# Cours d'informatique pour tous

Lycée Faidherbe

Année scolaire 2013-2014

# Table des matières


**1 Découverte de python** **1**
- I Contexte général de ce cours . . . 1
- II Principales caractéristiques . . . 2
- III Notion de variable . . . 4
- IV Type de donnée . . . 7

**2 Les structures conditionnelles** **19**
- I Les mots clés . . . 19
- II Les structures conditionnelles . . . 19
- III Les opérateurs de test . . . 21
- IV Des structures conditionnelles imbriquées . . . 23
- V Thèmes . . . 24
- VI Solutions . . . 24

**3 Les fonctions** **27**
- I Les mots clés . . . 27
- II Pourquoi des fonctions ? . . . 27
- III Des fonctions prédéfinies . . . 28
- IV Définir ses propres fonctions . . . 30

**4 Les boucles : Conditionnelles ou non** **37**
- I Les mots-clés . . . 37
- II Pourquoi des boucles ? . . . 37
- III Les boucles conditionnelles . . . 39
- IV Boucles inconditionnelles . . . 40
- V Résumé . . . 42
- VI Complexité . . . 43
- VII Thèmes . . . 44
- VIII Boucles imbriquées . . . 44
- IX Solutions . . . 45

**5 Listes** **47**
I Types composés . . . 47
II Listes comme tableaux . . . 48
III Parcours dune liste . . . 49
IV Constructions de nouvelles listes . . . 51
V Modification sur place . . . 52
VI Structure de liste . . . 53
VII Listes de listes . . . 55

**6 Structures de données : les chaînes de caractères et les tuples** **59**
I Quelques méthodes associées à l'objet chaîne de caractères . . . 59
II Retour sur les entrées-sorties . . . 60
III Retour sur les types rencontrés . . . 62
IV Recherche naïve d'un mot dans une chaîne de caractères . . . 62
V les n-uplets (tuple) . . . 64
VI Mutable, non mutable . . . 65

**7 Machine et système d'exploitation** **67**
I Qu'est ce qu'un ordinateur ? . . . 67
II En pratique . . . 72
III Le système d'exploitation . . . 80

# Chapitre 1

# Découverte de python

## I Contexte général de ce cours

### I.1 Langages

Le dialogue avec l'ordinateur se fait à l'aide de langages qui sont de niveaux plus ou moins profond. Chaque processeur possède un langage propre, directement exécutable : *le langage machine*. Il est formé de 0 et de 1 et n'est pas portable c'est-à-dire qu'il est propre au processeur qui l'utilise, mais c'est le seul que l'ordinateur puisse utiliser.

Le *langage d'assemblage* est un codage alphanumérique du langage machine. Il est plus lisible que le langage machine, mais n'est toujours pas portable. On le traduit en langage machine par un assembleur.

Par exemple, un certain type de processeur reconnaît une instruction du type 10110000 01100001

En langage assembleur, cette instruction est représentée par un équivalent plus facile à comprendre pour le programmeur : `movb 0x61,\%al` avec 10110000 à la place de `movb \%al` et 01100001 pour `0x61`.

Ce qui signifie : «écrire le nombre 97 (la valeur est donnée en hexadécimal : $\overline{61}^{16} = 6 \times 16 + 1 = 97$) dans le registre AL».

Les *langages de haut niveau* souvent normalisés permettent le portage d'une machine à l'autre. Ils sont traduits en langage machine par un compilateur ou un interpréteur. Dans ce cours d'informatique pour tous, nous utiliserons un langage de haut niveau : Python.

### I.2 Historique succint des langages

Bref historique des langages

- Années 50 (approches expérimentales) : FORTRAN, LISP, COBOL, ALGOL...
- Années 60 (langages universels) : PL/1, Simula, Smalltalk, Basic...
- Années 70 (génie logiciel) : C, PASCAL, ADA, MODULA-2...
- Années 80 (programmation objet) : C++, LabView, Eiffel, Perl, VisualBasic...
- Années 90 (langages interprétés objet) : Java, tcl/Tk, Ruby, Python...
- Années 2000 (langages commerciaux propriétaires) : C#, VB.NET...

Des centaines de langages ont été créés, mais l'industrie n'en utilise qu'une minorité.

### I.3 Historique du langage Python

- 1991 : Guido van Rossum travaille aux Pays-Bas au Centrum voor Wiskunde en Informatica sur le projet AMOEBA (un systeme d'exploitation distribué). Il conçoit Python (à partir du langage ABC) et publie la version 0.9.0 sur un forum Usenet
- 1996 : sortie de Numerical Python
- 2001 : naissance de la PSF (Python Software Fundation)
- Les versions se succèdent. Un grand choix de modules sont disponibles, des colloques annuels sont organisés, Python est enseigné dans plusieurs universités et est utilisé en entreprise...
- Fin 2008 : sorties simultanées de Python 2.6 et de Python 3.0
- 2013 : versions en cours : v2.7.3 et v3.3.0

## II Principales caractéristiques

- Langage Open Source
  - Licence Open Source CNRI, compatible GPL, mais sans la restriction copyley. Python est libre et gratuit meme pour les usages commerciaux
  - GvR (Guido van Rossum) est le Benevolent Dictator for Life (dictateur bénévole à vie !)
  - Importante communaute de développeurs
  - Nombreux outils standard disponibles : Batteries included
- Travail interactif
  - Nombreux interpréteurs interactifs disponibles
  - Importantes documentations en ligne
  - Développement rapide et incrémentiel
  - Tests et débogage outillés

  - Analyse interactive de données
- Langage interprété rapide
  - Interprétation du bytecode compilé
  - De nombreux modules sont disponibles è partir de bibliothèques optimisées (souvent écrites en C ou C++)
- Simplicité du langage
  - Syntaxe claire et cohérente
  - Indentation significative
  - Gestion automatique de la mémoire (garbage collector)
  - Typage dynamique fort : pas de déclaration
- Orientation objet
  - Modèle objet puissant mais pas obligatoire
  - Structuration multifichier très facile des applications : facilite les modifications et les extensions
  - Les classes, les fonctions et les méthodes sont des objets dits de première classe. Ces objets sont traités comme tous les autres (on peut les affecter, les passer en paramètre)
- Ouverture au monde
  - Interfaçable avec C/C++/FORTRAN
  - Langage de script de plusieurs applications importantes
  - Excellente portabilité
- Disponibilité de bibliothèques
  - Plusieurs milliers de packages sont disponibles dans tous les domaines

## II.1 Objectif

Le référentiel ministériel précisant les notions à aborder dans ce cours dit que «*Au niveau fondamental, on se fixe pour objectif la maîtrise d'un certain nombre de concepts de base, et avant tout, la conception rigoureuse d'algorithmes et le choix de représentations appropriées des données.*»

Nous allons dans un premier temps préciser les notions d'algorithmes et de données.

**Définition 1.1** (Algorithme)**.** *Un algorithme est un ensemble d'étapes permettant d'atteindre un but en répétant un nombre fini de fois un nombre fini d'instructions.*

Un algorithme doit se terminer en un temps fini.

