

Généralité sur le langage



<http://www.python.org/>

Généralité sur le langage

Wiki

Définition Wikipédia :

Python est un langage de programmation objet, interprété, multi-paradigme, et multi-plateformes.

Il favorise la programmation impérative structurée et orientée objet.

Il est doté d'un typage dynamique fort, d'une gestion automatique de la mémoire par ramasse-miettes et d'un système de gestion d'exceptions ; il est ainsi similaire à Perl, Ruby, Scheme, Smalltalk et Tcl.

Généralité sur le langage

Historique et communauté

- Crée par [Guido van Rossum](#) en 1990



- Maintenu par une communauté de bénévoles, sous couvert de la *Python Software Foundation*

Généralité sur le langage

Deux façons de faire tourner les programmes

- Session interactive

Interpréteur = Calculatrice améliorée

Cf. le [chapitre « 3. Introduction informelle à Python » du Tutoriel Python officiel \(*\)](#)

- Lancement de scripts

- Ecrire un fichier source (.py) dans un éditeur
- Lancer son exécution (pas de compilation nécessaire : langage interprété)
- Le distribuer à d'autres

Algorithme et langage python

Gilles VERONIE

Généralité sur le langage

Outils de programmation et test

- interpréteur standard
- IDLE
- IPython

Généralité sur le langage

Interpréteur Python

Lancement de l'interpréteur, en mode *interactif* :

```
$ python
Python 2.7.3 (default, Jan  2 2013, 16:53:07)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Prompt primaire : >>>

```
>>> le_monde_est_plat = 1
>>> if le_monde_est_plat:
...     print "Attention au rebord!"
...
Attention au rebord!
```

Prompt secondaire : . . . , à l'intérieur d'une suite d'instructions.

Retour chariot ↵ pour revenir au prompt primaire.

Généralité sur le langage

Outil de développement IDLE

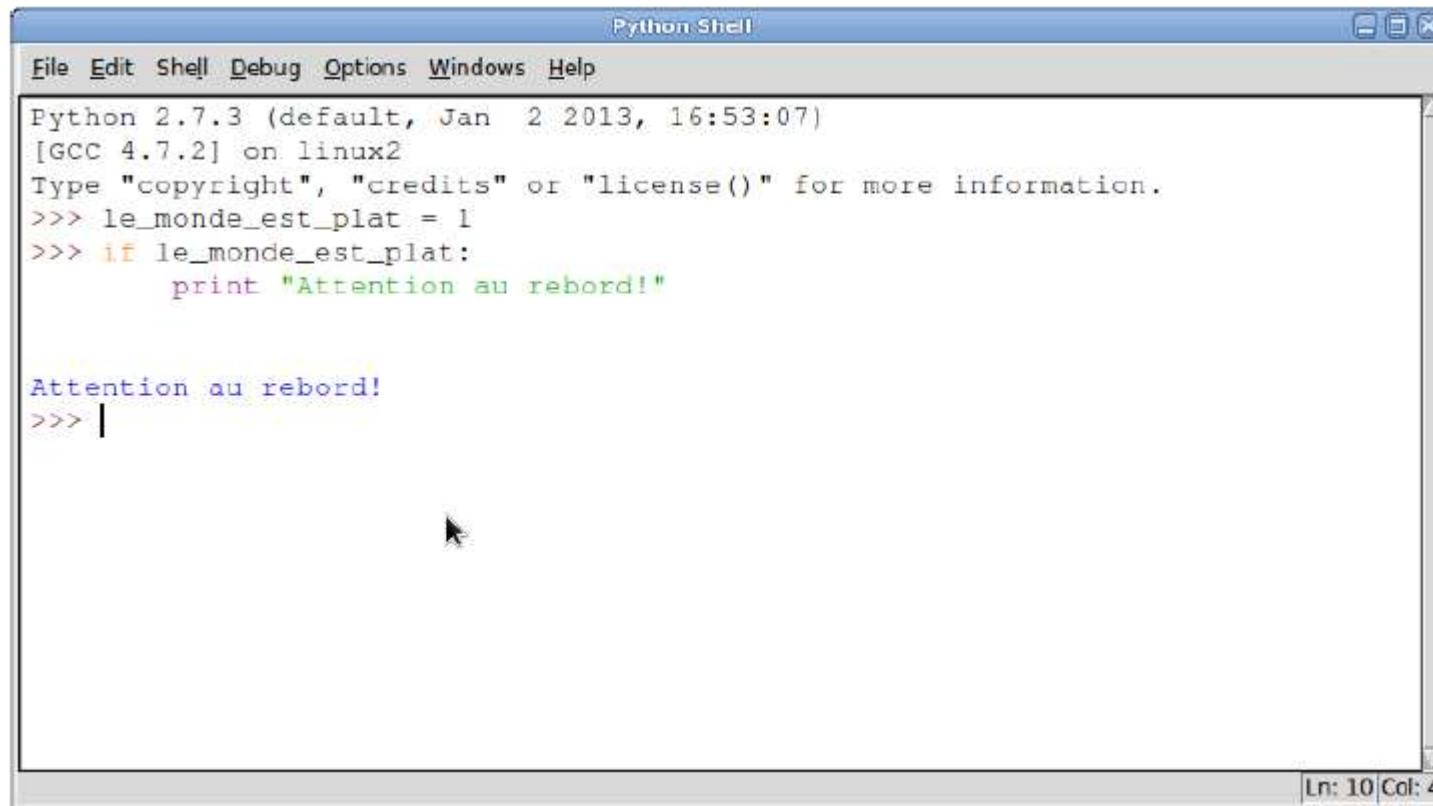
- Editeur + interpréteur (*python shell*)
- Fonctions intéressantes : débogueur, etc.
- 100% Python « standard » (tkinter) : portable
- Ergonomie discutable (ancien)

Algorithme et langage python

Gilles VERONIE

Généralité sur le langage

Copie écran type Shell



The screenshot shows a Python Shell window with the following content:

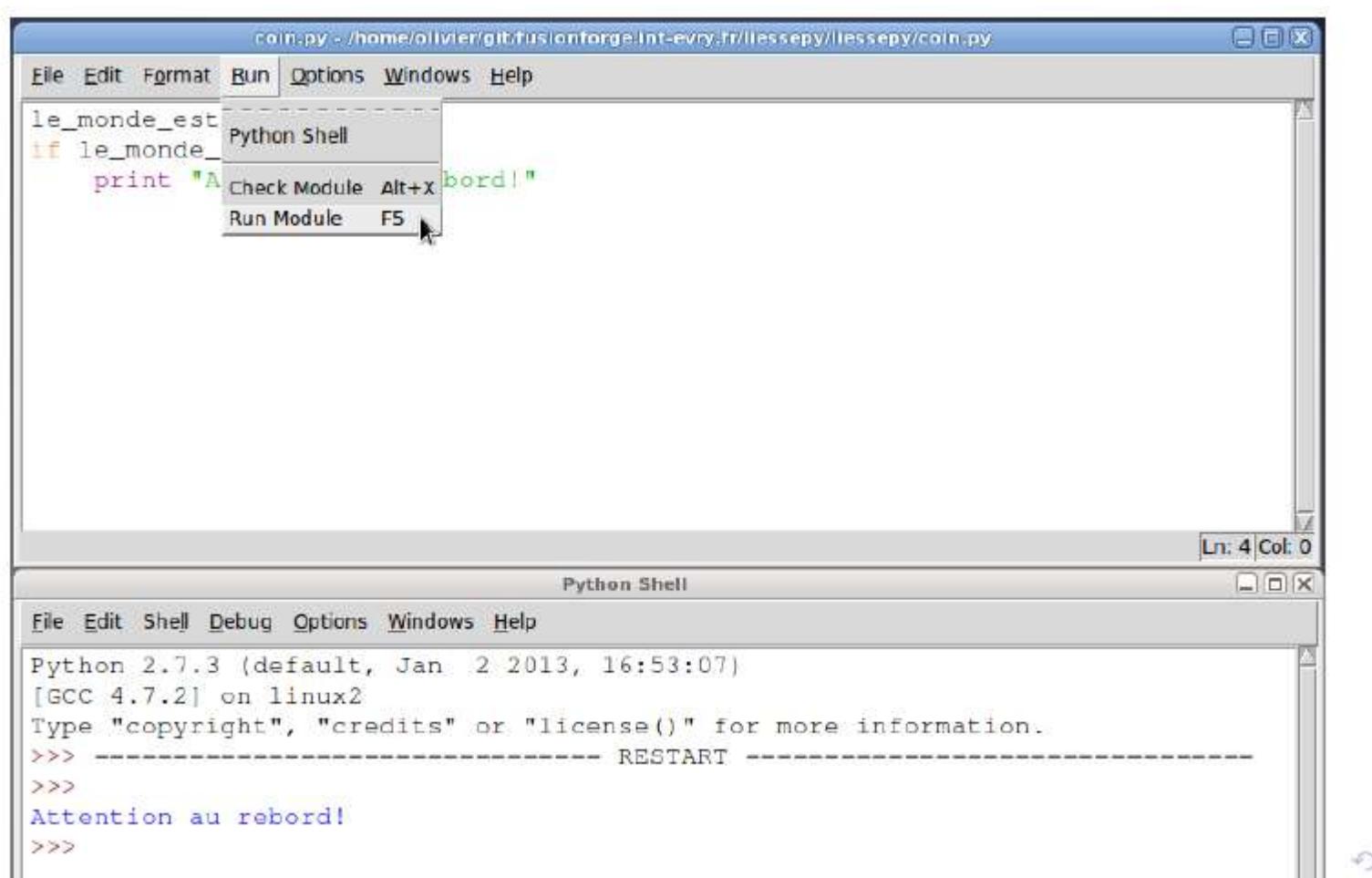
```
Python 2.7.3 (default, Jan  2 2013, 16:53:07)
[GCC 4.7.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> le_monde_est_plat = 1
>>> if le_monde_est_plat:
    print "Attention au rebord!"

