Bases de la programmation impérative

Flot de contrôle

Ensimag 1ère année

Sommaire

Booléens

```
1  def multiple_de_6(entier):
2      """
3      Renvoie si l'entier donne est un
           multiple de 6.
4      """
5      multiple_de_2 = (entier % 2) == 0
6      print(type(multiple_de_2))
7      multiple_de_3 = (entier % 3) == 0
8      return multiple_de_2 and
           multiple_de_3
9
10 def main():
11     print(multiple_de_6(3))
12     print(multiple_de_6(12))
```

- vrai ou faux (True, False)
- opérateurs logiques : and, or, not

Implémentation

- entier restreint à $\{0, 1\}$
  - consomme beaucoup plus d'espace qu'un bit
- `False` correspond à 0
- `True` correspond à 1
- `["a", "b"][False]` s'évalue à `"a"`
  - souvent utile pour éviter les conditionnelles

Conditionnelle

```
1  def repondre(message):
2      if message == "
           bonjour":
3          print("bonjour")
4      else:
5          print("merci de
               saluer d'
               abord")
```

- 2 branches, choisies en fonction du booléen
- délimitées par l'indentation
- la seconde branche est optionnelle
- souvent difficile à prédire, y-compris lors de l'exécution
  - surcoût en temps
- complexifie l'analyse du programme
  - ne pas imbriquer trop de conditionnelles

Conversions implicites

- de nombreuses expressions peuvent s'évaluer en booléens
- sont considérés comme `False` :
  - `False`
  - `None`
  - 0, 0.0
  - `""`, `()`, `[]`
  - `...`

Conversions implicites

- de nombreuses expressions peuvent s'évaluer en booléens
- sont considérés comme `False` :
  - `False`
  - `None`
  - 0, 0.0
  - `""`, `()`, `[]`
  - ...

Exemple

```
1  def vecteur_vide(vecteur):
2      if not vecteur:
3          print("le vecteur est vide")
```

Chaînage

```
1  def identification(variable):
2      if isinstance(variable,
           int):
3          print("la variable est
               un entier")
4      elif isinstance(variable,
           float):
5          print("la variable est
               un flottant")
6      else:
7          print("je ne sais pas
               ce que c'est")
```

- `elif`
- évite de faire exploser le niveau d'indentation
- mettre du plus probable au moins probable

Gestion des cas spéciaux

```
1  from math import sqrt
2
3  def racine(entier):
4      """
5      Renvoie la racine de l'entier positif donne
           ou None s'il est negatif.
6      """
7      # je verifie que tout est ok en entrant dans
           la fonction
8      if entier < 0:
9          return
10     # plus de probleme, je peux meme re-indenter
           normalement
11     return sqrt(entier)
```

Sommaire

Boucle sur les vecteurs

```
1  #!/usr/bin/env python3
2  def main():
3      vecteur = [3, 1, 2, 4]
4      for element in vecteur:
5          print(element)
6
7  main()
```

```
3
1
2
4
```

Opération essentielle

- une des plus utiles
- attention au coût en temps

Énumérer les éléments

```
1       positions = [(0.0, 0.0), (1.0, 2.2), (3.0,
            2.4)]
2       for index, position in enumerate(positions):
3           print("la position numero {} est {}".
                format(index, position))
```

Boucler sur deux vecteurs

```
1       animaux = ["oiseau", "tigre", "cheval"]
2       couleurs = ["rouge", "vert", "bleu"]
3       for animal, couleur in zip(animaux, couleurs)
            :
4           print(animal, couleur)
```

```
oiseau rouge
tigre vert
cheval bleu
```

Attention !

Arrêt à la fin du plus court.

