

Cours de Python en Licence

- Ce cours est basé sur le MOOC de Charles Severance accessible sur la plateforme Coursera : <https://www.coursera.org/course/pythonlearn>
- Pour compléter ce cours, nous vous conseillons de visionner les vidéos de Charles Severance accessibles via l'Espace Numérique de Travail
- Il est également dans votre intérêt de refaire à la maison les exercices des Travaux Dirigés, dans un but d'apprentissage et d'auto-évaluation

Cours de Python en Licence

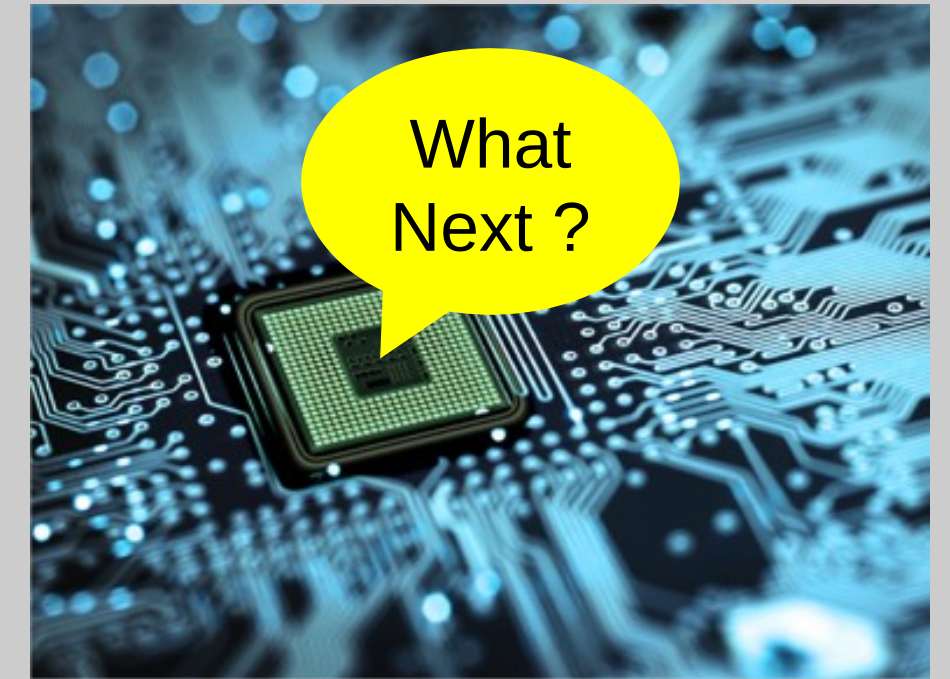
- Séquence 1 : Pourquoi programmer ? Notions d'architecture matérielle
- Séquence 2 : Constantes, variables, types, expressions, entrées-sorties
- Séquence 3 : Structures conditionnelles, gestion des exceptions
- Séquence 4 : Fonctions prédéfinies et définies par l'utilisateur
- Séquence 5 : Structures itératives, les types liste et chaîne

Chapitre 1

Pourquoi programmer ?

Les ordinateurs « veulent » nous être utiles...

- Les ordinateurs sont construits dans un seul objectif : **faire des choses pour nous...**
- Mais nous avons besoin de « parler leur langue » pour qu'ils puissent faire ce que nous voulons...
- Mais notre tâche a été facilitée : des spécialistes ont intégré beaucoup de **programmes** différents dans l'ordinateur
- et les utilisateurs n'ont qu'à choisir les programmes qu'ils veulent exécuter



What
Next ?

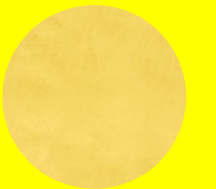
What
Next ?

What
Next ?

What
Next ?

What
Next ?

What
Next ?



Les programmeurs anticipent les besoins

- Exemple : les applications pour smartphone est un marché :
- De 2008 à fin 2012, il y a eu 40 milliards d'applications téléchargées sur la boutique applicative AppStore !
- Des programmeurs quittent leur poste pour devenir développeurs iPhone à plein temps

Qu'est-ce qu'un programmeur ?



```
line = raw_input()
with open(filename, 'r') as f:
    text = f.read()
words = text.split()

counts = dict()
for word in words:
```



Choisi
Moi !

Choisi
Moi !

Achète
Moi !

Choisi
Moi !

Choisi
Moi !

Choisi
Moi !

Utilisateurs vs. Programmeurs

- Les **utilisateurs** voient les ordinateurs comme un ensemble d'outils : traitement de texte, tableur, agenda, logiciels de communication, etc.
- Les **programmeurs** quant à eux :
 - savent comment fonctionnent les ordinateurs
 - apprennent les méthodes et les langages informatiques
 - ont des outils qui leur permettent de construire de nouveaux outils
 - certains sont destinés aux utilisateurs, d'autres aux programmeurs
- **Pourquoi programmer ?**
 - pour nos propres besoins = programmeur & utilisateur à la fois
 - pour produire un logiciel utilisable par d'autres = programmeur de métier

Qu'est-ce que du code ? Un logiciel ? Un programme?

- C'est une **séquence d'instructions** mémorisées dans l'ordinateur
- C'est une **expression de notre intelligence** que nous donnons aux autres :
 - Nous comprenons quelque chose □ nous le traduisons en langage informatique □ nous en faisons profiter d'autres personnes pour leur faire gagner du temps et de l'énergie
- Si nous produisons une bonne « user experience » un programme peut aussi être considéré comme un objet créatif, voire artistique

Un programme pour humains...



<http://www.youtube.com/watch?v=vlzwwuFkn88U>

Un programme pour humains...

Main gauche vers l'extérieur et en haut
Main droite vers l'extérieur et en haut
Retournez la main gauche
Retournez la main droite
Main gauche sur l'épaule droite
Main droite sur l'épaule gauche
Main gauche derrière la tête
Main droite derrière la tête
Main gauche sur la manche droite
Main droite sur la manche gauche
Main gauche sur fesse gauche
Main droite sur fesse droite
Se déhancher
Se déhancher
Sauter



<http://www.youtube.com/watch?v=vlzwwuFkn88U>

Un programme pour humains...

Main gauche vers l'extérieur et en haut
Main droite vers l'extérieur et en haut
Retournez la main gauche
Retournez la main droite
Main gauche sur l'épaule droite
Main droite sur l'épaule gauche
Main gauche derrière la tête
Nain droite derrière la tête
Main gauche sur la **manche** droite
Main droite sur la **manche** gauche
Main gauche sur fesse gauche
Main droite sur fesse droite
Se déhancher
Se déhancher
Sauter

BUGS ?



<http://www.youtube.com/watch?v=vlzwwuFkn88U>

Un programme pour humains...

Main gauche vers l'extérieur et en haut
Main droite vers l'extérieur et en haut
Retournez la main gauche
Retournez la main droite
Main gauche sur l'épaule droite
Main droite sur l'épaule gauche
Main gauche derrière la tête
Main droite derrière la tête
Main gauche sur la **hanche** droite
Main droite sur la **hanche** gauche
Main gauche sur fesse gauche
Main droite sur fesse droite
Se déhancher
Se déhancher
Sauter

FIXED !



<http://www.youtube.com/watch?v=vlzwwuFkn88U>

Petit exercice :

Quel mot apparaît le plus souvent dans le petit texte ci-dessous ?



le clown couru après la voiture et la voiture fonça sur la tente et la tente tomba sur le clown et la voiture

Combien de temps vous a-t-il fallu pour trouver une réponse ?

Un programme pour ordinateurs

```
filename = input('Nom du fichier: ')
with open(filename, 'r') as file:
    text = file.read()
words = text.split()

counts = dict()
for word in words:
    counts[word] = counts.get(word,0) + 1

bigword, bigcount = None, None
for word, count in counts.items():
    if bigcount == None or count > bigcount:
        bigword, bigcount = word, count

print(bigword, bigcount)
```

Exemple d'exécution :

```
python words.py
Nom du fichier: clown.txt
la 5
```

*Ce programme peut traiter
d'autres programmes,
y compris lui-même :*

```
python words.py
Nom du fichier: words.py
= 9
```

Chapitre 1 (suite)

Architecture matérielle

*« L'informatique n'est pas plus la science des ordinateurs
que l'astronomie n'est celle des télescopes »*

Edsger Dijkstra (1930-2002)

Un peu d'histoire...

- Le métier à tisser de Jacquard (1806) :

Les métiers à tisser de Joseph-Marie Jacquard sont des machines automatisées programmées par cartes perforées qui décrivent le motif qui doit être tissé.

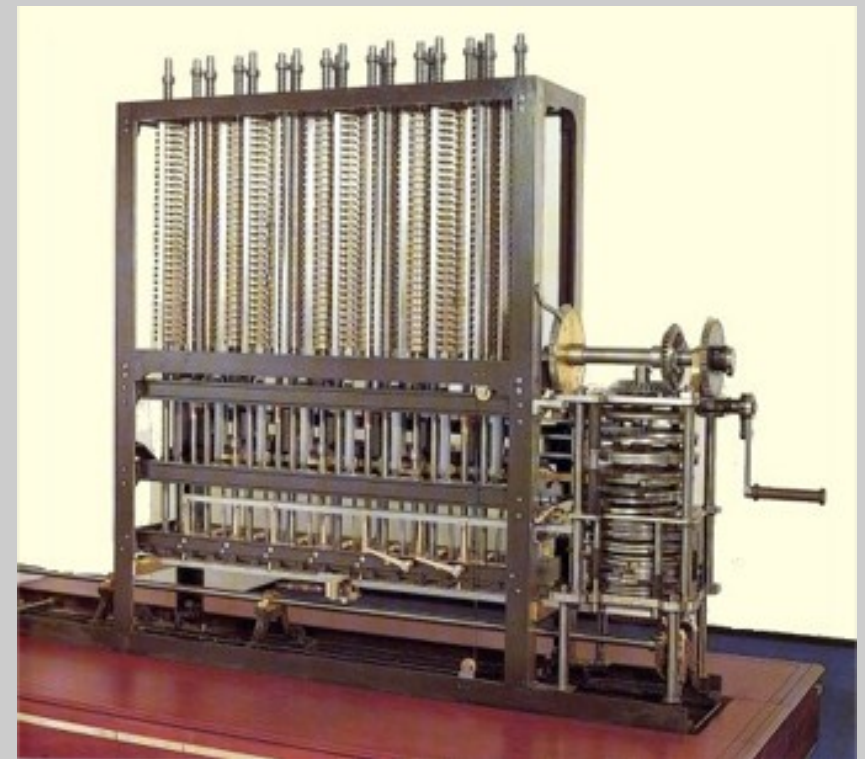
L'utilisation de cartes perforées fait que le métier Jacquard est parfois considéré comme un ancêtre des ordinateurs.



- La machine analytique de Babbage (1834) :

La machine analytique de Charles Babbage est une calculatrice mécanique qui repose sur l'utilisation de cartes perforées et qui fait la distinction entre unité de calcul et unité de mémoire.

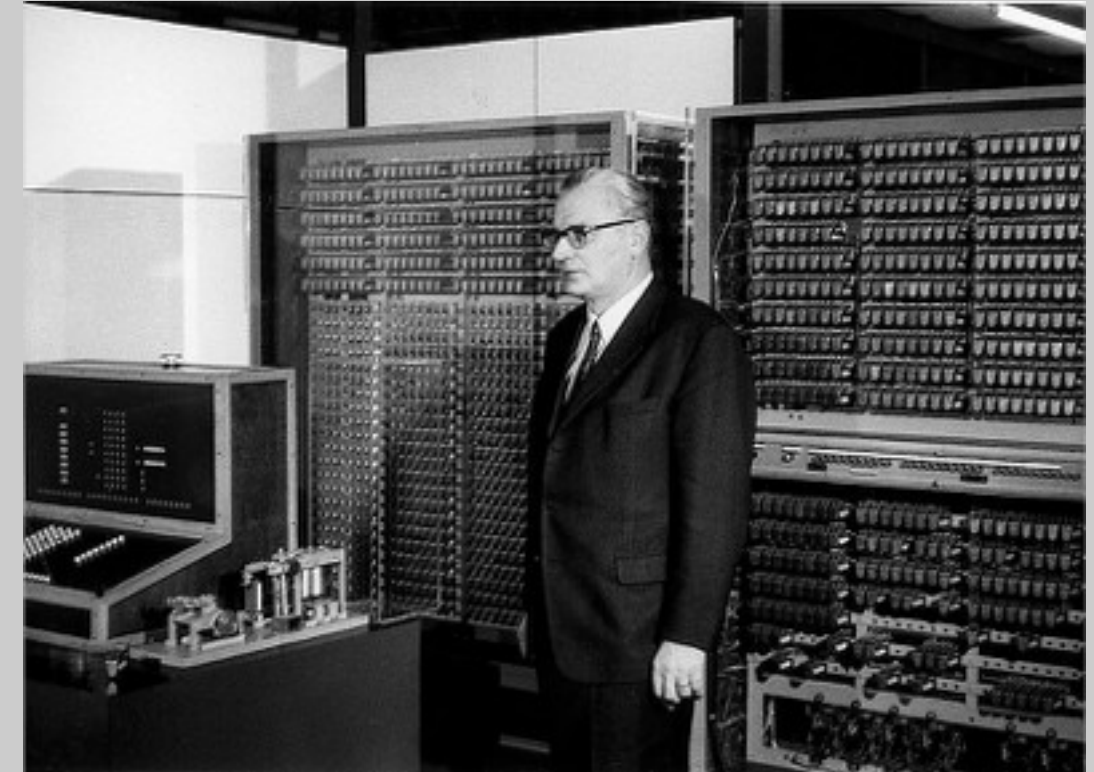
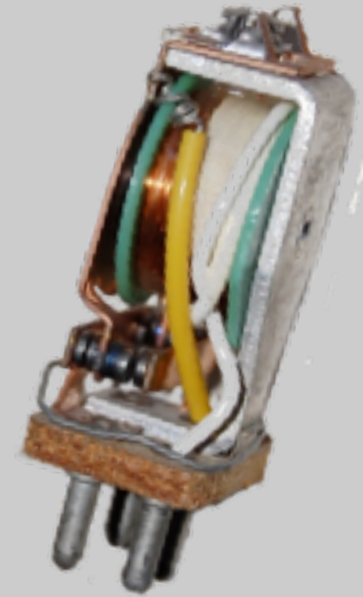
Babbage échoue dans la mise en œuvre de sa machine car les techniques d'usinage de l'époque ne sont pas assez précises.



Un peu d'histoire...

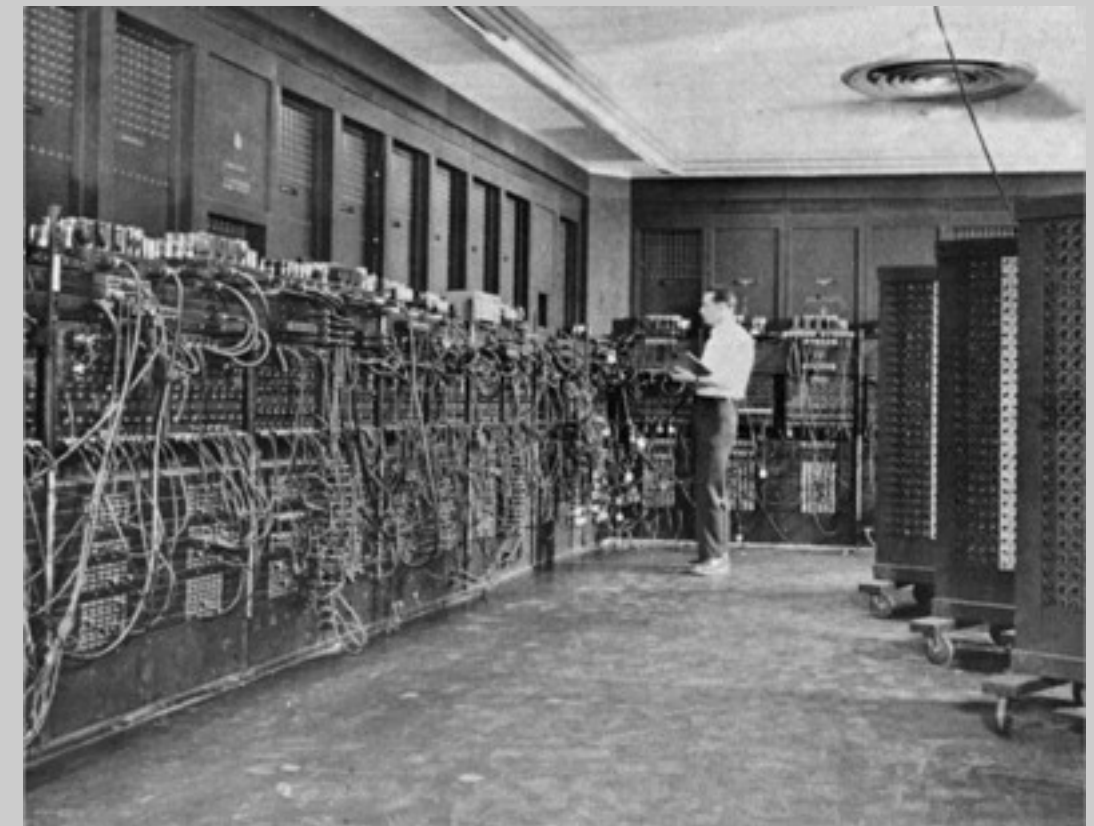
- Calculateurs électromécaniques (1930 ~ 45) :

Exemple du Z3 (1941) : les 2000 relais fonctionnaient à une fréquence d'horloge de 5 à 10 Hz. Le code et les données étaient stockées sur des rubans perforés en celluloïd. Il pouvait effectuer 4 additions par seconde ou une multiplication en 4 secondes.



- Calculateurs électroniques (1945 ~ 55) :

Exemple de l'ENIAC (1956) : 10.000 condensateurs, 17.000 tubes à vide, 70.000 résistances, 5 millions de soudures, 30 tonnes, 150 m². Il pouvait effectuer 5000 additions par seconde et seulement 357 multiplications ou 38 divisions par seconde. Applications militaires (balistique, bombe A).



Un peu d'histoire...

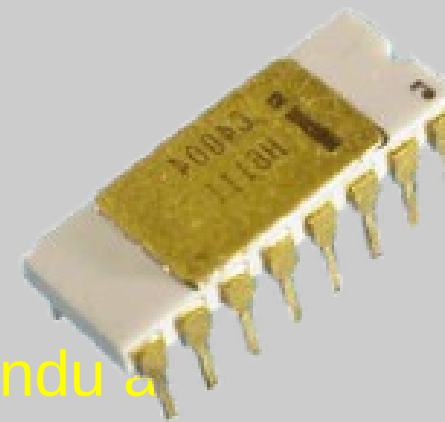
- Ordinateurs à transistors (1955 ~ 70) :

Exemple du PDP-1 (1959) : Doté d'un écran à tube cathodique, c'est le premier ordinateur interactif, assez proche dans son utilisation des micro-ordinateurs vendus 20 ans plus tard. Le premier jeu vidéo (Space War) a été programmé sur cette machine.



- Ordinateurs à microprocesseurs (1970 ~) :

Exemple du MICRAL-80 (1980) : Equipé d'un Zilog Z80 ou d'un Intel 8080 à 3Mhz, doté d'une mémoire de 64 à 256 Ko, de deux lecteurs de disquettes 5.25" et d'un écran monochrome. Destiné à l'éducation nationale, il fut vendu à un prix exorbitant (46.400 Francs).



Un peu d'histoire...

- Un des premiers micro-ordinateurs (1975) :

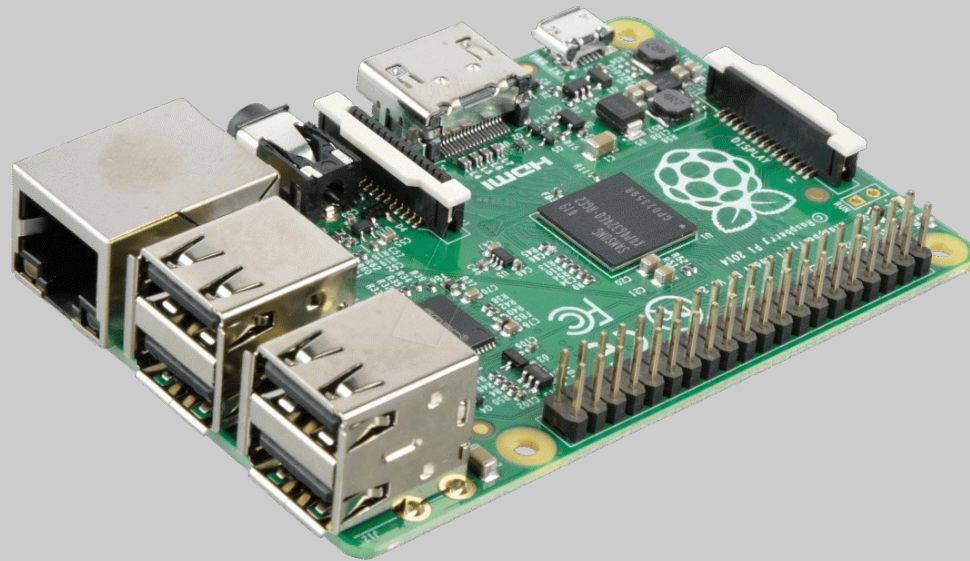
L'Altair 8800 est un micro-ordinateur basé sur le μ P Intel 8080 et vendu en kit à quelque milliers d'exemplaires. Deux pionniers de la micro-informatique, Bill Gates et Paul Allen, découvrent l'Altair dans une revue et proposent à la société MITS de développer le langage "Altair Basic". Paul Allen et Bill Gates fondent alors immédiatement une société au nom de Microsoft, et passent un accord de commercialisation avec MITS pour toucher 35 dollars (~100€ en 2012) par copie de leur logiciel vendu avec chaque Altair... La suite vous la connaissez ?