On peut l'écrire dans un *pseudo-langage* utilisant des instructions élémentaires comme l'affectation, la comparaison, *tant que. . .faire. . .*, *Si . . .alors. . .sinon. . .*, etc.

Un *programme* est la traduction d'un algorithme en un langage compilable ou interprétable par un ordinateur. Il est souvent écrit en plusieurs parties dont une qui pilote les autres : le programme principal.

On s'efforcera au cours de ces deux années d'appliquer la *méthodologie procédurale*. On emploie l'analyse descendante (division des problèmes) et remontante (réutilisation d'un maximum de sous-algorithmes). On s'efforce ainsi de décomposer un problème complexe en sous-programmes plus simples. Ce modèle structure d'abord les actions.

Afin de faciliter la relecture de programme, on pensera à les commenter. Pour cela il suffira de précéder un texte explicite par un dièse : **#**.

# III Notion de variable

L'essentiel du travail effectué par un programme d'ordinateur consiste à manipuler des données. Ces données peuvent être très diverses, mais dans la mémoire de l'ordinateur elles se ramènent toujours en définitive à une suite finie de nombres binaires. Pour pouvoir accéder aux données, le programme d'ordinateur (quel que soit le langage dans lequel il est écrit) fait abondamment usage d'un grand nombre de variables de différents types. Une variable apparaît dans un langage de programmation sous un nom à peu près quelconque (voir ci-après), mais pour l'ordinateur il s'agit d'une référence désignant une adresse mémoire, c'est-à-dire un emplacement précis dans la mémoire vive. À cet emplacement est stockée une valeur bien déterminée. C'est la donnée proprement dite, qui est donc stockée sous la forme d'une suite de nombres binaires, mais qui n'est pas nécessairement un nombre aux yeux du langage de programmation utilisé. Cela peut être en fait à peu près n'importe quel «objet»susceptible d'être placé dans la mémoire d'un ordinateur, par exemple : un nombre entier, un nombre réel, un nombre complexe, un vecteur, une chaîne de caractères typographiques, un tableau, une fonction, etc.

Pour distinguer les uns des autres ces divers contenus possibles, le langage de programmation fait usage de différents types de variables (le type entier, le type réel, le type chaîne de caractères, le type liste, etc).

Les noms de variables sont des noms que vous choisissez vous-même assez librement. Efforcez-vous cependant de bien les choisir : de préférence assez courts, mais aussi explicites que possible, de manière à exprimer clairement ce que la variable est sensée contenir.

Sous Python, les noms de variables doivent en outre obéir à quelques règles simples :

- Un nom de variable est une séquence de lettres (a ... z , A ... Z) et de

chiffres (0 ... 9), qui doit toujours commencer par une lettre ;
- seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que $, #, etc. sont interdits, à l'exception du caractère _ (underscore) ;
- la casse est significative (les caractères majuscules et minuscules sont distingués).

Prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules (y compris la première lettre). Il s'agit d'une simple convention, mais elle est largement respecté. N'utilisez les majuscules qu'a l'interieur même du nom, pour en augmenter eventuellement la lisibilité.

En plus de ces règles, il faut encore ajouter que vous ne pouvez pas utiliser comme nom de variables les 33 mots clés réservés ci-dessous (ils sont utilisés par le langage lui-même) :

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

On retiendra qu'une *variable* est un identificateur (une chaîne de caractères bien construite) associé à une valeur.

## III.1 Affectation

**Définition 1.2** (Affectation)**.** *L'opération par laquelle on établit un lien entre le nom de la variable et sa valeur (son contenu) est appelée* affectation *ou assignation.*

En Python comme dans de nombreux autres langages, l'opération d'affectation est représenté par le signe égale `=`

L'affectation est un exemple d'*instruction* c'est-à-dire un ordre que l'on donne à la machine.

Remarque : Ce choix du symbole `=` est un parti pris, on aurait pu opter pour le symbole $\leftarrow$. Cela se voit dans certains cas du langage Caml. En turbo-Pascal, on utilise :=

On sera vigilant sur le fait que le `=` de Python n'a pas la même signification que dans un cours de mathématique, de physique ou de sciences de l'ingénieur.

En Python, `x = 4` devrait se lire *x reçoit la valeur 4* et non pas *x égal 4*.

La fonction `print( )` permet l'affichage des valeurs des variables. Cette fonction print est aussi une instruction que l'on donne à la machine.

Exemple : commenter les lignes suivantes. Ce qui suit l'invite >>> a été tapé par l'utilisateur dans la console, le reste constitue les réponses de la machine après un «retour charriot»(touche entrée).

```
>>> a=6        #a recoit 6
>>> print(a)   # on affiche la valeur de a
6
>>> b=7.9
>>> print(b)
7.9
>>> print(a+b)
13.9
>>> message="Python for ever"
>>> print(message)
Python for ever
>>> message
'Python for ever'
```

En Python, il est loisible de faire des affectations multiples

```
>>> x=y=8
>>> print(x,y)
8 8
>>> x=5.6
>>> print(x,' et ', y)
5.6  et  8
```

et même des affectations paralèlles

```
>>> f,g = 5.7, 3
>>> print('f vaut ',f,' et g vaut ',g)
f vaut 5.7  et g vaut 3
```

Une même variable peut voir sa valeur évoluer. Les anciennes valeurs sont oubliées par la machine.

```
>>> graal='Arthur'
>>> graal
'Arthur'
>>> graal=1975
>>> graal=graal-1 #decrementation
>>> print(graal)
1974
>>> graal=graal+ 1 # incrementation
>>> print(graal)
```

```
1975
```

On notera que lors d'une affectation, la machine calcule d'abord la valeur du terme de droite avant de l'affecter à l'identificateur.

Signalons une ≪Pythonnerie≫c'est-à-dire une spécificité de la syntaxe de Python, le symbole `+=` permet d'ajouter une valeur à une variable sous la forme

*nom de la variable* += *valeur à ajouter*

```
>>> m = 6
>>> m += 5
>>> m
11
>>> m += -3
>>> m
8
```

On retient que `m += 5` a le même effet que `m= m+5`

# IV Type de donnée

Dans l'exemple du dessus, la variable nommée `a` (on dira la variable `a`) a pour valeur 6 et `b` a pour valeur 7.9 ou 7,9.

La machine fait la différence entre le caractère (nombre) entier de 6 et le caractère réel (on dira *flottant*) de 7,9.

On parle de *typage de données*, on remarque que Python effectue un typage *dynamique* dans le sens où il suffit d'écrire que `a` reçoit 6 pour que la machine sache que `a` est de type entier.

En Java ou en Turbo-Pascal, ce n'est pas le cas, toute variable doit être typée avant de lui assigner une valeur.

Le type de donnée détermine la façon dont la machine va stocker la variable.

Passons en revue quelques types de données.

## IV.1 Type int

Il est utilisé dans le cadre des nombres entiers. En Python, le type int n'est limité en taille que par la mémoire de la machine.

