Algorithmique de base

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

2019-2020

Sommaire

Objective

- Apprendre la priorité des opérateurs
- Maîtriser les structures algorithmique itératives
  - tant que .. faire
  - pour .. faire
- Mettre en œuvre quelques structures pour résoudre des problèmes simples

Sommaire

LA PRÉCÉDENCE DES OPÉRATEURS

Précédence

Les opérateurs n'ont pas tous la même priorité

- la priorité (ou précédence) des opérateurs **définit l'ordre** dans lequel l'expression doit être analysée
- si deux opérateurs ont la **même priorité**, ils peuvent être évalués de gauche à droite

`var_1` **`op1`** `var_2` **`op2`** $\texttt{var}_3$

Multiplication

une multiplication est prioritaire sur une addition : $3 + 2 \times 2 = 7$

Chaque langage dispose de sa propre table de précédence : pour contourner ces problèmes, on **utilise les parenthèses** !

Précédence - Python

Opérateurs	Description	Priorité
`-x`	négation unaire	+++
`x / y, x % y`	division/reste de la div. (modulo)	++
`x * y`	multiplication ou répétition	+
`x + y, x - y`	addition ou concaténation/soustraction	...
`<, <=, >, >=, ==, <>, !=`	opérateurs de comparaison	...
`not x`	négation logique	−
`x and y`	et logique	−−
`x or y`	ou logique	−−−

Exercice - Precedence I

Question : Quel est l'affichage du programme precedence si

- `var_1 = TRUE`
- `var_2 = FALSE`
- `var_3 = TRUE`

A) TRUE, TRUE, FALSE, FALSE
B) FALSE, TRUE, FALSE, TRUE
C) TRUE, TRUE, TRUE, FALSE
D) TRUE, TRUE, TRUE, TRUE

Algorithme : precedence

Data : a,b,c : booleen

```
1  if (var_1 AND var_2) OR var_3 then
2  |   afficher "TRUE"
3  else
4  |   afficher "FALSE"
5  if var_1 AND (var_2 OR var_3) then
6  |   afficher "TRUE"
7  else
8  |   afficher "FALSE"
9  if NOT var_1 OR var_3 then
10 |   afficher "TRUE"
11 else
12 |   afficher "FALSE"
13 if NOT (var_1 OR var_3) then
14 |   afficher "TRUE"
15 else
16 |   afficher "FALSE"
```

Exercice - Precedence I

Question : Quel est l'affichage du programme precedence si

- `var_1 = TRUE`
- `var_2 = FALSE`
- `var_3 = TRUE`

A) TRUE, TRUE, FALSE, FALSE
B) FALSE, TRUE, FALSE, TRUE
C) TRUE, TRUE, TRUE, FALSE
D) TRUE, TRUE, TRUE, TRUE

Algorithme : precedence

Data : a,b,c : booleen

```
1  if (var_1 AND var_2) OR var_3 then
2  |   afficher "TRUE"
3  else
4  |   afficher "FALSE"
5  if var_1 AND (var_2 OR var_3) then
6  |   afficher "TRUE"
7  else
8  |   afficher "FALSE"
9  if NOT var_1 OR var_3 then
10 |   afficher "TRUE"
11 else
12 |   afficher "FALSE"
13 if NOT (var_1 OR var_3) then
14 |   afficher "TRUE"
15 else
16 |   afficher "FALSE"
```

Exercice - Precedence II

Question : Quel est l'affichage du programme precedence ?

A) FALSE et FALSE
B) FALSE et TRUE
C) TRUE et FALSE
D) TRUE et TRUE

```
""" algo: precedence
données: les variables var1, var2 et var3
résultat: affiche True ou False
"""
# déclaration et initialisation
# des variables
var_1: int = 0
var_2: int = 1
var_3: int = 1
# séquence d'opérations
if var_1 * var_2 + var_3:
    print("True")
else:
    print("False")

if var_1 * (var_2 + var_3):
    print("True")
else:
    print("False")
```

Exercice - Precedence II

Question : Quel est l'affichage du programme precedence ?

A) FALSE et FALSE
B) FALSE et TRUE
C) TRUE et FALSE
D) TRUE et TRUE

```
""" algo: precedence
données: les variables var1, var2 et var3
résultat: affiche True ou False
"""
# déclaration et initialisation
# des variables
var_1: int = 0
var_2: int = 1
var_3: int = 1
# séquence d'opérations
if var_1 * var_2 + var_3:
    print("True")
else:
    print("False")

if var_1 * (var_2 + var_3):
    print("True")
else:
    print("False")
```

Sommaire

STRUCTURES ITÉRATIVES

Les boucles

Définition

Les **boucles** sont des structures itératives : une itération ou structure itérative est une séquence d'instructions destinée à être exécutée plusieurs fois

- tant que (while) : **`tant que` `condition` `faire`**
- pour (for) : **`pour` `element` ⟵ `début` `à` `fin` `faire`**

Les boucles - Principe

- La boucle est un élément très simple au premier abord
- Le but d'une boucle est de répéter un bloc d'instructions plusieurs fois
- Selon le type de boucle, le bloc d'instructions va être répété :
  - un nombre fixe de fois (n fois)
  - ou selon un certain nombre de critères (un test d'une ou de plusieurs conditions)

Exemple simple

Acheter un téléphone

```
condition == avoir CHF 1500
total = CHF 0 (mois 0)
```

```
tant que total < 1500 faire
```

1. mois 1 : épargner CHF 300 (`total 300`)

Exemple simple

Acheter un téléphone

```
condition == avoir CHF 1500
total = CHF 0 (mois 0)
```

```
tant que total < 1500 faire
```

