Structures algorithmiques

1 Instructions conditionnelles

1.1 Test simple

Définition

Une instruction conditionnelle est une instruction pouvant s'exécuter seulement si une condition est vérifiée par l'état courant.

Pour en créer une en Python, on utilise le mot-clé **if**, avec la syntaxe suivante :

python
```
if condition :
    #bloc d'instructions
```

A la suite de **if** et avant les **:**, il est possible de mettre toute expression booléenne.

1.2 Opérateurs de comparaison

Algorithme

1 $2 = 8$
2 $2 \neq 8$
3 $2 \geq 8$
4 $2 > 8$
5 $2 \leq 8$
6 $2 < 8$

python
```
>>> 2==8
      False
>>> 2!=8
      True
>>> 2>=8
      False
>>> 2>8
      False
>>> 2<=8
      True
>>> 2<8
      True
```

1.3 Opérateurs entre booléens

1.4 Test avec alternative

On souhaite programmer la relation de récurrence liée à la suite de Syracuse. Cette dernière est définie par récurrence avec la formule suivante :

$$u_0 = \mathrm{N} \in \mathbb{N} \text{ et pour tout entier } n \in \mathbb{N}, u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } U_n \text{ est pair} \\ 3u_n + 1 & \text{si } U_n \text{ est impair} \end{cases}$$

Un essai naïf pourrait être :

python
```
Un=20
if Un%2==0:
    Un1=Un/2
if not(Un%2==0):
    Un1=3*Un+1
```

Si ce programme a le mérite de fonctionner, il utilise deux tests au lieu de un (ce qui coûte du temps tant au programmeur qu'à la machine).

Pour pallier à ces problèmes, on a enrichi la syntaxe des instructions conditionnelles de tests avec alternative (si ... sinon...) :

```
if condition :
    #bloc d'instructions si la condition est réalisée

else :

    #bloc d'instructions si la condition n'est pas réalisée
```

Par exemple la relation de récurrence de la suite de Syracuse s'écrit :

1.5 Tests imbriqués

Il est possible de réaliser une instruction conditionnelle dans une instruction conditionnelle. On parle alors de tests imbriqués.

Lorsque les tests imbriqués ne servent qu'à séparer une situation en plus de deux cas de figures, l'indentation obligatoire en Python peut rendre difficile la compréhension du programme.

Par exemple considérons le critère de réussite au bac suivant :

```
N=int(input())
if (N<8):
    print("Echec au bac")
else:
    if (N<10):
        print("Rattrappage")
    else :
        if(N<12):
            print("Bac obtenu au premier tour")
        else:
            if(N<14):
                print("Mention AB")
            else:
                if(N<16):
                    print("Mention B")
                else: #On a alors N>=16
                    print("Mention TB")
```

L'indentation rend la lecture de ce programme (pourtant très simple) particulièrement peu aisée et peu naturelle. Il est possible de mettre toutes ces instructions au même niveau

en utilisant le mot clé **elif** qui est une contraction de *else* et *if*.

Sur l'exemple précédent cela donnerait :

```python
N=int(input())
if (N<8):
    print("Echec au bac")
elif (N<10):
    print("Rattrappage")
elif (N<12):
    print("Bac obtenu au premier tour")
elif (N<14):
    print("Mention AB")
elif (N<16):
    print("Mention B")
else: # Il n'est pas nécessaire de spécifier la dernière alternative
    print("Mention TB")
```

1.6 Application 1

Écrire le programme qui affichera le nombre de jours dans un mois de l'année, connaissant le numéro du mois. Le numéro du mois est compris entre 1 et 12 inclus. On rappelle que si le numéro est 2, alors le mois à 28 jours. Sinon, si le numéro est strictement inférieur à 8, alors si il est pair, le mois à 30 jours, et 31 jours s'il est impair. Si le numéro est supérieur ou égal à 8, alors si il est pair, le mois à 31 jours, et 30 jours s'il est impair.

2 Manipulation de variables booléennes

2.1 Information booléenne

Une variable booléenne est un objet mathématique qui ne peut prendre que deux valeurs (`False` ou `True`, 0 ou 1). Elle porte une information (sur le monde réel, sur l'état du système numérique, etc.). Elle peut représenter :

- une assertion sur le monde réel (le prof est absent, le ciel est bleu) ou numérique (un email a été reçu)
- un ordre de fonctionnement (allumer la lumière, envoyer l'email)

Elle est notée, comme les autres variables informatiques, sous la forme d'une chaine de caractères alphanumériques.

