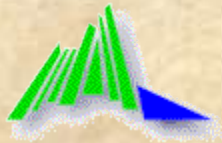


Techniques de Programmation Algorithmique & Langage C

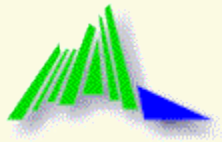
Manuel Munier

IUT des Pays de l'Adour - Mont de Marsan

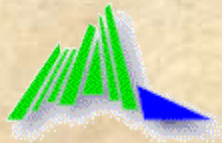
Département GTR

2002-2003

Plan du cours



- Algorithmique & Python
  - variables, opérateurs, expressions
  - séquence d'instructions
  - conditionnelle (`if`)
  - boucles (`while`, `for`)
  - fonctions
- Langage C
  - types, instructions C, compilation, fonctions, tableaux, pointeurs, listes, arbres, fichiers, récursivité,... bref, de quoi s'occuper ;-)



Partie n°1

Algorithmique
Langage Python

Introduction

- Programmation ?
  - Règle 1: un ordinateur est une **machine stupide**
  - Règle 2: c'est le programmeur qui doit lui dire, étape par étape, ce qu'il doit faire
  - Règle 3: les ordres "élémentaires" sont très peu nombreux

Introduction

- Programmation ?
  - Corollaire: la programmation consiste, à partir d'un problème donné, à expliquer en détail à la machine ce qu'elle doit faire pour aboutir au résultat attendu
  - Comme il n'y a que peu d'ordres élémentaires, toute la difficulté est de construire un enchaînement correct

Introduction

- Langage de programmation
  - A la base, un ordinateur n'est qu'un ensemble de circuits électroniques travaillant sur des signaux électriques
  - Ces signaux ne peuvent prendre que deux états (ex: 0V et +5V) ➔ on parle de logique binaire (deux états codés 0 et 1)
  - En interne, un ordinateur ne peut donc traiter que des successions de 0 et de 1: c'est le **format binaire**

Introduction

- Langage de programmation
  - Tous les ordres à exécuter et toutes les données à traiter (nombres entiers, réels,…) doivent donc être traduits en binaire
  - On parle de **langage machine**
    - vous y reviendrez plus en détail en cours d'architecture des ordinateurs… ☺
  - Parler en 0 et en 1 est bien trop fastidieux !

  ➔D'où les **langages de programmation**

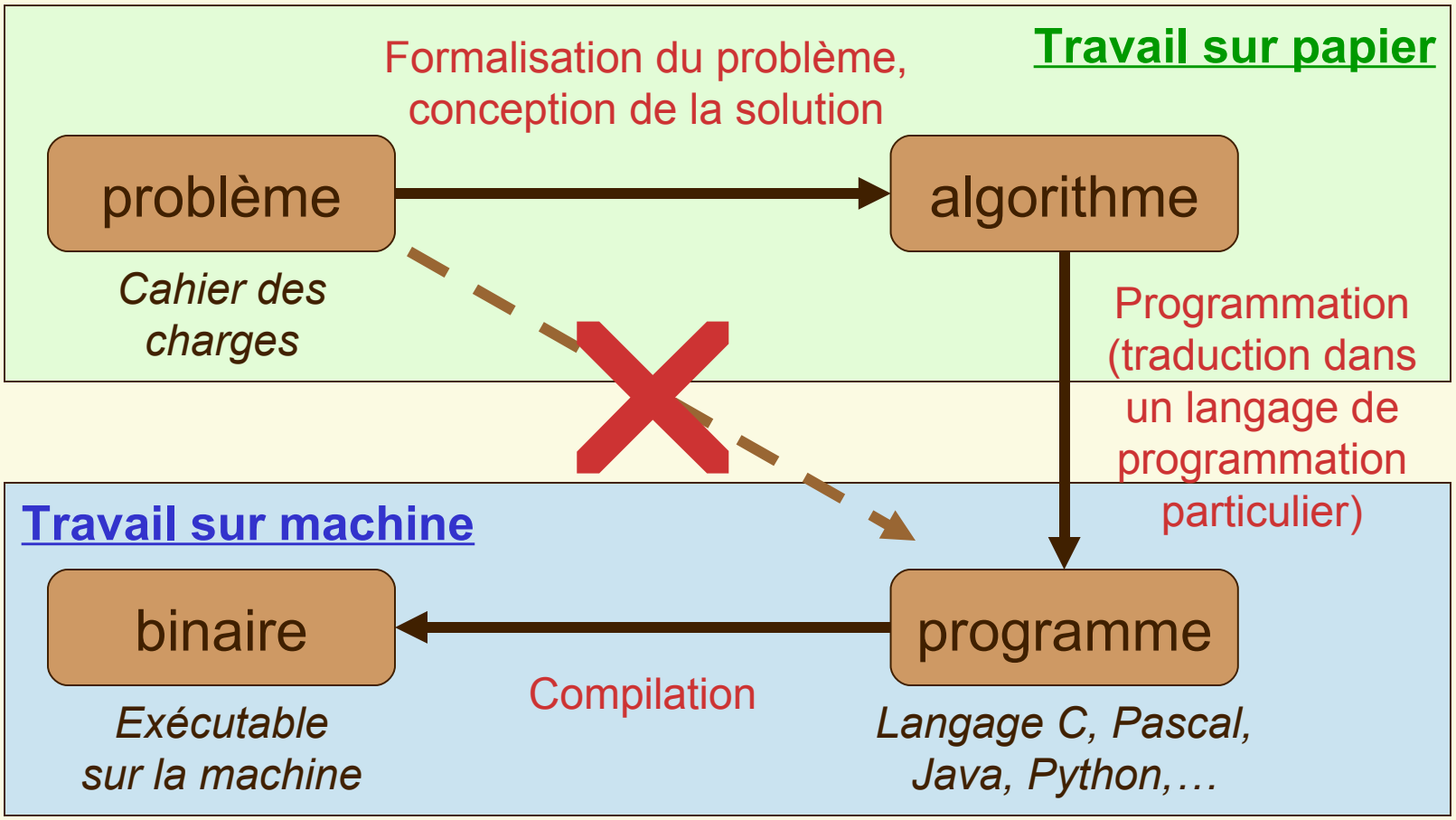
Introduction

- Langage de programmation
  - Pour "parler" à un ordinateur (lui donner des ordres), nous utiliserons donc un langage de programmation
  - C'est le langage de programmation qui définit les ordres que l'on peut utiliser
  - Il existe de nombreux langages de programmation différents
    - plus ou moins riches en ordres (les **mots** du langage)
    - plus ou moins compliqués (la **syntaxe** du langage)

Introduction

- Et l'algorithmique alors ?
  - L'algorithme est un niveau intermédiaire entre l'énoncé du problème (cahier des charges) et le programme
  - L'algorithme permet de formaliser la solution indépendamment d'un langage de programmation particulier
  - Un algorithme n'existe que **sur papier** ! Il n'est **pas exécutable** sur machine

Introduction



Introduction

- Et Python dans tout ça ?
  - Inconvénient des algo: on ne peut pas les exécuter pour les tester
  - Inconvénient des langages de prog: on doit connaître beaucoup de notions avant de pouvoir écrire son premier programme:
    - types de données
    - entête d'un programme
    - phases de compilation (pour le traduire en binaire)
    - etc

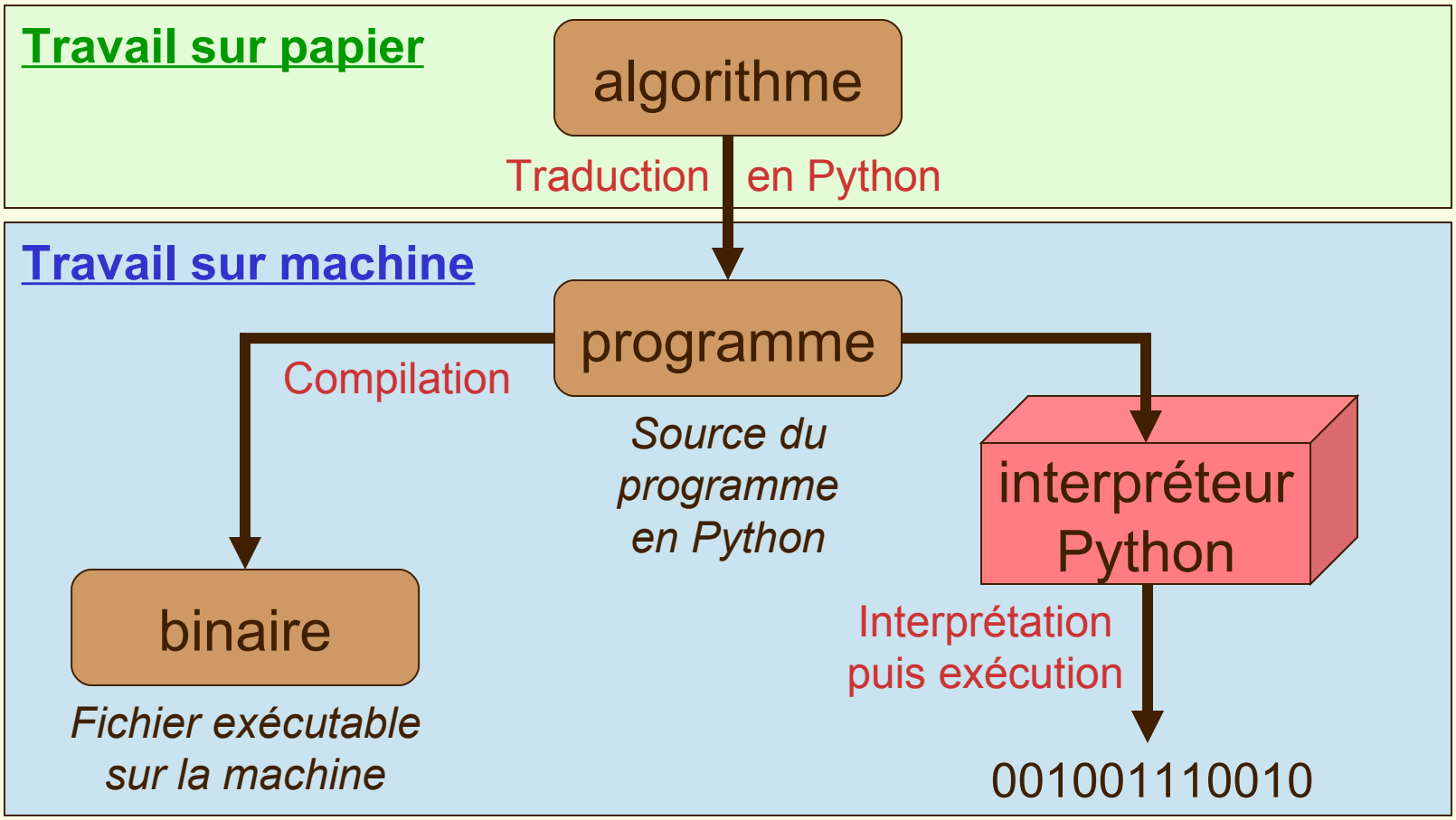
Introduction

- Et Python dans tout ça ?
  - Python est un langage de prog récent (1989)
  - Bien que regroupant les "bonnes" propriétés d'autres langages, ses concepteurs l'ont voulu facile à utiliser