Commenter les lignes suivantes et interpréter les opérations

\+ - * ** / // % abs.

```
>>> 34+7
41
>>> 56-98
-42
>>> 6*6
36
>>> 4**3
64
>>> 56/19
2.9473684210526314
>>> 56//19
2
>>> 45%7
3
>>> abs(78-45)
33
>>> abs(45-89)
44
```

Les exemples précédents permettent d'appréhender la notion d'*expression*.

**Définition 1.3** (Expression)**.** *Une* expression *est une portion de code que l'interpréteur Python peut évaluer pour obtenir une valeur.*

Les expressions peuvent être simples ou complexes. Elles sont formées d'une combinaison de littéraux représentant directement des valeurs, d'identificateurs et d'opérateurs.

```
>>> a=6  # ceci est une instruction
>>> 4+a  # ceci est une expression
10
```

Dans l'expression `4+a`, 4 est une valeur, `a` un identificateur et `+` un opérateur.

## IV.2 Type float

La notion mathématique de réel n'a pas d'équivalent en informatique. Certains nombres dits *irrationnels* n'ont pas d'écriture décimale périodique, il est impossible de les contenir en mémoire avec leur valeur exacte. C'est le cas pour $\sqrt{2}$ ou $\pi$. On se contente de valeurs approchées. Ce point fera l'objet d'un chapitre spécifique au cours de l'année.

Les valeurs approchant les nombres réels sont appelés *nombres flottants* pour rappeler que leur virgule «flotte»en fonction de la précision d'approximation.

En Python, ces valeurs ont le type float. On les représentente avec un point à la place de notre virgule ou bien avec un exposant `e` qui signifie puissance 10.

```
>>> 8.98547
8.98547
>>> 898547e-5
8.98547
>>> 89854.7e-4
8.98547
```

Les flottants supportent les mêmes opérations que les entiers.

Ils ont une précision finie limitée.

L'import du module `math` autorise un grand nombre d'opérations mathématiques usuelles. Par exemple :

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sin(math.pi/4)
0.7071067811865475
>>> math.log(5)
1.6094379124341003
>>> math.log(2.7)
0.9932517730102834
```

## IV.3 Conversion de type

Il est possible de convertir une variable float en variable de type int avec la fonction `int()`.

```
>>> int(4.3)
4
>>> int(4.9)
4
>>> int(-3.2)
-3
>>> int(-3.8)
-3
>>> int(0.3)
0
>>> a=6.7
>>> b=int(a)
```

```
>>> b
6
```

La fonction `float()` réalise l'inverse.

```
>>> float(5)
5.0
```

## IV.4 Type booléen

Le type booléen est réservé aux variables pouvant prendre deux valeurs : `False` ou `True`.

Les comparaisons ont pour résultat des valeurs booléenes. Avec l'exemple ci-dessous, donnons les significations des opérateurs suivants

$==$ $!=$ $>$ $>=$ $<$ $<=$

```
>>> (5+7)==12
True
>>> (5+7) != 12
False
>>> (+7) !=4
True
>>> 6 > 8
False
>>> 6>5
True
>>> 6>6
False
>>> 6>=(4+2)
True
```

Une erreur classique sera de confondre = et ==, ce qui ne vous arrivera jamais bien sûr.

On dispose aussi d'opéteurs logiques : `and`, `or` et `not`. Compléter les tableaux suivants.

| a | b | a and b | a or b |
|---|---|---|---|
| true | true | | |
| true | false | | |
| false | true | | |
| false | false | | |

puis

| a | not a |
|---|---|
| true | |
| false | |

```
>>> (6>7) and (4<9)
False
```

```
>>> (6>=3) and (4<9)
True
```

La fontion `int` transforme `True` en 1 et `False` en 0.

On voit le caractère *polymorphe* de la fonction qui supporte en entrée une variable flottante ou une variable booléene.

## IV.5 Fonction type

La fonction `type` permet d'afficher le type d'une expression ou d'une valeur ou d'une variable.

```
>>>a=6
>>> type(a)
<class 'int'>
>>> type(6>3)
<class 'bool'>
>>> type(a+8.0)
<class 'float'>
```

Sur le dernier exemple, on voit que la machine additionne un entier et un flottant et que le résultat est un flottant.

Commenter les lignes suivantes.

```
>>> type(a==6)
<class 'bool'>
>>> type(a=6)
Traceback (most recent call last) :
  File "<console>", line 1, in <module>
TypeError : type() takes 1 or 3 arguments
```

Remarque : certains langages dits fortement typés donnent un type aux instructions.

## IV.6 Chaîne de caractère

Une donnée de type `string` peut se définir comme une suite quelconque de caractères. Dans un script python, on peut délimiter une telle suite de caractères, soit par des apostrophes (simple quotes), soit par des guillemets (double ou triple quotes). Exemples :

```
>>> marcel='Longtemps, '
>>> marcel
'Longtemps, '
```

```
>>> proust='je me suis couché de bonne heure'
>>> print(marcel,proust)
Longtemps, je me suis couché de bonne heure
>>> type(marcel)
<class 'str'>
>>> a='"u"'
>>> a
'"u"'
>>> print(a)
"u"
>>> z="'t'"
>>> print(z)
't'
```

Remarquez l'utilisation des guillemets pour délimiter une chaîne dans laquelle il y a des apostrophes, ou l'utilisation des apostrophes pour délimiter une chaîne qui contient des guillemets.

La fonction `print()` insère un espace entre les éléments affichés.

Le caractère spécial \ (antislash) permet quelques subtilités complémentaires.

En premier lieu, il permet d'écrire sur plusieurs lignes une commande qui serait trop longue pour tenir sur une seule (cela vaut pour n'importe quel type de commande).

À l'intérieur d'une chaîne de caractères, l'antislash permet d'insérer un certain nombre de codes spéciaux (sauts à la ligne, apostrophes, guillemets, etc.).

Exemples :

```
>>> txt3 = '"N\'est-ce pas ?" répondit-elle.'
>>> Salut = "Ceci est une chaîne plutôt longue\n contenant
    plusieurs lignes \
... de texte (Ceci fonctionne\n de la même façon en C/C
   ++.\n\
... Notez que les blancs en début\n de ligne sont
   significatifs.\n"
>>> print(Salut)
Ceci est une chaîne plutôt longue
 contenant plusieurs lignes ... de texte (Ceci fonctionne
 de la même façon en C/C++.
... Notez que les blancs en début
 de ligne sont significatifs.
```

Remarques :

- la séquence \n dans une chaîne provoque un saut à la ligne ;
- la séquence \' permet d'insérer une apostrophe dans une chaîne délimitée par des apostrophes ; de la même manière, la séquence \" permet d'insérer des guillemets dans une chaîne délimitée elle-même par des guillemets ;
- rappelons encore ici que la casse est significative dans les noms de variables (il faut respecter scrupuleusement le choix initial de majuscules ou minuscules) ;
- triple quotes : pour insérer plus aisément des caractères spéciaux dans une chaîne, sans faire usage de l'antislash, ou pour faire accepter l'antislash lui-même dans la chaîne, on peut encore délimiter la chaîne à l'aide de triples guillemets ou de triples apostrophes.