Attention au rebord!
>>> |
```

The window has a menu bar with File, Edit, Shell, Debug, Options, Windows, and Help. The status bar at the bottom right shows Ln: 10 Col: 4.

Généralité sur le langage

Copie écran IDLE



Généralité sur le langage

Fonctionnalités IDLE

- colorisation
- complétion
- documentation
- *stack viewer*

Généralité sur le langage

Les Identifiants

- Mots clés du langage : `if`, `for`, `while`, `import`, `class`, etc.
- Noms des objets de bibliothèques : déconseillé (`sys`, `string`)
- Tout le reste : identifiants d' « objets » (ASCII uniquement) : variables, fonctions, classes, modules, etc.

Algorithme et langage Python

Gilles VERONIE

Généralité sur le langage

Les Identifiants / instructions de base

Statement	Result
<code>pass</code>	Null statement
<code>print([s1] [, s2]*)</code>	Writes to <code>sys.stdout</code> . Puts spaces between arguments <code>s1</code> . Puts newline at end unless arguments <code>end</code> with <code>end=</code> (ie : <code>end=' '</code>). <code>print</code> is not required when running interactively, simply typing an expression will print its value, unless the value is <code>None</code> .
<code>a = b</code>	Basic assignment - assign object <code>b</code> to label <code>a</code>
<code>if condition :</code> <code>suite</code> <code>[elif condition : suite]*</code> <code>[else :</code> <code>suite]</code>	Usual <code>if/else if/else</code> statement.
<code>while condition :</code> <code>suite</code>	Usual <code>while</code> statement.
<code>for element in sequence :</code> <code>suite</code>	Iterates over <code>sequence</code> , assigning each element to <code>element</code> . Use built-in <code>range</code> function to iterate a number of times.

Généralité sur le langage

Affectations

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 _ (souligné).

La « casse » est significative : les caractères majuscules et minuscules sont distingués. Ainsi, `python`, `Python`, `PYTHON` sont des variables différentes.

Par convention, on écrira l'essentiel des noms de variable en caractères minuscules (y compris la première lettre). On n'utilisera les majuscules qu'à l'intérieur même du nom pour en augmenter éventuellement la lisibilité, comme dans `programmePython` ou `angleRotation`. Une variable dont la valeur associée ne varie pas au cours du programme (on parle alors de constante) pourra être écrite entièrement en majuscule, par exemple `PI` ($\pi = 3.14$).

Le langage lui-même peut se réservier quelques noms comme c'est le cas pour `PYTHON` (figure 3.5). Ces mots réservés ne peuvent donc pas être utilisés comme noms de variable.

Généralité sur le langage

Affectations

Fig. 2.2 : MOTS RÉSERVÉS EN PYTHON

and	del	for	is	raise
assert	elif	from	lambda	return
break	else	global	not	try
class	except	if	or	while
continue	exec	import	pass	with
def	finally	in	print	yield

Généralité sur le langage

Variable définition

Une variable est un objet informatique qui associe un nom à une valeur qui peut éventuellement varier au cours du temps.

Les noms de variables sont des identificateurs arbitraires, de préférence assez courts mais aussi explicites que possible, de manière à exprimer clairement ce que la variable est censée référencer (la sémantique de la donnée référencée par la variable). Les noms des 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.

Généralité sur le langage

Affectation définition

L'affectation est l'opération qui consiste à attribuer une valeur à une variable.

L'instruction d'affectation est notée = en PYTHON : **variable = valeur**. Le nom de la variable à modifier est placé dans le membre de gauche du signe =, la valeur qu'on veut lui attribuer dans le membre de droite. Le membre de droite de l'affectation est d'abord évalué sans être modifié puis la valeur obtenue est affectée à la variable dont le nom est donné dans le membre de gauche de l'affectation ; ainsi, cette opération ne modifie que le membre de gauche de l'affectation. Le membre de droite peut être une constante ou une expression évaluable.

variable = constante : La constante peut être d'un type quelconque (figure 2.3) : entier, réel, booléen, chaîne de caractères, tableau, matrice, dictionnaire... comme le suggèrent les exemples suivants :

Généralité sur le langage

Affectations

Fig. 2.3 : TYPES DE BASE EN PYTHON

<i>type</i>	<i>nom</i>	<i>exemples</i>
<i>booléens</i>	<code>bool</code>	<code>False, True</code>
<i>entiers</i>	<code>int</code>	<code>3, -7</code>
<i>réels</i>	<code>float</code>	<code>3.14, 7.43e-3</code>
<i>chaînes</i>	<code>str</code>	<code>'salut', "l'eau"</code>
<i>n-uplets</i>	<code>tuple</code>	<code>1,2,3</code>
<i>listes</i>	<code>list</code>	<code>[1,2,3]</code>
<i>dictionnaires</i>	<code>dict</code>	<code>{'a' :4, 'r' :8}</code>

Algorithme et langage Python

Gilles VERONIE

Généralité sur le langage

Variables = constantes

```
booleen = False  
entier = 3  
reel = 0.0  
chaine = "salut"  
tableau = [5,2,9,3]  
matrice = [[1,2],[6,7]]  
nUplet = 4,5,6  
dictionnaire = {}
```

```
autreBooleen = True  
autreEntier = -329  
autreReel = -5.4687e-2  
autreChaine = 'bonjour, comment ça va ?'  
autreTableau = ['a',[6,3.14],[x,y,[z,t]]]  
autreMatrice = [[1,2],[3,4],[5,6],[7,8]]  
autreNUplet = "e",True,6.7,3,"z"  
autreDictionnaire = {"a":7, "r":-8}
```

Généralité sur le langage

Variables = expression

variable = expression : L'expression peut être n'importe quelle expression évaluable telle qu'une opération logique (`x = True or False and not True`), une opération arithmétique (`x = 3 + 2*9 - 6*7`), un appel de fonction (`y = sin(x)`) ou toute autre combinaison évaluable (`x = (x != y) and (z + t >= y) or (sin(x) < 0)`). ■ TD2.2

```
reste = a%b          quotient = a/b
somme = n*(n+1)/2    sommeGeometrique = s = a*(b**(n+1) - 1)/(b-1)
delta = b*b - 4*a*c  racine = (-b + sqrt(delta))/(2*a)
surface = pi*r**2     volume = surface * hauteur
```

L'expression du membre de droite peut faire intervenir la variable du membre de gauche comme dans `i = i + 1`. Dans cet exemple, on évalue d'abord le membre de droite (`i + 1`) puis on attribue la valeur obtenue au membre de gauche (`i`) ; ainsi, à la fin de cette affectation, la valeur de `i` a été augmentée de 1 : on dit que `i` a été incrémenté de 1 (figure 2.4) et on parle d'incrémantation de la variable `i` (remarque 2.6). PYTHON propose un opérateur d'incrémantation (`+=`) et d'autres opérateurs d'affectation qui peuvent toujours se ramener à l'utilisation de l'opérateur `=`, l'opérateur d'affectation de base (figure 2.5).

Généralité sur le langage

Variables = expression

Remarque 2.6 : *Avec l'exemple de l'incrémentation ($i = i + 1$), on constate que l'affectation est une opération typiquement informatique qui se distingue de l'égalité mathématique. En effet, en mathématique une expression du type $i = i+1$ se réduit en $0 = 1$! Alors qu'en informatique, l'expression $i = i+1$ conduit à ajouter 1 à la valeur de i (évaluation de l'expression $i+1$), puis à donner cette nouvelle valeur à i (affectation).*

Généralité sur le langage

Variables = expression

Fig. 2.4 : DÉFINITION DE L'ACADEMIE (5)

INCRÉMENT *n. m.* *XVe siècle, encrément. Emprunté du latin incrementum, « accroissement ».* *INFORM.* Quantité fixe dont on augmente la valeur d'une variable à chaque phase de l'exécution du programme.

DÉCRÉMENT *n. m.* *XIXe siècle. Emprunté de l'anglais decrement, du latin decrementum, « amoindrissement, diminution. Principales affectations en PYTHONution ».* *MATH. INFORM.* Quantité fixe dont une grandeur diminue à chaque cycle.

Généralité sur le langage

Variables = expression

Fig. 2.5 : PRINCIPALES AFFECTATIONS EN PYTHON

$a = b$	<hr/>	
$a += b$	\equiv	$a = a + b$
$a -= b$	\equiv	$a = a - b$
$a *= b$	\equiv	$a = a * b$
$a /= b$	\equiv	$a = a / b$
$a \% b$	\equiv	$a = a \% b$
$a **= b$	\equiv	$a = a ** b$

Généralité sur le langage

Affectations exemple

```
>>> a = 2
>>> a
2
>>> b = a + 3
>>> b
5
```

Pas de déclaration de variables. Typage dynamique.