  ➔Python est très proche des langages algorithmiques

  - La compilation n'est pas nécessaire: Python est un langage interprété

Introduction



Introduction

- Finalement ?
  - Pour la première partie du cours (algorithmique) nous utiliserons Python
    - adopter la démarche intellectuelle pour "commander" un ordinateur
    - écrire des algorithmes exécutables
    - s'exercer à trouver les bugs et à les corriger ☺
  - Quand vous serez rodés, nous étudierons ensuite un langage de programmation particulier: le langage C

Plan

- Instructions élémentaires
  - variables, opérateurs, expressions
  - affichage/lecture de données (`print`/`input`)
- Instructions de contrôle
  - séquence d'instructions
  - exécution conditionnelle (`if`)
  - instructions composées
- Instructions répétitives
  - boucle (`while`,`for`)
- Fonctions
- Tableaux

Variables

- Etymologiquement le mot **informatique** signifie "science des informations"
- Il nous faut donc stocker ces informations (ou données) dans l'ordinateur: le plus simple est d'utiliser la **mémoire centrale**
- Une **variable** représente une "case" de la mémoire centrale

Variables

- Une variable est caractérisée par:
  - le **nom de la variable**, appelé identificateur
    - ce nom est plus parlant pour le programmeur
    - pour l'ordinateur il s'agit simplement d'une étiquette reliée à l'adresse mémoire de la variable
  - le **type des données** que la variable peut contenir: numérique, vecteur, chaîne de caractères,...
    - les types des variables sont facultatifs en Python !
  - la **valeur** stockée dans la variable
    - cette valeur pourra changer au cours du programme

Variables

- Identificateur
  - c'est une séquence de lettres (**a** → **z**, **A** → **Z**) et de chiffres (**0** → **9**) qui doit obligatoirement commencer par une lettre
  - pas d'accent, de cédille, d'espace, de caractère tel que **$**, **#**, **@**, etc… (excepté le **_**)
  - la casse est significative: **Joseph**, **joseph** et **JOSEPH** sont 3 identificateurs différents
  - attention aux mots réservés (mots-clés)

Variables

- Mots réservés en Python

```
and       else      import    raise
assert    except    in        return
break     exec      is        try
class     finally   lambda    while
continue  for       not
def       from      or
del       global    pass
elif      if        print
```

Variables

- Affectation (ou assignation)
  - Affecter une valeur à une variable est l'opération qui consiste à **modifier la valeur** qui se trouve dans la case mémoire
  - Notation

```
n ← 7
msg ← "Quoi de neuf ?"
pi ← 3,14159
```

"langage" algorithmique

```
n = 7
msg = "Quoi de neuf ?"
pi = 3.14159
```

Python

Variables

- Affectation
  - Le signe = de Python correspond à ← en algo
    - Ce n'est pas une égalité au sens des mathématiques
    - On peut, à tout instant dans le programme, lui réaffecter une nouvelle valeur
    - Signification: "on met la valeur qui se trouve à droite du signe dans la variable dont le nom est indiqué à gauche"

<nom de variable> ← <valeur>

Variables

- Affectations multiples en Python
  - `x = y = 7`
  - `a,b = 4, 8.33`
- Nombres réels
  - Le séparateur décimal est le **point décimal** et non la virgule !

Variables

- Affichage de la valeur

```
n ← 7
msg ← "Quoi de neuf ?"
pi ← 3,14159
afficher(n)
afficher(msg,pi)
```

Algo

```
n = 7
msg = "Quoi de neuf ?"
pi = 3.14159
print n
print msg,pi
```

Python

Expressions

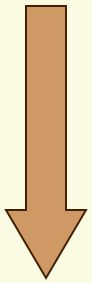
- Les **opérateurs** permettent de combiner les valeurs et les variables pour construire des **expressions**

```
a,b = 7.3, 12
y = 3*a + b/5
```

- Signification de ces 2 lignes:
  - On affecte la valeur `7.3` à la variable `a` puis la valeur `12` à la variable `b`
  - On récupère la valeur de `a`, on la multiplie par `3`, et on lui ajoute la valeur de `b` divisée par `5`. Le résultat de cette évaluation (valeur) est placé dans la variable `y`

Expressions

- Opérateurs
  - +, -, *
  - / entre deux entiers est la division entière
  - / avec au moins un réel est la division réelle
  - ** pour l'exponentiation
  - % pour le modulo (reste de la division entière)
- Règles de priorité (décroissante)



Parenthèses
Exposants
Multiplication et division
Addition et soustraction

Si deux opérateurs ont la même priorité, l'évaluation est effectuée de gauche à droite

Expressions

- Composition

```
h,m,s = 11,45,23     # 11h45 et 23 secondes
total = h*3600 + m*60 + s
print "nombre de secondes", total
```

peut également s'écrire

```
h,m,s = 11,45,23     # 11h45 et 23 secondes
print "nombre de secondes", h*3600 + m*60 + s
```

- Remarques
  - `m+1 = b` est incorrect car "`m+1`" n'est pas un nom de variable
  - `a = a+1` est correct (on augmente `a` de 1) alors qu'en mathématiques…

Saisies

- Plutôt que de mettre des valeurs fixes, vous pouvez demander à l'utilisateur de saisir une valeur

Algo

```
afficher("entrez une valeur")
lire(a)
afficher("valeur lue: ",a)
```

Python

```
print "entrez une valeur"
a=input()
print "valeur lue: ",a
```

- Ceci interromps l'exécution de votre programme jusqu'à ce que l'utilisateur ait saisi une valeur
  - `15.4 ↵`
  - `"les vacances sont terminées !!!" ↵`

Plan

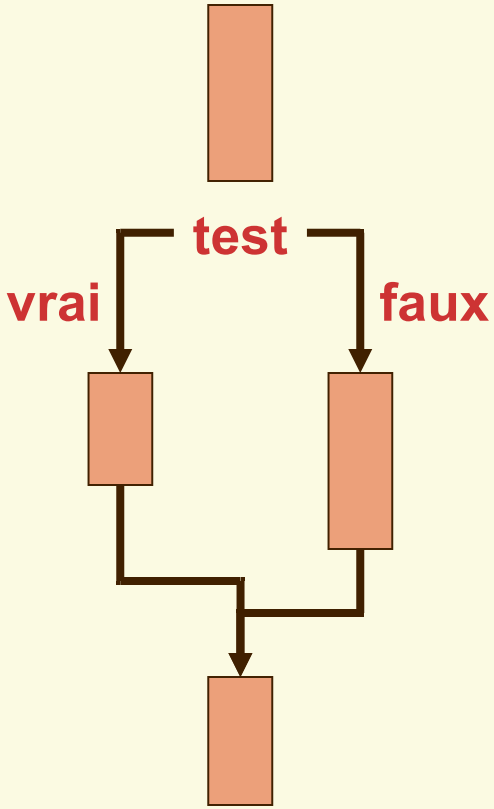
- Instructions élémentaires
  - variables, opérateurs, expressions
  - affichage/lecture de données (`print`/`input`)
- Instructions de contrôle
  - séquence d'instructions
  - exécution conditionnelle (`if`)
  - instructions composées
- Instructions répétitives
  - boucle (`while`,`for`)
- Fonctions
- Tableaux

Instructions de contrôle

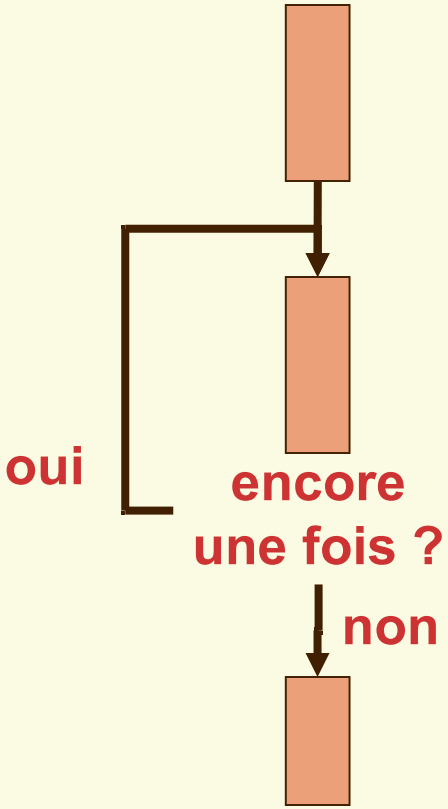
- Sauf mention explicite, les instructions s'exécutent les unes après les autres, dans l'ordre où elles ont été écrites à l'intérieur du programme (de la première à la dernière)
- Les instructions qui permettent de modifier ce flux d'instructions sont appelées des instructions de contrôle (de flux)
  - conditionnelle (`if`)
  - boucles (`while`, `for`)
  - etc…
- Le programme peut ainsi suivre des "chemins" différents selon les circonstances

Instructions de contrôle

- Conditionnelle (`if`)



- Boucle (`while`, `for`)



Conditionnelle

- Cette instruction permet de n'exécuter une partie du programme que si une certaine condition est vérifiée

```
lire(nom)
lire(note)
si (note > 10) alors
        afficher(nom, " a obtenu la
 moyenne")
fsi
afficher(nom, ": ", note, "/20")
```

Conditionnelle

- L'expression placée entre parenthèses est une **condition booléenne**: la valeur de cette expression sera **vrai** ou **faux**
- L'instruction **si** permet de tester la validité de cette condition
  - Si la condition est vraie, les instructions placées entre le **alors** et le **fsi** (fin de si) sont exécutées. Ensuite on continue en séquence.
  - Si la condition est fausse, on "saute" directement à la fin du test et on continue en séquence.

