# Option complémentaire informatique : programmation

***Version 0.1***

**Cédric Donner**

02 October 2013

Table des matières

Cette partie du cours d'OC informatique a essentiellement pour but d'apprendre à programmer un ordinateur à l'aide du langage Python, ce qui nécessite d'apprendre à penser et formuler des problèmes de manière compréhensibles pour une machine.

---

**Astuce :** Si vous rencontrez des erreurs d'orthographe ou d'autres problèmes sur ce site, merci de bien vouloir les signaler sur le dépôt github du cours : https ://github.com/donnerc/oci-programming/issues

---

**Version PDF de la Théorie**

**Téléchargement** : `theorie.pdf`

---

CHAPITRE 1

# Théorie France IOI, niveau 01

## 1.1 Chapitre 1 – Affichage de texte, suite d'instructions

### 1.1.1 Afficher du texte

En Python, pour afficher du texte, on écrit print puis, entre parenthèses, le texte à afficher, entre guillemets. Ainsi, pour afficher le mot `bonjour`, on écrit :

```
print("bonjour")
```

Une ligne d'un programme est appelée une **instruction**.

Un texte apparaissant dans un programme entre des guillemets s'appelle une **chaîne de caractères**.

### 1.1.2 Afficher du texte : erreurs possibles

Lorsqu'on écrit un programme informatique il faut être très rigoureux car sinon l'ordinateur ne comprend pas ce qu'on lui demande et affiche des erreurs.

Face à une erreur, il faut d'abord lire et comprendre le message d'erreur et ensuite relire attentivement la zone de code concernée, en particulier pour s'assurer qu'il ne manque pas de symbole tel qu'une parenthèse ou un guillemet.

Voici quelques exemples de messages d'erreurs.

#### Attention aux parenthèses !

Si on oublie les parenthèses, cela produit une des erreurs suivantes :

```
>>> print("bonjour"
SyntaxError: unexpected EOF while parsing

>>> print "bonjour")
SyntaxError: invalid syntax
```

#### Attention aux guillemets !

Si on oublie un ou plusieurs guillemets, cela produit une des erreurs suivantes :

```
>>> print("bonjour)
SyntaxError: EOL while scanning string literal

>>> print(bonjour)
NameError: name 'bonjour' is not defined

>>> print(bonjour tout le monde)
SyntaxError: invalid syntax
```

#### Conclusion

Il faut juste être précis et ne rien oublier. Si l'on rencontre une erreur on pensera à bien tout vérifier. Les messages d'erreur vous indiquent très souvent la source exacte du problème, il faut les lire et essayer de comprendre leur signification.

Les messages d'erreurs vous donneront aussi toujours le numéro de la ligne à laquelle l'erreur s'est produite. Ce numéro n'est pas toujours exact mais vous donne une bonne idée d'où chercher votre erreur.

### 1.1.3 Afficher plusieurs lignes de texte

Pour faire effectuer plusieurs choses à un programme, on place les différentes instructions les unes en dessous des autres. On peut ainsi afficher plusieurs lignes de texte.

Voici un exemple complet.

```
print("Bonjour !")
print("Comment vas-tu ?")
```

```
Bonjour !
Comment vas-tu ?
```

On peut représenter l'exécution de cet exemple par le diagramme suivant.

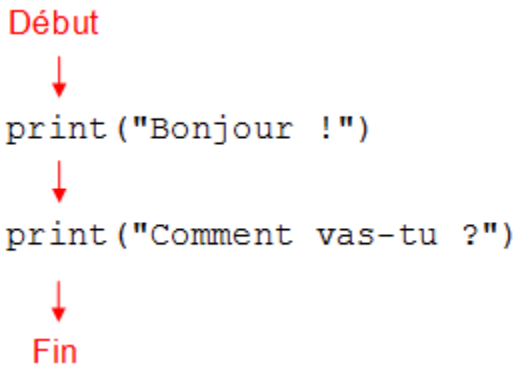


FIGURE 1.1 – Séquence de deux instructions

### 1.1.4 Afficher du texte sans retour à la ligne

L'instruction

```
print("Bonjour")
```

effectue en fait deux choses distinctes :
– premièrement, afficher le mot “Bonjour”,
– deuxièment, passer à la ligne suivante ou, autrement dit, effectuer un retour à la ligne.
Bien qu’il soit généralement pratique de faire en une seule instruction un affichage et un retour à la ligne, il peut également être utile de pouvoir effectuer chaque tâche indépendamment.

Pour afficher le mot “Bonjour” sans le suivre d’un retour à la ligne, on utilise la commande suivante :

```
print("Bonjour", end = "")
```

Le texte qui suit le “Bonjour” est une option qui permet de dire que l’on ne veut rien faire du tout après avoir affiché “Bonjour”.

Pour vérifier que le retour à la ligne n’est pas effectué, exécutons deux fois de suite l’instruction en question.

```
print("Bonjour", end = "")
print("Bonjour", end = "")
```

Ce programme affiche le mot “Bonjour” deux fois de suite sur la même ligne, sans aucun espacement entre les deux.

```
BonjourBonjour
```

Il nous reste à expliquer comment aller à la ligne sans rien afficher du tout. Pour cela, il suffit d’utiliser l’instruction print en lui disant d’afficher un texte vide :

```
print("")
```

Cette instruction demande donc d’afficher un texte vide puis de revenir à la ligne : elle n’affiche donc rien mais revient à la ligne.

Le programme suivant illustre l’utilisation de cette instruction :

```
print("Un ", end = "")
print("deux ", end = "")
print("trois ", end = "")
print("")
print("Soleil ! ")

Un deux trois
Soleil !
```

## 1.2 Chapitre 2 – Répétitions d’instructions

### 1.2.1 Répéter une action

Considérons une tâche répétitive, par exemple dire 5 fois “Bonjour !”. Il est tout à fait possible d’écrire le programme suivant :

```
print("Bonjour !")
print("Bonjour !")
print("Bonjour !")
print("Bonjour !")
print("Bonjour !")
```

Néanmoins, c'est très fastidieux ! Imaginez s'il avait fallu l'afficher 1000 fois !

Pour éviter de recopier plusieurs fois une instruction que l'on veut exécuter plusieurs fois, on peut utiliser le principe de la boucle. Ainsi, pour répéter 5 fois l'instruction qui affiche "Bonjour !" on va écrire le programme ci-dessous.

```
for loop in range(5):
    print("Bonjour !")
```

qui donne la sortie

```
Bonjour !
Bonjour !
Bonjour !
Bonjour !
Bonjour !
```

On peut représenter l'exécution du programme par le diagramme suivant :

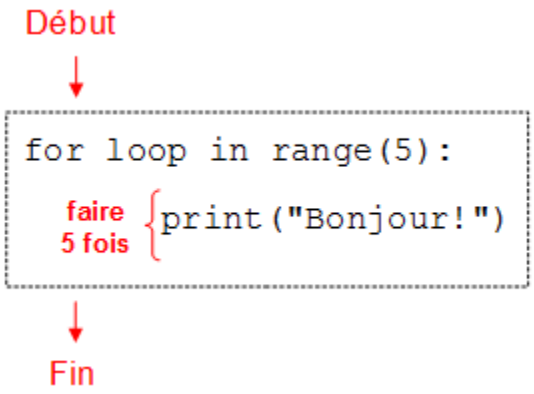


FIGURE 1.2 – Principe de la boucle 'répéter $n$ fois'

Pour réaliser ce programme, on a utilisé la structure suivante :

```
for loop in range(5):
    ...
```

qui signifie "répéter 5 fois".

On expliquera plus tard pourquoi une boucle s'écrit ainsi. En attendant on retiendra qu'une boucle s'écrit comme au-dessus, en remplaçant simplement le chiffre "5" par le nombre de répétitions souhaitées. On fera également attention à bien mettre les deux points à la fin.

On a donc, dans le programme ci-dessus, remplacé les "..." par l'instruction à répéter en prenant bien soin de la décaler de trois espaces vers la droite.

Le décalage de trois espaces vers la droite s'appelle une indentation. Cette indentation est obligatoire car elle sert à indiquer quelle(s) instruction(s) répéter dans la boucle.

### Remarque sur l'indentation

Avec l'éditeur utilisé sur le site, il vous suffit d'appuyer sur la touche "Tabulation" de votre clavier pour décaler un texte de 3 espaces, pas besoin d'appuyer trois fois sur la touche espace.

### 1.2.2 Répétition : erreurs possibles

Il est facile de se tromper dans les boucles lorsqu'on n'a pas l'habitude. Ainsi, si l'on oublie le " :" à la fin de la ligne, on obtient une erreur :

```
for loop in range(5)
   print("Bonjour !")

SyntaxError: invalid syntax
```

Et si l'on oublie d'indenter, c'est-à-dire si on oublie les trois espaces, on obtient également une erreur :

```
for loop in range(5):
print("Bonjour !")

SyntaxError: expected an indented block
```

Face à ce type d'erreur, on pensera donc à bien vérifier que les deux-points sont bien présents et que l'indentation a été faite.