```
>>> b = 'bonjour'
>>> b
'bonjour'
```

Généralité sur le langage

Affectations pour conclure

L'affectation a ainsi pour effet de réaliser plusieurs opérations dans la mémoire de l'ordinateur:

- créer et mémoriser un nom de variable,
- lui attribuer un type bien déterminé,
- créer et mémoriser une valeur particulière,
- établir un lien (par un système interne de pointeurs) entre le nom de la variable et l'emplacement mémoire de la valeur correspondante.

Généralité sur le langage

Types de base

- booléens (True, False)
- nombres
- chaînes de caractères (+ chaînes unicode)
- listes
- tableaux
- dictionnaires (tableaux associatifs)

Généralité sur le langage

Nombres

- entiers

```
>>> (50-5*6)/4  
5
```

- réels

```
>>> 7.0 / 2  
3.5
```

- opérateurs arithmétiques de base : +, -, /, *, **, etc

- conversions

```
>>> a = 3  
>>> a  
3  
>>> d = float(a)  
>>> d  
3.0
```

- complexes, fonctions de base

Généralité sur le langage

Chaines de caractères

Chaînes simples

```
>>> 'spam oeufs'  
'spam oeufs'
```

Affichage des chaînes

```
>>> "bonjour"  
'bonjour'  
>>> print "bonjour"  
bonjour  
>>> print 'coucou', 'tout', 'le', 'monde'  
coucou tout le monde
```

Généralité sur le langage

Chaines de caractères longues

Multi-lignes

```
>>> hello = "Ceci est une chaîne assez longue qui contient\n\"... plusieurs lignes de texte comme vous le feriez en C.\n\"... Notez que les espaces en début de ligne sont\n\"... significatifs."
>>> print hello
Ceci est une chaîne assez longue qui contient
plusieurs lignes de texte comme vous le feriez en C.
    Notez que les espaces en début de ligne sont significatifs.
```

Mieux :

```
>>> hello = """Ceci est une chaîne assez longue qui contient
... plusieurs lignes de texte comme vous le ferez en Python."""
>>> print hello
Ceci est une chaîne assez longue qui contient
plusieurs lignes de texte comme vous le ferez en Python.
```

Algorithme et langage Python

Gilles VERONIE

Généralité sur le langage

Chaînes de caractères unicode

unicodestrings.py

```
import sys
print sys.stdin.encoding ## cp1252 sous Windows, UTF-8 sous Linux
machaine = 'äâéç'
print machaine
chunicode = u'äâéç'
print chunicode.encode(sys.stdin.encoding)
```

Sur Linux :

UTF-8
äâéç
äâéç

Sur Windows :

cp1252
aÃ§Ã©Ã
äâéç

Généralité sur le langage

Opérations sur les chaînes

■ formatage : %

```
>>> a = "mais"
>>> b = "Non"
>>> c = "allo"
>>> d = "quoi"
>>> message = "%s %s %s, %s !" % (b, a, c, d)
>>> message
'Non mais allo, quoi !'
```

■ découpage

```
>>> e = message[4:13]
>>> e
'mais allo'
```

+---+---+---+---+---+
N o n m
+---+---+---+---+---+
0 1 2 3 4 5
-5 -4 -3 -2 -1 -0

Généralité sur le langage

Listes

Découpage :

```
>>> a = ['spam', 'oeufs', 100, 1234]
>>> a[1:]
['oeufs', 100, 1234]
```

Contenu modifiable :

```
>>> a[2] = a[2] + 23
>>> a
['spam', 'oeufs', 123, 1234]
```

Générateur de liste d'entiers : `range()`

```
>>> range(2, 10)
[2, 3, 4, 5, 6, 7, 8, 9]
```

Généralité sur le langage

Contrôle de flux

Instructions if, for, while, break, continue

Exemple : recherche de nombres premiers

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'egal a', x, '*', n/x
...             break
...     else:
...         # la boucle est terminee sans avoir trouve de facteur
...         print n, 'est un nombre premier'
...
2 est un nombre premier
3 est un nombre premier
4 egal a 2 * 2
5 est un nombre premier
6 egal a 2 * 3
7 est un nombre premier
8 egal a 2 * 4
9 egal a 3 * 3
```

Généralité sur le langage

Contrôle de flux: test

Sauf mention explicite, les instructions d'un algorithme s'exécutent les unes après les autres, dans l'ordre où elles ont été écrites. Le « chemin » suivi à travers un algorithme est appelé le flux d'instructions (figure 2.6), et les constructions qui le modifient sont appelées des instructions de contrôle de flux. On exécute normalement les instructions de la première à la dernière, sauf lorsqu'on rencontre une instruction de contrôle de flux : de telles instructions vont permettre de suivre différents chemins suivant les circonstances. C'est en particulier le cas de l'instruction conditionnelle qui n'exécute une instruction que sous certaines conditions préalables. Nous distinguerons ici 3 variantes d'instructions conditionnelles (figure 2.7) :

Généralité sur le langage

Contrôle de flux: test

Fig. 2.6 : FLUX D'INSTRUCTIONS

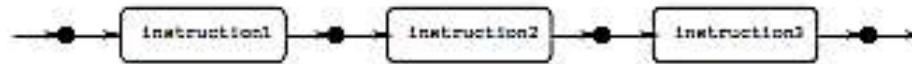


Fig. 2.7 : DÉFINITION DE L'ACADEMIE (6)

ALTERNATIVE *n. f. XV^e siècle, comme terme de droit ecclésiastique ; XVII^e siècle, au sens moderne. Forme féminine substantivée d'alternatif. Choix nécessaire entre deux propositions, deux attitudes dont l'une exclut l'autre.*

Algorithme et langage Python

Gilles VERONIE

Généralité sur le langage

Contrôle de flux: test

Instructions conditionnelles	
test simple	<code>if condition : blocIf</code>
alternative simple	<code>if condition : blocIf else : blocElse</code>
alternative multiple	<code>if condition : blocIf elif condition1 : blocElif1 elif condition2 : blocElif2 ... else : blocElse</code>

où `if`, `else` et `elif` sont des mots réservés, `condition` une expression booléenne (à valeur `True` ou `False`) et `bloc...` un bloc d'instructions.

Généralité sur le langage

Contrôle de flux: test simple définition

Le test simple est une instruction de contrôle du flux d'instructions qui permet d'exécuter une instruction sous condition préalable.

La condition évaluée après l'instruction « if » est donc une expression booléenne qui prend soit la valeur False (faux) soit la valeur True (vrai). Elle peut contenir les opérateurs de comparaison suivants :

x == y	# x est égal à y
x != y	# x est différent de y
x > y	# x est plus grand que y
x < y	# x est plus petit que y
x >= y	# x est plus grand que, ou égal à y
x <= y	# x est plus petit que, ou égal à y

Généralité sur le langage

Contrôle de flux: test simple

Mais certains problèmes exigent parfois de formuler des conditions qui ne peuvent pas être exprimées sous la forme d'une simple comparaison. Par exemple, la condition $x \in [0, 1[$ s'exprime par la combinaison de deux conditions $x \geq 0$ et $x < 1$ qui doivent être vérifiées en même temps. Pour combiner ces conditions, on utilise les opérateurs logiques `not`, `and` et `or` (figure 2.9). Ainsi la condition $x \in [0, 1[$ pourra s'écrire en PYTHON : `(x >= 0) and (x < 1)`.

Fig. 2.9 : PRINCIPAUX OPÉRATEURS PYTHON

Opérateurs logiques : `not a`, `a and b`, `a or b`

Opérateurs de comparaison : `x == y`, `x != y`,
`x < y`, `x <= y`, `x > y`, `x >= y`

Opérateurs arithmétiques : `+x`, `-x`, `x + y`,
`x - y`, `x * y`, `x / y`, `x % y`, `x**y`

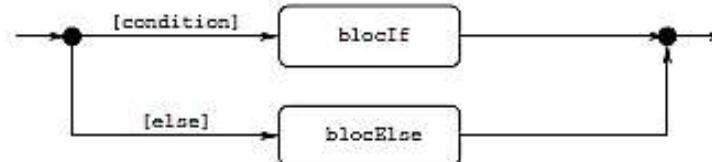
Généralité sur le langage

Contrôle de flux: alternative simple, définition:

L'alternative simple est une instruction de contrôle du flux d'instructions qui permet de choisir entre deux instructions selon qu'une condition est vérifiée ou non.

Fig. 2.11 : AIGUILAGE « if ... else »

```
if condition : blocIf  
else : blocElse
```



L'étiquette [condition] signifie qu'on passe par la voie correspondante si la condition est vérifiée (True), sinon on passe par la voie étiquettée [else].

Algorithme et langage Python

Gilles VERONIE

Généralité sur le langage

Contrôle de flux: alternative multiple, définition:

L'alternative multiple est une instruction de contrôle du flux d'instructions qui permet de choisir entre plusieurs instructions en cascadant des alternatives simples.

IF

ELIF

ELSE

Généralité sur le langage

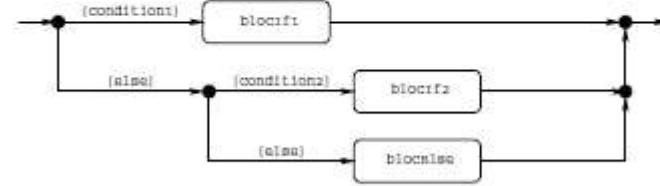
Contrôle de flux: alternative multiple

Aiguillage en cascade

Fig. 2.12 : « if ... else » IMBRIQUÉS

```
if condition1 : blocIf1
else :
```

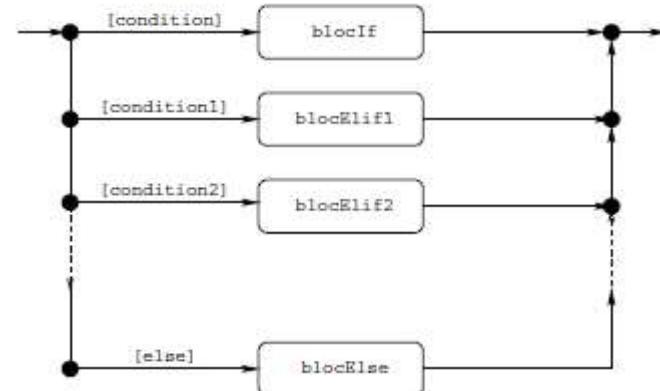
```
    if condition2 : blocIf2
    else : blocElse
```



Afin de simplifier l'écriture des tests imbriqués on peut contracter le « else: if » en elif et obtenir un algorithme plus compact fig. 2.13.