Conditionnelle


```
lire(nom)
lire(note)
si (note > 10) alors
    afficher(nom, " a obtenu la moyenne")
fsi
afficher(nom, ": ", note, "/20")
```

- Exemple 1:

```
"munier" ↲
19.5 ↲
"munier a obtenu la moyenne"
"munier: 19.5/20"
```

- Exemple 2:

```
"lespine" ↲
4 ↲
"lespine: 4/20"
```

Conditionnelle

- Dans le cas où la condition est fausse, il est possible de fournir une alternative, i.e. des instructions qui ne seront exécutées que si la condition est fausse

```
lire(nom)
lire(note)
si (note > 10) alors
    afficher(nom, " a obtenu la moyenne")
sinon
    afficher(nom, " peut mieux faire !")
fsi
afficher(nom, ": ", note, "/20")
```

Conditionnelle

- Exemple 1:

```
"munier" ↲
19.5 ↲
"munier a obtenu la moyenne"
"munier: 19.5/20"
```

exécuté car la condition est vraie

- Exemple 2:

```
"lespine" ↲
4 ↲
"lespine peut mieux faire !"
"lespine: 4/20"
```

exécuté car la condition est fausse

Conditionnelle

- Traduction du **si-alors-sinon** en Python:

```
nom=input()
note=input()
if (note > 10):
    print nom, " a obtenu la moyenne"
else:
    print nom, " peut mieux faire !"
print nom, ": ", note, "/20"
```

Conditionnelle

- Indentation
  - C'est le fait de décaler les instructions qui se trouvent dans le corps du test
  - Ceci améliore la lisibilité du programme
    - Donc **très fortement conseillé**, que ce soit en algo ou dans n'importe quel langage de programmation
    - **Obligatoire** en Python ;-)
  - Ces remarques sont valables pour toutes les **instructions composées**

Instructions composées

- Vous n'êtes pas limité à une seule instruction dans le corps d'un test
  - Vous pouvez mettre plusieurs instructions
  - On appelle ceci un **bloc d'instructions**
- Comment déterminer le début et la fin d'un bloc d'instructions ?
  - Algo: indentation et mot-clé `fsi`
  - Python: indentation uniquement
  - Autres: `{` et `}` en C, `begin/end` en Pascal

Condition booléenne

- Vous pouvez également construire des expressions booléennes avec les opérateurs suivants:
  - `x == y  # x est égal à y (double signe = !!!)`
  - `x != y  # x est différent de y`
  - `x > y   # x est plus grand que y`
  - `x < y   # x est plus petit que y`
  - `x >= y  # x est supérieur ou égal à y`
  - `x <= y  # x est inférieur ou égal à y`

  - `a and b # vrai si a et b sont tous les 2 vrais`
  - `a or b  # vrai si a est vrai ou si b est vrai`
    `# (ou les deux en même temps)`
  - `not a   # vrai si a est faux`

Exemple

- Condition booléenne + composition

```
ue1 = input()
ue2 = input()
ue3 = input()
moy = (ue1 + ue2 + ue3) / 3.0

if (moy >= 10):
    print "avoir la moyenne n'était pas si dur…"
    if ( (ue1 >= 8) and (ue2 >= 8) and (ue3 >= 8) ):
        print "année validée"
    else:
        print "redoublement"
else:
    print "réorientation !"
```

Plan

- Instructions élémentaires
  - variables, opérateurs, expressions
  - affichage/lecture de données (`print/input`)
- Instructions de contrôle
  - séquence d'instructions
  - exécution conditionnelle (`if`)
  - instructions composées
- Instructions répétitives
  - boucle (`while`,`for`)
- Fonctions
- Tableaux

Boucle

- Une boucle (ou itération) consiste à **répéter** plusieurs fois un ensemble d'instructions
  - Quel est le bloc d'instructions à répéter ?
    - ➔ **Corps de la boucle**
  - Quand doit-on s'arrêter ?
    - ➔ **Condition d'arrêt de la boucle**

Boucle tant-que

- Répéter une tâche tant qu'une certaine condition est vérifiée
- Exemple:

```
a ← 0
tant que (a<3) faire
    a ← a+1
    afficher(a)
ftant
afficher("on s'arrête là")
```

condition de continuation

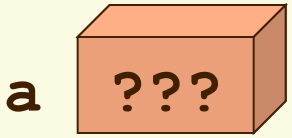
corps de la boucle

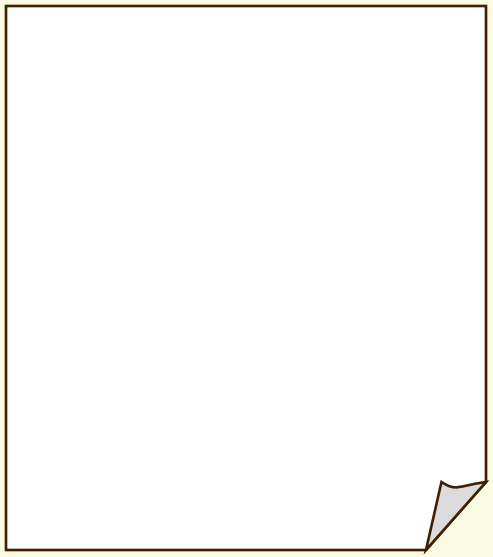
Boucle tant-que

- Exécution:

début du programme

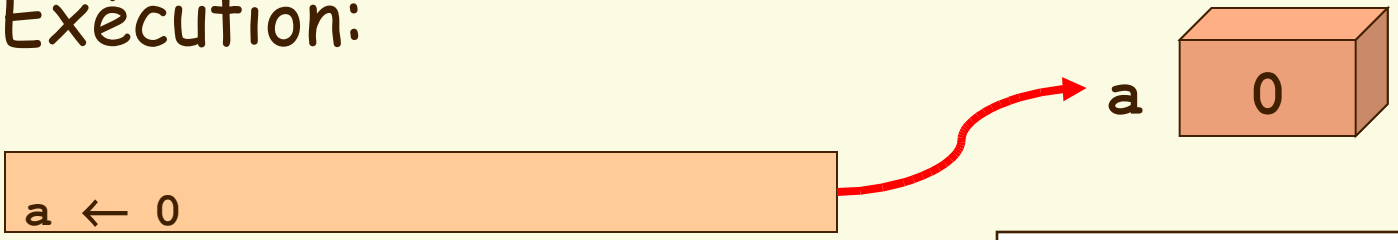
```
a ← 0
tant que (a<3) faire
    a ← a+1
    afficher(a)
ftant
afficher("on s'arrête là")
```





Boucle tant-que

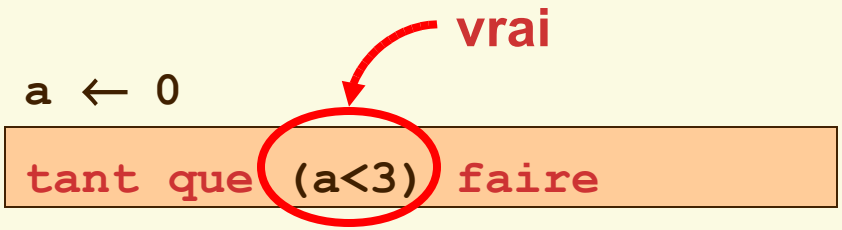
- Exécution:

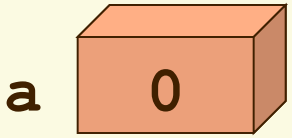


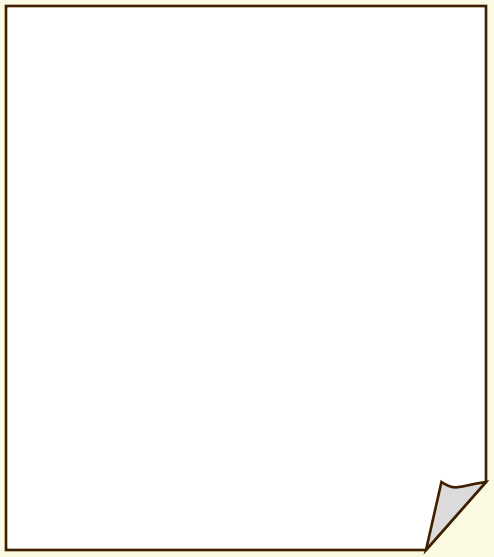

```
tant que (a<3) faire
    a ← a+1
    afficher(a)
ftant
afficher("on s'arrête là")
```

Boucle tant-que

- Exécution:



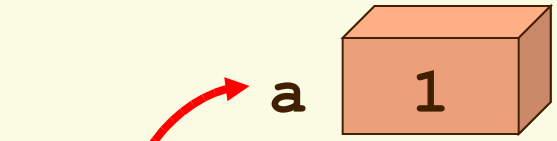


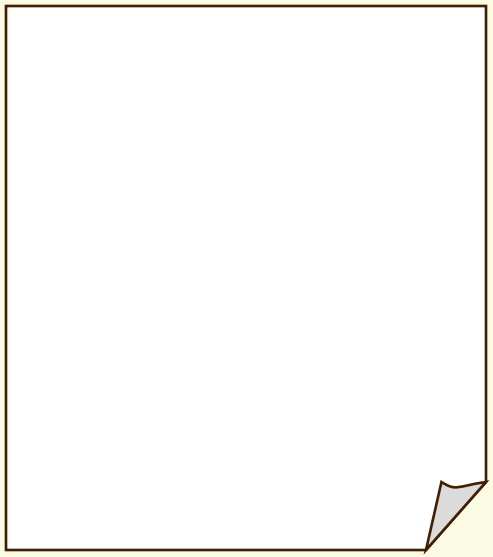


Boucle tant-que

- Exécution:

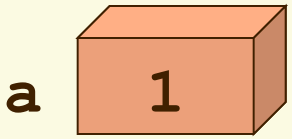
```
a ← 0
tant que (a<3) faire
    a ← a+1
    afficher(a)
ftant
afficher("on s'arrête là")
```



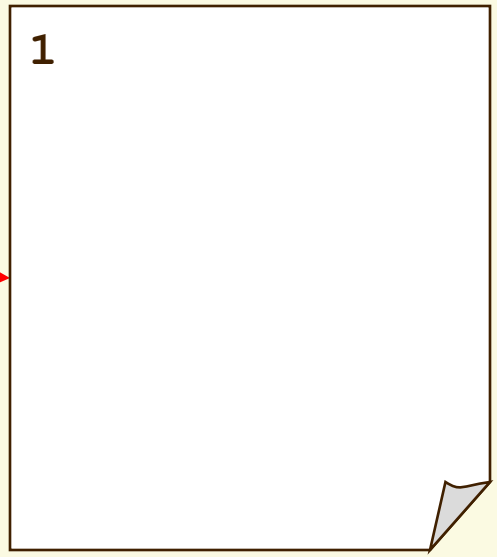


Boucle tant-que

- Exécution:

