>> plus1(x)
2
>>> x # ??
```

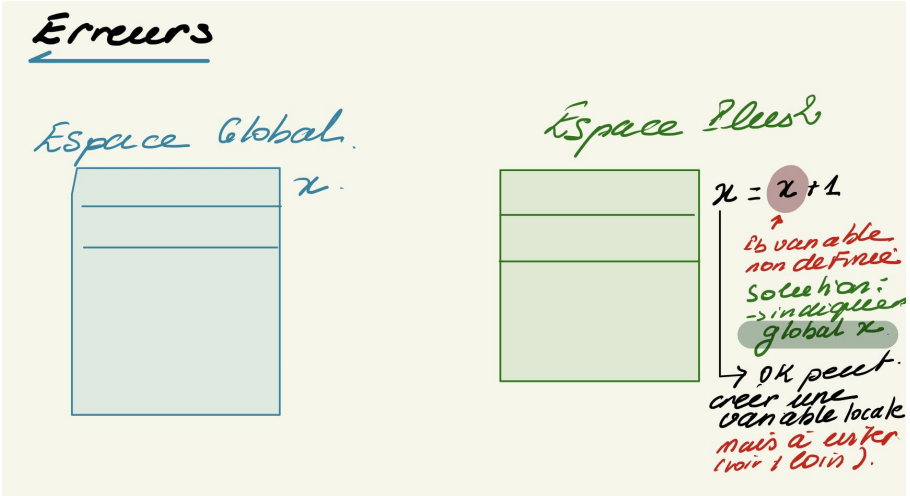


Erreurs

```
>>> def plus2() :
...     x = x + 2
...
>>> x = 1
>>> z = plus2()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 2, in plus2
UnboundLocalError: local variable 'x' referenced before assignment
```

Erreurs

```
>>> def plus2() :
...     x = x + 2
...
>>> x = 1
>>> z = plus2()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 2, in plus2
UnboundLocalError: local variable 'x' referenced before assignment
```

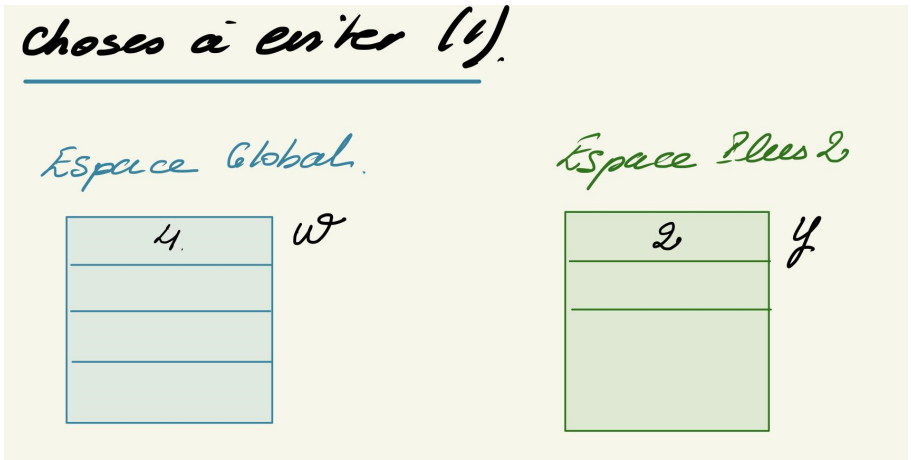


Choses à éviter (1)

```
>>> def plus2(y) :
...     global w
...     w = y + 2
...
>>> plus2(2)
>>> w
4
```

Choses à éviter (1)

```
>>> def plus2(y) :
...     global w
...     w = y + 2
...
>>> plus2(2)
>>> w
4
```



Global ou Local ?

Eviter

- Eviter autant que possible les variables globales
- Cela peut être risqué...

Une procédure ou une fonction doit

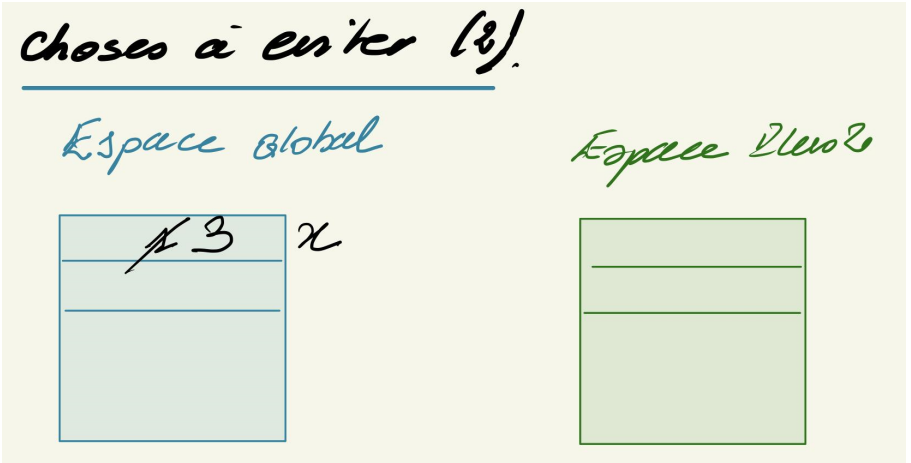
- effectuer la tâche qui lui a été confiée,
- en ne modifiant que l'état de ses variables locales

Choses à éviter (2)