Fig. 2.13 : L'ALTERNATIVE MULTIPLE

```
if condition : blocIf
elif condition1 : blocElif1
elif condition2 : blocElif2
...
else : blocElse
```



L'alternative multiple ci-dessus est équivalente à un ensemble d'alternatives simples imbriquées :

```
if condition : blocIf
else :
    if condition1 : blocElif1
    else :
        if condition2 : blocElif2
        else :
            ...
            else : blocElse
```

Généralité sur le langage

Contrôle de flux: instructions itératives (boucles).

1. Comment éviter de répéter explicitement plusieurs fois de suite la même séquence d'instructions ?
2. Comment éviter de savoir à l'avance combien de fois il faut répéter la séquence pour obtenir le bon résultat ?

De nouvelles instructions de contrôle de flux sont introduites pour répondre à ces questions : les instructions itératives. On parle également de boucles, de répétitions ou encore d'itérations (figure 2.14). Nous distinguerons 2 variantes d'instructions itératives :

Instructions itératives	
itération conditionnelle	<code>while condition : blocWhile</code>
parcours de séquence	<code>for element in sequence : blocFor</code>

où `while`, `for` et `in` sont des mots réservés, `condition` une expression booléenne (à valeur `True` ou `False`), `element` un élément de la séquence `sequence` et `bloc...` un bloc d'instructions.

Généralité sur le langage

Contrôle de flux: instructions itératives (boucles): définitions.

Fig. 2.14 : DÉFINITION DE L'ACADEMIE (7)

ITÉRATION *n. f. XVe siècle. Emprunté du latin iteratio, « répétition, redite ». Répétition. MATH. Répétition d'un calcul avec modification de la variable, qui permet d'obtenir par approximations successives un résultat satisfaisant.*

L'itération conditionnelle est une instruction de contrôle du flux d'instructions qui permet sous condition préalable de répéter zéro ou plusieurs fois la même instruction.

Généralité sur le langage

Contrôle de flux: instructions itératives conditionnelles.

Itération conditionnelle

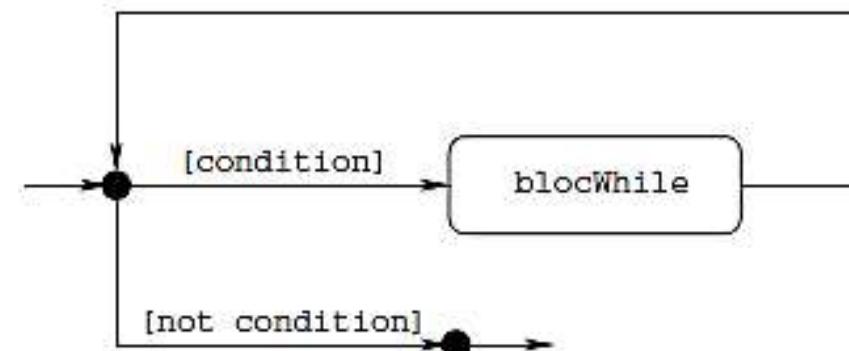
L'instruction « `while` » permet de répéter plusieurs fois une même instruction (figure 2.15) : le bloc d'instructions `blocWhile` est exécuté tant que (`while`) la condition est vérifiée. On arrête dès que la condition est fausse ; on dit alors qu'on « sort » de la boucle.

On commence par tester la condition ; si elle est vérifiée, on exécute le bloc d'instructions `blocWhile` (encore appelé le « corps » de la boucle) puis on reteste la condition : la condition est ainsi évaluée avant chaque exécution du corps de la boucle ; si la condition est à nouveau vérifiée on réexécute le bloc d'instructions `blocWhile` (on dit qu'on « repasse » dans la boucle) et ainsi de suite jusqu'à ce que la condition devienne fausse, auquel cas on « sort » de la boucle.

Généralité sur le langage

Contrôle de flux: instructions itératives conditionnelles.

Fig. 2.15 : BOUCLE while
while condition : blocWhile



Généralité sur le langage

Contrôle de flux: instructions itératives conditionnelles. (**séquence**)

Il est fréquent de manipuler des suites ordonnées de caractères

exemple: `s = "123"` chaîne.

Les tableaux

Exemple: `t = [1,2,3]`

les n-uplets

Exemple: `u = 1,2,3`

Chaque élément d'une séquence est accessible par son rang dans la séquence grâce à l'opérateur « crochets » : `sequence[rang]` (exemples : `s[1]`, `t[2]` ou `u[0]`) et par convention, le premier élément d'une séquence a le rang 0 (exemples : `s[1]` est le 2^{ème} élément de la chaîne `s`, `t[2]` le 3^{ème} élément du tableau `t` et `u[0]` le 1^{er} élément du n-uplet `u`).

```
>>> s = "123"  
>>> s[1]  
'2'
```

```
>>> t = [1,2,3]  
>>> t[2]  
3
```

```
>>> u = 1,2,3  
>>> u[0]  
1
```

Généralité sur le langage

Contrôle de flux: instructions itératives conditionnelles. (**séquence**)

Définition

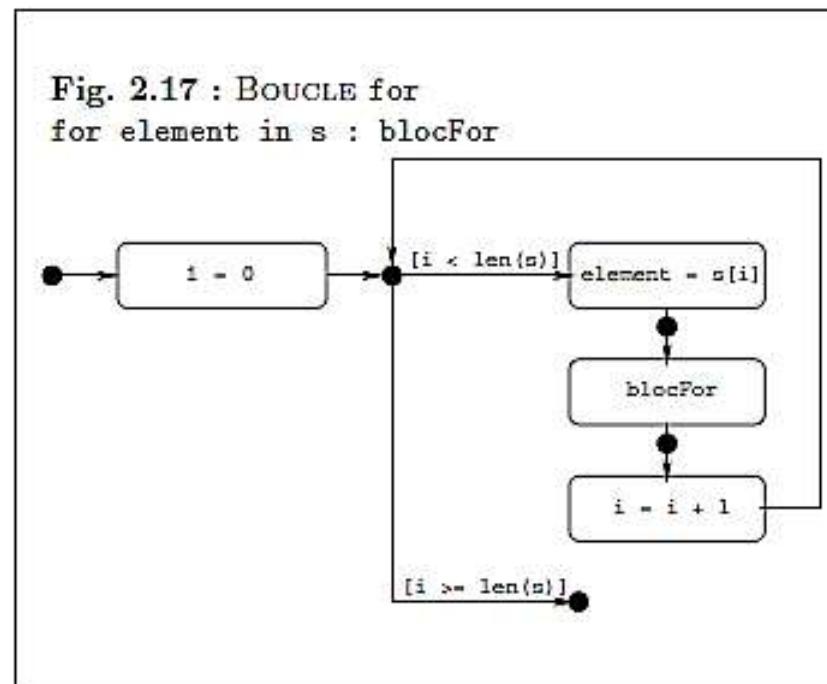
Une séquence est une suite ordonnée d'éléments, éventuellement vide, accessibles par leur rang dans la séquence.

Les principales opérations sur les séquences sont listées dans le tableau ci-dessous (d'après [10]).

Operation	Result
<code>x in s</code>	True if an item of s is equal to x, else False
<code>x not in s</code>	False if an item of s is equal to x, else True
<code>s1 + s2</code>	the concatenation of s1 and s2
<code>s * n, n*s</code>	n copies of s concatenated
<code>s[i]</code>	i'th item of s, origin 0
<code>s[i : j]</code>	Slice of s from i (included) to j(excluded). Optional step value, possibly negative (default : 1).
<code>s[i : j :step]</code>	
<code>len(s)</code>	Length of s
<code>min(s)</code>	Smallest item of s
<code>max(s)</code>	Largest item of s
<code>range([start,] end [, step])</code>	Returns list of ints from \geq start and $<$ end. With 1 arg, list from 0..arg-1 With 2 args, list from start..end-1 With 3 args, list from start up to end by step

Généralité sur le langage

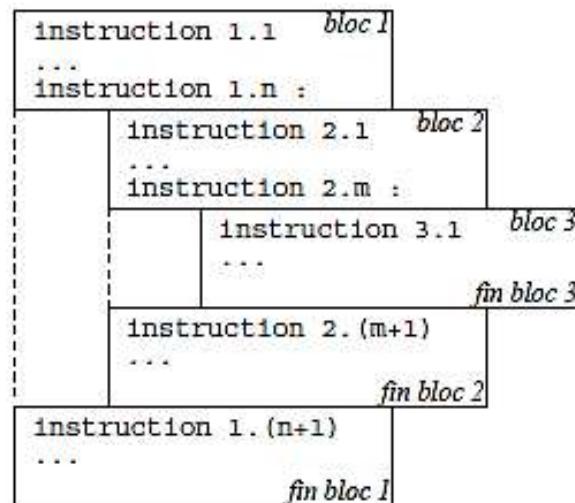
Contrôle de flux: instructions itératives conditionnelles. (**séquence**)