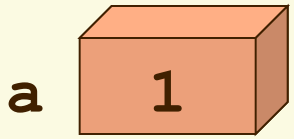



```
a ← 0
tant que (a<3) faire
    a ← a+1
    afficher(a)
ftant
afficher("on s'arrête là")
```

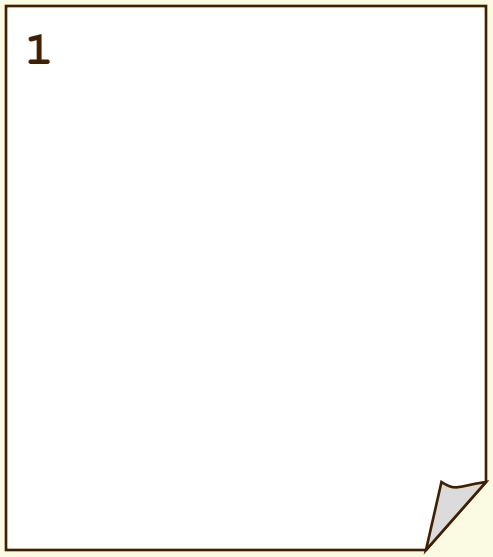


Boucle tant-que

- Exécution:

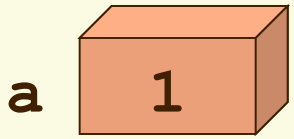



```
a ← 0
tant que (a<3) faire
    a ← a+1
    afficher(a)
ftant
afficher("on s'arrête là")
```



Boucle tant-que

- Exécution:



vrai

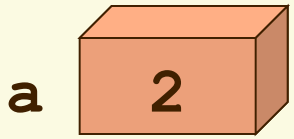
```
a ← 0
tant que (a<3) faire
    a ← a+1
    afficher(a)
ftant
afficher("on s'arrête là")
```

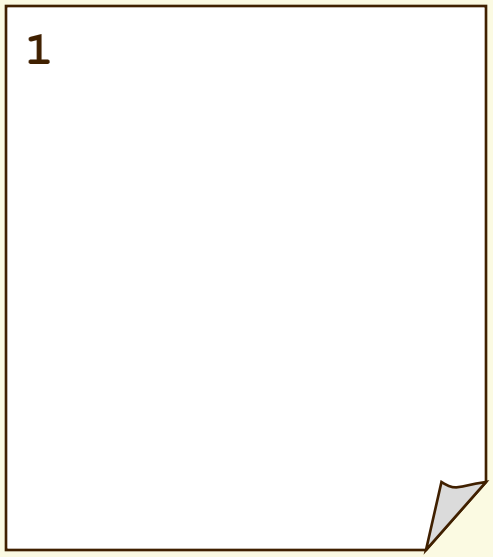
```
1
```

Boucle tant-que

- Exécution:

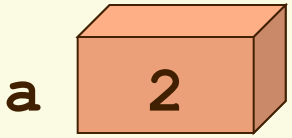
```
a ← 0
tant que (a<3) faire
    a ← a+1
    afficher(a)
ftant
afficher("on s'arrête là")
```



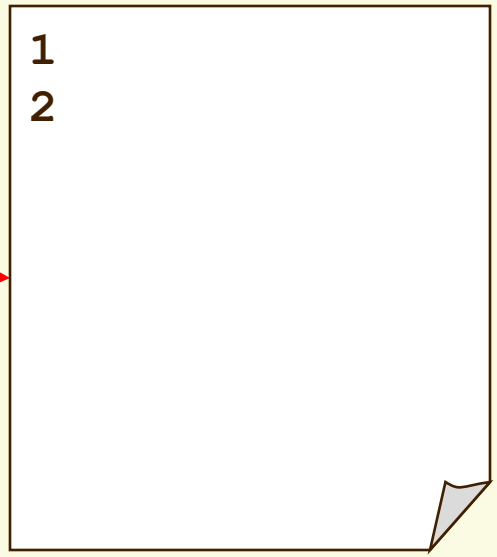


Boucle tant-que

- Exécution:

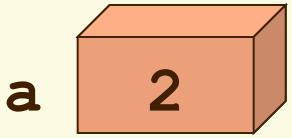



```
a ← 0
tant que (a<3) faire
    a ← a+1
    afficher(a)
ftant
afficher("on s'arrête là")
```



Boucle tant-que

- Exécution:

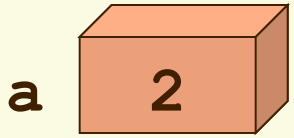


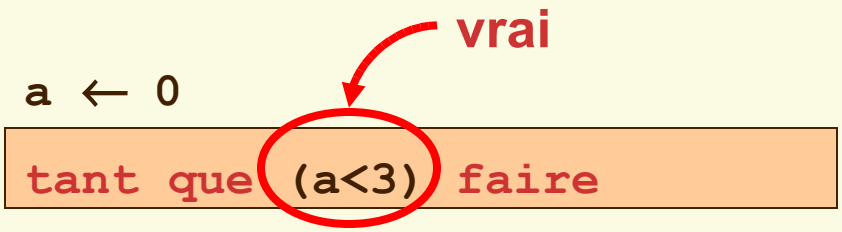

```
a ← 0
tant que (a<3) faire
    a ← a+1
    afficher(a)
ftant
afficher("on s'arrête là")
```

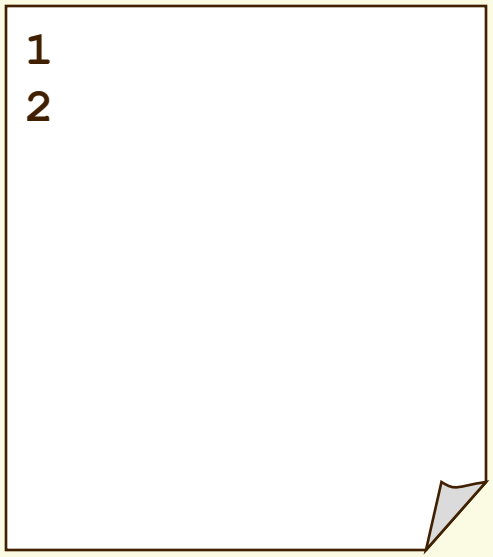
```
1
2
```

Boucle tant-que

- Exécution:



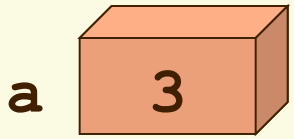




Boucle tant-que

- Exécution:

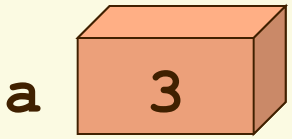
```
a ← 0
tant que (a<3) faire
    a ← a+1
    afficher(a)
ftant
afficher("on s'arrête là")
```



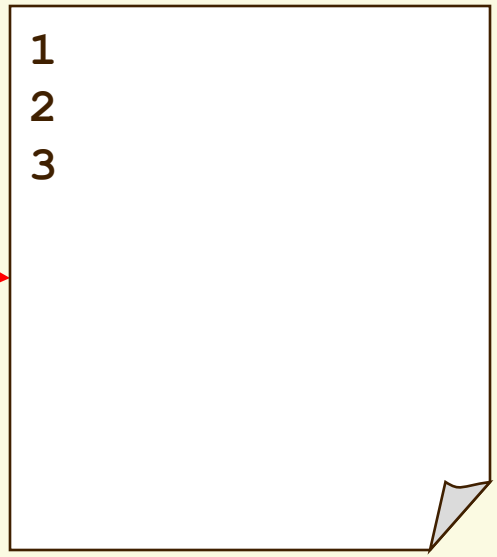

```
1
2
```

Boucle tant-que

- Exécution:

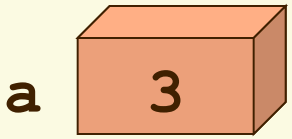



```
a ← 0
tant que (a<3) faire
    a ← a+1
    afficher(a)
ftant
afficher("on s'arrête là")
```



Boucle tant-que

- Exécution:

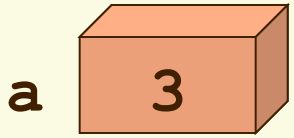



```
a ← 0
tant que (a<3) faire
    a ← a+1
    afficher(a)
ftant
afficher("on s'arrête là")
```

```
1
2
3
```

Boucle tant-que

- Exécution:



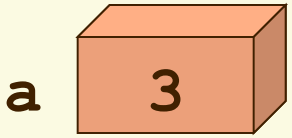
faux

```
a ← 0
tant que (a<3) faire
    a ← a+1
    afficher(a)
ftant
afficher("on s'arrête là")
```

```
1
2
3
```

Boucle tant-que

- Exécution:

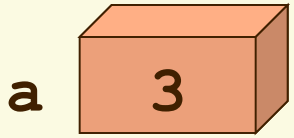



```
a ← 0
tant que (a<3) faire
    a ← a+1
    afficher(a)
ftant
afficher("on s'arrête là")
```

```
1
2
3
on s'arrête là
```

Boucle tant-que

- Exécution:




```
a ← 0
tant que (a<3) faire
    a ← a+1
    afficher(a)
ftant
afficher("on s'arrête là")
```

fin du programme

```
1
2
3
on s'arrête là
```

Boucle tant-que

- Algo ➔ Python

```
a ← 0
tant que (a<3) faire
    a ← a+1
    afficher(a)
ftant
afficher("on s'arrête là")
```