1. mois 1 : épargner CHF 300 (`total 300`)
2. mois 2 : épargner CHF 300 (`total 600`)

Exemple simple

Acheter un téléphone

```
condition == avoir CHF 1500
total = CHF 0 (mois 0)
```

```
tant que total < 1500 faire
```

1. mois 1 : épargner CHF 300 (`total 300`)
2. mois 2 : épargner CHF 300 (`total 600`)
3. mois 3 : épargner CHF 300 (`total 900`)

Exemple simple

Acheter un téléphone

```
condition == avoir CHF 1500
total = CHF 0 (mois 0)
```

```
tant que total < 1500 faire
```

1. mois 1 : épargner CHF 300 (`total 300`)
2. mois 2 : épargner CHF 300 (`total 600`)
3. mois 3 : épargner CHF 300 (`total 900`)
4. mois 4 : épargner CHF 300 (`total 1200`)

Exemple simple

Acheter un téléphone

```
condition == avoir CHF 1500
total = CHF 0 (mois 0)
```

```
tant que total < 1500 faire
```

1. mois 1 : épargner CHF 300 (`total 300`)
2. mois 2 : épargner CHF 300 (`total 600`)
3. mois 3 : épargner CHF 300 (`total 900`)
4. mois 4 : épargner CHF 300 (`total 1200`)
5. mois 5 : épargner CHF 300 (`total 1500`)

Autre exemple

Cas où un utilisateur doit répondre à une question parmi une liste de réponses imposées (ex. : oui et non) :

→ si l'utilisateur répond à autre chose, il faut lui reposer la question, jusqu'à ce qu'il donne une réponse attendue (ex. : oui ou non)

Exemple - Demander une réponse à l'utilisateur

```
condition = réponse oui ou non
réponse = ?
```

```
tant que réponse ≠ oui et réponse ≠ non
faire
```

- pas 1 : afficher "Voulez vous un café ?"

Autre exemple

Cas où un utilisateur doit répondre à une question parmi une liste de réponses imposées (ex. : oui et non) :

→ si l'utilisateur répond à autre chose, il faut lui reposer la question, jusqu'à ce qu'il donne une réponse attendue (ex. : oui ou non)

Exemple - Demander une réponse à l'utilisateur

```
condition = réponse oui ou non
réponse = ?
```

```
tant que réponse ≠ oui et réponse ≠ non
faire
```

- pas 1 : afficher "Voulez vous un café ?"
- pas 1 : saisir réponse (réponse "peut-être")

Autre exemple

Cas où un utilisateur doit répondre à une question parmi une liste de réponses imposées (ex. : oui et non) :

→ si l'utilisateur répond à autre chose, il faut lui reposer la question, jusqu'à ce qu'il donne une réponse attendue (ex. : oui ou non)

Exemple - Demander une réponse à l'utilisateur

```
condition = réponse oui ou non
réponse = ?
```

```
tant que réponse ≠ oui et réponse ≠ non
faire
```

- pas 1 : afficher "Voulez vous un café ?"
- pas 1 : saisir réponse (réponse "peut-être")
- pas 2 : afficher "Voulez vous un café ?"

Autre exemple

Cas où un utilisateur doit répondre à une question parmi une liste de réponses imposées (ex. : oui et non) :

→ si l'utilisateur répond à autre chose, il faut lui reposer la question, jusqu'à ce qu'il donne une réponse attendue (ex. : oui ou non)

Exemple - Demander une réponse à l'utilisateur

```
condition = réponse oui ou non
réponse = ?
```

```
tant que réponse ≠ oui et réponse ≠ non
faire
```

- pas 1 : afficher "Voulez vous un café ?"
- pas 1 : saisir réponse (réponse "peut-être")
- pas 2 : afficher "Voulez vous un café ?"
- pas 2 : saisir réponse (réponse "j'sais pas")

Autre exemple

Cas où un utilisateur doit répondre à une question parmi une liste de réponses imposées (ex. : oui et non) :

→ si l'utilisateur répond à autre chose, il faut lui reposer la question, jusqu'à ce qu'il donne une réponse attendue (ex. : oui ou non)

Exemple - Demander une réponse à l'utilisateur

```
condition = réponse oui ou non
réponse = ?
```

```
tant que réponse ≠ oui et réponse ≠ non
faire
```

- pas 1 : afficher "Voulez vous un café ?"
- pas 1 : saisir réponse (réponse "peut-être")
- pas 2 : afficher "Voulez vous un café ?"
- pas 2 : saisir réponse (réponse "j'sais pas")
- pas 3 : afficher "Voulez vous un café ?"

Autre exemple

Cas où un utilisateur doit répondre à une question parmi une liste de réponses imposées (ex. : oui et non) :

→ si l'utilisateur répond à autre chose, il faut lui reposer la question, jusqu'à ce qu'il donne une réponse attendue (ex. : oui ou non)

Exemple - Demander une réponse à l'utilisateur

```
condition = réponse oui ou non
réponse = ?
```

```
tant que réponse ≠ oui et réponse ≠ non
faire
```

- pas 1 : afficher "Voulez vous un café ?"
- pas 1 : saisir réponse (réponse "peut-être")
- pas 2 : afficher "Voulez vous un café ?"
- pas 2 : saisir réponse (réponse "j'sais pas")
- pas 3 : afficher "Voulez vous un café ?"
- pas 3 : saisir réponse (réponse "oui")

La boucle tant que (while)

