Algorithmique I

Module : Fondement de la programmation

Douglas Teodoro

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts
Western Switzerland

2018-2019

Sommaire

Objective

- Revision de fonction et de procédure
- Correction du CC
- Décomposition fonctionnelle descendante

Sommaire

RAPPEL

Les fonctions

Fonctions

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

Fonction ≡ expression

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

Exemple

```
y = sin(x)
```

renvoie la valeur du sinus de x

nom	sin
paramètres	x:numerique
retourne	numerique

Les procédures

Procédures

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

Procédure ≡ instruction

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

Exemple

```
print(x, y, z)
```

affiche les valeurs de x, y et z

nom print

paramètres x:entier, y:reel, z:numerique

Déclaration et définition - dans un algorithme

Fonction

Algorithme : declare_fonction

```
1 Fonction fonction(parametres) : type
2     bloc d'instructions
3     ...
4
5     retourner...
6
  Données : ...
  Résultat : ...
  // séquence d'instructions
7 ...
```

Procedure

Algorithme : declare_procedure

```
1 Procédure procedure(parametres)
2     bloc d'instructions
3     ...
4
5
  Données : ...
  Résultat : ...
  // séquence d'instructions
6 ...
```

Déclaration et définition - en Python

Python - Fonction

```
# séquence d'opérations
def nom_fonction(parametres) :
      # bloc d'instructions
      ...
      return ...
```

Python - Procédure

```
# séquence d'opérations
def nom_procedure(parametres) :
      # bloc d'instructions
      ...
      # pas de return
```

Déclaration et définition - dans un algorithme

Algorithme : puissance

```
1  Fonction calcule_puissance(x:numerique, n:entier) : numerique
2      resultat=1:numerique
3      compteur = 1
4      tant que compteur ≤ n faire
5          resultat = resultat * x
6          compteur = compteur + 1
7      retourner resultat
8  Procédure affiche_puissance(x:numerique,n:entier,y:numerique)
9      afficher ``la puissance de '',x,``'',n,`èst'',y
10 Données : x:numerique, n:entier
   Résultat : la valeur de la puissance x^n
11 resultat:numerique                    // déclaration des variables
12 afficher ``x,n?''                        // séquence d'opérations
13 saisir x,n
14 resultat = calcule_puissance(x,n)
15 affiche_puissance(x,n,resultat)
```

- lignes 10 : entrée et sortie du algorithme
- lignes 11 - 15 : programme principal
- lignes 1 - 7 : fonction qui calcule la puissance
- lignes 8 - 9 : procédure qui affiche le résultat

Variables locales et globales

Exemple d'algorithme

1. **********\n
2. **********\n
3. **********\n
4. **********\n
5. **********\n
6. **********\n
7. **********\n
8. **********\n
9. **********\n
10. **********\n

Algorithme : lignes

```
1  Procédure repete_caractere()
       // déclaration des variables
2      i:entier
3      pour i ← 1 à 20 faire
4          afficher "*"
   Données :
   Résultat : affiche 10 lignes de 20 caractères
   // déclaration des variables
5  i:entier
   // séquence d'opérations
6  afficher "Voici 10 lignes de 20 caractères"
7  pour i ← 1 à 10 faire
8      repete_caractere()
9      afficher "\n"
10 afficher "Le programme est terminé"
```

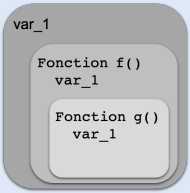
La portée - Scope

L'exemple de la procédure `repete_caractere()` met en évidence l'utilisation de deux variables de même nom (`i`) : l'une dans le programme principal `lignes`, l'autre dans le sous-programme `repete_caractere`

L'endroit où les variables sont déclarées est très important : selon cet endroit, les variables ont une portée différente

Portée (*Scope*)

La portée d'une variable est sa visibilité au sein des différentes parties du programme : peut-on accéder à son contenu à cet endroit ?



