

Programmation en Python - Cours 2 : Premiers programmes

MPSI - Lycée Thiers

2014/2015

Diverses utilisations de python

Utilisation en mode interactif

Ecriture d'un programme python

Programmation en python

Saisie de données par l'utilisateur : `input()`

Un nouveau type de variables : booléen

Structures de contrôle

Structure de boucle `while`

Structure de test

python en mode interactif

Nous avons vu comment python peut s'utiliser en mode interactif en le lançant dans une console par la commande `python`.

C'est pratique pour utiliser python comme une calculatrice ou pour programmer des algorithmes simples. Mais :

- L'environnement de calcul est pauvre. Un environnement plus riche est obtenu en lançant `ipython`.
- Le mode interactif ne permet pas de développer des programmes complexes : à chaque utilisation il faut réécrire le programme. La modification d'une ligne oblige à réécrire toutes les lignes qui la suivent.

Pour développer un programme plus complexe on saisit son code dans un fichier texte : plus besoin de tout retaper pour modifier une ligne ; plus besoin de tout réécrire à chaque lancement.

On utilisera plutôt un environnement de développement qui permettra de saisir le programme, de l'exécuter, déboguer avec une meilleure ergonomie.

ipython

La commande `ipython` permet de lancer un environnement interactif plus riche.

```
(py3k)macbook-pro-de-jean-philippe:~ JPh$ ipython
Python 3.3.5 |Anaconda 2.0.1 (x86_64)| (default, Mar 10 2014, 11:22:25)
Type "copyright", "credits" or "license" for more information.
```

```
IPython 2.1.0 -- An enhanced Interactive Python.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
?          -> Introduction and overview of IPython's features.
%quickref  -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.
```

```
In [1]: █
```

```
In [1]: a = 15
```

```
In [2]: print(a)
15
```

```
In [3]: chaine = "Bonjour l'univers"█
```

La définition d'une variable ne produit plus son affichage; l'instruction `print` y trouve toute son utilité.

ipython

```
In [4]: %whos
```

Variable	Type	Data/Info
a	int	15
chaine	str	Bonjour l'univers

La méta-commande %whos produit l'affichage de l'état des variables.

```
In [11]: chaine = "Bonjour"
```

```
In [12]: type(chaine)
```

```
Out[12]: str
```

ipython

```
In [5]: help()
```

Welcome to Python 3.3! This is the interactive help utility.

If this is your first time using Python, you should definitely check out the tutorial on the Internet at <http://docs.python.org/3.3/tutorial/>.

Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules", "keywords", or "topics". Each module also comes with a one-line summary of what it does; to list the modules whose summaries contain a given word such as "spam", type "modules spam".

`help()` lance l'aide interactive.

ipython

```
help> keywords
```

Here is a list of the Python keywords. Enter any keyword to get more help.

False	def	if	raise
None	del	import	return
True	elif	in	try
and	else	is	while
as	except	lambda	with
assert	finally	nonlocal	yield
break	for	not	
class	from	or	
continue	global	pass	

keywords retourne les mots-clefs réservés (à ne pas utiliser pour la définition d'une variable ou fonction).

Ce sont les noms des trois constantes False, True, None et de toutes les commandes (basiques) du langage python permettant d'écrire des instructions .

ipython

```
The ``while`` statement
*****
```

The ``while`` statement is used for repeated execution as long as an expression is true:

```
while_stmt ::= "while" expression ":" suite
             ["else" ":" suite]
```

This repeatedly tests the expression and, if it is true, executes the first suite; if the expression is false (which may be the first time it is tested) the suite of the ``else`` clause, if present, is executed and the loop terminates.

A ``break`` statement executed in the first suite terminates the loop without executing the ``else`` clause's suite. A ``continue`` statement executed in the first suite skips the rest of the suite and goes back to testing the expression.

