INSTRUCTIONS CONDITIONNELLES
INSTRUCTIONS ITERATIVES

I) Instructions

Algorithme et programme

Un **algorithme** est une procédure permettant de résoudre un problème, procédure suffisamment détaillée pour que l'on puisse suivre ses étapes sans compétence particulière concernant le problème à résoudre.

On trouve différents algorithmes classiques en Mathématiques : algorithme de la division euclidienne, algorithme de la multiplication, algorithme d'écriture d'un entier dans une base autre que 10, algorithme d'Euclide, algorithme d'Horner, algorithme permettant de tracer les bissectrices dans un triangle etc.… On en rencontre également dans d'autres domaines : en informatique (Algorithme Page-Rank de Google,…), en génétique (algorithme de sélection naturelle), en cuisine (recette d'un plat), en bricolage (montage d'une étagère…), …

Un **programme** est la traduction d'un algorithme en un langage interprétable par la machine et compréhensible par l'homme. L'exécution d'un programme commence de la première instruction et s'effectue en suivant des règles précises. Le parcours des instructions tout au long de l'exécution est le *flot d'exécution*

Instructions

Une **instruction** est un ordre de modification de l'état courant des variables. Il y en a de plusieurs types :

- La déclaration, l'initialisation ou l'affectation d'une variable ont déjà été vues.
- la séquence ou **le bloc d'instructions** qui exécute des instructions successivement dans l'ordre où on les a écrites
- le test, ou **l'instruction conditionnelle**, qui sert à n'exécuter une instruction (ou un bloc d'instructions) que dans certains états
- la boucle, ou **l'instruction itérative**, qui exécute plusieurs fois une même instruction.

II) Instructions conditionnelles

Test simple

Une **instruction conditionnelle** n'est exécutée que si une condition donnée est vérifiée par l'état courant.

Pour écrire cette condition, on utilise l'instruction **if** qui utilise en Python la syntaxe suivante :

if *condition* **:**

bloc_d_instructions

Le bloc d'instructions n'est exécuté que si la *condition* est vérifiée.

Par exemple, dans le test suivant, l'affectation x ← x + 1 n'est effectuée que si la variable de nom x possède pour valeur un entier impair.

```
if x % 2 == 1 :
    x = x + 1
```

Indentation signifiante

Le test simple est en fait l'algorithme suivant :

Si *condition* **alors** *instructions_1* **Fin si**

Dans certains langages de programmation, le "**alors**" est exprimée avec le terme "then" mais, surtout, le "**Fin si**", qui représente la fin du bloc d'instructions à effectuer si la *condition* est vérifiée, est exprimée par un "end" ou "end if", voire, par une parenthèse fermante, parenthèse ouverte par le "**alors**"

Sur Python, le "**alors**" est exprimé par les deux points "**:**" et la parenthèse par l'indentation (retrait par rapport à la ligne)

Le niveau d'indentation correspond à la distinction de différents blocs d'instructions.

Par exemple, les trois codes suivants ont une interprétation différente :

Cas 1	Cas 2	Cas 3

if x % 2 == 1 **:**
  x = x + 1
x = x * 3
	if x % 2 == 1 **:**
	  x = x + 1
	  x = x * 3
		if x % 2 == 1 **:**
		  x = x + 1
		  x = x * 3

Dans le cas 1 : l'instruction x = x * 3 est effectuée, que la condition "x % 2 == 1" soit vérifiée ou non.
Dans le cas 2 : l'instruction x = x * 3 n'est effectuée que si la condition "x % 2 == 1" est vérifiée.
Dans le cas 3 : Python renvoie un message d'erreur à cause de l'indentation.

On peut imbriquer un "**if**" dans un autre "**if**". Dans ce cas, le "**if**" secondaire doit apparaitre avec un niveau d'indentation simple, par contre le bloc d'instructions correspondant à ce "**if**" secondaire doit être écrit avec un niveau d'indentation double.
Par exemple, dans le code suivant :

```
if x % 2 == 1 :
    x = x + 1
    if  y % 2 == 0 :
        y = 1 + y
    x = x + y
```

L'instruction " y = 1 + y" n'est effectuée que si les 2 conditions "x%2 == 1" et "y%2== 0" sont vérifiées, alors que l'instruction "x = x + y" est exécutée dès que "x % 2 == 1" est vraie

Test avec alternative

L'**instruction conditionnelle** n'est exécutée que si une condition donnée est vérifiée par l'état courant ; si ce n'est pas le cas, on peut demander qu'un autre bloc d'instructions soit exécuté.

