Algorithmique et programmation 1

Répétitions - Boucles

L1 M-I-SPI – Université de Lorraine
Marie Duflot-Kremer
avec l'aide des collègues de Nancy et Metz

Transparents disponibles sur la plateforme de cours en ligne

Instructions répétitives

- Deuxième façon de contrôler le flux d'exécution,
- permet de choisir combien de fois exécuter un ensemble d'instructions

Ce qu'on veut :

```
Bonjour !
Bonjour !
Bonjour !
Bonjour !
Bonjour !
Bonjour !
```

Ce qu'on sait faire :

```
# Algorithme affichage
Variables
    # pas besoin de variables
Début
    afficher(" Bonjour ! ")
    afficher(" Bonjour ! ")
    afficher(" Bonjour ! ")
    afficher(" Bonjour ! ")
    afficher(" Bonjour ! ")
    afficher(" Bonjour ! ")
Fin
```

- On devrait pouvoir faire mieux

Boucle

- Besoin de faire quelquechose de manière répétitive :
  - écrire 5 fois "bravo" à l'écran,
  - faire la somme des 6 premiers entiers,
  - écrire tous les multiples de 7 plus petits que 65,
  - faire deviner à l'utilisateur un nombre entre 0 et 20.
- La boucle permet de répéter un morceau d'algorithme autant de fois que nécessaire.

Nombre de répétitions

- Nombre fixe de répétitions
  - ⇝ écrire 5 fois "bravo" à l'écran,
  - ⇝ faire la somme des 6 premiers entiers,
  - ⇝ écrire tous les multiples de 7 strictement plus petits que 65,
  - ⇝ afficher toutes les valeurs contenues dans un tableau.
- Nombre variable de répétitions
  - ⇝ faire deviner à l'utilisateur un nombre entre 0 et 20,
  - ⇝ trouver combien de fois il faut diviser un entier par 2 pour qu'il devienne plus petit que 1.

Boucle Tant que

Tant que *Condition* **Faire**
  Instructions
Fintantque

Principe

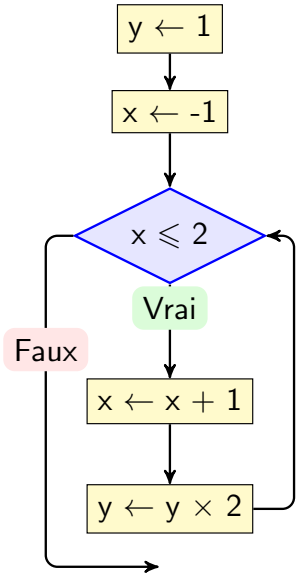
1. on teste la Condition
2. si elle est fausse, on arrête la boucle et on passe à la suite de l'algorithme
3. si elle est vraie on effectue les Instructions puis on retourne au 1.

On va répéter les Instructions **tant que** la condition est vraie.

Boucle Tant que - exemple

```
# Algorithme boucle
# Marie, le 03/10/13
# Une boucle tant que toute simple
Variables
  x, y : entiers
Début
  y ← 1
  x ← -1
  Tant que x ⩽ 2 Faire
    x ← x + 1
    y ← y × 2
  Fintantque
Fin
```

- Que vaut x après la boucle ?
- Que vaut y après la boucle ?



Boucle Tant que - intérêt

- Permet de répéter un ensemble d'instructions,
- nombre fixe/variable de répétitions,
- conditions très riches :
  - pas juste compter jusqu'à 5
  - peut porter sur plusieurs variables

    Tant que *y ⩽ (x**2) + 3*x + 12* **Faire**
  - on peut mettre des **et** , des **ou** ,...