### 1.2.3 Répéter plusieurs actions

Il est souvent utile de répéter un groupe de plusieurs instructions, et non pas seulement une seule instruction. Par exemple, supposons que l'on veuille afficher :

```
Bonjour !
Comment vas-tu ?
Bonjour !
Comment vas-tu ?
```

On peut écrire un programme qui affiche ce texte à l'aide d'une boucle qui répète deux instructions à chaque fois. Pour obtenir cela, il suffit d'indenter les deux instructions à répéter, comme montré ci-dessous :

```
for loop in range(2):
   print("Bonjour !")
   print("Comment vas-tu ?")

Bonjour !
Comment vas-tu ?
Bonjour !
Comment vas-tu ?
```

Si l'on indente la première instruction d'affichage mais pas la seconde, on obtient un programme différent où seule la première instruction est répétée deux fois, tandis que la seconde n'est exécutée qu'une seule fois :

```
for loop in range(2):
   print("Bonjour !")
print("Comment vas-tu ?")

Bonjour !
Bonjour !
Comment vas-tu ?
```

En conclusion, il est très important de bien faire attention à indenter toutes les instructions qui doivent être répétées, et uniquement celles-là.

### 1.2.4 Répétition : cohérence de l'indentation

Attention à toujours utiliser 3 espaces pour indenter le code. Si l'indentation n'est pas toujours la même, vous obtiendrez une erreur :

```
for loop in range(2):
   print("Bonjour !")
  print("Comment vas-tu ?")

SyntaxError: unindent does not match any outer indentation level
```

### 1.2.5 Répéter de manière imbriquée

Nous avons vu comment les boucles permettent de répéter une action donnée. Là où ça devient vraiment puissant, c'est qu'il est possible de répéter une action qui elle-même répète une action.

Par exemple, imaginons qu'on souhaite écrire un programme dessinant un rectangle rempli de X, haut de 5 lignes et large de 10 colonnes, c'est-à-dire :

```
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
```

Il nous faut donc un programme qui va répéter 5 fois les deux choses suivantes :

# répéter 10 fois l'affichage d'un caractère `X`, sans retour à la ligne, # passer à la ligne suivante. Il nous faut donc écrire une boucle dans une boucle !

On arrive donc au programme suivant :

```
for loop in range(5):
   for loop in range(10):
      print("X", end = "")
   print("")
```

Étudions de plus près le code du programme. Il débute par une instruction de répétition :

```
for loop in range(5):
```

qui va répéter tout le bloc de code suivant :

```
for loop in range(10):
   print("X", end = "")
print("")
```

Ce bloc contient lui-même une instruction de répétition :

```
for loop in range(10):
```

Et cette instruction s'applique uniquement à l'instruction qui affiche un `X` :

```
print("X", end = "")
```

On peut représenter l'exécution du programme par le diagramme suivant :

Lorsqu'une boucle apparaît à l'intérieur d'une autre boucle, comme c'est le cas ici, on parle de boucle imbriquée.

## 1.3 Chapitre 3 – calculs et variables

### 1.3.1 Mémoriser des informations

#### Principe

Supposons que l'on souhaite écrire un programme qui affiche d'abord la distance qui séparait la Terre et Mars le 27 Août 2003 (55'758'000 km), puis la distance à parcourir par la lumière pour faire l'aller-retour depuis la Terre (le double). On pourrait écrire le programme ainsi :

```
print(55758000)
print(2 * 55758000)
```

```
55758000
111516000
```

Ce programme fonctionne parfaitement bien, mais il n'est pas idéal car on a dû écrire deux fois le nombre `55758000` dans le code du programme. Cela signifie que si on veut afficher ces informations pour un autre jour, pendant lequel la distance entre les deux planètes est différente, il faudra modifier le programme à deux endroits différents.

Pour éviter de devoir faire des modifications en double, on va utiliser une **variable**, appelée `distance`, pour stocker la valeur `55758000`. Grâce à cette variable, on peut réécrire le programme précédent en ne faisant apparaître qu'une seule fois le nombre `55758000` :

```
distance = 55758000
print(distance)
print(2 * distance)
```

```
55758000
111516000
```

Le programme fonctionne exactement comme avant. Peu importe qu'il soit un peu plus long, ce qui compte vraiment, c'est qu'on a maintenant deux fois moins de chance de se tromper lorsqu'on modifie la distance.

On peut voir une variable comme une boîte qui porte un nom et qui contient quelque chose : on peut y stocker des informations et les retrouver plus tard. Dans notre exemple, la boîte porte le nom `distance` et contient la valeur `55758000`.

Une boîte contenant un nombre

La boîte `distance` a été créée par l'instruction suivante, qui s'appelle une **affectation** :

```
distance = 55758000
```

Les deux instructions d'affichage ont consulté le contenu de la boîte distance pour y lire la valeur `55758000`.

```
print(distance)
print(2 * distance)
```

### 1.3.2 Règles pour choisir les noms des variables

Lorsqu'on a besoin d'une nouvelle variable, on cherchera toujours à trouver un nom pour cette variable qui décrive le mieux possible ce que représente la valeur contenue dans cette variable. Il ne faut pas hésiter à mettre à la suite plusieurs mots pour construire un nom précis. Nous vous conseillons de nommer vos variables par une suite de mots ou abréviations accolés les uns aux autres, en mettant en majuscule la première lettre de chaque mot (sauf le premier). Voici quelques exemples :

```
longueurFeuille
nbPiecesJaunes
maxHauteurPiquets
```

En particulier, on évitera l'utilisation de noms réduit à une simple lettre, comme `a` ou `a`. Par exemple, le programme suivant est un exemple de très mauvais code :

```
a = 1000
b = 50
c = a + b
```

En effet, on comprend beaucoup mieux ce qui se passe lorsque des noms plus parlants sont employés :

```
prixFour = 1000
prixBatteur = 50
prixTotal = prixFour + prixBatteur
```

### Noms valides et invalides

En Python, le nom d'une variable peut être choisi "presque" librement. Il faut retenir les règles suivantes :
– Le premier caractère du nom d'une variable ne peut pas être un chiffre. Le nom `1erNombre` est ainsi invalide.
– Certains mots font partie du langage Python et ne peuvent être utilisés pour nommer des variables. C'est par exemple le cas du mot `for` adsf asdf asdf
– Les caractères autorisés sont essentiellement les chiffres de `0` à `9`, le caractère _, et les lettres majuscules ou minuscules.

**Mots-clés de Python**

Voici une liste exhaustive des mots-clés de Python qu'il est interdit d'utiliser pour les noms de variables :

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

**Référence** : http ://docs.python.org/3/reference/lexical_analysis.html#identifiers

### Plusieurs variables

Dans l'exemple ci-dessus, nous avons utilisé une seule variable, mais les programmes en utilisent généralement plusieurs. Le programme suivant illustre cela : il utilise deux variables nommées `largeur` et `longueur` afin de calculer l'aire, en [$mm^2$] d'une feuille A4 (21cm x 29,7cm), et il enregistre le résultat dans une variable nommée surface. Le contenu de cette variable est ensuite affiché.

```
largeur = 210
longueur = 297

surface = longueur * largeur
print(surface)
```

```
62370
```

### 1.3.3 Erreurs fréquentes

#### Débogage : variables inexistantes

Si on utilise une variable qui n'existe pas encore, on obtient une erreur. Par exemple, le programme suivant définit une variable longueur, et tente ensuite d'afficher le contenu d'une variable nommée largeur qui n'a jamais été définie.

```
longueur = 297
print(largeur)
```

```
NameError: name 'largeur' is not defined
```

Il faut faire particulièrement attention au fait que les minuscules et majuscules ne sont pas considérées comme équivalentes. Ainsi, la variable nommée longueur n'a strictement rien à voir avec la variable nommée Longueur.

```
longueur = 10
print(Longueur)
```

```
NameError: name 'Longueur' is not defined
```

Si on rencontre une erreur de la forme `NameError: name 'xxxxx' is not defined`, on pensera à bien vérifier que l'on n'a pas fait de faute de frappe dans les noms de variables que l'on a utilisés dans le programme.

#### Modifications d'une variable

Comme son nom l'indique, une variable a vocation à varier, c'est-à-dire à stocker différentes valeurs au cours du temps. Pour illustrer cette possibilité, considérons un programme qui utilise une variable nommée taille pour représenter la taille d'une plante qui a pour taille initiale 180 cm et qui grandit ensuite de 20 cm. Ce programme, dont le code apparaît ci-dessous, affiche d'abord la taille initiale de la plante, puis sa taille finale (200 cm).

Le programme

```
taille = 180
print(taille)
taille = 200
print(taille)
```

produit la sortie

```
180
200
```

Encore une fois, on a un programme qui fonctionne correctement, mais qui n'est pas idéal. En effet, si l'on veut modifier la taille initiale de la plante, on est obligé de modifier deux valeurs dans le code programme.