Algo

```
a=0
while (a<3):
    a=a+1
    print a
print "on s'arrête là"
```

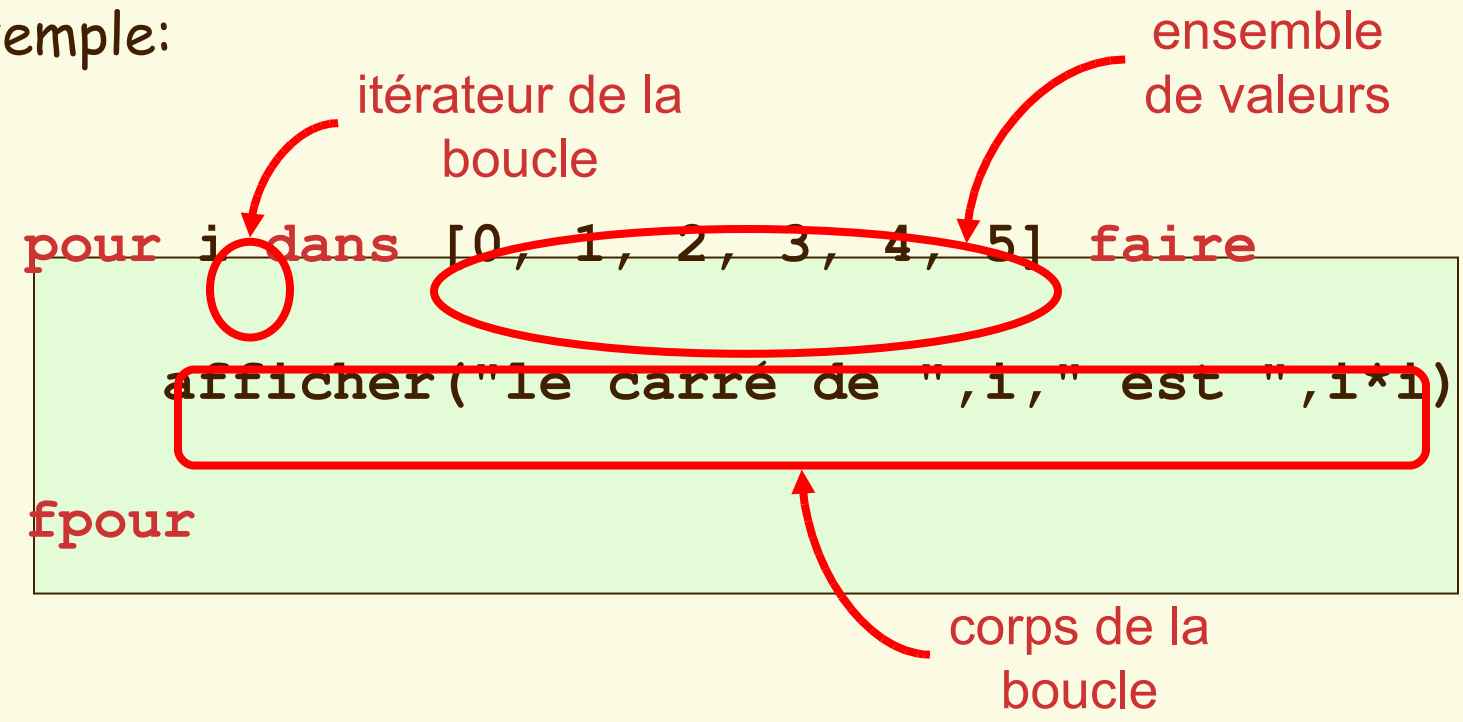
Python

Boucle tant-que

- La condition est évaluée **avant** d'exécuter le corps de la boucle
  - si la condition est fausse dès le départ, on n'entre même pas dans la boucle
- La condition ne sera réévaluée qu'une fois le corps exécuté **entièrement**
  - le fait qu'elle devienne fausse n'interromps pas l'itération en cours
  - si l'itération ne fait rien pour que la condition devienne fausse un iour ➔ boucle infinie

Boucle pour

- La boucle **pour** permet d'itérer sur un ensemble de valeurs
- Exemple:



Boucle pour

- On trouve aussi:

```
pour i variant de 0 à 5 faire
    afficher("le carré de ",i," est ",i*i)
fpour
```

- Ou encore:

```
pour i variant de 0 à 10 par pas de 2 faire
    afficher(i," est pair")
fpour
```

Boucle pour

- Traduction en boucle tant-que:

```
pour i dans [0, 1, 2, 3, 4, 5] faire
    afficher("le carré de ",i," est ",i*i)
fpour
```

```
i ← 0    # première valeur
tant que (i<=5) faire
    afficher("le carré de ",i," est ",i*i)
    i ← i+1    # valeur suivante
ftant
```

Boucle **pour**

- Définir un intervalle en Python
  - explicitement (à la main)
    - `[0, 1, 2, 3, 4, 5]`
    - `[-2, 0, 5, 7, -10, 6, -1]`
  - fonction **`range`**
    - `range(5)` ➔ `[0, 1, 2, 3, 4]`
    - `range(5,10)` ➔ `[5, 6, 7, 8, 9]`
    - `range(0,10,2)` ➔ `[0, 2, 4, 6, 8]`
  - syntaxe: **`range([start,] stop [,step])`**

Boucle pour

- Algo ➔ Python

```
pour i variant de 0 à 10 par pas de 2 faire
    afficher(i," est pair")
fpour
afficher("on s'arrête là")
```

Algo

```
for i in range(0,10,2):
    print i," est pair"
print "on s'arrête là"
```

Python

Récapitulatif

- On a vu les bases essentielles de la prog
  - Variables, opérateurs, expressions
  - Affichage/saisie d'informations
  - Exécution conditionnelle
  - Répétition d'un bloc d'instructions
- Après on verra (entre autres):
  - Informations structurées plus complexes
  - Découpage d'un prog en sous-prog
  - Stockage de données dans des fichiers

Récapitulatif

- Exercice N°1
  - Calculer puis afficher la somme de tous les entiers compris entre 1 et N. La valeur de N n'est pas connue et sera saisie par l'utilisateur.

Récapitulatif

- Exercice N°1

```
# N: borne supérieure
# i: compteur de boucle qui énumère les entiers
# S: somme

N = input('borne supérieure: ')

S = 0
i = 1

while (i<=N):
    S = S+i
    i = i+1

print 'somme =',S
```

Récapitulatif

- Exercice N°2
  - On considère N valeurs entières et on veut calculer leur somme.
    - La valeur de N n'est pas connue et sera saisie par l'utilisateur
    - Les entiers que l'ont veut additionner seront saisis au fur et à mesure par l'utilisateur

Récapitulatif

- Exercice N°2

```
# N: nombre de valeurs
# i: compteur de boucle
# V: ième valeur lue
# S: somme

N = input('nombre de valeurs: ')

S = 0
i = 1

while (i<=N):
    print 'valeur N°',i,' ',
    V = input()
    S = S+V
    i = i+1

print 'somme =',S
```

Récapitulatif

- Exercice N°3
  - Idem que l'exercice 2, mais on veut cette fois:
    - La somme des entiers positifs d'un côté et la somme des entiers négatifs d'un autre côté
    - Le nombre d'entiers positifs et le nombre d'entiers négatifs saisis

Récapitulatif

- Exercice N°3

```
# N : nombre de valeurs
# i : compteur de boucle
# V : ième valeur lue
# SN: somme des entiers négatifs
# SP: somme des entiers positifs
# NN: nombre d'entiers négatifs
# NP: nombre d'entiers positifs

N  = input('nombre de valeurs: ')

SN = 0
SP = 0
NN = 0
NP = 0
i  = 1
```

```
while (i<=N):
    print 'valeur N°',i,' ',
    V = input()

    if (V>=0):
        SP = SP+V
        NP = NP+1
    else:
        SN = SN+V
        NN = NN+1

    i = i+1

print 'somme des',NN,
print 'entiers négatifs =',SN
print 'somme des',NP,
print 'entiers positifs =',SP
```

Plan

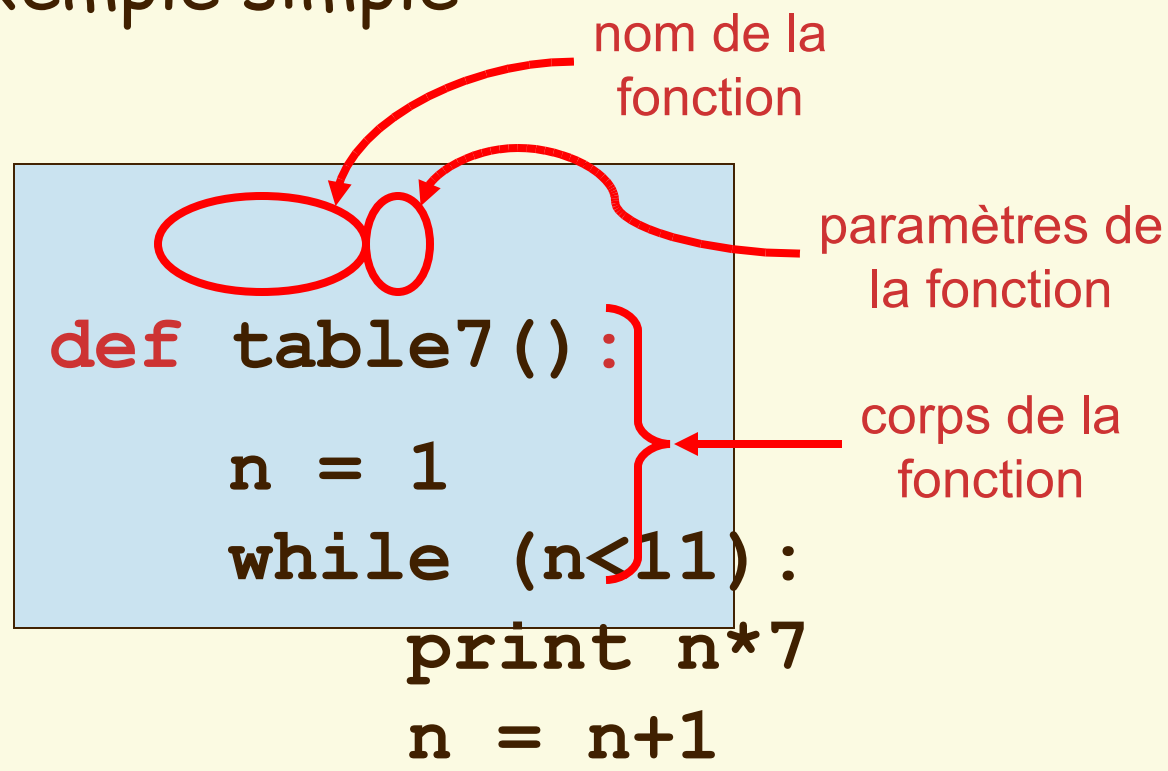
- Instructions élémentaires
  - variables, opérateurs, expressions
  - affichage/lecture de données (`print/input`)
- Instructions de contrôle
  - séquence d'instructions
  - exécution conditionnelle (`if`)
  - instructions composées
- Instructions répétitives
  - boucle (`while,for`)
- Fonctions
- Tableaux

Fonctions

- Décomposition
  - Pour l'instant, nos programmes sont courts
  - Pb: programmes plus complexes ➔ lignes plus nombreuses ➔ programmes illisibles
  - Idée: **décomposer** un problème en **sous-problèmes** plus simples et étudiés séparément
- "Factorisation"
  - Quand une même séquence d'instructions est utilisée plusieurs fois à des endroits différents d'un programme ➔ **instruction personnalisée**