La portée - Scope

- Le cas général dit qu'une variable n'est visible et accessible par défaut que dans le bloc d'instructions où elle a été **déclarée**
- Une variable déclarée dans un sous-programme sous les mots-clés **Procédure** ou **Fonction** ne pourra dans ce cas qu'être lisible et modifiable uniquement dans ce sous-programme
- Idem pour le programme principal déclaré après le mots-clé **Données** et **Résultat** : une variable déclarée dans cet endroit ne sera accessible que par celui-ci par défaut

Algorithme : portee

```
1 Procédure p(parametres)
2     // déclaration des variables
3     var:type
      // bloc d'instructions
      // ...
4 Fonction f(parametres):type
5     // déclaration des variables
6     var:type
      // bloc d'instructions
      // ...
7     retourner
Données :
Résultat :
8   // déclaration des variables
9   var:type
    // bloc d'instructions
    // ...
```

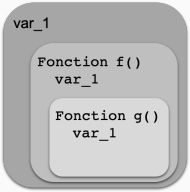
Variables locales

Variables locales

Les variables accessibles uniquement par le programme ou sous-programme dans lesquels elles sont déclarées, sont appelées des variables locales

Les variables locales de **même nom** n'ont aucun rapport entre elles : elles sont totalement indépendantes les unes des autres et aucune interaction n'est possible

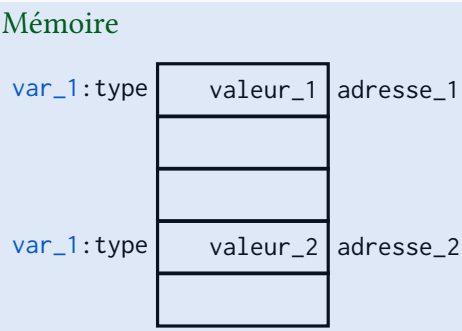
- les variables locales peuvent donc parfaitement porter un même nom
- toutes les variables que on a rencontrées jusqu'à présent sont des variables locales



Variables locales

Du côté de la mémoire, le contenu de ces deux variables est cloisonné et distinct, à des adresses différentes

- La variable `i` de `repete_caractere()` n'est pas du tout la même que la variable `i` du programme `lignes`
- Il n'y a aucun risque d'accéder à la valeur ou de modifier celle-ci par accident de l'un vers l'autre programme



Variables locales

i :entier	1..20	0x000000
distance :caracters	"km"	0x000001
		0x000002
		0x000003
i :entier	1..10	0x000004
		0x000005
distance :entier	10	0x000006
		0x000007

Variables locales

- Variable visible dans le bloc d'instructions qui l'a créée

Algorithme : variables_locales

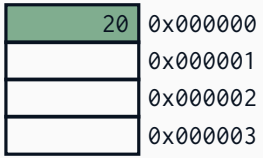
```
1 Procédure affiche_local()
2     i:entier
3     i=10
4     afficher i
  Données : données
  Résultat : résultat
5 i=20:entier
6 affiche_local()
7 afficher i
```

Variables locales

- Variable visible dans le bloc d'instructions qui l'a créée
- Elle a une durée de vie depuis sa **déclaration** jusqu'à la **fin du bloc** d'instructions qui l'a déclarée

Algorithme : variables_locales

```
1 Procédure affiche_local()
2     i:entier
3     i=10
4     afficher i
  Données : données
  Résultat : résultat
5 i=20:entier
6 affiche_local()
7 afficher i
```

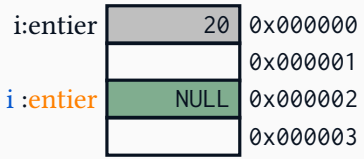


Variables locales

- Variable visible dans le bloc d'instructions qui l'a créée
- Elle a une durée de vie depuis sa **déclaration** jusqu'à la **fin du bloc** d'instructions qui l'a déclarée
- Les variables définit à l'intérieur du corps d'une fonction ne sont accessibles qu'à la fonction elle-même
  - Elle n'a pas de valeur initiale

Algorithme : variables_locales

```
1 Procédure affiche_local()
2     i:entier
3     i=10
4     afficher i
  Données : données
  Résultat : résultat
5 i=20:entier
6 affiche_local()
7 afficher i
```

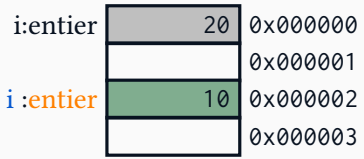


Variables locales

- Variable visible dans le bloc d'instructions qui l'a créée
- Elle a une durée de vie depuis sa **déclaration** jusqu'à la **fin du bloc** d'instructions qui l'a déclarée
- Les variables définit à l'intérieur du corps d'une fonction ne sont accessibles qu'à la fonction elle-même
  - Elle n'a pas de valeur initiale
- Attention : une variable locale est prioritaire par rapport à une variable globale de même nom
  ex. : `affiche_local()` affiche 10 (et pas 20)