Pour éviter ce problème, on va modifier le code du programme pour calculer la taille finale en ajoutant 20 cm à la taille initiale. L'instruction ci-dessous permet de modifier le contenu de la variable taille en le remplaçant par son contenu actuel augmenté de 20.

```
taille = taille + 20
```

Observez que le symbole d'égalité n'a pas du tout la même signification qu'en mathématiques. En mathématiques, l'égalité énonce un fait. Ainsi $x = y + z$ signifie qu'il est vrai maintenant et pour toujours que x a la même valeur que y + z. Au contraire, dans une affectation telle que `taille = taille + 20`, on décrit une action, en indiquant qu'il faut enregistrer dans la variable dont le nom est écrit à gauche du signe égal le résultat du calcul écrit à droite du signe =.

En utilisant l'affectation `taille = taille + 20`, on peut réécrire notre programme en n'utilisant que les nombres `180` et `20` (une seule fois chacun).

: : taille = 180 print(taille) taille = taille + 20 print(taille)

Sortie :

```
180
200
```

L'exécution de ce programme est détaillée ci-dessous, la ligne rouge indiquant à quel endroit du programme on est arrivé :

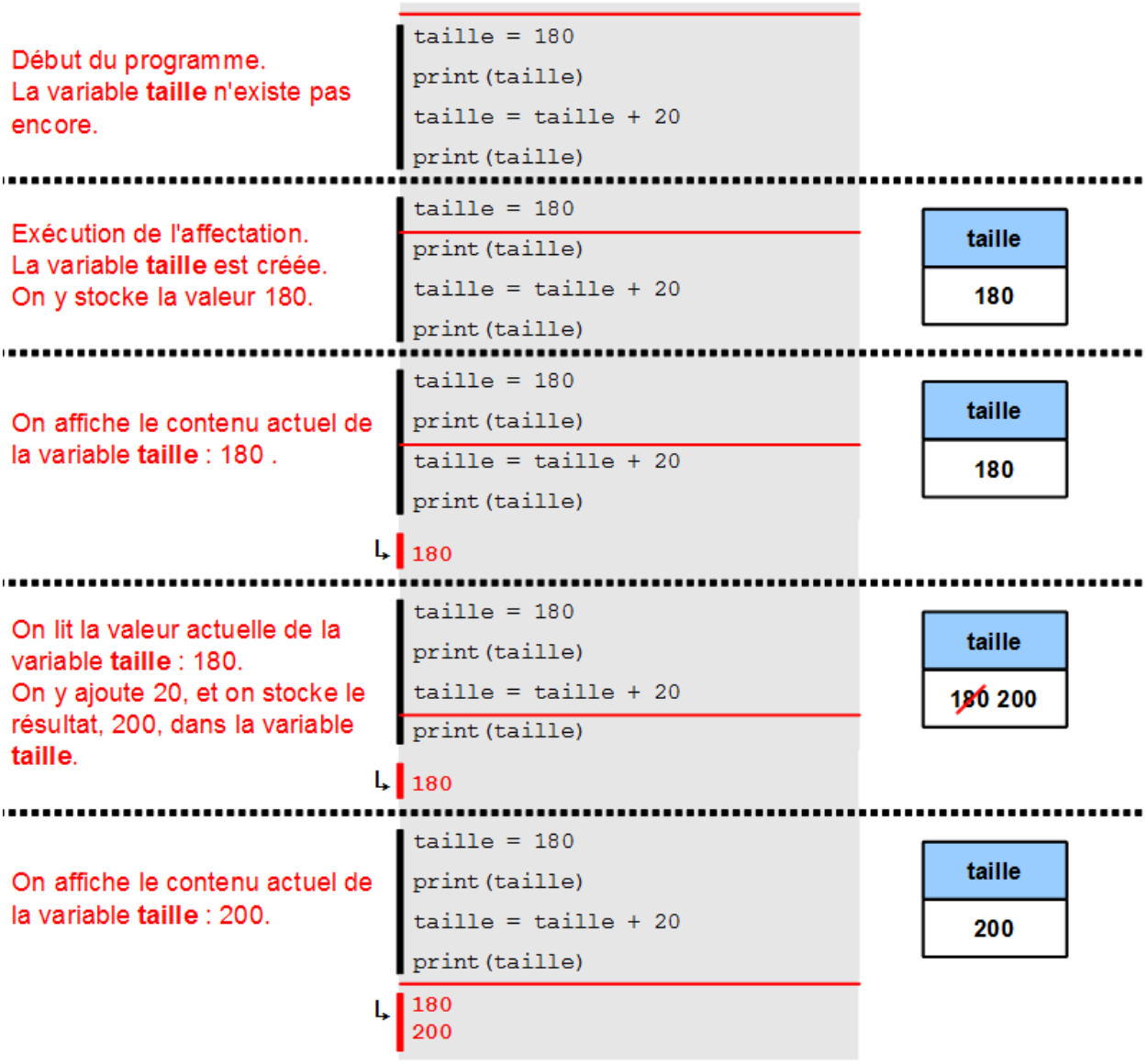


FIGURE 1.3 – Exécution du programme ligne après ligne

#### Modification d'une variable mal nommée

Comme vous le savez, les noms de variables font la distinction entre majuscules et minuscules. Cependant, quand on se trompe, on n'obtient pas forcément une erreur de la forme `"truc is not defined"`. Parfois, le programme peut faire tout simplement autre chose que ce que l'on voudrait !

Par exemple, le programme suivant contient une petite erreur à la troisième ligne, car on a mis `Taille` au lieu de `taille`.

```
taille = 180
print(taille)
Taille = taille + 20
print(taille)
```

Au lieu d'afficher `180` puis `200`, comme on voudrait, le programme modifié affiche deux fois de suite le nombre `180`.

```
180
180
```

Aucun message d'erreur ne nous prévient que l'on s'est trompé ! Pourtant, il est très facile de se tromper de la sorte, en particulier parce qu'on a l'habitude de commencer les lignes par des majuscules.

---

**Astuce :** Pour éviter ce genre de problème, on adoptera la convention de toujours faire commencer les noms de variables par une lettre minuscule.

---

## 1.4 Chapitre 4 – lecture de l'entrée

### 1.4.1 Introduction

#### Lire des entiers

On souhaite écrire un programme demandant deux entiers à l'utilisateur, la longueur et la largeur d'un rectangle, et qui affiche l'aire du rectangle. En Python on fera ainsi :

```
longueur = int(input())
largeur = int(input())

print(longueur * largeur)
```

#### Analyse du programme

La ligne

```
longueur = int(input())
```

est celle qui permet de récupérer l'entier donné par l'utilisateur (qui correspond à la longueur du rectangle) et de le stocker dans la variable longueur. Il en est de même avec la ligne suivante pour la largeur du rectangle.

Comme `int` signifie "entier" et `input` signifie ici récupérer, `int(input())` se lit "récupérer un entier". On demande à Python de nous donner la valeur du prochain entier qu'indiquera l'utilisateur.

#### Entrée et espaces

Lorsqu'on fournit les entrées au programme, il faut faire bien attention de donner un et un seul entier par ligne, sans espaces après les entiers.

### 1.4.2 Erreurs possibles

#### Erreur si l'on ne donne pas un entier

Imaginons que le programme demande un entier mais que l'utilisateur fournisse un texte, par exemple "coucou". Bien sûr, le programme produit une erreur, car "coucou" ne peut pas être interprété comme un nombre.

```
taille = int(input())
print(taille)
```

Si on fournit en entrée la chaine de caractères `"coucou"` à ce programme, il va se produire l'erreur suivante :

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

Notez qu'une erreur similaire peut se produire si vous ajoutez des espaces autour d'un nombre.

#### Lecture d'entiers : autres erreurs possibles

Si on oublie le `int(...)` et que l'on écrit juste `input()` à la place de `int(input())`, on peut avoir de grosses surprises, comme le montre l'exemple suivant.

```
valeur1 = input()
valeur2 = input()
print(valeur1 + valeur2)
```

---

**Entrée**

```
11
22
```

---

**Sortie**

```
1122
```

---

Si on oublie le `int(...)` autour du `input()`, les valeurs ne sont pas traitées comme des entiers mais comme du texte. Le symbole + agit alors comme un opérateur qui **concatène** (c'est-à-dire qui met bout à bout) deux textes, et du coup on obtient `11` collé à `22` (c'est-à-dire `1122`) à la place de `11` additionné à `22` (c'est-à-dire `33`).

Faîtes donc toujours attention à ne pas utiliser `input()` tout seul. De plus, si les résultats des calculs sont manifestement faux, pensez à vérifier si les nombres ne sont pas traités comme du texte.