Fonctions

- Exemple simple



Fonctions

- Fonctionnement
  - Une fois définie, votre fonction peut être utilisée comme toute autre fonction du langage
  - Appel de fonction:
    - On mémorise l'endroit du programme où l'on se trouve
    - On va exécuter le corps de la fonction
    - Une fois la fonction exécutée, on revient dans le programme à l'endroit de l'appel et on poursuit en séquence

Fonctions

- Fonctionnement
  - Les appels de fonction peuvent être imbriqués
    - Dans le corps d'une fonction on peut tout à fait faire appel à une autre fonction, et ainsi de suite
    - Exemple:

```
def table7triple():
    print "La table de 7 en triple exemplaire:"
    table7()
    table7()
    table7()
```

Fonctions

- Passage de paramètre(s)

signature de la fonction

paramètre utilisé comme une variable à l'intérieur de la fonction

```
def table(base):
    n = 1
    while (n<11):
        print n * base
        n = n+1
```

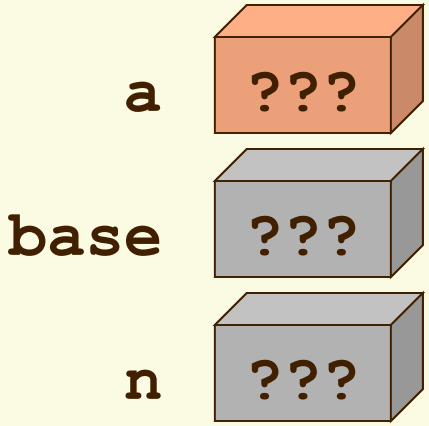
Fonctions

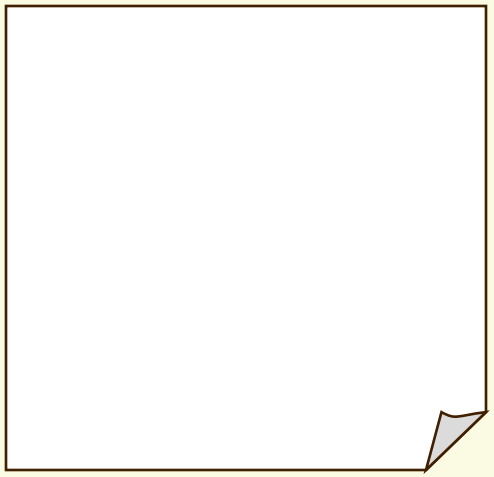
- Exécution:

```
def table(base):
    n = 1
    while (n<11):
        print n * base
        n = n+1
```

début du programme

```
a = input("table pour ?")
table(a)
print "on s'arrête là"
```



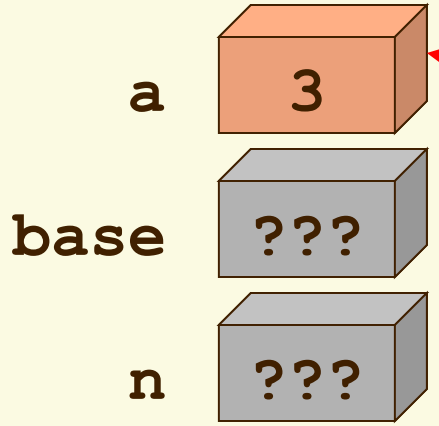


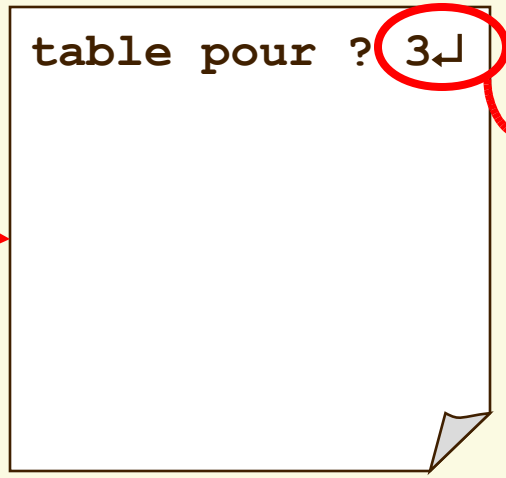
Fonctions

- Exécution:

```
def table(base):
    n = 1
    while (n<11):
        print n * base
        n = n+1
```

```
a = input("table pour ?")
table(a)
print "on s'arrête là"
```



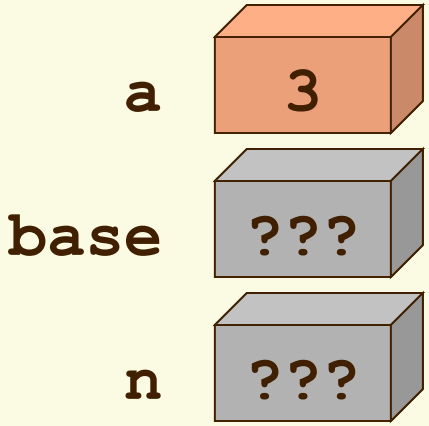


Fonctions

- Exécution:

```
def table(base):
    n = 1
    while (n<11):
        print n * base
        n = n+1

a = input("table pour ?")
table(a)
print "on s'arrête là"
```



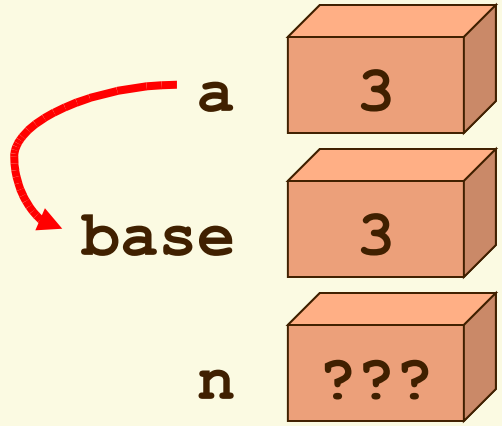

```
table pour ? 3↵
```

Fonctions

- Exécution:

```
def table(base):
    n = 1
    while (n<11):
        print n * base
        n = n+1
```

```
a = input("table pour ?")
table(a)
print "on s'arrête là"
```

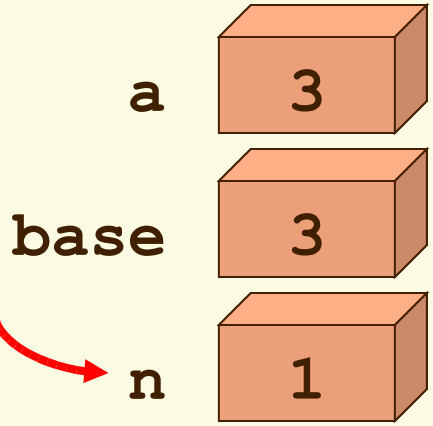



```
table pour ? 3↵
```

Fonctions

- Exécution:

```
def table(base):
    n = 1
    while (n<11):
        print n * base
        n = n+1
```




```
a = input("table pour ?")
table(a)
print "on s'arrête là"
```

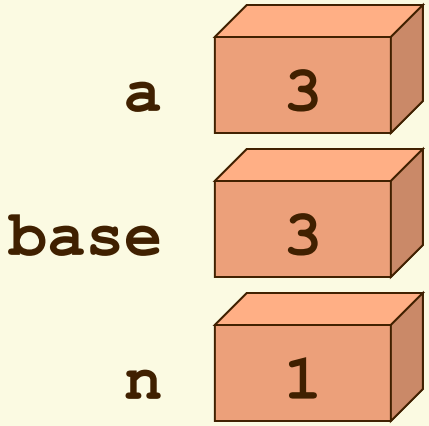
```
table pour ? 3↵
```

Fonctions

- Exécution:

```
def table(base):
    n = 1
    while (n<11):
        print n * base
        n = n+1
```

vrai




```
a = input("table pour ?")
table(a)
print "on s'arrête là"
```

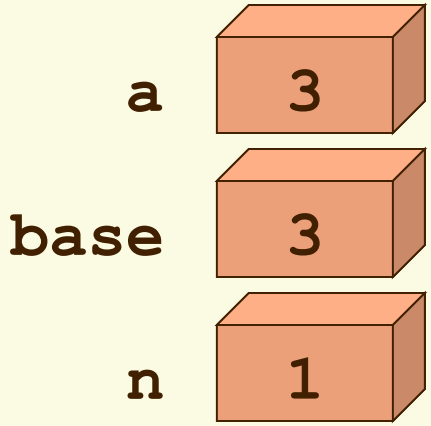
```
table pour ? 3↵
```

Fonctions

- Exécution:

```
def table(base):
    n = 1
    while (n<11):
        print n * base
        n = n+1

a = input("table pour ?")
table(a)
print "on s'arrête là"
```



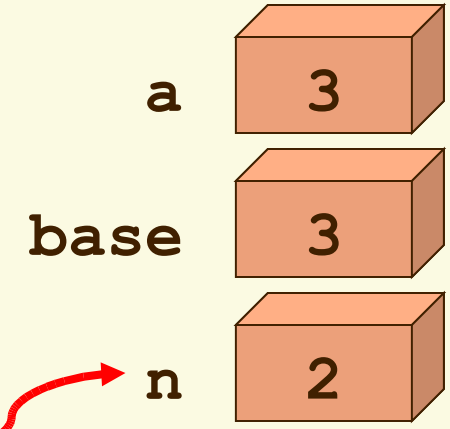

```
table pour ? 3↵
3
```

Fonctions

- Exécution:

```
def table(base):
    n = 1
    while (n<11):
        print n * base
        n = n+1

a = input("table pour ?")
table(a)
print "on s'arrête là"
```