Algorithme : variables_locales

```
1 Procédure affiche_local()
2     i:entier
3     i=10
4     afficher i
  Données : données
  Résultat : résultat
5 i=20:entier
6 affiche_local()
7 afficher i
```



Variables locales

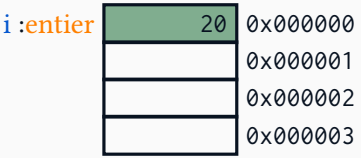
- Variable visible dans le bloc d'instructions qui l'a créée
- Elle a une durée de vie depuis sa **déclaration** jusqu'à la **fin du bloc** d'instructions qui l'a déclarée
- Les variables définit à l'intérieur du corps d'une fonction ne sont accessibles qu'à la fonction elle-même
  - Elle n'a pas de valeur initiale
- Attention : une variable locale est prioritaire par rapport à une variable globale de même nom
  ex. : `affiche_local()` affiche 10 (et pas 20)

Algorithme : variables_locales

```
1 Procédure affiche_local()
2     i:entier
3     i=10
4     afficher i
```

Données : données
Résultat : résultat

```
5 i=20:entier
6 affiche_local()
7 afficher i
```



Variables globales

Variables globales

De fois, il est très pratique de pouvoir accéder à une variable depuis n'importe quel endroit du programme, qu'il soit principal ou un sous-programme : ce type de variable s'appelle une variable globale

- La portée d'une telle variable s'étendrait à tout le code
- Étant globale, elle est accessible de partout, tant en accès (lecture du contenu) qu'en modification (affectation d'une nouvelle valeur)
- Les variables globales existent tant en algorithmique que dans la plupart des langages

Variables globales - Déclaration

Une variable globale est déclarée en dehors des sous-programmes et du programme principal, avant ceux-ci, c'est-à-dire en premier dans l'algorithme :

Algorithme : variables_globales

```
1  // déclarations de variables globales
2  global var_glob_1:type
3  global var_glob_2 = valeur:type
4  Procédure p(parametres)
       // séquence d'opérations
5  Fonction f(parametres):type
       // séquence d'opérations
6      retourner
   Données : données
   Résultat : résultat
   // séquence d'opérations
```

Python

```
global var_glob
# programme et sous-programme
# ...
```

Variables globales - Déclaration

Une variable globale est déclarée en dehors des sous-programmes et du programme principal, avant ceux-ci, c'est-à-dire en premier dans l'algorithme :

Algorithme : variables_globales

```
1  // déclarations de variables globales
2  global var_glob_1:type
3  global var_glob_2 = valeur:type
4  Procédure p(parametres)
       // séquence d'opérations
5  Fonction f(parametres):type
       // séquence d'opérations
6      retourner
   Données : données
   Résultat : résultat
   // séquence d'opérations
```

Python

```
global var_glob
# programme et sous-programme
# ...
```

Variables globales - Portée

Algorithme : variables_globales

```
// déclarations de variables globales
1 global i=1:entier
2 Procédure affiche_global()
3 |   afficher i
4 |   i = i + 1
  Données :
  Résultat :
// séquence d'opérations
5 afficher i
6 i = i + 1
7 affiche_global()
8 afficher i
```

Séquence d'opérations :

1. `i = 1`

Variables globales - Portée

Algorithme : variables_globales

```
// déclarations de variables globales
1 global i=1:entier
2 Procédure affiche_global()
3 |   afficher i
4 |   i = i + 1
  Données :
  Résultat :
  // séquence d'opérations
5 afficher i
6 i = i + 1
7 affiche_global()
8 afficher i
```

Séquence d'opérations :

1. `i = 1`
2. `affiche 1`

Variables globales - Portée

Algorithme : variables_globales

```
// déclarations de variables globales
1 global i=1:entier
2 Procédure affiche_global()
3 |   afficher i
4 |   i = i + 1
  Données :
  Résultat :
  // séquence d'opérations
5 afficher i
6 i = i + 1
7 affiche_global()
8 afficher i
```

Séquence d'opérations :

1. `i = 1`
2. `affiche 1`
3. `i = 2`

Variables globales - Portée

Algorithme : variables_globales

```
// déclarations de variables globales
1 global i=1:entier
2 Procédure affiche_global()
3 |   afficher i
4 |   i = i + 1
  Données :
  Résultat :
  // séquence d'opérations
5 afficher i
6 i = i + 1
7 affiche_global()
8 afficher i
```