#### Erreur si on lit trop de choses

Une erreur qui peut se produire est d'essayer de lire trop de données. Imaginons par exemple qu'il faille lire deux entiers et afficher leur produit mais qu'on se soit trompé dans le programme et qu'on lise trois entiers :

```
premierNombre = int(input())
secondNombre = int(input())
troisiemeNombre = int(input())
print(premierNombre * secondNombre * troisiemeNombre)
```

**Sortie**

```
4
5

Traceback (most recent call last):
  File "./run/exe", line 3, in
    troisiemeNombre = int(input())
EOFError: EOF when reading a line
```

On sait que le programme n'est pas bon car il essaie de lire un troisième nombre qui n'existe pas, mais que signifie exactement cette erreur ?

`EOF` est une abréviation pour **"End Of File"**, c'est-à-dire, si on traduit en français "Fin Du Fichier". **"EOFError : EOF when reading a line"** peut donc se traduire par "Erreur de Fin Du Fichier : fin du fichier atteinte alors qu'on essaie de lire une nouvelle ligne". Le "fichier" dont il est question ici est le fichier qui contient toutes les données que le programme va lire. Le message signifie donc qu'on a essayé de lire quelque chose (ici un entier) alors qu'on avait atteint la fin du fichier contenant les données à lire. Une erreur s'est donc produite.

On peut également remarquer que le message d'erreur nous indique exactement où cette erreur s'est produite, au moment d'exécuter la ligne

```
troisiemeNombre = int(input())"
```

En résumé, si on obtient une erreur avec un `EOF` c'est qu'on a essayé de lire trop de choses.

### 1.4.3 Portée d'une variable

En Python toute variable déclarée au sein d'un programme peut être lue ou modifiée depuis n'importe quel endroit du programme. Par exemple :

```
nbValeurs = int(input())
for loop in range(nbValeurs):
   derniereValeurLue = int(input())
print(derniereValeurLue)
```

**Entrée**

```
2
10
25
```

**Sortie**

```
25
```

Ainsi, la variable `dernieValeurLue`, déclarée au sein de la boucle, peut être affichée après la boucle. Cela peut vous sembler naturel, mais beaucoup de langages de programmation ne fonctionnent pas de cette manière.

On appelle **portée** d'une variable l'ensemble des endroits du programme où elle peut être utilisée, c'est-à-dire où elle existe. En Python, la portée d'une variable est donc tout le programme. Notez cependant que cette règle ne marchera plus complétement si votre programme contient des **fonctions**. Nous verrons cela plus tard dans le chapitre sur les fonctions.

## 1.5 Chapitre 5 : Tests et conditions

### 1.5.1 Tester une condition : le "si"

La célèbre attraction du train fou est interdite aux moins de 10 ans. On souhaite écrire un programme qui demande à l'utilisateur son âge et qui, si la personne a moins de 10 ans, affiche le texte "Accès interdit". Dans cette phrase on a utilisé le mot "si", on va voir comment cela se traduit en Python :

```
age = int(input())
if age < 10:
   print("Accès interdit")
```

On écrit donc le mot-clef `if`, la traduction en anglais de "si", la condition à tester, à savoir `age < 10`, puis on termine la ligne avec un deux-points, comme on le faisait pour la boucle de répétition.

Ainsi, l'accès est interdit à un enfant de 8 ans :

**Entrée**

```
8
```

**Sortie**

```
Accès interdit
```

Au contraire, le programme n'affiche rien pour un âge de 13 ans.

**Entrée**

```
13
```

**Sortie**

On peut représenter l'exécution du programme par le diagramme suivant :

Pour exprimer la condition du `if` dans ce programme, on a utilisé le symbole <, qui est l'opérateur de comparaison strictement inférieur.

On a vu que l'opérateur < permet de tester si un nombre est strictement inférieur à un autre. De manière symétrique, l'opérateur > permet de tester si un nombre est strictement supérieur à un autre.

Lorsqu'on veut tester si un nombre est inférieur ou égal à un autre, on utilise le symbole <=. De manière symétrique, le symbole >= permet de tester si un nombre est supérieur ou égal à un autre.

Par exemple, le code suivant permet de tester si la température de l'eau a atteint 100 degrés.

```
temperature = int(input())
if temperature >= 100:
   print("L'eau bout !")
```

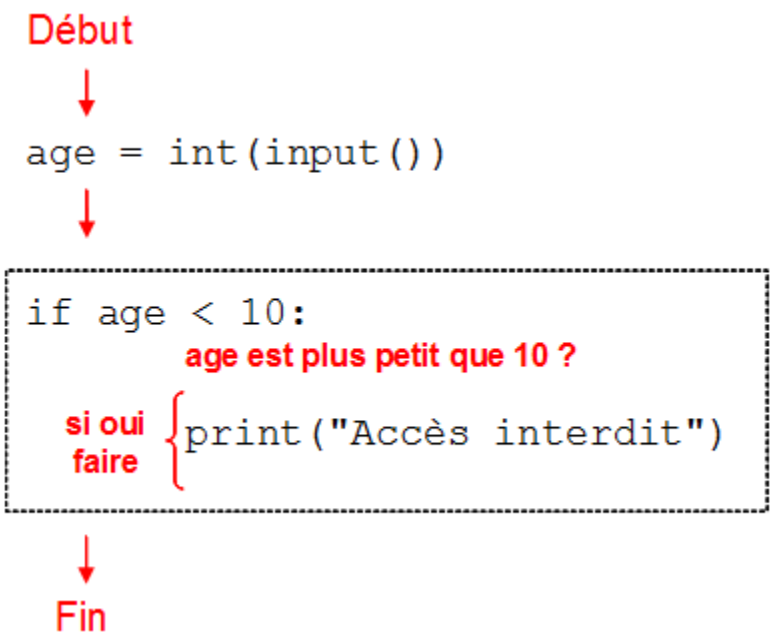


FIGURE 1.4 – Test d'une condition et exécution conditionnelle

### 1.5.2 Opérateurs de comparaison (résumé)

Voici un résumé des différents opérateurs de comparaison utilisables dans les conditions

TABLE 1.1 – Opérateurs de comparaison

| | |
|---|---|
| `a == b` | Vrai lorsque $a = b$ |
| `a != b` | Vrai lorsque $a \neq b$ |
| `a < b` | Vrai lorsque $a < b$ |
| `a > b` | Vrai lorsque $a > b$ |
| `a <= b` | Vrai lorsque $a \leq b$ |
| `a >= b` | Vrai lorsque $a \geq b$ |

### 1.5.3 Tester une condition : le "sinon"

Il arrive que l'on ait besoin de tester une condition puis son contraire. Par exemple, supposons que l'on souhaite tester si l'on peut ranger tous ses oeufs dans une seule boîte de 12. On pourrait programmer cela ainsi :

```
nbOeufs = int(input())
if nbOeufs <= 12:
   print("Une boîte suffit")
if nbOeufs > 12:
   print("Plusieurs boîtes nécessaires")
```

Ce programme a donc traduit la phrase suivante : si le nombre d'oeufs est plut petit que 12 afficher "Une boîte suffit" et si c'est strictement supérieur à 12 afficher "Plusieurs boîtes nécessaires".

Cependant, cette phrase s'exprime plus simplement sous la forme suivante : si le nombre d'oeufs est plus petit que 12, afficher "Une boîte suffit", sinon afficher "Plusieurs boîtes nécessaires".

Il est possible d'exprimer en Python la notion de "sinon", ce qui évite de tester le contraire de la condition que l'on vient de tester. Plus précisément, il suffit d'insérer le mot clé `else` suivi d'un deux-points à l'endroit où l'on veut dire "sinon". Ainsi, le code précédent peut s'écrire de manière beaucoup plus élégante.

```
nbOeufs = int(input())
if nbOeufs <= 12:
   print("Une boîte suffit")
else:
   print("Plusieurs boîtes nécessaires")
```

Il est important de toujours utiliser un `else` lorsque c'est possible. Cela permet non seulement d'éviter d'introduire des informations redondantes, comme par exemple `nbOeufs <= 12` et `nbOeufs > 12`, mais aussi de rendre la logique du programme plus facile à comprendre.