```
>>> a1 = """
... Exemple de texte préformaté, c'est-à-dire
... dont les indentations et les
...  caractères spéciaux \ ' " sont
... conservés sans
... autre forme de procès."""
>>> print(a1)
... Exemple de texte préformaté, c'est-à-dire
... dont les indentations et les
...  caractères spéciaux \ ' " sont
... conservés sans
... autre forme de procès.
```

Les chaînes de caractères ont un type qui s'appelle `str` pour string en Python. Il n'est pas modifiable c'est-à-dire qu'une donnée, une fois créée en mémoire, ne pourra plus être changée, toute transformation résultera en la création d'une nouvelle valeur distincte.

## IV.7 Opération sur les chaînes

- longueur : `len(chaine)` donne la longueur de lstinline ?chaine ?
- Concaténation avec le symbole `+`

```
        >>> marcel
'Longtemps, '
>>> marcel+proust
'Longtemps, je me suis couché de bonne heure'
>>> phrase=marcel+proust
>>> print(phrase)
Longtemps, je me suis couché de bonne heure
```

- Répétition comme une multiplication

```
>>> victor='Waterloo, '
>>> victor*3
'Waterloo, Waterloo, Waterloo, '
```

## IV.8 Méthode

Il existe d'autres façons d'agir sur les chaînes. On utilise la forme *variable*.`fonction()`. Attention ! la chaîne d'origine n'est pas affectée par les modifications, le résultat doit être récupéré par une variable : `variable=chaine.fonction()`.

Cette façon d'opérer avec la méthode pointée est spécifique à la programmation objet.

Pour une variable dénommée `chaine` :

- `chaine.upper()` convertit en majuscule par exemple

```
>>> print('petit'.upper())
PETIT
```

- `chaine.lower()` convertit en minuscule par exemple

```
>>> marcel='Longtemps, '
>>> petit=marcel.lower()
>>> print(petit, marcel)
longtemps, Longtemps,
```

- `chaine.isupper()` et `chaine.islower()` retournent True si la chaîne ne contient respectivement que des majuscules ou minuscules.
- `chaine.capitalize()` passe la première lettre de la chaîne en capitale
- `chaine.title()` passe la première lettre de chaque mot de la chaîne en capitale
- `chaine.swapcase()` transforme minuscules en majuscules et inversement
- `chaine.split()` renvoie une liste de tous les mots (séparation : l'espace) d'une chaîne-phrase
- `chaine.split('*')` le séparateur peut être une chaîne autre qu'un espace

Il existe d'autres méthodes, vous les trouverez facilement sur les aides en lignes.

## IV.9 Les chaînes de caracteres : indexation simple

Pour indexer une chaine, on utilise l'operateur [ ] dans lequel l'index, un entier, indique la position d'un caractère en débutant l'indexation à 0. On

peut mettre un index négatif le terme d'index $-1$ est le dernier caractère, celui d'index moins la longueur de la chaîne est le premier caratère.

```
>>> victor='Waterloo, '
>>> victor[0]
'W'
>>> type(victor[0])
<class 'str'>
>>> victor[1]+victor[5]
'al'
>>> len(victor)
10
>>> victor[-10]
'W'
>>> victor[-1]
' '
```

## IV.10 Tranche

Il est possible d'extraire des morceaux de chaîne de caractères. Le résultat est une nouvelle chaîne de caractère. Cette opération ne modifie pas la chaîne sur laquelle elle s'applique.

La syntaxe est la suivante, soit `machaine` une variable de type string. Le code `machaine[debut :fin]` retourne la chaîne de caractère avec les lettres de `machaine` dont les indices sont compris entre `debut` et `fin -1`. Si on ne met pas de valeur pour `debut`, la machine met 0 par défaut. Si on ne met pas de valeur pour `fin`, la machine met la longueur de `machaine` par défaut.

Il est possible de mettre des valeurs négatives comme on l'a vu au-dessus.

```
>>> wam='La flûte enchantée'
>>> wam[1:4]
'a f'
>>> wam[-4:]
'ntée'
>>> wam[:3]
'La '
>>> wam[6:]
'te enchantée'
```

On peut aussi sélectionner des caractère avec un pas constant, par défaut le pas est 1. La syntaxe est la suivante. Le code `machaine[debut :fin :pas]` retourne la chaîne de caractère avec les lettres de `machaine` dont les indices sont compris entre `debut` et `fin -1` avec un pas de longueur `pas`.

Par exemple `machaine[4 : 18 : 3]` retourne la chaîne avec les lettres d'indices 4, 7, 10, 13, 16 de `machaine`

```
>>> wam[::1]  # intérêt très limité !!
'La flûte enchantée'
>>> wam[4:-3:6]
'ln'
>>> wam[3:len(wam) :2]
'fûeecate'
```

Remarque : cette indexation avec 0 pour le premier terme peut sembler étrange. Elle l'est moins si on considère que ce sont les interstices entre les lettres qui sont indexés.

| | L | | a | | | f | |
|---|---|---|---|---|---|---|---|
| 0 | | 1 | | 2 | 3 | | 4 |

Ainsi l'exemple `wam[1:4]` signifie que l'on prend la tranche entre les interstices 1 et 4, c'est-à-dire `'a f'`.

Dans la doc de Python (http://docs.python.org/3/library/functions.html?highlight=input#input), on peut lire au sujet de `input` :

*The function [...] reads a line from input, converts it to a string (stripping a trailing newline), and returns that.*

*La fonction lit une ligne depuis l'input, la convertit en une chaîne de caractère (en supprimant le saut de ligne) et la retourne.*

Autrement dit, la fonction donne la main à l'utilisateur qui tape un texte au clavier, la fonction convertit en chaîne de caractères et retourne cette chaîne. Que fait le code suivant ?

```
prenom = input("Entrez votre prenom  : ")
prenom_maj = prenom[0].upper() + prenom[1:]
print (prenom_maj)
```

Réponse : l'utilisateur rentre son prénom au clavier, la variable `prenom` reçoit cette valeur, `prenom_maj` reçoit la première lettre mise en majuscule concaténé avec le reste du prénom et on affiche la valeur de `prenom_maj`.

Par exemple

```
>>> prenom = input("Entrez votre prenom  : ")
Entrez votre prenom  : monthy
>>> prenom_maj = prenom[0].upper() + prenom[1:]
>>> print (prenom_maj)
Monthy
```

– «Une introduction à Python 3», Bob CORDEAU et Laurent POINTAL
– «Apprendre à programmer avec Python 3», Gérard Swinnen

# Chapitre 2

# Les structures conditionnelles

## I Les mots clés

A la fin de ce cours, il faudra appréhender les notions suivantes :
- La mise en place de structures conditionnelles : *if*, *elif* et *else*
- L'utilisation des opérateurs de comparaison
- L'exploitation des opérateurs booléens : *or*, *and*, *not*

## II Les structures conditionnelles

### II.1 Instruction if

L'instruction `if` permet de vérifier des conditions avant d'exécuter un bloc d'instructions. La forme la plus simple de l'instruction `if` est la suivante :

Programme 2.1 – instruction if