Séquence d'opérations :

1. `i = 1`
2. `affiche 1`
3. `i = 2`
4. `affiche 2`

Variables globales - Portée

Algorithme : variables_globales

```
// déclarations de variables globales
1 global i=1:entier
2 Procédure affiche_global()
3     afficher i
4     i = i + 1
Données :
Résultat :
// séquence d'opérations
5 afficher i
6 i = i + 1
7 affiche_global()
8 afficher i
```

Séquence d'opérations :

1. `i = 1`
2. `affiche 1`
3. `i = 2`
4. `affiche 2`
5. `i = 3`

Variables globales - Portée

Algorithme : variables_globales

```
// déclarations de variables globales
1 global i=1:entier
2 Procédure affiche_global()
3 |   afficher i
4 |   i = i + 1
  Données :
  Résultat :
  // séquence d'opérations
5 afficher i
6 i = i + 1
7 affiche_global()
8 afficher i
```

Séquence d'opérations :

1. `i = 1`
2. `affiche 1`
3. `i = 2`
4. `affiche 2`
5. `i = 3`
6. `affiche 3`

Variables globales - Remarques

La variable globale amène quatre remarques :

- Elle n'est déclarée qu'une seule fois pour l'intégralité du programme
- Elle permet indirectement de passer des valeurs aux sous-programmes qui l'utilisent
- Comme corollaire, ne donnez jamais le même nom à une variable locale et globale
- On les utilise avec beaucoup de modération : il y a un coût de mémoire et en plus on risque par accident, pensant à une variable locale, d'en modifier certaines sans y prendre garde

Exercice - Variables locales/globales I

Question : Quel est la sortie de l'algorithme suivant :

A)
```
*
**
***
****
*****
```

B)
```
*****
*****
*****
*****
*****
```

C)
```
#
##
###
####
#####
```

D)
```
#####
#####
#####
#####
#####
```

Algorithme : afficher_x

```
global c
c="#"

def repete_car():
    for i in range(1,nbcar+1):
        print(c,end=" ")
    print()

c="*"
for nbcar in range(1,6):
    repete_car()
```

Exercice - Variables locales/globales I

Question : Quel est la sortie de l'algorithme suivant :

A)
```
*
**
***
****
*****
```

B)
```
*****
*****
*****
*****
*****
```

C)
```
#
##
###
####
#####
```

D)
```
#####
#####
#####
#####
#####
```

Algorithme : afficher_x

```
global c
c="#"

def repete_car():
    for i in range(1,nbcar+1):
        print(c,end=" ")
    print()

c="*"
for nbcar in range(1,6):
    repete_car()
```

Exercice - Variables locales/globales II

Question : Quel est la sortie de l'algorithme suivant :

A)
```
*
**
***
****
*****
```

B)
```
*****
*****
*****
*****
*****
```

C)
```
#
##
###
####
#####
```

D)
```
#####
#####
#####
#####
#####
```

Algorithme : afficher_x

```
global c

def repete_car(nbcar,c):
    for i in range(1,nbcar+1):
        print(c,end=" ")
    print()

c="*"
for nb in range(1,6):
    repete_car(nb,"#")
```

Exercice - Variables locales/globales II

Question : Quel est la sortie de l'algorithme suivant :

A)
```
*
**
***
****
*****
```

B)
```
*****
*****
*****
*****
*****
```

C)
```
#
##
###
####
#####
```

D)
```
#####
#####
#####
#####
#####
```

Algorithme : afficher_x

```
global c

def repete_car(nbcar,c):
    for i in range(1,nbcar+1):
        print(c,end=" ")
    print()

c="*"
for nb in range(1,6):
    repete_car(nb,"#")
```

Exercice - Variables locales/globales III

Question : Quel est la sortie de l'algorithme suivant :

A)
```
*
**
***
****
*****
```

B)
```
*****
*****
*****
*****
*****
```

C) error

D)
```
#####
#####
#####
#####
#####
```

Algorithme : afficher_x

```
def repete_car(c):
    nbcar=5
    for i in range(1,nbcar+1):
        print(c,end=" ")
    print()

c="*"
for nbcar in range(1,6):
    repete_car("#")
```

Exercice - Variables locales/globales III

Question : Quel est la sortie de l'algorithme suivant :