Généralité sur le langage

Contrôle de flux: instructions itératives conditionnelles. (**Boucles**)

Fig. 2.18 : BLOCS D'INSTRUCTIONS

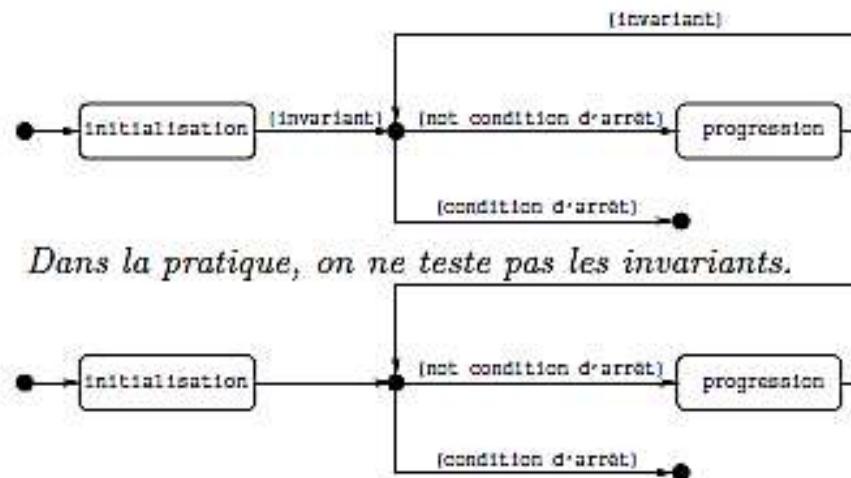


Généralité sur le langage

Contrôle de flux: instructions itératives conditionnelles. (**Invariant de boucles**)

Définition: un invariant de boucle est une propriété vérifiée tout au long de l'exécution de la boucle.

Fig. 2.20 : INVARIANT DE BOUCLE



Généralité sur le langage

Procédures et fonctions: réutiliser

Réutiliser: exemple numérotation en base b

Un entier positif en base b est représenté par une suite de chiffres $(r_n r_{n-1} \dots r_1 r_0)_b$ où les r_i sont des chiffres de la base b ($0 \leq r_i < b$). Ce nombre a pour valeur :

$$r_n b^n + r_{n-1} b^{n-1} + \dots + r_1 b^1 + r_0 b^0 = \sum_{i=0}^{i=n} r_i b^i$$

Exemples :

$(123)_{10}$	$=$	$1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$	$=$	$(123)_{10}$
$(123)_5$	$=$	$1 \cdot 5^2 + 2 \cdot 5^1 + 3 \cdot 5^0$	$=$	$(38)_{10}$
$(123)_8$	$=$	$1 \cdot 8^2 + 2 \cdot 8^1 + 3 \cdot 8^0$	$=$	$(83)_{10}$
$(123)_{16}$	$=$	$1 \cdot 16^2 + 2 \cdot 16^1 + 3 \cdot 16^0$	$=$	$(291)_{10}$

Fig. 3.1 : RÉUTILISABILITÉ D'UN ALGORITHME

La réutilisabilité d'un algorithme est son aptitude à être réutilisé pour résoudre des tâches équivalentes à celle pour laquelle il a été conçu.

Généralité sur le langage

Procédures et fonctions: réutiliser

Sous l'interpréteur PYTHON, pour calculer successivement la valeur décimale n des nombres $(123)_5$ et $(123)_8$, nous devrons donc recopier 2 fois l'algorithme ci-dessus.

```
>>> b = 5
>>> code = [1,2,3]
>>> n = 0
>>> for i in range(len(code)):
...     n = n + code[i]*b**(len(code)-1-i)
...
>>> n
38
```

```
>>> b = 8
>>> code = [1,2,3]
>>> n = 0
>>> for i in range(len(code)):
...     n = n + code[i]*b**(len(code)-1-i)
...
>>> n
83
```

Généralité sur le langage

Procédures et fonctions: encapsuler

Fig. 3.2 : ENCAPSULATION

L'encapsulation est l'action de mettre une chose dans une autre : on peut considérer que cette chose est mise dans une « capsule » comme on conditionne un médicament dans une enveloppe soluble (la capsule). Ici, il s'agit d'encapsuler des instructions dans une fonction ou une procédure.



DIVISER
POUR
MIEUX
REGNER

Généralité sur le langage

Procédures et fonctions: structurer

Exemple de l'écriture fractionnaire

Un nombre fractionnaire (nombre avec des chiffres après la virgule : $(r_n r_{n-1} \dots r_1 r_0.r_{-1} r_{-2} \dots)_b$) est défini sur un sous-ensemble borné, incomplet et fini des rationnels. Un tel nombre a pour valeur :

$$r_n b^n + r_{n-1} b^{n-1} + \dots + r_1 b^1 + r_0 b^0 + r_{-1} b^{-1} + r_{-2} b^{-2} + \dots$$

En pratique, le nombre de chiffres après la virgule est limité par la taille physique en machine.

$$(r_n r_{n-1} \dots r_1 r_0.r_{-1} r_{-2} \dots r_{-k})_b = \sum_{i=-k}^{i=n} r_i b^i$$

Un nombre x pourra être représenté en base b par un triplet $[s, m, p]$ tel que $x = (-1)^s \cdot m \cdot b^p$ où s représente le signe de x , m sa mantisse et p son exposant (p comme puissance) où :

- signe s : $s = 1$ si $x < 0$ et $s = 0$ si $x \geq 0$
- mantisse m : $m \in [1, b[$ si $x \neq 0$ et $m = 0$ si $x = 0$
- exposant p : $p \in [\min, \max]$

Généralité sur le langage

Procédures et fonctions: structurer

4 étapes à suivre pour coder une valeur x en base n :

Ainsi, le codage de $x = -9.75$ en base $b = 2$ s'effectuera en 4 étapes :

1. coder le signe de x : $x = -9.75 < 0 \Rightarrow s = 1$
2. coder la partie entière de $|x|$: $9 = (1001)_2$
3. coder la partie fractionnaire de $|x|$: $0.75 = (0.11)_2$
4. et coder $|x|$ en notation scientifique normalisée : $m \in [1, 2[$
$$(1001)_2 + (0.11)_2 = (1001.11)_2 = (1.00111)_2 \cdot 2^3$$
$$= (1.00111)_2 \cdot 2^{(11)}_2$$

Généralité sur le langage

Procédures et fonctions: encapsuler

CALCUL DE $\sin(\pi/2)$

```
>>> from math import sin, pi
>>> sin(pi/2)
1.0
>>> y = sin(pi/2)
>>> y
1.0
```

Cette petite session PYTHON illustre quelques caractéristiques importantes des fonctions.

- Une fonction à un nom : `sin`.
- Une fonction est en général « rangée » dans une bibliothèque de fonctions (ici la bibliothèque `math` de PYTHON, figure 3.3) ; il faut aller la chercher (on « importe » la fonction) :
`from math import sin`.
- Une fonction s'utilise (s'« appelle ») sous la forme d'un nom suivi de parenthèses ; dans les parenthèses, on « transmet » à la fonction un ou plusieurs arguments : `sin(pi/2)`.
- L'évaluation de la fonction fournit une valeur de retour ; on dit aussi que la fonction « renvoie » ou « retourne » une valeur (`sin(pi/2) → 1.0`) qui peut ensuite être affectée à une variable : `y = sin(pi/2)`.

Généralité sur le langage

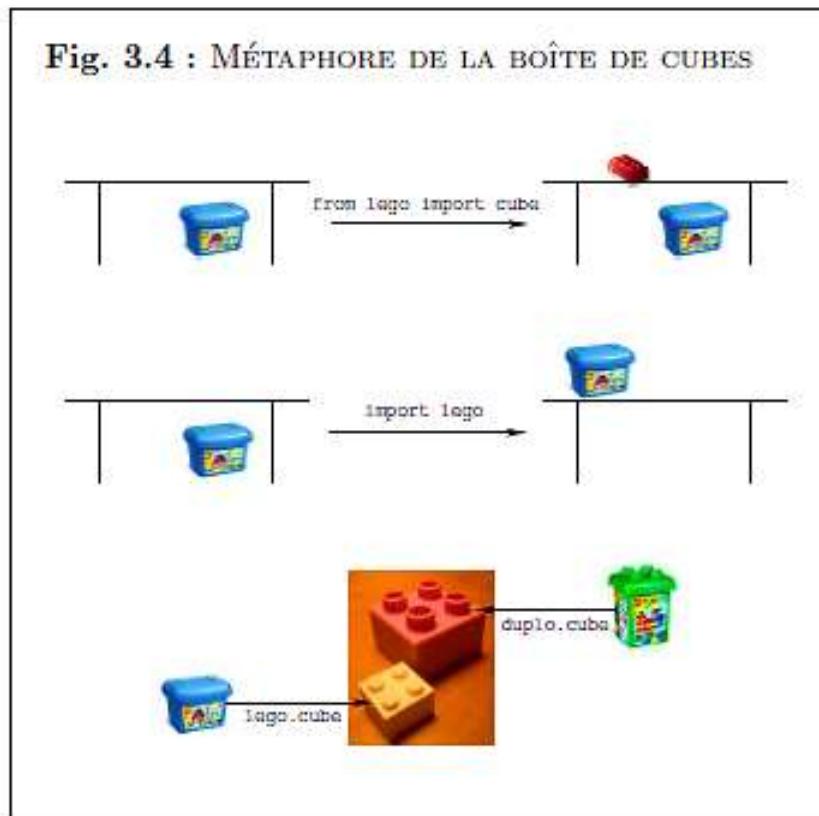
Procédures et fonctions: fonctions

Pour encapsuler un algorithme dans une fonction, on suivra pas à pas la démarche suivante :

1. donner un nom explicite à l'algorithme,
2. définir les paramètres d'entrée-sortie de l'algorithme,
3. préciser les préconditions sur les paramètres d'entrée,
4. donner des exemples d'utilisation et les résultats attendus,
5. décrire par une phrase ce que fait l'algorithme et dans quelles conditions il le fait,
6. encapsuler l'algorithme dans la fonction spécifiée par les 5 points précédents.

