Boucles et conditions

Cours

Lycée Jean Zay – 21 rue Jean Zay – 63300 Thiers – Académie de Clermont-Ferrand

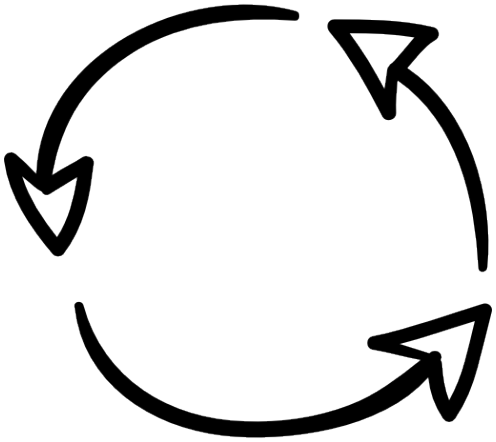

Table des matières

1 Instructions conditionnelles

1.1 Les variables booléennes

L'utilisation des boucles/conditions va nous conduire à devoir effectuer des tests pour obtenir des conditions.

Pour cela, introduisons les variables de type booléennes. Elles valent 1, ou 0, ou plutôt, `True` (juste) ou `False` (faux). Pour obtenir une variable booléenne, il faut effectuer un test :

`==`	Vérifie l'égalité
`!=`	Vérifie la différence
`>`	Comparaison
`>=`	
`<`	
`<=`	
`a <= b < c`	On peut écrire des inégalités à plusieurs comparaisons
`in`	`'A' in ['A','B']` renvoie True

Les résultats de ces tests sont soit `True`, soit `False`, on peut effectuer des opérations entre variables booléennes :

`and`	Fonction « et »
`or`	Fonction « ou »

`not(Cond)`	Donne la condition inverse :
	`not(True)` → `False`
	`not(False)` → `True`

1.2 Tests simples

Une instruction conditionnelle n'est exécutée que si une condition donnée est vérifiée. Pour traduire cela, on utilise l'instruction `if`, qui a en Python la syntaxe suivante :

```
1 if condition:
2     bloc_d_instruction
```

Exemple :

Exprimer la valeur de la variable `x` après exécution en fonction de sa valeur initiale dans chacun des trois cas ci-dessous.

```
1 y=x*x
2 if y%2==0:
3     y=y+1
4 x=x+y
```

```
1 y=x*x
2 if y%2==0:
3     y=y+1
4     x=x+y
```

```
1 y=x*x
2 if y%2==0:
3     y=y+1
4         x=x+y
```

1.3 Tests avec alternative

On enrichit la syntaxe des tests pour proposer une alternative lorsqu'une condition n'est pas vérifiée. Pour traduire cela, on utilise l'instruction `else`, qui a en Python la syntaxe suivante :

```
if condition :
    bloc_d_instructions_si_la_condition_est_vérifiée
else :
    bloc_d_instructions_si_la_condition_n_est_pas_vérifiée
```

Exemple :

Écrire en langage Python une suite d'instructions qui transforme un entier n en sa moitié s'il est pair, en $3n+1$ sinon.

Lorsque qu'une situation se décompose en plus de deux cas (mutuellement exclusifs), on utilise l'instruction `elif`.

Exemple :

On s'intéresse aux racines du polynôme $P(X)=\mathtt{a}X^2+\mathtt{b}X+\mathtt{c}$ à coefficients réels.

```
Delta = b**2 - 4*a*c
if Delta < 0:
    print('P admet deux racines non réelles conjuguées')
if Delta == 0:
    print('P admet une racine double réelle')
else :
    print('P admet deux racines simples réelles')
```

Exemple :

Exprimer la valeur de la variable `x` après exécution dans chacun des deux cas ci-dessous.

```
x=9
if x%4==0:
    x=2*x
if x%4==1:
    x=x+2
if x%4==2:
    x=x-1
if x%4==3:
    x=x**2
```

```
x=9
if x%4==0:
    x=2*x
elif x%4==1:
    x=x+2
elif x%4==2:
    x=x-1
else :
    x=x**2
```

2 Boucles `while`

2.1 Implémentation d'un compteur

L'utilisation de boucles va souvent être associée à l'utilisation d'un compteur. Ce compteur sera toujours initialisé avant la boucle : Compteur = 0 Puis incrémenté, par exemple de 1, à chaque itération, en écrivant au choix :

- `Compteur = Compteur + 1`
- `Compteur += 1`

2.2 Syntaxe

En Python, la syntaxe d'une boucle conditionnelle est :

```
1 while condition:
2     bloc_d_instructions
```

La fin du bloc d'instructions est marquée par le retour au niveau d'indentation du `while`.