Exemple :

Variable	information modélisée par la variable.
m	le métro est en grêve
t	ce jour est travaillé
P	venir au lycée à pied
M	venir au lycée en métro

Une combinaison d'assertion s'obtient par des opérations entre les variables booléennes. Par exemple :

Venir à pied au lycée = Métro en grève ET Journée travaillée

$\iff$ P = m ET t

Venir en métro au lycée = NON(Métro en grève) ET Journée travaillée

$\iff$ M = NON(m) ET t

2.2 Algèbre de Boole

L'algèbre de Boole formalise les opérations sur les booléns, nous ne retiendrons que deux opérateurs : "and" et "or" et une transformation, appelé complémentaire, inversion ou contraire : "not"

Opérateur and

`A and B` est `True` $\Longleftrightarrow$
`A` est `True` et `B` est `True`.

A	B	A and B
False	False	False
True	False	False
False	True	False
True	True	True

Opérateur or

`A or B` est `True` $\Longleftrightarrow$
`A` est `True` ou `B` est `True`.

A	B	A or B
False	False	False
True	False	True
False	True	True
True	True	True

Opérateur not

Le contraire de `A` est noté : `not A`

A	not(A)
False	True
True	False

2.2.1 Propriétés d'algèbre

Certaines propriétés des opérateurs permettent la simplification des expressions booléennes :

- Associativité :
  - ⋆ `(A and B) and C` $\Longleftrightarrow$ `A and (B and C)`
  - ⋆ `(A or B) or C` $\Longleftrightarrow$ `A or (B or C)`
- Commutativité de « and » et de « or » :
  - ⋆ `A and B` $\Longleftrightarrow$ `B and A`
  - ⋆ `A or B` $\Longleftrightarrow$ `B or A`
- Distributivité :
  - ⋆ `A and (B or C)` $\Longleftrightarrow$ `(A and B) or (A and C)`
  - ⋆ `A or (B and C)` $\Longleftrightarrow$ `(A or B) and (A or C)`
- Existence d'éléments neutres :
  - ⋆ `True and A` $\Longleftrightarrow$ `A`
  - ⋆ `False or A` $\Longleftrightarrow$ `A`
- Eléments absorbants :
  - ⋆ `True or A` $\Longleftrightarrow$ `True`
  - ⋆ `False and A` $\Longleftrightarrow$ `False`

De plus, l'opération « `not` » a certaines propriétés particulières supplémentaires :

- `not(not A)` $\Longleftrightarrow$ `A`
- `A or not A` $\Longleftrightarrow$ `True`
- `A and (not A)` $\Longleftrightarrow$ `False`

Enfin, les lois de De Morgan :

- `not (A or B)` $\Longleftrightarrow$ `(not A) and (not B)`
- `not (A and B)` $\Longleftrightarrow$ `(not A) or (not B)`

2.3 Evaluation et précédence

Lors de l'évaluation d'une expression booléenne, les opérations sont réalisées en respectant les priorités :

1. les opérations entre paranthèses
2. `not`
3. `and`
4. `or`
5. enfin, en suivant l'ordre de lecture de gauche à droite.

2.4 Caractère « paresseux » de l'évaluation en python

En python, l'évaluation des opérations n'est réalisée que si cela est strictment nécessaire. Si un résultat peut être connu avant la fin de l'expression, alors les opérations restantes ne sont pas évaluées. Par exemple :

L'expression `x and y` évalue d'abord `x` ; si `x` est `False`, le résultat sera `False` quelque soit la valeur de `y`, qu'il devient inutile de vérifier.

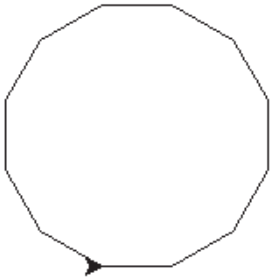
De même, avec l'expression `x or y` si `x` est `True`, `y` est sans influence sur le résultat.

3 Boucles inconditionnelles

3.1 Exemple introductif

Tracer un carré de côté 40mm

Tracer un dodécagone de côté 5mm



3.2 Syntaxe

```python
for variable in iterable :
    # bloc d'instruction
```

Ici, **itérable** peut être tout élement de type itérable (liste, chaîne de caractère, tuple, etc.) L'expression **range** est donc un itérable :

range(n) renvoie la liste des entiers entre 0 et $n-1$, classés par ordre croissant ;

range(m, n) renvoie la liste des entiers entre m et $n-1$, classés par ordre croissant ;

range(m, n, p) renvoie la liste d'entiers dont le plus petit est m, chaque élément est incrémenté de p par rapport au précédent et le dernier élément est strictement inférieur à n.