A)
```
*
**
***
****
*****
```

B)
```
*****
*****
*****
*****
*****
```

C) error

D)
```
#####
#####
#####
#####
#####
```

Algorithme : afficher_x

```
def repete_car(c):
    nbcar=5
    for i in range(1,nbcar+1):
        print(c,end=" ")
    print()

c="*"
for nbcar in range(1,6):
    repete_car("#")
```

Exercice - Variables locales/globales IV

Question : Quel est la sortie de l'algorithme suivant :

A)
```
*
**
***
****
*****
```

B)
```
*****
*****
*****
*****
*****
```

C) error

D)
```
#####
#####
#####
#####
#####
```

Algorithme : afficher_x

```
global c
def repete_car():
    nbcar=5
    for i in range(1,nbcar+1):
        print(c,end=" ")
    print()


for nbcar in range(1,6):
    repete_car()
c="*"
```

Exercice - Variables locales/globales IV

Question : Quel est la sortie de l'algorithme suivant :

A)
```
*
**
***
****
*****
```

B)
```
*****
*****
*****
*****
*****
```

C) error

D)
```
#####
#####
#####
#####
#####
```

Algorithme : afficher_x

```
global c
def repete_car():
    nbcar=5
    for i in range(1,nbcar+1):
        print(c,end=" ")
    print()


for nbcar in range(1,6):
    repete_car()
c="*"
```

La passage de paramètres

Syntaxe - Passage des paramètres pour les procédures et fonctions

La syntaxe de passage des paramètres en pseudocode :

Algorithmique

```
Procédure proc_nom(p1:type, p2:type, ..., pn:type)
# séquence d'instructions
# ...
```

Algorithmique

```
Fonction fonc_nom(p1:type, p2:type, ..., pn:type)
# séquence d'instructions
# ...
retourner ...
```

Syntaxe - Passage des paramètres pour les procédures et fonctions

La syntaxe de passage des paramètres en Python :

Python

```
def proc_nom(p1, p2, ..., pk=valeur, ..., pn=valeur):
# séquence d'instructions
# ...
```

Python

```
def proc_nom(p1, p2, ..., pk=valeur, ..., pn=valeur):
# séquence d'instructions
# ...
return ...
```

Passage de paramètres

- Les paramètres passés à un sous-programme sont généralement des variables locales au programme ou sous-programme l'appelant
  - ils ne portent pas forcément le même nom
- Ils sont récupérés au sein du sous-programme comme des variables locales au sous-programme

Algorithme : pass_param_local

```
1 Procédure affiche_lignes(nb_car:entier, type_car:caracters)
2     pour i ← 1 à nb_car faire
3         afficher type_car
4
```

Données :
Résultat : caractère affiché

```
// séquence d'instructions
5 i=10:entier
6 car="#":caracters
7 affiche_lignes(i, car)
```

Passage de paramètres par valeur ou par référence

Il y a deux méthodes pour passer des variables en paramètre dans une fonction ou procèdure : le passage par valeur et le passage par référence

Passage par valeur

La valeur de l'expression passée en paramètre est copiée dans une variable locale : c'est cette variable qui est utilisée pour faire les calculs dans la fonction appelée

Passage par référence

La passage par référence consiste à passer non plus la valeur des variables comme paramètre, mais à passer les variables elles-mêmes (ou des références en mémoire à ceux-ci)

Passage par valeur

- Le contenu de l'expression passée en paramètre est copié dans la variable locale
- Aucune modification de la variable locale dans la fonction appelée ne modifie la variable passée en paramètre, parce que ces modifications ne s'appliquent qu'à une copie de cette dernière

Python : objets immutables (int, float, string, tuple, bytes)

Passage par référence

- Il n'y a plus de copie et de variable locale avec la passage par référence
- Toute modification du paramètre dans la fonction appelée entraîne la modification de la variable passée en paramètre

Python : objets mutables (list, dict, set, byte array)

Avantages et inconvénients des deux méthodes

- Les passages par références sont plus rapides et plus économes en mémoire que les passages par valeur, puisque les étapes de la création de la variable locale et la copie de la valeur ne sont pas faites
- Il faut donc éviter les passages par valeur dans les cas d'appels récursifs de fonction ou de fonctions travaillant avec des grandes structures de données (matrices par exemple)
- Les passages par valeurs permettent d'éviter de détruire par mégarde les variables passées en paramètre

Exercice - Passage de paramètres I

Question : Quel est la sortie de l'algorithme suivant :