```
a=-5
if a<0:
   val_abs=-a
print(val_abs)
```

⇒ 5

Déroulement du programme avec une valeur positive pour `a` :

```
a=5
if a<0:
   val_abs=-a
print(val_abs)
```

⇒ `NameError: name "val-abs" is not defined`

L'affectation `val_abs=-a` est réalisée uniquement si le nombre est négatif d'où le message d'erreur dans le second cas.

La structure conditionnelle est repérée par l'instruction `if` suivi d'une expression booléenne. Si sa valeur est `True` (Vrai), alors le bloc d'instructions indenté est executé. Sinon, **Python** passe à la suite.

Indenté signifie que les lignes d'instructions sont décalées vers la droite à l'aide d'espaces (4 espaces au minimum ou une tabulation en veillant à ce que l'appui sur une tabulation soit bien converti en espaces) afin de montrer l'appartenance de ces instructions à la structure conditionnelle.

## II.2 Instruction elif et else

Une seconde forme élaborée de l'instruction `if` permet d'exécuter une ou plusieurs instructions alternatives. Il s'agit de l'instruction `elif` (qui est la contraction de `else if`) (programme 2.3). Enfin, l'instruction `else` permet de déterminer les autres cas ( programme 2.2).

Programme 2.2 – Valeur absolue

```
#Programme indiquant la valeur absolue d'un nombre
a=5
#Debut de la structure conditionnelle
if a<0:
    val_abs=-a
else :
    val_abs=a
#Reprise du programme
print(val_abs)
```

⇒ 5

Programme 2.3 – Signe d'un nombre

```
#Programme indiquant le signe d'un nombre
a=5
#Debut de la structure conditionnelle
if a>0:
    print("le nombre est positif")
elif a<0:
    print("le nombre est negatif")
else :
    print("le nombre est nul")
```

Dans une structure conditionnelle, plusieurs instructions `elif` peuvent être utilisées. par contre,il y a au plus une instruction `else`.

## II.3 Un exemple de synthèse

Considérons l'exemple d'un programme de bataille navale dans une version sommaire (2.4). Il ne s'agit pas de couler le porte-avions mais plutôt une barque tenant sur une seule case repérée par un chiffre (compris entre 0 et 9) pour la ligne et un chiffre pour la colonne. L'utilisateur choisit une case et le programme indique alors suivant les cas :

- Soit **coulé** : si la barque est exactement sur la case considérée
- Soit **en vue** : si le tir atteint la bonne ligne ou la bonne colonne.
- Soit **à l'eau** : si le tir est totalement raté.

Programme 2.4 – Bataille Navale

```
#choix de la position de la barque
ligne = 3
colonne = 6

#demande a l'utilisateur son choix de tir
util_ligne=int(input("Quelle ligne ?"))
util_colonne=int(input("Quelle colonne ?"))

#Test conditionnel afin de savoir si le tir a atteint la
    cible
if ligne==util_ligne and colonne==util_colonne :
    print("COULE")
elif ligne==util_ligne or colonne==util_colonne :
    print("EN VUE")
else :
    print("A L'EAU")
```

# III Les opérateurs de test

Un structure conditionnelle nécessite la définition d'une **condition** dont le résultat est booléen (True ou False).

## III.1 Les opérateurs de comparaison

L'analyse du programme (2.4) met en évidence la nécessité de comparer si la ligne choisie par l'opérateur est identique à celle où se situe le bateau. Cette vérification est réalisée par l'opérateur ==

```
#comparateur d'egalite
ligne == util_ligne
```

Il ne s'agit pas de l'opérateur = qui est réservé à l'affectation d'une valeur à une variable.

```
#affectation de la valeur 3 a la variable ligne
ligne = 3
```

Les autres opérateurs de comparaison sont :

Comparateurs

| | | | |
|---|---|---|---|
| Égalité | == | Différence | != |
| Strictement supérieur | > | Strictement inférieur | < |
| Supérieur ou égal | >= | Inférieur ou égal | <= |

## III.2 Les opérateurs booléens

Afin de savoir si le bateau est touché, il est nécessaire de réaliser un test à la fois sur la ligne et sur la colonne. Les opérateurs booléens vus dans le chapitre sur les variables permettent de réaliser ces tests multiples.

```
#operateur and (ET). Il faut les deux pour que le
   resultat du test soit vrai.
ligne==util_ligne and colonne==util_colonne

#operateur or (OU). Il faut l'un des deux ou bien les
   deux pour que le resultat du test soit vrai
ligne==util_ligne or colonne==util_colonne
```

Remarque, l'opérateur `or` n'est pas exclusif.

**Exercice 1** — *Analyser* Permutons les test `and` et `or` dans le programme de bataille navale

```
#Test conditionnel afin de savoir si le tir a atteint la
   cible
if ligne==util_ligne or colonne==util_colonne :
    print("EN VUE")
elif ligne==util_ligne and colonne==util_colonne :
    print("COULE")
```

```
else :
    print("A L'EAU")
```

Quelle serait alors le résultat du programme si l'utilisateur trouve la bonne case ?

# IV Des structures conditionnelles imbriquées

Dans notre exemple, l'usage des opérateurs booléens peut être évité, mais alors l'écriture du programme nécessite l'usage de tests imbriqués (2.5).

Programme 2.5 – Bataille Navale

```
#choix de la position de la barque
ligne= 3
colonne=6

#demande a l'utilisateur son choix de tir
util_ligne=int(input("Quelle ligne ?"))
util_colonne=int(input("Quelle colonne ?"))

#Test conditionnel afin de savoir si le tir a atteint la
    cible
if ligne==util_ligne :
    if colonne==util_colonne :
        print("COULE")
    else :
        print("EN VUE")
else :
    if colonne==util_colonne :
        print("EN VUE")
    else :
        print("A L'EAU")
```

Remarquez, le décalage vers la droite correspondant aux différentes indentations. L'imbrication des structures conditionnelles imposent la mise en place de différents niveaux d'indentations qu'il faut respecter.

**Exercice 2** — *Tester* L'utilisateur a choisi la ligne 3 et la colonne 2.

Suivre alors le déroulement du programme 2.5 de bataille navale en indiquant en vis à vis le résultat de la compilation.

Faire de même avec le choix ligne=5 et colonne=6.

Cette forme d'écriture utilisant des structures conditionnelles imbriquées est plus compliquée. L'usage des opérateurs booléens sera privilégié par la suite.

# V Thèmes

**Exercice 3** — *Écrire un programme* Écrire un programme qui étant donnée une équation du second degré $ax^2 + bx + c = 0$ identifiée par ses 3 coefficients, détermine le nombre de solutions réelles ainsi que leurs valeurs éventuelles.

**Exercice 4** — *Modifier un programme* En général, à la bataille navale, un bateau n'est "en vue" que si la case touchée est immédiatement voisine de celle du bateau. Modifier le programme 2.4 afin de tenir compte de cette règle. On pourra traiter le cas où les cases diagonalement adjacentes du bateau sont "en vue" et le cas où elle ne le sont pas.

**Exercice 5** — *Écrire un programme* On souhaite réaliser un test pour savoir si un entier est compris dans un intervalle allant de 2 à 8 inclus

Réaliser le programme en utilisant des structures conditionnelles imbriquées.

Reprendre le programme et le simplifier en utilisant les opérateurs booléens.

**Exercice 6** — *Écrire un programme* Vous allez devoir réaliser un programme permettant de savoir si une année saisie par l'utilisateur est une année bissextile.

Un année est dite bissextile si c'est un multiple de 4, sauf si c'est un multiple de 100. Toutefois, si c'est un multiple de 400, alors elle est considérée comme bissextile.

Aide : pour savoir si c'est un nombre est multiple d'un autre, utiliser l'instruction `%` qui donne le reste de la division.

# VI Solutions

**Solution de l'exercice 6** Écrire un programme

Programme 2.6 – Année bissextile