(END)

On obtient une description d'une commande en saisissant son nom : `help> while` produit ce résultat (`ctrl` + `z` pour fermer). Il faut comprendre l'anglais...

Voir aussi : www.python.org/doc/

ipython

```
In [7]: %%timeit
...: u,v = 0,1
...: i = 0
...: while i<999:
...:     u,v = u+v,u+2*v
...:     i += 1
...:
1000 loops, best of 3: 347 µs per loop
```

%%timeit permet de mesurer la durée d'exécution d'un petit script.

Le script est saisi dans le bloc qui suit.

Ici le calcul des termes de rangs 999 et 1000 de la suite de Fibonacci a pris 347 μ s.

(où 1 μ s = 10^{-6} seconde = 1 microseconde = 1 millionième de seconde.)

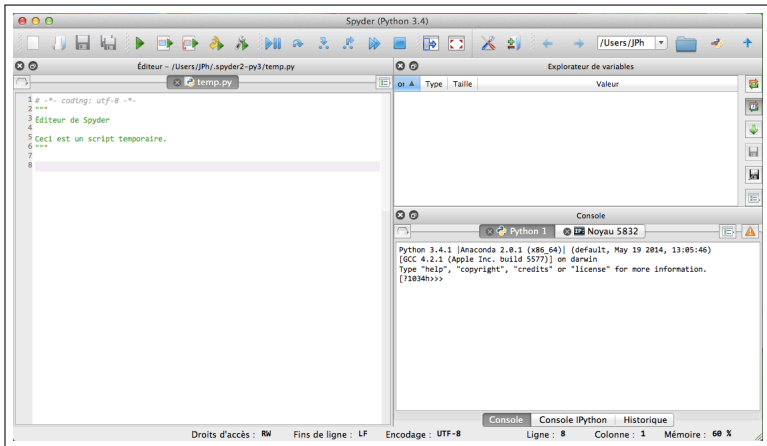
Ecrire un programme python

Le programme s'écrit dans un fichier texte que l'on sauvegarde avec l'extension `.py`. Le lancement du programme peut se faire à partir d'une console par la commande `python fichier.py`.

Il est préférable d'utiliser un environnement de développement ; il permet d'écrire des programmes, de les sauvegarder, modifier, etc..., de lancer leur exécution, de la stopper, et d'avoir une fenêtre interactive qui aide au développement du programme et retourne les résultats de l'exécution.

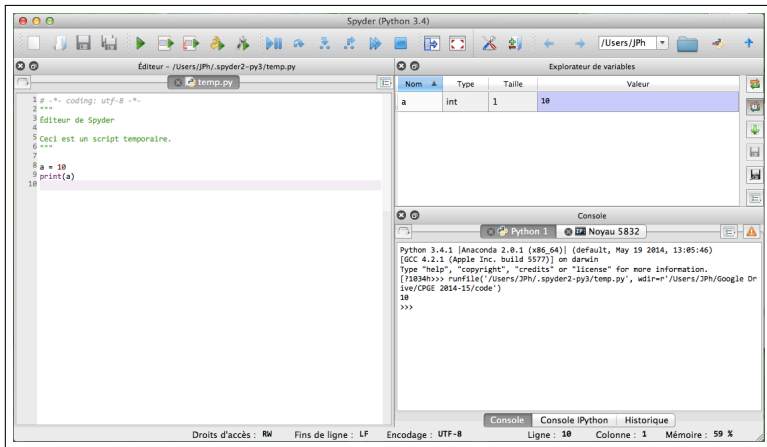
Environnement de programmation

Par exemple anaconda, à télécharger au lien suivant <http://continuum.io/downloads> avec la version 3.4 du langage :