A) 1 2 1 2
B) 1 2 2 1
C) 2 1 1 2
D) 2 1 2 1

Algorithme : afficher_x

```
def print_car(x,y):
    print(x,y,end=" ")

x = 1 # int
y = 2 # int

print_car(y,x)
print(x,y)
```

Exercice - Passage de paramètres I

Question : Quel est la sortie de l'algorithme suivant :

A) 1 2 1 2
B) 1 2 2 1
C) 2 1 1 2
D) 2 1 2 1

Passage par valeur

```
print_car(2,1)
print(1,2)
```

Algorithme : afficher_x

```
def print_car(x,y):
    print(x,y,end=" ")

x = 1 # int
y = 2 # int

print_car(y,x)
print(x,y)
```

Exercice - Passage de paramètres II

Question : Quel est la sortie de l'algorithme suivant :

A) 1 2 1 2
B) 1 2 2 3
C) 2 3 1 2
D) 2 3 2 3

Algorithme : afficher_x

```
def print_car(x,y):
    x = x + 1
    y = y + 1
    print(x,y,end=" ")

x = 1 # int
y = 2 # int

print_car(x,y)
print(x,y)
```

Exercice - Passage de paramètres II

Question : Quel est la sortie de l'algorithme suivant :

A) 1 2 1 2

B) 1 2 2 3

C) 2 3 1 2

D) 2 3 2 3

Passage par valeur

```
print_car(1,2)
print(1,2)
```

Algorithme : afficher_x

```
def print_car(x,y):
    x = x + 1
    y = y + 1
    print(x,y,end=" ")

x = 1 # int
y = 2 # int

print_car(x,y)
print(x,y)
```

Exercice - Passage de paramètres III

Question : Quel est la sortie de l'algorithme suivant :

A) 1 1 1
B) 1 2 1
C) 1 2 2

Algorithme : afficher_x

```
def print_car(x):
    x[0] = x[0] + 1
    print(x[0], end=" ")

x = [1] # list

print(x[0], end=" ")
print_car(x)
print(x[0])
```

Exercice - Passage de paramètres III

Question : Quel est la sortie de l'algorithme suivant :

A) 1 1 1

B) 1 2 1

C) 1 2 2

Passage par référence

```
print(1)
print_car(x) # x : mutable
print(2)
```

Algorithme : afficher_x

```
def print_car(x):
    x[0] = x[0] + 1
    print(x[0], end=" ")

x = [1] # list

print(x[0], end=" ")
print_car(x)
print(x[0])
```

Exercice - Passage de paramètres IV

Question : Quel est la sortie de l'algorithme suivant :

A) 1 2 2 3 2 2
B) 1 2 1 2 2 2
C) 1 2 2 2 2 3
D) 1 2 2 2 1 2

Algorithme : afficher_x

```
def print_car(x, y):
    x[0] = x[0] + 1
    y = y + 1
    print(x[0], y, end=" ")

x = [1] # list
y = 2   # int

print(x[0], y, end=" ")
print_car(x, y)
print(x[0], y)
```

Exercice - Passage de paramètres IV

Question : Quel est la sortie de l'algorithme suivant :

A) 1 2 2 3 2 2
B) 1 2 1 2 2 2
C) 1 2 2 2 2 3
D) 1 2 2 2 1 2

Passage par valeur et par référence

```
print(1, 2)
print_car(x, 2) # x : mutable
print(2, 2)
```

Algorithme : afficher_x

```
def print_car(x, y):
    x[0] = x[0] + 1
    y = y + 1
    print(x[0], y, end=" ")

x = [1] # list
y = 2   # int

print(x[0], y, end=" ")
print_car(x, y)
print(x[0], y)
```

Sommaire

CONCEPTION DESCENDANTE

Application

Dans le contexte :

- spécification
- analyse et conception
- programmation
- vérification, validation - test

+ documentation (tout au long)

Concept et stratégie

La conception consiste à élaborer à partir de la spécification du problème une solution informatique

étant donné **[les données]**, on demande **[le résultat attendu]**

Stratégie générale : décomposer le problème en sous-problèmes et identifier les problèmes pertinents

Deux façons de décomposer :

- selon les fonctionnalités, traitements (conception fonctionnelle)
- selon les données à manipuler (conception objet) ($2^{ème}$ semestre - 632-1)