```
>>> def plus2() :
...     global x
...     x = x + 2
...
>>> x = 1
>>> plus2()
>>> x
3
```

Choses à éviter (2)

```
>>> def plus2() :
...     global x
...     x = x + 2
...
>>> x = 1
>>> plus2()
>>> x
3
```

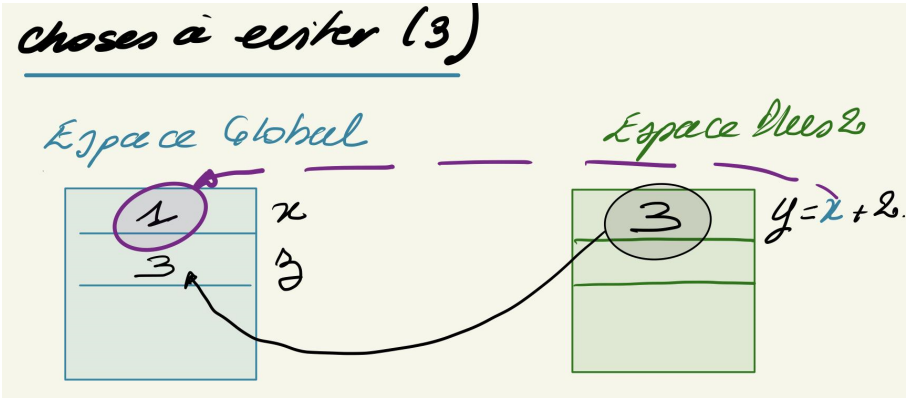


Choses à éviter (3)

```
>>> def plus2() :
...     y = x + 2
...     return y
...
>>> x = 1
>>> z = plus2()
```

Choses à éviter (3)

```
>>> def plus2() :
...         y = x + 2
...         return y
...
>>> x = 1
>>> z = plus2()
```



Modes de paramètres

On distingue généralement deux modes de passage de paramètres :

- Par valeur
  - Seule la valeur de l'argument est transmise au paramètre. L'argument ne peut donc pas être modifié
- Par référence (ou par adresse, ou encore par variable)
  - Lorsque l'argument est une variable, tout se passe comme si la fonction travaillait directement avec cette variable. L'argument peut alors être modifié.

En python, aucun de ces deux modes ne s'applique vraiment :

- Il s'agit plutôt d'un passage par référence d'objet
- Lorsque l'argument est une **variable**, il ne peut en général pas être modifié par la fonction (ou la procédure)
- En revanche, les éléments d'une liste (et donc d'un tableau) peuvent être modifiés

Rappel

```
>>> def plus1(x) :
...     y = x + 1
...     return y
...
>>> a = 5
>>> b = plus1(a)
>>> b
6
>>> print(y)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'y' is not defined
>>> y = 1
>>> z = plus1(y)
>>> z
2
>>> y # ???
```

Modification d'un tableau (effet de bord) (1)

```
>>> def replace (old, new, tab) :
...     for i in range(len(tab)) :
...         if tab[i] == old :
...             tab[i] = new
...
>>> t = ["mais", "où", "et", "donc", "hors", "ni", "car"]
>>> replace("hors", "or", t)

>>> t
```

Modification d'un tableau (effet de bord) (2)

```
>>> def replace (old, new, tab) :
...     for i in range(len(tab)) :
...         if tab[i] == old :
...             tab[i] = new
...
>>> t = ["mais", "où", "et", "donc", "hors", "ni", "car"]
>>> replace("hors", "or", t)

>>> t
```

```
>>> t
['mais', 'où', 'et', 'donc', 'or', 'ni', 'car']
```

Modification d'un tableau (effet de bord) (3)

Si on veut faire cela plus proprement :