```
#Choix de l'annee par l'utilisateur
annee = int(input("saisissez une annee  :"))
print (annee)
```

```
#Résolution avec l'usage de boucles imbriquees
if annee % 4  != 0:
    print("l'annee n'est pas bissextile")
elif annee % 100 == 0:
    if annee % 400  != 0:
        print("l'annee n'est pas bissextile")
    else :
        print("l'annee est bissextile")
else :
    print("l'annee est bissextile")

print("avec les operateurs booleens")

if annee % 400 == 0 or (annee % 4 == 0 and not annee % 100
    == 0) :
        print("l'annee est bissextile")
else :
        print("l'annee n'est pas bissextile")
```

# Chapitre 3

# Les fonctions

## I Les mots clés

A la fin de ce cours, il faudra appréhender les notions suivantes :
- Utilisation de fonctions prédéfinies
- Importation de modules de fonctions : *import*
- Déclaration d'une nouvelle fonction : *def*
- Définition du corps d'une fonction en blocs d'instructions indentées
- Renvoi de valeurs *return*
- Documentation d'une fonction : *help*

## II Pourquoi des fonctions ?

### II.1 Pour structurer un développement de projet

Ou comment répondre à un projet complexe où les lignes de codes vont s'accumuler.

Un programme principal est structuré par décomposition en sous-programmes. Chaque sous-programme peut alors être implantée comme une fonction [1] du programme principal et être traitée séparément (vision descendante), voire par des personnes différentes.

On peut se placer du point de vue :
- du programmeur en charge de la réalisation d'une fonction particulière qui établit la séquence d'instructions.
- ou de l'utilisateur de la fonction qui n'est pas obligé de savoir comment elle est programmée mais qui a besoin de connaître quelles sont les

1. Cet usage du terme fonction est spécifique à Python et diffère de la définition plus restrictive vue en mathématique. Nous tâcherons par la suite de bien cerner les différences.

paramètres à fournir en entrée et quels sont les valeurs récupérables en sortie.

En outre, il n'est pas rare d'avoir des séquences qui se répètent à plusieurs reprises dans un programme. L'usage d'une fonction permet alors de définir une seule fois la séquence et d'y faire appel à de multiples reprises par la suite (vision remontante).

### II.2 Pour enrichir le langage

Ou l'art d'apprendre à un ordinateur à réaliser des tâches qu'il n'était pas capable de réaliser auparavant.

Il s'agit alors de définir ses propres fonctions qui vont permettre d'ajouter de nouvelles instructions au langage de programmation.

On peut ensuite regrouper un ensemble de fonctions dans un module et les importer afin de les utiliser dans un programme à la manière du module "math".

## III Des fonctions prédéfinies

Par défaut, le langage propose de multiples fonctions prédéfinies . D'autres peuvent être importées suivant l'usage au travers des modules. La première partie du cours sur les variables a permis de mettre en évidence l'usage de quelques fonctions incontournables. L'objectif ici est d'analyser la typologie de mise en uvre de ces fonctions, typologie qui sera réutilisée lorsque nous définirons nos propres fonctions par la suite.

### III.1 Quelques fonctions incontournables

Afin de faire le point, étudions l'exemple suivant d'un programme permettant de calculer le prix TTC. Ce programme fait appel à des fonction implantés d'origine en Python.

**Exercice 7** — *Commenter* Indiquer le rôle de chacune des lignes du programme 3.1 en ajoutant des commentaires.

**Exercice 8** — *Modifier* Adapter le programme 3.1 afin que l'utilisateur puisse choisir le taux de TVA.

**Remarque :** Le programme 3.1 utilise des fonctions pour :

Programme 3.1 – Calcul du prix TTC

```
#Programme de calcul du prix TTC
prix_ht = input ("Quel est le prix hors taxes ? \n")
print(type(prix_ht)) #la fonction input renvoie une
    chaine de caracteres
prix_ht=float(prix_ht)
print(type(prix_ht))
prix_ttc=prix_ht+prix_ht*19.6/100
print("le prix toutes taxes comprises est  : ", prix_ttc,
    "euros")
```

- **pour afficher un résultat** : *print(paramètre1,paramètre2)*. Les paramètres à afficher sont disposés à l'intérieur des parenthèses et sont séparés par des virgules.
- **pour recueillir l'avis de l'utilisateur** : *variable=input(paramètre)*. Cette fonction comporte des paramètres, le texte à afficher à l'utilisateur. Elle renvoie une valeur stockée dans une variable, la donnée entrée par l'utilisateur
- **pour traiter (typer) les données** : *type(paramètre)*, *variable=float(paramètre)* ... Ces fonctions permettent d'afficher le type de données (entier, flottant...) ou de modifier le type (exemple passage d'un entier à un flottant).

La liste des fonctions ne pourrait être exhaustive. Au fur et à mesure de notre pratique de Python, nous appréhenderons l'usage de nouvelles fonctions.

## III.2 Des fonctions intégrées en module

Pour des usages spécifiques, de nouvelles fonctions qui ne sont pas disponibles à l'origine, peuvent être importées en faisant appel à des modules. Par exemple, le module math rajoute des fonctionnalités mathématiques (voir programme 3.2).

Un autre exemple avec le module *random* qui permet d'effectuer des tirages aléatoires (voir programme 3.3).

Programme 3.2 – Module math

```
#importation des fonctions du module math
import math
#Definition des cotes d'un triangle rectangle
adjacent=7
oppose=4
hypothenuse=math.sqrt(adjacent**2+oppose**2) #appel  :
    math.fonction()
print(hypothenuse)
#Utilisation des fonctions trigonometriques
angle=math.atan(oppose/adjacent)
print(angle)
print(hypothenuse*math.cos(angle))
print(hypothenuse*math.sin(angle))
```

Programme 3.3 – Module random

```
#Module random permettant d'effectuer des tirages
    aleatoires
import random
a=random.randint(1,6) #Renvoi un entier tire au hasard
    dans l'intervalle [1,6], un peu comme au jeu de des
     :)
print(a)
```

# IV Définir ses propres fonctions

Les fonctions constituent le principal élément structurant d'un programme. Afin de s'initier à la déclaration de ses propres fonctions, commençons par le calcul d'une fonction mathématique simple :

$$y = f(x) = \frac{x^2+3x+2}{1+x^2}$$

Ce qui donne en Python (3.4) :

Programme 3.4 – Une première fonction