On peut représenter l'exécution du programme par le diagramme suivant :

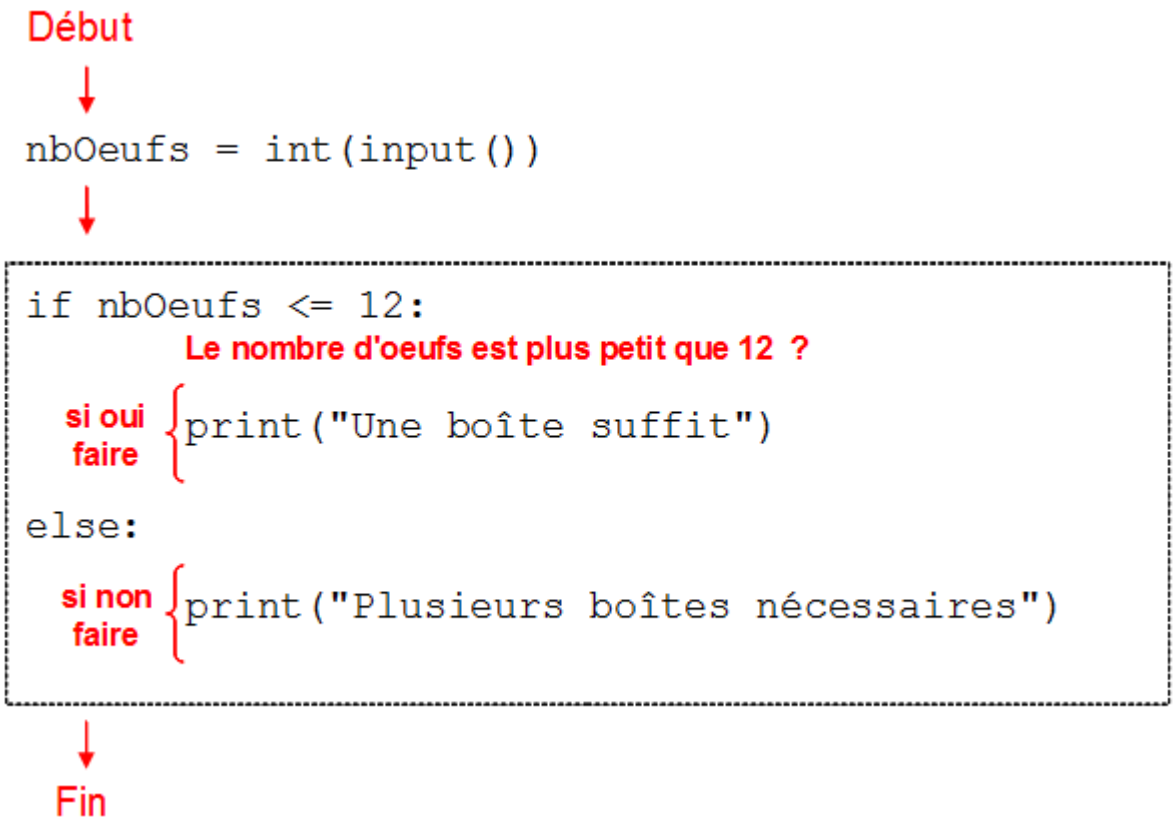


FIGURE 1.5 – Test d'une condition et exécution conditionnelle if/else

### 1.5.4 Blocs conditionels formés de plusieurs instructions

On peut bien sur placer plusieurs instructions à la suite d'un test. L'exemple suivant permet d'afficher deux messages à la suite dans le cas où la température de l'eau est supérieure ou égale à 100 degrés.

```
temperature = int(input())
if temperature >= 100:
   print("L'eau bout !")
   print("Préparons le thé")
```

Il est très important d'indenter l'instruction qui affiche "Préparons le thé". Si l'on n'indente pas la seconde instruction, c'est-à-dire si l'on écrit :

```
temperature = int(input())
if temperature >= 100:
   print("L'eau bout !")
print("Préparons le thé")
```

ualors la phrase "Préparons le thé" va s'afficher quelque soit la température de l'eau, ce qui n'est pas ce que l'on souhaite ici.

Notez que, de la même manière, il est possible de placer plusieurs instructions dans un bloc `else`.

### 1.5.5 Les opérateurs d'égalité et de différence

Vous avez vu comment tester des inégalités strictes, à l'aide des opérateurs < et >, ainsi que des inégalités larges, à l'aide des opérateurs <= et >=. Voyons maintenant comment tester l'égalité ou la non-égalité de deux valeurs.

Par exemple, pour tester si Marie et Robin ont le même âge, on utilise l'opérateur ==, comme illustré ci-dessous.

```
ageMarie = int(input())
ageRobin = int(input())
if ageMarie == ageRobin:
   print("Marie et Robin ont le même âge")
else:
   print("Marie et Robin n'ont pas le même âge")
```

---

**Note :** Il faut faire bien attention à ne pas confondre l'opérateur == avec le simple =, car les deux ont des rôles très différents :
– = sert à affecter une valeur à une variable,
– == sert à comparer deux valeurs dans un if.

---

Lorsqu'on veut uniquement tester si deux valeurs sont différentes, on utilise l'opérateur !=, qui se lit "différent de". Par exemple, le code suivant affiche un message si un animal n'a aucune chance d'être une araignée car il n'a pas 8 pattes.

```
nbPattes = int(input())
if nbPattes != 8:
   print("L'animal n'est pas une araignée")
```

## 1.6 Chapitre 6 : Structures avancées

### 1.6.1 Structures imbriquées

Vous avez déjà appris que, lorsqu'on place une boucle de répétition à l'intérieur d'une autre boucle de répétition, la boucle imbriquée (celle qui se situe à l'intérieur de l'autre) doit avoir tout son code indenté (décalé vers la droite). De la même manière, il faut indenter correctement le code lorsqu'un test (`if` ou `if/else`) est placé à l'intérieur d'une boucle de répétition ou d'un autre `if`, ou bien lorsqu'une boucle de répétition est placé à l'intérieur du corps d'un test.

Le programme suivant illustre cela dans le cas d'une boucle de répétition placé à l'intérieur d'un `if`. Ce programme lit un entier nommé cible. Si cible est un nombre positif, le programme affiche tous les entiers compris entre `1` et `cible` à l'aide d'une boucle de répétition. Sinon, le programme affiche le texte `"Rien à faire"`.

```
cible = int(input())
if cible >= 0:
   valeur = 1
   for loop in range(cible):
      print(valeur)
      valeur = valeur + 1
else:
   print("Rien à faire")
```

Observez la manière dont est indenté le code qui se trouve entre le `if` et le `else`.

## 1.7 Chapitre 7 : Conditions avancées, opérateurs booléens

### 1.7.1 Les opérateurs booléens : le “et”

La carte 12-25 n'est disponible que **si** vous avez moins de 25 ans **et si** vous avez plus de 12 ans. Écrivons un programme qui nous dise si on a droit à la carte ou pas :

```
age = int(input())
if age <= 25:
   if age >= 12:
      print("Carte possible")
   else:
      print("Carte impossible")
else:
   print("Carte impossible")
```

On voit que le programme n'est pas très efficace, car on a deux fois l'instruction

```
print("Carte impossible")
```

De plus, si on regarde la phrase ci-dessus on voit qu'on a utilisé le mot **“et”** pour dire qu'on voulait les deux conditions vraies en même temps. En Python, il est aussi possible de combiner des conditions :

```
age = int(input())
if (age <= 25) and (age >= 12):
   print("Carte possible")
else:
   print("Carte impossible")
```

On a donc utilisé l'opérateur `and`, la traduction en anglais du **“et”**, qui permet de combiner les deux conditions `(age <= 25)` et `(age >= 12)`.

Le programme est tout de suite plus court, plus clair, et ressemble plus à la phrase en langage naturel.

---

**Note :** Une condition est toujours soit vraie, soit fausse. Une valeur qui ne peut être que vraie ou fausse est appelée valeur *booléenne*. Cela donne donc son nom aux opérateurs *booléens* qui permettent de manipuler ces valeurs, de la même manière que les opérateurs numériques permettent de manipuler les nombres. L'opérateur and est donc un opérateur *booléen*.

---

Comme pour les opérateurs numériques (`+`, `-`, `/`, `*`), il existe un ordre de priorité entre les opérateurs numériques et booléens de Python mais nous ne rentrerons pas dans ces détails. Il est, dans tous les cas, préférable de faire comme ci-dessus : utiliser des parenthèses pour avoir un code bien clair, facilement compréhensible.

### 1.7.2 Les opérateurs booléens : le “ou”

Vous avez vu comment combiner deux conditions lorsqu'on veut que les deux soient vraies en même temps. On veut parfois en avoir au moins une des deux, c'est-à-dire soit l'une, soit l'autre, soit les deux. Par exemple on peut avoir une réduction si on a moins de 25 ans ou si on a plus de 60 ans. On a ici utilisé le mot **“ou”**, qui est une autre manière de combiner des conditions et qui se traduit en Python par l'opérateur booléen or :

```
age = int(input())
if (age <= 25) or (age >= 60):
   print("Réduction possible")
else:
   print("Pas de réduction")
```

Dans la réalité il faut en fait avoir entre 12 et 25 ans et non pas simplement moins de 25 ans. On peut donc combiner les conditions en utilisant à la fois un **“et”** et un **“ou”**, en n'oubliant pas de mettre les bonnes parenthèses :

```
age = int(input())
if ( (12 <= age) and (age <= 25) ) or (age >= 60):
   print("Réduction possible")
else:
   print("Pas de réduction")
```

On peut donc combiner facilement les opérateurs booléens pour construire des conditions complexes à partir de conditions simples.

### 1.7.3 Les booléens

En Python le programme suivant :

```
if prix < 10:
   print("Pas cher")
```

peut aussi s'écrire

```
estPasCher = (prix < 10)
if estPasCher:
   print("Pas cher")
```

On a donc stocké dans la variable `estPasCher` la valeur de la comparaison `prix < 10` pour la réutiliser plus tard dans un `if`.