Pour écrire cette condition, on utilise l'instruction **if … else** qui utilise en Python la syntaxe suivante :

if *condition* **:**
    bloc_1
else :
    bloc_2

Par exemple, dans le test suivant, l'affectation x ← x + 1 n'est effectuée que si la variable de nom x possède pour valeur un entier impair, et si cet entier est pair, c'est l'instruction x = x//2 qui est effectuée.

```
if x % 2 == 1 :
    x = x + 1
else  :
    x = x // 2
```

Tests imbriqués

La structure précédente peut encore être complétée. En effet si la première condition n'est pas vérifiée, on peut vouloir effectuer un second bloc d'instructions que si une seconde condition est alors vérifiée. On peut aussi avoir d'autres conditions entrainant d'autres blocs d'instructions.

Pour décrire cette situation, on utilise l'instruction **if … elif .. else** qui utilise en Python la syntaxe suivante :

if *condition_1* **:**
    bloc_1
elif *condition_2* **:**
    bloc_2
else :
    bloc_3

Par exemple, le processus d'attribution d'une mention au bac peut-être écrite sous la forme :

```
if note >= 16 :
    mention = "TB"
elif note >= 14 :
    mention = "B"
elif note >= 12 :
    mention = "AB"
else  :
    mention = "P"
```

III) Instructions itératives

Boucles for

Lorsque l'on veut répéter une même instruction un certain nombre de fois, on dispose d'un processus permettant d'effectuer cette répétition. Il s'agit de la structure algorithmique nommée boucle **for**.

Elle consiste à écrire que pour la variable k décrivant un ensemble fini donné, une même instruction sera exécutée. Dans beaucoup de cas, la variable k sera prise dans l'intervalle entier [n,m] et les langages de programmation utilise une syntaxe du style : "pour k allant de n jusqu'à m faire"

Dans Python, l'intervalle entier [n,m] est disponible par l'instruction **range**(n, m+1) et **range**(n) fournit l'intervalle entier [0,n-1], donc un ensemble de n termes.

La boucle **for** s'écrit alors : (ici *E* est une variable de type iterable : liste, tableau, tuple)

```
for k in E :
    bloc_instructions
```

Par exemple, pour calculer 2^n avec n un entier dans l'état courant des variables, on peut utiliser le programme suivant :

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

On peut également utiliser cette boucle **for** pour traiter tous les éléments d'un iterable. Par exemple, on peut calculer la somme des éléments d'une liste donnée *E* par le bloc d'instructions suivant :

```
somme = 0
for x in E:
    somme = somme + x
somme
```

Remarque : Python possède des symboles d'affectation abrégée :

x += a signifiera que l'on affecte à x la somme de x et de a

x *= a signifiera que l'on affecte à x le produit de x et de a

Interruption de boucles

Lorsque l'on effectue une boucle pour laquelle on sait que dans de nombreux cas la réponse sera trouvée sans avoir parcouru tout l'iterable, on peut utiliser une instruction d'interruption de boucle : l'instruction **break**.

Par exemple, si on veut tester la primalité d'un entier n, et que l'on teste la divisibilité de n par les entiers compris entre 2 et $\sqrt{n}$, si l'on trouve un diviseur k de n, il n'est pas utile de tester la divisibilité de n par tous les autres entiers compris entre k et $\sqrt{n}$.

```
premier = True
for k in range(2, int(n**0.5) + 1):
    if n % k == 0:
        premier = False
        break
```

On utilisera avec parcimonie ces interruptions de boucles : cela peut nuire à la compréhension des différents cas de sortie de boucles

Boucles imbriquées

On peut insérer une boucle à l'intérieur d'une autre…..

Comme pour les tests imbriqués, c'est le niveau d'indentation qui détermine dans quelle boucle on se trouve.