```
def funcF(e) :
    """Calcul d'une fonction mathematique simple"""
    s=(e**2+3*e+2)/(1+e**2)
    return s
```

Le résultat de la fonction peut alors être évalué(3.5) :

Programme 3.5 – l'appel à une fonction

```
>>> funcF(2)
8.0
>>> funcF(5)
5.5
>>> a=2
>>> b=funcF(a)
>>> b
8.0
```

**Définition 3.1.** *Une fonction est donc définie par :*

1. *une* ***introduction*** *déclarée par le mot-clef* ***def*** *;*
2. *un* ***corps de fonction*** *constitué d'un bloc d'****instructions indentées*** *;*
3. *une* ***finalisation*** *pouvant être initiée par le mot-clef* ***return****.*

Voici un second exemple (3.6).

Programme 3.6 – Une seconde fonction

```
from math import * #importe les fonctions math au meme
   plan que les fonctions d'origine
#plus besoin donc d'utiliser math.fonction()

def hypothenuse(a, b) :
    """Calcul la longueur de l'hypothenuse """
    c=sqrt(a**2 + b**2)
    return c
```

Recherchons les éléments constituants la fonction :

| Nom de la fonction : Hypothénuse | | |
|---|---|---|
| Introduction | Corps de fonction | Finalisation |
| Paramètres en entrée<br>a et b | Traitement<br>c=sqrt(a**2 + b**2) | Valeur en sortie<br>c |

D'un point de vue mathématique, cela s'écrirait :

$$c = hypothenuse(a, b) = \sqrt{a^2 + b^2}$$

## IV.1 L'initialisation de la fonction

La ligne d'initialisation de la fonction s'établit dans l'ordre suivant :

1. ***def***, mot-clef qui est l'abréviation de **"define"** et qui constitue le prélude à toute construction de fonction ;
2. Le **nom de la fonction** : choisissez un nom caractérisant de façon explicite le rôle de la fonction (dans notre exemple "hypothenuse") ;
3. La **liste des paramètres** qui seront fournis lors d'un appel de la fonction, séparés par des virgules et encadrés par des parenthèses ;
4. les **deux points " :"** qui clôturent la ligne et qui indiquent que les instructions suivantes seront incluses dans le corps de la fonction

```
#L'initialisation
def nom_fonction (parametre1 , parametre2)  :
```

## IV.2 Le corps de la fonction

Le corps de la fonction est constitué des lignes d'instructions qui ont été regroupées au sein de la fonction afin d'assurer le traitement souhaité. Afin d'identifier leur appartenance à la fonction, ces lignes sont indentées.

```
#L'initialisation
def nom_fonction (parametre1 , parametre2)  :
....#Le corps de la fonction
....Ligne instructions 1
....Ligne instructions 2
....#Indentation OBLIGATOIRE de 4 espaces vers la droite
```

## IV.3 La finalisation de la fonction

La finalisation de la fonction peut s'établir de deux manières :

– Lorsque le mot-clef ***return*** apparait. Cette instruction signifie qu'on **renvoie** une valeur (exemple résultat du calcul effectué par la fonction mathématique simple), pour pouvoir la récupérer ensuite et la stocker dans une variable par exemple. Cette instruction arrête le déroulement de la fonction, le code situé après le return ne s'exécutera pas.

– Lorsque **toutes les lignes** du corps de la fonction délimitées par les indentations ont été **traitées** en l'**absence** de mot-clef ***return***. La fonction ne renvoie donc pas de valeur. Il s'agit alors d'une suite d'instructions regroupée dans une fonction (plus précisément une procédure mais python ne fait pas de distinction) afin de structurer le programme (voir exemple 3.7 d'un code pouvant être utilisé à plusieurs reprises).

Programme 3.7 – Fonction à paramètres

```
def depart(destination,horaire) :
    print("-----------------------------------")
    print("L'avion a destination de ", destination," part
        a", horaire)
    print("-----------------------------------")

depart("tokyo","8h30")
depart("Miami","11h00")
depart("Moscou","12h25")
```

⇒

```
L'avion a destination de tokyo part a 8h30
-----------------------------------
-----------------------------------
L'avion a destination de Miami part a 11h00
-----------------------------------
-----------------------------------
L'avion a destination de Moscou part a 12h25
-----------------------------------
```

## IV.4 L'usage de la fonction

La compilation d'un programme s'effectuant ligne par ligne, la définition d'une fonction doit s'effectuer en amont de son usage. Ainsi pour notre fonction hypoténuse, son utilisation s'écrit :

```
valeur_hyp=hypothenuse(3,5)
print(valeur_hyp)
```

⇒ 5.830951894845301

### IV.4.a les paramètres d'une fonction

Les paramètres d'une fonction doivent être rentrées dans l'ordre (exemple 3.8).

Programme 3.8 – l'usage des paramètres

```
def devoirs(lycee,annee,jour)  :
    print("Au lycee", lycee, ", les devoirs sur table en",
        annee ,"ont lieu le",jour)
```

```
devoirs("Faidherbe","1ere annee","samedi")  :
```

⇒ Au lycee Faidherbe, les devoirs sur table en 1ere annee ont lieu le samedi

```
devoirs("samedi","1ere annee","Faidherbe")  :
```

⇒ Au lycee samedi, les devoirs sur table en 1ere annee ont lieu le Faidherbe

Il est également possible de définir des fonctions ne nécessitant pas de paramètre (voir programme 3.9).

Programme 3.9 – une fonction sans paramètre

```
def bonjour() :
    print("Bienvenue a Faidherbe")

>>> bonjour()
Bienvenue a Faidherbe
>>>
```

### IV.4.b Les valeurs renvoyées par une fonction

Le renvoie d'une valeur s'effectue à l'aide du mot-clef *return*. Ce n'est pas parce que la fonction affiche un résultat (avec *print()* dans le corps de la fonction) que la fonction renvoie une valeur pouvant être stockée. La valeur renvoyée par une fonction ne possédant pas de *return* est par défaut *none*.

La valeur peut être constituée de plusieurs éléments (voir programme 3.10).

Programme 3.10 – une fonction renvoyant plusieurs valeurs

```
def sphere(rayon) :
```

```
    """fonction calculant le volume et la surface d'une
       sphere
    Parametre d'entree  : rayon
    Valeurs renvoyees dans l'ordre surface, volume
    """
    surf=4*3.14*rayon**2
    vol=(4/3)*3.14*rayon**3
    return surf, vol

>>> surface, volume = sphere (2)
>>> print("La surface de la sphere est",surface,"et la
   volume est",volume)
La surface de la sphere est 50.24 et la volume est
   33.49333333333333
>>>
```

Une fonction peut comporter plusieurs *return*. auquel cas, la fonction se finalisera au premier *return*. La suite du code ne sera pas compilée (voir programme 3.11).

Programme 3.11 – une fonction avec plusieurs *return*

```
def parite(nombre) :
    if nombre%2==0:
        return "pair"
    else :
        return "impair"