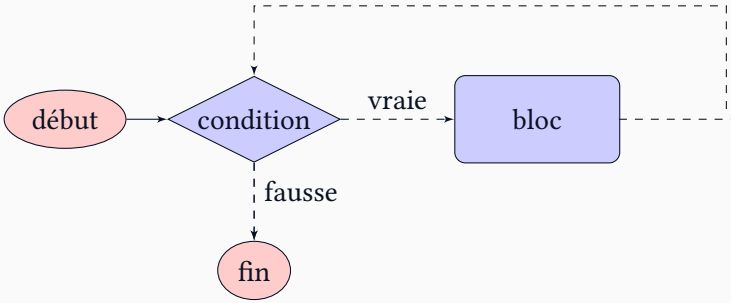
La boucle tant que - Principe

La boucle de type **tant que** permet la répétition d'un bloc d'instructions tant que la **condition** testée est vraie

Lors de l'exécution du programme, quand celui-ci arrive sur l'instruction **tant que**

- il évalue la `condition`
- si l'expression retourne `TRUE`, le programme exécute le bloc d'instructions suivante
- à la fin du bloc, il remonte au **tant que** et évalue de nouveau l'expression booléenne
- si c'est `TRUE` il exécute de nouveau le bloc d'instructions, et ainsi de suite, **tant que** l'expression retourne `TRUE`
- si l'expression devient `FALSE`, après l'évaluation de la condition, le programme saute à l'instruction située juste après le bloc d'instruction **tant que**

Tant que (While) - Structure algorithmique




```
tant que condition faire
    bloc
```

Python - Tant que

```
while condition:
    # bloc d'instructions
    ...
```

Des exemples - Compteur

```
""" algo: compteur
données: valeur initial
résultat: affiche compteur
"""
### décl. et init.
### des variables
compteur: int = 1

### séquence d'opérations
while compteur <= 10:
  print("compteur:", compteur)
  compteur = compteur + 1
```

- compteur=1: condition == True -> affiche 1

Des exemples - Compteur

```
""" algo: compteur
données: valeur initial
résultat: affiche compteur
"""
### décl. et init.
### des variables
compteur: int = 1

### séquence d'opérations
while compteur <= 10:
  print("compteur:", compteur)
  compteur = compteur + 1
```

- compteur=1: condition == **True** -> affiche 1
- compteur=2: condition == **True** -> affiche 2

Des exemples - Compteur

```
""" algo: compteur
données: valeur initial
résultat: affiche compteur
"""
### décl. et init.
### des variables
compteur: int = 1

### séquence d'opérations
while compteur <= 10:
  print("compteur:", compteur)
  compteur = compteur + 1
```

- compteur=1: condition == True -> affiche 1
- compteur=2: condition == True -> affiche 2
- compteur=3: condition == True -> affiche 3

Des exemples - Compteur

```python
""" algo: compteur
données: valeur initial
résultat: affiche compteur
"""
### décl. et init.
### des variables
compteur: int = 1

### séquence d'opérations
while compteur <= 10:
  print("compteur:", compteur)
  compteur = compteur + 1
```

- compteur=1: condition == True -> affiche 1
- compteur=2: condition == True -> affiche 2
- compteur=3: condition == True -> affiche 3
- ...

Des exemples - Compteur

```python
""" algo: compteur
données: valeur initial
résultat: affiche compteur
"""
### décl. et init.
### des variables
compteur: int = 1

### séquence d'opérations
while compteur <= 10:
  print("compteur:", compteur)
  compteur = compteur + 1
```

- compteur=1: condition == True -> affiche 1
- compteur=2: condition == True -> affiche 2
- compteur=3: condition == True -> affiche 3
- ...
- compteur=9: condition == True -> affiche 9

Des exemples - Compteur

```
""" algo: compteur
données: valeur initial
résultat: affiche compteur
"""
### décl. et init.
### des variables
compteur: int = 1

### séquence d'opérations
while compteur <= 10:
  print("compteur:", compteur)
  compteur = compteur + 1
```

- compteur=1: condition == True -> affiche 1
- compteur=2: condition == True -> affiche 2
- compteur=3: condition == True -> affiche 3
- ...
- compteur=9: condition == True -> affiche 9
- compteur=10: condition == True -> affiche 10

Des exemples - Compteur

```
""" algo: compteur
données: valeur initial
résultat: affiche compteur
"""
### décl. et init.
### des variables
compteur: int = 1

### séquence d'opérations
while compteur <= 10:
  print("compteur:", compteur)
  compteur = compteur + 1
```

- compteur=1: condition == True -> affiche 1
- compteur=2: condition == True -> affiche 2
- compteur=3: condition == True -> affiche 3
- ...
- compteur=9: condition == True -> affiche 9
- compteur=10: condition == True -> affiche 10
- compteur=11: condition == False

Des exemples - Une table de multiplication

```
""" algo: table_multiplication
données: table: int
résultat: affiche une table de multi.
"""
### décl. et init.
### des variables
cpt: int = 1
table: int = None

### séquence d'opérations
table = int(input("Quelle table ?"))

while cpt <= 10:
  res: int = table * cpt
  print("%d x %d = %d" %(table, cpt, res))
  cpt += 1
```

table = 10

▶ cpt=1: condition == **True**
affiche 10 × 1 = 10

Des exemples - Une table de multiplication

```python
""" algo: table_multiplication
données: table: int
résultat: affiche une table de multi.
"""
### décl. et init.
### des variables
cpt: int = 1
table: int = None

### séquence d'opérations
table = int(input("Quelle table ?"))

while cpt <= 10:
  res: int = table * cpt
  print("%d x %d = %d" %(table, cpt, res))
  cpt += 1
```

table = 10

- cpt=1: condition == **True**
  affiche 10 × 1 = 10
- cpt=2: condition == **True**
  affiche 10 × 2 = 20

Des exemples - Une table de multiplication

```
""" algo: table_multiplication
données: table: int
résultat: affiche une table de multi.
"""
### décl. et init.
### des variables
cpt: int = 1
table: int = None

### séquence d'opérations
table = int(input("Quelle table ?"))

while cpt <= 10:
  res: int = table * cpt
  print("%d x %d = %d" %(table, cpt, res))
  cpt += 1
```

table = 10

- cpt=1: condition == **True**
  affiche 10 × 1 = 10
- cpt=2: condition == **True**
  affiche 10 × 2 = 20
- ...