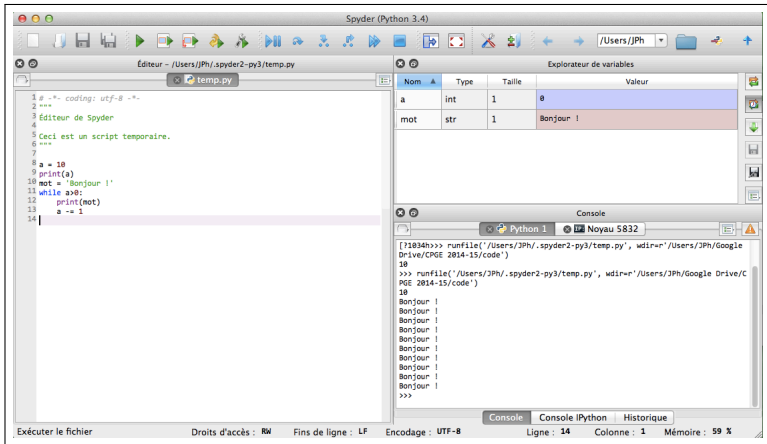
Environnement de programmation

Premier programme : saisie du programme dans l'éditeur (à gauche). Le résultat apparaît (en bas à droite) après avoir lancé l'exécution (icône : ▶). En haut à gauche (optionnel), l'état des variables.



Environnement de programmation

On peut modifier le programme aisément, en rajoutant des commentaires, des espaces, et de nouvelles lignes d'instruction).



Environnement de programmation

Saisie du programme dans l'éditeur (à gauche).

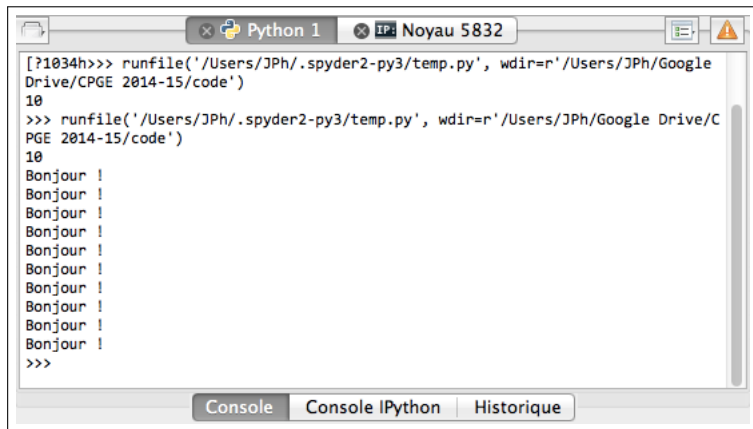


The screenshot shows a window titled 'Éditeur - /Users/JPh/.spyder2-py3/temp.py'. The code inside is as follows:

```
1 # -*- coding: utf-8 -*-
2 """
3 Éditeur de Spyder
4
5 Ceci est un script temporaire.
6 """
7
8 a = 10
9 print(a)
10 mot = 'Bonjour !'
11 while a>0:
12     print(mot)
13     a -= 1
14
```

Environnement de programmation

La fenêtre 'console' en bas à droite fonctionne en mode interactif (python ou python).



```
[?1034h>>> runfile('/Users/JPh/.spyder2-py3/temp.py', wdir=r'/Users/JPh/Google Drive/CPGE 2014-15/code')
10
>>> runfile('/Users/JPh/.spyder2-py3/temp.py', wdir=r'/Users/JPh/Google Drive/CPGE 2014-15/code')
10
Bonjour !
Bonjour !
Bonjour !
Bonjour !
Bonjour !
Bonjour !
Bonjour !
Bonjour !
Bonjour !
Bonjour !
>>>
```

L'icône en haut à droite permet d'interrompre l'exécution d'un programme (qui bogue); celle à sa gauche de réinitialiser la console.

Environnement de programmation

La fenêtre 'explorateur de variable' (en haut à droite) donne l'état des variables après exécution.