Généralité sur le langage

Procédures et fonctions: nommer



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

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

Généralité sur le langage

Procédures et fonctions: fonctions_nommer

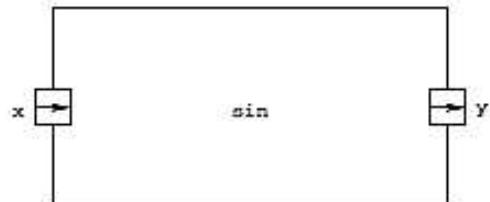
```
def fibonacci():
    return
>>> from fibo import fibonacci
>>> fibonacci()
>>>
```

Le code de la partie gauche est la définition actuelle de la fonction fibonacci que l'on a éditée dans un fichier fibo.py : le module associé a donc pour nom fibo. La partie droite montre comment on utilise couramment le module fibo et la fonction fibonacci sous l'interpréteur PYTHON. Dans l'état actuel, cette fonction n'a pas d'arguments et ne fait rien ! Mais elle est déjà compilable et exécutable. On peut la décrire de la manière suivante : « La fonction fibonacci calcule un nombre de Fibonacci ».

Généralité sur le langage

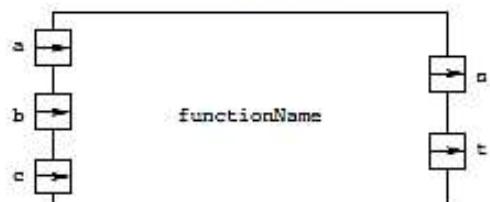
Procédures et fonctions: fonctions_paramétrer

Fig. 3.6 : PARAMÈTRES D'UNE FONCTION



sin est le nom de la fonction ; x est le seul paramètre d'entrée, y le seul paramètre de sortie.

$y = \sin(x)$



functionName est le nom de la fonction ; a, b et c sont les 3 paramètres d'entrée, s et t les 2 paramètres de sortie.

$(s,t) = \text{functionName}(a,b,c)$

Généralité sur le langage

Procédures et fonctions: fonctions_ paramétriser

Définition 3.5 : PARAMÈTRE D'ENTRÉE

Les paramètres d'entrée d'une fonction sont les arguments de la fonction qui sont nécessaires pour effectuer le traitement associé à la fonction.

Définition 3.6 : PARAMÈTRE DE SORTIE

Les paramètres de sortie d'une fonction sont les résultats retournés par la fonction après avoir effectué le traitement associé à la fonction.

```
def fibonacci(n):          >>> from fibo import fibonacci
    u = 0                  >>> fibonacci(5)
    return u                0
                                >>> fibonacci(-5)
                                0
                                >>> fibonacci('n')
                                0
```

Généralité sur le langage

Procédures et fonctions: fonctions_préconditionner

Définition 3.7 : PRÉCONDITION

Les préconditions d'une fonction sont les conditions que doivent impérativement vérifier les paramètres d'entrée de la fonction juste avant son exécution.

Définition 3.8 : POSTCONDITION

Les postconditions d'une fonction sont les conditions que doivent impérativement vérifier les paramètres de sortie de la fonction juste après son exécution.

En plus des préconditions et des postconditions, on pourra quelquefois imposer que des conditions soient vérifiées tout au long de l'exécution de la fonction : on parle alors d'invariants.

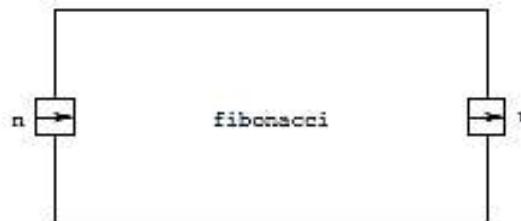
Définition 3.9 : INVARIANT

Les invariants d'une fonction sont les conditions que doit impérativement vérifier la fonction tout au long de son exécution.

Généralité sur le langage

Procédures et fonctions: fonctions_ paramétrier

Fig. 3.7 : FONCTION fibonacci (1)



fibonacci est le nom de la fonction; *n* est le seul paramètre d'entrée, *u* le seul paramètre de sortie.

`u = fibonacci(n)`

Fig. 3.11 : FONCTION fibonacci (2)



n :int est le paramètre d'entrée de nom *n* et de type int qui doit vérifier la précondition $[n \geq 0]$.
u :int est le paramètre de sortie de nom *u* et de type int.

`u = fibonacci(n)`

Généralité sur le langage

Procédures et fonctions: fonctions_protéger

```
def fibonacci(n):                                >>> fibonacci(-5)
    assert type(n) is int                         Traceback ...
    assert n >= 0                                 assert n >= 0
    u = 0                                         AssertionError
    return u                                     >>> fibonacci('n')
                                                Traceback ...
                                                assert type(n) is int
                                                AssertionError
```

Fig. 3.9 : L'INSTRUCTION assert EN PYTHON
assert expr[, message]

expr est évaluée : si *expr* == True, on passe à l'instruction suivante, sinon l'exécution est interrompue et une exception AssertionError est levée qui affiche le message optionnel.

Exemple :

```
>>> assert False, "message d'erreur"
Traceback (most recent call last) :
  File "<stdin>", line 1, in <module>
AssertionError : message d'erreur
>>>
```

Généralité sur le langage

Fonctions

Exemple suite de fibonacci :

```
>>> def fib(n):      # Ecrire la suite de fibonacci jusqu'a n
...     """Affiche la suite de Fibonacci jusqu'a n."""
...     a, b = 0, 1
...     while a < n:
...         print a,
...         t = a+b
...         a = b
...         b = t
...         # ou bien : a, b = b, a+b
...
...
>>> # Maintenant, appelons la fonction qu'on vient de definir :
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Généralité sur le langage

Paramètres

Les paramètres peuvent être nommés.

```
>>> def perroquet(voltage, etat='raide', action='wouff'):
...     print "-- Ce perroquet ne va pas faire", action
...     print "si vous le branchez sur du", voltage, "volts."
...     print "Il a l'air", etat, "!"
...
>>> perroquet(voltage = "quatre millions", etat = "sacrement fichu",
...             action = "SCHLACK")
-- Ce perroquet ne va pas faire SCHLACK
si vous le branchez sur du quatre millions volts.
Il a l'air sacrement fichu !
```

Généralité sur le langage

Retour de valeurs

Retour d'une liste des valeurs de la suite de Fibonacci

```
>>> def fib2(n): # renvoie la suite de Fibonacci jusqu'à n
...     resultat = []
...     a, b = 0, 1
...     while b < n:
...         resultat.append(b)
...         a, b = b, a+b
...     return resultat
...
>>> # Maintenant, appelons la fonction qu'on vient de définir :
... a = fib2(2000)
>>> print a
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597]
```

Généralité sur le langage

Chaînes de documentations

Attribut `__doc__`

```
>>> def ma_fonction():
...     """Ne rien faire, mais le documenter.
...
...     Non, vraiment, ne rien faire.
...
...     """
...
...     pass
...
>>> print ma_fonction.__doc__
Ne rien faire, mais le documenter.
```

Non, vraiment, ne rien faire.

Généralité sur le langage

Documentation en ligne

```
>>> help(ma_fonction)

Help on function ma_fonction in module __main__:

ma_fonction()
    Ne rien faire, mais le documenter.
    Non, vraiment, ne rien faire.
```

Structures de données

Listes

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[3]
>>> a
[-1, 1, 66.25, 333, 1234.5]
```

Structures de données

Tuples et séquences

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Les tuples ne sont pas modifiables :
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # Comme dans : a, b = 1, 2
>>> t = 1, 2
>>> a, b = t
>>> a
1
>>> b
2
```

Structures de données

Types

■ `type(x)`

```
>>> type(3.14)
<type 'float'>
>>> type('3.14')
<type 'str'>
>>> type(print)
File "<stdin>", line 1
    type(print)
    ^
SyntaxError: invalid syntax
```

■ `isinstance()`

```
>>> isinstance('3.14', float)
False
>>> isinstance('3.14', str)
True
```

Structures de données

Egalités

- égalité : ==
- identité : is

```
>>> '3.14' == '3.14'  
True  
>>> '3.14' is '3.14'  
True  
>>> a = ['3.14']  
>>> b = ['3.14']  
>>> a == b  
True  
>>> a is b  
False
```

Structures de données

Les variables

nom	exemple	signification
int	x = 0	un nombre entier
float	x = 1.5	un nombre décimal
string	x = "du texte"	une chaîne de caractères
unicode	x = u"écris"	une chaîne de caractère unicode
list	x = [1,2,3,4,5]	une liste de variables
tuple	x = (1,2,3,4,5)	une liste de variables non modifiable
set	x = set((1,2,3,4,5))	une liste de variables uniques, ordonnées
dictionnaire	x = {10:1,20:2}	un tableau associatif
bool	x = True	un VRAI ou FAUX

Structures de données

Les opérations

Operation	example	resultat	notes
addition	10 $+$ 5	15	rien
soustraction	10 $-$ 5	5	rien
multiplication	10 $*$ 5	50	rien
exposant	10 ** 2		attention, taille des variables !
division	10 // 5	2	attention c'est une division entière
division avec virgule	10 / 5.0	2.0	un des deux doit être float
modulo	10 \% 5	0	reste de la division entière

Structures de données

Les opérateurs de comparaison

symbole	signification	exemple	résultat
<code>==</code>	est égal à	<code>1 == 2</code>	False
<code>!=</code>	n'est pas égal à	<code>1 != 2</code>	True
<code><</code>	est inférieur	<code>1 < 2</code>	True
<code>></code>	est supérieur	<code>1 > 2</code>	False
<code>>=</code>	est supérieur ou =	<code>1 >= 2</code>	False
<code><=</code>	est inférieur ou =	<code>1 <= 2</code>	True

Structures de données

Console

```
view plain copy to clipboard print ?  
  