Un **micro-ordinateur** est un ordinateur conçu autour d'un **micro-processeur** sur une base de **circuit imprimé** comportant des **circuits intégrés** (puces de silicium) => optimisation + miniaturisation = faible coût

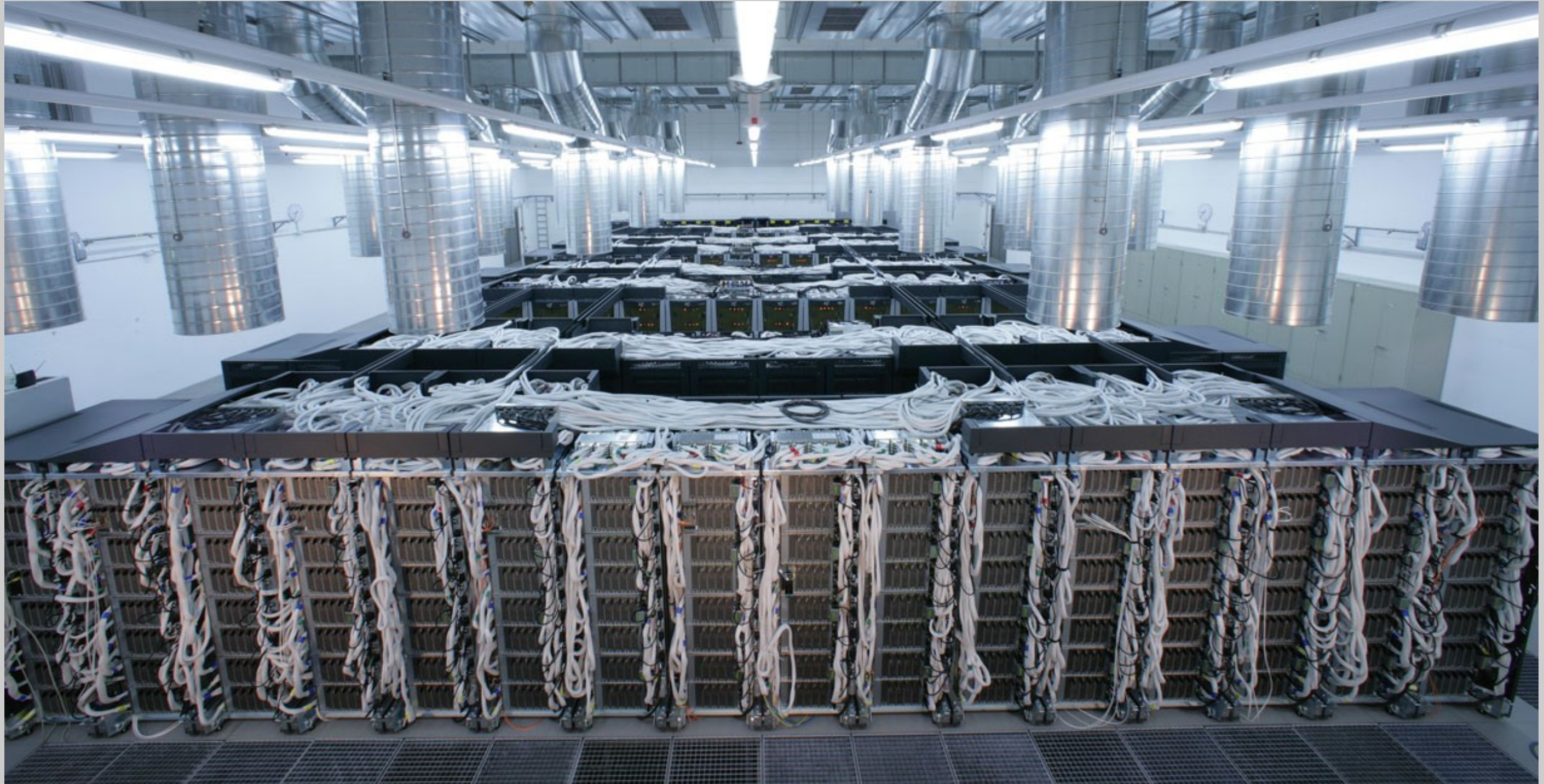


Qu'est-ce qu'un ordinateur ?

- Les ordinateurs peuvent prendre de multiples apparences...
 - Mais leur architecture interne et leur fonctionnement reste les mêmes
 - Ces machines sont conçues selon l'architecture dite de **Von Neumann**



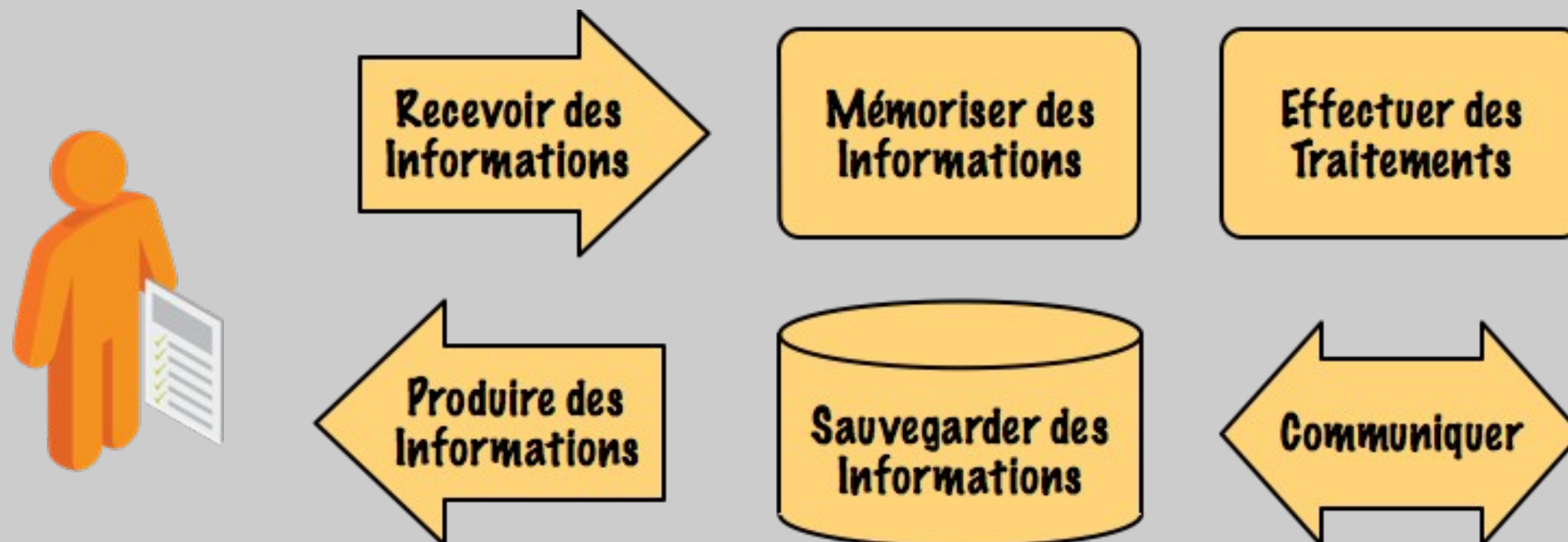
Certains ordinateurs ne tiennent pas sur une table :



SGI Altix / 4700 cores / 62 Teraflops = 62×10^{12} opérations / seconde

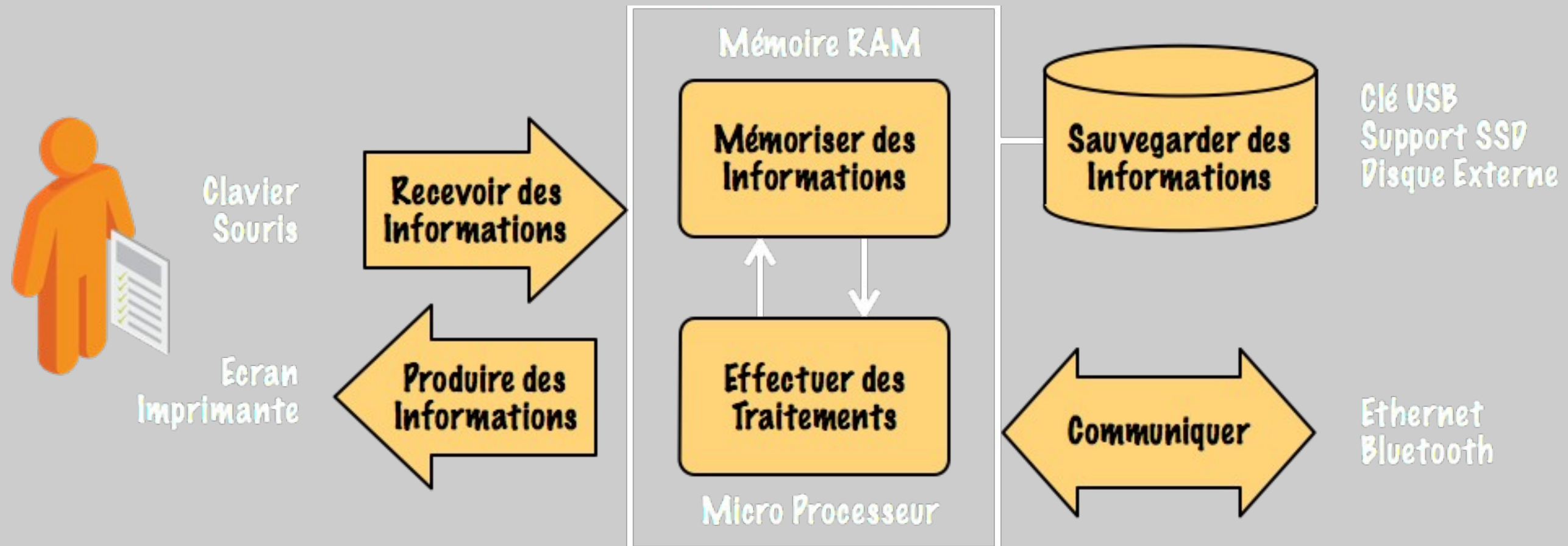
Architecture de Von Neumann (1946)

- Un ordinateur est un **système de traitement de l'information** qui peut :
 - **Recevoir** des informations et les encoder (périphériques d'entrée)
 - **Mémoriser** temporairement des informations (mémoire vive)
 - **Sauvegarder** et retrouver des informations (supports de stockage)
 - **Effectuer des traitements** sur ces informations (microprocesseur)
 - **Produire des informations** décodées (périphériques de sortie)
 - **Communiquer** avec d'autres systèmes (périphériques d'entrée/sortie)



Architecture de Von Neumann (1946)

- **Microprocesseur** : exécute le programme pas à pas mais très rapidement
- **Périphériques d'entrée** : clavier, souris, trackpad, microphone, etc.
- **Périphériques de sortie** : écran, imprimante, haut-parleur, etc.
- **Mémoire vive** : mémoire très rapide mais volatile (mémoire RAM, DRAM)
- **Mémoire de masse** : mémoire lente mais permanente (disque, clé USB, etc.)



Ou est le programme ?

- Le programme est chargé **dans la mémoire RAM** avec les données
- Le microprocesseur consulte une instruction du programme et l'exécute
- Cette opération est effectuée plusieurs millions de fois par seconde !



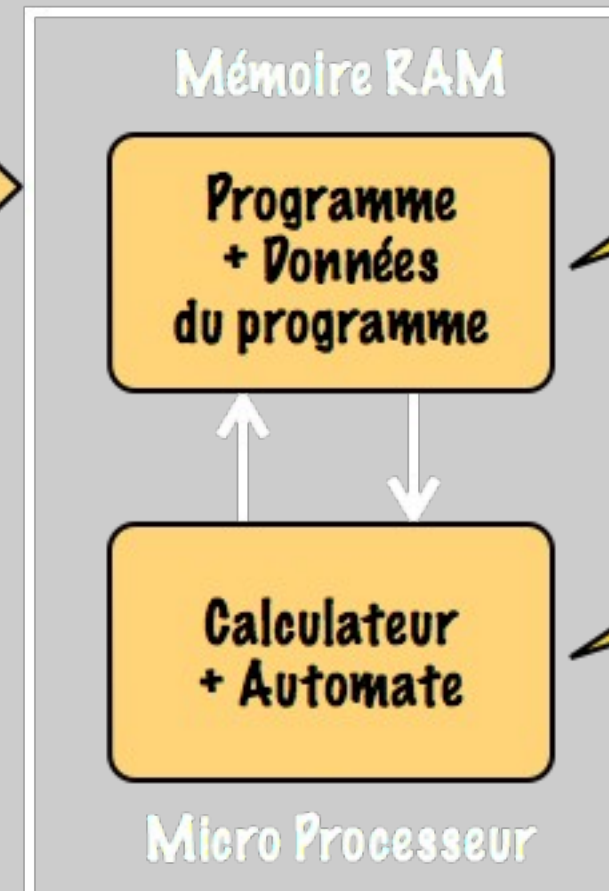
```
filename = raw_input("Enter file: ")
with open(filename, "r") as file:
    text = file.read()
words = text.split()

counts = dict()
for word in words:
    counts[word] = counts.get(word,0) + 1

bigcount, bigword = None, None
for word, count in counts.items():
    if bigcount == None or count > bigcount:
        bigword, bigcount = word, count

print bigword, bigcount
```

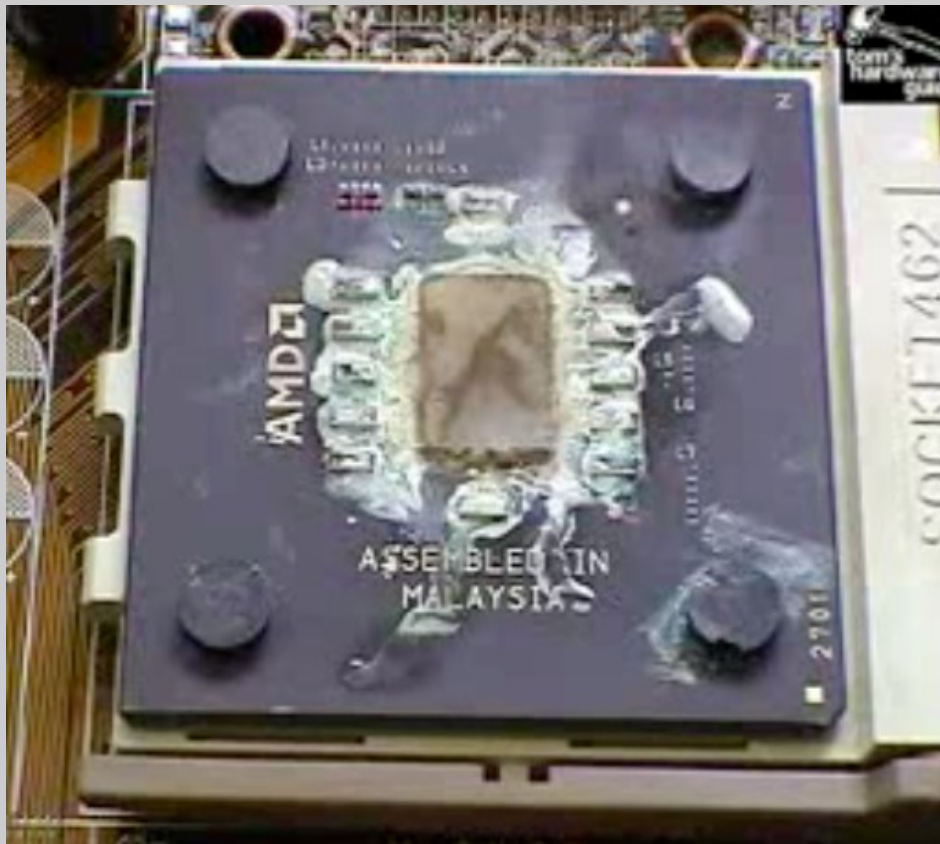
```
1. Default (bash)
Last login: Fri Jul 3 23:50:19 on ttys000
MacBook-de-Lehuen:~ lehuen$ cd Desktop/
MacBook-de-Lehuen:Desktop lehuen$ python test.py
Enter file: clown.txt
the 7
MacBook-de-Lehuen:Desktop lehuen$
```



001011010
011010011

What Next?

Surchauffe du processeur Un disque dur en action



<http://www.youtube.com/watch?v=y39D4529FM4>

<http://www.youtube.com/watch?v=9eMWG3fwiEU>

D'où vient le terme « ordinateur » ?

- En 1954, la compagnie IBM-France (International Business Machines) cherche un néologisme pour désigner ce que l'on appelle alors un « **calculateur** »
- Jacques Perret (1906-1992), professeur de philologie latine à la Sorbonne, propose un mot du vocabulaire théologique tombé en désuétude :
« **ordinateur** » (désigne Dieu qui met de l'ordre dans le monde)
- Protégé pendant quelques mois par IBM-France, le mot fut rapidement adopté par les spécialistes et par l'administration, puis passa dans le domaine public

> [Lettre de J. Perret](#)

Chapitre 1 (suite)

Le langage Python

Python est un langage de programmation

- Initialement développé par Guido van Rossum
 - Nom donné en hommage aux Monty Python
- Multi-plateformes et multi-paradigmes :
 - Programmation impérative, objet, fonctionnelle
- Bon compromis entre simplicité et puissance
- Outils de haut niveau + syntaxe relativement simple
- Typage dynamique fort + ramasse-miettes
 - La gestion de la mémoire est automatisée
- Types de données structurés :
 - Tuples, ensembles, listes, dictionnaires
- Système de gestion des exceptions



```
prompt$ python
```

```
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 00:54:21)
```

```
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
```

```
Type "copyright", "credits" or "license()" for more information.
```

```
>>> █
```



What
Next ?


```
prompt$ python
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 00:54:21)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> x = 1
>>> print(x)
1
>>> x = x + 1
>>> print(x)
2
>>> exit()
```

Il s'agit d'une utilisation en **mode interactif** :
vous tapez directement dans le terminal une ligne
après l'autre et vous obtenez la réponse après
chaque « retour-chariot »

Ceci est un bon test afin de s'assurer que vous
avez correctement installé Python.

Remarquez qu'il est aussi possible de taper **quit()**
pour terminer la session interactive.

Éléments de Python

- **Vocabulaire / Mots** : Variables et mots réservés (chapitre 2)
- **Structure de phrase** : Modèles de syntaxe valide (chapitres 3-5)
- **Structure de l'histoire** : Programmer dans un but particulier

Tout usage illégal d'un mot réservé ou toute formulation qui ne respecte pas la syntaxe de Python provoque une **SyntaxError** :

```
>>> print = 100
      File "<stdin>", line 1
        print = 100
              ^
SyntaxError: invalid syntax
>>> █
```

```
>>> x y = 100
      File "<stdin>", line 1
        x y = 100
          ^
SyntaxError: invalid syntax
>>> █
```

Mots réservés (vocabulaire)

- Vous ne pouvez pas utiliser ces **mots réservés** du langage en tant que noms de variables / de fonctions (identificateurs) :

Python 3.3

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

Lignes de code (instructions)

| | | | | | | |
|-------|---|---|---|---------------------------|-------------------------|------------------------------|
| x | = | 2 | ← | Instruction d'affectation | | |
| x | = | x | + | 2 | ← | Affectation d'une expression |
| print | (| x |) | ← | Affichage (sur l'écran) | |

Variable

Opérateur

Constante

Fonction

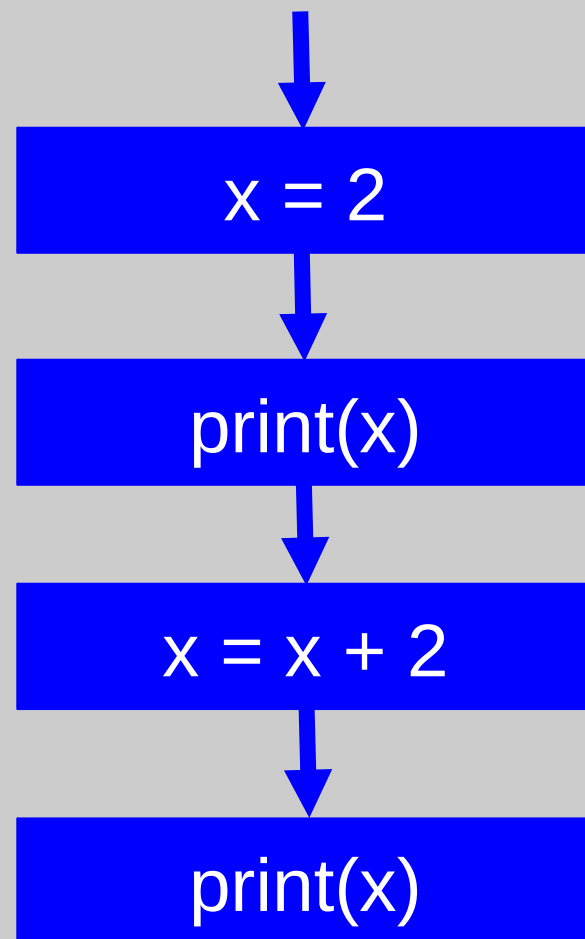
Scripts (programmes)

- Python en **mode interactif** peut être pratique pour tester des expressions ou des programmes de 3-4 lignes mais...
- La plupart des programmes sont beaucoup plus longs, c'est pourquoi nous les écrivons **dans un fichier** afin que python exécute les instructions à partir de ce fichier
- D'une certaine manière nous « donnons un **script** à Python »
- Par convention, nous ajoutons le suffixe « **.py** » à la fin de ces fichiers pour indiquer qu'ils contiennent des instructions Python

Flux d'exécution

- Tout comme une recette ou une chorégraphie, un programme est une **séquence d'étapes** à effectuer dans un certain **ordre**
- Certaines étapes sont **conditionnelles** = elles peuvent être « **sautées** »
- Parfois une étape (ou un groupe d'étapes) doivent être **répétés**
- Il est également possible d'identifier un groupe d'étapes susceptibles d'être **réutilisées** à différents endroits du programme

Etapes Séquentielles



Programme :

```
x = 2  
print(x)  
x = x + 2  
print(x)
```

Affichage :

```
→ 2  
→ 4
```

Quand un programme s'exécute, il passe d'une étape à la suivante. En tant que programmeurs, nous mettons en place des « chemins » que le programme doit suivre.