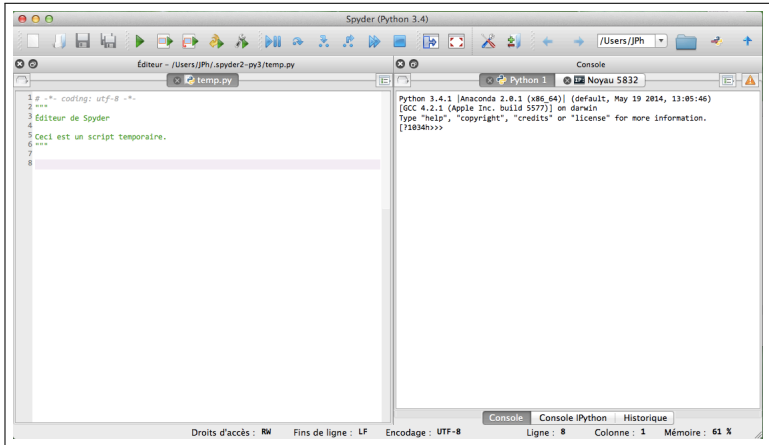


Nom ▲	Type	Taille	Valeur
a	int	1	0
mot	str	1	Bonjour !

Elle est surtout utile pour le débogage.

Environnement de programmation

On peut aussi la fermer pour ne garder que les fenêtres essentielles : l'éditeur et une console.



La fonction input()

Elle attend la saisie par l'utilisateur d'une chaîne de caractère, et retourne la valeur saisie. La chaîne de caractère doit être saisie sans apostrophes '.' ou guillemets '"

```
>>> # La fonction input()
... chaine = input(); print("Vous avez saisi :",chaine)
Ceci est ma réponse
Vous avez saisi : Ceci est ma réponse
```

On peut passer à la fonction input() en argument une chaîne de caractère qui sera inscrite à l'écran.

```
>>> str = input('Saisissez votre réponse : '); print('Vous avez saisi :',str)
Saisissez votre réponse : Voici ma réponse !
Vous avez saisi : Voici ma réponse !
```

Si l'on n'a besoin d'une valeur numérique on convertit le résultat retourné à l'aide d'une fonction de conversion de type : int() ou float.

```
>>> n = int(input('Saisissez un nombre entier ')); print('Son carré est ',n**2)
Saisissez un nombre entier 3
Son carré est 9
```

La fonction input()

Elle attend la saisie par l'utilisateur d'une chaîne de caractère, et retourne la valeur saisie. La chaîne de caractère doit être saisie sans apostrophes '.' ou guillemets '"'.

```
>>> # La fonction input()
... chaine = input(); print("Vous avez saisi :",chaine)
Ceci est ma réponse
Vous avez saisi : Ceci est ma réponse
```

On peut passer à la fonction input() en argument une chaîne de caractère qui sera inscrite à l'écran.

```
>>> str = input('Saisissez votre réponse : '); print('Vous avez saisi :',str)
Saisissez votre réponse : Voici ma réponse !
Vous avez saisi : Voici ma réponse !
```

Si l'on n'a besoin d'une valeur numérique on convertit le résultat retourné à l'aide d'une fonction de conversion de type : int() ou float.

```
>>> n = input('Saisissez un nombre entier '); print('Son carré est ',n**2)
Saisissez un nombre entier 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Autrement un calcul produira erreur ou résultat inattendu.

La fonction input()

Mon premier programme devient plus interactif :

```
>>> # Mon deuxième programme :  
... r = input('Comment vous appelez-vous ? '); print('Bonjour',r)  
Comment vous appelez-vous ? Toto  
Bonjour Toto
```

L'utilisateur peut interagir avec le programme durant son exécution.

Type booléen : bool

Lorsque l'on saisit les conditions suivantes les valeurs retournées peuvent prendre 2 valeurs : True ou False. Elles sont de type booléen 'bool' :