# afficher dans la console  
print "une valeur"  
  
# on utilise , pour "print" plusieurs variables à la fois  
print "x =", 1  
  
# demander un mot à l'utilisateur  
mot = raw_input("entrez un mot :")  
print mot  
  
# empêcher la console de se refermer immédiatement  
print "pouvez vous lire ce texte ?"  
  
raw_input() # l'utilisateur doit presser la touche entrée
```

Structures de données

Console

L'incrémentation : changer la valeur d'une variable en fonction de sa valeur initiale.

```
view plain copy to clipboard print ?  
  
x += 1 # x est augmenté de 1  
x -= 1 # x est diminué de 1  
x *= 1 # x est multiplié par 1  
x /= 1 # x est divisé par 1  
x %= 1 # x devient le résultat de la division euclidienne de x par 1
```

Les conditions : si une comparaison est vrai ou fausse, on fait...

```
view plain copy to clipboard print ?  
  
if x == 2:  
    print "x est égal à 2"  
elif x < 40:  
    print "x est plus petit que 40"  
else:  
    print "autre chose"
```

Algorithme et langage Python

Gilles VERONIE

Structures de données

Console

L'indentation : pour expliquer qu'un bout de code doit être exécuté après une condition, boucle,... python utilise des espaces.

```
view plain copy to clipboard print ?  
  
if condition == True:  
    Faire quelque chose # une instruction apres une condition -> 1 espace  
        if condition2 == True: # meme condition, meme nombre d'espaces -> 1 espace  
            Faire autre chose # une nouvelle condition imbriquee -> 2 espace
```

Les boucles : permettent d'effectuer des opérations un certain nombre de fois, (TANT QUE...)

```
view plain copy to clipboard print ?  
  
x = 0  
while x < 100:  
    print x += 1
```

Structures de données

Console

L'itération : obtenir chaque élément d'une liste séparément, dans une variable

```
view plain copy to clipboard print ?  
  
for element in liste: # pour chaque élément de la liste  
    print element  
  
# Pour obtenir chaque index d'une liste, utilisez xrange.  
for index in xrange(liste):  
    print liste[index]  
  
# Pour obtenir chaque index et élément d'une liste, utilisez le générateur enumerate.  
for index, element in enumerate(liste):  
    print index, element
```

Structures de données

Console

les exceptions : gérer les erreurs qui peuvent survenir lors de l'exécution du programme.

```
view plain copy to clipboard print ?  
nombre = raw_input("entrez un nombre :")  
  
try:  
    nombre = int(nombre) # conversion dangereuse, peut generer une erreur  
except: # si une erreur survient  
    nombre = 0
```

Les classes

Définition et implémentation d'une classe

```
#début définition
class Personne:
    """Classe Personne"""

    #constructeur
    def __init__(self):
        #lister les champs
        self.nom = ""
        self.age = 0
        self.salaire = 0.0
    #fin constructeur

#fin définition
```

- **class** est un mot clé permettant de définir la structure
- **Personne** est le nom de la classe ici
- **"""Classe Personne"""** sert à documenter la classe
- Regarder le rôle des « : » et des **indentations**
- **__init__** est une méthode standard appelée **constructeur**, automatiquement appelée lors de l'instanciation
- **self** représente l'instance elle-même, **elle doit apparaître en première position dans la définition de toutes les méthodes**, mais il ne sera pas nécessaire de la spécifier lors de l'appel
- nous exploitons le constructeur pour énumérer les champs de la classe (pas obligatoire, mais bonne habitude à prendre), ils sont non typés
- le constructeur peut prendre des paramètres en entrée. Ex. initialisation des champs
- contrairement aux autres langages, un seul constructeur par classe seulement
- noter le rôle du « . » dans l'accès aux champs

Les classes

Instanciation et utilisation dans le programme principal

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

#appel du module
import ModulePersonne as MP

#instanciation
p = MP.Personne()

#affiche tous les membres de p
print(dir(p))

#affectation aux champs
p.nom = input("Nom : ")
p.age = int(input("Age : "))
p.salaire = float(input("Salaire : "))

#affichage
print(p.nom, ", ", p.age, ", ", p.salaire)
```

- Il faut d'abord **importer** le module contenant la classe, nous lui attribuons l'alias **MP** ici
- Pour la création de l'instance **p**, nous spécifions le module puis le nom de la classe
- Il y a **()** parce que c'est bien une méthode que nous appelons à la création : le constructeur
- Le paramètre **self** du constructeur n'est pas à spécifier sur l'instance
- Noter le rôle de « **.** » lors de l'accès aux champs de l'instance
- Les champs sont accessibles en lecture / écriture
- L'accès direct aux champs n'est pas conseillé en programmation objet, **mais on fera avec cette année** (sous Python, pour rendre un attribut privé, il faut mettre un double **_** devant le nom du champ et créer alors des accesseurs et des mutateurs, à voir en M1)

Les classes

Programmer des méthodes supplémentaires

```
#début définition
class Personne:
    """Classe Personne"""

#constructeur
def __init__(self):
    #lister les champs
    self.nom = ""
    self.age = 0
    self.salaire = 0.0
#fin constructeur

#saisie des infos
def saisie(self):
    self.nom = input("Nom : ")
    self.age = int(input("Age : "))
    self.salaire = float(input("Salaire : "))
#fin saisie
```

Nous avons implémenté 2 méthodes supplémentaires dans la classe.

Le programme principal s'en trouve grandement simplifié



```
# -*- coding: utf -*-
#appel du module
import ModulePersonne as MP

#instanciation
p = MP.Personne()

#saisie
p.saisie()

#méthode affichage
p.affichage()
```

Remarque : Noter le comportement de **self**

Algorithme et langage Python

Gilles VERONIE

Les Classes

Programmer des méthodes paramétrées

Rajouter la méthode `retraite()` qui calcule le nombre d'années avant l'âge `limite` de la retraite

```
#début définition
class Personne:
    """Classe Personne"""

    ...

    #reste avant retraite
    def retraite(self, limite):
        reste = limite - self.age
        if (reste < 0):
            print("Vous êtes à la retraite")
        else:
            print("Il vous reste %s années" % (reste))
    #fin retraite

#fin définition
```



Programme principal

```
# -*- coding: utf -*-
#appel du module
import ModulePersonne as MP
#instanciation
p = MP.Personne()
#saisie
p.saisie()
#méthode affichage
p.affichage()
#reste avant retraite
p.retraite(62)
```

```
>>>
Nom : Toto
Age : 51
Salaire : 1200
Son nom est Toto
Son âge : 51
Son salaire : 1200.0
Il vous reste 11 années ]->
>>> |
```

Saisie

Affichage

Retraite

Les Classes

Gérer une collection d'objets

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

#appel du module
import ModulePersonne as MP

#liste vide
liste = []

#nb. de pers ?
n = int(input("Nb de pers : "))

#saisie liste
for i in range(0,n):
    a = MP.Personne()
    a.saisie()
    liste.append(a)

#affichage
print("*** début affichage 1")
for p in liste:
    print("-----")
    p.affichage()
```

Créer l'objet référencé par `a`, effectuer la saisie. Ajouter la référence dans la `liste`.

Le typage est automatique, `p` est bien de type Personne.

Python propose des outils pour la gestion des collections d'objets hétérogènes (tuple, liste, dictionnaire). Ils sont opérationnels pour les instances de nos classes.

Exemple d'exécution

```
Nb de pers : 2
Nom : Toto
Age : 35
Salaire : 1200
Nom : Tata
Age : 36
Salaire : 3000
*** début affichage 1
-----
Son nom est Toto
Son âge : 35
Son salaire : 1200.0
-----
Son nom est Tata
Son âge : 36
Son salaire : 3000.0
```

Algorithme et langage Python

Gilles VERONIE

Les Classes

Accès indicé et modification des objets

```
#accès par numéro
numero = int(input("N° ind. à traiter :"))
if (numero < len(liste)) :
    b = liste[numero]
    b.salaire = b.salaire * 2
    #affichage de nouveau
    print("xxx début affichage 2")
    for p in liste:
        print("-----")
        p.affichage()
else:
    print("indice non valable")
```

Vérifier toujours que l'indice utilisé est valide.

Récupération de la référence de l'objet, indice = numéro

Accès au champ `salaire` de l'objet référencé par `b`.

`liste[numero].salaire = ...`
fonctionne également !

Exemple d'exécution