> Chapitre 2

Etapes Conditionnelles

Programme :

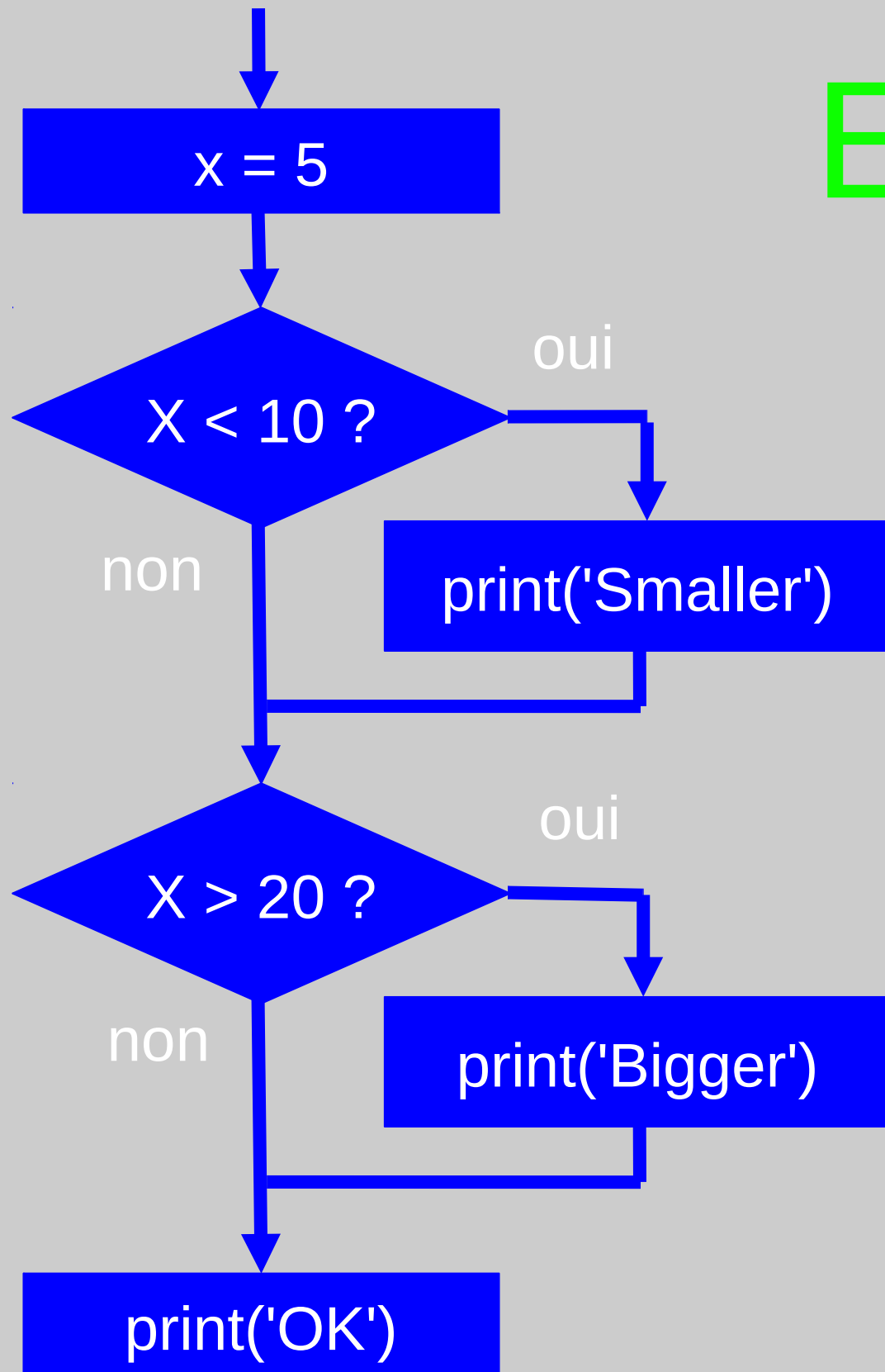
Affichage :

```
x = 5
if x < 10 :
    print('Smaller')
if x > 20 :
    print('Bigger')
print('OK')
```

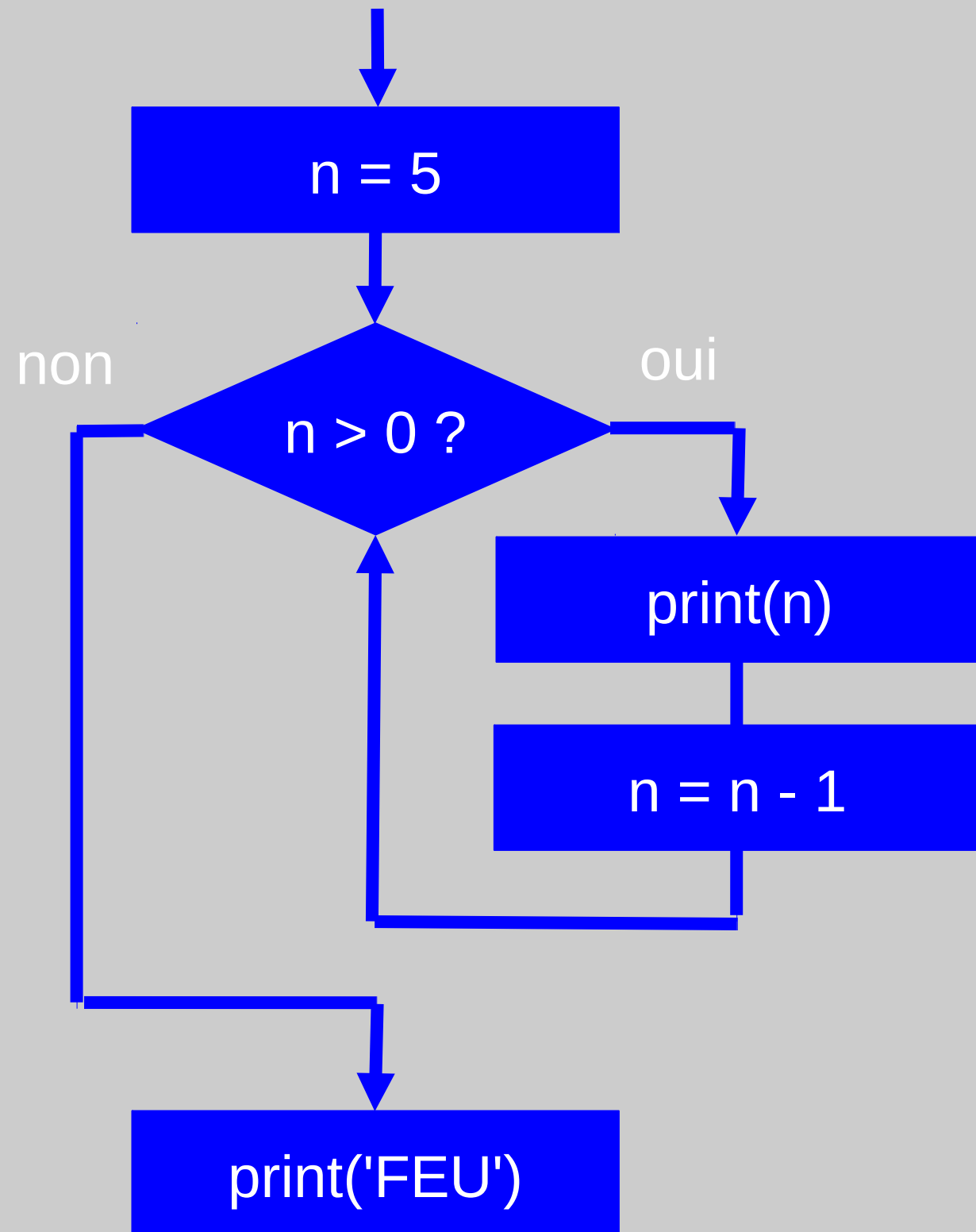
→ Smaller

→ OK

> Chapitre 3



Etapes Répétées



Programme :

```
n = 5
while n > 0 :
    print(n)
    n = n - 1
```

```
print('FEU')
```

Affichage :

5
4
3
2
1
FEU

Python 2 ou Python 3 ?

- Les langages informatiques évoluent (souvent dans le bon sens) :
 - Parfois il y a une « compatibilité ascendante »
 - Parfois il y a une « rupture de compatibilité »
- Les programmes écrits en **Python 2** ne sont pas complètement compatibles avec les interpréteurs **Python 3** (et inversement)...
- Pour l'informatique en Licence, nous avons opté pour **Python 3**

Attention : beaucoup de tutoriels et de forums traitent de Python 2



Remerciements / Contributions



These slides are Copyright 2010- Charles R. Severance () of the University of Michigan School of Information and made available under a Creative Commons Attribution 4.0 License. Please maintain this last slide in all copies of the document to comply with the attribution requirements of the license. If you make a change, feel free to add your name and organization to the list of contributors on this page as you republish the materials.

Initial Development: Charles Severance, University of Michigan School Of Information

Translation : Frederic Foiry

Revision : Stephanie Kamidian

Adaptation : Jérôme Lehuen

Chapitre 2

Constantes, Variables, Types, Expressions, Entrées-sorties

Constantes littérales

- Les valeurs fixes telles que les nombres, les lettres (caractères) et les chaînes de caractères sont appelées **constantes littérales** parce que leur valeur (donnée par le texte) ne peut pas changer
- Les **valeurs numériques** sont écrites naturellement
- Les **chaînes de caractères** utilisent des apostrophes simples ' ou doubles "

Pourquoi ?

```
>>> print(123)
123
>>> print(98.6)
98.6
>>> print('Hello world')
Hello world
```

Variables

- Une **variable** est un « emplacement nommé » dans la mémoire où un programmeur peut stocker des données (une à la fois) et les récupérer en utilisant le nom (identificateur) de la **variable**
- Il appartient aux programmeurs de choisir le nom des **variables**
- Le « contenu » (valeur) d'une **variable** change à chaque affectation

x = 12.2

y = 14

x = 100

x

y

~~12.2~~ 100

14

Règles de nommage des variables Python

- Commencent obligatoirement par une lettre ou un underscore _
- Contiennent des lettres, des chiffres et des underscores
- Sont sensibles à la case (majuscule / minuscule)
 - **Bon :** spam eggs spam23 _speed alert_on
 - **Mauvais :** 23spam #sign var.12
 - **Différents :** spam Spam SPAM

Que fait ce programme ?

| | | |
|-----------------------------------|-----------|----------------------|
| x1q3z9ocd = 35.0 | a = 35.0 | heures = 35.0 |
| x1q3z9afd = 12.50 | b = 12.50 | taux = 12.50 |
| x1q3p9afd = x1q3z9ocd * x1q3z9afd | c = a * b | paye = heures * taux |
| print(x1q3p9afd) | print(c) | print(paye) |

***Moralité : choisissez
intelligemment le nom
de vos variables !***

Mots réservés (vocabulaire)

- Vous ne pouvez pas utiliser ces **mots réservés** du langage en tant que noms de variables / de fonctions (identificateurs) :

Python 3.3

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

Lignes de code (instructions)

| | | | | | | |
|-------|---|---|---|---------------------------|-------------------------|------------------------------|
| x | = | 2 | ← | Instruction d'affectation | | |
| x | = | x | + | 2 | ← | Affectation d'une expression |
| print | (| x |) | ← | Affichage (sur l'écran) | |

Variable

Opérateur

Constante

Fonction

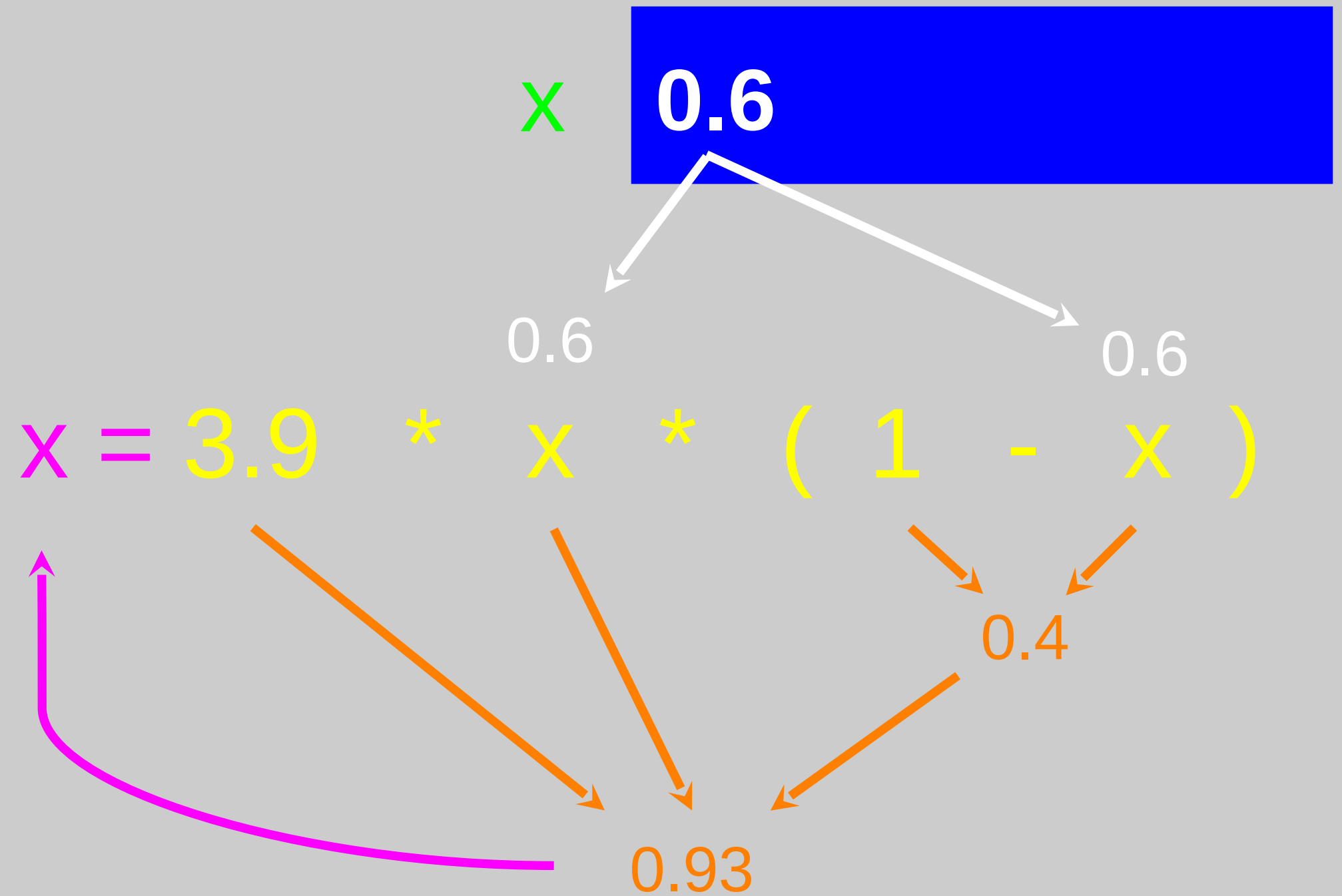
Instruction d'affectation

- Nous attribuons une valeur à une variable avec l'instruction d'affectation =
- Une affectation est composée d'une expression (à droite) et d'une variable (à gauche) qui va stocker le résultat de l'évaluation

$$\boxed{x} = \boxed{3.9 * x * (1 - x)}$$

- Tout ce qui renvoie quelque chose (possède une valeur) est une expression (constante, variable, opération, appel de fonction)

Au début, la valeur de la variable x est 0.6



L'expression est à droite

L'expression est évaluée

Le résultat est placé dans x

Après l'affectation, la valeur de la variable x est 0.93

x

~~0.6~~

0.93

La valeur 0.6
n'existe plus

$$x = 3.9 * x * (1 - x)$$

L'expression est à droite

L'expression est évaluée

0.93

Le résultat est placé dans x

Expressions Numériques

- En raison du manque de symboles mathématiques sur les claviers d'ordinateur nous utilisons le « dialecte de l'ordinateur » pour exprimer les opérations mathématiques classiques
- L'astérisque est la multiplication
- L'exposant (élévation à la puissance) est représenté par la double astérisque

| Opérateur | Opération |
|-----------|------------------|
| + | Addition |
| - | Soustraction |
| * | Multiplication |
| / | Division |
| // | Division entière |
| ** | Puissance |
| % | Modulo |

Expressions Numériques

```
>>> a = 2
>>> b = a + 2
>>> print(b)
4
>>> c = 440 * 12
>>> print(c)
5280
>>> d = c / 1000
>>> print(d)
5.28
```

```
>>> e = c // 1000
>>> print(e)
5
>>> f = c % 1000
>>> print(f)
280
```

| | | | | |
|-------------------|------|--|------|--------------------|
| | 5280 | | 1000 | |
| | 280 | | 5 | |
| Modulo (reste) | | | | Quotient entier |

| Opérateur | Opération |
|-----------|------------------|
| + | Addition |
| - | Soustraction |
| * | Multiplication |
| / | Division |
| // | Division entière |
| ** | Puissance |
| % | Modulo |

Evaluation des expressions

- Quand nous utilisons plusieurs opérateurs ensemble, il faut savoir dans quel ordre les opérateurs sont appliqués...
- Cet ordre est défini par des règles de priorité des opérateurs
- *Dans quel ordre les opérateurs suivants sont-ils appliqués ?*

$$x = 1 + 2 ** 3 // 4 * 5$$

Règles de priorité

Priorité la plus haute à la plus basse :

- > Les parenthèses sont toujours respectées
- > L'exposant (élévation à la puissance)
- > Multiplication, division et modulo (reste)
- > Addition et soustraction
- > De gauche à droite

Parenthèses
Puissances
Multiplications & div
Additions & sous
De gauche à droite



Règles de priorité

```
>>> x = 1 + 2 ** 3 // 4 * 5
>>> print(x)
11
>>>
```

Parenthèses
Puissances
Multiplications & div
Additions & sous
De gauche à droite



1 + 2 ** 3 // 4 * 5



1 + 8 // 4 * 5



1 + 2 * 5



1 + 10



11

Règles de priorité

Parenthèses
Puissances
Multiplications & div
Additions & sous
De gauche à droite



- Souvenez-vous des règles (du haut vers le bas)
- Quand vous écrivez du code => utilisez les parenthèses
- Quand vous écrivez du code => gardez les expressions mathématiques assez simple pour être comprises
- Décomposez de longues séries d'opérations mathématiques afin de les rendre plus lisibles

Qu'est-ce qu'un type ?

- En Python, les constantes littérales et les variables sont **typées**
- Python fait la différence entre un **entier** et une **chaîne de caractères**
- C'est grâce à cette différence que...

```
>>> v1 = 1 + 4
>>> print(v1)
5
>>> v2 = 'hello ' + 'there'
>>> print(v2)
hello there
>>> v3 = 'abc' * 5
>>> print(v3)
abcabcabcabcab
```

+ signifie « **addition** » pour les nombres et « **concaténation** » pour les chaînes

* signifie « **multiplication** » pour les nombres et « **concaténation multiple** » pour les chaînes

Qu'est-ce qu'un type ?

- Certaines opérations sont interdites :
- Par exemple, il n'est pas possible d'ajouter un nombre à une chaîne
- Nous pouvons demander à Python le type de quelque chose en utilisant la fonction `type()`

```
>>> v1 = 'hello ' + 'there'
>>> v2 = v1 + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and
'int' objects
>>> type(v1)
<class 'str'> —————→ string (chaîne)
>>> type('hello')
<class 'str'>
>>> type(1)
<class 'int'> —————→ integer (entier)
>>> type(1.0)
<class 'float'> —————→ virgule flottante
>>>
```


Conversions de type

- Quand nous utilisons des entiers et des nombres à virgule flottante dans une expression, les entiers sont **implicitement** convertis en float

Le type float est « contaminant »

- On peut également **forcer** une conversion de type avec les fonctions **int()** et **float()**

```
>>> print(2 * 2.5)
5.0
>>> n = 42
>>> type(n)
<class 'int'>
>>> f = float(n)
>>> print(f)
42.0
>>> type(f)
<class 'float'>
>>> i = int(f)
>>> print(i)
42
>>> type(i)
<class 'int'>
```

Type d'une division

- Exercices :

```
>>> print(4 / 2)
```

```
2.0
```

```
>>> print(4 // 2)
```

```
2
```

```
>>> print(4.5 / 2)
```

```
2.25
```

```
>>> print(4.5 // 2)
```

```
2.0
```

```
>>> print(int(4.5) / 2)
```

```
2.0
```

```
>>> print(int(4.5) // 2)
```

```
2
```

| Type du dividende | Type de la division | Type du diviseur | Type du quotient |
|------------------------------|---------------------|------------------------------|------------------|
| int int float float | / | int float int float | float |
| int | // | int | int |
| int float float | // | float int float | float |

Le type float est « contaminant »

Conversion des chaînes

- On peut utiliser `int()` ou `float()` pour effectuer la conversion d'une chaîne vers un entier ou vers un float :
- On obtient une **erreur** si la chaîne contient autre-choses que des chiffres :

```
>>> sval = '123'
>>> type(sval)
<class 'str'>
>>> print(sval + 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to
str implicitly
```

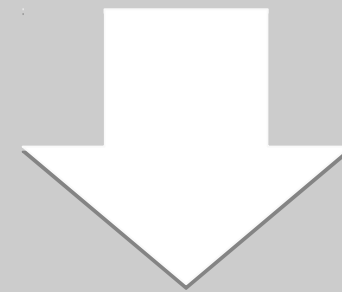
```
>>> ival = int(sval)
>>> type(ival)
<class 'int'>
>>> print(ival + 1)
124
```

```
>>> sval= 'hello bob'
>>> ival = int(sval)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int()
with base 10: 'hello bob'
```

Instructions d'E/S

- La fonction `input()` bloque le déroulement du programme et attend une saisie au clavier. Elle retourne une chaîne de caractères

```
name = input('Qui êtes-vous ?')  
print('Bienvenue', name)
```



- La fonction `print()` permet d'afficher des données (constantes, variables, expressions) séparées par des **virgules** (une virgule = un espace)

Qui êtes-vous ?
Chuck
Bienvenue Chuck

Les sorties formatées

- Python possède une fonctionnalité de **formatage** des chaînes de caractères à l'aide de l'opérateur **%**

chaîne à formater **%** **variable**, **expression** ou « **tuple** »

```
>>> prenom = 'Jérôme'
>>> print('Bonjour %s' % prenom)
Bonjour Jérôme
```

```
>>> v1 = 745.8112
>>> v2 = 745.8183
>>> print('Résultat = %.2f' % v1)
Résultat = 745.81
>>> print('Résultat = %.2f' % v2)
Résultat = 745.82
```

```
>>> num = 41
>>> rue = 'rue des Fauvettes'
>>> pos = 72000
>>> vil = 'Le Mans'

>>> adr = '%i %s %i %s' % (num, rue, pos, vil)
>>> print(adr)
41 rue des Fauvettes 72000 Le Mans
```

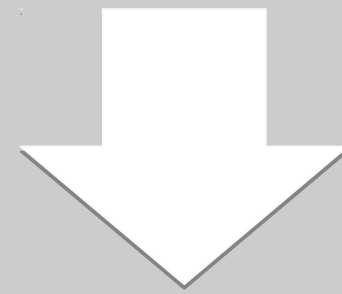
tuple



Conversion des entrées

- Si nous voulons lire un nombre de l'utilisateur, nous devons le convertir de chaîne à nombre avec la fonction `int()`

```
an_cour = 2015
an_nais = input('Année de naissance :')
age = an_cour - int(an_nais)
print('Vous avez', age, 'ans')
```



*Plus tard nous verrons
comment traiter les
« mauvaises » entrées...*

Année de naissance : 1968
Vous avez 47 ans


```
# Get the name of the file and open it
filename = input('Nom du fichier: ')
with open(filename, 'r') as file:
    text = file.read()
words = text.split()

# Count word frequency
counts = dict()
for word in words:
    counts[word] = counts.get(word,0) + 1

# Find the most common word
bigword, bigcount = None, None
for word, count in counts.items():
    if bigcount == None or count > bigcount:
        bigword, bigcount = word, count

# Print the result
print(bigword, bigcount)
```

Encodage du code source

- En informatique, les chaînes de caractères sont mémorisées selon une norme de codage **MAIS il y a plusieurs normes de codage :**

ASCII Windows-1252 UTF-8 ISO-8859-1
UTF-16 MacRoman
ISO-8859-6 UTF-32

- Pour éviter tout problème, vous sauvegarderez vos fichiers en **UTF-8** et vous préciserez l'encodage au tout début de vos programmes :

```
# -*- coding: utf-8 -*-
```



Remerciements / Contributions



Thes slides are Copyright 2010- Charles R. Severance (www.dr-chuck.com) of the University of Michigan School of Information and open.umich.edu and made available under a Creative Commons Attribution 4.0 License. Please maintain this last slide in all copies of the document to comply with the attribution requirements of the license. If you make a change, feel free to add your name and organization to the list of contributors on this page as you republish the materials.