```
def replaceReturn(old, new, tab) :
    'Pour être plus clair'
    for i in range(len(tab)) :
        if tab[i] == old :
            tab[i] = new
    return tab
t = ["mais", "où", "et", "donc", "hors", "ni", "car"]
t = replaceReturn("hors", "or", t)
print(t)
```

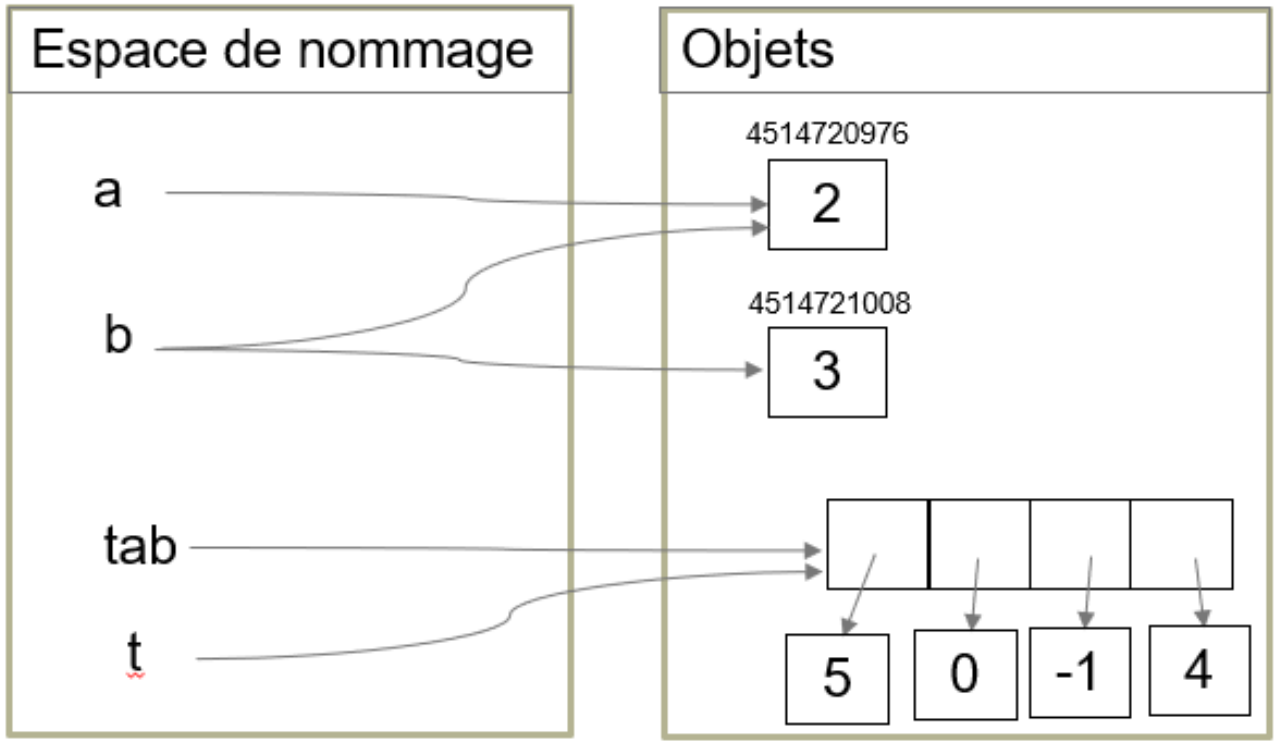
Modification d'un tableau (effet de bord) (4)

Si on ne veut pas modifier le tableau :

```
def replaceSansModif(old, new, tab) :
    tab2 = tab.copy()
    # tab2 = tab # Attention pas une copie
    for i in range(len(tab)) :
        if tab2[i] == old :
            tab2[i] = new
    print(tab2)
t = ["mais", "où", "et", "donc", "hors", "ni", "car"]
replaceSansModif("hors", "or", t)
print(t)
```

Représentation mémoire) (1)