Énumérer un vecteur de couples

```
1       points = [(1.0, 1.0), (0.5, 0.2), (3.1, 3.2)]
2       for index, (x, y) in enumerate(points):
3           print("point {} a {},{}".format(index, x,
                y))
```

Ranges

- objet représentant un intervalle entier
- syntaxe : `range(max+1)` ou `range(min, max+1)` ou `range(min, max+1, pas)`
- utilisable pour boucler : `for chiffre in range(0, 10)`
- utilisable pour tester : `if nombre in range(0, 10):`

Ranges

- objet représentant un intervalle entier
- syntaxe : `range(max+1)` ou `range(min, max+1)` ou `range(min, max+1, pas)`
- utilisable pour boucler : `for chiffre in range(0, 10)`
- utilisable pour tester : `if nombre in range(0, 10):`

À éviter

```
1  for index in range(len(vecteur)):
2      element = vecteur[index]
```

While

- certaines boucles sont plus irrégulières
- on boucle tant qu'une condition est vérifiée

```
1  def division(dividende, diviseur):
2      """
3      Calcule manuellement le quotient et le reste
           de la division du
4      dividende par le diviseur.
5      """
6      quotient = 0
7      reste = dividende
8      while reste >= diviseur:
9          reste -= diviseur
10         quotient += 1
11     return quotient, reste
```

Interruptions

```
1  def somme(vecteur_aiguille, vecteur_foin
       ):
2      """
3      Renvoie la somme des elements pairs
           du vecteur aiguille
4      presents dans le vecteur foin.
5      """
6      somme = 0
7      for aiguille in vecteur_aiguille:
8          if aiguille % 2:
9              continue  # on saute les
                   impairs
10         for foin in vecteur_foin:
11             if aiguille == foin:
12                 somme += aiguille
13                 break
14     return somme
```

- `break` quitte la boucle
- `continue` quitte l'itération courante
- `return` quitte la fonction courante

Opérations optimisées

- certaines itérations courantes sur les vecteurs disposent d'une implémentation de bas niveau
- à privilégier pour des raisons de performance et de clarté
  - sum : `somme = sum(vecteur)`
  - min, max : `element_max = max(vecteur)`
  - in : `if element in vecteur:`
  - index : `index_de_3 = vecteur.index(3)`

Opérations optimisées

- certaines itérations courantes sur les vecteurs disposent d'une implémentation de bas niveau
- à privilégier pour des raisons de performance et de clarté
  - sum : `somme = sum(vecteur)`
  - min, max : `element_max = max(vecteur)`
  - in : `if element in vecteur:`
  - index : `index_de_3 = vecteur.index(3)`

Attention !

Toutes ces opérations bouclent sur le vecteur.

Variants, Invariants

- peut-on analyser le comportement d'une boucle ?

Variants, Invariants

- peut-on analyser le comportement d'une boucle ?
  - variant : propriété changeant à chaque itération
    - utile pour montrer la complétion des calculs
  - invariant : propriété vraie à toute itération
    - donne une garantie sur l'état à la sortie
    - permet des preuves simples, par récurrence

Analyse : Exemple

```
1  def division(dividende, diviseur):
2      """
3      Calcule manuellement le quotient et le reste
           de la division du
4      dividende par le diviseur.
5      """
6      quotient = 0
7      reste = dividende
8      while reste >= diviseur:
9          reste -= diviseur
10         quotient += 1
11     return quotient, reste
```

Analyse : Exemple

```
1  def division(dividende, diviseur):
2      """
3      Calcule manuellement le quotient et le reste
           de la division du
4      dividende par le diviseur.
5      """
6      quotient = 0
7      reste = dividende
8      while reste >= diviseur:
9          reste -= diviseur
10         quotient += 1
11     return quotient, reste
```

- variant : contenu de `reste` ; baisse du diviseur, tout en restant positif $\rightarrow$ on finit par s'arrêter

Analyse : Exemple

```
1  def division(dividende, diviseur):
2      """
3      Calcule manuellement le quotient et le reste
           de la division du
4      dividende par le diviseur.
5      """
6      quotient = 0
7      reste = dividende
8      while reste >= diviseur:
9          reste -= diviseur
10         quotient += 1
11     return quotient, reste
```

- variant : contenu de `reste` ; baisse du diviseur, tout en restant positif $\rightarrow$ on finit par s'arrêter
- invariant : à chaque fin d'itération on a :

  `reste + quotient*diviseur = dividende`