Attention

Lorsque l'on rentre dans cette boucle, la condition est vérifiée. Si on ne modifie pas cette condition dans la boucle, le programme y restera indéfiniment !

Exemple :

On suppose avoir construit une fonction prim qui prend un entier n en entrée et retourne 'premier' si n est premier ou 'non_premier' sinon.
Que retourne la série d'instructions suivantes :

```
1 a=2
2 compteur=0
3 while compteur<10:
4     if prim(a) == 'premier':
5         print(a)
6     compteur += 1
7     a=a+1
```

2.3 Terminaison de boucle

Lorsqu'on écrit une boucle `while`, il est utile de savoir justifier que cette boucle se termine effectivement.

Exemple :

Justifier que la boucle `while` de l'exemple précédent se termine.

Exemple :

Justifier que les boucles des exemples ci-dessous se terminent.

```python
n=0
while 2**n<1000:
    n=n+1
print(n)
```

```python
val=0
while n % 2==0:
    val = val+1
    n=n / / 2
print(val)
```

```python
n=42
while n>1:
    if n%2==0:
        n=n//2
    else :
        n=3*n+1
```

où `n` est un entier positif

3 Boucles `for`

3.1 Syntaxe

Lorsqu'on connait à l'avance le nombre d'itérations qu'il faudra effectuer, les boucles conditionnelles deviennent inutiles.

```python
for i in range(n):
    bloc_d_instructions
```

Exemple :

Que retourne la suite d'instructions ci-dessous où `n` désigne un entier positif :

```python
p=1
for i in range(n):
    p=2*p
```

Exemple :

Que retourne les suites d'instructions ci-dessous :

```python
t=[2,5,2]
for e in t:
    print(e**2)
```

```python
chaine='Georges Perec'
compt=0
for c in chaine:
    if c=='e':
        compt+=1
print(compt)
```

```python
E={2,5,2}
for e in E:
    print(e**2)
```

3.2 Invariant de boucle

Lorsque l'on a écrit un programme il reste à vérifier qu'il est correct, autrement dit qu'il calcule bien ce que l'on attend. Pour cela on peut utiliser un *invariant de boucle*, c'est-à-dire une propriété qui :

- est vérifiée avant d'entrer dans la boucle,
- si elle est vérifiée avant une itération, est vérifiée après celle-ci,
- lorsqu'elle est vérifiée en sortie de boucle permet d'en déduire que le programme est correct.

Exemple :

Montrer que, dans l'exemple 1 ci-dessus, en notant p_i la valeur de la variable `p` après i itérations, la propriété "pour tout itération i, on a $p_i = 2^i$" est un invariant de boucle.

3.3 Remarques

Remarque *Break*

On peut stopper une boucle `for` avec la commande `break` :

```
1 for i in range(5):
2     print(i)
3     if i==3:
4         break
```

Qui renvoie après exécution :

```
0
1
2
3
```

Remarque *Parcourir une liste*

On peut parcourir des termes d'une liste avec `for t in L` . Je vous recommande de ne pas écrire `for i in L` afin de ne pas confondre i avec un indice...

```
1 L=[3,7,9]
2 for i in range(len(L)):
3     terme=L[i]
4     print(terme)
```

```
1 L=[3,7,9]
2 for terme in L):
3     print(terme)
```

Remarque *Modification de l'indice*

La modification de l'indice dans une boucle for n'a aucun effet. C'est la boucle for qui pilote l'incrémentation de i et donc au pas suivant "écrase" la valeur que l'on aurait modifiée.

```
1 n=5
2 for i in range(n):
3     print('Avant : ',i)
4     i=i+2
5     print('Après : ',i)
```

Qui renvoie après exécution :

```
Avant :  0
Après :  2

Avant :  1
Après :  3

Avant :  2
Après :  4

Avant :  3
Après :  5

Avant :  4
Après :  6
```