Type de conception

Peu importe la méthode de décomposition choisie, la conception peut être :

descendante (top-down) : l'approche descendante commence par décomposer le problème initial en sous-problèmes puis chaque sous-problème en de nouveaux sous-problèmes et ainsi de suite jusqu'aux problèmes que l'on peut résoudre par des opérations primitives (ou des fonctions simples)

ascendante (bottom-up) : l'approche ascendante construit des opérations primitives que l'on assemble pour obtenir des opérations plus complexes et ainsi de suite jusqu'à une opération globale qui résout le problème initial

→ à ce cours : décomposition fonctionnelle descendante

Décomposition fonctionnelle descendante

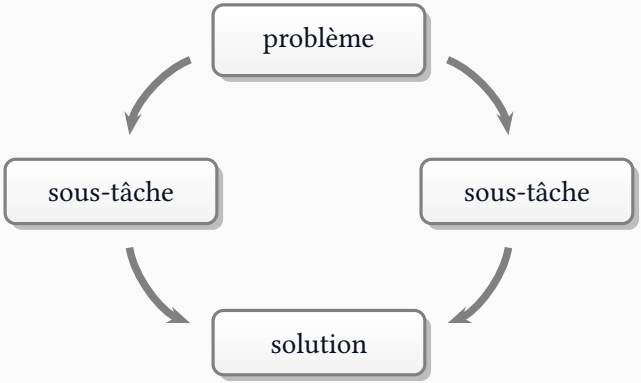
Basée sur la stratégie de résolution **diviser pour régner**

Chaque étape de décomposition est suivie d'une étape de spécification des sous-problèmes

Décomposition fonctionnelle descendante

Basée sur la stratégie de résolution **diviser pour régner**

Chaque étape de décomposition est suivie d'une étape de spécification des sous-problèmes



Approche

Les techniques utilisées dans cette méthode :

raffinement successif : chaque étape de décomposition fait intervenir une seule décision
masquage d'information : programmation modulaire

- Les décisions propres à un module sont cachées aux autres modules

Approche

Les techniques utilisées dans cette méthode :

raffinement successif : chaque étape de décomposition fait intervenir une seule décision

masquage d'information : programmation modulaire

- Les décisions propres à un module sont cachées aux autres modules
- Les données et opérations accessibles aux autres modules le sont au travers d'une interface bien définie

Approche

Les techniques utilisées dans cette méthode :

raffinement successif : chaque étape de décomposition fait intervenir une seule décision

masquage d'information : programmation modulaire

- Les décisions propres à un module sont cachées aux autres modules
- Les données et opérations accessibles aux autres modules le sont au travers d'une interface bien définie
- Les données et opérations non utiles aux autres modules sont inaccessibles

Exemple - La date du lendemain

étant donné une date, **on demande** déterminer la date du lendemain

Exemple - La date du lendemain

étant donné une date, **on demande** déterminer la date du lendemain

On commence par écrire l'interface de la fonction qui apportera la réponse au problème posé :

Définition de l'interface de la fonction `lendemain`

nom	`lendemain`
type	`date -> date`
paramètres	`d`
préconditions	la date `d` est valide
postcondition	retourne la date du lendemain du jour de date `d`
raises	lève l'exception `date_invalide` si `d` n'est pas valide
tests	à écrire dès maintenant

Exemple - Solution informelle

L'analyse rapide du problème amène à la solution informelle suivante (premier niveau d'algorithme)

```
Fonction lendemain(d:date):date
if date est valide then
 if jour est le dernier jour du mois then
   retourner passer au 1er du mois suivant
 else
   retourner passer au jour suivant
else échec
```

Exemple - Solution informelle

L'analyse rapide du problème amène à la solution informelle suivante (premier niveau d'algorithme)

```
Fonction lendemain(d:date):date
if date est valide then
 if jour est le dernier jour du mois then
   retourner passer au 1er du mois suivant
 else
   retourner passer au jour suivant
else échec
```

On fait apparaître deux sous-problèmes :

- le calcul du dernier jour d'un mois
- la vérification de la validité d'une date

Exemple - Solution informelle

L'analyse rapide du problème amène à la solution informelle suivante (premier niveau d'algorithme)

```
Fonction lendemain(d:date):date
if date est valide then
 if jour est le dernier jour du mois then
   retourner passer au 1er du mois suivant
 else
   retourner passer au jour suivant
else échec
```

On fait apparaître deux sous-problèmes :

- le calcul du dernier jour d'un mois
- la vérification de la validité d'une date

On prend aussi à ce stade ou plus tard une décision de représentation des données : une date sera représentée par un triplet (jour, mois, année)