    Tant que *(somme<1000)* **et** *(***non** *drapeau)* **Faire**

Boucle Tant que - dangers

- Si initialement la condition est fausse...
  -
- bien vérifier qu'on fait le bon nombre d'itérations
  - un $<$ au lieu d'un $\leqslant$ et on peut manquer une itération (ou en faire une de trop)

  ⇝ **bien** tester les boucles
- risque de boucle infinie

```
compteur ← 0
somme ← 1
Tant que compteur < 2 Faire
  somme ← somme+1
Fintantque
```

Tant que - somme d'entiers

Spécification

Ecrire un fragment d'algorithme qui, demande un entier `max` à l'utilisateur puis calcule et affiche la somme des entiers allant de 0 à `max`

Tant que - nombre de divisions

Spécification

Ecrire une fonction qui, étant donné un réel `x`, calcule et retourne combien de fois il faut diviser `x` par deux pour obtenir un nombre strictement plus petit que 1.

Tant que - trouver un multiple

Spécification

Ecrire une fonction qui, étant donnés quatre entiers positifs `min`, `max`, `a` et `b`, trouve le plus petit entier compris entre `min` et `max` qui soit multiple de `a` et de `b`. Si un tel entier n'existe pas, on retourne -1.

Tant que - que fait cet algorithme ?

```
d ← 0
Tant que (d ⩽ 9) Faire
    u ← 0
    Tant que (u ⩽ 9) Faire
        afficher (d,u)
        u ← u+1
    Fintantque
    d ← d+1
Fintantque
```

- Cet algorithme va afficher
- la première boucle (extérieure) va être effectuée fois,
- la deuxième boucle (intérieure) va être effectuée fois,

Tant que - que fait cet algorithme ? (2)

```
d ← 0
Tant que (d ⩽ 1) Faire
    d ← d + 2
    afficher (d)
    d ← d - 2
Fintantque
```

- Cet algorithme va afficher

Boucle Pour

```
Pour variable allant de valeur1 à valeur2 Faire
    Instructions
Finpour
```

Principe

1. la valeur de variable est initialisée à valeur1
2. on teste qu'elle ne dépasse pas valeur2
3. si oui on arrête la boucle et on passe à la suite de l'algorithme
4. si non,
   - on effectue les Instructions
   - on augmente la valeur de variable de 1
   - on repart au point 2.

Boucle Pour - variantes

Par défaut la valeur de la variable de contrôle augmente de 1 en 1 mais on peut écrire :

- **Pour** *i allant de 1 à n* **Faire**
  - ⇝ la valeur de fin est dans une une variable
- **Pour** *i allant de 12 à 6 en descendant* **Faire**
  - ⇝ la valeur de la variable de contrôle diminue
- **Pour** *i allant de 1 à 20 de 3 en 3* **Faire**
  - ⇝ la valeur de la variable de contrôle varie d'une valeur choisie
- **Pour** *i allant de 12 à n de 3 en 3 en descendant* **Faire**
  - ⇝ on combine le tout

Boucle Pour - exemples

- Afficher 5 fois "bravo" à l'écran :
- Afficher tous les multiples de 7 strictement plus petits que 65
- Faire la somme des 6 premiers entiers,

Boucle Pour - exemples (2)

Qu'affichent ces boucles ?

- **Pour** *i allant de 1 à 1* **Faire**
  afficher("bravo")
  Finpour

  Réponse :

- n ← 0
  Pour *i allant de 2 à n* **Faire**
  afficher(i)
  Finpour

  Réponse :

- n ← 0
  Pour *i allant de 2 à n en descendant* **Faire**
  afficher(i)
  Finpour

  Réponse :

Pour *vs* Tant que

	Boucle Pour	Boucle Tant que
nb répétitions	fixe	fixe/variable
initialisation	dans l'instruction	avant la boucle
in-dé-crémentation	dans l'instruction	dans le corps de la boucle
condition	dans l'instruction	dans l'instruction

La boucle **Pour** :

- a une notation plus compacte...
- ... mais est plus limitée.

La boucle **Tant que**

- a besoin d'initialisation et incrémentation séparées...
- ... mais offre plus de liberté.

Pour *vs* Tant que (2)

Et le vainqueur est...

- Toute boucle **Pour** peut être traduite en une boucle **Tant que**.
- L'inverse n'est pas vrai.