Initial Development: Charles Severance, University of Michigan School Of Information

Translation : Frederic Foiry
Revision : Stephanie Kamidian
Adaptation : Jérôme Lehuen

Chapitre 3

Structures Conditionnelles
Expressions Booléennes
Gestion des Exceptions

Etapes Conditionnelles

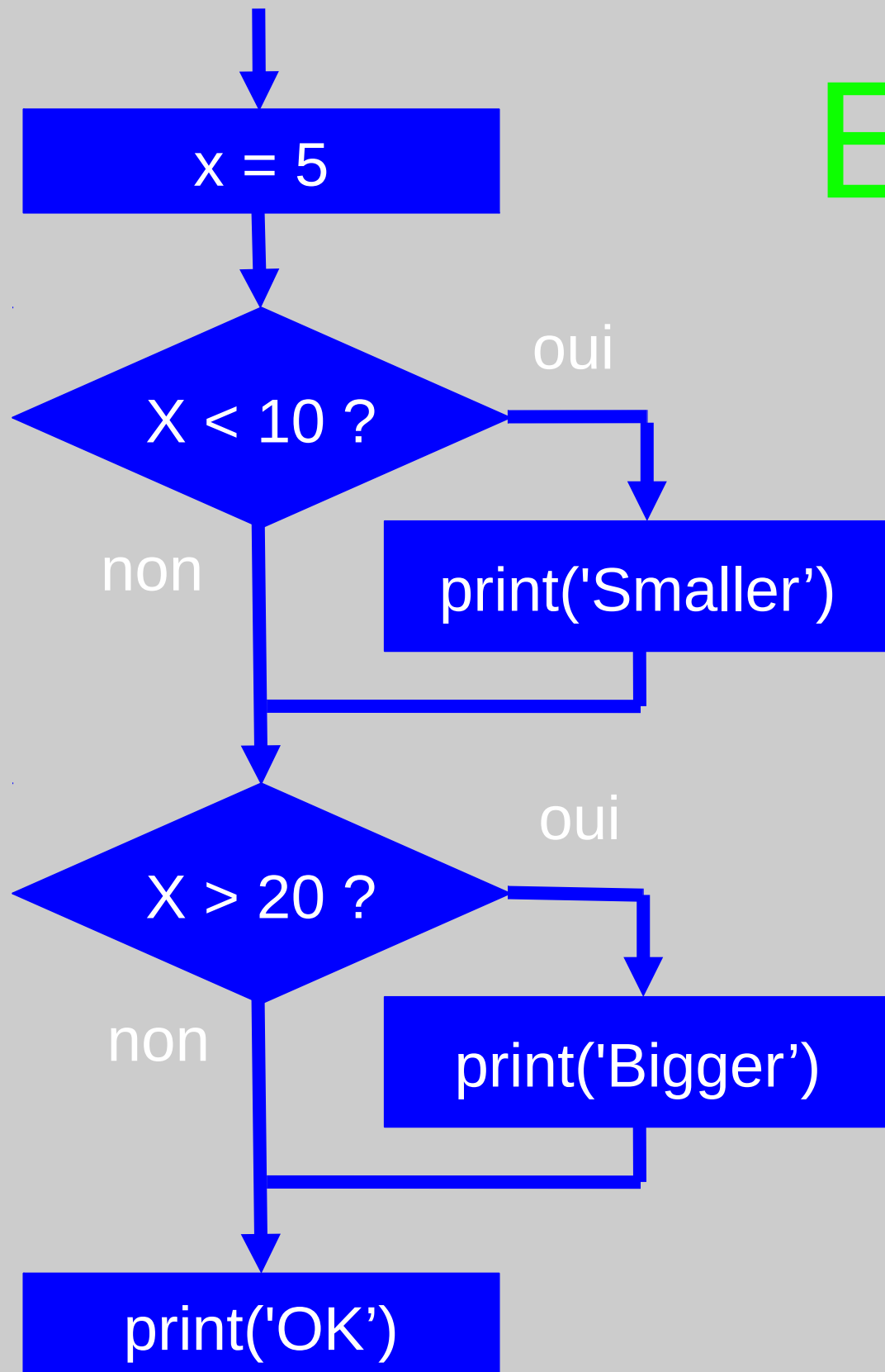
Programme :

Affichage :

```
x = 5
if x < 10 :
    print('Smaller')
if x > 20 :
    print('Bigger')
print('OK')
```

→ Smaller

→ OK



Opérateurs de Comparaison

- Les expressions Booléennes permettent de contrôler le déroulement du programme
- Les expressions Booléennes utilisent les opérateurs de comparaison et retournent les valeurs vrai / faux (oui / non) lorsqu'elles sont évaluées
- Les opérateurs de comparaison contrôlent les variables mais ne les modifient pas !!!

| Opérateur binaire | Signification |
|-------------------|---------------------|
| < | Inférieur à |
| <= | Inférieur ou égal à |
| == | Egal à |
| >= | Supérieur ou égal à |
| > | Plus grand que |
| != | Différent de |

| Opérateur unaire | Signification |
|------------------|---------------|
| not | négation |

Opérateurs de Comparaison

| | | |
|---------------------------------|---|------------------------|
| x = 5 | | Affichage à l'écran : |
| if x == 5 : | | |
| print('Egal à 5') | → | Egal à 5 |
| if x > 4 : | | |
| print('Plus grand que 4') | → | Plus grand que 4 |
| if x >= 5 : | | |
| print('Plus grand ou égal à 5') | → | Plus grand ou égal à 5 |
| if x < 6 : | | |
| print('Inférieur à 6') | → | Inférieur à 6 |
| if x <= 5 : | | |
| print('Inférieur ou égal à 5') | → | Inférieur ou égal à 5 |
| if x != 6 : | | |
| print('Différent de 6') | → | Différent de 6 |

Décisions “tout ou rien”

```
x = 5
print('Avant test 5')
if x == 5 :
    print('Egal à 5')
    print('Toujours 5')
    print('Encore 5')
print('Après test 5')

print('Avant test 6')
if x == 6 :
    print('Egal à 6')
    print('Toujours 6')
    print('Encore 6')
print('Après test 6')
```

→ Avant test 5

→ Egal à 5

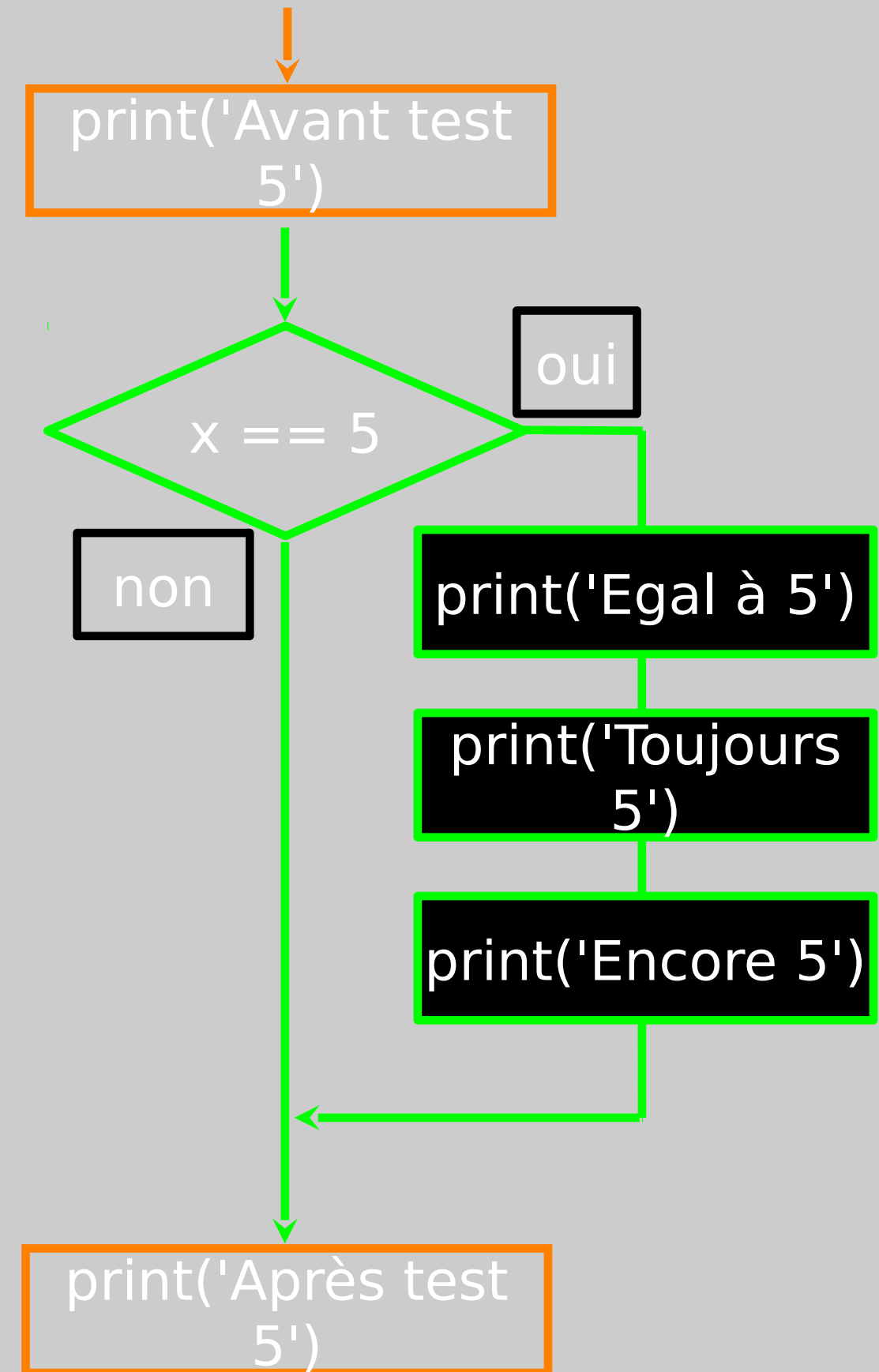
→ Toujours 5

→ Encore 5

→ Après test 5

→ Avant test 6

→ Après test 6

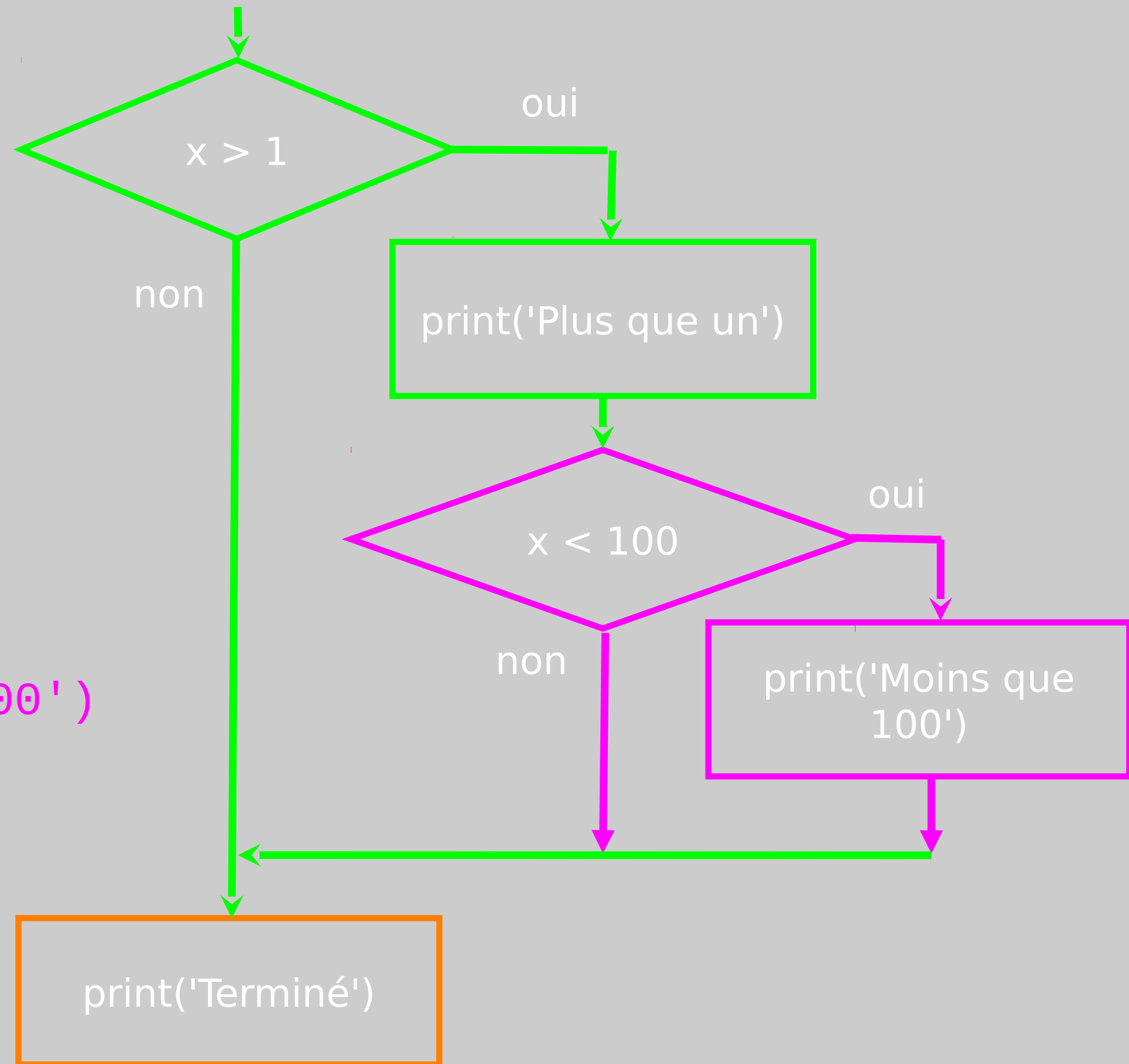


Parlons d'indentation...

- Augmenter l'indentation après le mot-clé **if** pour définir le **début du bloc**
- Maintenir l'indentation pour indiquer la **portée du bloc**
- Ramener l'indentation au niveau du mot-clé **if** pour définir la **fin du bloc**
- Les lignes blanches (vides) sont ignorées : elles n'affectent pas l'indentation
- **ATTENTION : Ne mélangez pas tabulations et espaces !!!**
 - Vous aurez des « indentation errors » même si tout vous paraît normal

Décisions imbriquées

```
x = 42
if x > 1 :
    print('Plus que un')
    if x < 100 :
        print('Moins que 100')
print('Terminé')
```



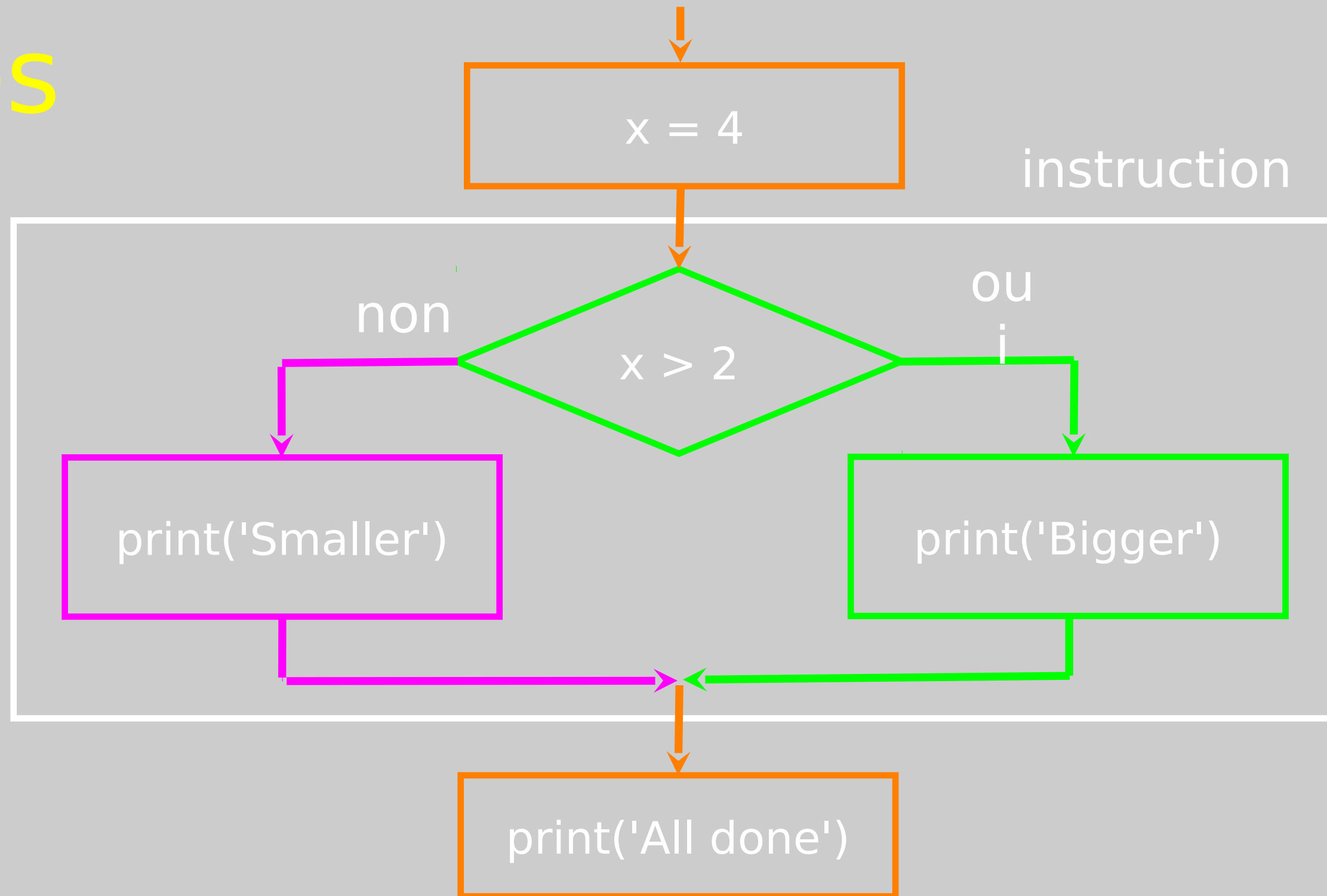
Décisions à deux voies

`x = 4`

instruction

```
if x > 2 :  
    print('Bigger')  
else :  
    print('Smaller')
```

`print('All done')`



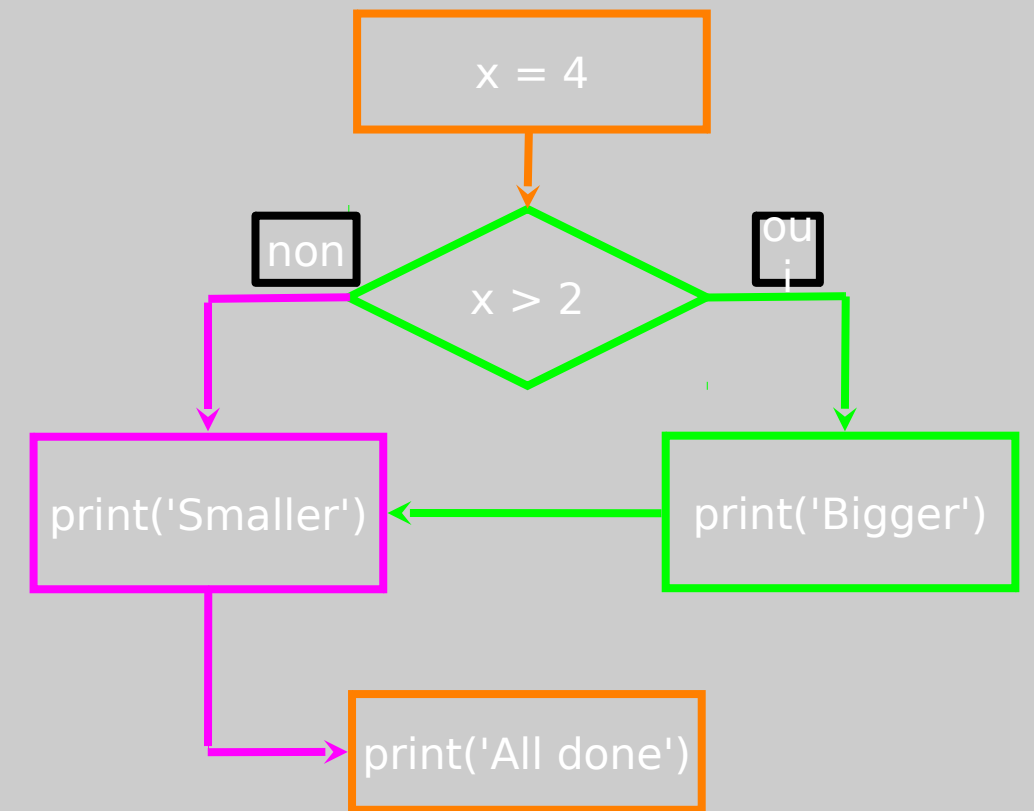
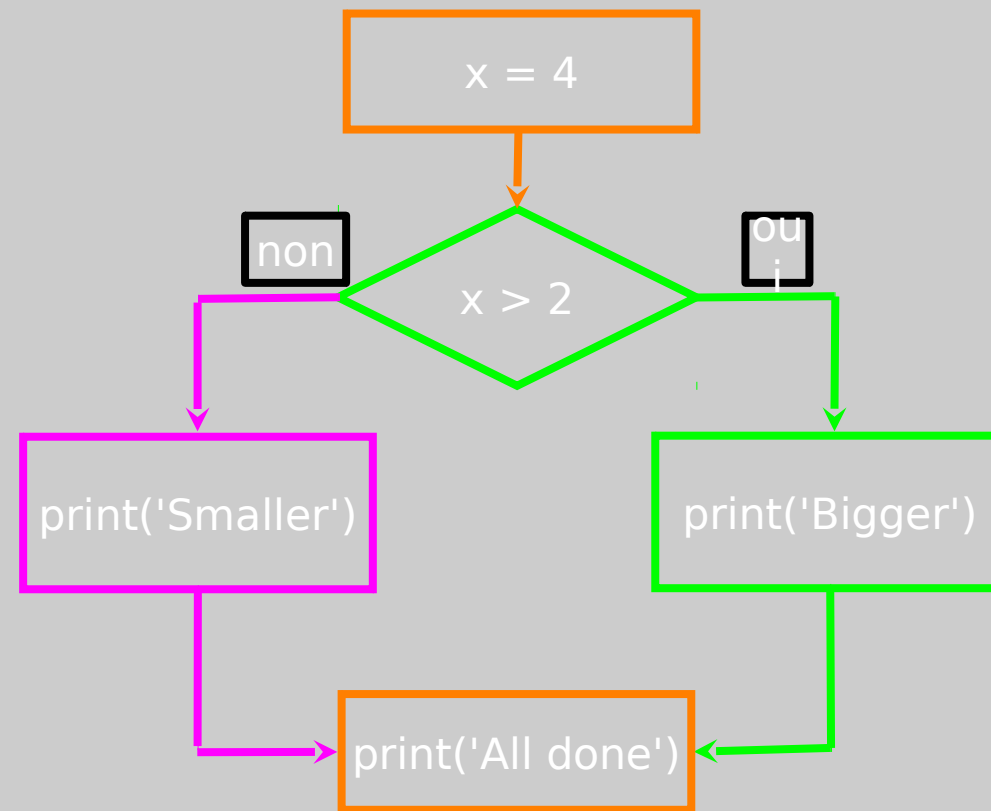
Décisions à deux voies

```
x = 4
if x > 2 :
    print('Bigger')
else :
    print('Smaller')
print('All done')
```

```
x = 4
if x > 2 :
    print('Bigger')

print('Smaller')
print('All done')
```