Remarque *Deux boucles avec le même indice*

Le fait d'utiliser deux fois la lettre i ne pose pas de problèmes à l'exécution du code comme si l'on avait utilisé i et j. Comme précédemment, la modification de i dans la seconde boucle for ne modifie las l'enchaînement des i de la première boucle.

```
1 for i in range(4):
2     print('i1 : ',i)
3     for i in range(2):
4         print('i2 : ',i)
```

Qui renvoie après exécution :

```
i1 :  0
i2 :  0
i2 :  1

i1 :  1
i2 :  0
i2 :  1

i1 :  2
i2 :  0
i2 :  1

i1 :  3
i2 :  0
i2 :  1
```

4 `for` ou `while`

Lorsque l'on connait à l'avance le nombre d'itération, la structure de boucle `for` est à privilégier car plus simple d'écriture et plus sûre. Ces deux codes sont "identique", mais celui de gauche est le bon :

```
1 L=[1,5,3,2,5,9,2]
2 Taille=len(L)
3 for i in range(Taille):
4     Terme=L[i]
5     print(Terme)
```

```
1 L=[1,5,3,2,5,9,2]
2 Taille=len(L)
3 i=0
4 while i<Taille:
5     Terme=L[i]
6     print(Terme)
7     i=i+1
```

5 Pour s'entraîner

5.1 Compréhension d'algorithmes

Q1 Pour chacun des algorithmes suivants, donner ce qui est affiché en fin d'exécution du programme.

N°	Programme	Résultat

1	`n = 10`
	`L = [2*i for i in range(n)]`
	`Resultat = 0`
	`for i in range(len(L)):`
	`    Resultat = L[i]`
	`print(Resultat)`	

2	`n = 5`
	`L = [i for i in range(1,n+1)]`
	`Resultat = 0`
	`for i in range(len(L)):`
	`    Resultat += L[i]`
	`print(Resultat)`	

3	`n = 10`
	`Resultat = 0`
	`for i in range(n):`
	`    if i%2==0:`
	`        Resultat += i`
	`print(Resultat)`	

4	`L = [0,1,58,50,74,65,1,2,9,86,5,45,1,71,23,25,74,65,19,37,50]`
	`T = 5`
	`while len(L)> T:`
	`    L.pop()`
	`Resultat = L`
	`print(Resultat)`	

5	`L = [0,1,58,50,74,65,1,2,9,86,5,45,1,71,23,25,74,65,19,37,50]`
	`Resultat_1 = []`
	`Resultat_2 = []`
	`Resultat_3 = []`
	`Med = 50`
	`for i in range(len(L)):`
	`    if L[i] < Med:`
	`        Resultat_1.append(L[i])`
	`    elif L[i] > Med:`
	`        Resultat_2.append(L[i])`
	`    else:`
	`        Resultat_3.append(L[i])`
	`print(Resultat_1," - ",len(Resultat_1))`
	`print(Resultat_2," - ",len(Resultat_2))`
	`print(Resultat_3," - ",len(Resultat_3))`	

5.2 Détection des erreurs de programmation

Q2 Pour chacun des algorithmes suivants, trouvez les erreurs qui empêchent son exécution ou qui donnent un résultat faux

N°	Programme	Erreurs

1	`Age = input(Entrer votre age :)`
	`Delta = 100 - Age`
	`print("Dans ",Delta," ans, vous aurez 1 siècle !!!")`	

2	`Notes = [10.5,5,20,18,12]`
	`Somme = 0`
	`for i in range(Notes)`
	`    Somme += Notes[i]`
	`Moyenne = Somme / len(Notes)`
	`print("Moyenne: ",Moyenne)`	

3	`L = [1,2,4,6,10,25,32,44,56,74,82,98]`
	`Sol = []`
	`while L[i] <= 50 : # Récupération des valeurs <=50`
	`    print(i,L[i])`
	`    i =+ 1`
	`    Sol.append(L[i])`
	`print(Sol)`	

4	`Binaire = '11001'`
	`Decimal = 0`
	`for i in range(len(Binaire)):`
	`    P_2 = 2^i`
	`Decimal += int(Binaire[i]) * P_2`
	`print(Decimal)`	