La variable `estPasCher` est appelée une variable booléenne ou un booléen car elle ne peut être que vraie ou fausse, ce qui correspond en Python aux valeurs `True` (pour vrai) et `False` (pour faux).

Ainsi en Python, l'expression `(3 < 10)` vaut `True` et `(11 == 7)` vaut `False`. `True` et `False` sont donc des constantes du langage, au même titre que `0`, `1` ou encore `42`.

On peut donc affecter directement la valeur `True` ou `False` à une variable :

```
toujoursVraie = True
if toujoursVraie:
   print("La variable toujoursVraie vaut 'True'")
```

On peut également utiliser les opérateurs `and` et `or` comme le montre l'exemple suivant :

```
estSenior = (age >= 60)
estJeune = (age <= 25) and (age >= 12)
reductionPossible = (estSenior or estJeune)
if reductionPossible:
   print("Réduction!")
```

### 1.7.4 Les opérateurs booléens : la négation

Imaginons la situation suivante : vous avez le droit à une réduction si vous avez entre 12 et 25 ans ou si vous avez plus de 60 ans mais, si vous n'avez pas de réduction et que vous faites plus de 5 000km par an, alors vous avez le droit à un cadeau.

Un programme Python traduisant cela pourrait être :

```
age = int(input())
nbKm = int(input())
if ( (12 <= age) and (age <= 25) ) or (age >= 60):
   print("Réduction possible")
else:
   print("Pas de réduction")
if ( ( (age < 12) or (age > 25) ) and (age < 60) ) and (nbKm >= 5000):
   print("Cadeau")
else:
   print("Pas de cadeau")
```

Ce programme est complexe et on sent qu'il y a des répétitions. En effet, si on définit la variable

```
reductionPossible = ( (12 <= age) and (age <= 25) ) or (age >= 60)
```

alors on a un cadeau si `eductionPossible` n'est pas vraie et si la longueur du trajet est plus grande que 5000km.

Mais, si une condition n'est pas vraie, c'est que la condition contraire est vraie !

Il est possible de calculer le contraire d'une condition en Python, en utilisant l'opérateur booléen `not` qui renvoie le contraire de la valeur qu'on lui donne :

```
age = int(input())
nbKm = int(input())
reductionPossible = ( (12 <= age) and (age <= 25) ) or (age >= 60)

if reductionPossible:
   print("Réduction possible")
else:
   print("Pas de réduction")
if ( not (reductionPossible) ) and (nbKm >= 5000):
   print("Cadeau")
else:
   print("Pas de cadeau")
```

Le programme est tout de suite beaucoup plus clair !

Lorsqu'on utilise l'opérateur booléen `not` pour avoir le contraire d'une condition, on dit qu'on a pris la négation de la condition.

### 1.7.5 Booléens : choses à ne pas faire

Une maladresse classique avec les booléens est de faire quelque chose comme ceci

```
prix = int(input())
estCher = (prix > 100)
if estCher == True:
   print("C'est cher !")
```

Le code est correct mais on n'a pas besoin de tester si quelque chose est égal à `True` ou `False`, si ce quelque chose est lui même déjà `True` ou `False` !

On remplacera donc

```
if estCher == True:
   print("Cher")
```

par

```
if estCher:
   print("Cher")
```

et

```
if estCher == False:
   print("Pas Cher")
```

par

```
if not(estCher):
   print("Pas Cher")
```

Un programme ne doit jamais contenir de `== True` ou `== False`.

### 1.7.6 Faire des tests : le "sinon si"

Un grand magasin propose une offre spéciale : si on achète pour plus de 300 euros de produits on a une remise de 40 euros, sinon si on achète pour plus de 200 euros on a une remise de 25 euros, sinon si on achète pour plus de 100 euros on a une remise de 10 euros sinon on a aucune remise.

Si on traduit ce programme en Python cela donne :

```
prix = int(input())
if prix >= 300:
   prix = prix - 40
else:
   if prix >= 200:
      prix = prix - 25
   else:
      if prix >= 100:
         prix = prix - 10
print(prix)
```

On a cependant beaucoup de `if/else` imbriqués et le programme est très indenté, imaginez si on avait 10 conditions différentes !

Mais, dans la phrase du début on a beaucoup utilise le terme **"sinon si"** et il existe une structure en Python qui correspond exactement à cela, il s'agit de la construction `elif` qui s'utilise ainsi :

```
prix = int(input())
if prix >= 300:
   prix = prix - 40
elif prix >= 200:
   prix = prix - 25
elif prix >= 100:
   prix = prix - 10
print(prix)
```

On utilisera donc le `elif` quand il y a beaucoup de cas différents à tester.

## 1.8 Chapitre 8 : Répétitions "tant que"

### 1.8.1 Faire la même chose plusieurs fois : le *"tant que"*

On a parfois besoin de répéter certaines instructions sans savoir à l'avance combien de fois il faudra le faire. Par exemple demander un mot de passe tant que l'utilisateur n'a pas donné le bon.

On a ici utilisé dans la phrase le terme **"tant que"**, ce qui signifie qu'on a bien une condition pour savoir quand s'arrêter mais on ne sait pas à l'avance combien de fois la personne va se tromper !

Heureusement Python a une instruction **"tant que"**, que nous allons pouvoir utiliser. Il s'agit de l'instruction `while`, qui signifie **"tant que"** en anglais.

```
secret = 123456
motdepasse = 0
while motdepasse != secret:
   print("Tapez le mot de passe : ")
   motdepasse = int(input())
print("Vous avez trouvé !")
```

Ainsi tant que la condition `motdepasse != secret` est vraie on continue à demander un nouveau mot de passe.

On peut représenter l'exécution du programme par le diagramme suivant :

Il est bien sûr possible d'utiliser des opérateurs booléens pour combiner des conditions et les valeurs booléennes sont également utilisables. Voici quelques extraits de code à titre d'exemple :

```
while True:
   print("J'attends")
```

Le premier de ces trois exemples est ce qu'on appelle une **"boucle infinie"**, c'est-à-dire que le programme ne s'arrête jamais : comme `True` est toujours vrai on ne quitte jamais le `while`.

```
while (motdepasse != secret) or agePersonne <= 3:
   print("Accès refusé : mauvais mot de passe ou personne trop jeune")
   agePersonne = int(input())
   motdepasse = int(input())


while nbPersonnes <= nbMax and temperature <= 45:
   print("Portes ouvertes")
   nbPersonnes = nbPersonnes + 1
   temperature = int(input())
```