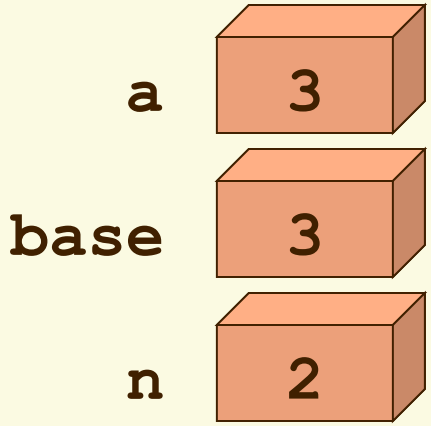

```
table pour ? 3↵
3
```

Fonctions

- Exécution:

```
def table(base):
    n = 1
    while (n<11):
        print n * base
        n = n+1
```

vrai




```
a = input("table pour ?")
table(a)
print "on s'arrête là"
```

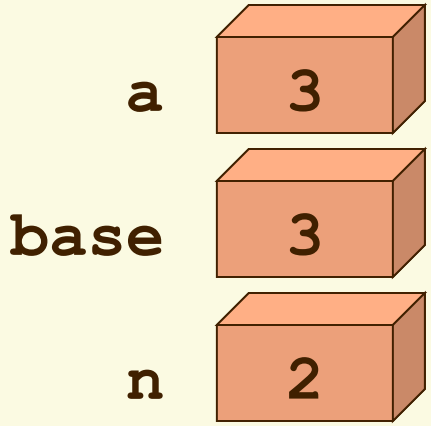
```
table pour ? 3↵
3
```

Fonctions

- Exécution:

```
def table(base):
    n = 1
    while (n<11):
        print n * base
        n = n+1

a = input("table pour ?")
table(a)
print "on s'arrête là"
```



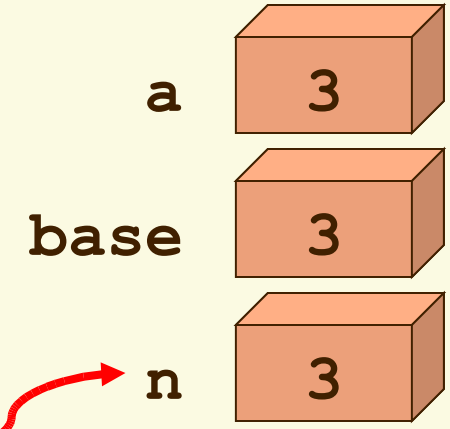

```
table pour ? 3↵
3
6
```

Fonctions

- Exécution:

```
def table(base):
    n = 1
    while (n<11):
        print n * base
        n = n+1

a = input("table pour ?")
table(a)
print "on s'arrête là"
```

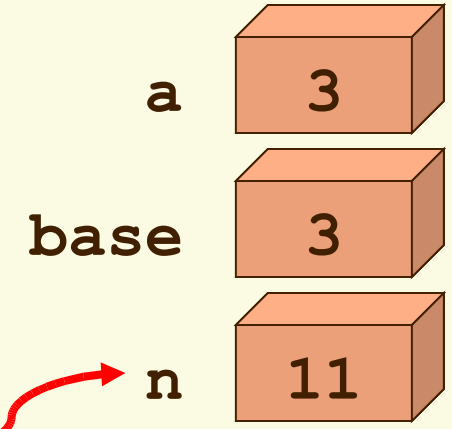



```
table pour ? 3↵
3
6
```

Fonctions

- Exécution:

```
def table(base):
    n = 1
    while (n<11):
        print n * base
        n = n+1
```




```
a = input("table pour ?")
table(a)
print "on s'arrête là"
```

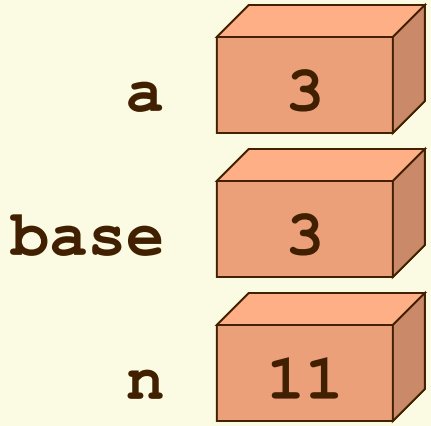
```
table pour ? 3↵
3
6
…
30
```

Fonctions

- Exécution:

```
def table(base):
    n = 1
    while (n<11):
        print n * base
        n = n+1
```

faux




```
a = input("table pour ?")
table(a)
print "on s'arrête là"
```

```
table pour ? 3↵
3
6
…
30
```

Fonctions

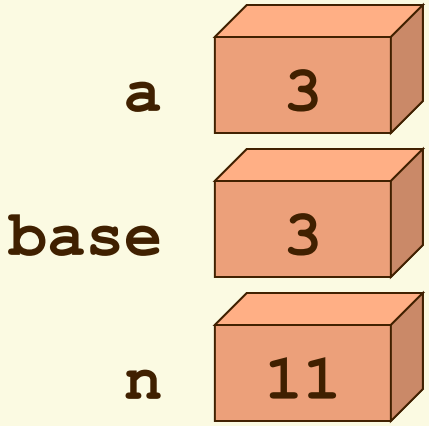
- Exécution:

```
def table(base):
    n = 1
    while (n<11):
        print n * base
        n = n+1
```

fin de la fonction

```
a = input("table pour ?")
table(a)
print "on s'arrête là"
```

appel terminé



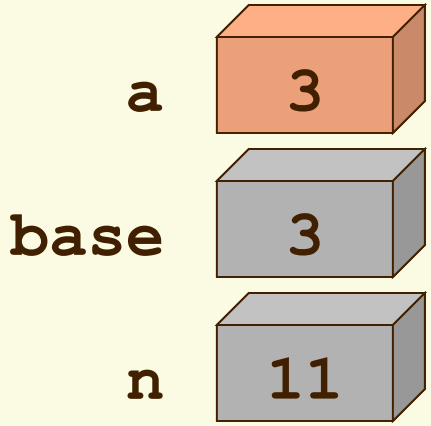

```
table pour ? 3↵
3
6
…
30
```

Fonctions

- Exécution:

```
def table(base):
    n = 1
    while (n<11):
        print n * base
        n = n+1


a = input("table pour ?")
table(a)
print "on s'arrête là"
```




```
table pour ? 3↵
3
6
…
30
on s'arrête là
```

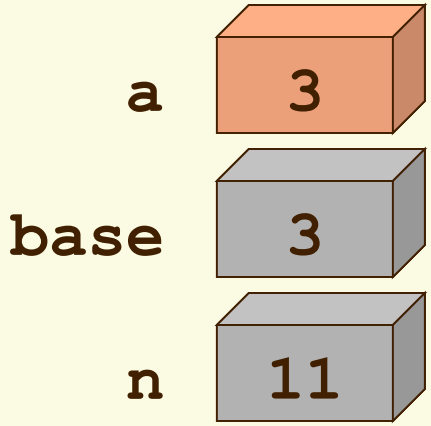
Fonctions

- Exécution:

```
def table(base):
    n = 1
    while (n<11):
        print n * base
        n = n+1

a = input("table pour ?")
table(a)
print "on s'arrête là"
```

fin du programme




```
table pour ? 3↵
3
6
…
30
on s'arrête là
```

Fonctions

- Variables locales
  - **ATTENTION:** les paramètres d'une fonction ainsi que les variables définies à l'intérieur d'une fonction sont **locales** à cette fonction
    - Elles ne sont pas visibles de l'extérieur
    - Elles masquent d'éventuelles variables du programme qui auraient le même identificateur
    - Ces variables sont détruites lorsque la fonction se termine

Fonctions

- Variables locales

```
a,b=3,23

def modif1(z):
    z=z+1

def modif2():
    b=15

print 'a =',a,'et b =',b
modif1(a)
modif2()
print 'a =',a,'et b =',b
```

Toutes les modifications apportées au paramètre z restent **locales** à la fonction

Il s'agit de la déclaration d'une **nouvelle** variable b qui **masque** celle définie au début du programme

```
a = 3 et b = 23
a = 3 et b = 23
```

Fonctions

- Variables locales
  - Les paramètres transmis à une fonction sont donc **passés par valeur**
    - La valeur transmise est **recopiée** dans le paramètre de la fonction
    - Une fonction **ne peut pas modifier** les variables du programme qui l'appelle (**effets de bord**)
    - Si une fonction a besoin d'informations, c'est au programme appelant de les lui fournir en arguments

Fonctions

- Variables globales
  - Le mot-clé **global** permet de donner à une fonction l'accès à une variable du programme

```
b=23

def modif2():
    global b
    b=15

print 'b =',b
modif2()
print 'b =',b
```

Il s'agit de la variable (globale) b définie au début du programme

```
b = 23
b = 15
```

  - **Effets de bord ➔ EVITER** les var. globales

Fonctions

- Résultat d'une fonction
  - Une fonction peut retourner un résultat au programme qui l'a appelée avec un `return`
  - Exemple:

```
def cube(w):
    return w*w*w
```

  - Utilisation:

```
b = cube(9)
print b
```

Fonctions

- Résultat d'une fonction

```
def jury(ue1,ue2,ue3):
    moy = (ue1+ue2+ue3)/3.0

    if (moy>=10):
        if ( (ue1>=8) and (ue2>=8) and (ue3>=8) ):
            decision='admission'
        else:
            decision='redoublement'
    else:
        decision='reorientation'

    return decision

note1 = input("Note de l'UE1: ")
note2 = input("Note de l'UE2: ")
note3 = input("Note de l'UE3: ")
print 'Le jury a décidé:',jury(note1,note2,note3)
```

Un seul return à la fin de la fonction pour plus de lisibilité

Fonctions

- Règles pour le passage de paramètres
  - Les arguments sont donnés **dans l'ordre** des paramètres
  - On donne **une valeur pour chacun** des paramètres
- Python autorise une certaine souplesse
  - On peut omettre certains paramètres
    - ➔Poly voir "valeurs par défaut pour les paramètres"
  - On peut les fournir dans un ordre différent
    - ➔Poly voir "arguments avec étiquettes"

Fonctions

- Une fonction doit avoir été définie avoir de pouvoir être utilisée
  - Les fonctions sont déclarées au début du script
  - Le "**programme principal**" se trouve à la fin du script
    - ➔Fonction particulière `__main__`
    - ➔Pour comprendre ce que fait un script il faut commencer par la fin (i.e. fonction `__main__`)

Récapitulatif

- Exercice N°1
  - Écrire la fonction **`max`** calculant le maximum de deux entiers passés en paramètres.
  - NB: prenez l'habitude d'écrire un programme de test permettant de valider votre fonction.