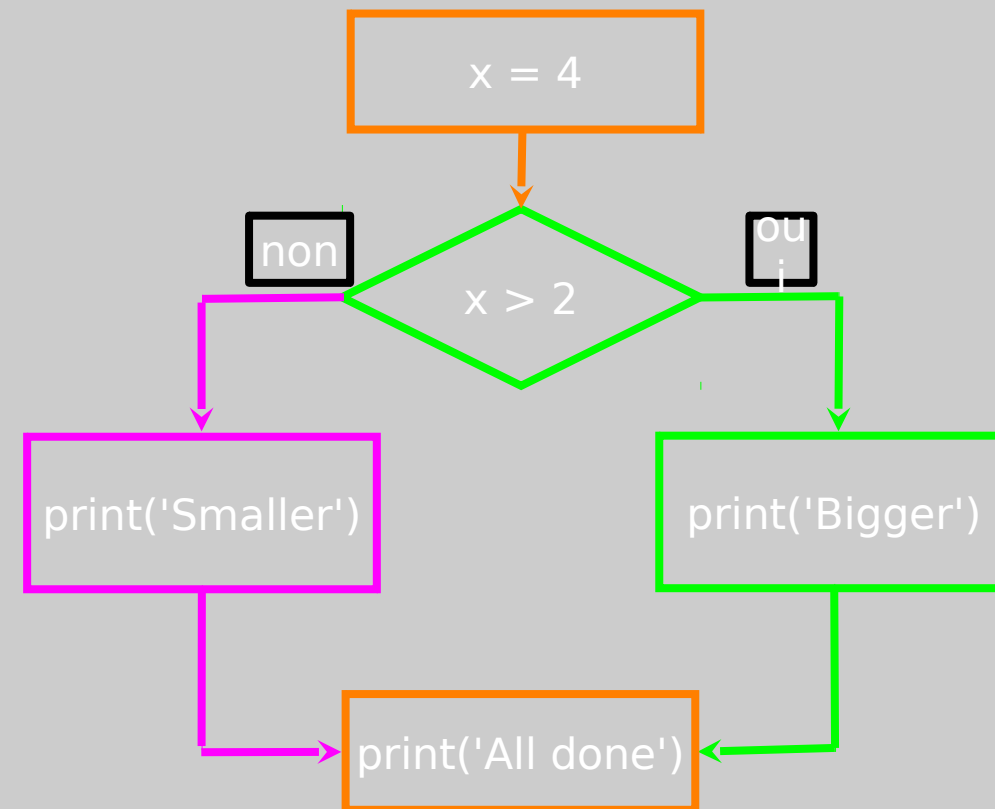
*Pourquoi
le « else »
est utile ?*



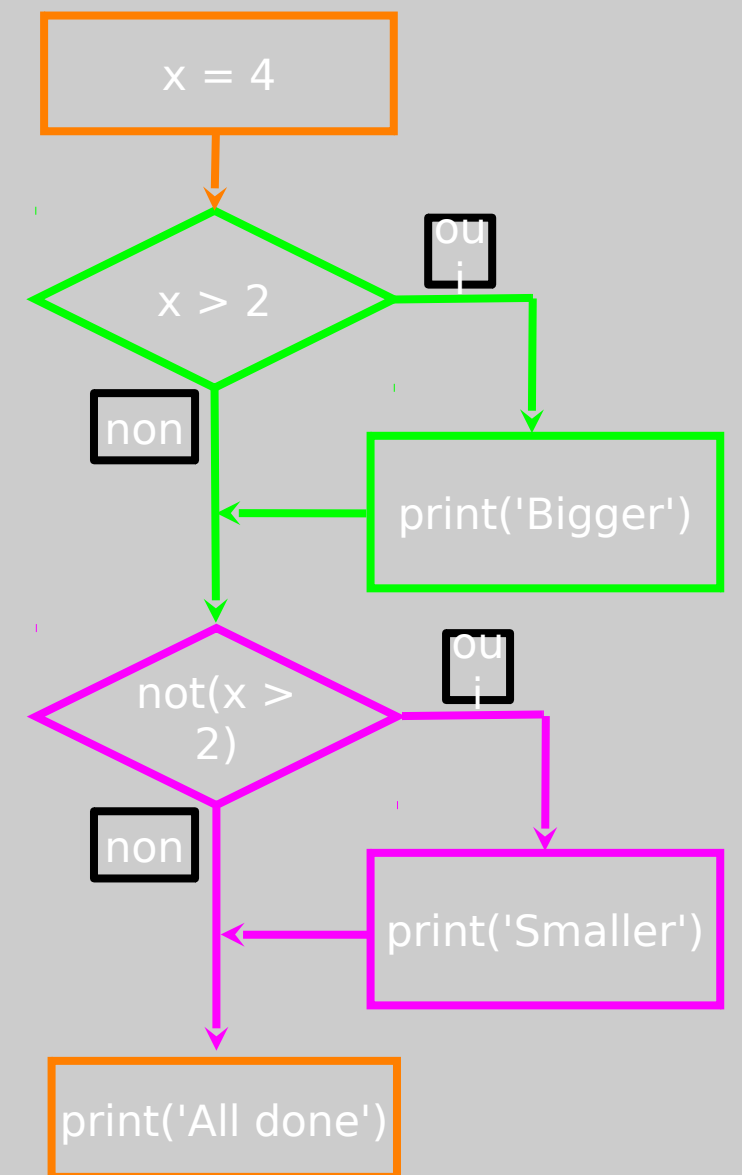
Décisions à deux voies

```
x = 4
if x > 2 :
    print('Bigger')
else :
    print('Smaller')
print('All done')
```

*Pourquoi
le « else »
est utile ?*

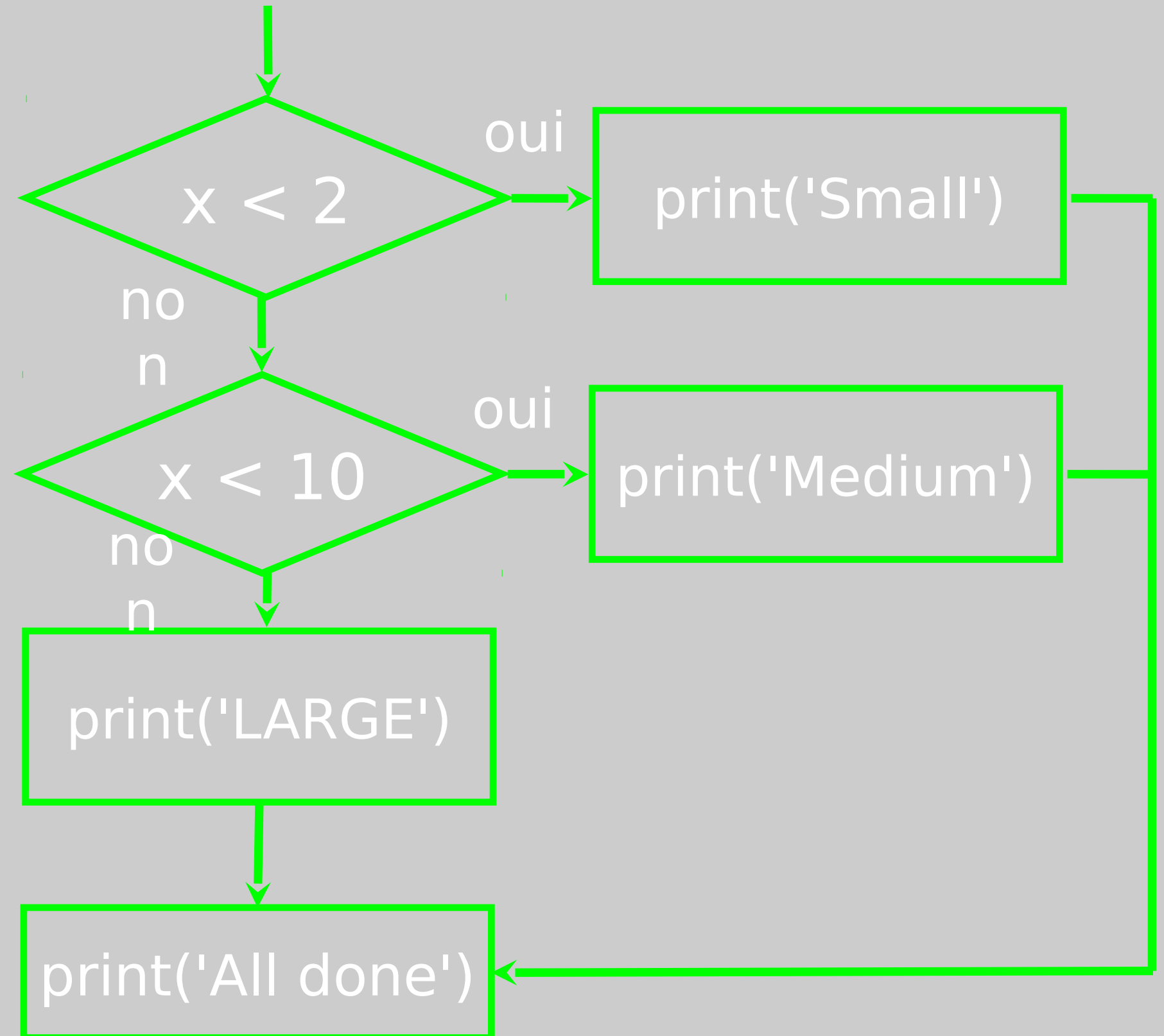


```
x = 4
if x > 2 :
    print('Bigger')
if not(x > 2):
    print('Smaller')
print('All done')
```



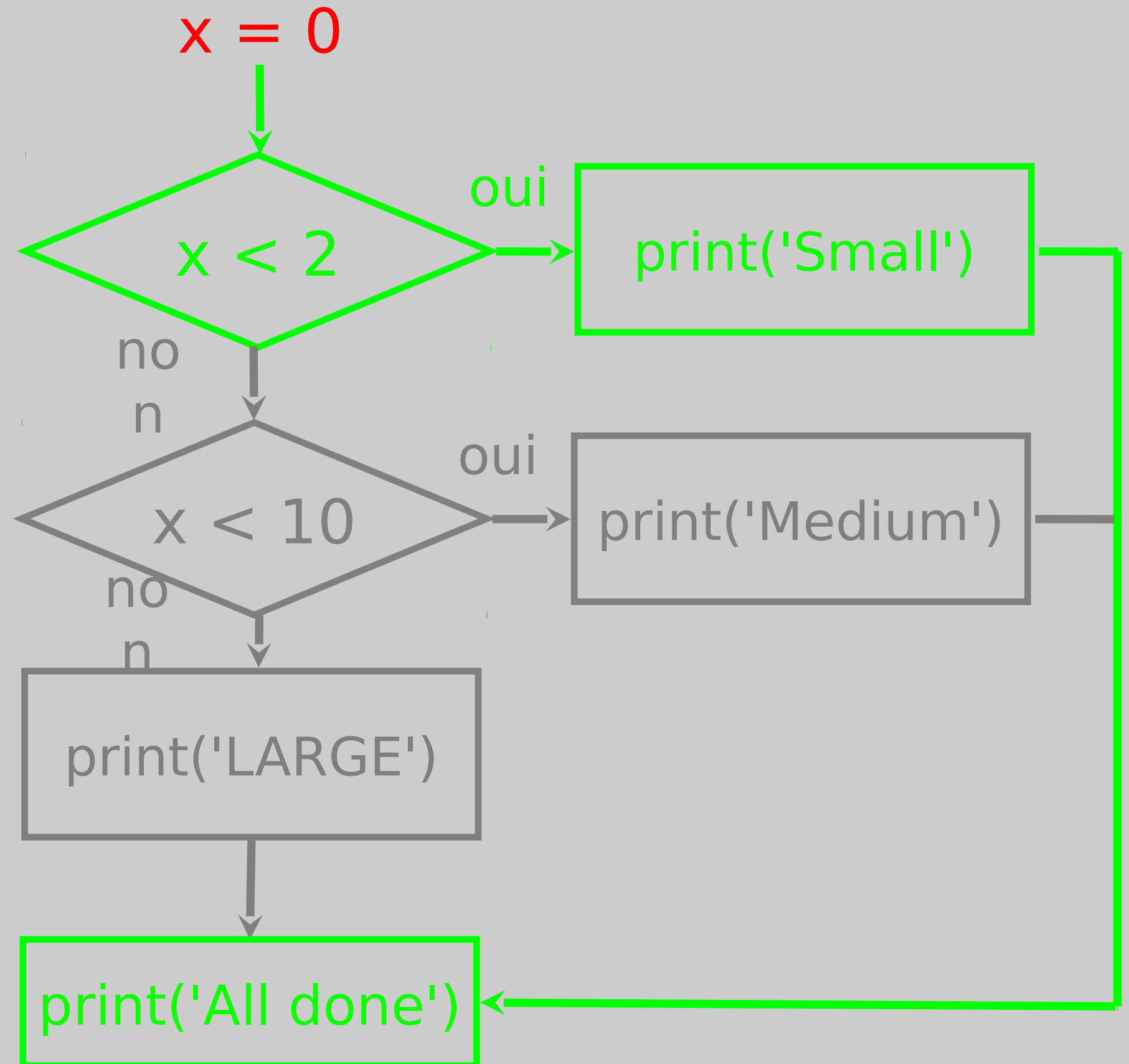
Décisions à voies multiples

```
if x < 2 :  
    print('Small')  
elif x < 10 :  
    print('Medium')  
else :  
    print('LARGE')  
print('All done')
```



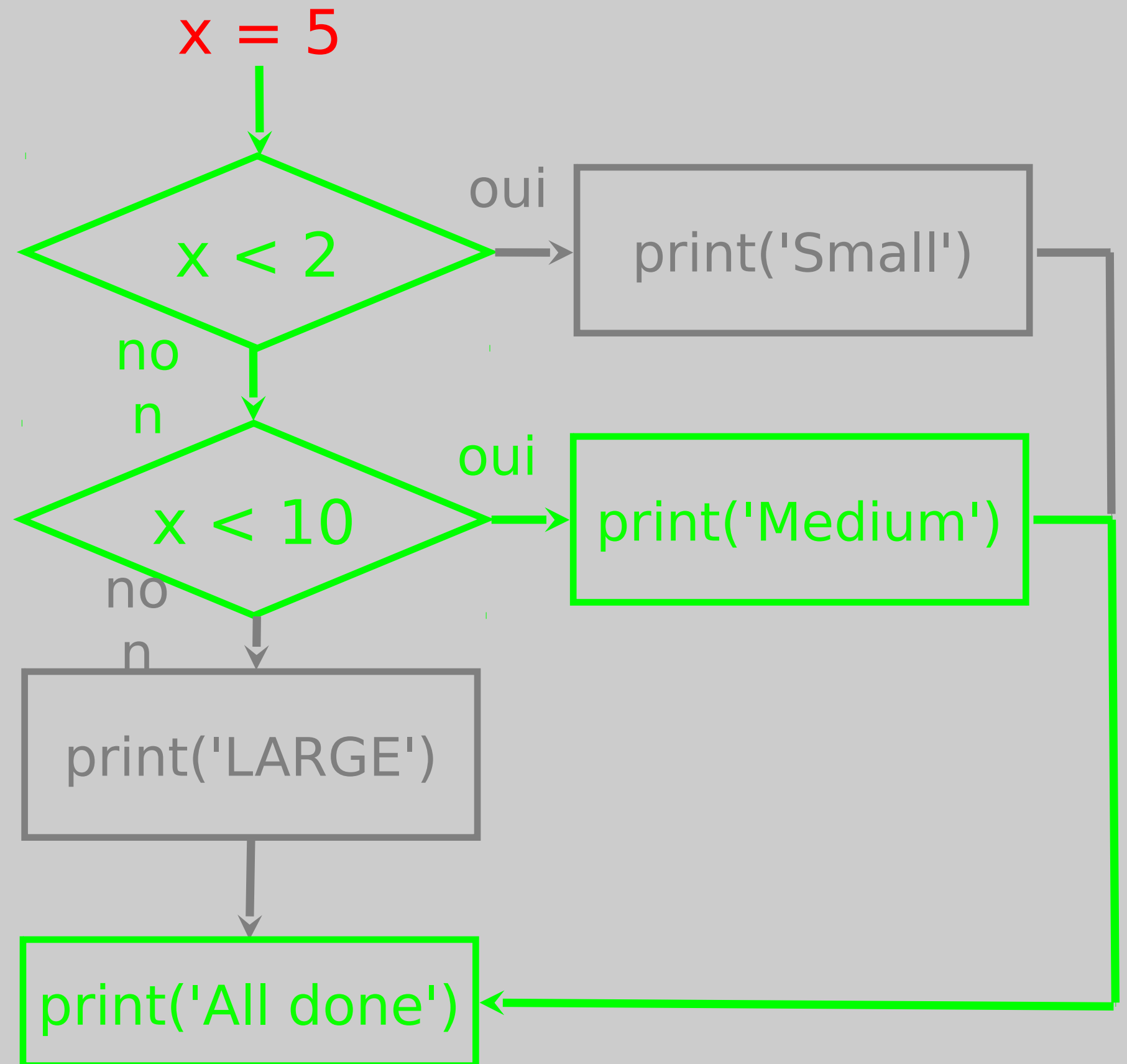
Décisions à voies multiples

```
x = 0
if x < 2 :
    print('Small')
elif x < 10 :
    print('Medium')
else :
    print('LARGE')
print('All done')
```



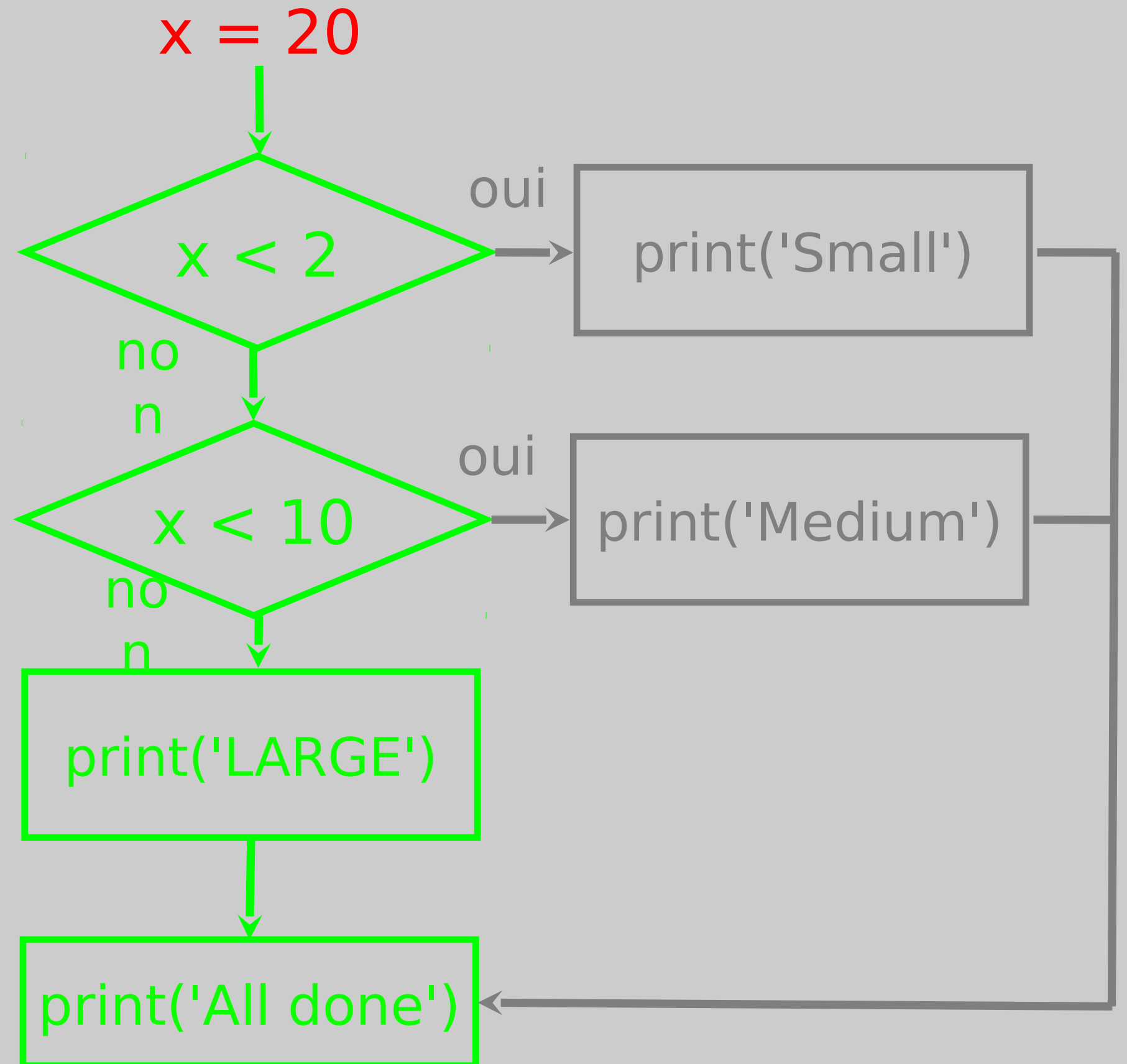
Décisions à voies multiples

```
x = 5
if x < 2 :
    print('Small')
elif x < 10 :
    print('Medium')
else :
    print('LARGE')
print('All done')
```



Décisions à voies multiples

```
x = 20
if x < 2 :
    print('Small')
elif x < 10 :
    print('Medium')
else :
    print('LARGE')
print('All done')
```



Que penser de ces structures ?

```
if x < 2 :  
    print('Below 2')  
elif x >= 2 :  
    print('Two or more')  
else :  
    print('Something else')
```

```
if x < 2 :  
    print('Below 2')  
elif x < 20 :  
    print('Below 20')  
elif x < 10 :  
    print('Below 10')  
else :  
    print('Something else')
```

Que penser de ces structures ?

```
if x < 2 :  
    print('Below 2')  
elif x >= 2 :  
    print('Two or more')  
else :  
    print('Something else')
```

```
if x < 2 :  
    print('Below 2')  
elif x < 20 :  
    print('Below 20')  
elif x < 10 :  
    print('Below 10')  
else :  
    print('Something else')
```

Les lignes en rouge ne sont pas accessibles !