```
>>> a = 2
>>> b = a
>>> id(a)
4514720976
>>> id(b)
4514720976
>>> b = b + 1
>>> id(a)
4514720976
>>> id(b)
4514721008
>>> tab = [5, 0, -1]
>>> t = tab
>>> t.append(4)
>>> tab
[5, 0, -1, 4]
```

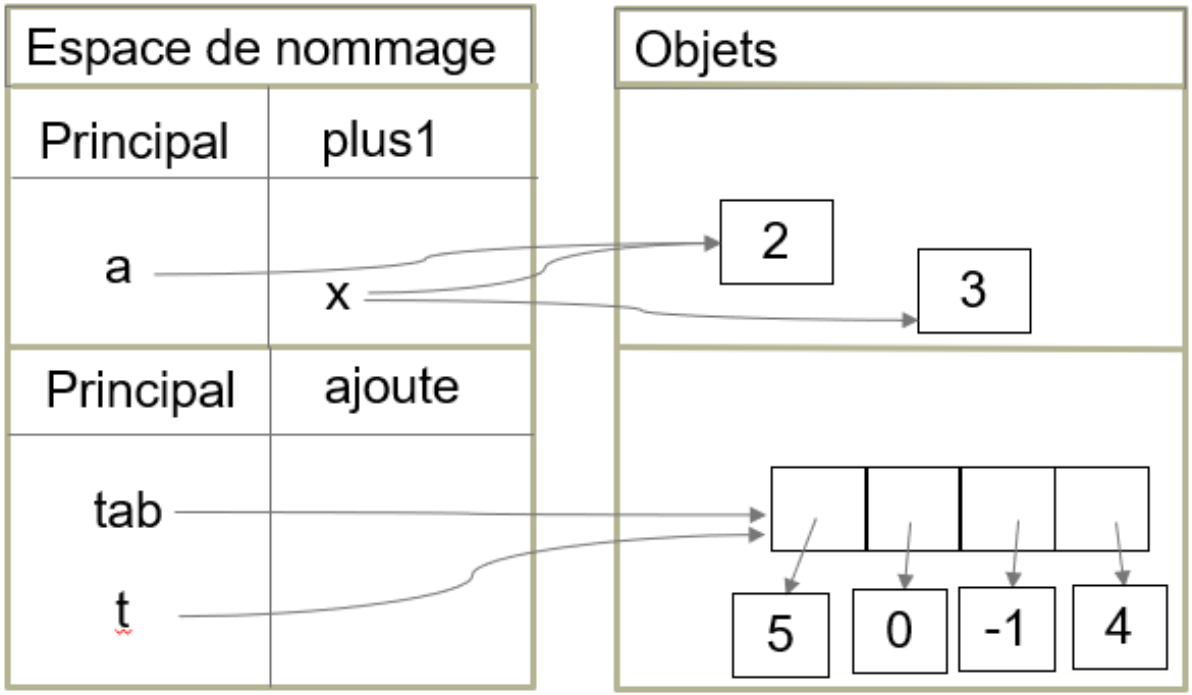


N.B. : La fonction id retourne l'identifiant (adresse) de l'objet associé au nom

Représentation mémoire) (2)

```
def plus1(x):
    x += 1
    print(x)
Test :
>>> a = 2
>>> plus1(a)
3
>>> a
2
```

```
def ajoute(elt, t):
    t.append(elt)
Test :
>>> tab = [5, 0, -1]
>>> ajoute(4, tab)
>>> tab
[5, 0, -1, 4]
```



N.B. : a n'est pas modifié, alors que tab l'est

A retenir

- Une variable ne peut en général pas être modifiée lorsqu'on la passe en paramètre
- Elle peut être modifiée si son type est « mutable » (ou muable)
- Un tableau passé en paramètre peut être modifié. Il s'agit alors d'un « effet de bord »
- Les effets de bord sont à proscrire. Ils nuisent à la lisibilité du programme et à la réutilisabilité des fonctions et procédures = renvoyer le tableau.

Documentation

Il est souvent utile de "documenter" une fonction à l'aide d'une chaine de caractères (docstring)

```
>>> def cube(x) :
...     "Retourne le cube d'un nombre x"
...     return x**3
...
>>> help(cube)
...
Help on function cube in module __main__:

cube(x)
    Retourne le cube d'un nombre x
```

Importation de modules

Un module est un fichier qui regroupe un ensemble de fonctions (et éventuellement de variables, de constantes, de classes)

- Pour importer un module : *import nom_du_module*
- Liste des fonctions disponibles : *dir*(*nom_du_module*)

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin',
 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees',
 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt',
 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter',
 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh',
 'tau', 'trunc', 'ulp']
>>> # Exemples d'utilisation
>>> r1 = (- b + math.sqrt(5)) / (2 * a)
>>> perimetre = 2 * math.pi * r
```

Autres formes d'importation

On peut choisir de n'importer qu'un ou plusieurs éléments :

- Exemple : *from math import pi, sin, cos, tan*

On peut aussi importer complètement un module par :

- Exemple : *from math import* $*$

Avantage : utilisation simplifiée

```
>>> from math import *
>>> r1 = (- b + sqrt(5)) / (2 * a)
>>> perimetre = 2 * pi * r
```

Inconvénient : tous les noms sont importés. Il faut donc veiller à ne pas les réutiliser

Structure d'un programme Python

Ordre :

- commentaires
- Importation de modules
- Définition éventuelle de variables globales
- Définition de fonctions
- Corps principal du programme

Eviter

- **Les fonctions doivent être définies avant leur utilisation**
- **Une fonction peut faire appel à une autre fonction** (imbriquée ou non)

Imbrication de fonctions

On peut imbriquer une fonction dans une autre

```
# Imbrication de fonctions
def f(x) :
    # fonction cube imbriquee dans f
    def cube (z) :
        return z**3
    # corps de la fonction f
    y = 3 * cube(x) + 1
    return y

# corps de la fonction principale
y = f(2)
print(y)
# affichera 25
```

Remarque

la fonction cube n'est pas utilisable dans le programme principal

Questions...