Des exemples - Une table de multiplication

```
""" algo: table_multiplication
données: table: int
résultat: affiche une table de multi.
"""
### décl. et init.
### des variables
cpt: int = 1
table: int = None

### séquence d'opérations
table = int(input("Quelle table ?"))

while cpt <= 10:
  res: int = table * cpt
  print("%d x %d = %d" %(table, cpt, res))
  cpt += 1
```

table = 10

- cpt=1: condition == **True**
  affiche 10 × 1 = 10
- cpt=2: condition == **True**
  affiche 10 × 2 = 20
- ...
- cpt=9: condition == **True**
  affiche 10 × 9 = 90

Des exemples - Une table de multiplication

```
""" algo: table_multiplication
données: table: int
résultat: affiche une table de multi.
"""
### décl. et init.
### des variables
cpt: int = 1
table: int = None

### séquence d'opérations
table = int(input("Quelle table ?"))

while cpt <= 10:
  res: int = table * cpt
  print("%d x %d = %d" %(table, cpt, res))
  cpt += 1
```

table = 10

- cpt=1: condition == **True**
  affiche 10 × 1 = 10
- cpt=2: condition == **True**
  affiche 10 × 2 = 20
- ...
- cpt=9: condition == **True**
  affiche 10 × 9 = 90
- cpt=10: condition == **True**
  affiche 10 × 10 = 100

Des exemples - Une table de multiplication

```
""" algo: table_multiplication
données: table: int
résultat: affiche une table de multi.
"""
### décl. et init.
### des variables
cpt: int = 1
table: int = None

### séquence d'opérations
table = int(input("Quelle table ?"))

while cpt <= 10:
  res: int = table * cpt
  print("%d x %d = %d" %(table, cpt, res))
  cpt += 1
```

table = 10

- cpt=1: condition == **True**
  affiche 10 × 1 = 10
- cpt=2: condition == **True**
  affiche 10 × 2 = 20
- ...
- cpt=9: condition == **True**
  affiche 10 × 9 = 90
- cpt=10: condition == **True**
  affiche 10 × 10 = 100
- cpt=11: condition == **False**

QCM - Tant que

Question : Quelle sera la valeur (`resultat`) afficher par l'algorithme suivant ?

A) 1

B) 5

C) 4

D) 120

E) 0

```
""" algo: boucle_x
données: i: int
résultat: la valeur x
"""

### décl. et init. des variables
x: int = 1
i: int = 5

### séquence d'opérations
while i > 0:
  x = x * i
  i = i - 1

print("resultat:", x)
```

QCM - Tant que

Question : Quelle sera la valeur (`resultat`) afficher par l'algorithme suivant ?

A) 1

B) 5

C) 4

D) 120

E) 0

```
""" algo: boucle_x
données: i: int
résultat: la valeur x
"""

### décl. et init. des variables
x: int = 1
i: int = 5

### séquence d'opérations
while i > 0:
  x = x * i
  i = i - 1

print("resultat:", x)
```

QCM - Tant que

- i = 5 : condition == **True**
  x = 1 × 5
- i = 4 : condition == **True**
  x = 5 × 4
- i = 3 : condition == **True**
  x = 20 × 3
- i = 2 : condition == **True**
  x = 60 × 2
- i = 1 : condition == **True**
  x = 120 × 1
- i = 0 : condition == **False**

resultat: 120

```
""" algo: boucle_x
données: i: int
résultat: la valeur x
"""

### décl. et init. des variables
x: int = 1
i: int = 5

### séquence d'opérations
while i >= 1:
  x = x * i
  i = i -1

print("resultat:", x)
```

Factorielle

$$n! = n \times (n-1) \times \ldots \times 2 \times 1$$
$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Cas particulier - à éviter !

- Le bloc d'instructions d'une boucle **tant que** peut ne jamais être exécuté
  Exemple :

```
i = 0
tant que i > 0 faire bloc
```

Cas particulier - à éviter !

- Le bloc d'instructions d'une boucle **tant que** peut ne jamais être exécuté
  Exemple :

```
i = 0
tant que i > 0 faire bloc
```

- On peut ne jamais sortir d'une boucle **tant que** (condition toujours vérifiée)
  Exemple :

```
tant que TRUE faire bloc
```

PyCharm

exercice_1.py

La boucle pour (`for`)

La boucle pour - Principe

Deuxième structure itérative de l'algorithmique : le `pour ... faire` est une boucle à l'usage des compteurs et séquences

Principe

À chaque passage dans la boucle, un **compteur est incrémenté ou décrémenté**, selon le cas : on dit alors qu'il s'agit d'une structure incrémentale

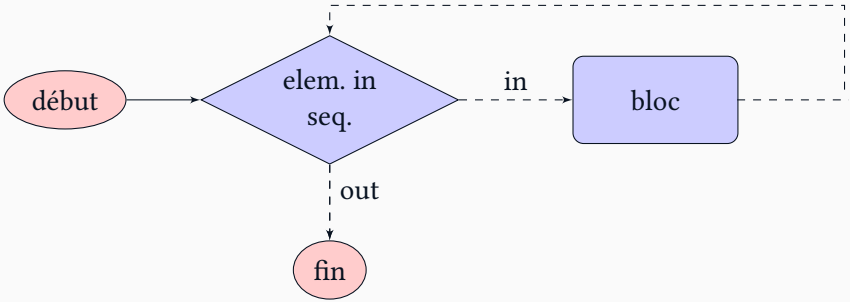
```
pour element ⟵ debut à fin [pas] faire bloc
```