Pour vérifier une structure, construire son
« puzzle »

Chapitre 3 (suite)

Expressions booléennes

Les expressions booléennes

- Une **expression booléenne** est une expression dont les deux seules valeurs possibles sont **True** et **False**
- Le type d'une expression booléenne est le type booléen noté **<class 'bool'>** en Python
- On peut mémoriser et manipuler les expressions booléennes à l'aide de variables

George Boole (1815-1864) est un logicien, mathématicien, philosophe britannique. Il est le créateur de la logique moderne, fondée sur une structure algébrique et sémantique, que l'on appelle aujourd'hui algèbre de Boole.

```
>>> print(3 > 4)
False
>>> print(3 < 4)
True
>>> type(3 < 4)
<class 'bool'>
>>> a = 3
>>> b = 4
>>> c = a > b
>>> print(c)
False
>>> type(c)
<class 'bool'>
```

Les opérateurs booléens

- Les **opérateurs booléens** permettent de combiner des expressions booléennes afin de construire des expressions de + en + complexes mais dont les valeurs sont toujours **True** ou **False**
- On représente souvent les opérateurs booléens par leur **table de vérité** :

| A | not A |
|-------|-------|
| False | True |
| True | False |

| A | B | A and B |
|-------|-------|---------|
| False | False | False |
| False | True | False |
| True | False | False |
| True | True | True |

| A | B | A or B |
|-------|-------|--------|
| False | False | False |
| False | True | True |
| True | False | True |
| True | True | True |

Les opérateurs booléens

- Comme pour les expressions numériques, il est conseillé de combiner **opérateurs** et **parenthèses** lorsque les expressions se complexifient :
- Toute expression booléenne (simple ou complexe) peut être utilisée comme **condition** dans une **structure conditionnelle** :

```
valide = ((a > b) and (b > c)) or (a == 0)
```

```
if not(valide):
```

```
...
```

```
...
```

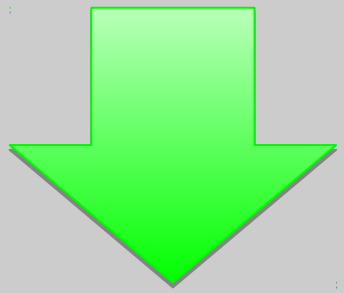

Chapitre 3 (suite)

Gestion des Exceptions

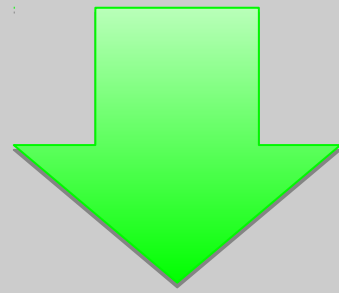
Il y a des erreurs qu'on ne peut pas éviter :

```
an_cour = 2015
an_nais = input('Année de naissance :')
age = an_cour - int(an_nais)
print('Vous avez %i ans' % age)
```

*... on les appelle
des exceptions*



Année de naissance : 1968
Vous avez 47 ans



Année de naissance : SALUT

Traceback (most recent call last):

File "/Users/lehuen/Desktop/test.py", line 4, in <mod

age = an_cour - int(an_nais)

ValueError: invalid literal for int() with base 10: 'SALUT'

En revanche, on peut les anticiper et les gérer :

- Pour les gérer on dispose de la structure **try / except** pour la quelle il faut :

- Identifier la **portion de code** où l'exception peut survenir (être levée)
- Ecrire le **code à exécuter** s'il n'y a pas d'exception (traitement normal)

- Ecrire le **code à exécuter** en cas d'exception (traitement exceptionnel)

```
# -*- coding: utf-8 -*-
an_cour = 2015
an_nais = input("Année de naissance : ")
try:
    age = an_cour - int(an_nais)
    print('Vous avez %i ans' % age)
except:
    print("Erreur: vous n'avez pas saisi une année")
```

La gestion des exceptions

```
# -*- coding: utf-8 -*-
```

```
an_cour = 2015
```

```
an_nais = input('Année:')
```

```
try:
```

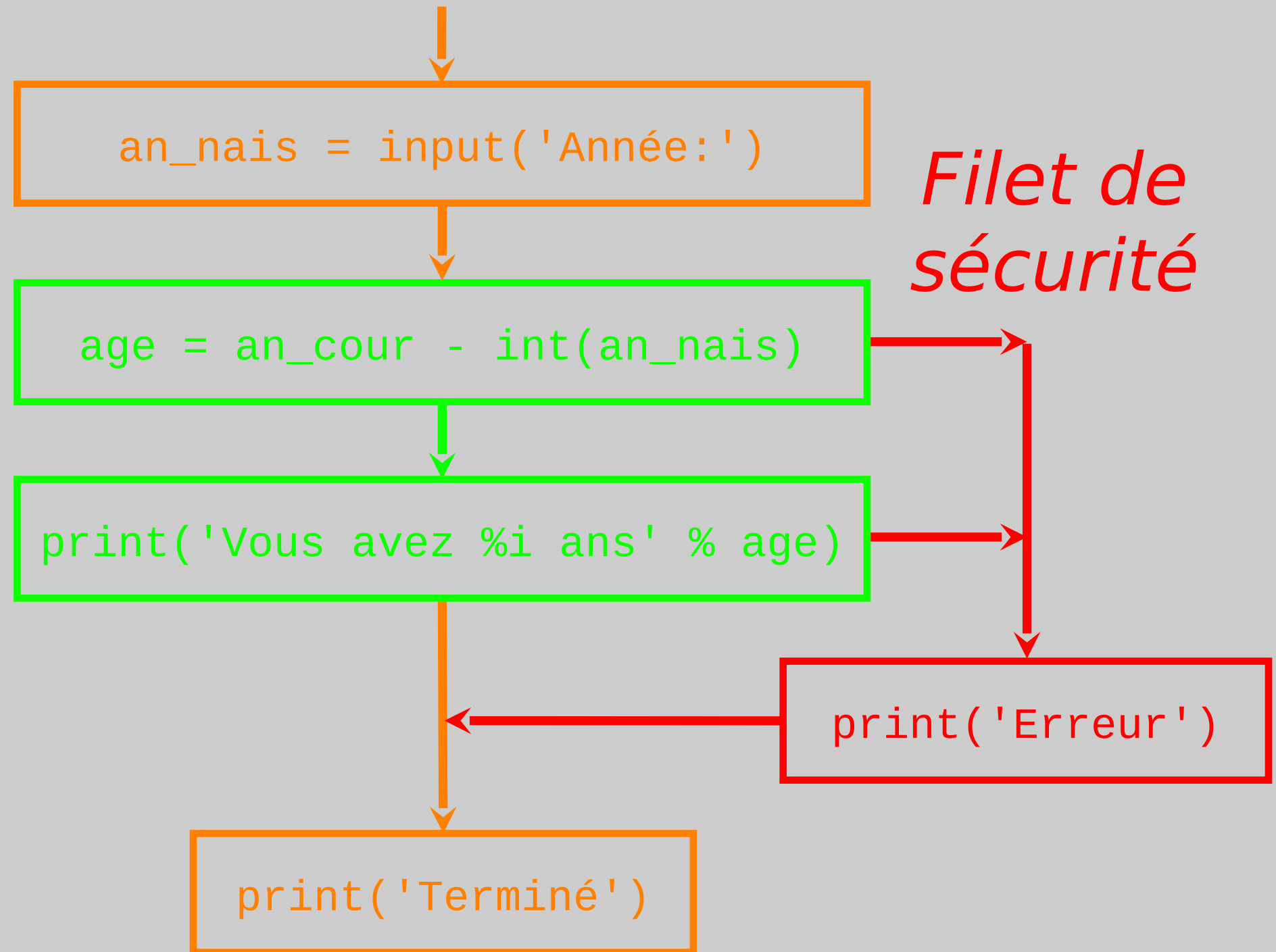
```
    age = an_cour - int(an_nais)
```

```
    print('Vous avez %i ans' % age)
```

```
except:
```

```
    print('Erreur: pas une année')
```

```
print('Terminé')
```





Remerciements / Contributions



Thes slides are Copyright 2010- Charles R. Severance (www.dr-chuck.com) of the University of Michigan School of Information and open.umich.edu and made available under a Creative Commons Attribution 4.0 License. Please maintain this last slide in all copies of the document to comply with the attribution requirements of the license. If you make a change, feel free to add your name and organization to the list of contributors on this page as you republish the materials.

Initial Development: Charles Severance, University of Michigan School Of Information

Translation : Frederic Foiry
Revision : Stephanie Kamidian
Adaptation : Jérôme Lehuen

Chapitre 4

Fonctions prédéfinies et définies par l'utilisateur

Les fonctions en Informatique

$\sin(x)$

$f(x) = x * \sin(x)$

- Comme en maths, on peut considérer deux familles de **fonctions** :
 - > Les fonctions **prédéfinies** qui sont fournies avec le langage
 - > Les fonctions **définies par l'utilisateur** (le programmeur en fait)
- Contrairement à une fonction mathématique, une fonction en informatique ne renvoie pas forcément un **résultat**, elle peut juste avoir des **effets de bord** (affichages, traitements en mémoire)

Les fonctions en Informatique

- On peut voir les fonctions comme des « boîtes noires » qui savent effectuer des calculs ou des traitements particuliers :



- Dans la boîte, il y a du code informatique mais ce n'est pas nécessaire de le connaître, il faut juste savoir comment l'utiliser

Quelques fonctions Python

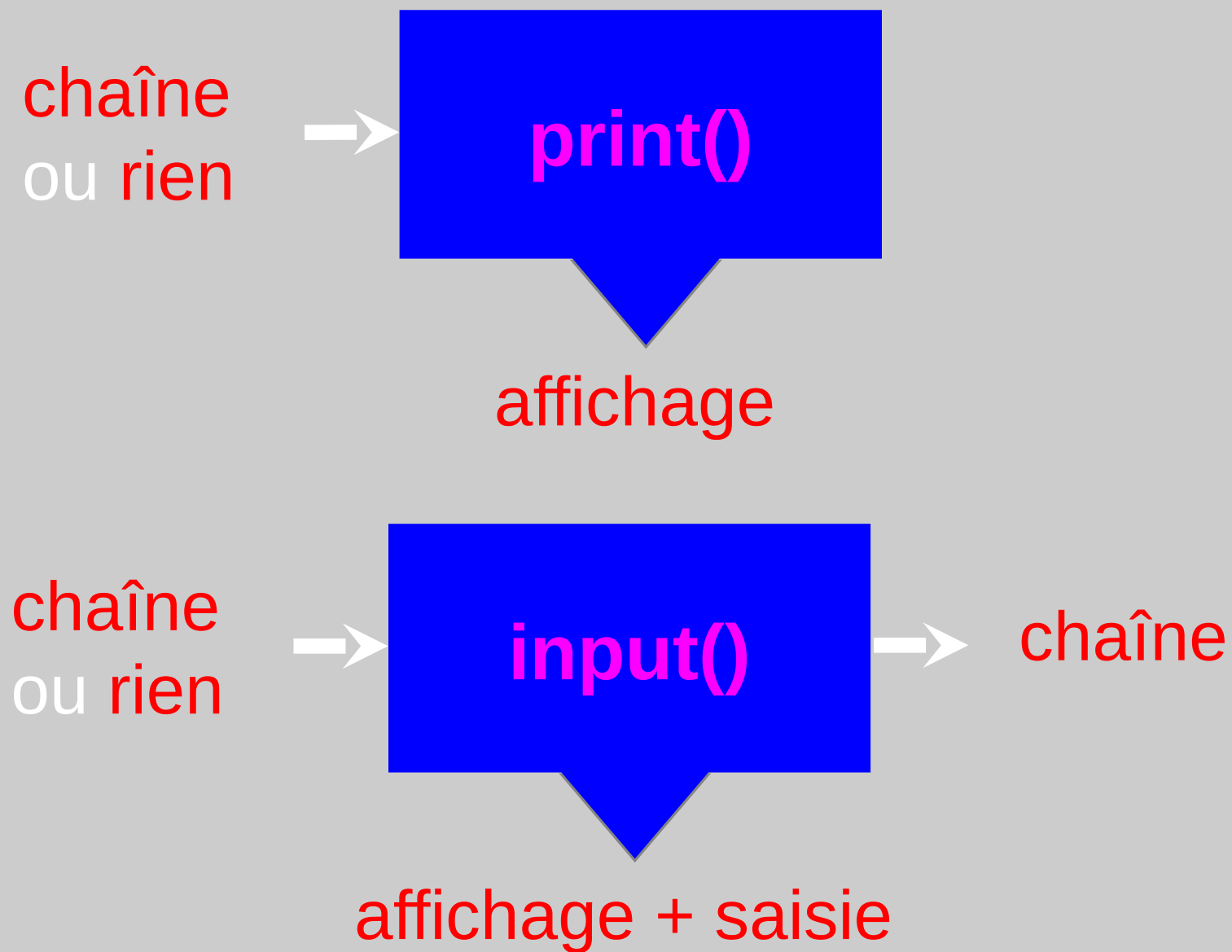
entier
ou flottant
ou chaîne → **float()** → flottant

entier
ou flottant → **sin()** → flottant

random() → flottant

```
>>> print(float(3))  
3.0  
>>> print(float('3'))  
3.0  
>>> print(sin(pi/2))  
1.0  
>>> print(random())  
0.737897260521  
>>> print(random())  
0.972452891561
```

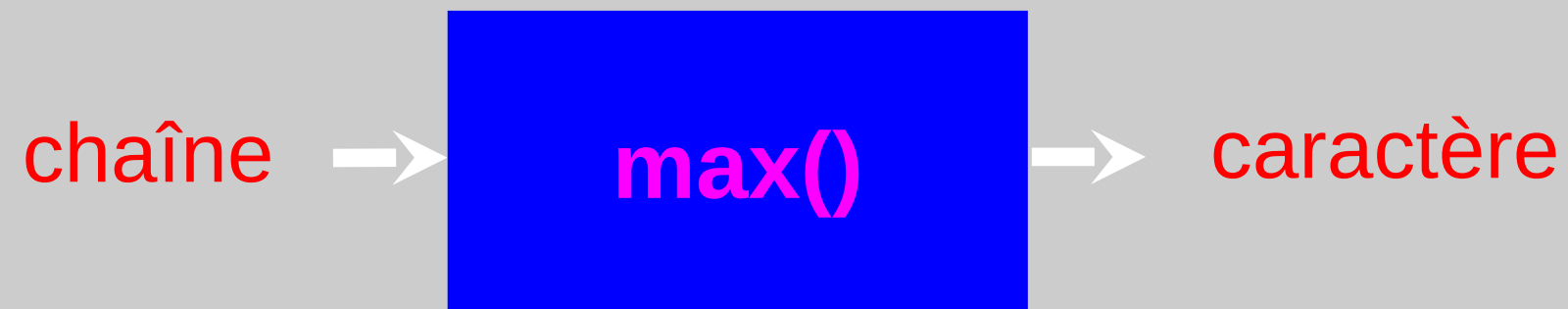
Fonctions avec effets de bord



```
>>> print('texte: ')\ntexte:\n>>> print()
```

```
>>> input('texte: ')\ntexte: bonjour\n>>> n = input('texte: ')\ntexte: hello\n>>> n = input()\nhello
```

Une fonction « polymorphe »



```
>>> print(max('hello world'))
```

```
w
```

```
>>> print(max('Hello World'))
```

```
r
```

```
>>> print(max(5))
```

```
TypeError: 'int' object  
is not iterable
```

```
>>> print(max(5,6))
```

```
6
```

```
>>> print(max(5,6,4))
```

```
6
```

Nombre et types des arguments

- *Attention au nombre et aux types des arguments !!!*

> Si les types des arguments changent, ce n'est pas forcément la même fonction pour un même nom de fonction :

| | | |
|-------------------------|-------------------------|-------------------------|
| <code>float(3)</code> | <code>float(3.0)</code> | <code>float('3')</code> |
| <code>max('abc')</code> | <code>max(3)</code> | |

> Si le nombre des arguments change, ce n'est pas forcément la même fonction pour un même nom de fonction :

| | |
|---------------------|------------------------|
| <code>max(3)</code> | <code>max(3, 4)</code> |
|---------------------|------------------------|

Les fonctions prédéfinies

| Built-in Functions | | | | |
|----------------------------|--------------------------|---------------------------|-------------------------|-----------------------------|
| <code>abs()</code> | <code>dict()</code> | <code>help()</code> | <code>min()</code> | <code>setattr()</code> |
| <code>all()</code> | <code>dir()</code> | <code>hex()</code> | <code>next()</code> | <code>slice()</code> |
| <code>any()</code> | <code>divmod()</code> | <code>id()</code> | <code>object()</code> | <code>sorted()</code> |
| <code>ascii()</code> | <code>enumerate()</code> | <code>input()</code> | <code>oct()</code> | <code>staticmethod()</code> |
| <code>bin()</code> | <code>eval()</code> | <code>int()</code> | <code>open()</code> | <code>str()</code> |
| <code>bool()</code> | <code>exec()</code> | <code>isinstance()</code> | <code>ord()</code> | <code>sum()</code> |
| <code>bytearray()</code> | <code>filter()</code> | <code>issubclass()</code> | <code>pow()</code> | <code>super()</code> |
| <code>bytes()</code> | <code>float()</code> | <code>iter()</code> | <code>print()</code> | <code>tuple()</code> |
| <code>callable()</code> | <code>format()</code> | <code>len()</code> | <code>property()</code> | <code>type()</code> |
| <code>chr()</code> | <code>frozenset()</code> | <code>list()</code> | <code>range()</code> | <code>vars()</code> |
| <code>classmethod()</code> | <code>getattr()</code> | <code>locals()</code> | <code>repr()</code> | <code>zip()</code> |
| <code>compile()</code> | <code>globals()</code> | <code>map()</code> | <code>reversed()</code> | <code>__import__()</code> |
| <code>complex()</code> | <code>hasattr()</code> | <code>max()</code> | <code>round()</code> | |
| <code>delattr()</code> | <code>hash()</code> | <code>memoryview()</code> | <code>set()</code> | |

<https://docs.python.org/3/library/functions.html>

Les librairies de Python

- En Python, la plupart des fonctions sont « rangées » par familles :
 - Les fonctions mathématiques sont dans la librairie **math**
 - Les générateurs aléatoires sont dans la librairie **random**
 - Les fonctions systèmes sont dans les librairies **sys** et **os**
- Pour utiliser ces fonctions, il faut **importer leur librairie** au début du programme. Deux syntaxes différentes pour utiliser une librairie :

```
>>> import math
>>> print(math.sin(math.pi/2))
1.0
```

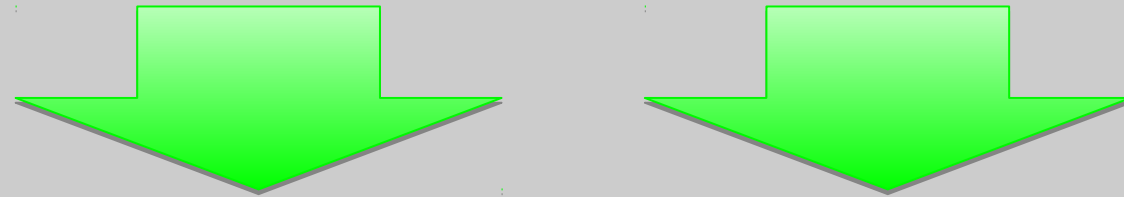
```
>>> from math import *
>>> print(sin(pi/2))
1.0
```

Chapitre 4 (suite)

Fonctions définies par l'utilisateur

Définir vos propres fonctions ?