3.3 Usage avec des suites numériques

3.3.1 Exemple 1

On cherche à afficher les 100 premiers termes de la suite u_n (n variant de 0 à 99) :

$$u_n = 2^n + n$$

Le programme associé serait :

```python
for i in range(0,100):
    print(2**i+i)
```

3.3.2 Exemple 2 : récurrence

On cherche à afficher les 100 premiers termes de la suite v_n : (n variant de 0 à 99) :

$$\begin{aligned} v_0 &= 3 \\ v_{n+1} &= v_n^2 + n \end{aligned}$$

Le programme associé serait :

```python
n=100
vn=3
for i in range(0,n):
    vn1=vn**2+i
    print(vn1)
    vn=vn1
```

3.3.3 Exemple 3 : somme des termes d'une suite

On cherche à calculer et à afficher la somme des $n = 100$ premier termes de la suite v_n vue précédemment :

$$S = \sum_{i=0}^{99} v_i$$

Le programme associé serait :

```python
n=100
vn=3
S=vn
for i in range(0,n):
    vn1=vn**2+i
    print(vn1)
    S=S+vn1
    vn=vn1
print(S)
```

3.4 Vocabulaire

3.5 Quelques conseils

1. Repérer les opérations qui doivent se répéter (introduire un compteur si nécessaire) ;
2. Mettre en place la récurrence ;
3. Associer une ou des variables aux termes de la récurrence ;
4. Déterminer le nombre de fois que ces opérations doivent se répéter ;
5. Prévoir les instructions d'initialisation nécessaires.

3.6 Application 2

Ecrire un programme qui calcule et affiche la somme des n premières puissances de 2 (où n est renseigné par l'utilisateur) :

$$S = \sum_{i=0}^{n-1} 2^i$$

3.7 Boucles `for` à connaître

Somme : $\sum_{k=0}^{n} a_k$	`s=0`
	`for k in range(0,n+1):`
	`  s=s+ak`

Produit $\prod_{k=0}^{n} a_k$	`p=1`
	`for k in range(0,n+1):`
	`  p=p*ak`

Conjonction de variables binaires : $(a_0$ `and` a_1 `and ... and` $a_n)$	`v=True`
	`for k in range(0,n+1):`
	`  v = v and ak`

Disjonction de variables binaires : $(a_0$ `or` a_1 `or ... or` $a_n)$	`v=False`
	`for k in range(0,n+1):`
	`  v = v or ak`

Suite récurrente : $\begin{cases} u_0 = u0 \\ u_{n+1} = f(u_n) \end{cases}$	`# initialisation`
	`u=u0`
	`for k in range(0,n+1):`
	`  # hérédité et décalage de rang`
	`  u=f(u)`

Suite récurrente d'ordre 2 : $\begin{cases} u_0 = u0 \\ u_1 = u1 \\ u_{n+2} = f(u_n, u_{n+1}) \end{cases}$	`# initialisation`
	`u,v=u0,u1`
	`for k in range(0,n+2):`
	`  # hérédité`
	`  w=f(u,v)`
	`  # décalage de rang`
	`  u=v`
	`  v=w`

4 Boucle conditionnelle

A l'inverse d'une boucle inconditionnelle (dont on sait par avance le nombre d'exécutions), il se peut que l'on souhaite répéter un bloc d'instruction un nombre *a priori* inconnu de fois. Par exemple, lorsque le programme a trouvé une valeur recherchée.

4.1 Exemple introductif

Un programme doit vérifier le mot de passe d'un utilisateur. Si le mot de passe saisi correspond à celui attendu, le programme affiche äccès accordé.̈ Sinon, le programme redemande le mot de passe jusqu'à ce que celui-ci soit bon. Le début du programme est indiqué ci-dessous. Compléter le.

```
secret="PCSI"
mdp=input("Tapez votre mot de passe :")
```

4.2 Syntaxe

```
while condition vraie :
    #bloc d'instruction
```

4.3 Quelques conseils

1. Comme pour la boucle **for**, il faut rechercher la relation de récurrence et la reformuler en algorithme ;
2. Prévoir un compteur si nécessaire ;
3. Associer avec cette relation une condition d'arrêt (la condition d'arrêt doit dépendre de la relation de récurrence) ;
4. Prévoir les instructions d'initialisation nécessaires.

4.4 Exemple

Ecrire un programme détaillé permettant de vérifier que un entier n strictement supérieur à 1 est premier.

Approche naïve

Dans un premier temps, on considère $n = 7$.

```python

if 7%2!=0 and 7%3!=0 and 7%4!=0 and 7%5!=0 and 7%6!=0:
    print(True)
```

Utilisation d'une boucle `for`

Mettre en place le même test avec une boucle `for` et n quelconque. Il faudra utiliser un accumulateur.

```python
A=True
for i in range(2,n)):
    A=A and n%i!=0
if A:
    print(True)
```

Utilisation d'une boucle `while`

Optimisation du code : le programme doit s'arrêter dès qu'une division entière est possible.

```python
A=True
i=2
while A and i<n:
    A=A and n%i!=0
    i=i+1

if A:
    print(True)
```

4.5 Application 3

Ecrire un programme qui détermine et affiche le plus petit nombre entier n nécessaire pour que la somme des n premières puissances de 2 soit supérieur à 1000.

$$\sum_{i=0}^{n-1} 2^i \geq 1000$$