Python - Pour

```
for element in sequence:
    # bloc d'instructions
    ...
```

```
for cpt in range(debut, fin, pas):
    # bloc d'instructions
    ...
```

Structure algorithmique - Pour (For)



Sequence

- `1, 2, 3, ..., 9, 10` — `pour compteur ← 1 à 10 faire ...`
- `0, 2, 4, ..., 8, 10` — `pour cpt_pair ← 0 à 10 [pas 2] faire ...`
- `10, 9, 8, ..., 2, 1` — `pour cpt_rev ← 10 à 1 [pas -1] faire ...`

Un exemple simple

Créer une table de multiplication de 3

`table` ⟶ `3`
`séquénce` ⟶ `cpt` ≥`1` `et` `cpt` ≤`10`

- $3 \times 1 = 3$
- $3 \times 2 = 6$
- ...
- $3 \times 9 = 27$
- $3 \times 10 = 30$

Un exemple simple

Créer une table de multiplication de 3

`table` ⟶ `3`
`séquénce` ⟶ `cpt` ≥1 `et` `cpt` ≤10

pour `cpt` ← 1 **à** 10 **faire**
- pas 1 : `table` × `cpt` $(3 \times 1 = 3)$

- $3 \times 1 = 3$
- $3 \times 2 = 6$
- ...
- $3 \times 9 = 27$
- $3 \times 10 = 30$

Un exemple simple

Créer une table de multiplication de 3

`table` ⟶ `3`
`séquénce` ⟶ `cpt` ≥1 et `cpt` ≤10

- $3 \times 1 = 3$
- $3 \times 2 = 6$
- ...
- $3 \times 9 = 27$
- $3 \times 10 = 30$

pour `cpt` ← 1 **à** 10 **faire**

- pas 1 : `table` × `cpt` $(3 \times 1 = 3)$
- pas 2 : `table` × `cpt` $(3 \times 2 = 6)$

Un exemple simple

Créer une table de multiplication de 3

`table` ⟶ `3`
`séquénce` ⟶ `cpt` ≥1 et `cpt` ≤10

- $3 \times 1 = 3$
- $3 \times 2 = 6$
- ...
- $3 \times 9 = 27$
- $3 \times 10 = 30$

pour `cpt` ← 1 **à** 10 **faire**

- pas 1 : `table` × `cpt` $(3 \times 1 = 3)$
- pas 2 : `table` × `cpt` $(3 \times 2 = 6)$
- pas 3 : `table` × `cpt` $(3 \times 3 = 9)$

Un exemple simple

Créer une table de multiplication de 3

`table` ⟶ `3`
`séquénce` ⟶ `cpt` ≥1 et `cpt` ≤10

- $3 \times 1 = 3$
- $3 \times 2 = 6$
- ...
- $3 \times 9 = 27$
- $3 \times 10 = 30$

pour `cpt` ← 1 **à** 10 **faire**

- pas 1 : `table` × `cpt` $(3 \times 1 = 3)$
- pas 2 : `table` × `cpt` $(3 \times 2 = 6)$
- pas 3 : `table` × `cpt` $(3 \times 3 = 9)$
- ...

Un exemple simple

Créer une table de multiplication de 3

`table` ⟶ `3`
`séquénce` ⟶ `cpt` ≥1 et `cpt` ≤10

- $3 \times 1 = 3$
- $3 \times 2 = 6$
- ...
- $3 \times 9 = 27$
- $3 \times 10 = 30$

pour `cpt` ← 1 **à** 10 **faire**

- pas 1 : `table` × `cpt` $(3 \times 1 = 3)$
- pas 2 : `table` × `cpt` $(3 \times 2 = 6)$
- pas 3 : `table` × `cpt` $(3 \times 3 = 9)$
- ...
- pas 9 : `table` × `cpt` $(3 \times 9 = 27)$

Un exemple simple

Créer une table de multiplication de 3

`table` ⟶ `3`
`séquénce` ⟶ `cpt` ≥1 et `cpt` ≤10

- $3 \times 1 = 3$
- $3 \times 2 = 6$
- ...
- $3 \times 9 = 27$
- $3 \times 10 = 30$

pour `cpt` ← 1 **à** 10 **faire**

- pas 1 : `table` × `cpt` $(3 \times 1 = 3)$
- pas 2 : `table` × `cpt` $(3 \times 2 = 6)$
- pas 3 : `table` × `cpt` $(3 \times 3 = 9)$
- ...
- pas 9 : `table` × `cpt` $(3 \times 9 = 27)$
- pas 10 : `table` × `cpt` $(3 \times 10 = 30)$

Factorielle avec pour

Avec une factorielle de n, on sait à l'avance le nombre d'itérations nécessaire $\rightarrow n$
C'est donc une application de choix pour la structure `pour ... faire`

```
""" algo: factorielle
données: n: int
résultat: la valeur de la factorielle
"""

### décl. et init. des variables
resultat: int = 1
n: int = None

### séquence d'opérations
n = int(input("Quel factoriel ?"))

for i in range(1, n+1):
  resultat = resultat * i

print("resultat:", resultat)
```

QCM - Pour

Question : Quelle sera la valeur (`resultat`) afficher par l'algorithme suivant ?