- Si une même portion de code est **dupliquée à plusieurs endroits**
- Si vous avez conçu et implémenté un **algorithme réutilisable**




- Alors vous avez intérêt à définir une nouvelle fonction :

```
def print_lyrics():  
    print("I'm a lumberjack, and I'm okay.")  
    print('I sleep all night and I work all day.')
```

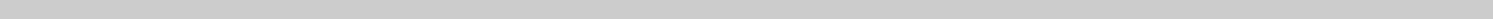
Définir une fonction ne signifie pas l'exécuter :

```
print('Hello')
```



Hello

```
def print_lyrics():
```




Yo

```
    print("I'm a lumberjack, and I'm okay.")  
    print('I sleep all night and I work all day.')
```

```
print('Yo')
```

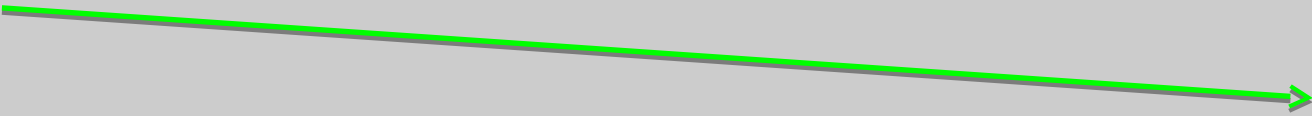


```
print('Hello')
```



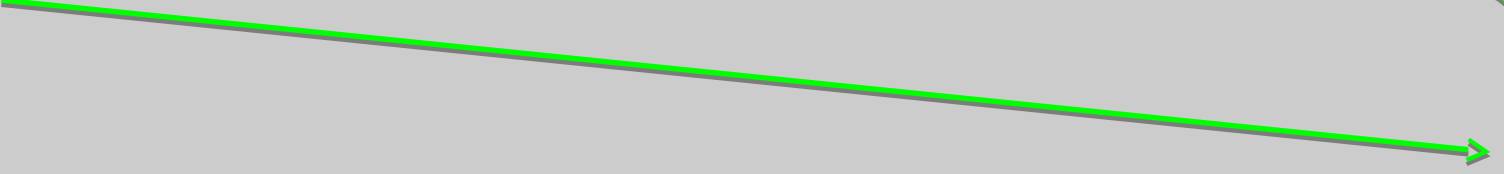
Hello

```
print_lyrics()
```



I'm a lumberjack, and...

```
print('Yo')
```



I sleep all night and...

Yo

Argument vs. paramètre

- Un **argument** est une **valeur** que nous transmettons à la fonction comme entrée (donnée) au moment de l'appel de la fonction
- Un **paramètre** est une **variable** que nous utilisons dans la définition de la fonction et qui prend la valeur d'un **argument** lors de l'appel

```
def greet(lang):  
    if lang == 'es':  
        print('Hola')  
    elif lang == 'fr':  
        print('Bonjour')  
    else:  
        print('Hello')
```

```
>>> greet('en')  
Hello  
>>> greet('es')  
Hola  
>>> greet('fr')  
Bonjour
```

Portée des paramètres

- La **portée d'une variable** correspond à la **portion de code** dans laquelle la variable en question possède une valeur
- La portée des paramètres d'une fonction correspond à la définition de la fonction => **en dehors, les paramètres ne sont pas définis**

```
def greet(lang):  
    if lang == 'es':  
        print('Hola')  
    elif lang == 'fr':  
        print('Bonjour')  
    else:  
        print('Hello')
```

```
>>> greet('fr')  
Bonjour  
>>> print(lang)  
Traceback (most recent call last):  
  File "Untitled.py", line 10, in <module>  
    print(lang)  
NameError: name 'lang' is not defined
```

Retourner une valeur

- Les fonctions qui retournent (renvoient) un résultat dans l'expression appelante doivent comporter le mot-clé **return**

```
def greet():  
    return "Hello"
```

```
>>> print(greet(), "Glenn")  
Hello Glenn
```

```
>>> print(greet(), "Sally")  
Hello Sally
```

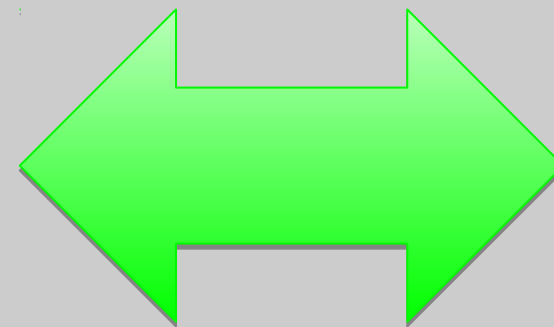
- Le **type d'une fonction** correspond au **type de la valeur retournée**

Retourner une valeur

- Il peut y avoir plusieurs **return** dans une fonction :

> Exemple d'une fonction qui retourne le mois en toutes lettres à partir du numéro du mois dans l'année :

```
def mois(n):  
    if n == 1:  
        return('janvier')  
    elif n == 2:  
        return('février')  
    elif n == 3:  
        return('mars')  
    ...
```



```
def mois(n):  
    if n == 1:  
        return('janvier')  
    if n == 2:  
        return('février')  
    if n == 3:  
        return('mars')  
    ...
```

Dès qu'un return est exécuté, le flux d'exécution « sort de la fonction »

Retourner plusieurs valeurs

- Les fonctions peuvent accepter plusieurs arguments en entrée...

> *Est-il possible de retourner plusieurs valeurs en sortie ?*



```
def foobar(a,b):  
    x = a + b  
    y = a - b  
    return(x,y)  
  
>>> foobar(1,2)  
(3, -1)
```

> *Comment récupérer les deux valeurs retournées ?*

```
>>> (i,j) = foobar(1,2)  
print(i, j)  
3 -1
```

Un **TUPLE** est un groupe de valeurs réunies au sein d'un même « objet »

Lever des exceptions

- Certaines fonctions peuvent lever (générer) des exceptions :

```
>>> math.log(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

- > *Comment une fonction définie par l'utilisateur peut-elle lever des exceptions également définies par l'utilisateur ?*
- Exemple : (dans la fonction qui retourne le mois) si le numéro du mois est inférieur à 1 ou supérieur à 12 la fonction doit lever une exception avec le message « month number out of range »

Lever des exceptions

- Le mot-clé **raise** génère une exception et **stoppe le flux d'exécution** du programme si vous laissez Python la gérer :

```
def mois(n):  
    if (n < 1) or (n > 12):  
        raise Exception('month number out of range')  
    if n == 1:  
        return('janvier')  
    if n == 2:  
        return('février')  
    ...
```

- Mais il est également possible de la gérer avec un **try...except** dans le code qui appelle cette fonction (cf. diaporama #3)

Fonctions récursives

- Une fonction récursive est **une fonction qui s'utilise elle-même** pour produire un résultat (effet de bord ou valeur retournée) :

```
def facto(n):  
    return n * facto(n-1)
```

```
>>> print(facto(3))
```

```
Traceback (most recent call last):
```

```
File "/Users/lehuen/Desktop/test.py", line 4, in <module>
```

```
    print(facto(3))
```

```
    ...
```

```
RuntimeError: maximum recursion depth exceeded
```

Quel est le problème ???

Fonctions récursives

- Une fonction récursive est **une fonction qui s'utilise elle-même** pour produire un résultat (effet de bord ou valeur retournée) :

```
def facto(n):  
    if n == 0:  
        return 1  
    else:  
        return n * facto(n-1)
```

```
>>> print(facto(3))  
6
```

- Toujours identifier un cas (ou plusieurs) **sans appel récursif** sinon on obtient une récursion « infinie » qui génère une erreur ou un plantage

Fonctions ou pas fonctions ?

- Organisez votre code sous forme de « paragraphes » en capturant une pensée complète puis « nommez la »
- Si votre code fonctionne quelque part, ne le répétez pas => réutilisez le (transformez-le en fonction)
- Si votre code devient trop long ou trop complexe, séparez-le en parties logiques et faites des fonctions avec ces morceaux
- Construisez une bibliothèque des opérations que vous répétez continuellement, partagez-la avec vos ami(e)s...



Remerciements / Contributions



Thes slides are Copyright 2010- Charles R. Severance (www.dr-chuck.com) of the University of Michigan School of Information and open.umich.edu and made available under a Creative Commons Attribution 4.0 License. Please maintain this last slide in all copies of the document to comply with the attribution requirements of the license. If you make a change, feel free to add your name and organization to the list of contributors on this page as you republish the materials.

Initial Development: Charles Severance, University of Michigan School Of Information

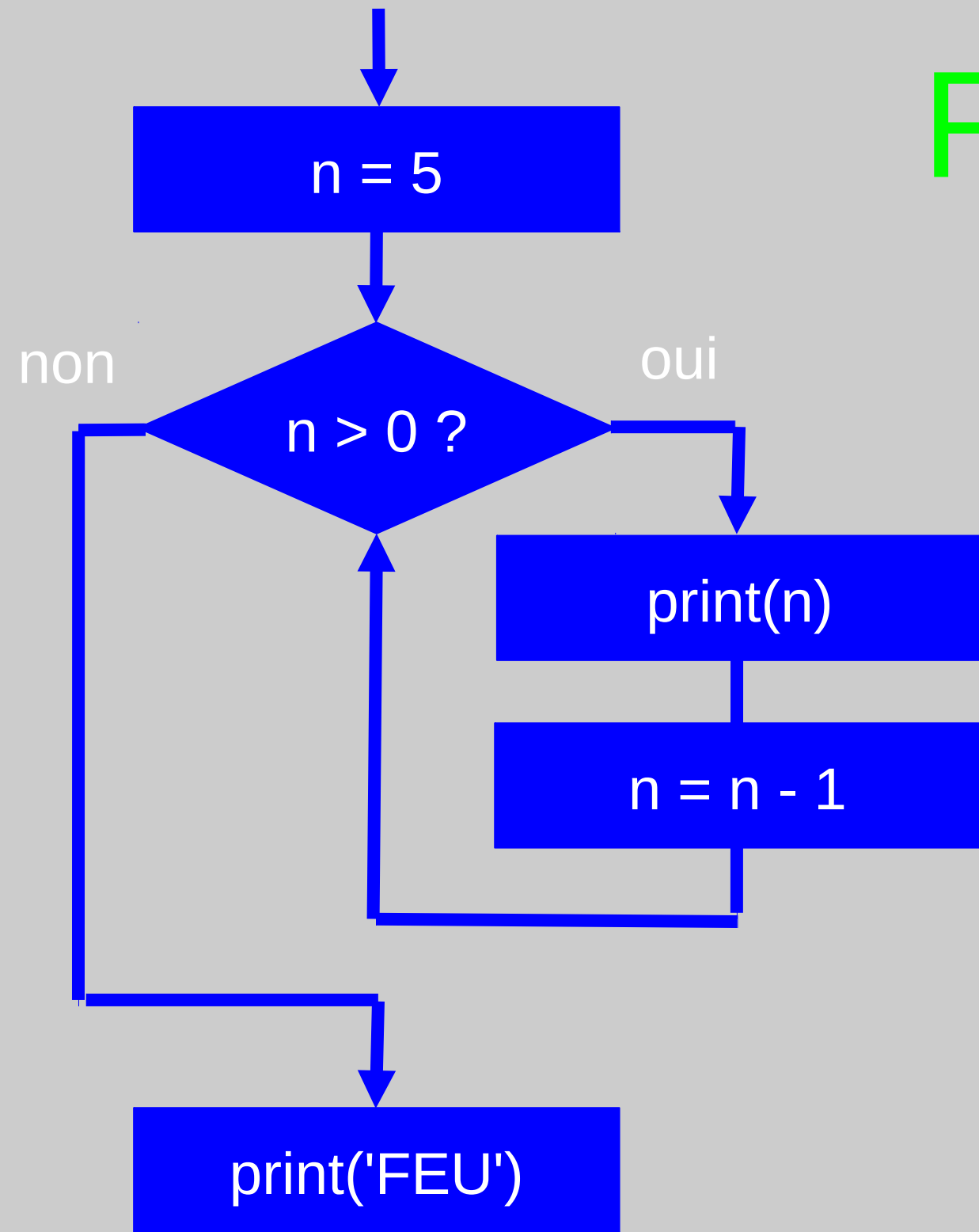
Translation : Frederic Foiry
Revision : Stephanie Kamidian
Adaptation : Jérôme Lehuen

Chapitre 5

Structures Itératives

Les types liste et chaîne

Répétition d'opérations



Programme :

```
n = 5
while n > 0 :
```

```
    print(n)
    n = n - 1
```

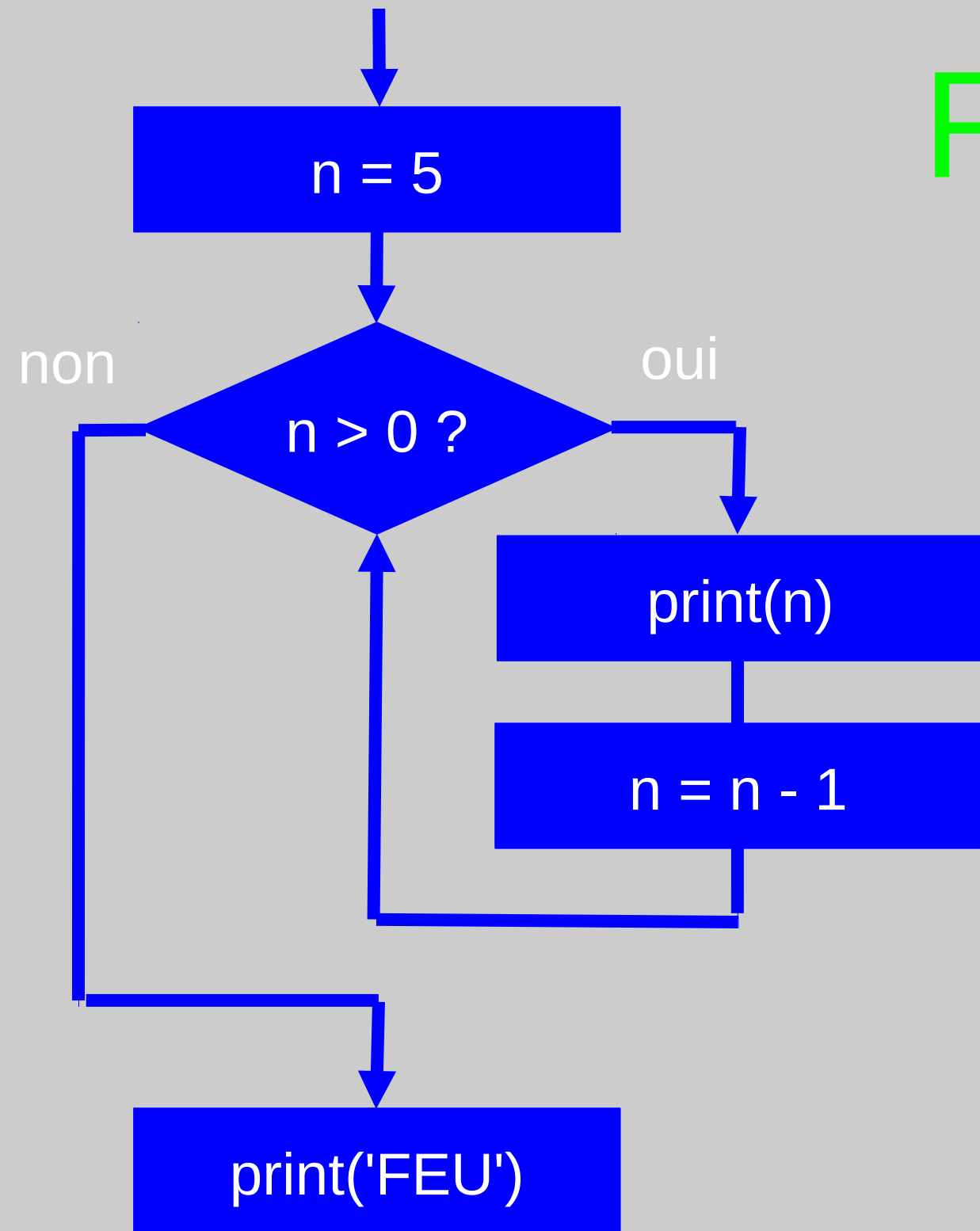
```
print('FEU')
```

Affichage :

5
4
3
2
1
FEU

Que fait ce programme ?

Répétition d'opérations



Programme :

```
n = 5
while n > 0 :
```

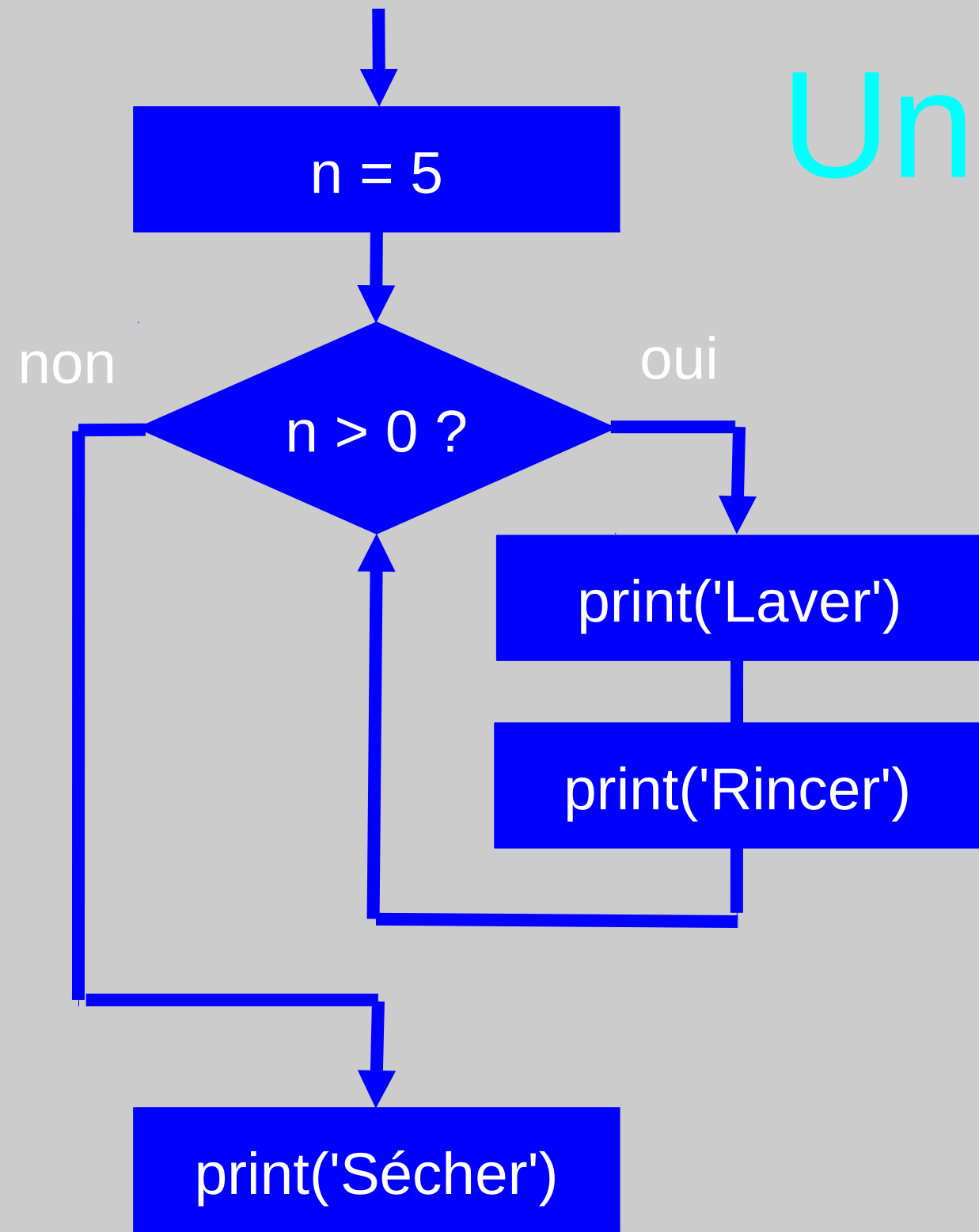
```
    print(n)
    n = n - 1
```

```
print('FEU')
```

Condition de
la boucle (vraie / fausse)

Corps de
la boucle (un bloc)

Une boucle infinie



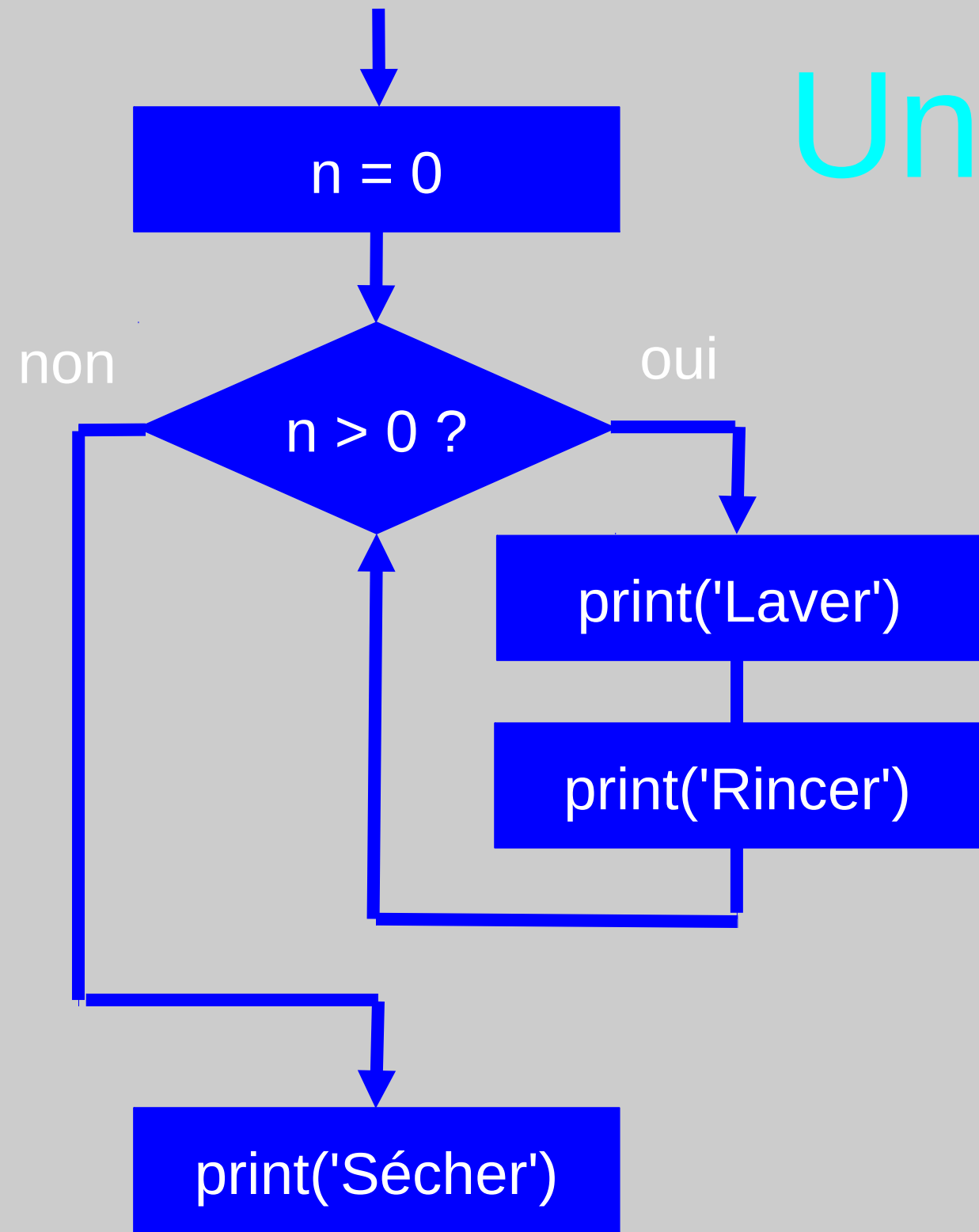
```
n = 5
while n > 0 :
    print('Laver')
    print('Rincer')
print('Sécher')
```



*La condition de la boucle est toujours vraie
=> pas de sortie possible de la boucle*

Quel est le problème de ce programme ?

Une boucle inaccessible