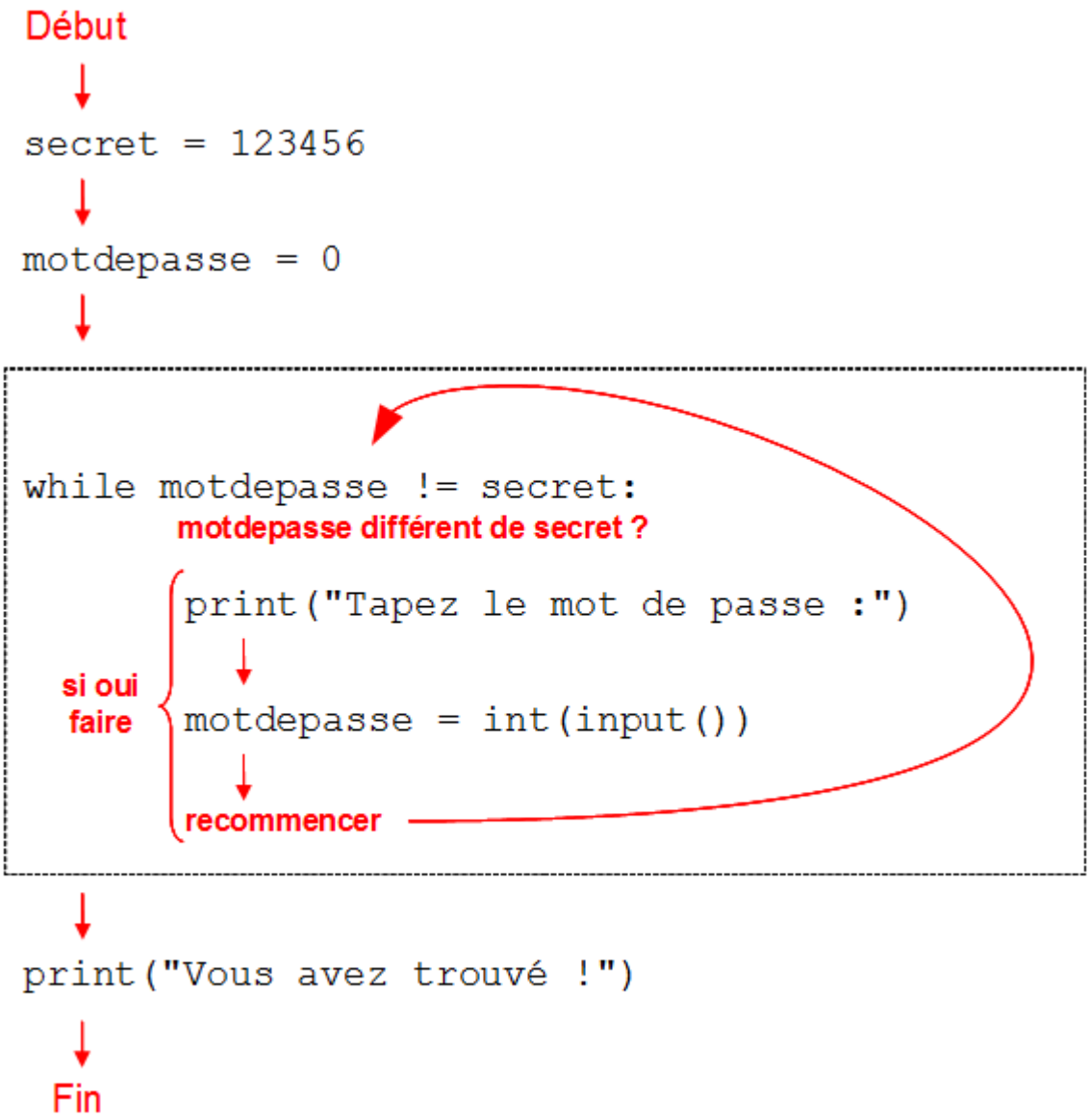


FIGURE 1.6 – Schéma d'exécution du while

**Note :** Nous parlions de **"boucle 'de répétition"'** pour désigner la structure `for loop`, nous parlerons de **"boucle 'tant que"'**, de **"boucle while"** ou tout simplement de **"boucle"** pour désigner la structure `while`.

### 1.8.2 Boucle infinie

Dans un code utilisant une boucle *"tant que"* ou boucle *"while"*, si on utilise une mauvaise condition, il est possible que le programme ne s'arrête jamais, par exemple :

```
valeur <- 1
Tant que valeur <= 10 faire
   valeur <- valeur - 1
```

Ici on a mis un `-1` au lieu d'un `+1` et du coup la variable valeur ne fait que diminuer au lieu d'augmenter. Elle ne sera donc jamais plus grande que `10` et le programme ne s'arrêtera jamais !

**Note :** Si vous écrivez un tel programme, le système d'évaluation automatique du site vous indiquera que votre programme a *"dépassé la limite de temps"*. En effet, pour éviter les programmes qui ne s'arrêtent jamais, nous avons un système qui les bloque automatiquement s'ils mettent trop de temps à donner leur réponse.

Si vous voyez un tel message, essayez de vérifier les conditions dans vos boucles.

**Note :** Lorsqu'un programme ne s'arrête jamais car il est resté piégé dans une boucle, on dit de manière plus courte que le programme boucle.

CHAPITRE 2

# Notions fondamentales de programmation

## 2.1 Commentaires

### 2.1.1 Utilisation de commentaires

Lorsqu'un de vos programmes fait plus que quelques lignes, il est parfois difficile pour quelqu'un de le comprendre tout de suite (et ce quelqu'un peut très bien être vous-même quelques semaines plus tard !). Pour aider à comprendre la structure du programme et les difficultés qu'il contient, on peut écrire des commentaires dans son programme.

Un commentaire est un texte qui commence par le caractère # appelé dièse.

Il peut commencer tout au début d'une ligne ou bien être placé à la suite d'une instruction.

Dans tous les cas, tout le texte qui suit le # sera complètement ignoré lors de l'exécution du programme. Voici un exemple :

```
# Ce programme a été écrit par Hermione Granger le 10/01/1994
# Quatrième année, cours d'étude des moldus"

# Affiche un rectangle rempli de X
for loop in range(5):
   for loop in range(10):
      print("X", end = "")  # pas de retour à la ligne ici
   print("")
```

**Bien programmer**

La priorité est toujours d'écrire le code le plus clair possible, pour qu'on puisse en comprendre un maximum sans explications. On réservera donc l'utilisation des commentaires aux explications des parties les plus complexes d'un programme.

## 2.2 Types de base

**Prérequis**
– Afficher

**Résumé**
– entier
– flottant

– booléen
– chaîne.

### 2.2.1 La notion de type

Les objets manipulées par un programme Python ont un type, au sens d'un langage de programmation : un type correspond à une catégorie d'objets (entiers, nombres réels, valeurs logiques, etc).

### 2.2.2 Exemples de types

On va décrire les types les plus usuels.

```
1  print(42)
2  print(4 + 6)
3  print(-5 * 2)
4
5  print(421.5)
6  print(2.7 + 10)
7
8  print(421 > 42)
9  print(421 < 42)
10
11 print("toto")
```

**Sortie**

```
42
10
-10
421.5
12.7
True
False
toto
```

– Lignes 1-3 : type entier. Les entiers peuvent être positifs comme négatifs.
– Lignes 5-6 : type flottant. Pour simplifier, il s'agit de nombres dit « à virgule », ci-dessus représentés en base 10. Le point utilisé dans le nombre représente notre virgule décimale (« flottant » fait allusion à la notion de virgule « flottante »).
– Lignes 8-9 et 17-18 : type booléen. Une expression de type booléen a une valeur de vérite True ou False.
– Ligne 11 : le type chaîne. En première approximation, une chaîne représente une suite de caractères. Dans l'exemple, pour que Python reconnaisse le mot toto comme une donnée de type chaîne, on entoure le mot d'une paire de guillemets.

### 2.2.3 Usage des types en Python

Python est un langage typé : toute donnée utilisée possède un type (entier, flottant, complexe, chaîne de caractères, etc). Le langage Python contient de très nombreux types (plusieurs dizaines).

En Python, le type est un outil plutôt théorique et détermine en partie
– les opérations qu'on peut effectuer entre données
– les valeurs que peuvent prendre les données.

Par exemple

```
print(3 * "Hello")
```

**Sortie**

```
HelloHelloHello
```

– Ligne 1 : on peut « multiplier » (avec l'opérateur `*`) un entier (ici 3) et une chaîne (`"Hello"`) :

Soit maintenant le code :

```
1 print(3 + "Hello")
```

**Sortie**

```
Traceback (most recent call last):
  File "somme_chaine.py", line 1, in <module>
    print(3 + "Hello")
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

– Ligne 1 : on veut « additionner » un entier et une chaîne
– Lignes 2-5 : on obtient un message TypeError (ligne 5) ce qui signifie une erreur de type : on ne peut pas « additionner » un entier et une chaîne (cf. ligne 1).

On voit donc que certaines opérations sont possibles et d'autres pas : le typage permet entre autres de savoir qu'elles sont les opérations permises ou pas.

Le typage en Python est réel et même assez complexe mais d'importance pratique secondaire. Quand on débute en Python, l'existence de types est peu importante. Même lorsqu'on est plus avancé, l'utilisation explicite de types est plutôt à éviter.

## 2.3 Opérations avec des variables

**Prérequis**

Opérations sur les nombres Affichage et variables

**Résumé**

Intérêt pratique des variables. Modification de variable, réaffectation.

### 2.3.1 Opérations usuelles

Quand on veut récupérer (« sauvegarder ») le résultat d'un calcul, on le place dans une nouvelle variable.

```
1 x = 42
2 y = 5
3 z = (x * y + 5) * (x ** 2 + y **2 ) - 100
4 u = (z - 10000) * x
5
6 print(u)
```

qui affiche

```
15730470
```

– Lignes 3 : on stocke sous le nom de `z` le résultat d'un calcul en mémoire
– Lignes 4 : on stocke sous le nom de `u` le résultat d'un autre calcul
– Ligne 6 : on affiche le contenu de la variable

### 2.3.2 Variable modifiée ou pas ?

Une opération utilisant une variable n'affecte pas le contenu de la variable :

```
1 toto = 42
2 print(toto + 1)
3 print(toto)
```

**Sortie**

```
43
42
```

– Ligne 2 : on ajoute `1` à la valeur de `toto`. Le résultat (ici 43) est affichée mais ce calcul est « perdu » puisque la valeur n'est pas replacé dans une variable et n'est donc pas réutilisable par la suite du programme.
– Ligne 5 : bien qu'on ait ajouté `1` à `toto`, le contenu de la variable `toto` reste inchangé par rapport à sa valeur initiale.

### 2.3.3 Réaffectation

Il peut arriver qu'un calcul avec une variable, disons `x`, soit suivi de la modification de la valeur de la variable `toto`.

```
1 x = 42
2 x = 2 * x + 10
3 print(x)
```

**Sortie**

```
94
```

– Ligne 2 : L'expression $2 * x + 10$ est évaluée et la valeur obtenue (94) est réaffectée à `x`. Ce type d'affectation (ligne 1 vs ligne 2) s'appelle une réaffectation. La valeur initiale d'affectation (ici 42) est « perdue ».

Augmenter de `1` la valeur d'une variable (comme `toto` ci-dessous) s'appelle une **incrémentation**.