A) 0

B) 5

C) 15

D) 20

E) 21

```
""" algo: boucle_x
données: i: int
résultat: la valeur x
"""

### décl. et init. des variables
x: int = 0
n: int = 5

### séquence d'opérations
for i in range(1, n+1):
  x = x + i

print("resultat:", x)
```

QCM - Pour

Question : Quelle sera la valeur (`resultat`) afficher par l'algorithme suivant ?

A) 0

B) 5

C) 15

D) 20

E) 21

```
""" algo: boucle_x
données: i: int
résultat: la valeur x
"""

### décl. et init. des variables
x: int = 0
n: int = 5

### séquence d'opérations
for i in range(1, n+1):
  x = x + i

print("resultat:", x)
```

QCM - Pour

- `i = 1` : condition == **True**
  `x = 0 + 1`
- `i = 2` : condition == **True**
  `x = 1 + 2`
- `i = 3` : condition == **True**
  `x = 3 + 3`
- `i = 4` : condition == **True**
  `x = 6 + 4`
- `i = 5` : condition == **True**
  `x = 10 + 5`
- `i = 6` : condition == **False**

resultat: 15

```
""" algo: boucle_x
données: n: int, x: int
résultat: la valeur x
"""

### décl. et init. des variables
x: int = 0
n: int = 5

### séquence d'opérations
for i in range(1, n+1):
  x = x + i

print("resultat:", x)
```

Somme n premiers entiers

somme(n) = 1 + 2 + ... + n-1 + n

somme(5) = 1 + 2 + 3 + 4 + 5 = 15

Quelle structure choisir

Pour

- on emploie une structure **pour** lorsqu'on connaît à l'avance le nombre d'itérations nécessaires au traitement
- la boucle **pour** est déterministe : son nombre d'itérations est fixé une fois pour toute et est en principe invariable

Tant que

- on emploie les structures **tant que** lorsqu'on ne connaît pas forcément à l'avance le nombre d'itérations qui seront nécessaires à l'obtention du résultat souhaité

Un piège à éviter

Tout comme il faut éviter les boucles `tant que` infinies, il faut aussi éviter quelques erreurs qui peuvent se révéler très surprenantes selon le cas

Voici un exemple de ce qu'il ne faut pas faire :

Algorithme : bad_pour

Data : x : int

```
1 for x ← 1 to 10 do
2     x = x + 2
```

PyCharm

exercice_2.py

Les boucles imbriquées

Des boucles imbriquées

Boucles imbriquées - une boucle dans une autre boucle

De même qu'une structure si ... alors peut contenir d'autres structures si ... alors, une boucle peut tout à fait contenir d'autres boucles

Calcule de la moyenne par étudiant-e

« prenons tou-te-s les étudiant-e-s du module 631-1 un par un »

« pour chaque étudiant-e, prenons toutes les notes et calculons la moyenne »

Structures des boucles imbriquées

```
tant que condition_1 faire
  tant que condition_2 faire
    ...
      tant que condition_n faire
        bloc_n
```

```
pour i ← debut_1 à fin_1 faire
  pour j ← debut_2 à fin_2 faire
    ...
      pour n ← debut_n à fin_n faire
        bloc_n
```

Python - Tant que imbriquée

```
while condition_1:
    while condition_2:
        ...
            while condition_n:
                # bloc d'instructions n
                ...
```

Python - Pour imbriquée

```
for i in seq_1:
    for j in seq_2:
        ...
            for n in seq_n:
                # bloc d'instructions n
                ...
```

Des boucles imbriquées - Exemple

Tables de multiplication 10 x 10

Calculer et afficher toutes les tables de multiplication de 1 à 10 :

- pour cela deux boucles doivent être utilisées
- la première va représenter la table à calculer, de 1 à 10
- la seconde à l'intérieur de la première va multiplier la table donnée de 1 à 10

- première boucle : table des **1**
  seconde boucle : exécution de `1*1`, `1*2`, `1*3`, `...`, `1*9`, `1*10`

Des boucles imbriquées - Exemple

Tables de multiplication 10 x 10

Calculer et afficher toutes les tables de multiplication de 1 à 10 :

- pour cela deux boucles doivent être utilisées
- la première va représenter la table à calculer, de 1 à 10
- la seconde à l'intérieur de la première va multiplier la table donnée de 1 à 10

- première boucle : table des **1**
  - seconde boucle : exécution de `1*1`, `1*2`, `1*3`, `...`, `1*9`, `1*10`
- première boucle : table des **2**
  - seconde boucle : exécution de `2*1`, `2*2`, `2*3`, `...2*9`, `2*10`

Des boucles imbriquées - Exemple

Tables de multiplication 10 x 10

Calculer et afficher toutes les tables de multiplication de 1 à 10 :

- pour cela deux boucles doivent être utilisées
- la première va représenter la table à calculer, de 1 à 10
- la seconde à l'intérieur de la première va multiplier la table donnée de 1 à 10

- première boucle : table des **1**
  - seconde boucle : exécution de `1*1, 1*2, 1*3, ..., 1*9, 1*10`
- première boucle : table des **2**
  - seconde boucle : exécution de `2*1, 2*2, 2*3, ...2*9, 2*10`
- première boucle : table des **3**
  - seconde boucle : exécution de `3*1, 3*2, 3*3, ... ,3*9,3*10`

Des boucles imbriquées - Exemple

Tables de multiplication 10 x 10

Calculer et afficher toutes les tables de multiplication de 1 à 10 :

- pour cela deux boucles doivent être utilisées
- la première va représenter la table à calculer, de 1 à 10
- la seconde à l'intérieur de la première va multiplier la table donnée de 1 à 10

- première boucle : table des **1**
  - seconde boucle : exécution de `1*1, 1*2, 1*3, ..., 1*9, 1*10`
- première boucle : table des **2**
  - seconde boucle : exécution de `2*1, 2*2, 2*3, ...2*9, 2*10`
- première boucle : table des **3**
  - seconde boucle : exécution de `3*1, 3*2, 3*3, ... ,3*9,3*10`
- ...