Récapitulatif

- Exercice N°1

```
# Maximum de deux entiers

def max(a,b):
    plusgrand = 0

    if (a>b):
        plusgrand = a
    else:
        plusgrand = b

    return plusgrand

# ----- Programme principal -----

val1 = input('Première valeur: ')
val2 = input('Deuxième valeur: ')
print 'Le maximum de',val1,'et de',val2,'est',max(val1,val2)
```

Récapitulatif

- Exercice N°2
  - Écrire la fonction `fact` prenant un entier en paramètre et calculant sa factorielle
  - Rappel: `N! = 1*2*3*4*…*N`

Récapitulatif

- Exercice N°2

```
# Factorielle: version itérative

def fact(n):
    resultat = 1
    i = 1
    while (i<=n):
        resultat = resultat*i
        i = i+1
    return resultat

# ----- Programme principal -----

n = input('Valeur de n ? ')
print 'n! =',fact(n)
```

Récapitulatif

- Exercice N°2

```
# Factorielle: version récursive

def fact(n):
    if (n==0):
        return 1
    else:
        return n * fact(n-1)


# ----- Programme principal -----

n = input('Valeur de n ? ')
print 'n! =',fact(n)
```

Condition d'arrêt de la récursivité

Récursivité: la fonction fact effectue un appel vers elle-même

Plan

- Instructions élémentaires
  - variables, opérateurs, expressions
  - affichage/lecture de données (`print/input`)
- Instructions de contrôle
  - séquence d'instructions
  - exécution conditionnelle (`if`)
  - instructions composées
- Instructions répétitives
  - boucle (`while,for`)
- Fonctions
- Tableaux

Tableaux

- En prog on est souvent amené à traiter des séries d'informations
  - Liste des notes d'un étudiant
  - Liste des noms de prof
  - Suite $a_1, a_2, \ldots, a_n$ comme en maths
  - Vecteurs, matrices

Tableaux

- Idée:
  - Plutôt que de d'utiliser autant de variables que d'éléments à stocker (a,b,c,…) on va définir une seule variable pouvant stocker plusieurs éléments: un **tableau**
  - Les cases de ce tableau seront **numérotées à partir de 0**
    - La 1ère case du tableau `tab` sera notée `tab[0]`
    - La 2ème case du tableau `tab` sera notée `tab[1]`
  - Chaque case se comporte comme une variable

Tableaux

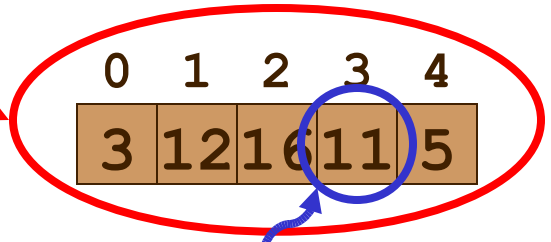
- Exemple

```
tableau = [3,12,16,11,5]

somme = 0
i = 0
while (i<5):
    print 'Valeur',i,'=',tableau[i]
    somme = somme + tableau[i]
    i = i+1

print 'Somme =',somme
```

Tableau de 5 entiers



tableau[3]

Ceci se lit "**tableau indice i**"

```
Valeur 0 = 3
Valeur 1 = 12
Valeur 2 = 16
Valeur 3 = 11
Valeur 4 = 5
Somme = 47
```

Tableaux

- Remarques
  - La notation `[3,12,16,11,5]` est également valable lors de la saisie (fonction **`input()`**)
  - Si l'indice ne désigne pas une case du tableau, vous obtenez une erreur
    - Ex: si le tableau **`tab`** contient 5 cases (numérotées de 0 à 4), **`tab[5]`** "**sort du tableau**"
  - La plupart des langages imposent que tous les éléments du tableau soient du même type
    - Sauf Python… mais ce n'est pas une raison suffisante pour en abuser ➔ prenez de bonnes habitudes

Tableaux

- Remarques
  - `[]` définit un tableau vide (i.e. ne contenant aucune case)
  - Si `tab` désigne un tableau d'entiers, `tab.append(10)` ajoute une nouvelle case contenant l'entier 10 à la fin du tableau `tab`

```
>>> tab = [2,4,6,8]
>>> tab.append(10)
>>> tab
[2, 4, 6, 8, 10]
```

Tableaux

- Les cases d'un tableau peuvent contenir n'importe quel type de données
  - Ex 1: des chaînes de caractères

```
>>> jours = ['lundi','mardi','jeudi','mercredi','vendredi','samedi']
>>> jours
['lundi', 'mardi', 'jeudi', 'mercredi', 'vendredi', 'samedi']
>>> temp = jours[2]
>>> jours[2] = jours[3]
>>> jours[3] = temp
>>> jours
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi']
>>> jours.append('dimanche')
>>> jours
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche']
```

Tableaux

- Les cases d'un tableau peuvent contenir n'importe quel type de données
  - Ex 2: des tableaux

```
>>> matrice = [[1,2,3],
               [4,5,6],
               [7,8,9]]
>>> matrice
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> for i in range(3):
        print 'ligne',i,':',
        for j in range(3):
            print matrice[i][j],',',
        print '<fin>'
ligne 0 : 1 , 2 , 3 , <fin>
ligne 1 : 4 , 5 , 6 , <fin>
ligne 2 : 7 , 8 , 9 , <fin>
```

Tableaux

- La fonction **`len()`** permet de connaître le nombre d'éléments dans un tableau

```
>>> len(jours)
7
>>> len(matrice)
3
>>> len(matrice[0])
3
>>> for i in range(len(matrice)):
        ligne = matrice[i]
        for j in range(len(ligne)):
            print ligne[j],
        print ''

1 2 3
4 5 6
7 8 9
```

ligne

Équivalent à **`matrice[i][j]`**

Tableaux

- Conclusion
  - On a vu:
    - Définir un tableau
    - Parcourir un tableau (avec des boucles)
    - Fonctions `len()` et `append()`
  - Python va beaucoup plus loin avec les tableaux
    - Ce n'est pas portable à d'autres langages (C,...)
    - C'est relativement complexe pour les débutants
    - Ça s'éloigne de la notion classique de tableau

Récapitulatif

- Exercice N°1
  - Écrire un programme qui demande à l'utilisateur de saisir un tableau de valeurs et qui calcule:
    - la somme de ces valeurs
    - la moyenne de ces valeurs
  - Vous ferez ceci en deux étapes:
    a) sans utiliser de fonction
    b) en définissant deux fonctions `sommeTab()` et `moyenneTab()` prenant chacune un tableau en paramètre et renvoyant un nombre réel

Récapitulatif

- Exercice N°1-a

```
# Somme et moyenne des valeurs d'un tableau
# 1ere version: sans fonction

tab = input('Tableau de valeurs: ')

somme = 0
for i in range(len(tab)):
    somme = somme + tab[i]

moyenne = somme / len(tab)

print 'somme   =',somme
print 'moyenne =',moyenne
```

```
Tableau de valeurs: [23,5,-4,3]
somme   = 27
moyenne = 6
```

Récapitulatif

- Exercice N°1-b

```
# Somme et moyenne des valeurs d'un tableau
# 2eme version: avec fonction

def sommeTab(tablo):
    somme = 0
    for i in range(len(tablo)):
        somme = somme + tablo[i]
    return somme

def moyenneTab(t):
    return sommeTab(t)/len(t)

# ----- Programme principal -----

tab = input('Tableau de valeurs: ')
print 'somme   =',sommeTab(tab)
print 'moyenne =',moyenneTab(tab)
```

Récapitulatif

- Exercice N°2
  - Nous allons maintenant travailler sur des matrices, c'est-à-dire des tableaux à deux dimensions
    - Fonction de création d'une matrice vide (que des 0) à partir du nombre de lignes et du nombre de colonnes
    - Fonction d'addition de deux matrices
    - Fonction de multiplication de deux matrices
    - Programme de test

Récapitulatif

- Exercice N°2

```
# Programme sur les matrices

def creerMatrice(l,c):
    # Fonction créant une matrice vide (i.e. toutes
    # les valeurs à 0) contenant l lignes et c colonnes

    matrice = []
    for i in range(l):
        ligne = []
        for j in range(c):
            ligne.append(0.0)
        matrice.append(ligne)

    return matrice
```

Récapitulatif

- Exercice N°2 (suite)

```
# Programme sur les matrices (suite)

# Quelques fonctions qui nous seront bien utiles pour la suite

def nbLignes(m):
    # Fonction retournant le nombre de lignes de la
    # matrice passée en paramètre

    return len(m)

def nbColonnes(m):
    # Fonction retournant le nombre de colonnes de
    # la matrice passée en paramètre

    return len(m[0])
```

Récapitulatif

- Exercice N°2 (suite)

```
# Programme sur les matrices (suite)

def additionnerMatrices(a,b):
    # Fonction additionnant les deux matrices a et b
    # passées en paramètres. Si les nombres de lignes
    # et de colonnes des deux matrices ne correspondent
    # pas, cette fonction se contente simplement
    # d'afficher un message d'erreur

    if ((nbLignes(a)!=nbLignes(b)) or (nbColonnes(a)!=nbColonnes(b))):
        print 'ERREUR: les deux matrices ne sont pas de même taille !'
    else:
        res = creerMatrice(nbLignes(a),nbColonnes(a))
        for i in range(nbLignes(a)):
            for j in range(nbColonnes(a)):
                res[i][j] = a[i][j] + b[i][j]
        return res
```

Récapitulatif

- Exercice N°2 (suite)

```
# Programme sur les matrices (suite)

def multiplierMatrices(a,b):
    # Fonction multipliant...

    if (nbLignes(b) != nbColonnes(a)):
        print 'ERREUR: le nombre de lignes de la seconde matrice ne ',
        print 'correspond pas au nombre de colonnes de la première !'
    else:
        res = creerMatrice(nbLignes(a),nbColonnes(b))
        for i in range(nbLignes(a)):
            for j in range(nbColonnes(b)):
                temp = 0.0
                for k in range(nbColonnes(a)):
                    temp = temp + a[i][k] * b[k][j]
                res[i][j] = temp
        return res
```

Tuples

- En prog on a parfois besoin de construire des informations composées à partir de données plus élémentaires ➔ **structures de données**
  - Idée: regrouper plusieurs variables **de types différents** ➔ les **champs** de la structure
  - Python permet ceci via les tuples
    - Ex: `('Opel','Corsa',5,(42,'AFQ',57))`

Tuples

- Comment manipuler des tuples en Python ?
  - Composition
    - `immat = 42,'AFQ',57`
    - `voiture = 'Opel','Corsa',5,immat`
    - `date = jour,mois,annee`
  - Décomposition
    - `marque,modele,puiss,plaque = voiture`
    - `jour,mois,annee = (23,9,1971)`
  - Accès au i^{ème} champ (lecture uniquement)
    - `puissance = voiture[2]  # 3ème champ`

Plan

- Instructions élémentaires
  - variables, opérateurs, expressions
  - affichage/lecture de données (`print`/`input`)
- Instructions de contrôle
  - séquence d'instructions
  - exécution conditionnelle (`if`)
  - instructions composées
- Instructions répétitives
  - boucle (`while`,`for`)
- Fonctions
- Tableaux