Par exemple les séquences d'instructions suivantes ne fournissent pas le même résultat :

Cas 1	Cas 2

x = 0
for k **in range**(5):
  for j **in range**(5):
  x += j
  x += k
	x = 0
	for k **in range**(5):
	  for j **in range**(5):
	  x += j
	  x += k

Le résultat trouvé pour x est 100	Le résultat trouvé pour x est 60

IV) Boucles conditionnelles

Boucles while

On dispose d'une autre instruction pour effectuer une instruction itérative très utile lorsque l'on ne connait pas par avance le nombre d'itérations nécessaires mais que l'on connait une condition de sortie. Il s'agit de la structure algorithmique nommée boucle **while**. Elle consiste à écrire qu'une même instruction sera exécutée tant qu'une condition est vérifiée.

Dans Python, la boucle **while** s'écrit :

while *condition* **:**

bloc_instructions

Par exemple, si on veut calculer la plus petite puissance de 2 strictement supérieure à un nombre n, on peut utiliser le programme suivant :

```
p = 1
while p < n :
    p = 2 * p
```

Comme pour les tests **if** et les boucles **for**, on fera attention aux niveaux d'indentation

On peut en fait transformer une boucle **for** en une boucle **while** en introduisant un compteur.

Par exemple, pour calculer 2^n avec n un entier dans l'état courant des variables, on peut utiliser le programme suivant :

```
p = 1
c = n
while c > 0 :
    p = 2 * p
    c = c – 1
```

Danger

Ces boucles **while** sont un moyen assez efficace pour obtenir un résultat sans connaître par avance le nombre d'itérations nécessaire à son obtention. Cependant cet aspect pratique à un revers important constitué par le risque de boucle infinie. En effet, si la condition suivant le **while** est toujours vérifiée, le processus ne termine jamais. Il conviendra donc de bien s'assurer que les changements de valeurs des variables de l'état permettront bien de "sortir" de la boucle car la condition testée ne sera plus vérifiée.

Par ailleurs, il convient également de vérifier que le résultat donné correspond bien à la valeur recherchée.

Un moyen de s'assurer de cela est de déceler dans la boucle un "invariant de boucle".

Par exemple dans le dernier programme écrit, on peut constater, d'une part que la boucle va bien s'arrêter car c finira bien par être négatif … (pour peu que l'initialisation "c = n" soit effectuée avec n un nombre que l'on peut comparer à 0) et, d'autre part que la valeur p * 2^c est constante après chaque itération, et que cette constante vaut 2^n. Il ne reste plus qu'à constater que la boucle va s'arrêter lorsque que l'on trouvera c <= 0 et en fait c = 0 si on part avec n entier.

V) Exercices

1) Ecrire un programme qui détermine le minimum des trois valeurs a, b et c.

2) Ecrire un programme permettant de calculer le 100-ième terme (u_{100}) de la suite récurrente définie par : $u_0 = 1$ et $\forall n \in \mathbb{N},\ u_{n+1} = \frac{1}{9}(u_n - 3)^2$.

3) Calculer le 100-ième terme (u_{100}) de la suite récurrente définie par : $u_0 = 1$ et $\forall n \in \mathbb{N},\ u_{n+1} = 4 - n\,u_n$

4) Calculer le 100-ième terme de la suite récurrente définie par : $u_0 = 1$, $u_1 = 1$ et $\forall n \in \mathbb{N},\ u_{n+2} = u_{n+1} + u_n$

5) Déterminer le plus petit entier n tel que la somme des entiers de 1 à n dépasse 1000

6) On désire connaitre le plus grand entier n vérifiant : $n! < 10^{30}$

Commenter et éventuellement corriger les propositions suivantes (on suppose chargé le module math) :

a)
```
n = 1
while factorial(n+1) < 10**30 :
        n = n + 1
print("n = ",n)
```

b)
```
p = 1
n = 2
while  p < 10**30 :
        p = p * n
        n = n + 1
print("n = ",n)
```

c)
```
n = 1
p = 1
while p < 10**30 :
        n = n + 1
        p = p * n
print("n = ", n)
```

d)
```
p = 1
for n in range (1,10**10) :
        p = p * n
        if p >= 10**30:
                break
print (" n = ",n)
```