Des boucles imbriquées - Exemple

Tables de multiplication 10 x 10

Calculer et afficher toutes les tables de multiplication de 1 à 10 :

- pour cela deux boucles doivent être utilisées
- la première va représenter la table à calculer, de 1 à 10
- la seconde à l'intérieur de la première va multiplier la table donnée de 1 à 10

- première boucle : table des **1**
  seconde boucle : exécution de `1*1, 1*2, 1*3, ..., 1*9, 1*10`
- première boucle : table des **2**
  seconde boucle : exécution de `2*1, 2*2, 2*3, ...2*9, 2*10`
- première boucle : table des **3**
  seconde boucle : exécution de `3*1, 3*2, 3*3, ... ,3*9,3*10`
- ...
- première boucle : table des **10**
  seconde boucle : exécution de `10*1, 10*2, 10*3, ..., 10*9, 10*10`

Toutes les tables de multiplication

```
""" algo: table_multiplication
données: nombre maximum pour calculer la multiplication
résultat: tables de multiplication
"""
### déclaration et initialisation
### des variables
TABLE_MAX: int = 10

### séquence d'opérations
table1: int = 1
while table1 <= TABLE_MAX:      # traite la boucle extérieur
  print("### table de", table1, "###")

  table2: int = 1
  while table2 <= TABLE_MAX:    # traite la boucle intérieur
    resultat: int = table1 * table2
    print("%d x %d = %d" %(table1, table2, resultat))
    table2 += 1

  table1 += 1
```

Autre exemple - Carré

Créez un algorithme pour afficher un carré à l'écran comme ci-dessous :

```
**********
**********
**********
**********
**********
**********
**********
**********
**********
**********
```

```python
""" algo: carre
donées: n: int -> côté du carré
résultat: affiche un carré de taille n x n à l'écran
"""

### déclaration et initialisation
### des variables
n: int = None

### séquence d'opérations
n = int(input("Côté du carré ?"))

i: int = 1
while i <= n:           # traite les lignes
    j: int = 1
    while j <= n:       # traite les colonnes
        print("*", end="")
        j += 1
    print()
    i += 1
```

PyCharm

exercice_[3,4].py

Conclusion

Contenu vu :

- précédence des opérateurs
- les boucles simple
  - tant que
  - pour

Référence

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

Cyberlearn : 19_HES-SO-GE_631-1 FONDEMENT DE LA PROGRAMMATION (*welcome*)
http://cyberlearn.hes-so.ch

EXTRA

Des examples - Une table de multiplication reverse

Algorithme : table_multi_rev

Data : table, compteur, resultat:entier

```
// séquence d'opérations
1 afficher ``Quelle table de multiplication ?''
2 saisir table
3 compteur: int = 10
4 while compteur > 0 do
5     resultat: int = table * compteur
6     afficher
        table,``x'',compteur,``='',resultat
7     compteur = compteur - 1
```

- compteur = 10 : condition == **True**
  `afficher` $10 \times 10 = 100$
- compteur = 9 : condition == **True**
  `afficher` $10 \times 9 = 90$
- compteur = 8 : condition == **True**
  `afficher` $10 \times 8 = 80$
- ...
- compteur = 2 : condition == **True**
  `afficher` $10 \times 2 = 20$
- compteur = 1 : condition == **True**
  `afficher` $10 \times 1 = 10$
- compteur = 0 : condition == **False**

Des examples - Une table de multiplication reverse

Algorithme : table_multi_rev

Data : table, compteur, resultat:entier

```
// séquence d'opérations
1 afficher ``Quelle table de multiplication ?''
2 saisir table
3 compteur: int = 10
4 while compteur > 0 do
5     resultat: int = table * compteur
6     afficher
        table,``x'',compteur,``='',resultat
7     compteur = compteur - 1
```

Saisie

```
table = 10
```

- compteur = 10 : condition == **True**
  `afficher` $10 \times 10 = 100$
- compteur = 9 : condition == **True**
  `afficher` $10 \times 9 = 90$
- compteur = 8 : condition == **True**
  `afficher` $10 \times 8 = 80$
- ...
- compteur = 2 : condition == **True**
  `afficher` $10 \times 2 = 20$
- compteur = 1 : condition == **True**
  `afficher` $10 \times 1 = 10$
- compteur = 0 : condition == **False**

Des exemples - x à la puissance y

Algorithme : puissance

Data : x, n, compteur, resultat:entier

```
// initialisation des variables
resultat : int = 1
// séquence d'opérations
afficher ``x,n?''
saisir x,n
compteur: int = 1
while compteur ≤ n do
    resultat = resultat * x
    compteur = compteur + 1
afficher resultat
```

Rappel

$$x^n = \overbrace{x * x * \ldots * x * x}^{=n}$$

$$2^4 = 2 * 2 * 2 * 2$$

Saisie

```
x = 2
n = 4
```

- compteur = 1 : condition == **True**
  $\texttt{resultat} = 1 \times 2$
- compteur = 2 : condition == **True**
  $\texttt{resultat} = 2 \times 2$
- compteur = 3 : condition == **True**
  $\texttt{resultat} = 4 \times 2$
- compteur = 4 : condition == **True**
  $\texttt{resultat} = 8 \times 2$
- compteur = 5 : condition == **False**