```
1 toto = 42
2 print(toto)
3
4 toto = toto + 1
```

**Sortie**

```
42
43
```

– Ligne 4 : `toto + 1` est évalué et la valeur obtenue (43) est réaffectée à `toto`.

**Note :** La pratique de la réaffectation est extrêmement courante.

## 2.4 Utilisation d'un «compteur»

**Prérequis**
– Instruction if dans une boucle for
– Opérations avec des variables

**Résumé**

Implémentation en Python d'une technique élémentaire d'algorithmique pour dénombrer à l'aide d'une variable initialement placée à 0 et qu'on augmente, en général, d'une unité, chaque fois qu'une certaine propriété se produit.

### 2.4.1 Problème-type

On va illustrer la technique du compteur en se basant sur le problème suivant :

On donne une liste `t` d'entiers et on demande de donner le nombre d'entiers de `t` compris entre 18 et 65 au sens large. Par exemple, si `t = [12, 81, 82, 9, 31, 65, 46]`, la réponse demandée est 3.

### 2.4.2 La technique

La technique est de principe très simple et s'assimile au fait de compter sur ses doigts :
– Au début du programme, on définit une variable, nommé par exemple `cpt` et initialisée à `0`. À la fin du programme, la valeur de `cpt` sera le nombre recherché.
– On parcourt la liste `t` avec la technique vue à l'unité **Boucle for : parcours d'une liste par indices**.
– À chaque étape du parcours, on teste si l'élément courant est entre 18 et 65 au sens large. Si oui, on augmente de 1 la valeur du compteur `cpt`.
– Dans le jargon, on dit parfois que la sélection par le critère « être entre 18 et 65 » est un **filtrage** et que l'augmentation du compteur d'une unité est une **incrémentation**

### 2.4.3 Le code

Voici le code implémentant la technique du compteur pour le problème posé ci-dessus :

```
1  t = [12, 81, 82, 9, 31, 65, 46]
2  cpt = 0
3  i = 0
4
5  for loop in range(len(t)):
6      x = t[i]
7      if 18 <= x <= 65:
8          cpt = cpt +1
9      i = i + 1
10
11 print(cpt)
```

**Sortie**

```
3
```

– Ligne 2 : variable compteur définie en début de programme et initialisée à `0`
– Ligne 6 : extraction de l'élément courant de la liste
– Ligne 8 : incrémentation du compteur si (cf. ligne 7) l'élément courant de la liste satisfait le critère.
– Ligne 9 : en fin de parcours, `cpt` contient le nombre requis.

### 2.4.4 Usage

La technique du compteur est extrêmement fréquente en programmation. Elle s'utilise chaque fois que l'on veut déterminer le nombre de fois qu'une certaine propriété se rencontre et pas nécessairement en la présence d'une liste. En général, la technique du compteur est associée à une boucle.

## 2.5 Boucle while

**Prérequis**
– Affichage amélioré
– Affichage et passage à la ligne
– Les booléens
– Réaffectation de variable

**Résumé**

Exemple typique d'utilisation d'une boucle `while`. Description syntaxique, contexte d'utilisation, examen pas à pas de son exécution.

### 2.5.1 Un problème typique

Étant donné un entier $n$, on cherche un entier noté m qui soit le premier multiple de 10 supérieur ou égal à $n$. Si par exemple $n = 42$ alors $m = 50$ : en effet, $50$ est bien un multiple de 10 et il n'y en pas d'autre entre $42$ et $50$. Voici d'autres exemples :

| | | | |
|---|---|---|---|
| $n$ | 2013 | 420 | 0 |
| $m$ | 2020 | 420 | 0 |

#### L'idée à implémenter

Les multiples de 10 sont les entiers : $0, 10, 20, 30$ etc. Ils s'écrivent sous la forme $10k$ où $k$ varie à partir de $0$.

Pour trouver $m$, on parcourt les multiples de 10 depuis $0$ et on s'arrête une fois qu'on a atteint ou dépassé l'entier n donné. Autrement dit, on énumère les multiples $m$ de 10 tant que $m < n$. Le parcours est montré dans le tableau ci-dessous où on suppose que $n = 42$.

| | | | | | | |
|---|---|---|---|---|---|---|
| $k$ | 0 | 1 | 2 | 3 | 4 | 5 |
| $10k < n$? | oui | oui | oui | oui | oui | non |

### 2.5.2 Implémentation en Python

La notion de « dès que », « une fois que » ou de « tant que » est traduite dans le code Python par l'instruction while.

Le code suivant répond au problème posé :

**while_typique.py**

```
1 n = 42
2 k = 0
3
4 while 10 * k < n:
5     k = k + 1
6
7 print(10 * k)
```

**Sortie**

50

On voit (cf. lignes 7 et 8) que la solution au problème est 50.

### 2.5.3 Analyse de code

On passe en revue le code de `while_typique.py`.

#### Vocabulaire de la boucle while

La boucle `while` se trouve aux lignes 4-5. Ces deux lignes forment une seule et même instruction, dite instruction `while`. Ligne 4 : l'en-tête de la boucle `while` est la partie qui commence avec `while` et se termine avant le séparateur **:** (deux-points). Ce qu'on appelle le corps de la boucle `while` est tout le bloc indenté (ici ligne 5) qui est sous les deux- points. Répétition

Une boucle `while` répète une action (ligne 5) tant qu'une condition , ici (cf. ligne 4) :

```
10 * k < n
```

reste vraie. Plus précisément, si la condition est vraie, l'exécution entre dans le corps de la boucle et à la fin du corps de la boucle, la condition est à nouveau testée (d'où le terme de boucle) si la condition est fausse, l'exécution du programme va au-delà du corps de la boucle, ici à la ligne 6.

#### Variable de contrôle

Une boucle `while` fait souvent appel à une variable de contrôle, ici $k$, qui évolue pendant la boucle. Typiquement,
- cette variable est initialisée avant la boucle, ici ligne 2
- cette variable est réaffectée, dans le corps de la boucle while, ici par l'instruction ligne 5
- la condition à tester (ici ligne 4) est différente d'un tour de boucle à l'autre car elle dépend de k qui, entre-temps, a varié.

#### Remarques

– Lignes 4 : dans l'immense majorité des cas, le corps d'une boucle `while` est indenté.
– Il est possible de faire d'autres essais de code en changeant juste la valeur de n à la ligne 1. Par exemple, si on change `n` en `2013`, le programme affichera `2020`.
– Il se peut que l'exécution du code n'entre même pas dans le corps de la boucle `while`. Par exemple, si ligne 1, on choisit `n=0` au lieu de `n=42`, le test `10 * k < n` (ligne 4) échoue immédiatement et donc le programme continue à la ligne 6 sans même passer par la ligne 5.

### 2.5.4 Comprendre comment s'exécute une boucle while

Modifions le code précédent `while_typique.py` pour mieux comprendre l'exécution de la boucle `while` :

```
1  n = 42
2  k = 0
3  print("avant while", "k=", k)
4  print()
5
6  while 10 * k < n:
7    print("debut while", "k=", k, "10*k=", 10*k)
8    k = k + 1
9    print("fin while", "k=", k)
10   print()
11 print("apres while")
12 print(10 * k)
```

**Sortie**

```
avant while : k= 0

debut while : k= 0 10*k= 0
fin while : k= 1

debut while : k= 1 10*k= 10
fin while : k= 2

debut while : k= 2 10*k= 20
fin while : k= 3

debut while : k= 3 10*k= 30
fin while : k= 4

debut while : k= 4 10*k= 40
fin while : k= 5

apres while
50
```

En plus du code initial, `while_typique_affichage.py` contient des instructions d'affichage (lignes 3, 7, 9 et 11) et des instructions de sauts de ligne (lignes 4 et 10) dans la sortie pour observer l'évolution de `k` et de `10 * k` avant, pendant et après la boucle while.

L'exécution du programme est la suivante :
– Lignes 2 et 7 : la valeur de `k` avant le commencement de la boucle `while`
– Ligne 6 : k vaut 0. Le test de la boucle `while` est effectué : a-t-on `0 < 42` ? La réponse est oui donc, l'exécution du code continue dans le corps de la boucle `while`

– Lignes 7-10 et lignes 15-16 : Les affichages sont effectués : `k` est changé de `0` à `1`.
– Ligne 6 : le test de la boucle `while` est à nouveau effectué : a-t-on `10<42` ? La réponse est oui et l'exécution entre à nouveau dans le corps de la boucle. Cette opération se répète jusqu'à ce que `k = 5` (ligne 8 et lignes 18-28).
– Ligne 6 : le test de la boucle `while` est effectué : a-t-on `50<42` ? Cette fois, la réponse est non donc l'exécution quitte la boucle et continue lignes 11 et 12, cf. lignes 30 et 31.
– Ligne 12 : le résultat affiché (cf. ligne 31) est bien le résultat demandé. Comme le test `10 * k < n` a échoué pour la première fois, c'est qu'on a `10 * k >= n` avec `k` minimal et c'est bien ce que l'on cherchait.