```
>>> 1 < 2
True
>>> 1 > 2
False
```

```
>>> a = (1<2)
>>> type(a)
<class 'bool'>
```

Opérateurs de comparaison :

a < b	a a une valeur strictement inférieure à celle de b
a <= b	a a une valeur inférieure ou égale à celle de b
a == b	a et b ont même valeur.
a != b	a et b ont des valeurs différentes.

et pareillement a > b et a >= b.

Opérations sur les booléens

On peut effectuer des opérations logiques sur les booléens (par ordre de priorité) : or, and, not() :

```
>>> a = (1 <= 2); b = (a == False); print(a,b)
True False
>>> a or b ; a and b ; not(a) ; not(b)
True
False
False
True
```

```
>>> var = "chaîne"; var == "Chaîne"; var != "Chaîne"; not(var == "Chaîne")
False
True
True
>>> (1 == 1.00) and ("chapeau" == 'chapeau')
True
```

La fonction suivante définit le connecteur logique xor (ou exclusif) :

```
>>> # Connecteurs logique xor : a xor b= a and not(b) or not(a) and b
... def xor(a,b):
...     return (a and not(b)) or (not(a) and b)
...
>>> xor(True,False), xor(False,True), xor(True,True), xor(False,False)
(True, True, False, False)
```

Conversion en booléen : `bool()`

La fonction de conversion `bool()` convertit une valeur de n'importe quel type en un booléen, selon les règles suivantes :

- Un nombre `'int'` ou `'float'` est converti à `True` s'il est non nul, à `False` sinon.
- Une chaîne de caractère `'str'` est convertie à `True` si elle est non vide, à `False` si c'est la chaîne vide : `""`.

```
>>> bool(3.14)
True
>>> bool(-7)
True
>>> bool('toto')
True
>>> bool('')
False
```

Structures de contrôle

Pour l'instant nous avons vu que python est un langage de programmation impérative (un programme est une suite de ligne d'instructions), nous allons voir que c'est un langage de programmation structuré. Il reconnaît les structures de contrôles :

- une structure de test : `if ... [elif] ... [else]`
- deux structures de boucles : `while ...`
`for ...`

sans lesquelles les programmes s'exécuteraient séquentiellement ligne après ligne.

Nous allons commencer par voir la structure de boucle `while`

puis la structure de test `if (elif) (else)`.

Nous étudierons plus tard la deuxième structure de boucle `for`.

`while` et `if` nous suffisent pour programmer. Tandis que `for` et `if` ne suffisent pas. Cependant `for` est particulièrement pratique, surtout en python !

L'instruction while

L'instruction while permet de répéter une séquence d'instruction, en boucle, tant qu'une *condition* est vérifiée.

```
while condition:
    .....   Instruction1
    .....   Instruction2
    .....   :
    .....   Dernière instruction
```

 └──────────┘
 bloc d'instructions

└──────────┘
meme espace

En général *condition* s'évalue en un booléen, par exemple :
True, $a \leq 0$, $(a == 0) \text{ and } (b \neq -1)$.

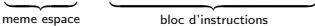
```
>>> x = 1e10      # Notation scientifique 1*10^10
>>> x
10000000000.0
>>> n = 0         # Initialisation de n : nombre de passages dans la boucle
>>> while (2**n < x):
...     n = n + 1
...
>>> print(n-1, '; ', 2**(n-1), '<=', x, '<=', 2**n)
33 ; 8589934592 < 10000000000.0 <= 17179869184
```

Ainsi $33 < \log_2(10^{10}) < 34$.

L'instruction while

L'instruction `while` permet de répéter une séquence d'instruction, en boucle, tant qu'une *condition* est vérifiée.