```
N° ind. à traiter :1
xxx début affichage 2
-----
Son nom est  Toto
Son âge :  35
Son salaire :  1200.0
-----
Son nom est  Tata
Son âge :  36
Son salaire :  6000.0
```

Les Classes

Collection d'objets -Commentaires

- Les collections de nos objets (instances de nos classes) ouvre la porte à une programmation organisée et efficace
- Leur intérêt décuplera lorsque nous étudierons les fichiers
- Il est judicieux d'élaborer une classe dédiée à la gestion de la collection (où la liste serait un champ de la classe)
- Les objets de la collection peuvent être différents. Quand ce sont des instances de classes héritières du même ancêtre, on parle de liste polymorphe
- Le gestionnaire de collection peut être un dictionnaire, le mécanisme « clé – valeur » (« clé – objet » en l'occurrence pour nous) ouvre des possibilités immenses (ce mécanisme est très en vogue, ex. bases NoSQL)

Les Classes (héritage)

Classe mère

Script :

```
import string
class Alphabet_majuscules (object):
    # classe mère
    def __init__(self):
        self.lettres = string.ascii_uppercase
    # équivalent à une chaîne reprenant tout l'alphabet avec la méthode upper ()
```

Commentaires :

class Alphabet_majuscules (object): vous construisez votre classe mère comme une classe normale et ordinaire !

self.lettres = string.ascii_uppercase importation de la constante "string.ascii_uppercase" du module "string" qui est l'alphabet en majuscules.

Les Classes (héritage)

Classe fille

```
class Alphabet_minuscules (Alphabet_majuscules):
    # classe fille
    def __init__(self):
        Alphabet_majuscules.__init__(self)
        self.lettres = self.lettres.lower ()
```

class Alphabet_minuscules (Alphabet_majuscules): en paramètre de la classe fille, le nom de la classe mère.

Alphabet_majuscules.__init__(self) dans le constructeur de la classe fille, importation du constructeur de la classe mère avec ses attributs.

en paramètre de la classe fille, le nom de la classe mère.

dans le constructeur de la classe fille, importation du constructeur de la classe mère avec ses attributs.

Démonstration de l'importation du constructeur de la classe mère dans le constructeur de la classe fille.

Les attributs de l'objet de la classe mère peuvent être les attributs de l'objet de la classe fille :

Script modifié :

Les Classes (héritage)

Classe fille script:

```
import string
class Alphabet_majuscules (object):
    # classe mère
    def __init__(self):
        self.lettres_maj = string.ascii_uppercase
        # équivalent à une chaîne reprenant tout l'alphabet avec la méthode upper ()

class Alphabet_minuscules (Alphabet_majuscules):
    # classe fille
    def __init__(self):
        Alphabet_majuscules.__init__(self)
        self.lettres_min = self.lettres_maj.lower ()
```

Exécution :

```
>>> test = Alphabet_majuscules ()
>>> test.lettres_maj
'ABCDEFGHIJKLMNPQRSTUVWXYZ'
>>> test2 = Alphabet_minuscules ()
>>> test2.lettres_min
'abcdefghijklmnopqrstuvwxyz'
>>> test2.lettres_maj
'ABCDEFGHIJKLMNPQRSTUVWXYZ'
```

L'attribut de l'objet de la classe mère **test** peut être l'attribut de l'objet de la classe fille **test2** :

test.lettres_maj et **test2.lettres_maj**

Algorithme et langage Python

Gilles VERONIE

Les Classes (héritage)

Classe petite-fille :

```
import string

class Alphabet_majuscules (object):
    # classe mère
    def __init__(self):
        self.lettres = string.ascii_uppercase
        # équivalent à une chaîne reprenant tout l'alphabet en minuscules avec la méthode upper ()
        # ou plus simplement une chaîne avec tout l'alphabet en majuscules

class Alphabet_minuscules (Alphabet_majuscules):
    # classe fille
    def __init__(self):
        Alphabet_majuscules.__init__(self)
        self.lettres = self.lettres.lower ()

class Alphabet_tri (Alphabet_minuscules):
    # classe petite-fille
    def __init__(self):
        Alphabet_minuscules.__init__(self)
        self.voyelles = []
        self.consonnes = []
        for lettre in self.lettres :
            if lettre in "aeiouy":
                self.voyelles.append (lettre)
            else :
                self.consonnes.append (lettre)

    def listes_vers_chaines (self) :
        self.voyelles_chaine = "".join (self.voyelles)
        self.consonnes_chaine = "".join (self.consonnes)

On a ajouté la classe “petite-fille” de Alphabet_majuscules () et “fille” de Alphabet_minuscules () dans laquelle on a créé une méthode listes_vers_chaines () en plus du constructeur.
```

Algorithme et langage Python

Gilles VERONIE

Les Classes (héritage)

Classe petite-fille exécution:

```
>>> test = Alphabet_majuscules ()
>>> test.lettres
'ABCDEFGHIJKLMNPQRSTUVWXYZ'
>>> test2 = Alphabet_minuscules ()
>>> test2.lettres
'abcdefghijklmnopqrstuvwxyz'
>>> test3 = Alphabet_tri ()
>>> test3.lettres
'abcdefghijklmnopqrstuvwxyz'
>>> test3.voyelles
['a', 'e', 'i', 'o', 'u', 'y']
>>> test3.consonnes
['b', 'c', 'd', 'f', 'g', 'h', 'j', 'k', 'l', 'm', 'n', 'p', 'q', 'r', 's', 't', 'v', 'w', 'x', 'z']
>>> test3.listes_vers_chaines ()
>>> test3.voyelles_chaine
'aeiouy'
>>> test3.consonnes_chaine
'bcdfghjklmnnpqrstvwxz'
```

Exercices de programmation en Python

1) Test d'appartenance

Ecrire une fonction `mem` qui étant donnés une liste a et un objet x renvoie True ssi x est un élément de la liste.

2) Suite définie par une relation de récurrence

Ecrire une fonction `suite` qui étant donnés deux arguments n et a renvoie la valeur de u_n , où la suite $(u_k)_{k \in \mathbb{N}}$ est la suite définie par $u_0 = 1$ et $\forall k \in \mathbb{N}, u_{k+1} = \frac{1}{2} \left(u_k + \frac{a}{u_k} \right)$.

3) Maximum de deux entiers et maximum d'une liste (non vide)

- Ecrire une fonction `maximum` qui étant donnés deux entiers renvoie leur valeur maximale.
- Ecrire une fonction `max` qui étant donnée une liste non vide d'entiers renvoie leur valeur maximale.

Par exemple, `print(max([1, 3, 5, 1]))` affiche 5.

Remarque : Cette fonction `max` est en fait déjà définie en Python, avec la syntaxe `max(iterable)`. Elle est aussi valable pour tous les itérables (tuples, chaînes, dictionnaires, etc ...).

4) Nombre maximum de zéros (consécutifs) dans une liste

On suppose que la fonction maximum définie dans l'exercice précédent est connue.

- a) Ecrire une fonction nombre qui étant donnée une liste renvoie le nombre de zéros dans la liste.

Par exemple, `print(nombre([1,0,2,0,0,4]))` renvoie 3.

- b) Ecrire une fonction consec qui étant donnée une liste renvoie le nombre maximum de zéros consécutifs dans la liste.

Par exemple, `print(nombre([1,0,2,0,0,4]))` renvoie 2.

5) Tri par sélection

- a) Ecrire une fonction `indMin` on renvoie l'indice du minimum (en renvoyant le plus petit s'il en existe plusieurs). Par exemple, `indMin([5,2,3,4,2])` renvoie 1 (les éléments sont indicés à partir de 0).

- b) Ecrire une procédure `echange` qui étant donnés une liste `a` et deux entiers `i` et `j` permute dans `a` les éléments d'indices `i` et `j`. Par exemple : `a = [1,2,3,4,5] ; echange(a,0,2) ; print(a)` affiche `[3,2,1,4,5]`.

- c) Ecrire une procédure `tri` qui étant donnée une liste `a` modifie cette liste de sorte à trier ses éléments (sans modifier l'ensemble de ses valeurs). On utilisera un tri par sélection (des minima) : Le principe de l'algorithme consiste à déterminer la position du plus petit élément et à le mettre en première position (par un échange), puis d'itérer le procédé

sur le sous-tableau restant. Il faut $(n - 1)$ comparaisons pour déterminer la position du plus petit élément, donc il faut $(n - 1) + (n - 2) + \dots = \frac{1}{2}n(n - 1)$ comparaisons pour trier une liste selon ce procédé.

6) Tests de permutation

Il s'agit de déterminer si une liste de longueur n correspond à une permutation de $\{0, 1, \dots, n - 1\}$, c'est-à-dire si tout entier compris entre 0 et $(n - 1)$ apparaît une et une seule fois dans la liste. Il s'agit d'écrire une fonction à valeurs booléennes qui étant donnée une liste a de longueur n renvoie *True* ssi a est une permutation de $\{0, 1, \dots, n - 1\}$. On propose deux méthodes, la seconde étant plus efficace que la première (le nombre d'opérations est moindre).

- a) Ecrire une première fonction `test1` qui pour tout entier $i \in \{0, 1, \dots, n - 1\}$, vérifie qu'il existe un unique élément de la liste a valant j , et interrompt la procédure dès que ce test est faux.

La complexité dans le pire des cas est en $O(n^2)$.

- b) Ecrire une première fonction `test2` qui pour tout entier $i \in \{0, 1, \dots, n - 1\}$ construit en temps linéaire $O(n)$ le tableau b des occurrences, c'est-à-dire que $b[j]$ vaut le nombre de termes de a valant j . On vérifie ensuite (en temps linéaire $O(n)$) que le tableau des occurrences ne contient que des 1. La complexité est $O(n)$.