```
n = 0
while n > 0 :
    print('Laver')
    print('Rincer')
print('Sécher')
```



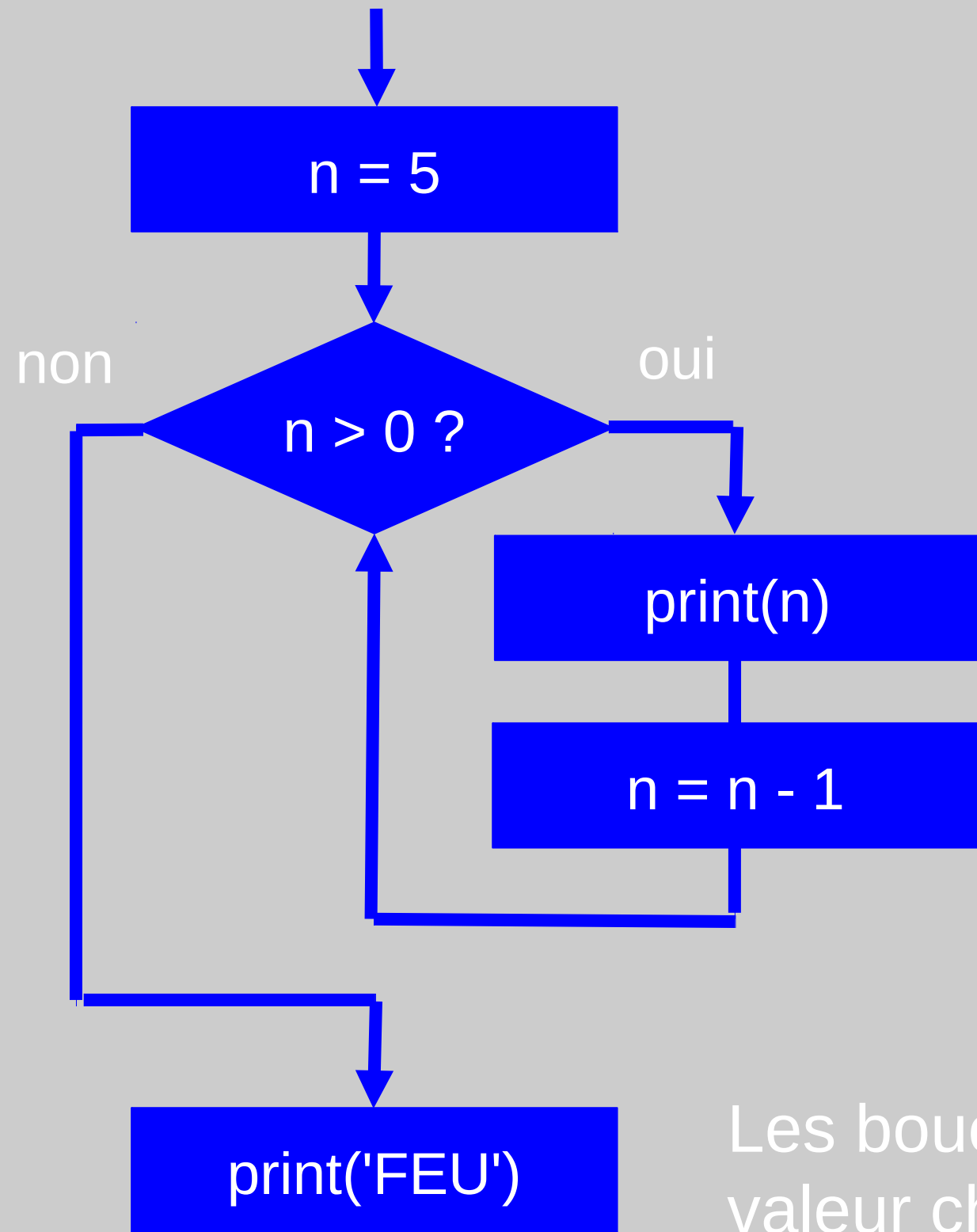
*La condition de la boucle est fausse dès le début
=> pas d'entrée possible dans la boucle*

Quel est le problème de ce programme ?

La boucle while

- Une **boucle while** permet de répéter un **bloc d'instructions** en fonction de la valeur d'une **expression booléenne** (condition) :
 - > Si la condition est fausse dès le début, le bloc est « sauté »
 - > Sinon, le bloc est répété tant que la condition reste vraie
- Il n'est pas toujours facile d'être certain qu'une boucle while se terminera mais il est facile de vérifier les points suivants :
 - > La condition de la boucle doit pouvoir **changer de valeur**
 - > La modification est effectuée dans le corps de la boucle

Variable d'itération



Programme :

```
n = 5
while n > 0 :
    print(n)
    n = n - 1
```

```
print('FEU')
```

Affichage :

5
4
3
2
1
FEU

Les boucles while comportent souvent des **variables** dont la valeur change à chaque **itération** (passage dans la boucle). Ces **variables d'itération** sont souvent numériques : leurs valeurs successives définissent une séquence de nombres.

Le type Liste

- La liste est un **type structuré** = une **structure de données**
- C'est une **structure linéaire ordonnée** composée de valeurs quelconques (entiers, flottants, chaînes, etc.) éventuellement **hétérogène**

- En Python, les éléments d'une liste sont séparés par des virgules, l'ensemble étant enfermé par des crochets :

```
>>> lst = [1, 2, 3, 'hello', 3.14]
print(lst)
[1, 2, 3, 'hello', 3.14]
>>> type(lst)
<class 'list'>
>>> print(len(lst))
5
```

Accéder à un élément

- On accède **directement** à un élément d'une liste en utilisant son **index** avec la syntaxe suivante :

*L'index du
premier
élément est 0*

*Impossible
d'accéder en
dehors de la liste*

```
>>> lst = [1, 2, 3, 'hello', 3.14]
```

```
>>> print(lst[0])
```

```
1
```

```
>>> print(lst[3])
```

```
hello
```

```
>>> print(lst[5])
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: list index out of range
```

Modifier un élément

- Pour modifier un élément on utilise une **instruction d'affectation** comme pour une variable :

```
>>> lst = [1,2,3,'hello',3.14]
```

```
>>> lst[3] = 'bonjour'
```

```
>>> print(lst)
```

```
[1,2,3,'bonjour',3.14]
```

```
>>> lst[5] = 9999
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
IndexError: list assignment index out  
of range
```

*Impossible de
modifier un
élément inexistant*

Supprimer un élément

- Pour supprimer un élément on applique une **instruction del** sur cet élément (attention : pas de parenthèses) :

```
>>> lst = [1,2,3, 'hello', 3.14]
```

```
>>> del lst[4]
```

```
>>> print(lst)
```

```
[1,2,3, 'hello']
```

```
>>> del lst[4]
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
IndexError: list assignment index out  
of range
```

*Impossible de
supprimer un
élément inexistant*

Ajouter un élément

- Pour ajouter un élément on invoque la **méthode append** sur cet élément (attention à la syntaxe particulière) :

*Attention à
la syntaxe*

```
>>> lst = [1, 2, 3, 'hello', 3.14]
>>> lst.append(9999)
>>> print(lst)
[1, 2, 3, 'hello', 3.14, 9999]
```

- Cette syntaxe correspond à l'utilisation d'une **méthode** en **programmation objet** (les listes sont des **objets** à part entière)

Les listes de listes

- Une liste pouvant contenir des valeurs de n'importe quel type, elle peut également contenir des listes :

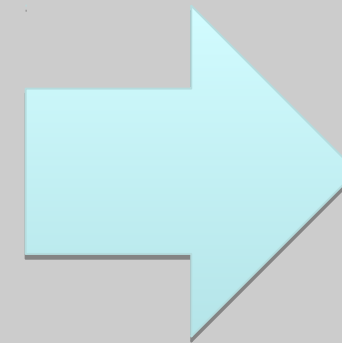
```
>>> lst = [[1,2,3],[4,5,6]]
>>> print(lst[0])
[1,2,3]
>>> print(lst[0][1])
2
>>> lst.append([7,8,9])
>>> print(lst)
[[1,2,3],[4,5,6],[7,8,9]]
```

- On dispose ainsi d'une **structure de données à 2 entrées**

La boucle for

- Une **boucle for** permet de parcourir une liste et exécuter un **bloc d'instructions** pour chacun des éléments de la liste :

```
liste = [1,2,3,'hello',3.14]  
for element in liste:  
    print('bonjour', element)
```

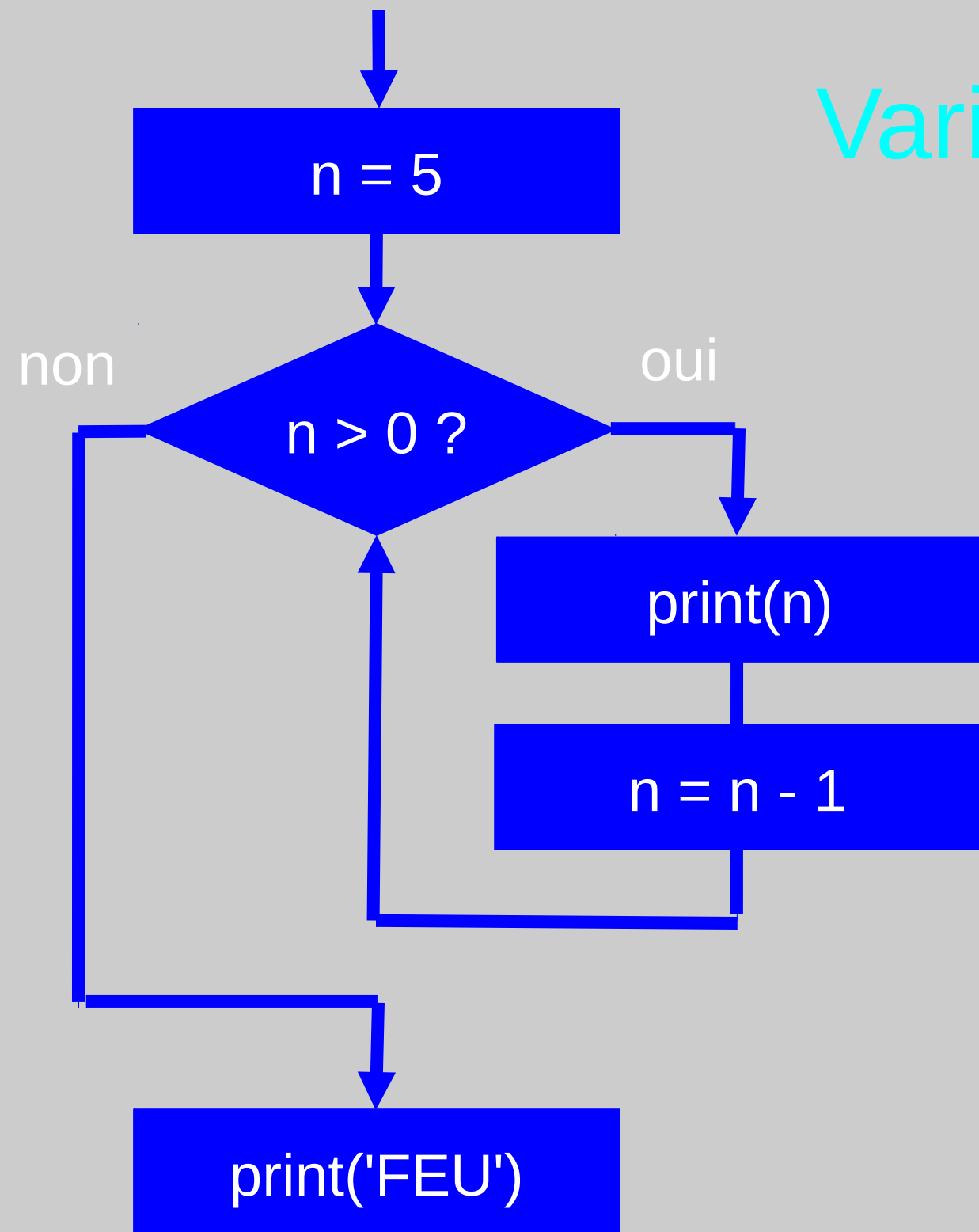


```
bonjour 1  
bonjour 2  
bonjour 3  
bonjour hello  
bonjour 3.14
```

- > Si la liste est vide, le bloc est « sauté »
- > Sinon, le bloc est répété autant de fois qu'il y a d'éléments

- *Contrairement à la boucle while, la boucle for s'arrête à coup sûr !*

Variable d'itération d'une boucle while



Programme :

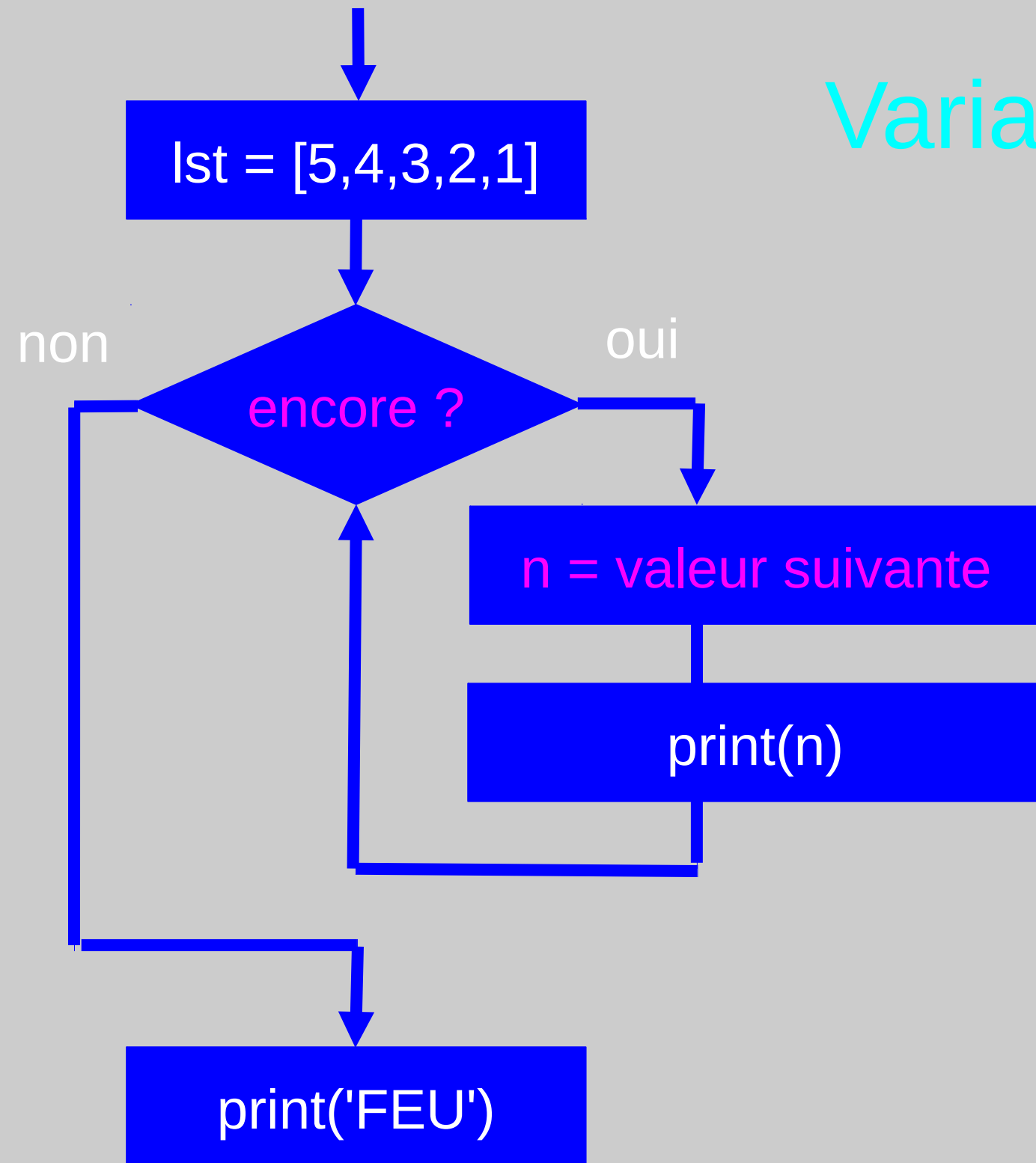
```
n = 5
while n > 0 :
    print(n)
    n = n - 1

print('FEU')
```

Affichage :

5
4
3
2
1
FEU

Variable d'itération d'une boucle for



Programme :

```
lst = [5,4,3,2,1]
for n in lst :
    print(n)
```

```
print('FEU')
```

Affichage :

5
4
3
2
1

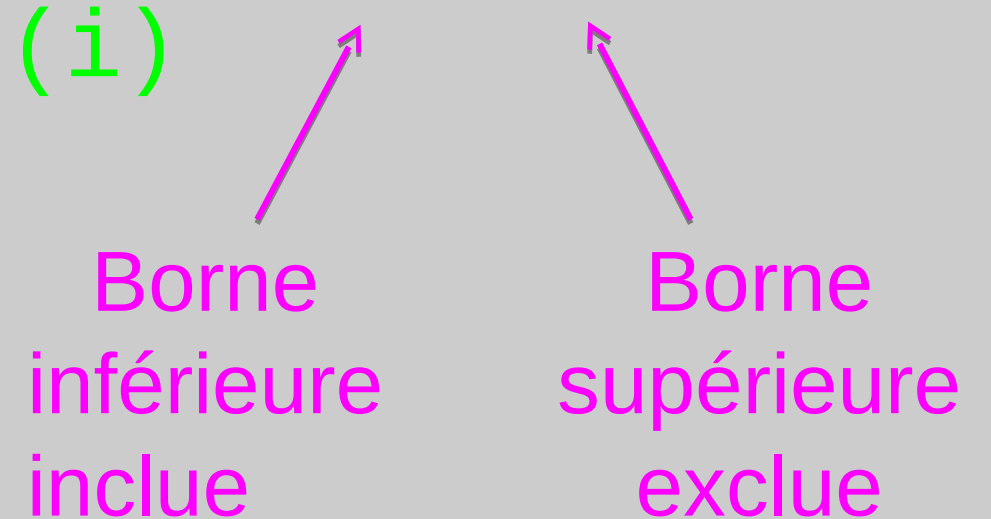
FEU

Pour i allant de 1 à 1000 ?

- La façon la plus naturelle pour réaliser une telle itération est d'utiliser une **boucle while** avec une variable d'itération...
- Mais il est également possible d'utiliser une **boucle for** sur une liste de 1000 entiers => *comment initialiser cette liste ?*

```
i = 1
while i < 1001:
    print(i)
    i = i + 1
```

```
for i in range(1, 1001):
    print(i)
```



Borne inférieure incluse

Borne supérieure exclue

Boucle while ou boucle for ?

- Lorsque nous connaissons à l'avance le nombre d'éléments sur lesquels on itère => **boucle for**
- Lorsque nous ne connaissons pas à l'avance le nombre d'éléments sur lesquels on itère => **boucle while**
- Ce qu'on peut faire avec une boucle for, on peut le faire avec une boucle while mais le résultat est un peu plus « verbeux »

Qu'est-ce qu'une chaîne ?

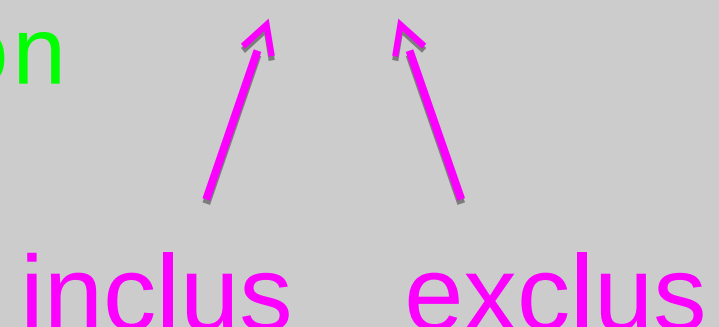
- C'est un type de données **non modifiables** qui permet de représenter des séquences de caractères Unicode
 - > Une donnée non modifiable **ne peut plus être modifiée** une fois créée
 - > Toute transformation d'une chaîne **produit une nouvelle valeur** distincte
 - On peut voir une chaîne comme une liste de caractères
 - > On traite un chaîne en utilisant des **fonctions** communes aux séquences
 - > On peut également utiliser des **méthodes** spécifiques aux chaînes
- ```
>>> len('Python 3.5') 10
>>> 'Python 3.5'.upper() 'PYTHON 3.5'
```



# Opérations sur les chaînes

- Pour indexer une chaîne, on utilise la notation `[]` où l'index commence à zéro, comme avec les listes
- On peut également extraire des sous-chaînes (ou « *tranches* ») :

```
>>> ch = 'Python 3.5'
>>> print(ch[0], ch[3], ch[5])
P h n
>>> ch[2:6]
thon
```



inclus    exclus

```
>>> ch[:6]
Python
>>> ch[2:]
thon 3.5
```

# Opérations sur les chaînes

- Exemples de **méthodes** de test de l'état d'une chaîne :

```
>>> ch.isupper()
>>> ch.isalpha()
>>> ch.startswith('Py')
>>> ch.endswith('thon')
```

- Exemples de **méthodes** retournant une nouvelle chaîne :

```
>>> ch.lower()
>>> ch.upper()
>>> ch.capitalize()
>>> ch.strip()
```

# Opérations sur les chaînes

- Quelques autres méthodes en vrac :

```
>>> ch.find('thon')
>>> ch.replace('thon', 'tule')
>>> ch.split()
>>> '-'.join(['Python', '3.5'])
```

- Et parce que les chaînes sont avant tout des séquences :

```
for c in ch:
 print(c)
```

*Fin du cours de Python  
du premier semestre...*