Exemple - Solution informelle

L'analyse rapide du problème amène à la solution informelle suivante (premier niveau d'algorithme)

```
Fonction lendemain(d:date):date
if date est valide then
 if jour est le dernier jour du mois then
   retourner passer au 1er du mois suivant
 else
   retourner passer au jour suivant
else échec
```

On fait apparaître deux sous-problèmes :

- le calcul du dernier jour d'un mois
- la vérification de la validité d'une date

On prend aussi à ce stade ou plus tard une décision de représentation des données : une date sera représentée par un triplet (jour, mois, année)

Par la suite `date` est un synonyme pour le type `int*int*int` (Python module : `datetime`)

Exemple - Decomposition

On définira ainsi deux fonctions :

- `date_valide` de type `entree:date -> sortie:bool`
- et `nombre_de_jours` de type `entree:int*int -> sortie:intt*int*int`

Exemple - Decomposition

On définira ainsi deux fonctions :

- `date_valide` de type `entree:date -> sortie:bool`
- et `nombre_de_jours` de type `entree:int*int -> sortie:intt*int*int`

On écrit les interfaces des deux fonctions :

interface `date_valide`

nom	`date_valide`
type	`date -> bool`
paramètres	`d`
préconditions	aucune
postcondition	retourne true si `d` correspond à une date valide, false sinon
tests	à écrire dès maintenant

interface `nb_jours`

nom	`nb_jours`
type	`int*int -> int*bool`
paramètres	`m,a`
préconditions	`m` est compris entre 1 et 12, `a` est une année
postcondition	retourne le nombre de jours du mois `m` de l'année `a`
tests	à écrire dès maintenant

```
fonction lendemain(jour,mois,an:int):int*int*int
if date_valide(jour,mois,an) then
  if jour == nb_jours(mois,an) then
   if mois == 12 then
    retourner (1, 1, an+1)
   else
    retourner (1, mois+1, an)
  else
   retourner (jour+1, mois, an)
else raise date_invalide
```

Et on recommence avec les fonctions manquantes `nb_jours` et `date_valide`

Avantages

Les avantages de l'approche descendante sont les suivants :

- ceci va rendre le programme beaucoup plus lisible que si il était entièrement écris dans un seul programme

Avantages

Les avantages de l'approche descendante sont les suivants :

- ceci va rendre le programme beaucoup plus lisible que si il était entièrement écris dans un seul programme
- les briques intermédiaires sont réutilisables pour d'autres programmes

Avantages

Les avantages de l'approche descendante sont les suivants :

- ceci va rendre le programme beaucoup plus lisible que si il était entièrement écris dans un seul programme
- les briques intermédiaires sont réutilisables pour d'autres programmes
- le travail est plus facilement partageable entre plusieurs programmeurs

Avantages

Les avantages de l'approche descendante sont les suivants :

- ceci va rendre le programme beaucoup plus lisible que si il était entièrement écris dans un seul programme
- les briques intermédiaires sont réutilisables pour d'autres programmes
- le travail est plus facilement partageable entre plusieurs programmeurs
- on facilite la correction des erreurs

Exercice - Calculer des entiers premiers

Un entier est **premier** s'il est divisible uniquement par 1 et par plus même. L'expressions "est divisible" signifie que le **reste de la division est nulle** (par exemple 9 est divisible par 3, mais pas par 4). L'**objectif** de cet exercice est d'extraire efficacement **tous entiers premiers inférieurs** à une valeur fixée n.

A) Fonction **est_premier** : implémenter une fonction qui prend en paramètre un entier n et qui teste si celui-ci est premier ou non en regardant si il existe un entier plus petit que lui qui le divise.

B) Écrire un **programme** faisant appel à votre fonction permettant de déterminer les entiers premiers inférieurs à p. Vous testerez des valeurs de p qui permettent de ne pas trop attendre.

C) **Modifier la fonction** en constatant que si un entier n ne peut pas être divisible par un entier strictement plus grand que $n/2$.

D) **Tester** de nouveaux les valeurs de p pour lesquels vous étiez bloqué par le temps de calcul.

Référence

Algorithmique - Techniques fondamentales de programmation
Chapitre : Les sous-programmes
Ebel et Rohaut, https://aai-logon.hes-so.ch/eni

Cyberlearn : HES-SO-GE_631-1 Fondements de la programmation
(*welcome*)
http://cyberlearn.hes-so.ch