>>> print(parite(12))
pair
>>> print(parite(13))
impair
>>>
```

### IV.4.c La documentation d'une fonction

Afin de savoir quel est le rôle d'une fonction, dans quel ordre il faut rentrer les paramètres ou quelles sont les valeurs renvoyées, il est important de documenter sa fonction (voir 3.12).

Programme 3.12 – l'usage des paramètres

```
def devoirs(lycee,annee,jour)  :
```

```
    """ Fonction affichant le jour des devoirs sur table
    Premier parametre  : le nom du lycee
    Second parametre  : le niveau
    Troisieme parametre : le jour des devoirs
    """
    print("Au lycee", lycee, ", les devoirs sur table en",
        annee ,"ont lieu le",jour)
```

La documentation de la fonction s'écrit dans le corps de la fonction, juste après la première ligne de déclaration. La documentation commence par """ et se termine par """. Il est possible de faire appel à cette documentation dans une console en tapant *help(fonction)*. Cette documentation doit donc indiquer :

- le rôle de la fonction ;
- la liste des paramètres attendues dans l'ordre ;
- la ou les valeurs renvoyées.

```
>>> help(devoirs)
Help on function devoirs in module __main__ :

devoirs(lycee, annee, jour)
    Fonction affichant le jour des devoirs sur table
    Premier parametre  : le nom du lycee
    Second parametre  : le niveau
    Troisieme parametre : le jour des devoirs

>>>
```

La documentation a pour rôle de permettre à un utilisateur de mettre en oeuvre une fonction dont il ne connait pas la structure interne.

La documentation ne remplace pas les commentaires que vous êtes invités à mettre dans le corps de votre fonction comme dans tout programme.

# Chapitre 4

# Les boucles : Conditionnelles ou non

## I Les mots-clés

A la fin de ce chapitre, il faudra savoir énoncer une définition des mots suivants :

- *while*
- *for*
- *range*
- boucle conditionnelle
- boucle inconditionnelle
- compteur de boucle
- condition de boucle
- complexité en temps de calcul

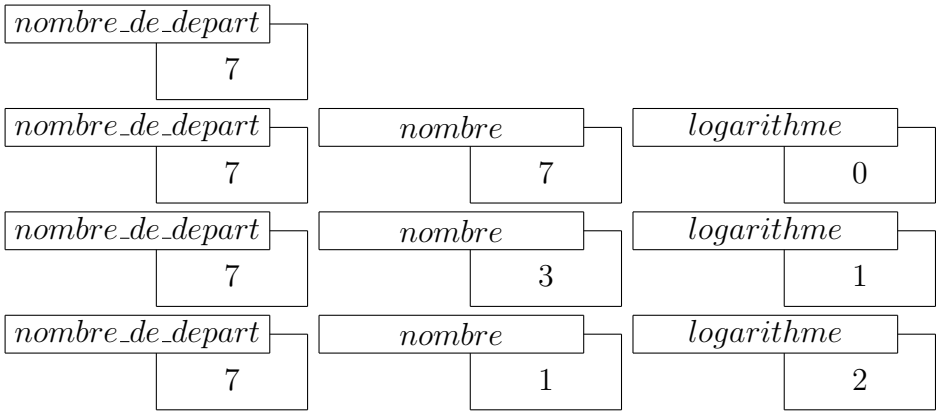
## II Pourquoi des boucles ?

Considérons le problème de la recherche du logarithme entier.
On se donne un *nombre_de_depart*, on le divise par 2 (avec une division euclidienne) et si le quotient est supérieur strictement à 1, on recommence jusqu'à ce que le quotient soit inférieur (ou égal) à 1. Le nombre de divisions effectué s'appelle logarithme entier de *nombre_de_depart*. Le programme 4.1 permet dans certains cas de déterminer le logarithme entier.
**Remarque :** On peut montrer en exercice que le logarithme entier d'un nombre $a$ est l'entier $n$ tel que $2^{n-1} < a \leqslant 2^n$.

Programme 4.1 – Un premier programme

```
# Tentative de calcul du logarithme entier
nombre_de_depart=7
nombre=nombre_de_depart
logarithme=0
if nombre>1: # si nombre<=1, la machine arrete
    nombre=nombre//2 # la double barre renvoie le quotient
        de la division euclidienne
    logarithme=logarithme+1
    if nombre>1: # si nombre<=1 la machine arrete
        nombre=nombre//2
        logarithme=logarithme+1
        if nombre>1: # meme chose
            nombre=nombre//2
            logarithme=logarithme+1
print('le logarithme entier de ',nombre_de_depart,' est ',
   logarithme)
```

On peut dessiner l'évolution de l'état des variables de la façon suivante :




Le programme s'arrête alors car la condition *nombre¿1* n'est plus vérifiée.

**Exercice 9** — *Evolution*

Pourquoi définir 2 variables *nombre_de_depart* et *nombre* dans le programme 4.1 ?

**Exercice 10** — *Limites du programme* Tester le programme 4.1 pour différentes valeurs de *nombre_de_depart*. Pour quelles valeurs le résultat est il celui attendu ?

On observe que ce programme répète plusieurs fois la même instruction

et qu'on l'a réécrite à chaque fois. On aimerait éviter d'avoir à effectuer cette réécriture et pouvoir demander à la machine, de façon concise, de répéter un certain nombre de fois un ensemble fixé d'instructions.
C'est le rôle des *boucles*, nous allons voir qu'on peut considérer qu'il y a deux sortes de boucles.

# III Les boucles conditionnelles

Programme 4.2 – Première boucle *while*

```
#Recherche du logarithme entier
nombre_de_depart=7
nombre=nombre_de_depart
logarithme=0
while nombre>1: # si nombre<=1, la boucle arrete
    nombre=nombre//2
    logarithme=logarithme+1
print('le logarithme entier de ',nombre_de_depart,' est ',
    logarithme)
```

Le programme 4.2 donne une solution au problème de recherche du logarithme entier. Le mot clé *while* indique à la machine qu'elle va devoir répéter un ensemble d'instructions qui suit le symbole ' :' tant que la *condition* est vérifiée. (c'est pour cela qu'on parle de *boucle conditionnelle*)
Afin de bien marquer le début et la fin de cet ensemble d'instructions, on décale (indente) chaque instruction de 4 espaces (ou une tabulation).
Tout le bloc d'instructions est décalé, par exemple dans le programme 4.2, le bloc est constitué de deux lignes d'instructions, l'instruction *print* ne fait pas partie du bloc. On remarquera enfin la compacité du code !

**Définition 4.1.** *On définit les différentes parties d'une boucles conditionnelle : on peut décomposer son code en 4 parties successives :*

1. ***L'initialisation*** *dans laquelle on définit la valeur des paramètres de départ (ici logarithme et nombre).*
2. *La définition d'une* ***condition de boucle*** *qui porte sur une (ou plusieurs) variable. (ici nombre¿1)*
3. ***Le bloc d'instructions*** *à répéter qui modifie les valeurs des paramètres et en particulier du paramètre qui sert au test. (ici nombre)*