Des exemples - x à la puissance n

Algorithme : puissance

Data : n : int, x : float

```
1  resultat : float = 1                          // initialisation des variables
2  signe : int = 1
3  afficher ``x,n?''                             // séquence d'opérations
4  saisir x,n
5  if n ≠ 0 then                                 // x^0 = 1
6      if n ≤ 0 then                             // test le signal de n
7          n = -n                                // valeur absolue
8          signe = -1                            // puissance negative
9      compteur : int = 1
10     while compteur ≤ n do
11         resultat = resultat * x
12         compteur = compteur + 1
13     if signe < 0 then
14         resultat = 1/resultat
15 afficher resultat
```

QCM - Tant que I

Question : Quel est le résultat de l'algorithme suivant :

A) 1234567

B) 123456

C) 23456

D) 234567

Algorithme : boucle_x

Data : jour:entier

```
// initialisation des variables
jour : int = 1
// séquence d'opérations
while jour < 7 do
    jour = jour + 1
    afficher jour
```

QCM - Tant que I

Question : Quel est le résultat de l'algorithme suivant :

A) 1234567

B) 123456

C) 23456

D) 234567

Algorithme : boucle_x

Data : jour:entier

```
// initialisation des variables
jour : int = 1
// séquence d'opérations
while jour < 7 do
    jour = jour + 1
    afficher jour
```

QCM - Tant que II

Question : Quel est le résultat de l'algorithme suivant :

A) 1

B) 9

C) 11

D) 13

Algorithme : boucle_x

Data : plus:entier

```
// initialisation des variables
plus: int = 1
// séquence d'opérations
while plus ≤ 10 do
    plus = plus + 2
afficher jour
```

QCM - Tant que II

Question : Quel est le résultat de l'algorithme suivant :

A) 1

B) 9

C) 11

D) 13

Algorithme : boucle_x

Data : plus:entier

```
// initialisation des variables
plus: int = 1
// séquence d'opérations
while plus ≤ 10 do
    plus = plus + 2
afficher jour
```

QCM - Tant que III

Question : Quel est le résultat de l'algorithme suivant :

A) 15

B) 1

C) 0

D) -1

Algorithme : boucle_x

Data : x:entier

```
// initialisation des variables
x : int = 15
// séquence d'opérations
while x > 0 do
    x = x - 2
afficher x
```

QCM - Tant que III

Question : Quel est le résultat de l'algorithme suivant :

A) 15

B) 1

C) 0

D) -1

Algorithme : boucle_x

Data : x:entier

```
// initialisation des variables
x : int = 15
// séquence d'opérations
while x > 0 do
    x = x - 2
afficher x
```

QCM - Tant que IV

Question : Quel est le résultat de l'algorithme suivant :

A) `rien`

B) `110`

C) `10`

D) `100`

Algorithme : boucle_x

Data : x:entier

```
// initialisation des variables
x: int = 10
// séquence d'opérations
while x > 0 do
    x = x + 10
afficher x
```

QCM - Tant que IV

Question : Quel est le résultat de l'algorithme suivant :

A) `rien`

B) `110`

C) `10`

D) `100`

Algorithme : boucle_x

Data : x:entier

```
// initialisation des variables
x: int = 10
// séquence d'opérations
while x > 0 do
    x = x + 10
afficher x
```

QCM - Tant que V

Question : Quel est le résultat de l'algorithme suivant :

A) `rien`

B) `2345678910...`

C) `12345678910...`

D) `TRUE`

Algorithme : boucle_x

Data : x:entier

```
// initialisation des variables
x: int = 1
// séquence d'opérations
while TRUE do
    x = x + 1
    afficher x
```

QCM - Tant que V

Question : Quel est le résultat de l'algorithme suivant :

A) `rien`

B) `2345678910...`

C) `12345678910...`

D) `TRUE`

Algorithme : boucle_x

Data : x:entier

```
// initialisation des variables
x: int = 1
// séquence d'opérations
while TRUE do
    x = x + 1
    afficher x
```

QCM - Tant que VI

Question : Quel est le résultat de l'algorithme suivant :

A) `0-0-0-`

B) `2-3-6-`

C) `2-5-11-`

D) `3-3-3-`

Algorithme : boucle_x

Data : x,y,res:entier

```
// séquence d'opérations
x: int = 3
while x > 0 do
    res: int = 0
    y: int = 3
    while y > 0 do
        res = res + y
        y = y - 1
    res = res / x
    afficher res,"-"
    x = x - 1
```

QCM - Tant que VI

Question : Quel est le résultat de l'algorithme suivant :

A) `0-0-0-`

B) `2-3-6-`

C) `2-5-11-`

D) `3-3-3-`

Algorithme : boucle_x

Data : x,y,res:entier

```
// séquence d'opérations
x: int = 3
while x > 0 do
    res: int = 0
    y: int = 3
    while y > 0 do
        res = res + y
        y = y - 1
    res = res / x
    afficher res,"-"
    x = x - 1
```

QCM - Pour I

Question : Quel est le résultat du algorithme suivant :

A) 3-6-9-

B) 1-3-6-9

C) 11

D) 0

Algorithme : pour_x

Data : resultat:entier

```
// initialisation des variables
resultat = 0
// séquence d'opérations
for i ⟵ 1 to 10 do
    for j ⟵ 1 to 10 do
        if i == 3 * j then
            afficher i,"-"
```

QCM - Pour I

Question : Quel est le résultat du algorithme suivant :

A) 3-6-9-

B) 1-3-6-9

C) 11

D) 0

Algorithme : pour_x

Data : resultat:entier

```
// initialisation des variables
resultat = 0
// séquence d'opérations
for i ⟵ 1 to 10 do
    for j ⟵ 1 to 10 do
        if i == 3 * j then
            afficher i,"-"
```