Pour *i allant de* *12* *à n* *de 3 en 3 en descendant* **Faire**
  somme ← somme + i
Finpour

se traduit en :

La méthode de Héron d'Alexandrie (2)

La méthode de Héron d'Alexandrie (≃50-100 ap JC)[Hé98]

Puisque 720 n'a pas son côté rationnel, on peut obtenir son côté avec une très petite différence comme suit. Comme le premier nombre carré successeur de 720 est 729 qui a 27 pour côté, on divise 720 par 27 . Cela donne 26 $\frac{2}{3}$ .On ajoute 27,ce qui fait 53 $\frac{2}{3}$ et l'on en prend la moitié, soit 26 $\frac{1}{2}\frac{1}{3}$. Le côté de 720 sera par conséquent très proche de 26 $\frac{1}{2}\frac{1}{3}$. En fait, si l'on multiplie 26 $\frac{1}{2}\frac{1}{3}$ par lui-même, le produit est 720 $\frac{1}{36}$, de sorte que la différence sur le carré est $\frac{1}{36}$. Si l'on désire rendre la différence inférieure encore à $\frac{1}{36}$, on prendra 720 $\frac{1}{36}$ au lieu de 729, et en procédant de la même façon, on trouvera que la différence résultante est beaucoup moindre que $\frac{1}{36}$.

Vers un algorithme pour la méthode de Héron

Spécification

- **Entrée :** un entier A
- **Problème :** trouver un réel x tel que $|x^2 - A| < 0,0001$ lorsque A n'est pas un carré.
- **Résultat :** une approximation de $\sqrt{A}$ (un réel)

Analyse

- Trouver le plus petit entier x tel que $x^2 > A$
- Remplacer x par $(A/x + x)/2$
- Recommencer jusqu'à obtenir la précision souhaitée

Un algorithme pour la méthode de Héron

```
# Algorithme de Héron
# Calcule une valeur approchée de la racine carrée
# Marie, 04/10/13
Variables
    carre, premier : entier
    racine : réel
Début
    premier ← 0
    carre ← saisir("Entrez un entier")
    # On va chercher le plus petit entier plus grand que la racine
    Tant que (premier**2) ⩽ carre Faire
        premier ← premier + 1
    Fintantque
    racine ← premier * 1.0                # transforme un entier en réel
    # maintenant on itère le calcul de la racine
    Tant que abs(racine**2-carre) ⩾ 0.0001 Faire     # abs = valeur absolue
        racine ← (carre/racine + racine)/2
    Fintantque
    afficher("La valeur approchée de la racine est",racine)
Fin
```


Boucle while en Python

```
while Condition:
    Instructions
```

Principe

1. même fonctionnement que la boucle Tant que
2. `Tant que` est changé en `while`
3. `Faire` est remplacé par `:`
4. pas de `Fintantque`
5. comme pour les conditionnelles, l'indentation est cruciale

Boucle while - exemple

```
# Les variables ont des noms obscurs pour ne pas vous aider
x = 1
y = 1
while x < 5:
    y = y * x
    x = x + 1
print("Le résultat est", y)
```

- Quelle valeur de y affiche l'algorithme ?
- Que vaut `x` à la fin ?
- Si on remplace `5` par `n`, que calcule cet algorithme ?

Boucle for - ce qui change en Python

Attention

Pour ceux qui ont déjà programmé en autre chose (Java, C, ...) la structure du `for` est différente en python

- Permet de parcourir une séquence d'éléments
  - une suite de nombres
  - mais aussi les caractères d'une chaîne de caractères
  - les éléments d'une liste
  - ...
- Pour chaque élément on va exécuter une (suite d') instruction(s)

Boucle for - fonctionnement

```
for elem in seq:
    Instructions
```

Principe

Pour chaque élément de la séquence `seq`

1. La variable `elem` reçoit cet élément,
2. on effectue les instructions
3. on passe à l'élément suivant, s'il existe.

Si la séquence est vide, la boucle ne va rien faire.