```
while condition:
    ..... Instruction1
    ..... Instruction2
    ..... :
    ..... Dernière instruction
```



Mais n'importe quelle valeur, de type quelconque peut être utilisée : elle est convertie en booléen.

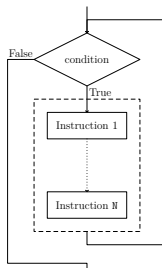
Exemple : pour attendre la saisie d'une chaîne non vide :

```
>>> # Pour attendre la saisie d'une chaîne non-vide
... var = ''
>>> while not(var):
...     var = input('Saisissez une chaîne non-vide ')
...
Saisissez une chaîne non-vide
Saisissez une chaîne non-vide
Saisissez une chaîne non-vide ok
>>> █
```

L'instruction while

L'instruction `while` permet de répéter une séquence d'instruction, en boucle, tant qu'une *condition* est vérifiée.

Organigramme dune boucle `while` :



On l'écrit en langage algorithmique :

Tant que condition faire :
Instruction 1
⋮
Instruction N

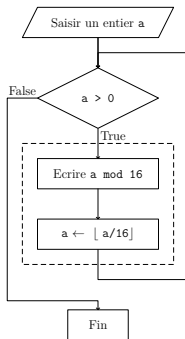
Exemple : Ecrire un nombre en base 16. (1^{ère} ébauche.)

Code python :

```
a = int(input('Saisissez un entier'))  
while a > 0 :  
    print a % 16  
    a = a // 16
```

En langage algorithmique :

Saisir un entier a
Tant que a > 0 faire :
 écrire a modulo 16
 $a \leftarrow \lfloor a / 16 \rfloor$



Pour comprendre son fonctionnement, modifions le code et regardons son exécution :

```
passage = 0
a = int(input('Saisissez un entier'))
while a > 0 :
    passage = passage + 1
    print('Passage ', passage)
    print('a modulo 16 : ', a % 16)
    a = a // 16
    print('a devient a//16 : ', a)
print('Sortie de la boucle')
```

```
Saisissez un entier : 72
Passage 1
a modulo 16 : 8
a devient a//16 : 4
Passage 2
a modulo 16 : 4
a devient a//16 : 0
Sortie de la boucle
```

Lorsque not($a > 0$) la boucle while s'arrête.

Amélioration du code (2^{ème} ébauche) :

```
a = int(input('Saisissez un entier'))
resultat = ''
while a > 0:
    resultat = str(a % 16) + ' ' + resultat
    a = a // 16
print(resultat)
```

L'exécution produit :

```
Saisissez un nombre 72
4 8
```

```
Saisissez un nombre 16 ** 3 + 15 * 16 ** 2 + 7
1 15 0 7
```

str() retourne la valeur de son paramètre convertie en chaîne de caractère.
Pour les chaînes de caractère '+' est l'opération de *concaténation* :
'aaaa' + 'bbbb' s'évalue en la chaîne 'aaaabbbb'.

Boucle sans fin

Attention une boucle peut ne jamais se terminer, et provoquer une exécution sans fin :

```
while True :    print('bonjour')
```

provoque une affichage sans fin...

(remarquer que dans le cas d'instructions courtes la boucle peut s'écrire sur une seule ligne).

Cela peut être volontaire pour lancer une procédure indéfiniment : la procédure doit permettre l'arrêt :

```
while True :    menu()
```

Ici menu() est une procédure (à écrire) qui proposera plusieurs choix à l'utilisateur. Il doit pouvoir choisir l'arrêt de l'exécution du programme.

Nous verrons plus tard qu'il est important de justifier de l'arrêt d'une boucle (à l'aide d'un invariant de boucle).

structure de test

```
if condition :
```

```
.....
```

Bloc d'Instructions

```
else :
```

```
.....
```

Bloc d'instructions

⏟
meme espace

⏟
bloc d'instructions

La condition est évaluée en booléen.

- Si elle a valeur `True` le premier bloc d'instruction est exécuté, le deuxième ne l'est pas, puis l'exécution du programme se poursuit.
- Si elle a valeur `False` le deuxième bloc d'instruction est exécuté (le premier ne l'est pas), puis l'exécution du programme se poursuit.

L'instruction `else` est optionnelle.

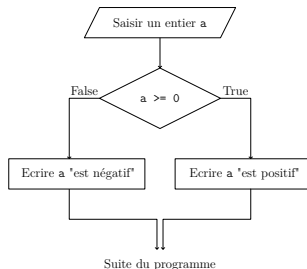
Structure de test if [else]

Code python :

```
a = float(input('Saisissez un nombre'))  
if a >= 0 :  
    print(a, 'est positif')  
else :  
    print(a, 'est négatif')
```

En langage algorithmique :

```
Saisir un nombre a  
Si a >= 0 faire :  
    écrire a, "est positif"  
Sinon faire :  
    écrire a, "est négatif"
```



structure de test if [else]

- Exemple 1 : Une fonction pour déterminer si un nombre entier est pair ou impair :

```
def pair(n):  
    if n%2 == 0:  
        return True  
    else:  
        return False
```

- Exemple 2 : Dans le calendrier grégorien (actuel), une année est bissextile si soit :
 - Elle est divisible par 4 sans être divisible par 100.
 - Elle est divisible par 400.

Ecrire une fonction qui décide si une année est ou non bissextile.

```
def bissextile(n):  
    if n%400 == 0:  
        return True  
    if n%4 == 0:  
        if n%100 == 0:  
            return False  
        else:  
            return True  
    else:  
        return False
```

Remarque : une commande return provoque la sortie de la fonction.

structure de test if [else]

- Année bissextile :

Si dans le dernier programme on remplace les commandes return par print() (ce n'est plus une fonction mais une procédure), le programme bogue :

```
def bissextile(n):  
    if n%400 == 0:  
        print(n,'est une année bissextile')  
    if n%4 == 0:  
        if n%100 == 0:  
            print(n,"n'est pas une année bissextile")  
        else:  
            print(n,'est une année bissextile')  
    else:  
        print(n,"n'est pas une année bissextile")
```

```
>>> bissextile(2000)  
2000 est une année bissextile  
2000 n'est pas une année bissextile
```

En effet la condition `n%400 == 0` est vérifiée et produit l'affichage de la première ligne, mais les conditions suivantes `n%4 == 0` et `n%100 == 0` sont aussi vérifiées et produisent l'affichage de la seconde ligne.

structure de test if [else]

La solution consiste à imbriquer ces deux tests en conditionnant le second à l'échec du premier, de la façon suivante :

```
def bissextile(n):  
    if n%400 == 0:  
        print(n,'est une année bissextile')  
    else:  
        if n%4 == 0:  
            if n%100 == 0:  
                print(n,"n'est pas une année bissextile")  
            else:  
                print(n,'est une année bissextile')  
        else:  
            print(n,"n'est pas une année bissextile")
```

Le programme ne bogue plus :

```
>>> bissextile(2000)  
2000 est une année bissextile
```

Structure de test if [elif] [else]

L'écriture de tests imbriqués :

```
if (condition1) :  
    Bloc d'instructions 1  
else :  
    if (condition2) :  
        Bloc d'instructions 2  
    else :  
        Bloc d'instructions 3
```

s'écrit à l'aide d'une seule structure de test :

```
if (condition1) :  
    Bloc d'instructions 1  
elif (condition2) :  
    Bloc d'instructions 2  
else :  
    Bloc d'instructions 3
```

elif et else sont optionnels. elif peut être utilisé plusieurs fois dans une structure de test : if ... elif ... elif ... else.

structure de test if [elif] [else]

Reprenons l'exemple de la fonction bissextile. En voici une version plus concise et lisible :

```
def bissextile(n):  
    if n%400 == 0:  
        print(n,'est une année bissextile')  
    elif n%4 == 0 and n%100 != 0:  
        print(n,'est une année bissextile')  
    else:  
        print(n,"n'est pas une année bissextile")
```

Le programme ne bogue plus :

```
>>> bissextile(2000)  
2000 est une année bissextile
```