Boucle for - exemple

```
chaine = "Nancy"
for elem in chaine:
    print(elem,"*-*",end=" ")
```

- pourrait se faire avec un `while` ...
- mais plus compact et plus simple que

```
chaine = "Nancy"
index = 0
while index < len(chaine):
    print(chaine[index],"*-*",end=" ")
    index = index + 1
```

- Ce programme va afficher :

La fonction range

"Et si je veux traduire la boucle `Pour` du cours en Python ?"

C'est possible ! ! !

Il suffit de générer une séquence d'entiers, avec la fonction `range` :

- `range(n)` génère une séquence contenant les n premiers entiers naturels
  - ⇝ donc de 0 à n-1,
- `range(m,n)` commence à partir de m
  - ⇝ donc de m à n-1,
- `range(m,n,p)` fixe également le pas p
  - ⇝ si p positif : de m à n-1, de p en p
  - ⇝ si p négatif : de m à n+1, de p en p

`Attention :` la valeur de "fin", ici n, est exclue ! !

For et range

- Calculer et afficher la somme des carrés des entiers de 0 à 10

```
somme = 0
for compteur in :
    somme =
print("La somme des carrés vaut ",somme)
```

- Écrire un bout de programme qui affiche une chanson

```
for compteur in range(10,0,-1):
    print("C'est dans", compteur, "ans je m'en irai")
    print("J'entends le loup et le renard chanter")
    print("J'entends le loup, le renard et la belette")
    print("J'entends le loup et le renard chanter")
print("Chant traditionnel breton remasterisé")
```


Indentation

Pour les boucles aussi, l'indentation est cruciale :

```
while x > 2:
    if y == 2:
        x = x - 1
    else:
        y = 2
        x = x + 1
print("nouvelle valeur : ", x)
```

```
while x > 2:
    if y == 2:
        x = x - 1
    else:
        y = 2
    x = x + 1
print("nouvelle valeur : ", x)
```

Si au début `x` vaut 6 et `y` vaut 1 les programmes affichent :

-
-

Combien fait-on de tours de boucle ?

-
-

Un programme pour la méthode de Héron

```
# Programme Héron
# Calcule une valeur approchée de la racine carrée
# Marie, 04/10/13
# Variables
# carre, premier : entier
# racine : réel
premier = 0
carre = int(input("Entrez un entier"))
# On va chercher le plus petit entier plus grand que la racine
while (premier**2) <= carre:
    premier = premier + 1
racine = float(premier)                    # transforme un entier en réel
# maintenant on itère le calcul de la racine
while abs(racine**2-carre) >= 0.0001:              # abs = valeur absolue
    racine = (carre/racine + racine)/2
print("La valeur approchée de la racine est",racine)
```


Commenter une boucle

Recherche

On cherche un caractère dans un texte. Si on le trouve on donne la position, sinon on dit qu'on a échoué.

```
texte = input("Tapez une chaîne de caractères")
x = 0
trouve = False
while (x < len(texte)) and (not trouve):
                # on continue tant qu'on n'est pas à la fin du mot
                    # et qu'on n'a pas trouvé la lettre cherchée
    if texte[x]=="c":                # Si on trouve la lettre désirée
        trouve = True                    # On signale qu'on a trouvé
    x = x+1    # dans tous les cas on passe à la prochaine lettre

      # ici soit on a atteint la fin du mot soit on a trouvé un c
if trouve:                      # On s'est arrêté parce qu'on a trouvé
    print("La lettre c a été trouvée en position",
else:
                          # On est sorti de la boucle à la fin du mot
    print("La lettre c n'a pas été trouvée")
```

Sources

- G.Swinnen, *Apprendre à programmer avec Python 3 (3ème édition*, Eyrolles, 2012, Disponible en ligne à l'adresse `http://inforef.be/swi/python.htm`.
- Héron, *Metrica III,I,8, d'après : Caveing M., l'irrationalité dans les mathématiques grecques jusqu'à euclide*, p. 28, Septentrion, 1998.